DERS KİTABI

Java ile Nesne Yönelimli Programlama

Sercan Külcü

Giresun Üniversitesi · Bilgisayar Mühendisliği Bölümü

Java ile Nesne Yönelimli Programlama

Sercan Külcü
Giresun Üniversitesi, Bilgisayar Mühendisliği Bölümü
Giresun, Türkiye

Bu kitap, şu adreste yayımlanan ders notlarının basılı hâlidir:
`sercankulcu.github.io/teaching/object-oriented-programming-with-java/`

Sunumlar ve çalışma soruları o sayfadadır. Bölümlerin kodu `github.com/sercankulcu/object-oriented-programming-java` deposundadır.

© Sercan Külcü. Tüm hakları saklıdır.

Notların 14 Kasım 2026 tarihli hâlinden üretilmiştir.

Önsöz

Bu kitap programlamanın temellerinden başlayıp nesne yönelimli programlamaya ve Java ile gerçek uygulamalara uzanır: sınıflar ve nesneler, sarmalama, kalıtım, çok biçimlilik ve soyutlama; koleksiyonlar, istisnalar, test, dosya işlemleri, grafik arayüzler, iş parçacıkları ve veri tabanı programlama. Kitap, Giresun Üniversitesi Bilgisayar Mühendisliği Bölümü'nde verilen Java ile Nesne Yönelimli Programlama dersinin notlarından oluşur ve bölümleri dersteki sırayla izler.

Kitap dört kısımdır. Kısımlar sırasıyla şunlardır: **Programlama Temelleri** (Bölüm 1–3), **Nesne Yönelimli Programlama** (Bölüm 4–5), **Sağlam Program** (Bölüm 6–7) ve **Java ile Uygulamalar** (Bölüm 8–11).

Bazı bölümler birden çok ders notundan oluşur. Bu bölümler kısa bir açılış sayfasıyla başlar; notlar ardından 4.1, 4.2 gibi numaralı alt bölümler olarak gelir.

Her bölüm aynı düzeni izler. Başta öğrenme çıktıları yer alır. Biçimsel tanımlar, teoremler ve çözülmüş örnekler adlandırılmış kutulardadır. *Sık yapılan hata* kutuları, öğrencilerin en sık düştüğü yanılgıları gösterir. Kod izlemeleri, bir programın ya da algoritmanın adım adım nasıl çalıştığını gösterir. Bölüm; bir özet, bir kavram listesi, cevaplarıyla birlikte *Kendini Dene* soruları, ileri okuma önerileri ve kaynakçayla kapanır. Kitabın sonundaki terim sözlüğü, her İngilizce terim için notlarda kullanılan tek Türkçe karşılığı verir.

Bölümlerin kodu derse eşlik eden `github.com/sercankulcu/object-oriented-programming-java` deposundadır; klasörün adı bölümün başında yazılıdır. Sunumlar ve çalışma soruları ders sayfasındadır.

Sercan Külcü
Giresun

İçindekiler

Dersin Haritası 1

KISIM I · PROGRAMLAMA TEMELLERİ

1 Bilgisayarlar, Hesaplama ve Algoritmalar 4

1.1 Bilgisayarlar 5
1.2 Hesaplama ve Hesaplamalı Düşünme 50
1.3 Algoritmalar 100
1.4 Akış Diyagramları 153

2 Programlama ve Java Söz Dizimi 203

2.1 Programlama 204
2.2 Söz Dizimi Kuralları 252

3 Kontrol Yapıları, Fonksiyonlar, Diziler ve Dizgiler 314

3.1 Kontrol Yapıları 315
3.2 Fonksiyonlar 374
3.3 Diziler 438
3.4 Dizgiler 509
3.5 Özyineleme 559

KISIM II · NESNE YÖNELİMLİ PROGRAMLAMA

4 Nesneler ve Sınıflar, Kalıtım ve Çok Biçimlilik 609

4.1 Nesneler ve Sınıflar 610
4.2 Sarmalama 696
4.3 Kalıtım 766
4.4 Çok Biçimlilik 849
4.5 Soyut Sınıflar ve Arayüzler 912

5 Koleksiyonlar 1006

5.1 Koleksiyon Çerçevesi 1007
5.2 Listeler 1017
5.3 Kuyruklar 1032
5.4 Kümeler 1042
5.5 Eşlemeler 1053
5.6 Yineleyici 1062
5.7 Algoritmalar 1067
5.8 Akış API'si 1077
5.9 Doğru koleksiyonu seçmek 1085
5.10 Örnek problemler 1087
5.11 Sık yapılan hatalar 1092
5.12 Özet 1095

5.13 Kavram listesi 1096
5.14 Kendini Dene 1097
5.15 İleri okuma 1098
5.16 Kaynaklar 1099

KISIM III · SAĞLAM PROGRAM

6 İstisnalar 1101
6.1 İstisna kavramı 1102
6.2 try ve catch 1107
6.3 finally ve kaynak yönetimi 1115
6.4 İstisna hiyerarşisi 1124
6.5 Kontrol edilen ve kontrol edilmeyen istisnalar 1127
6.6 İstisna fırlatmak 1131
6.7 Çalışma zamanı istisnaları 1139
6.8 Kontrol edilen istisna sınıfları 1150
6.9 Error sınıfları 1155
6.10 Özel istisnalar 1157
6.11 Yığın izi ve iyi uygulamalar 1163
6.12 Çözümlü problemler 1168
6.13 Sık yapılan hatalar 1172
6.14 Özet 1174
6.15 Kavram listesi 1175
6.16 Kendini Dene 1175
6.17 İleri okuma 1177
6.18 Kaynaklar 1178

7 Test ve Hata Ayıklama 1179
7.1 Test ve hata ayıklamanın temelleri 1180
7.2 Test türleri 1182
7.3 Birim testi ve JUnit 1185
7.4 İyi test tasarımı 1201
7.5 Regresyon ve otomasyon 1207
7.6 Hata ayıklama 1211
7.7 Analiz araçları 1221
7.8 Çözümlü problemler 1235
7.9 Sık yapılan hatalar 1240
7.10 Özet 1241
7.11 Kavram listesi 1242
7.12 Kendini Dene 1242
7.13 İleri okuma 1244
7.14 Kaynaklar 1244

KISIM IV · JAVA İLE UYGULAMALAR

8 Giriş Çıkış İşlemleri 1247

8.1 Akışlar 1248
8.2 Dosyalar ve kodlama 1253
8.3 Dosya sistemi: File sınıfı 1263
8.4 Metin okuma ve yazma 1266
8.5 Tamponlama 1280
8.6 Diğer bayt akışları 1286
8.7 Dosya kopyalama 1292
8.8 Rastgele erişim 1301
8.9 Serileştirme 1306
8.10 NIO 1312
8.11 Örnek problemler ve alıştırmalar 1326
8.12 Sık yapılan hatalar 1330
8.13 Özet 1331
8.14 Kavram listesi 1331
8.15 Kendini Dene 1332
8.16 İleri okuma 1333
8.17 Kaynaklar 1334

9 Grafikler ve Kullanıcı Arayüzleri 1335

9.1 Grafik programlama 1336
9.2 Pencere ve panel 1346
9.3 Çizim araçları 1350
9.4 Yerleşim yöneticileri 1355
9.5 Uygulama: giriş yasak işareti 1363
9.6 Fare olayları ve olay iş parçacığı 1368
9.7 Uygulama: çizim programı 1376
9.8 Düğme ve metin bileşenleri 1384
9.9 Uygulama: metin dönüştürücü 1390
9.10 Uygulama: para üstü hesaplayıcı 1393
9.11 İyi uygulamalar 1398
9.12 Çözümlü problemler 1399
9.13 Sık yapılan hatalar 1401
9.14 Özet 1403
9.15 Kavram listesi 1405
9.16 Kendini Dene 1405
9.17 İleri okuma 1407
9.18 Kaynaklar 1407

10 İş Parçacıkları 1409

10.1 Temel kavramlar 1410
10.2 Thread sınıfı 1415
10.3 Yaşam döngüsü 1418
10.4 Oluşturma ve başlatma 1422
10.5 Yönetim ve durdurma 1431
10.6 Senkronizasyon 1437
10.7 Görünürlük 1446

10.8 Açık kilitler ve kilitlenme 1450
10.9 Koşul senkronizasyonu 1463
10.10 Senkronizasyon nesneleri 1472
10.11 Diğer araçlar 1478
10.12 Çözümlü problemler 1490
10.13 Sık yapılan hatalar 1494
10.14 Özet 1495
10.15 Kavram listesi 1496
10.16 Kendini Dene 1497
10.17 İleri okuma 1499
10.18 Kaynaklar 1499

11 Veri Tabanı Programlama 1500

11.1 Veri tabanı kavramı 1501
11.2 SQLite 1506
11.3 SQL temelleri 1508
11.4 İlişkiler ve gruplama 1515
11.5 JDBC 1528
11.6 Temel işlemler 1533
11.7 PreparedStatement 1543
11.8 İleri JDBC 1550
11.9 İşlemler 1556
11.10 Örnek problemler ve alıştırmalar 1563
11.11 Sık yapılan hatalar 1565
11.12 Özet 1567
11.13 Kavram listesi 1568
11.14 Kendini Dene 1568
11.15 İleri okuma 1570
11.16 Kaynaklar 1570

EKLER

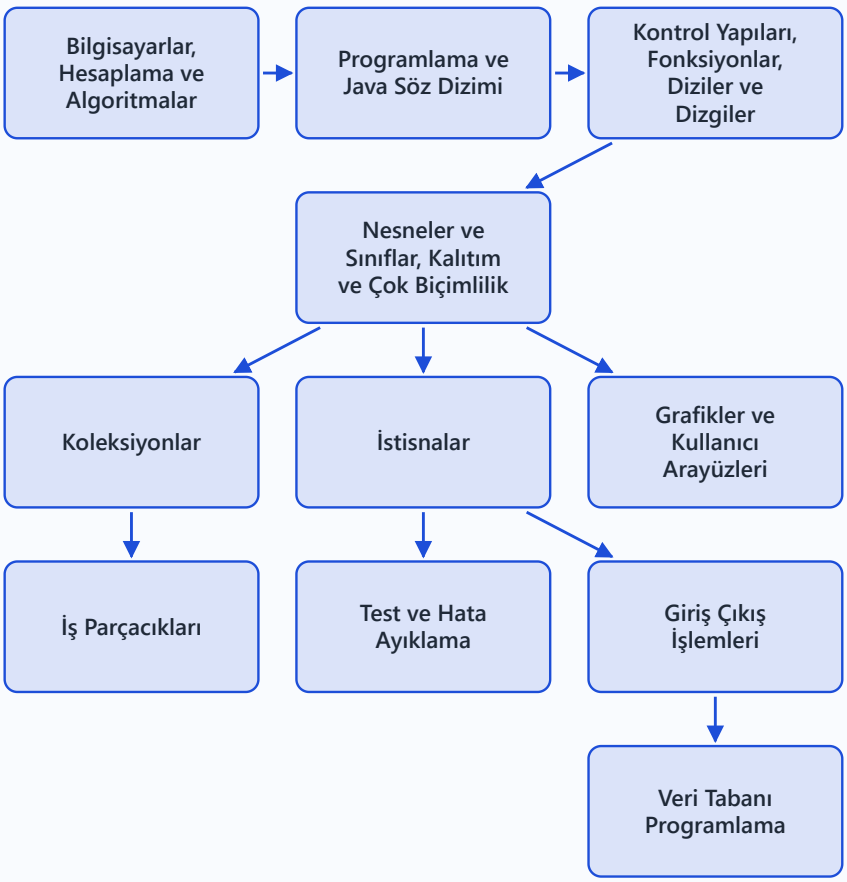
Ek A Terim Sözlüğü 1571

Ek B Java ile Nesne Yönelimli Programlama İçin Ön Bilgiler 1575

Ek C Java ve Nesne Yönelimli Programlamada Önemli İsimler 1595

Dersin Haritası

Bu sayfa dersin bütün bölümlerini tek bir haritada birleştirir. Her bölüm bir düğümdür; bir ok, sonraki konunun önceki konunun hangi fikrine dayandığını gösterir. Derse başlamadan önce okunursa konuların nereye gittiğini, ders sırasında okunursa bulunulan yeri gösterir.



Şekil 1: Dersin haritası. Oklar, bir konunun hangi konunun üzerine kurulduğunu gösterir. Programlama temelleri nesnelere, nesneler sağlam programlara ve uygulamalara götürür.

Şekil 1 konular arasındaki bağları gösterir. Her düğüm bir bölümdür; bir ok, sonraki konunun önceki-nin hangi fikrine dayandığını belirtir. Haritada yalnızca en yakın bağlar çizilmiştir; bir bölümün dayandığı bütün bölümler Tablo 1'dedir.

Konu (bölüm)	Dayandığı konu	Bağ
Bilgisayarlar, Hesaplama ve Algoritmalar (1)	—	Bilginin bellekte temsili, saklanan program ve algoritma; programlamanın zemini.
Programlama ve Java Söz Dizimi (2)	Bilgisayarlar, Hesaplama ve Algoritmalar	Programlama dilinin söz dizimi ve anlamı; Java'da derleme, tipler ve değişkenler.
Kontrol Yapıları, Fonksiyonlar, Diziler ve Dizgiler (3)	Programlama ve Java Söz Dizimi	Akışı yöneten yapılar, metotlar, diziler ve dizgiler; nesnelerden önceki son araçlar.
Nesneler ve Sınıflar, Kalıtım ve Çok Biçimlilik (4)	Kontrol Yapıları, Fonksiyonlar, Diziler ve Dizgiler	Veri ve davranış sınıfta birleşir: sarmalama, kalıtım, çok biçimlilik, arayüzler.
Koleksiyonlar (5)	Nesneler ve Sınıflar, Kalıtım ve Çok Biçimlilik	Hazır sınıflar ve arayüzler: List, Set, Map; jenerikler ve karşılaştırıcılar.
İstisnalar (6)	Nesneler ve Sınıflar, Kalıtım ve Çok Biçimlilik	Hata da bir nesnedir: istisna hiyerarşisi kalıtımla kurulur; try, catch ve finally.
Test ve Hata Ayıklama (7)	Nesneler ve Sınıflar, Kalıtım ve Çok Biçimlilik, İstisnalar	Sınıfların davranışını JUnit ile sınamak; beklenen istisnanın testi; hata ayıklama.
Giriş Çıkış İşlemleri (8)	Koleksiyonlar, İstisnalar	Akışlar ve dosyalar; G/Ç hataları kontrol edilen istisnalarla bildirilir.
Grafikler ve Kullanıcı Arayüzleri (9)	Nesneler ve Sınıflar, Kalıtım ve Çok Biçimlilik	Swing bileşenleri kalıtımla genişletilir; olaylar dinleyici arayüzleriyle yakalanır.
İş Parçacıkları (10)	Nesneler ve Sınıflar, Kalıtım ve Çok Biçimlilik, Koleksiyonlar	Runnable arayüzü ve Thread sınıfı; paylaşılan koleksiyonlarda yarış durumu.
Veri Tabanı Programlama (11)	İstisnalar, Giriş Çıkış İşlemleri	Kalıcı verinin ikinci biçimi: dosyadan veri tabanına; JDBC ve SQLException.

Tablo 1: Haritadaki bağlar.

Haritanın ana omurgası şöyle okunabilir: **temeller → nesne → sağlam program → uygulama**. İlk üç bölüm dili ve yapısal programlamayı kurar. Nesneler ve sınıflar bölümü dersin merkezidir: sonraki her bölüm onun bir aracını kullanır. Koleksiyonlar arayüz ve jenerikleri, istisnalar kalıtım hiyerarşisini, grafik arayüzler kalıtım ile dinleyici arayüzlerini kullanır. Test ve istisnalar programın sağlamlığını, giriş çıkış ve veri tabanı ise verinin kalıcılığını ele alır.

KISIM I

Programlama Temelleri

Bölüm 1–3

KISIM I · BÖLÜM 1

Bilgisayarlar, Hesaplama ve Algoritmalar

Programlamanın zemini, bilgisayarın bilgiyi nasıl temsil ettiği ve işlediğidir. Bölüm sayıların, harflerin ve dizgilerin bellekte gösterimiyle, saklanan program fikri ve von Neumann mimarisiyle başlar. Ardından bilgisayar biliminin tarihine, Turing makinelerine ve hesaplamanın sınırlarına değinilir; problem çözme süreci algoritma kavramı ve basit arama ve sıralama örnekleri üzerinden ele alınır. Son kısımda akış diyagramlarının sembolleri ile sıralı, koşullu ve tekrarlı akışlar örnek problemler üzerinde işlenir.

Bu bölümün kodu: `github.com/sercankulcu/object-oriented-programming-java/Bolum01/src`

BU BÖLÜMÜN NOTLARI

1.1 Bilgisayarlar

1.2 Hesaplama ve Hesaplamalı Düşünme

1.3 Algoritmalar

1.4 Akış Diyagramları

KISIM I · BÖLÜM 1.1

Bilgisayarlar

Java ile yazılan her satır, sonunda bir işlemcinin anladığı sayılara dönüşür. Bir `int` değişken neden 2.147.483.647'yi aşamaz? `"😀".length()` neden 2 verir, `0.1 + 0.2` neden tam 0,3 çıkmaz? Bu soruların cevabı o dönüşümdedir. Bu not, bilgisayarı katman katman kurar. Önce elektrik sinyalleri bitlere, bitler sayılara dönüşür. Sayılarla harfler ve metinler, sayı dizileriyle resim, ses ve film temsil edilir. En sonda komutların da sayı olarak bellekte durduğu saklanan program fikri ve onu donanıma taşıyan von Neumann mimarisi gelir. Her katmanda konunun Java'daki karşılığı kısa programlarla gösterilir ve programların çıktıları adım adım izlenir.

ÖĞRENME ÇIKTILARI

Donanım
İşlemci, bellek, disk ve veri yolunun görevini açıklamak; bir programın işlemcide Getir–Çöz–Yürüt döngüsüyle nasıl yürüdüğünü adım adım izlemek.

Sayılar
Sayıları ikili, sekizli, onaltılı ve ondalık tabanlar arasında çevirmek; ikili toplama yapmak ve negatif sayıları ikiye tümleyenle hesaplamak.

Metin
ASCII, Unicode ve UTF-8 arasındaki farkı açıklamak; bir dizginin bellekte sayılarla nasıl temsil edildiğini göstermek.

Java ile bağlantı
`byte` ve `int` sınırlarını ve taşmayı ikiye tümleyenle açıklamak; Java'da tabanlarla çalışmak; `char` ile `String` uzunluğunun UTF-16 ile ilişkisini kurmak; assembly komutlarının makine koduna karşılığını göstermek.

1.1.1 Bilgisayarın temelleri

Bilgisayar, verilen komutları çok hızlı ve hatasız biçimde art arda yürüten bir makinedir. Komutlar tek başına çok basittir: iki sayıyı topla, bir sayıyı bellekten oku, sonucu belleğe yaz, şu koşul doğruysa başka bir komuta geç. Bilgisayarı güçlü yapan, bu basit adımları saniyede milyarlarca kez yapabilme-

sidir. Nesne yönelimli bir programın sınıfları, nesneleri ve metotları da sonunda bu basit adımlara dönüşür. Bu yüzden Java öğrenmeye makinenin kendisinden başlamak, ileride karşılaşılacak pek çok davranışı açıklar.

1.1.1.1 Temel parçalar

Bir bilgisayarın içinde çok sayıda bileşen vardır, ama programcı açısından üçü öne çıkar. Bu üç parça birbirine veri taşıyan yollarla bağlıdır.

- **İşlemci** (*CPU, Central Processing Unit*; merkezî işlem birimi): bilgisayarın "beyni" sayılır. Bütün aritmetik ve mantıksal hesapları yapar. Toplama, çıkarma, iki sayıyı karşılaştırma ve bitler üzerinde mantık işlemleri onun işidir.
- **Bellek** (*RAM, Random Access Memory*; rastgele erişimli bellek): çalışan programı ve onun verilerini geçici olarak saklar. "Rastgele erişim", herhangi bir hücreye diğerleri üzerinde sırayla gezinmeden aynı sürede ulaşılabilmesi demektir. Belleğin kapasitesi ve erişim hızı, bilgisayarın hızını doğrudan etkiler.
- **Sabit disk** (HDD ya da SSD): verileri kalıcı olarak saklar. Elektrik kesildiğinde RAM'deki her şey silinir, diskteki dosyalar ise yerinde kalır. HDD dönen manyetik plakalar kullanır, SSD ise hareketli parçası olmayan yarı iletken bellek yongaları kullanır; ikisi de RAM'den çok daha büyük ve çok daha yavaştır.



Şekil 1.1.1: Disk, bellek ve işlemci arasındaki veri akışı. Program ve verisi diskten belleğe yüklenir, işlemci belleğe erişerek hesap yapar. Disk kalıcı ama yavaş, bellek geçici ama hızlıdır.

Bu üçlünün ilişkisi bir mutfak benzetmesiyle kolayca hatırlanır. Aşçı işlemcidir. Tezgâh RAM'dir: küçüktür ama aşçının hemen elinin altındadır. Kiler ise disktir: çok büyüktür ama uzaktadır. Aşçı kilerdeki malzemeyle doğrudan yemek yapmaz. Önce gereken malzemeyi kilerden tezgâha taşır, orada çalışır. Bilgisayarda da bir program çalıştırılınca kodu ve verisi önce diskten belleğe yüklenir. İşlemci yalnız bellekteki komut ve verilerle çalışır. Tezgâh dolarsa aşçı sık sık kilere gider ve yemek yavaşlar; RAM yetersiz kaldığında bilgisayarın yavaşlaması da aynı nedenledir.

Elektrik-elektronik açısından bakıldığında RAM ile disk arasındaki fark, bilginin nasıl tutulduğundan gelir. Yaygın kullanılan dinamik RAM (DRAM), her biti küçük bir kondansatördeki yük olarak saklar. Bu yük kendiliğinden boşaldığı için saniyede pek çok kez tazelenmesi gerekir; güç kesilince bilgi kay-

bolur. SSD'deki flaş bellek ise yükü yalıtılmış bir kapı içinde hapseder; bu yük güç olmadan da yıllarca kalır. Bu yüzden RAM'e **uçucu**, diske **kalıcı** bellek denir.

1.1.1.2 Bilgisayarın işleyişi ve programlama

Bilgisayarın işleyişi üç temel adımla özetlenir. İşlemciye talimatlar, yani komutlar gönderilir. İşlemci bu talimatları sırayla işler. Sonuçlar kullanıcıya ya da başka yazılımlara sunulur. Ekrana bir yazı basmak, bir dosyayı kaydetmek ya da bir ağ paketini göndermek hep bu döngünün ürünüdür.

> **Tanım 1.1.1**
>
> **Programlama**, bilgisayarın istenen görevi yerine getirmesi için ona anlayacağı biçimde talimatlar yazma sürecidir. Yazılan talimatların bütününe **program** denir.

En kısa Java programı, ekrana tek bir satır yazan programdır. İçinde yalnız bir talimat vardır: “ekrana yaz”.

```java
public class MerhabaDunya {
    public static void main(String[] args) {
        System.out.println("Merhaba, Dünya!");
    }
}
```

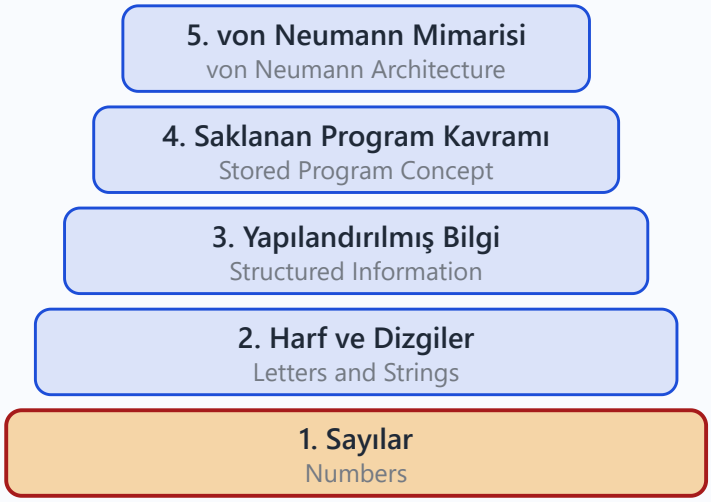
```
Merhaba, Dünya!
```

Programın kalıbı ilk anda kalabalık görünebilir. `public class MerhabaDunya` satırı bir sınıf tanımlar; Java'da her kod bir sınıfın içinde durur ve dosyanın adı da `MerhabaDunya.java` olmalıdır. `main` metodu programın başladığı yerdir; Java Sanal Makinesi (JVM, *Java Virtual Machine*) programı çalıştırırken ilk olarak bu metodu çağırır. Asıl iş, `System.out.println(...)` satırındadır. Bu satır, tırnak içindeki dizgiyi standart çıktıya yazar ve satırı bitirir.

İşlemci `println` sözcüğünü tanımaz. Java gibi bir dilde yazılan talimatlar, işlemcinin anladığı **makine koduna** çevrilerek çalıştırılır. Java'da bu çeviri iki aşamalıdır. Önce `javac` derleyicisi kaynak kodu **bytecode** denen ara bir koda çevirir. Ardından JVM bu kodu yorumlar ya da sık çalışan kısımlarını o işlemcinin makine koduna derler. Sonuçta ekrana harf basmak da işlemci için sayıları bir yerden bir yere taşımaktan ibarettir. Notun geri kalanı, bu sayıların neyi nasıl temsil ettiğini anlatır.

1.1.1.3 Katman katman bir bilgisayar

Bir bilgisayarın nasıl çalıştığını tek seferde kavramak zordur. Bunun yerine bilgisayarı katman katman kurmak daha kolaydır. Her katman bir alttakinin üzerine oturur ve onun sunduğu yapı taşlarını kullanır.



Şekil 1.1.2: Notun yol haritası. En altta sayılar vardır; sayılarla harfler, harflerle yapılandırılmış bilgi kurulur. Komutlar da sayı olarak saklanır ve hepsini von Neumann mimarisi bir araya getirir.

1. Önce elektrik sinyalleri sayılara dönüştürülür.
2. Sayılarla harfler, harflerle metinler temsil edilir.
3. Sayı dizileriyle resim, ses ve film gibi yapılandırılmış bilgi saklanır.
4. Komutlar da sayı olarak bellekte saklanır.
5. Hepsini von Neumann mimarisi bir araya getirir.

Bu sıra notun kesimlerini de belirler. Her katmanda aynı soru sorulur: bir üst katmandaki kavram, bir alttaki sayılarla nasıl ifade edilir? Bu soruya verilen cevaplar, Java'nın ilkel tiplerinin (`byte`, `int`, `char`, `double`) neden öyle tasarlandığını da açıklar.

1.1.2 İkili sayılar

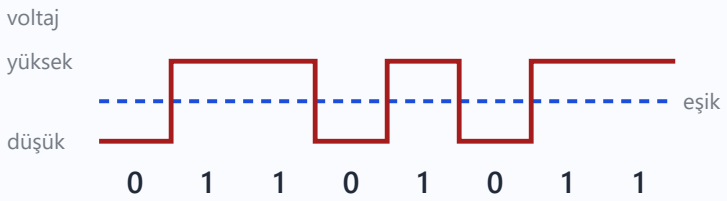
1.1.2.1 Elektrik sinyalinden bite

Bilgisayarın temel işi, elektrik sinyallerini belirli kurallara göre işlemektir. Bu kurallar sayesinde toplama, karşılaştırma ve kopyalama gibi pek çok işlev gerçekleşir. Sinyaller tek başına bir anlam taşımaz. Onları insanın anlayacağı sayı ve sembollerle ilişkilendirdiğimizde bilgisayar bize anlamlı bilgi sunabilir.

Bilgisayarlar sinyalleri **ikili** (*binary*, taban 2) temsille işler. Bir teldeki voltaj belirli bir eşiğin üstündeyse 1, altındaysa 0 okunur. Bütün işlemler bu ikili temelin üzerine kuruludur.

Tanım 1.1.2

Bit (*binary digit*, ikili basamak), yalnız 0 ya da 1 değerini alabilen en küçük bilgi birimidir. Sekiz bitlik bir gruba **bayt** (*byte*) denir.



Şekil 1.1.3: Bir teldeki voltaj zamanla değişir. Eşiğin üstündeki aralıklar 1, altındakiler 0 okunur; bu dalga 01101011 bit dizisini taşır.

0 ve 1 değerleri bağlama göre farklı adlar alır. Tablo 1.1.1 en sık kullanılan karşılıkları gösterir. Mantıkta 0 yanlış, 1 doğrudur; anahtarda 0 kapalı, 1 açıktır; devrede ise 0 düşük, 1 yüksek voltajdır. Hepsi aynı ikiliğin farklı yorumlarıdır.

0	1
Yanlış (`false`)	Doğru (`true`)
Kapalı	Açık
Düşük voltaj	Yüksek voltaj

Tablo 1.1.1: Bitin iki değerinin farklı yorumları.

Neden on değil de iki durum kullanılır? Gerçek bir telde voltaj hiçbir zaman tam istenen değerde durmaz. Isı, komşu tellerden gelen girişim ve güç kaynağındaki dalgalanma, sinyale **gürültü** ekler. Bir teldeki voltajın “var mı, yok mu” olduğunu gürültüye rağmen ayırt etmek kolaydır, çünkü iki durum arasında geniş bir güvenlik payı bırakılabilir. Aynı voltaj aralığını on eşit düzeye bölmek ise her düzeyi daraltır. Küçük bir gürültü sinyali komşu düzeye kaydırır ve 7 diye gönderilen rakam 6 diye okunur. Elektronikteki mantık kapıları bu yüzden iki durumla çalışır ve sayısal devreler bu güvenlik payı sayesinde son derece güvenilirdir.

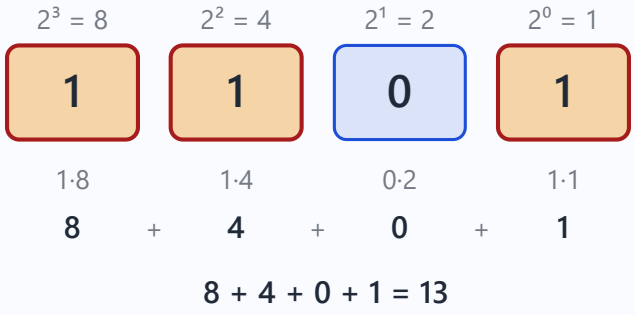
Bellek ve dosya boyutları bayt cinsinden ölçülür. Ön ekler iki farklı biçimde kullanılır. Uluslararası Birimler Sistemi'nde (SI) 1 kilobayt (KB) 1000 bayttır. İkinin kuvvetlerine dayanan ikili ön eklerde ise 1 kibibayt (KiB) 1024 bayttır. Fark, birimler büyüdükçe artar: 1 GB = 10^9 bayt, 1 GiB = 2^{30} = 1.073.741.824 bayttır. Bir diskin üzerinde yazan 1 TB'lık kapasitenin işletim sisteminde yaklaşık 931 “GB” görünmesinin nedeni, işletim sisteminin aslında GiB ile hesaplamasıdır.

1.1.2.2 Basamak değeri

Gündelik hayatta kullanılan ondalık sistem bir **basamak değerli** sistemdir. 603 sayısında 6, yüzler basamağında olduğu için 600 değer taşır. Aynı rakam başka bir basamakta başka bir değer taşır. İkili sistem de aynı ilkeyle çalışır, yalnız taban 10 yerine 2'dir.

Bir tam sayı, 2'nin kuvvetlerinin toplamı olarak yazılır ve her kuvvet ya hiç kullanılmaz (0) ya da bir kez kullanılır (1). Dört ardışık kuvvetle, yani 2^0, 2^1, 2^2, 2^3 ile 0–15 arasındaki bütün tam sayılar temsil edilir. 13 sayısı şöyle yazılır:

$$13_{10} = 1\cdot2^3 + 1\cdot2^2 + 0\cdot2^1 + 1\cdot2^0 = 8 + 4 + 0 + 1 = 1101_2$$



Şekil 1.1.4: 1101_2 sayısında basamak ağırlıkları soldan sağa 8, 4, 2 ve 1'dir. Değeri 1 olan basamakların ağırlıkları toplanır: 8 + 4 + 0 + 1 = 13.

Sayının sağ altındaki küçük rakam, yani **alt simge**, tabanı gösterir. 1101_2 "bir bir sıfır bir, ikili" diye okunur, "bin yüz bir" diye okunmaz. Alt simge yazılmayan sayılar ondalık kabul edilir. Aynı kural öteki tabanlarda da geçerlidir (Tablo 1.1.2).

Taban	Örnek
Ondalık (10)	$603_{10} = 6\cdot10^2 + 0\cdot10^1 + 3\cdot10^0$
Sekizli (8)	$207_8 = 2\cdot8^2 + 0\cdot8^1 + 7\cdot8^0 = 135_{10}$
Onaltılı (16)	$FF_{16} = 15\cdot16^1 + 15\cdot16^0 = 255_{10}$

Tablo 1.1.2: Basamak değeri kuralı her tabanda aynıdır.

İkili bir sayıyı ondalığa çevirmek bu yüzden kolaydır: 1 olan basamakların ağırlıkları toplanır. 10110110_2 sayısında 1'ler 7, 5, 4, 2 ve 1 numaralı basamaklardadır (en sağdaki basamak 0 numaralıdır). Değer *128 + 32 + 16 + 4 + 2 = 182*'dir. Ağırlıkları ezbere bilmek bu hesabı hızlandırır: 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024. Programcılar bu diziyi çarpım tablosu gibi bilir.

1.1.2.3 Herhangi bir taban ve bit sınırları

Basamak değeri fikri her tabana genellenir. Taban $b \geq 2$ ise her basamak $0, 1, \ldots, b-1$ değerlerinden birini alır. İkilide basamaklar 0–1, sekizlide 0–7, onaltılıda 0–9 ve A–F'dir. Onaltılıda A = 10, B = 11, C = 12, D = 13, E = 14 ve F = 15'tir. Tek basamakla 15'e kadar sayabilmek için altı yeni simge gerekir.

> **Teorem 1.1.1 (Basamak sınırı)**
>
> Taban $b \geq 2$ ve d basamakla, 0 ile $b^d - 1$ arasındaki her tam sayı yazılabilir ve başka sayı yazılamaz.

İspat. Her basamak b farklı değer alabildiği için d basamakla toplam b^d farklı rakam dizisi kurulur. En küçük dizi 00…0 = 0'dır. En büyük dizinin bütün basamakları $b-1$'dir; buna 1 eklenince elde ilerleyerek en solda yeni bir basamak açılır ve b^d elde edilir. Demek ki en büyük değer $b^d - 1$'dir. Arada b^d tane sayı vardır ve her biri farklı bir diziye karşılık gelir. ∎

Ondalıkta üç basamakla 0–999 yazılır: $10^3 - 1 = 999$. İkilide d bitle en büyük işaretsiz değer $2^d - 1$'dir (Tablo 1.1.3). 64 bitlik bir bilgisayar, işaretsiz olarak $2^{64} - 1$ = 18.446.744.073.709.551.615'e kadar sayabilir.

Bit	En büyük ($2^d - 1$)
4	15
8	255
16	65.535
32	4.294.967.295

Tablo 1.1.3: Bit sayısına göre en büyük işaretsiz değer.

Java'nın tam sayı tipleri ise **işaretlidir**: bitlerin temsil ettiği değerlerin yarısı negatif sayılara ayrılır. Bu yüzden 8 bitlik `byte` 255'e değil, 127'ye kadar çıkar. Aşağıdaki program dört tam sayı tipinin en büyük değerini yazdırır.

```
System.out.println(Byte.MAX_VALUE);    // 8 bit
System.out.println(Short.MAX_VALUE);   // 16 bit
System.out.println(Integer.MAX_VALUE); // 32 bit
System.out.println(Long.MAX_VALUE);    // 64 bit
// 64 bitin hepsi 1, işaretsiz okunursa:
System.out.println(Long.toUnsignedString(-1L));
```

```
127
32767
2147483647
9223372036854775807
18446744073709551615
```

Her sabit, tablodaki işaretsiz sınırın yaklaşık yarısıdır. `Byte.MAX_VALUE` = 127 = $2^7 - 1$, `Short.MAX_VALUE` = 32.767 = $2^{15} - 1$, `Integer.MAX_VALUE` = 2.147.483.647 = $2^{31} - 1$ ve `Long.MAX_VALUE` = $2^{63} - 1$'dir. Bir bitin, negatif sayıları ayırt etmek için kullanıldığı görülür. Son satırdaki `-1L` sayısının 64 bitinin hepsi 1'dir. `Long.toUnsignedString` bu bitleri işaretsiz okuyunca $2^{64} - 1$ değerini verir. Aynı bitler, nasıl yorumlandığına göre −1 de olabilir, 18 kentilyondan büyük bir sayı da. Bu ikilik negatif sayılar konusunda yeniden karşımıza çıkar.

Java'da `int` her platformda 32 bit, `long` her platformda 64 bittir. C dilinde `int`'in boyu derleyiciye ve işlemciye göre değişebilir. Java'nın sabit boyutları, aynı programın her makinede aynı sonucu vermesini sağlar.

1.1.2.4 Ondalıktan ikiliye: 2'ye bölme yöntemi

Ondalık bir sayıyı ikiliye çevirmenin en sistemli yolu, sayıyı art arda 2'ye bölmektir. Her bölmede kalan 0 ya da 1'dir ve bu kalan, sayının henüz yazılmamış en sağdaki bitidir. Bölüm 0 olunca durulur ve kalanlar aşağıdan yukarı okunur.

Yöntemin neden çalıştığını görmek için bir sayının ikili yazılışına bakmak yeter. $13 = 1101_2$ ise $13 = 2 \cdot 6 + 1$'dir. Kalan 1, en sağdaki bittir; bölüm 6 ise geri kalan bitlerin oluşturduğu sayıdır, yani 110_2. Aynı işlem 6'ya uygulanınca bir sonraki bit çıkar. Her bölme bir biti "koparır" ve sayıyı bir bit sağa kaydırır. Aşağıdaki program bu yöntemi Java ile uygular.

```
public static void main(String[] args) {
  int sayi = 13;
  String ikili = "";
  while (sayi > 0) {
    int kalan = sayi % 2;
    System.out.println(sayi + " % 2 = " + kalan);
    ikili = kalan + ikili;
    sayi = sayi / 2;
  }
  System.out.println("Sonuç: " + ikili);
}
```

Kodda dört nokta önemlidir. `sayi % 2`, kalan alma (mod) işlemidir ve son biti verir. `sayi / 2`, iki `int` arasında yapıldığı için **tam sayı bölmesidir**: kesir atılır, 13 / 2 = 6 olur. `ikili = kalan + ikili` satırında `kalan` bir `int`, `ikili` ise bir `String`'dir. Bir dizgiyle toplanan sayı dizgiye çevrilir

ve iki dizgi birleştirilir. Yeni bit dizginin *önüne* eklendiği için bitler doğru sırada birikir ve kalanları ayrıca ters çevirmek gerekmez. Döngü, `sayi` 0 olunca biter. Tablo 1.1.4 13 için her turu gösterir.

Tur	Koşul	`kalan`	Yazılan satır	`ikili`	Yeni `sayi`
1	13 > 0 doğru	1	`13 % 2 = 1`	`"1"`	6
2	6 > 0 doğru	0	`6 % 2 = 0`	`"01"`	3
3	3 > 0 doğru	1	`3 % 2 = 1`	`"101"`	1
4	1 > 0 doğru	1	`1 % 2 = 1`	`"1101"`	0
—	0 > 0 yanlış	Döngü biter; `Sonuç: 1101` yazılır.			

Tablo 1.1.4: `sayi = 13` için döngünün izlemesi.

```
13 % 2 = 1
6 % 2 = 0
3 % 2 = 1
1 % 2 = 1
Sonuç: 1101
```



Şekil 1.1.5: 13'ün 2'ye bölünmesi. Her satırda sayı, yarısı ve kalanı görülür. Kalanlar aşağıdan yukarı okununca 1101 elde edilir.

Döngü dört tur döndü, çünkü 13'ün ikili gösterimi dört bittir. Genel olarak $n \geq 1$ için tur sayısı $\lfloor \log_2 n \rfloor + 1$'dir: sayı her turda yarıya iner ve 1'den küçük olana kadar yarılanma sayısı bu kadardır. 1000 sayısı 10 turda, bir milyar 30 turda çevrilir.

Kodun bir sınırı da vardır: `sayi = 0` için döngü hiç çalışmaz ve `ikili` boş kalır; ekrana yalnız `Sonuç:` yazılır. Negatif sayılarda da koşul ilk turda yanlış olur. Gerçek bir programda 0 için ayrı bir durum yazmak gerekir. Java'nın hazır `Integer.toBinaryString` metodu bu ayrıntıları kendisi çözer.

Örnek 1.1.1

45 sayısı 8 bitlik ikili olarak nasıl yazılır? Bölme zinciri şöyledir: 45 / 2 = 22 kalan 1; 22 / 2 = 11 kalan 0; 11 / 2 = 5 kalan 1; 5 / 2 = 2 kalan 1; 2 / 2 = 1 kalan 0; 1 / 2 = 0 kalan 1. Kalanlar aşağıdan yukarı okununca 101101 çıkar. Sekiz bite tamamlamak için soluna iki sıfır eklenir: **00101101**. Sağlama: *32 + 8 + 4 + 1 = 45*.

Aynı sonuca kuvvetleri çıkararak da ulaşılır. 45'e sığan en büyük kuvvet 32'dir; kalan 13. 13'e sığan en büyük kuvvet 8'dir; kalan 5. Sonra 4, kalan 1; sonra 1, kalan 0. Kullanılan kuvvetler 32, 8, 4 ve 1'dir; bu basamaklara 1, ötekilere 0 yazılır.

1.1.2.5 Sekizli ve onaltılı tabanlar

İkili sayılar uzundur. 32 bitlik bir `int` 'i otuz iki rakamla yazmak hem yorucudur hem de hataya açıktır. Programcılar bu yüzden ikili sayıları daha kısa yazmak için sekizli ve onaltılı tabanları kullanır. Bu iki tabanın seçilme nedeni, 8 ve 16'nın 2'nin kuvveti olmasıdır: $8 = 2^3$ ve $16 = 2^4$.

Bu ilişki çevirmeyi hesapsız hâle getirir. Bir onaltılı basamak tam olarak dört bite, bir sekizli basamak tam olarak üç bite karşılık gelir. İkiliden onaltılıya çevirmek için bitler sağdan başlayarak dörderli gruplara ayrılır ve her grup tek bir onaltılı basamağa çevrilir. Sekizli için gruplar üçlüdür.

Örnek 1.1.2

10111110_2 sayısı onaltılı ve sekizli tabana çevrilecek.

Onaltılı için dörderli gruplar: `1011 1110` . 1011 = 8 + 2 + 1 = 11 = B, 1110 = 8 + 4 + 2 = 14 = E. Sonuç $\mathbf{BE}_{16}$'dır.

Sekizli için sağdan üçerli gruplar: `10 111 110` . En soldaki eksik grup başına sıfır eklenerek `010` olur. 010 = 2, 111 = 7, 110 = 6; sonuç $\mathbf{276}_{8}$'dir.

Sağlama: *11·16 + 14 = 190* ve *2·64 + 7·8 + 6 = 190*. İkili sayının ondalık değeri de *128 + 32 + 16 + 8 + 4 + 2 = 190*'dır.

Onaltılı taban günümüzde sekizliden çok daha yaygındır. Bir bayt sekiz bit olduğu için tam iki onaltılı basamağa sığar: 00–FF. Bellek adresleri, renk kodları (`#FF0000` kırmızı), dosyaların ham içerikleri ve Unicode kod noktaları hep onaltılı yazılır. Sekizli taban ise Unix dosya izinleri (`chmod 755`) gibi az sayıda yerde yaşar.

1.1.2.6 Java'da taban dönüşümleri

Java, tabanlar arasında çeviri için hazır metotlar ve tam sayı sabitlerini farklı tabanlarda yazmak için ön ekler sunar. Aşağıdaki program ikisini birlikte kullanır.

```
System.out.println(Integer.toBinaryString(13));  // ikili
System.out.println(Integer.toOctalString(135));  // sekizli
System.out.println(Integer.toHexString(255));    // onaltılı
System.out.println(Integer.parseInt("1101", 2)); // 2 → 10
System.out.println(Integer.parseInt("207", 8));  // 8 → 10
System.out.println(0b1101 + " " + 0207 + " " + 0xFF);
```

```
1101
207
ff
13
135
13 135 255
```

İlk üç satır bir `int` değeri dizgiye çevirir. `Integer.toBinaryString(13)` ikili, `toOctalString(135)` sekizli, `toHexString(255)` onaltılı gösterimi verir. `toHexString` küçük harf kullanır; çıktı `FF` değil `ff`'dir. Ters yöndeki çeviriyi `Integer.parseInt` yapar. İkinci argüman, dizginin hangi tabanda yazıldığını söyler. `parseInt("1101", 2)` dizgiyi ikili okur ve 13 döndürür. Dönen değer artık sıradan bir `int`'tir; taban yalnız yazılışın özelliğidir, sayının değil.

Son satır, kaynak kodda sabitleri doğrudan farklı tabanlarda yazmayı gösterir. Bu yazılışlara **sabit** ya da **değişmez** (*literal*) denir ve ön ekleri Tablo 1.1.5'te özetlenmiştir.

Ön ek	Taban	Örnek	Not
`0b` ya da `0B`	İkili	`0b1101` = 13	Java 7 ile geldi.
`0`	Sekizli	`0207` = 135	`010` on değil, 8'dir.
`0x` ya da `0X`	Onaltılı	`0xFF` = 255	Renk ve bayt değerlerinde sık kullanılır.
(yok)	Ondalık	`135`	Başında 0 olmamalıdır.

Tablo 1.1.5: Java'da tam sayı sabitlerinin ön ekleri.

Ön ekler yalnız yazılışı değiştirir. `0b1101`, `015`, `0xD` ve `13` aynı `int` değerini üretir ve bellekte aynı 32 bit olarak durur. Uzun sabitler alt çizgiyle gruplanabilir: `1_000_000` ya da `0b1011_1110`. Derleyici alt çizgileri yok sayar; okunurluk artar.

Sık yapılan hata

Sayıyı hizalamak için başına sıfır koymak. `int x = 010;` derlenir, ama sabit sekizli okunduğu için `x` 10 değil 8 olur. `int y = 08;` ise hiç derlenmez: sekizli tabanda 8 rakamı yoktur ve `javac` "`error: ';' expected`" iletisini verir. İleti ilk bakışta ilgisiz görünür; derleyici `0`'ı tam bir sabit sayıp ardından gelen `8`'i beklenmedik bir sözcük olarak görür.

1.1.2.7 İkili toplama

Toplama, hangi taban kullanılırsa kullanılsın aynı ilkelerle yapılır. Sayılar basamak değerlerine göre alt alta yazılır ve sağdan sola toplanır. Bir basamağa sığmayan değer, bir sonraki basamağa **elde** (*carry*, taşıma) olarak aktarılır. Onlukta $7 + 8 = 15$ olunca 5 yazılır ve 1 elde kalır. İkilide $1 + 1 = 2 = 10_2$ olunca 0 yazılır ve 1 elde kalır. Elde ile birlikte üç tane 1 toplanırsa $3 = 11_2$ olur: 1 yazılır, 1 elde kalır.

İkili toplamanın bütün durumları dört satırlık bir tabloya sığar: 0 + 0 = 0, 0 + 1 = 1, 1 + 0 = 1, 1 + 1 = 10. Elde de hesaba katılınca her sütunda üç bit toplanır ve sonuç 0 ile 3 arasında kalır. Elektronikte bu işi **tam toplayıcı** devresi yapar: üç bit girer, bir toplam biti ve bir elde biti çıkar. Sekiz tam toplayıcı zincirlenince 8 bitlik bir toplayıcı elde edilir.

Örnek 1.1.3

57 ile 28, 8 bitlik ikili sayılar olarak toplanıyor. $57 = 00111001_2$ ve $28 = 00011100_2$'dir. Toplama en sağdaki sütundan başlar (Tablo 1.1.6).

Sütun	57'nin biti	28'in biti	Gelen elde	Toplam	Yazılan	Yeni elde
1	1	0	0	1	1	0
2	0	0	0	0	0	0
3	0	1	0	1	1	0
4	1	1	0	$2 = 10_2$	0	1
5	1	1	1	$3 = 11_2$	1	1
6	1	0	1	$2 = 10_2$	0	1
7	0	0	1	1	1	0
8	0	0	0	0	0	0

Tablo 1.1.6: 57 + 28 toplamının sütun sütun hesabı (sütun 1 en sağdadır).



Şekil 1.1.6: 57 + 28 toplamının son hâli. Üst satırdaki kırmızı 1'ler eldelerdir; dördüncü sütunda doğan elde üç sütun boyunca ilerler. Sonuç $01010101_2 = 85$'tir.

Sonuç $01010101_2 = 64 + 16 + 4 + 1 = 85$'tir ve $57 + 28 = 85$ ile uyuşur. Sekizinci sütundan dışarıya elde çıkmadığı için sonuç 8 bite sığmıştır. Toplam 255'i aşsaydı sekizinci sütundan bir elde taşacak ve 8 bitlik sonuç yanlış olacaktı. Bu durum **taşmanın** ilk örneğidir ve negatif sayılarla birlikte ayrıntılı incelenir.

1.1.3 Negatif sayılar ve taşma

1.1.3.1 İkiye tümleyen

Şimdiye kadar yalnız sıfır ve pozitif tam sayılar yazıldı. Negatif sayılar için ilk akla gelen yol, en soldaki biti işaret için ayırmaktır: 0 pozitif, 1 negatif. Bu gösterime **işaret-büyüklük** denir ve kâğıt üzerinde doğal görünür. Ama iki kusuru vardır. Sıfırın iki ayrı gösterimi olur (+0 ve −0). Daha önemlisi, toplama devresi işaretlere göre ayrı ayrı davranmak zorunda kalır. Bilgisayarlar bu yüzden başka bir yöntem kullanır.

Tanım 1.1.3

İkiye tümleyen (*two's complement*) gösteriminde bir pozitif sayının negatifi şöyle bulunur: sayının bütün bitleri ters çevrilir (0 ↔ 1), sonra sonuca 1 eklenir. Bitleri yalnız ters çevirmekle elde edilen ara sonuca **birler tümleyeni** denir.

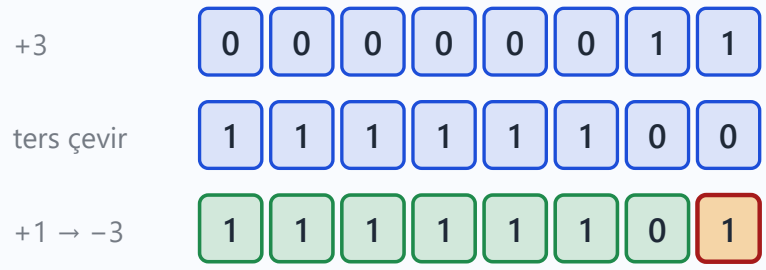
Bu gösterimde en soldaki bit bir **işaret biti** gibi davranır: 0 ise sayı pozitif, 1 ise negatiftir. 8 bitle temsil edilebilen aralık −128 … 127'dir. Genel olarak n bitle aralık $-2^{n-1} \dots 2^{n-1} - 1$'dir. Negatif tarafta bir sayı fazla vardır, çünkü sıfır pozitif tarafta yer tutar.

Örnek 1.1.4

8 bitlik bir bilgisayarda −3'ü bulalım. 3 sayısı `00000011` 'dir.

1. Bütün bitler ters çevrilir: `00000011` → `11111100` (birler tümleyeni).
2. 1 eklenir: `11111100` + 1 = `11111101` .

Sonuç `11111101` , −3'ün 8 bitlik ikiye tümleyen gösterimidir.



Şekil 1.1.7: −3'ün bulunuşu. Üst satır 3, orta satır bitleri ters çevrilmiş hâli, alt satır 1 eklendikten sonraki −3'tür. Alttaki sayılar basamak ağırlıklarıdır; en soldaki bitin ağırlığı −128'dir.

Ters yönde de aynı işlem uygulanır. Bir negatif sayının mutlak değerini bulmak için yine bitler ters çevrilir ve 1 eklenir: `11111101` → `00000010` → `00000011` = 3. İşlem kendi tersidir; iki kez uygulanınca başa dönülür.

Örnek 1.1.5

−20'nin 8 bitlik gösterimi şöyle bulunur. $20 = 16 + 4 = 00010100_2$. Bitler ters çevrilir: `11101011` . 1 eklenir: `11101100` . Sağlama için bu bit dizisi işaretsiz okunursa $128 + 64 + 32 + 8 + 4 = 236$ çıkar ve $236 = 256 - 20$'dir. Bu eşitlik rastlantı değildir; bir sonraki kesim nedenini açıklar.

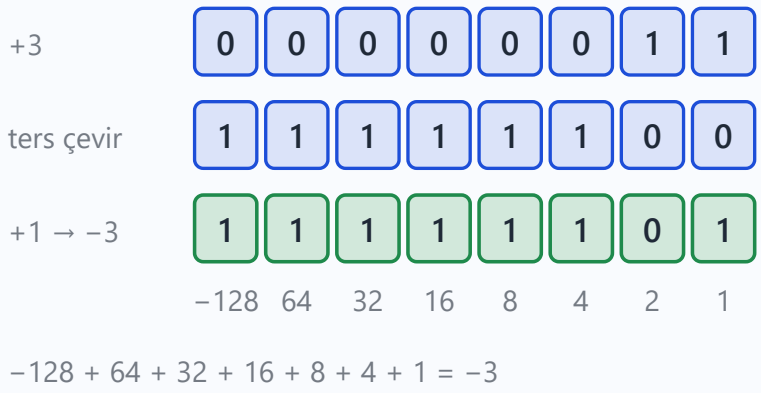
1.1.3.2 İkiye tümleyen neden işe yarar?

İkiye tümleyenin asıl üstünlüğü, toplama devresinin hiç değişmeden negatif sayılarla da çalışmasıdır. $3 + (-3)$ toplamına bakalım:

```
00000011 + 11111101 = 1 00000000
```

Sonuç dokuz bittir. Bilgisayar yalnız sekiz bit tuttuğu için en soldaki dokuzuncu bit atılır ve geriye tam olarak `00000000` = 0 kalır. Toplama devresi sayının işaretini hiç bilmez; bitleri her zamanki gibi toplar ve doğru sonucu verir. Çıkarma da ayrı bir devre istemez: $a - b$, $a + (-b)$ olarak hesaplanır. Ters çevirme kapıları ve bir toplayıcı, hem toplamayı hem çıkarmayı yapar.

Aynı gösterime ikinci bir bakış daha vardır ve işaretli bir sayıyı ondalığa çevirmenin en kısa yoludur. 8 bitlik ikiye tümleyende en soldaki bitin ağırlığı +128 değil, **−128**'dir. Öteki bitlerin ağırlıkları değişmez. Bu yüzden en soldaki bit 1 ise sayı mutlaka negatiftir: geri kalan yedi bit en fazla 127 ekleyebilir ve −128'i aşamaz.



Şekil 1.1.8: −3'ün ağırlıklarla okunuşu. En soldaki bitin ağırlığı −128 alınınca *−128 + 64 + 32 + 16 + 8 + 4 + 1 = −3* bulunur.

Örnek 1.1.6

`10110110` bit dizisi işaretsiz ve işaretli olarak kaçtır?

İşaretsiz okumada bütün ağırlıklar pozitiftir: *128 + 32 + 16 + 4 + 2 = 182*.

İşaretli okumada en soldaki bitin ağırlığı −128'dir: *−128 + 32 + 16 + 4 + 2 = −74*. Aynı sonuca *182 − 256 = −74* yoluyla da varılır. En soldaki bit 1 olduğunda işaretli değer, işaretsiz değerden her zaman $2^8 = 256$ eksiktir, çünkü o bitin katkısı +128 yerine −128 alınır ve fark 256'dır.

Bu gözlem genel bir kural verir. n bitlik bir ikiye tümleyen sayıda $-x$, işaretsiz olarak $2^n - x$ diye saklanır. −20'nin 236, −3'ün 253 olarak görünmesi bundandır. Sayıların bir saat kadranı gibi dizildiğini düşünmek yardımcı olur: 0'dan saat yönünde 127'ye kadar pozitifler, 127'den sonra −128 ve oradan −1'e kadar negatifler gelir. −1'den bir adım sonra yeniden 0'a dönülür.

1.1.3.3 Java'da ikiye tümleyen

Java'nın bütün tam sayı tipleri (`byte`, `short`, `int`, `long`) ikiye tümleyen kullanır. Bu, Java dil belirtiminde açıkça yazılıdır ve her platformda aynıdır. Aşağıdaki program aynı kuralı Java ile sınar.

```
int sayi = 3;
int eksi = ~sayi + 1; // ters çevir, 1 ekle
System.out.println(eksi);
byte b = (byte) 0b11111101;
System.out.println(b);
System.out.println(Integer.toBinaryString(-3));
System.out.println(Byte.MIN_VALUE + " .. "
    + Byte.MAX_VALUE);
```

```
-3
-3
11111111111111111111111111111101
-128 .. 127
```

`~` işleci bir tam sayının bütün bitlerini ters çevirir; adı **bit düzeyinde değil** (*bitwise NOT*) işlecidir. `~sayi + 1` tam olarak ikiye tümleyen kuralıdır ve sonuç −3 çıkar. Java'da `-x` ile `~x + 1` her tam sayı için aynı değeri verir.

İkinci kısımda `0b11111101` sabiti bir `byte`'a dönüştürülüyor. Bu sabit Java için bir `int`'tir ve değeri 253'tür. 253, `byte` aralığına sığmadığı için açık bir **tür dönüşümü** (*cast*) gerekir: `(byte)`. Dönüşüm yalnız en sağdaki 8 biti alır; bu bitler `11111101`'dir ve işaretli okununca −3 olur.

`Integer.toBinaryString(-3)` otuz iki bitin hepsini yazar: başta otuz tane 1 ve sonda `01`. Gösterim 32 bite genişleyince değişmez; sola eklenen bitler işaret bitinin kopyasıdır. Bu kurala **işaret genişletme** denir. Pozitif bir sayı sola sıfırlarla, negatif bir sayı sola birlerle genişletilir ve değer korunur. Son satır `byte` tipinin sınırlarını yazar: `-128 .. 127`.

Java'da `byte`'ın işaretli olması bazen şaşırtıcıdır. Ağdan ya da dosyadan okunan ham baytlar çoğu zaman 0–255 arasında düşünülür. Java'da ise 200 değerli bir bayt −56 olarak görünür. Baytı işaretsiz okumak için `b & 0xFF` ifadesi kullanılır: `&` işlemi işaret genişletmesiyle gelen birleri siler ve 200'ü geri verir. Aynı işi `Byte.toUnsignedInt(b)` metodu da yapar.

1.1.3.4 Taşma

> **Tanım 1.1.4**
>
> **Taşma** (*overflow*), bir işlemin sonucunun tipin aralığına sığmaması durumudur. Java'da tam sayı taşması hata vermez: sonuç "saat gibi" başa sarar ve aralığın öteki ucundan devam eder.

```
byte b = 127;
b++;
System.out.println(b);
int m = Integer.MAX_VALUE;
System.out.println(m + 1);
System.out.println(Math.addExact(m, 1));
```

```
-128
-2147483648
Exception in thread "main" java.lang.ArithmeticException: integer overflow
    at java.base/java.lang.Math.addExact(Math.java:911)
    at Tasma.main(Tasma.java:7)
```

İlk üç satırda `b` 127'dir ve 1 artırılınca 128 olması gerekir. 127 = `01111111`'e 1 eklenince `10000000` çıkar. Bu bit dizisi işaretli okununca −128'dir. Toplama devresi doğru çalıştı; yalnız sonuç, işaret bitine taşınan elde yüzünden negatif bölgeye geçti. Saat kadranında 127'den bir adım ileri gidince −128'e gelinir. `Integer.MAX_VALUE + 1` için de aynı şey olur ve sonuç -2^{31} = −2.147.483.648 çıkar.

Taşmanın tehlikesi sessiz olmasıdır. Program çökmez, uyarı vermez, yalnız yanlış bir sayıyla çalışmayı sürdürür. Bir banka hesabının bakiyesi ya da bir sayaç taşarsa hata çok sonra, başka bir yerde ortaya çıkar. Bu yüzden Java, denetimli işlemler için `Math.addExact`, `Math.subtractExact` ve `Math.multiplyExact` metotlarını sunar. Bu metotlar sonucun sığmadığını görünce `ArithmeticException` istisnası fırlatır ve program o noktada durur. Son satırdaki çağrı bu yüzden bir sayı yazmaz; istisna iletisi hatanın yerini ve nedenini bildirir. Çıktıdaki `at` satırları, istisnanın hangi metot çağrılarından geçerek fırlatıldığını gösterir. Burada kod `Tasma` adlı bir sınıfın `main` metoduna yerleştirilerek çalıştırıldı ve satır numaraları bu dosyaya göredir.

Taşmayı önlemenin bir yolu daha geniş bir tip kullanmaktır. İki `int`'in çarpımı `long`'a sığar, ama çarpımın kendisi `long` olarak yapılmalıdır. `long x = 100000 * 100000;` yazmak işe yaramaz: sağ taraf iki `int` çarpımıdır, önce 32 bitte hesaplanır ve taşar; taşmış sonuç ancak ondan sonra `long`'a atanır. Doğrusu, çarpanlardan birini `long` yapmaktır: `100000L * 100000`. Sabitin sonundaki `L` eki onu `long` yapar ve bütün çarpım 64 bitte yürür.

Örnek 1.1.7

`01101101 + 00110110` toplamı kaçtır ve sonuç 8 bitlik işaretli bir `byte` olarak nasıl okunur?

Ondalıkta *109 + 54 = 163*'tür. Bitler toplanınca `10100011` çıkar: *128 + 32 + 2 + 1 = 163*. Dokuzuncu bite elde taşmadı; işaretsiz yorumda sonuç doğrudur. İşaretli yorumda ise en soldaki bit 1'dir ve değer *163 − 256 = −93* olur. İki pozitif sayının toplamı negatif çıktı. İşaretli toplamada taşmanın belirtisi tam olarak budur: aynı işaretli iki sayı toplanıp işareti farklı bir sonuç elde edilir. Java'da `(byte) (0b01101101 + 0b00110110)` ifadesi −93 verir.

1.1.4 Kesirli sayılar: kayan nokta

1.1.4.1 Kesir ve üs

Sınırlı sayıda rakamla ancak sınırlı sayıda değer temsil edilebilir. 32 bitlik bir `int` yaklaşık ±2 milyara kadar sayar. Işık hızının (yaklaşık $3 \cdot 10^8$ m/s) karesini almaya bile yetmez. Bir elektronun yükünü (yaklaşık $1{,}6 \cdot 10^{-19}$ coulomb) ise hiç yazamaz. Mühendislik hesaplarında hem çok büyük hem çok küçük sayılar gerekir.

Çözüm, bilimsel gösterimdeki fikirdir. Sayı iki parçaya ayrılır: anlamlı rakamları taşıyan **kesir** (*mantissa*, mantis) ve ondalık noktanın yerini belirleyen **üs**. Parçalar ayrı ayrı saklanır. Sekiz rakamlık bir alanda `31415901` yazılı olsun. İlk altı rakam (314159) kesri, son iki rakam (01) 10'un kuvvetini,

yani 10^{1}'i belirtsin. Kesir 0,314159 diye okunursa sayı $0{,}314159 \times 10^{1} = 3{,}14159$ olur.



Şekil 1.1.9: Üstte sekiz rakamın altısı kesir, ikisi üs olarak kullanılıyor. Altta Java'nın `double` tipi: 1 işaret biti, 11 üs biti ve 52 kesir biti.

Bu gösterimin adı da buradan gelir: **kayan nokta** (*floating point*). Üs değiştikçe ondalık noktanın yeri kayar. Aynı kesir 10^{3} ile çarpılırsa $0{,}314159 \times 10^{3} = 314{,}159$ olur. Rakamlar aynıdır; yalnız noktanın yeri değişmiştir. Böylece sınırlı sayıda rakamla hem çok büyük hem çok küçük sayılar yazılır. Bedeli şudur: her sayı yalnız kesre sığan kadar anlamlı rakamla, yani *yaklaşık* olarak saklanır.

1.1.4.2 IEEE 754 ve Java'nın float, double tipleri

Bilgisayarlar aynı fikri ikili tabanda uygular. Kuralları **IEEE 754** standardı belirler ve bugün neredeyse bütün işlemciler bu standarda uyar. Java'da iki kayan noktalı tip vardır: `float` 32 bit, `double` 64 bittir.

Tip	İşaret	Üs	Kesir	Anlamlı ondalık rakam	En büyük değer
`float`	1	8	23	yaklaşık 7	$\approx 3{,}4 \cdot 10^{38}$
`double`	1	11	52	yaklaşık 15–16	$\approx 1{,}8 \cdot 10^{308}$

Tablo 1.1.7: Java'nın kayan noktalı tiplerinin bit dağılımı.

İkili kayan noktada sayı $(-1)^{işaret} \times 1{,}kesir_2 \times 2^{üs}$ biçimine getirilir. Sıfırdan farklı her ikili sayı "1,..." ile başlayacak şekilde kaydırılabildiği için baştaki 1 saklanmaz; kesir alanı yalnız virgülden sonraki bitleri tutar. Üs ise bir sabit eklenerek negatif olmayan bir sayı olarak yazılır; `float`'ta bu sabit 127, `double`'da 1023'tür.

Örnek 1.1.8

−6,5'in `float` gösterimi beş adımda bulunur.

1. İşaret negatiftir: işaret biti 1.

2. 6,5 = 4 + 2 + 0,5'tir. İkilide $110{,}1_2$ yazılır (virgülden sonraki ilk basamağın ağırlığı $2^{-1} = 0{,}5$'tir).
3. Virgül iki basamak sola kaydırılır: $110{,}1_2 = 1{,}101_2 \times 2^2$.
4. Üs 2'dir; 127 eklenince 129 = `10000001` saklanır.
5. Kesir, baştaki 1 atılarak `101` ve ardından 20 sıfırdır.

Sonuç `1 10000001 10100000000000000000000`'dır. Java'da `Integer.toBinaryString(Float.floatToIntBits(-6.5f))` tam olarak bu 32 biti yazar.

−6,5 gibi sayılar ikilide tam yazılır, çünkü 0,5 bir ikinin kuvvetidir. 0,1 ise öyle değildir. 1/3'ün ondalıkta 0,3333… diye sonsuza kadar sürmesi gibi, 0,1 de ikilide $0{,}000110011001100\ldots_2$ diye sonsuza kadar tekrar eder. Kesir alanı sonlu olduğu için bir yerde kesilir ve yuvarlanır. Bellekteki 0,1 aslında 0,1'e çok yakın ama ondan farklı bir sayıdır.

1.1.4.3 Yuvarlama hatası ve karşılaştırma

Yuvarlamanın en bilinen sonucu, Java'da `0.1 + 0.2` toplamının `0.30000000000000004` çıkmasıdır. 0,1 ve 0,2'nin saklanan değerlerindeki küçük hatalar toplanır ve sonuç, 0,3'ün saklanan değerine eşit olmaz. Bu yüzden `0.1 + 0.2 == 0.3` ifadesi `false` verir. Bu bir Java hatası değildir; aynı sonuç C, Python ve JavaScript'te de görülür, çünkü hepsi aynı IEEE 754 donanımını kullanır.

Kayan noktalı sayılar bu yüzden `==` ile karşılaştırılmaz. Bunun yerine iki sayının farkının yeterince küçük olup olmadığına bakılır:

```
double a = 0.1 + 0.2;
double b = 0.3;
System.out.println(a == b);                    // false
System.out.println(Math.abs(a - b) < 1e-9);    // true
```

Buradaki 10^{-9} sınırına **tolerans** denir ve problemin ölçeğine göre seçilir. Volt cinsinden ölçülen bir gerilim için 10^{-9} çok küçük bir farktır; ama 10^{-12} farad mertebesindeki bir sığa için çok büyüktür. Para gibi kuruşu kuruşuna doğru olması gereken değerlerde ise `double` hiç kullanılmaz. Ya tutarlar kuruş cinsinden `long` olarak saklanır ya da Java'nın ondalık tabanda hesap yapan `BigDecimal` sınıfı kullanılır.

Sık yapılan hata

Bir döngüyü kayan noktalı bir sayaçla eşitlik koşuluna bağlamak: `for (double x = 0; x != 1.0; x += 0.1)`. On adımda `x` 1,0 olacak gibi görünür, ama birikmiş yuvarlama hataları yüzünden `x` 0,9999999999999999 olur ve 1,0'a hiç eşit olmaz; döngü sonsuza kadar döner. Sayaçlar tam sayı olmalı ya da koşul `x < 1.0` gibi bir eşitsizlikle yazılmalıdır.

1.1.5 Harfler ve dizgiler

1.1.5.1 Karakter kodlaması ve ASCII

Bilgisayar yalnız sayılarla çalışır. Metin işleyebilmesi için her harfe bir sayı karşılık getirilmelidir. Hangi harfin hangi sayıyla gösterileceği keyfî bir seçimdir, ama herkesin aynı seçimi yapması gerekir. Yoksa bir bilgisayarda "A" diye yazılan sayı, ötekinde başka bir harf olarak okunur.

> **Tanım 1.1.5**
>
> **Karakter kodlaması** (*character encoding*), karakterlerin hangi sayılarla ve bellekte hangi bayt dizileriyle saklanacağını belirleyen kuraldır.

En eski ve en etkili kodlama **ASCII**'dir (*American Standard Code for Information Interchange*, Bilgi Değişimi İçin Amerikan Standart Kodu). ASCII'yi 1963'te ABD standartlar kurumu ASA (bugünkü ANSI) yayımladı. Büyük ve küçük İngiliz harflerini, rakamları, noktalama işaretlerini ve bazı özel karakterleri sayılarla temsil eder.

ASCII 7 bitlik bir koddur ve $2^7 = 128$ karakter tanımlar (0–127). 0–31 arası ve 127, ekranda görünmeyen **kontrol karakterleridir**. Bunlar ilk olarak yazıcıları ve teleks makinelerini yönetmek için tasarlandı: satır başına dön, bir satır aşağı in, zil çal. 32–126 arası ise yazdırılabilir karakterlerdir. Tablo 1.1.8 birkaç önemli kodu gösterir.

Karakter	Ondalık	İkili
`'A'`	65	`1000001`
`'a'`	97	`1100001`
`'0'`	48	`0110000`
boşluk	32	`0100000`
yeni satır `\n`	10	`0001010`

Tablo 1.1.8: Bazı ASCII kodları.

Tablodaki düzen bilinçli olarak seçilmiştir. Harfler alfabetik sırayla ardışık kodlara dizilir: `'A'` = 65, `'B'` = 66, …, `'Z'` = 90. Bu sayede harfleri karşılaştırmak, sayıları karşılaştırmaya dönüşür. Küçük harf, büyük harfin 32 fazlasıdır: `'a'` = 97 = 65 + 32. İkilide bu fark yalnız tek bir bittir, 2^5 basamağı. `'A'` = `1000001`, `'a'` = `1100001`'dir. Büyük-küçük harf dönüşümü tek bir biti açıp kapatmaktan ibarettir. Rakamlar da 48'den başlayarak dizilir; bir rakam karakterinin sayısal değeri `c - '0'` ile bulunur: `'7' - '0'` = 55 − 48 = 7.

Java'da `char` bir sayı tipidir ve aritmetiğe katılabilir. `(char) ('a' - 32)` ifadesi `'A'` verir. Bu yakınlık bazen şaşırtır: `System.out.println('a' + 'b');` satırı `ab` değil, `195` yazar. İki `char`'ın toplamı bir `int`'tir: *97 + 98 = 195*. Aynı satır `"" + 'a' + 'b'` diye yazılırsa önce boş dizgiye `'a'` ekle-

nir, sonuç dizgi olduğu için `'b'` de eklenir ve `ab` çıkar. Java'da `+` işleci soldan sağa işler ve işlenenlerden biri dizgi olduğu anda birleştirmeye döner.

7 bit, bir baytın sekiz bitinden birini boşta bırakır. 8 bitlik genişletmeler bu biti kullanarak 128–255 arasına ek harfler koyar ve 256 sembole çıkar. Türkçe için ISO 8859-9 (Latin-5) ve Windows-1254 bu türden kodlamalardır; ğ, ş, ı ve İ harflerini bu üst yarıya yerleştirirler. Ama her dil kendi üst yarısını tanımlayınca sorun çıktı. Aynı 253 değeri bir tabloda ı, ötekinde ý harfiydi. Türkçe bir metin yanlış kodlamayla açılınca harfler bozuk görünüyordu. Tek bir 8 bitlik tablo dünyanın bütün dillerine yetmez; Çince ve Japoncada binlerce karakter vardır.

1.1.5.2 ASCII tablosu

Yazdırılabilir ASCII karakterleri 16 sütunlu bir tabloya dizilince kodları kolayca okunur (Tablo 1.1.9). Bir karakterin kodu, satır başlığı ile sütun başlığının toplamıdır. Örneğin `'J'` 64 satırında ve 10 sütunundadır: *64 + 10 = 74*.

+	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
32	SP	!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Tablo 1.1.9: Yazdırılabilir ASCII karakterleri (32–126). Kod = satır + sütun; SP boşluk karakteridir.

Tablonun son hücresi DEL (127) bir kontrol karakteridir. 0–31 arasındaki kontrol karakterlerinden üçü bugün de her gün kullanılır: 9 sekme (`\t`), 10 yeni satır (`\n`) ve 13 satır başı (`\r`). Windows metin dosyalarında satır sonu `\r\n` ikilisiyle, Linux ve macOS'ta yalnız `\n` ile işaretlenir. Bir dosyanın bir sistemde düzgün, ötekinde tek satır görünmesinin nedeni çoğu zaman bu farktır. 0–255 arasının tamamı ve karakter ya da kodla arama, ASCII Tablosu sayfasında yer alır.

1.1.5.3 Unicode

Ülkeye özgü kod tablolarının yarattığı karmaşa, bütün diller için tek bir tablo fikrini doğurdu.

Tanım 1.1.6

Unicode, dünyadaki bütün yazı sistemlerinin harf, sembol ve işaretlerine birer numara veren standarttır. Bu numaraya **kod noktası** (*code point*) denir ve `U+` ön ekiyle onaltılı yazılır: U+0041 = A, U+00E7 = ç.

Unicode'u 1991'den beri Unicode Konsorsiyumu geliştirir. Uluslararası karşılığı olan ISO/IEC 10646 standardı onunla eş zamanlı tutulur. İlk sürüm 16 bitlikti ve 65.536 karakterin yeteceği düşünülmüştü. Tarihî yazılar, matematik sembolleri ve Çin-Japon-Kore ideogramlarının tamamı eklenince bu alan yetmedi. Kod alanı U+10FFFF'e kadar genişletildi. Böylece 1.114.112 kod noktası oluştu; bugün bunların 150.000'den fazlasına karakter atanmıştır.

Unicode'un ilk 128 kod noktası ASCII ile birebir aynıdır: U+0041 hem ASCII'de hem Unicode'da 65'tir. İlk 256 kod noktası da Batı Avrupa dilleri için kullanılan Latin-1 (ISO 8859-1) tablosuyla örtüşür; ç bu yüzden hem Latin-1'de hem Unicode'da 231'dir. Türkçeye özgü ğ, ş ve ı ise Latin-1'de yoktur ve 256'nın üstündedir (Tablo 1.1.10).

Karakter	Kod noktası	Ondalık	UTF-8 baytları
A	U+0041	65	`41`
ç	U+00E7	231	`C3 A7`
ğ	U+011F	287	`C4 9F`
€	U+20AC	8364	`E2 82 AC`
😀	U+1F600	128512	`F0 9F 98 80`

Tablo 1.1.10: Bazı karakterlerin kod noktaları ve UTF-8 baytları (onaltılı).

Kod noktası bir numaradır; bellekte nasıl saklanacağını söylemez. En basit yol, her karakteri sabit 32 bitle saklamaktır. Bu kodlamaya UTF-32 denir. Ama İngilizce bir metinde her harfin 7 bitlik bir ASCII kodu vardır ve 32 bit ayırmak yerin dörtte üçünü boşa harcar. Bu yüzden değişken uzunluklu **UTF** (*Unicode Transformation Format*, Unicode Dönüşüm Biçimi) kodlamaları tanımlanmıştır.

1.1.5.4 UTF-8

UTF-8, bir kod noktasını 1 ile 4 bayt arasında değişen uzunlukta saklar. Sık kullanılan karakterler az, nadir olanlar çok bayt kullanır. ASCII karakterleri tek bayttır ve bu bayt ASCII koduyla aynıdır. Bu yüzden saf ASCII bir metin, hiçbir değişiklik yapılmadan geçerli bir UTF-8 metnidir; UTF-8 ASCII ile **geriye uyumludur**. Bugün web sayfalarının çok büyük bölümü UTF-8 ile kodlanır.

Bir UTF-8 okuyucu, bir karakterin kaç bayt süreceğini ilk baytın başındaki bitlerden anlar. Tablo 1.1.11 kalıpları verir. `x` ile gösterilen yerlere kod noktasının bitleri sırayla yerleştirilir.

Kod noktası aralığı	Bayt kalıbı	Bayt sayısı	Taşınan bit
U+0000–U+007F	`0xxxxxxx`	1	7
U+0080–U+07FF	`110xxxxx 10xxxxxx`	2	11
U+0800–U+FFFF	`1110xxxx 10xxxxxx 10xxxxxx`	3	16
U+10000–U+10FFFF	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`	4	21

Tablo 1.1.11: UTF-8 bayt kalıpları.

Kalıpların tasarımı akıllıcadır. 0 ile başlayan bir bayt tek başına bir ASCII karakteridir. 110, 1110 ya da 11110 ile başlayan bayt bir karakterin ilk baytıdır ve baştaki 1'lerin sayısı toplam bayt sayısını verir. 10 ile başlayan bayt ise her zaman bir devam baytıdır. Okuyucu bir metnin ortasından okumaya başlasa bile devam baytlarını atlayıp bir sonraki karakterin başını bulabilir.

Örnek 1.1.9

ç karakterinin UTF-8 kodlaması şöyle bulunur. Kod noktası U+00E7 = 231'dir ve U+0080–U+07FF aralığında olduğu için iki bayt kullanır. 231, 11 bitle `00011 100111` diye yazılır. İlk beş bit birinci baytın `110xxxxx` kalıbına, son altı bit ikinci baytın `10xxxxxx` kalıbına yerleşir:

> `110 00011` = C3 `10 100111` = A7

Sonuç C3 A7'dir. Aynı yolla € (U+20AC = `0010 000010 101100`) üç bayta yerleşir: `1110 0010` = E2, `10 000010` = 82, `10 101100` = AC.



Şekil 1.1.10: ç karakterinin UTF-8 kodlaması. Renkli kutular kalıbın sabit bitleri, diğerleri kod noktasının bitleridir; iki bayt onaltılı C3 ve A7'dir.

Java'da bir dizginin UTF-8 ile kaç bayt tuttuğu `getBytes` metoduyla görülür. Kodun derlenmesi için dosyanın başına `import java.nio.charset.StandardCharsets;` satırı eklenmelidir.

```
String kelime = "çikolata";
System.out.println(kelime.length());
System.out.println(kelime.getBytes(
    StandardCharsets.UTF_8).length);
```

```
8
9
```

`"çikolata"` sekiz karakterlidir ve `length()` 8 verir. UTF-8'de ç iki bayt, öteki yedi harf birer bayt tutar: *2 + 7 = 9*. Bir dosyanın bayt cinsinden boyutu, içindeki karakter sayısıyla bu yüzden aynı değildir.

Kodlama yalnız uzunluğu değil, okunabilirliği de belirler. Bir metin hangi kodlamayla yazıldıysa o kodlamayla okunmalıdır. UTF-8 ile yazılmış “ç” iki bayttır (C3 A7). Bu iki bayt Windows-1254 tablosuyla okunursa her biri ayrı bir karakter sayılır ve ekranda “Ã§” görünür. Türkçe karakterlerin bozuk göründüğü dosyaların çoğunda sorun budur. Java'da dosya okurken ve yazarken kodlamayı her zaman açıkça belirtmek, örneğin `StandardCharsets.UTF_8` vermek, bu hatayı önler.

1.1.5.5 Java'da char, UTF-16 ve emoji

Java tasarlandığında Unicode henüz 16 bitlikti. Java'nın `char` tipi bu yüzden 16 bit seçildi ve her karakterin tek bir `char`'a sığacağı varsayıldı. Unicode genişleyince bu varsayım bozuldu. Bugün bir `char`, bir karakter değil, bir **UTF-16 birimidir**. UTF-16 kodlaması, U+FFFF'e kadar olan karakterleri tek bir 16 bitlik birimle yazar; ş = 01 5F, € = 20 AC. Daha büyük kod noktaları ise iki birimle yazılır. Bu ikiliye **vekil çift** (*surrogate pair*) denir.

Gündelik metinlerin çoğu bu ayrımı hissettirmez, çünkü Türkçe, Rusça, Arapça ve Çincedeki yaygın karakterlerin hepsi U+FFFF'in altındadır. Fark en çok emojilerde ortaya çıkar. Emojiler, duygu, ifade ve nesneleri hızlıca aktaran simgelerdir ve bilgisayar onları resim olarak değil, metin olarak işler. Standartlarını Unicode Konsorsiyumu belirler ve her emoji benzersiz bir kod noktasıdır: 😀 = U+1F600. Kod her cihazda aynıdır; böylece gönderilen emoji karşı tarafta aynı anlamla okunur. Çizimi, yani **glifi** ise yazı tipine ve üreticiye göre farklı görünebilir.

Bazı emojiler birden çok kod noktasının birleşimidir. Aile emojisi 👨‍👩‍👧, beş kod noktasından oluşur: adam, ZWJ, kadın, ZWJ, kız. ZWJ (*zero width joiner*, sıfır genişlikli birleştirici) U+200D'dir ve ekranda görünmez; yalnız komşu emojilerin tek bir glif olarak çizilmesini ister. Bu birleştiriciyi tanımayan bir sistem, üç ayrı yüz gösterir.

```
String yuz = "😀";
System.out.println(yuz.length());
System.out.println(
    yuz.codePointCount(0, yuz.length()));
String aile = "👨‍👩‍👧";
System.out.println(aile.length());
```

```
2
1
8
```

😀 dizgisinin `length()` değeri 2'dir, çünkü U+1F600 iki `char`'lık bir vekil çifttir: `\uD83D` ve `\uDE00`. `codePointCount` ise kod noktalarını sayar ve 1 verir. Aile emojisinde üç insan emojisi ikişer `char`, iki ZWJ birer `char` tutar: *3·2 + 2·1 = 8*. Ekranda tek bir simge görünse de `length()` 8, kod noktası sayısı 5'tir. UTF-8'de bu emoji *3·4 + 2·3 = 18* bayt tutar.

Bu kesimin Java için sonucu şudur: `String` karakterleri UTF-16 ile saklar ve `length()` "karakter sayısı" değil, `char` sayısıdır. Türkçe metinlerde ikisi neredeyse her zaman aynıdır. Emojili, matematik sembollü ya da tarihî yazılı metinlerde ise farklıdır. Böyle metinlerde karakterler üzerinde tek tek gezinmek için `charAt` yerine `codePoints()` ya da `codePointAt` kullanılır. Java 9'dan bu yana `String`, yalnız Latin-1 karakteri içeren dizgileri içeride karakter başına bir baytla saklar. Bu bir bellek iyileştirmesidir; `length()` ve `charAt()` her durumda UTF-16 birimleriyle çalışmaya devam eder.

> **Sık yapılan hata**
>
> Bir dizginin `length()` değerini ekranda görünen simge sayısı ya da bayt sayısı sanmak. `"😀".length()` 1 değil 2'dir; `"ağaç".length()` 4 iken aynı dizgi UTF-8'de 6 bayt tutar. Bir veritabanı alanına ya da ağ paketine sığıp sığmadığı denetlenecekse kodlanmış bayt sayısına, kullanıcıya gösterilecek uzunluk hesaplanacaksa kod noktalarına bakılmalıdır.

1.1.5.6 Kod izleme: bir dizginin karakter kodları

Bir dizgi bellekte iki parçayla durur: karakterlerin sayı olarak dizildiği bir dizi ve bu dizinin uzunluğu. Uzunluk ayrıca tutulur; metnin nerede bittiği böyle bilinir. Aşağıdaki program `"çikolata"` dizgisinin her karakterinin kodunu yazdırır.

```
String kelime = "çikolata";
for (int i = 0; i < kelime.length(); i++) {
  char ch = kelime.charAt(i);
  System.out.print((int) ch + " ");
}
System.out.println();
System.out.println("Uzunluk: " + kelime.length());
```

Döngü `i = 0`'dan başlar ve `i < kelime.length()` doğru olduğu sürece döner. `charAt(i)`, `i` konumundaki `char`'ı verir. `(int) ch` tür dönüşümü, karakterin kendisini değil, sayısal kodunu yazdırır; dönüşüm olmasa ekrana harfin kendisi basılırdı. `print`, `println`'den farklı olarak satırı bitirmez; kodlar aynı satıra yan yana dizilir. Döngüden sonraki `println()` satırı bitirir. Tablo 1.1.12 her turu gösterir.

`i`	**Koşul** `i < 8`	`ch`	`(int) ch`	**Açıklama**
0	doğru	`'ç'`	231	U+00E7; ASCII'de yok
1	doğru	`'i'`	105	
2	doğru	`'k'`	107	
3	doğru	`'o'`	111	
4	doğru	`'l'`	108	
5	doğru	`'a'`	97	

`i`	**Koşul** `i < 8`	`ch`	`(int) ch`	**Açıklama**
6	doğru	`'t'`	116	
7	doğru	`'a'`	97	5. konumdaki `'a'` ile aynı kod
8	yanlış	Döngü biter; satır sonu ve `Uzunluk: 8` yazılır.		

Tablo 1.1.12: `"çikolata"` için döngünün izlemesi.

```
231 105 107 111 108 97 116 97
Uzunluk: 8
```



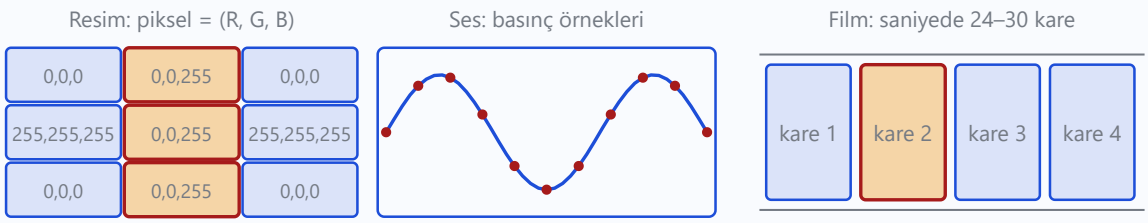
Şekil 1.1.11: `"çikolata"` dizgisi bellekte: solda uzunluk (8), sağda indisleri 0–7 olan karakterler ve altlarında kodları.

ç dışındaki bütün kodlar ASCII tablosundaki değerlerdir. 231 ise ç'nin Unicode kod noktasıdır (U+00E7). Aynı değer Latin-1 ve Türkçe ISO 8859-9 tablolarında da 231'dir. Bu yüzden program hangi tabloyla düşünülürse düşünülsün aynı sayıyı yazar. `(int) ch` ifadesi UTF-8 baytlarını değil, kod noktasını verir. ç'nin UTF-8 karşılığı C3 A7 baytlarıdır, ama Java belleğinde ç tek bir 16 bitlik `char` olarak durur ve değeri 231'dir.

1.1.6 Yapılandırılmış bilgi

1.1.6.1 Resim, ses ve film

Sayılar harfleri, harfler metinleri temsil ettiği gibi, sayı dizileri de çok daha karmaşık bilgiyi temsil eder. Bir dosyada ne saklanırsa saklansın, bilgisayar için o dosya bir bayt dizisidir. Dizinin anlamı, onu yazan ve okuyan programların uyduğu kurallardan, yani **dosya biçiminden** gelir. Bilgisayarla etkileşimimizin çoğu, bu kurallar sayesinde sayı dizilerini resim, ses ya da film olarak görmemize dayanır.



Şekil 1.1.12: Yapılandırılmış bilginin üç örneği. Solda her pikseli (R, G, B) üçlüsü olan bir resim, ortada zaman içinde düzenli aralıklarla ölçülen ses basıncı, sağda saniyede 24–30 kare hızında ardışık resimlerden oluşan film.

- **Resim.** Bir resim, satırlar ve sütunlar hâlinde dizilmiş küçük noktalardan, yani **piksellerden** oluşur. Her piksel kırmızı, yeşil ve mavi ışığın miktarını veren üç sayıdır ve her sayı 0–255 arasındadır. Bu üçlüye RGB (*red, green, blue*) denir. Mavi bir piksel (0, 0, 255), beyaz bir piksel (255, 255, 255), siyah bir piksel (0, 0, 0)'dır. Her renk bileşeni bir bayta sığar ve bir piksel üç bayt tutar. Web'de kullanılan `#FF0000` renk kodu, bu üç baytın onaltılı yazılışıdır: FF kırmızı, 00 yeşil, 00 mavi.
- **Ses.** Ses, havadaki basıncın zamanla değişmesidir. Mikrofon bu değişimi bir gerilime çevirir. Analog-sayısal dönüştürücü (ADC) gerilimi düzenli aralıklarla ölçer ve her ölçümü bir sayı olarak kaydeder. Bu işleme **örnekleme** denir. CD kalitesinde saniyede 44.100 örnek alınır ve her örnek 16 bitlik bir sayıdır. Örnekleme hızının seçimi rastgele değildir. İnsan kulağı yaklaşık 20 kHz'e kadar duyar. Bir sinyali kaybetmeden örneklemek için de en yüksek frekansının iki katından hızlı ölçmek gerekir.
- **Film.** Bir film, saniyede 24 ya da 30 kare hızında gösterilen ardışık resimlerdir. Göz bu hızda geçen resimleri kesintisiz bir hareket olarak algılar. Her kare, resim gibi bir sayı dizisidir; film ise bu dizilerin dizisidir ve yanına bir ses kanalı eklenir.

1.1.6.2 Bir video ne kadar yer tutar?

Sayıların boyutunu hesaplamak, bu temsilin bedelini somutlaştırır. Aşağıdaki program, sıkıştırılmamış bir dakikalık Full HD videonun ve bir dakikalık CD sesinin kaç bayt tuttuğunu hesaplar.

```
long piksel = 1920L * 1080;  // Full HD bir kare
long kare = piksel * 3;      // R, G, B: 3 bayt
long saniye = kare * 30;     // saniyede 30 kare
System.out.println("Piksel: " + piksel);
System.out.println("Kare (bayt): " + kare);
System.out.println("1 saniye (bayt): " + saniye);
System.out.println("1 dakika (MB): "
    + saniye * 60 / 1_000_000);
// 44,1 kHz, 16 bit (2 bayt), stereo, 1 dakika
long ses = 44_100L * 2 * 2 * 60;
System.out.println("1 dk CD sesi (bayt): " + ses);
```

```
Piksel: 2073600
Kare (bayt): 6220800
1 saniye (bayt): 186624000
1 dakika (MB): 11197
1 dk CD sesi (bayt): 10584000
```

Hesap adım adım şöyledir. Full HD bir kare *1920 × 1080 = 2.073.600* pikseldir. Her piksel üç bayt tuttuğu için bir kare *2.073.600 × 3 = 6.220.800* bayttır, yani yaklaşık 6,2 MB. Saniyede 30 kare *6.220.800 × 30 = 186.624.000* bayt eder. Bir dakika bunun 60 katıdır: 11.197.440.000 bayt. Program bunu `1_000_000`'a bölerek megabayta çevirir. Tam sayı bölmesi kesri attığı için çıktı 11197 MB'tır; yani sıkıştırmasız bir dakikalık Full HD video yaklaşık **11 GB** tutar. Ses tarafında 44.100 örnek, 2 bayt ve 2 kanal (stereo) çarpılınca saniyede 176.400 bayt, bir dakikada 10.584.000 bayt, yani yaklaşık 10,6 MB bulunur.

Gerçek hayatta video ve ses **sıkıştırılarak** saklanır. H.264 gibi video kodlayıcıları ardışık kareler arasındaki benzerliği, MP3 gibi ses kodlayıcıları kulağın duyamayacağı ayrıntıları kullanarak dosyayı onlarca kat küçültür. Ama sıkıştırılmış dosya da yine bir sayı dizisidir; değişen yalnız sayıların anlamını belirleyen kurallardır.

Kodun Java açısından en önemli ayrıntısı ilk satırdaki `L` ekidir. `1920L * 1080` yazımı çarpımı `long` tipinde yapar. `L` eki olmasaydı bütün çarpımlar `int` olarak yapılırdı. Bir saniyelik değer (186.624.000) `int`'e sığar, ama bir dakikalık değer (11.197.440.000) `Integer.MAX_VALUE`'yu beş kattan fazla aşar. `1920 * 1080 * 3 * 30 * 60` ifadesi bu yüzden sessizce taşar ve −1.687.461.888 gibi anlamsız bir negatif sayı verir. Ses hesabındaki `44_100L` da aynı nedenle `L` ekiyle yazılmıştır; orada sonuç `int`'e sığardı, ama ek yazmak alışkanlık olarak güvenlidir.

1.1.7 Saklanan program

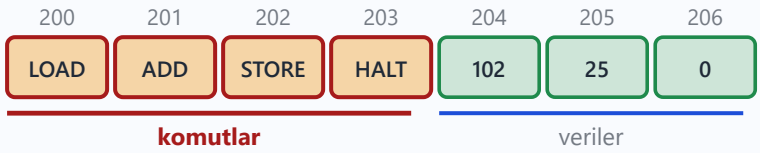
1.1.7.1 Saklanan program kavramı

Buraya kadar sayıların veriyi nasıl temsil ettiği görüldü. Son adım, işlemcinin ne yapacağını söyleyen komutların da sayı olmasıdır. Bu fikir modern bilgisayarların temel ilkesidir.

Tanım 1.1.7

Saklanan program kavramı (*stored-program concept*): komutlar da veriler gibi sayıdır ve verilerle aynı bellekte saklanır. Bellekte saklanan komutlar ve veriler birlikte programı oluşturur. Bu ilkeyle çalışan makineye **depolanan programlı bilgisayar** denir.

Her komut belirli bir görevi ya da işlemi temsil eder: bellekten oku, topla, belleğe yaz. Komutlar normalde ardışık olarak, bellekteki sıralarıyla işlenir. Ama bir koşul sağlandığında program farklı bir komuta **atlayabilir**. Programların duruma göre farklı yollar izlemesini sağlayan budur. Java'daki `if` deyimi ve döngüler, işlemci düzeyinde tam olarak bu tür atlama komutlarına dönüşür.



Şekil 1.1.13: Komutlar ve veriler aynı bellekte, yan yana adreslerde durur. 200–203 adreslerinde LOAD, ADD, STORE ve HALT komutları, 204–206 adreslerinde veriler vardır.

Fikrin önemi, ondan önceki makinelerle karşılaştırınca anlaşılır. 1945'te tamamlanan ENIAC, saniyede binlerce toplama yapabilen ilk büyük elektronik bilgisayarlardan biriydi. Ama programı bellekte değil, kablolarda ve anahtarlarda dururdu. Yeni bir hesap için operatörler panellerdeki kabloları elle yeniden takardı ve bu iş günler sürebilirdi.

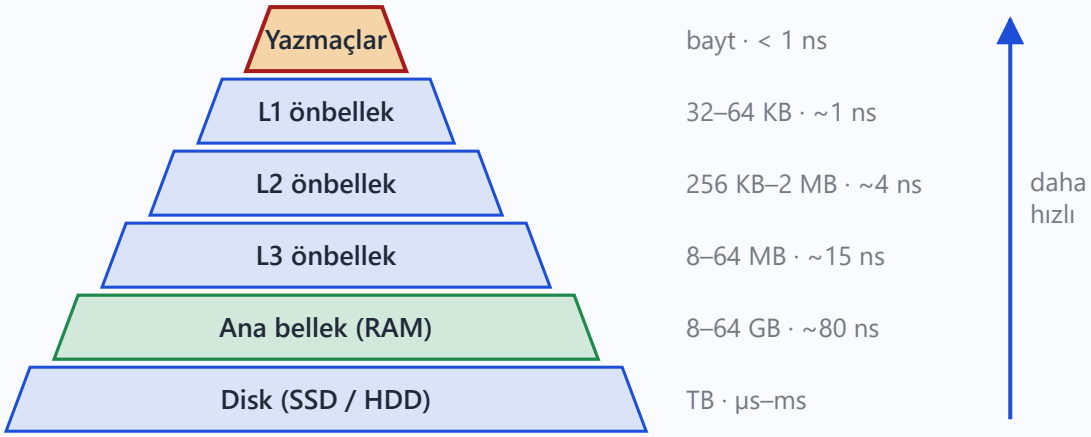
Aynı yıl ENIAC ekibinin üzerinde çalıştığı EDVAC tasarımı farklı bir yol önerdi: komutlar ve veriler bellekte sırayla saklanacaktı. Program değiştirmek, belleğe yeni sayılar yazmak demekti. Bu yaklaşım esnek ve genişletilebilirdi; bir program başka bir programı bile veri olarak okuyup üretebilirdi. Derleyiciler de tam olarak bunu yapar: kaynak kodu okur, makine kodunu üretir. Fikri, John von Neumann'ın 1945 tarihli “First Draft of a Report on the EDVAC” raporu yaydı. Saklanan programı ilk çalıştıran makine ise 1948'de Manchester Üniversitesinde çalışan ve “Baby” diye anılan küçük deneysel bilgisayar oldu.

1.1.7.2 Bellek hiyerarşisi

Bellek, çalışan programların ve verilerin geçici olarak saklandığı yerdir. İdeal bir bellek hem çok büyük, hem çok hızlı, hem de ucuz olurdu. Ne yazık ki bu üç özellik birbiriyle çelişir. Hızlı bellek teknolojileri pahalıdır ve yonga üzerinde çok yer kaplar. Ucuz ve büyük bellekler ise yavaştır. Bilgisayarlar bu çelişkiyi, farklı hız ve boyuttaki bellekleri katmanlar hâlinde dizerek çözer.

Tanım 1.1.8

Bellek hiyerarşisi (*memory hierarchy*), belleklerin işlemciye yakın, hızlı ve küçük düzeylerden uzak, yavaş ve büyük düzeylere doğru dizilmesidir. Sıra şöyledir: yazmaçlar, L1, L2 ve L3 önbellekler, ana bellek (RAM) ve disk.



Şekil 1.1.14: Bellek hiyerarşisi. Yukarı çıktıkça bellek küçülür ve hızlanır. Boyut ve erişim süreleri yaklaşık değerlerdir ve işlemciye göre değişir; 1 ns saniyenin milyarda biridir.

- **L1 önbellek** (*cache*): en hızlı önbellektir ve işlemcinin içindedir. Boyutu çekirdek başına birkaç on kilobayttır ve yaklaşık 1 ns'de erişilir.
- **L2 önbellek**: L1'i destekler ve ondan büyüktür. Çoğu işlemcide bir de bütün çekirdeklerin paylaştığı, birkaç on megabaytlık **L3** önbellek vardır.
- **Ana bellek (RAM)**: gigabaytlarca büyüktür, ama erişimi önbelleklerden onlarca kat yavaştır.
- **Disk**: terabaytlarca büyük ve kalıcıdır; erişim süresi mikrosaniyeler ile milisaniyeler arasındadır.

Hiyerarşinin işe yaramasının nedeni, programların belleğe rastgele değil, belirli bir düzenle erişmesidir. Bir program kısa süre önce kullandığı veriyi yakında yeniden kullanma eğilimindedir; döngü değişkenleri buna örnektir. Bir adrese erişen program, çoğu zaman hemen yanındaki adreslere de erişir; bir dizide baştan sona gezinmek buna örnektir. Bu iki eğilime **yerellik** denir. Önbellek, son kullanılan verileri ve komşularını işlemcinin yanında tutar. Böylece erişimlerin büyük çoğunluğu yavaş RAM'e gitmeden hızlı önbellekte karşılanır. İşlemciye yakın bellek daha hızlı ama daha küçük ve pahalıdır; bu hiyerarşinin nasıl kullanıldığı programın başarımını doğrudan etkiler. Aynı sayıda toplama yapan iki Java programından biri dizide sırayla, öteki rastgele atlayarak gezinirse aralarında birkaç kat hız farkı görülebilir.

1.1.7.3 İşlemci ve veri yolu

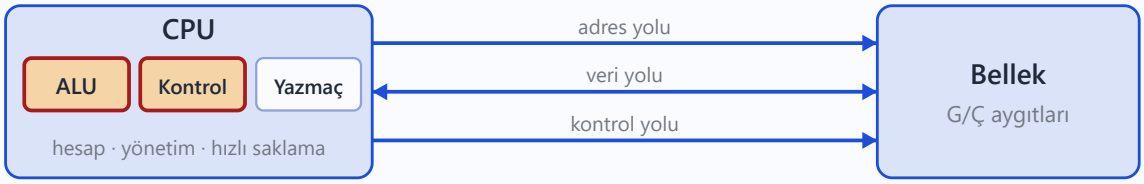
Merkezî işlem birimi (CPU), bilgisayarın bütün hesaplarının yapıldığı yerdir. İçinde üç temel bölüm vardır.

- **Aritmetik mantık birimi** (ALU, *Arithmetic Logic Unit*): toplama, çıkarma, karşılaştırma ve bit düzeyinde mantık işlemleri gibi hesapları yapar. İkili toplamada anlatılan tam toplayıcılar ALU'nun içindedir.
- **Kontrol birimi** (*control unit*): komutları yorumlar, işlemleri sıraya koyar ve yönlendirir. Hangi yazmacın ALU'ya bağlanacağına, belleğe okuma mı yazma mı sinyali gideceğine o karar verir.

- **Yazmaçlar** (*register*): geçici veriler için çok küçük ama çok hızlı saklama alanlarıdır. ALU'nun işlediği sayılar yazmaçlardan gelir ve sonuçlar yazmaçlara yazılır.

İşlemciyi bellek ve giriş/çıkış aygıtlarına bağlayan yapıya **veri yolu** (*bus*) denir. Veri yolu, birçok paralel telden oluşan bir iletişim sistemidir. Teller görevlerine göre üç gruba ayrılır.

- **Adres sinyalleri** (adres yolu): hangi bellek hücresine ya da aygıta erişileceğini seçer.
- **Veri sinyalleri** (veri yolu): işlenecek ya da saklanacak asıl veriyi taşır.
- **Kontrol sinyalleri** (kontrol yolu): işlemi yönlendirir; örneğin işlemin "oku" mu "yaz" mı olduğunu bildirir.

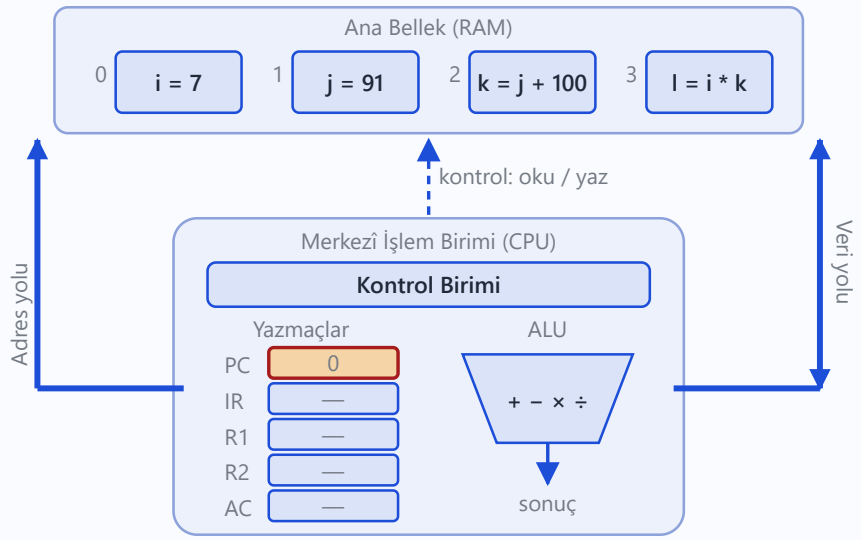


Şekil 1.1.15: İşlemci (ALU, kontrol birimi, yazmaçlar) ile bellek ve G/Ç aygıtları arasındaki adres, veri ve kontrol yolları.

Adres yolundaki tel sayısı, adreslenebilecek bellek miktarını belirler. Her tel bir bit taşır; n telli bir adres yolu 2^n farklı adres seçebilir. 32 hatlı bir adres yolu $2^{32} = 4.294.967.296$ farklı adres seçer. Her adres bir baytı gösteriyorsa bu 4 GiB bellek eder. 32 bitlik sistemlerin en fazla yaklaşık 4 GB RAM kullanabilmesinin nedeni budur. 64 bitlik işlemcilere geçişin başlıca gerekçelerinden biri de bu sınırdır. Teorem 1.1.1 açısından bakılınca bir adres yolunun her teli tek bir bitlik basamak gibidir ve n basamakla 2^n farklı sayı yazılır.

1.1.7.4 Yazmaçlar

İşlemci, bellek ve veri yolunun birlikte nasıl çalıştığını Şekil 1.1.16 özetler. Belleğin her hücresinin bir adresi ve bir içeriği vardır. İşlemcinin içindeki yazmaçlar ise belleğin işlemciye en yakın, en küçük ve en hızlı katmanıdır.



Şekil 1.1.16: Üstte ana bellek (0–3 adreslerinde dört hücre), altta işlemci: kontrol birimi, PC, IR, R1, R2 ve AC yazmaçları ve ALU. Solda adres yolu, sağda çift yönlü veri yolu, ortada kontrol sinyali vardır.

- İşlemci, adres yolundan hangi hücreyi istediğini söyler.
- Hücrenin içeriği veri yolundan işlemciye gelir (okuma) ya da işlemciden hücreye gider (yazma).
- **PC** (*program counter*, program sayacı): sıradaki komutun adresini tutar.
- **IR** (*instruction register*, komut yazmacı): o an çözülen komutu tutar.
- **R1, R2** ve **AC** (*accumulator*, akümülatör): ara sonuçları tutar.
- ALU, yazmaçlardaki sayılarla hesap yapar.

Yazmaçlar elektronikte flip-floplardan kurulur. Bir flip-flop bir biti saklar; 32 flip-flop yan yana dizilince 32 bitlik bir yazmaç elde edilir. Yazmaçlar saat sinyalinin her darbesinde yeni değer alabilir, bu yüzden bir saat çevriminde okunup yazılabilirler. Sayıları azdır: tipik bir işlemcide birkaç düzine genel amaçlı yazmaç bulunur. Derleyicinin önemli işlerinden biri, sık kullanılan değişkenleri bellekte değil yazmaçlarda tutmaktır.

1.1.7.5 Bellek, adresler ve program sayacı

Bellek, veri ve talimatların saklandığı alandır. İşlemci açısından bir program, bellekte art arda dizilmiş uzun bir talimat listesidir. Her bellek hücresinin bir **adresi** vardır ve hücrede bir talimatın ya da bir verinin sayıyla kodlanmış hâli bulunur. Talimatlar bellekten tek tek alınır, işlemcideki yazmaçlara yüklenir ve sırayla çalıştırılır.

Tanım 1.1.9

Program sayacı (PC, *program counter*), sıradaki talimatın adresini tutan yazmaçtır. İşlemci bir talimatı aldıktan sonra PC'yi bir sonraki adrese ilerletir.

İşlemci normalde adres sırasıyla ilerler: 200, 201, 202 … Ama bazı talimatlar bu sırayı değiştirir. Tablo 1.1.13'teki küçük bellek parçasında 204 adresinde bir `JUMP 7` talimatı vardır. Bu örnekte 7, talimatın kendi adresinden ne kadar ileri atlanacağını söyler: *204 + 7 = 211*. Talimat PC'ye 211 yazar ve işlemci 205–210 arasını atlayıp 211'deki talimatla devam eder. Atlama hedefi bazı makinelerde böyle göreli bir uzaklıkla, bazılarında da doğrudan mutlak bir adresle (`JUMP 211` gibi) verilir; iki durumda da sonuç PC'nin değişmesidir.

Adres	İçerik	Tür
200	`ADD to R1`	talimat
201	`102`	veri
202	…	
203	…	
204	`JUMP 7`	talimat (atla)
…	…	
211	sonraki talimat	talimat

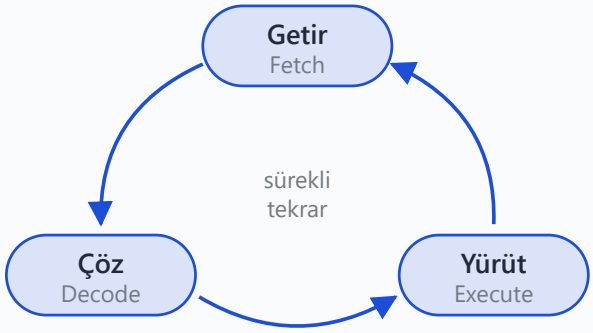
Tablo 1.1.13: Bellekte talimatlar ve veriler yan yana.

Tablonun “Tür” sütunu yalnız okura yardım etmek içindir; bellekte böyle bir sütun yoktur. Bir bellek hücresinin talimat mı veri mi olduğu, hücreye bakarak anlaşılmaz: ikisi de sayıdır. Fark, işlemcinin o hücreyi nasıl kullandığıdır. PC bir hücreyi gösterdiğinde içerik talimat olarak çözülür; bir talimat o hücreyi okuduğunda içerik veri olarak kullanılır. PC yanlışlıkla bir veri hücresini gösterirse işlemci o sayıyı talimat sanır ve anlamsız işler yapar. Bu yüzden atlama hedeflerinin doğru hesaplanması çok önemlidir.

1.1.7.6 Getir–Çöz–Yürüt döngüsü

> **Tanım 1.1.10**
>
> **Getir–Çöz–Yürüt döngüsü** (*fetch–decode–execute cycle*), işlemcinin her komut için tekrarladığı temel çalışma döngüsüdür. Programların işlemcide nasıl yürütüldüğünü tanımlar.

1. **Getir** (*fetch*): işlemci, PC'nin gösterdiği adresteki komutu bellekten alır ve IR'ye yazar. PC bir sonraki adrese ilerler.
2. **Çöz** (*decode*): komut çözülür. Hangi işlem yapılacak, hangi yazmaçlar kullanılacak, hangi adrese erişilecek? Her komutun belirli bir işlevi vardır ve kontrol birimi bu işlevi komutun bitlerinden okur.
3. **Yürüt** (*execute*): komutun gerektirdiği işlem yapılır. Bu bir veri işleme, bir sonuç hesaplama, bellekten okuma ya da belleğe yazma olabilir.



Şekil 1.1.17: Getir, Çöz ve Yürüt adımlarından oluşan döngü. Bilgisayar açık olduğu sürece sürekli tekrarlanır.

Ardından bir sonraki komut alınır ve döngü baştan başlar. Bu döngü bilgisayar açık olduğu sürece hiç durmadan tekrarlanır. Bir Java programının bütün çalışması, milyarlarca kez dönen bu döngüden ibarettir.

Döngüyü bir **saat** sinyali yönetir. 3 GHz'lik bir işlemcinin saati saniyede 3 milyar kez "tık" eder; bir saat çevrimi yaklaşık 0,33 ns sürer. Basit bir işlemcide her komut birkaç saat çevrimi alır. Modern işlemciler ise birden çok komutu üst üste bindirerek işler. Bir komut yürütülürken bir sonraki çözülür, ondan sonraki getirilir. Bu düzene **boru hattı** (*pipeline*) denir ve bir fabrikadaki montaj hattına benzer. Hat dolu olduğunda her saat çevriminde bir komut tamamlanır.

1.1.7.7 Kod izleme: oyuncak bilgisayarda 102 + 25

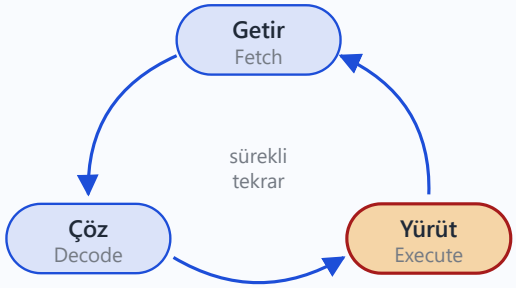
Döngünün işleyişi, oyuncak bir bilgisayarda kolayca izlenir. Bu bilgisayarda belleğin 200–203 adreslerinde dört komut, 205–207 adreslerinde veri vardır. Program, 205'teki sayı ile 206'daki sayıyı toplar ve sonucu 207'ye yazar. Satırlardaki noktalı virgülden sonraki kısımlar açıklamadır ve `[x]` gösterimi "x adresindeki bellek hücresi" demektir.

```
200: LOAD  R1, [205] ; R1 ← bellek[205]
201: ADD   R1, [206] ; R1 ← R1 + bellek[206]
202: STORE R1, [207] ; bellek[207] ← R1
203: HALT            ; dur
204: ...
205: 102             ; veri
206: 25              ; veri
207: 0               ; sonuç buraya
```

Komutların anlamı şöyledir. `LOAD` (yükle) bellekteki bir değeri yazmaca kopyalar. `ADD` (topla) bellekteki bir değeri yazmaçtaki değere ekler. `STORE` (sakla) yazmaçtaki değeri belleğe yazar. `HALT` (dur) işlemciyi durdurur. Başlangıçta PC = 200'dür. Tablo 1.1.14 her adımı gösterir.

Adım	Evre	Ne olur?	PC	IR	R1	[207]
0	Başlangıç	Program ve veriler bellekte.	200	—	—	0
1	Getir	200'deki komut IR'ye alınır, PC ilerler.	201	`LOAD`	—	0
2	Çöz	İşlem LOAD, hedef R1, adres 205.	201	`LOAD`	—	0
3	Yürüt	[205] = 102 okunur, R1'e yazılır.	201	`LOAD`	102	0
4	Getir	201'deki komut alınır.	202	`ADD`	102	0
5	Çöz	İşlem ADD; R1 ile [206] toplanacak.	202	`ADD`	102	0
6	Yürüt	ALU 102 + 25 = 127 hesaplar, sonuç R1'e.	202	`ADD`	127	0
7	Getir	202'deki STORE alınır.	203	`STORE`	127	0
8	Çöz	R1 207 adresine yazılacak (kontrol yolu: “yaz”).	203	`STORE`	127	0
9	Yürüt	[207] ← 127. Sonuç artık bellekte.	203	`STORE`	127	127
10	Getir	203'teki HALT alınır.	204	`HALT`	127	127
11	Çöz + Yürüt	HALT: işlemci durur.	204	`HALT`	127	127

Tablo 1.1.14: Oyuncak bilgisayarda Getir–Çöz–Yürüt adımları.



Şekil 1.1.18: 6. adımda döngünün durumu: ADD komutunun yürütme evresi etkin. ALU 102 + 25 = 127'yi hesaplar.

İzlemeden çıkan üç gözlem önemlidir. Birincisi, PC her Getir adımında hemen ilerler; komut yürütülürken PC zaten bir sonraki komutu gösterir. Program bittiğinde PC 204'tür, çünkü HALT getirildiği anda PC bir kez daha ilerlemiştir. İkincisi, yürütme evresinde veri yolu iki yönde de kullanılır. LOAD ve ADD bellekten okur, STORE belleğe yazar; hangisinin yapılacağını kontrol yolundaki “oku/yaz” sinyali belirler. Üçüncüsü, 204 adresindeki `...` hiç çalışmaz. HALT işlemciyi durdurduğu için PC'nin gösterdiği hücre bir daha getirilmez.

Java'daki `toplam = a + b;` satırı da sonunda bu tür komutlara dönüşür. Derleyici `a`'yı ve `b`'yi bellekten ya da yazmaçlardan alan, toplayan ve sonucu `toplam`'ın yerine yazan birkaç makine komutu üretir. Gerçek işlemcilerin komut kümeleri yüzlerce komut içerir ama her biri aynı Getir–Çöz–Yürüt döngüsüyle çalışır.

Örnek 1.1.10

Aynı oyuncak bilgisayarda 200 adresinden başlayan şu program çalışsın: 200: `LOAD R1, [206]` ; 201: `JUMP 205` ; 202–204: başka komutlar; 205: `HALT` ; 206: 40. Program bitince R1 ve PC kaçtır?

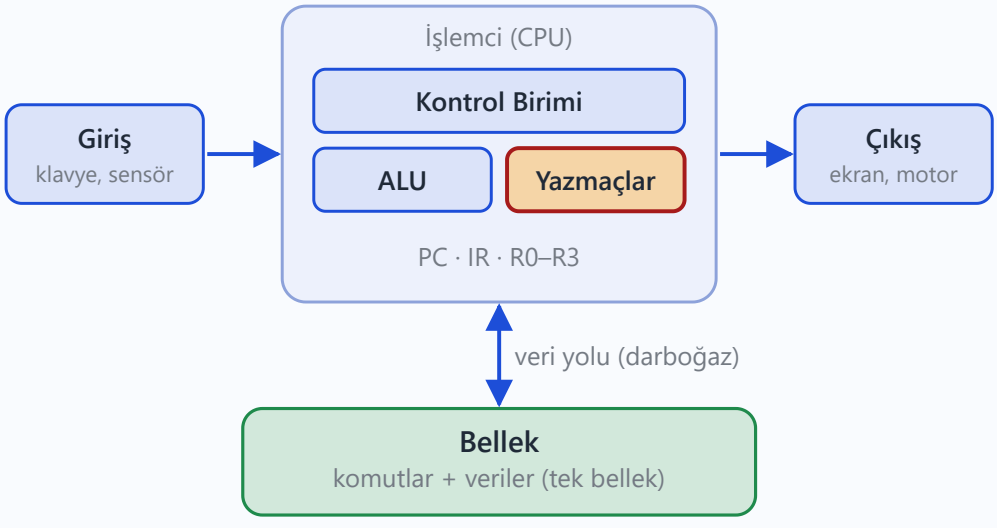
Getir 200: PC 201 olur; yürütmede [206] = 40 R1'e yazılır. Getir 201: PC önce 202 olur; yürütmede JUMP, PC'ye 205 yazar. Bu atlama mutlak adreslidir. Getir 205: PC 206 olur; HALT işlemciyi durdurur. Sonuçta R1 = 40 ve PC = 206'dır. 202–204 arasındaki komutlar hiç getirilmemiştir.

1.1.8 von Neumann mimarisi

1.1.8.1 Mimari ve darboğaz

Tanım 1.1.11

von Neumann mimarisi (*von Neumann architecture*), saklanan program fikrinin donanım planıdır: işlemci, bellek, giriş/çıkış birimleri ve bunları bağlayan veri yolu. Komutlar ve veriler aynı bellekte durur ve aynı yoldan taşınır.



Şekil 1.1.19: von Neumann mimarisi: giriş, işlemci (kontrol birimi, ALU, yazmaçlar), çıkış ve komutlarla verilerin birlikte durduğu bellek. İşlemci ile bellek arasında tek bir yol vardır.

Mimaride her komut bir **işlem kodu** (*opcode*) ile temsil edilir. İşlem kodu, komutun bitleri arasında hangi işlemin yapılacağını belirten kısımdır: toplama mı, yükleme mi, atlama mı? Komutun geri kalan bitleri işlenenleri, yani hangi yazmaçların ya da hangi adresin kullanılacağını söyler.

Bu tasarımın bir bedeli vardır. İşlemci ile bellek arasındaki tek yol, hem komutları hem verileri taşımak zorundadır. İşlemci bellekten çok daha hızlı çalıştığı için çoğu zaman bu yoldan gelecek bir komutu ya da veriyi bekler. Bu sınırlamaya **von Neumann darboğazı** denir. Önbellekler bu yüzden vardır: sık kullanılan komut ve verileri işlemcinin yanında tutarak yavaş yola gitme sayısını azaltırlar.

Karşılaştırma için **Harvard mimarisinde** komut belleği ile veri belleği ayrıdır ve her birinin kendi yolu vardır. İşlemci aynı anda hem bir komut getirebilir hem bir veri okuyabilir. Arduino kartlarındaki AVR gibi pek çok mikrodenetleyici bu yapıyı kullanır: program flaş bellekte, veriler SRAM'de durur. Modern masaüstü işlemcileri ise bir karma kullanır. Ana bellek tektir (von Neumann), ama işlemcinin içindeki L1 önbellek komutlar ve veriler için ikiye ayrılmıştır (Harvard).

1.1.8.2 Komutların kodlanması

Komutların sayı olması ne demektir? Yine oyuncak bir bilgisayar üzerinde düşünelim: 8 bitlik, 4 yazmaçlı (R0–R3), 256 adet 8 bitlik bellek hücresi olan ve yalnız 4 komut bilen bir makine. Komutlar şunlardır: toplama (00), çıkarma (01), çarpma (10) ve bölme (11). Her komut 8 bitlik bir sayıdır ve bitler dört alana bölünür.

- İlk 2 bit **işlem kodudur**. 4 komutu ayırt etmek için 2 bit yeter, çünkü $2^2 = 4$.
- Sonraki 2 bit **hedef yazmaçtır** (*destination register*): sonucun yazılacağı yazmaç. 4 yazmaç için 2 bit gerekir.
- Son 4 bit, iki **işlenen yazmaçtır**; her biri 2 bittir.

"R0 ile R2'nin içeriğini topla, sonucu R3'e yaz" komutu alanlarıyla *0 3 0 2* diye yazılır: işlem kodu 0 (toplama), hedef 3, işlenenler 0 ve 2. Her alan 2 bitle ikiliye çevrilip yan yana konunca `00 11 00 10` = `00110010` elde edilir.



Şekil 1.1.20: 8 bitlik komut 00110010'un alanları: işlem kodu 00 (toplama), hedef 11 (R3), birinci işlenen 00 (R0), ikinci işlenen 10 (R2).

İşlemcinin "çöz" adımı, bu bit alanlarını sayıdan geri çıkarmaktır. Aynı işlem Java'da basamak değeriyle bölme ve kalan alma ile yapılabilir.

```
int komut = 0b00110010;
int islemKodu = komut / 64;     // ilk 2 bit
int hedef = (komut / 16) % 4;   // sonraki 2 bit
int kaynak1 = (komut / 4) % 4;  // sonraki 2 bit
int kaynak2 = komut % 4;        // son 2 bit
System.out.println(komut);
System.out.println(islemKodu + " " + hedef
    + " " + kaynak1 + " " + kaynak2);
```

```
50
0 3 0 2
```

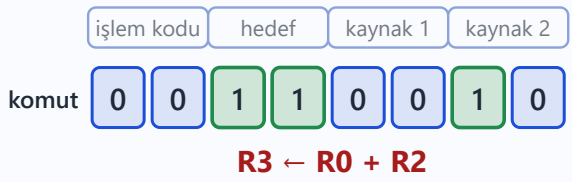
Kodun mantığı ondalıktaki bir benzetmeyle kolayca anlaşılır. 4728 sayısının yüzler basamağını bulmak için önce 100'e bölünür (47), sonra 10'a göre kalan alınır (7). İkilide de aynıdır. 2^k'ye bölmek sağdaki k biti atar; 4'e göre kalan almak ise son 2 biti verir, çünkü $4 = 2^2$. Her alan bu iki işlemle sırası gelince ayıklanır.

1.1.8.3 Kod izleme: komut çözme

Tablo 1.1.15, kodun her satırını ikili gösterimle birlikte izler.

Satır	Ara değer	Sonuç	Açıklama
`komut = 0b00110010`	—	50	*32 + 16 + 2 = 50*
`islemKodu = komut / 64`	—	0	*64 = 2^6*; sağdaki 6 bit atılır, geriye `00` kalır: toplama.
`hedef = (komut / 16) % 4`	*50 / 16 = 3* (`0011`)	3	Sağdaki 4 bit atılır; % 4 son 2 biti alır: `11` = R3.
`kaynak1 = (komut / 4) % 4`	*50 / 4 = 12* (`001100`)	0	Sağdaki 2 bit atılır; *12 % 4 = 0*: R0.
`kaynak2 = komut % 4`	—	2	Son 2 bit doğrudan alınır: `10` = R2.

Tablo 1.1.15: `komut = 0b00110010` için alanların çözülmesi.



Şekil 1.1.21: Çözülen alanlar: 0 3 0 2. 50 sayısı işlemci için "R3 ← R0 + R2" komutudur.

Program önce `50`, sonra `0 3 0 2` yazar. 50 sayısı, bir `int` olarak bakılınca sıradan bir tam sayıdır; bu oyuncak işlemci için ise "R3 ← R0 + R2" komutudur. Sayının anlamı, onu kimin nasıl kullandığına bağlıdır.

Gerçek Java kodunda bit alanları çoğu zaman bölme yerine **bit düzeyinde işleçlerle** ayıklanır. `komut >> 4` sayıyı 4 bit sağa kaydırır ve pozitif sayılarda `komut / 16` ile aynı sonucu verir. `& 3` ise son iki biti maskeleyerek alır, yani `% 4`'ün işini görür. Hedef alanı böylece `(komut >> 4) & 3` diye yazılır. Bu yazılış, donanımın gerçekte yaptığı işe daha yakındır: kontrol birimi bölme yapmaz, yalnız ilgili telleri seçer.

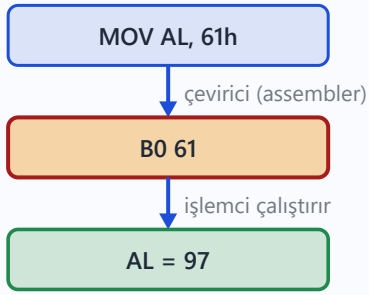
Örnek 1.1.11

Oyuncak bilgisayarda `10011100` komutu ne yapar?

Bitler 2'şerli alanlara ayrılır: `10 01 11 00`. İşlem kodu 10 çarpmadır. Hedef 01, yani R1'dir. İşlenenler 11 (R3) ve 00 (R0)'dır. Komut **R1 ← R3 × R0** anlamına gelir. Java koduyla sağlamak için komutun sayı değeri *128 + 16 + 8 + 4 = 156* alınır: *156 / 64 = 2*; *156 / 16 = 9* ve *9 % 4 = 1*; *156 / 4 = 39* ve *39 % 4 = 3*; *156 % 4 = 0*. Alanlar 2 1 3 0'dır.

1.1.8.4 Assembly dili

Komutları `00110010` gibi bit dizileriyle yazmak insan için çok zordur. **Assembly dili** (*assembly language*), her makine komutuna `ADD`, `MOV` gibi akılda kalan kısaltmalar verir. Assembly ile yazılmış metni makine koduna **çevirici** (*assembler*) adlı bir program çevirir.



Şekil 1.1.22: `MOV AL, 61h` komutu çevirici ile B0 61 makine koduna çevrilir; işlemci çalıştırınca AL yazmacına 97 yazılır.

- Assembly dili belirli bir işlemci mimarisine özeldir. x86 için yazılan kod ARM işlemcide çalışmaz, çünkü iki mimarinin komutları ve kodlamaları farklıdır.
- Her satır genellikle tek bir makine komutuna karşılık gelir. Java'nın tek bir satırı ise çoğu zaman birçok makine komutuna dönüşür.
- Bilgisayarın alt düzeyde nasıl çalıştığını anlamak için önemlidir.
- Başarım iyileştirmesinde ve donanımla doğrudan etkileşimde kullanılır: aygıt sürücüleri, gömülü sistemler, işletim sistemi çekirdeğinin bazı parçaları.

x86 komutu	Anlamı	Makine kodu
`MOV AL, 61h`	AL yazmacına 61h (= 97) değerini taşı	`B0 61`
`MOV AX, BX`	BX'teki değeri AX'e kopyala	`66 89 D8`
`ADD EAX, 10`	EAX'e 10 ekle	`83 C0 0A`
`XOR EAX, EAX`	EAX'i sıfırla (x XOR x = 0)	`31 C0`

Tablo 1.1.16: Bazı x86 komutları ve makine kodları (onaltılı).

Tablo 1.1.16'daki kodlar, oyuncak bilgisayardaki fikrin gerçek bir işlemcideki karşılığıdır. `B0 61` kodunda B0, "AL yazmacına bir sabit yükle" işlem kodudur; 61 ise yüklenecek sabittir. `ADD EAX, 10` komutunda son bayt 0A = 10'dur. x86 komutları oyuncak bilgisayardaki gibi sabit 8 bit değildir; 1 ile 15 bayt arasında değişir. `MOV AX, BX` kodunun başındaki 66 baytı, komutun 32 değil 16 bitlik yazmaçlarla çalışacağını bildiren bir ön ektir. `XOR EAX, EAX` bir yazmacı sıfırlamanın yaygın bir yoludur. Bir sayının kendisiyle XOR'u her bitte 0 verir. Üstelik bu komut, `MOV EAX, 0`'dan daha kısa kodlanır.

İlk satırdaki 61h, onaltılı 61 demektir (assembly'de `h` eki, Java'da `0x` ön eki). 61h = 0x61 = 97'dir ve 97 aynı zamanda ASCII `'a'`'dır. Bellekteki bir bayt sayı mı, harf mi, komut mu? Bunu baytın kendisi söylemez; programın onu nasıl kullandığı belirler. Bu cümle, notun bütün katmanlarının ortak sonucudur.

Java'nın bu katmanlar içindeki yeri de buradan anlaşılır. Programcı yüksek düzeyli bir dilde, Java'da yazar. `javac` bu kodu bytecode'a çevirir. Bytecode, gerçek bir işlemcinin değil, JVM'in komut kümesidir ve her işlemcide aynıdır. JVM çalışırken bytecode'u o makinenin assembly düzeyindeki makine komutlarına çevirir. Java'nın "bir kez yaz, her yerde çalıştır" ilkesi, bu ara katman sayesinde mümkündür.

1.1.9 Sık yapılan hatalar

Bu notta anlatılan konuların Java'daki en yaygın sonuçları, derleyicinin reddettiği satırlar ya da derlenip yanlış çalışan programlar olarak ortaya çıkar. Aşağıdaki listede her hata, derleyici iletisiyle ya da programın gerçek çıktısıyla birlikte verilmiştir. İletiler JDK 21 derleyicisine aittir.

Hata	Kod	Sonuç	Neden ve doğrusu
Aralık dışı sabit	`byte b = 200;`	`error: incompatible types: possible lossy conversion from int to byte`	`byte` en fazla 127'dir. 200 bir `int` sabitidir ve sığmaz; gerçekten isteniyorsa `(byte) 200` yazılır ve değer −56 olur.
`L` ekini unutmak	`long x = 3000000000;`	`error: integer number too large`	Eksiz tam sayı sabiti `int` sayılır ve `int` 2.147.483.647'yi aşamaz. Doğrusu `3000000000L`'dir.
Karakter ile dizgiyi karıştırmak	`char c = "A";`	`error: incompatible types: String cannot be converted to char`	Çift tırnak bir `String` üretir. Karakter tek tırnakla yazılır: `'A'`.

Hata	Kod	Sonuç	Neden ve doğrusu
Başta sıfır	`int x = 010;` `int y = 08;`	`x` = 8 `error: ';' expected`	Başta 0 sekizli demektir; 010 = 8'dir. Sekizli tabanda 8 rakamı olmadığı için ikinci satır derlenmez.
Taşmayı fark etmemek	`Integer.MAX_VALUE + 1`	`-2147483648`	Tam sayı taşması hata vermez, başa sarar. Büyük çarpımlarda `long` gerekir ve çarpım `long` olarak başlamalıdır: `1920L * 1080`. Denetim için `Math.addExact`.
Kesirlerde `==`	`0.1 + 0.2 == 0.3`	`false`	Kayan nokta değerleri yaklaşıktır. Farkın küçüklüğüne bakılır: `Math.abs(a - b) < 1e-9`.
`length()`'i karakter sayısı sanmak	`"😀".length()`	`2`	`length()` UTF-16 birimi sayar. Kod noktası için `codePointCount`, bayt için `getBytes(...).length`.
`char` toplamını dizgi sanmak	`System.out.println('a' + 'b');`	`195`	İki `char`'ın toplamı `int`'tir. Birleştirmek için `"" + 'a' + 'b'` ya da `String.valueOf`.

Tablo 1.1.17: Sık yapılan hatalar ve nedenleri.

Listedeki ilk dört satırın hatalarını, `010` dışında, derleyici yakalar ve program hiç çalışmaz. Bunlar can sıkıcıdır ama zararsızdır, çünkü hata satırı ve nedeni hemen görülür. `010` ile son dört satırdaki kodlar ise derlenir ve sessizce yanlış sonuç üretir. Bu yüzden daha tehlikelidir. Taşan bir sayaç, kayan noktalı bir eşitlik ya da yanlış hesaplanan bir dizgi uzunluğu ancak program başka bir yerde beklenmedik davranınca fark edilir. Bu hataların ortak kaynağı, bir değerin bellekte nasıl temsil edildiğini unutmaktır.

1.1.10 Özet

- **Parçalar.** CPU hesap yapar ve ALU, kontrol birimi ve yazmaçlardan oluşur. RAM geçici, disk kalıcı bellektir. Veri yolu adres, veri ve kontrol sinyallerini taşır; n hatlı adres yolu 2^n adres seçer.
- **İkili sayılar.** Bit 0 ya da 1'dir; 8 bit 1 bayttır. d basamak ve b tabanla 0 … $b^d - 1$ yazılır. Ondalıktan ikiliye çevirmek için sayı 2'ye bölünür ve kalanlar aşağıdan yukarı okunur. Onaltılı bir basamak 4, sekizli bir basamak 3 bittir.
- **Java'da tabanlar.** `0b` ikili, `0` sekizli, `0x` onaltılı sabittir. `Integer.toBinaryString`, `toHexString` ve `parseInt(s, taban)` çeviri yapar.
- **Negatif sayılar.** İkiye tümleyen: bitler ters çevrilir ve 1 eklenir. En soldaki bitin ağırlığı negatiftir. 8 bitle aralık −128 … 127'dir. Aralık aşılırsa Java'da sessiz taşma olur ve değer başa sarar.
- **Kesirli sayılar.** Kesir × taban$^{\text{üs}}$ biçiminde saklanır. IEEE 754'e göre `float` 32, `double` 64 bittir. Değerler yaklaşıktır: `0.1 + 0.2 != 0.3`.
- **Metin.** ASCII 7 bit ve 128 karakterdir. Unicode her karaktere bir kod noktası verir (U+0000–U+10FFFF). UTF-8 bir kod noktasını 1–4 baytla yazar ve ASCII ile uyumludur. Java'nın `char`'ı 16 bitlik bir UTF-16 birimidir; `length()` `char` sayısını verir.
- **Yapılandırılmış bilgi.** Resim RGB piksel dizisi, ses basınç örnekleri, film kare dizisidir. Sıkıştırmasız bir dakikalık Full HD video yaklaşık 11 GB tutar.

- **Saklanan program.** Komutlar da bellekte sayı olarak durur. PC sıradaki adresi tutar. İşlemci Getir–Çöz–Yürüt döngüsünü sürekli tekrarlar; atlama komutları sırayı değiştirir.
- **von Neumann mimarisi.** Komut ve veri aynı bellekte ve aynı yolda durur; tek yol darboğaz yaratır, önbellek bunu hafifletir. Harvard mimarisinde komut ve veri belleği ayrıdır.
- **Düzeyler.** Yüksek düzeyli dil → assembly → makine kodu (bitler). Assembly işlemciye özeldir ve her satırı genellikle tek bir komuttur. Bir baytın sayı mı, harf mi, komut mu olduğunu onu kullanan program belirler.

1.1.11 Kavram listesi

işlemci *CPU* · bellek *RAM* · bit *bit* · bayt *byte* · ikilik sayı sistemi *binary number system* · basamak değeri *positional value* · onaltılı taban *hexadecimal* · elde *carry* · ikiye tümleyen *two's complement* · taşma *overflow* · kayan noktalı sayı *floating-point number* · karakter kodlaması *character encoding* · ASCII *ASCII* · Unicode *Unicode* · kod noktası *code point* · UTF-8 *UTF-8* · vekil çift *surrogate pair* · depolanan programlı bilgisayar *stored-program computer* · bellek hiyerarşisi *memory hierarchy* · önbellek *cache* · veri yolu *bus* · yazmaç *register* · program sayacı *program counter* · getir–çöz–yürüt döngüsü *fetch–decode–execute cycle* · von Neumann mimarisi *von Neumann architecture* · işlem kodu *opcode* · assembly dili *assembly language*

1.1.12 Kendini Dene

1. 100 sayısı 8 bitlik ikili ve onaltılı olarak nasıl yazılır?

   Cevap

   100'e sığan kuvvetler 64 ve 32'dir; kalan 4. *100 = 64 + 32 + 4*, yani `01100100`. Dörderli gruplar `0110 0100` = 6 ve 4 olduğu için onaltılı değer `0x64`'tür.

2. `11010110` bit dizisi işaretsiz olarak ve 8 bitlik ikiye tümleyenle kaçtır?

   Cevap

   İşaretsiz: *128 + 64 + 16 + 4 + 2 = 214*. İşaretli: en soldaki bitin ağırlığı −128 alınır, *−128 + 64 + 16 + 4 + 2 = −42*; ya da *214 − 256 = −42*.

3. 8 bitlik ikiye tümleyende −1 ve −128 nasıl yazılır? `byte m = -128;` için `(byte) (-m)` neden yine

−128 verir?

Cevap

−1 = `11111111` (1 = `00000001`, ters `11111110`, +1). −128 = `10000000`. 128 sayısı `byte` aralığında yoktur. −128'in bitleri ters çevrilince `01111111`, 1 eklenince yine `10000000` çıkar. Java'da `-m` önce `int` olarak 128 hesaplar, ama `byte`'a dönüşüm yalnız son 8 biti alır ve sonuç −128 olur.

4. Bu satırlar ne yazdırır? `System.out.println((byte) 130);`, `System.out.println((char) ('A' + 2));`, `System.out.println('A' + 2);`

Cevap

−126, `C` ve 67. 130 = `10000010` işaretli okununca *130 − 256 = −126*'dır. `'A' + 2` bir `int`'tir (67); `(char)` dönüşümüyle 67 numaralı karakter `C` yazılır, dönüşüm olmadan sayının kendisi yazılır.

5. `"Işık"` dizgisi için `length()` ve UTF-8 bayt sayısı nedir?

Cevap

`length()` 4'tür; dört karakterin hepsi U+FFFF'in altındadır. UTF-8'de I (U+0049) 1 bayt, ş (U+015F) 2 bayt, ı (U+0131) 2 bayt, k 1 bayttır: *1 + 2 + 2 + 1 = 6*.

6. `System.out.println(0x2A + 012 + 0b11);` ne yazdırır?

Cevap

55. `0x2A` = *2·16 + 10 = 42*, `012` sekizli olduğu için *1·8 + 2 = 10*, `0b11` = 3. Toplam *42 + 10 + 3 = 55*'tir; sonuç ondalık yazılır.

7. Oyuncak bilgisayarda (2 bit işlem kodu: 00 topla, 01 çıkar, 10 çarp, 11 böl; 2 bit hedef; 2'şer bit iki işlenen) "R1 ← R2 − R3" komutu hangi 8 bitlik sayıdır? Notun Java koduyla çözülünce hangi alanlar çıkar?

Cevap

`01 01 10 11` = `01011011` = 91. Çözme: *91 / 64 = 1*; *91 / 16 = 5*, *5 % 4 = 1*; *91 / 4 = 22*, *22 % 4 = 2*; *91 % 4 = 3*. Alanlar 1 1 2 3'tür.

8. Bir işlemcinin adres yolu 20 hatlıdır ve her adres bir bayt seçer. En fazla ne kadar bellek adresle-

nebilir?

> **Cevap**
>
> $2^{20} = 1.048.576$ bayt, yani 1 MiB. Intel 8086 işlemcisinin adres yolu tam olarak bu kadardı.

9. Telefon kalitesinde ses saniyede 8000 örnekle, örnek başına 1 bayt ve tek kanalla (mono) kaydediliyor. Sıkıştırmasız bir dakikalık konuşma kaç bayttır? Hesap `int` ile yapılırsa taşma olur mu?

> **Cevap**
>
> $8000 \cdot 1 \cdot 1 \cdot 60 = 480.000$ bayt. Bu değer `Integer.MAX_VALUE` 'dan çok küçüktür, taşma olmaz. CD sesi için aynı hesap 10.584.000 bayt verir; o da `int` 'e sığar. Taşma, video gibi milyarları aşan çarpımlarda başlar.

10. Java'da `0.1f + 0.2f == 0.3f` ifadesi `true` verir. Bu, `float` ile eşitlik karşılaştırmasının güvenli olduğunu mu gösterir?

> **Cevap**
>
> Hayır. `float` 'ın daha kaba yuvarlaması bu üç değerde hataları rastlantı eseri aynı sayıya götürmüştür. Aynı toplam `double` ile `false` verir ve başka sayılarda `float` da yanılır. Kayan noktalı değerler her zaman bir toleransla karşılaştırılmalıdır.

1.1.13 İleri okuma

- C. Petzold, *Code: The Hidden Language of Computer Hardware and Software* (2. baskı, Microsoft Press, 2022): röle ve mantık kapılarından başlayıp toplayıcıyı, belleği ve basit bir işlemciyi adım adım kurar; bu notun katmanlı anlatımının en akıcı uzun hâlidir.
- R. E. Bryant ve D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective* (3. baskı, Pearson, 2016), 2. bölüm: işaretsiz ve ikiye tümleyen tam sayıları, taşmayı, bit düzeyinde işlemleri ve IEEE 754 kayan noktayı programcı gözüyle ve çok sayıda alıştırmayla işler.
- D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", *ACM Computing Surveys* 23(1), 1991: yuvarlama hatasının nereden geldiğini ve IEEE 754 standardının tasarım gerekçelerini ayrıntılı anlatan klasik makale.
- D. A. Patterson ve J. L. Hennessy, *Computer Organization and Design* (RISC-V sürümü, 2. baskı, 2020), 2. ve 5. bölümler: komutların kodlanmasını gerçek bir komut kümesi üzerinde ve bellek hiyerarşisini önbellek ayrıntılarıyla anlatır.

1.1.14 Kaynaklar

- D. A. Patterson ve J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, RISC-V sürümü, 2. baskı, Morgan Kaufmann, 2020.
- C. Petzold, *Code: The Hidden Language of Computer Hardware and Software*, 2. baskı, Microsoft Press, 2022.
- A. B. Downey ve C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019 (Bölüm 1).
- Oracle, *The Java Tutorials*: “Primitive Data Types” ve “Characters”.
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21, §3.10 (Sabitler) ve §4.2 (İlkel tipler).
- The Unicode Consortium, *The Unicode Standard*, unicode.org; IEEE 754-2019, *Standard for Floating-Point Arithmetic*.
- J. von Neumann, *First Draft of a Report on the EDVAC*, 1945.
- R. E. Bryant ve D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3. baskı, Pearson, 2016.
- D. Goldberg, “What Every Computer Scientist Should Know About Floating-Point Arithmetic”, *ACM Computing Surveys*, 23(1), 5–48, 1991.

KISIM I · BÖLÜM 1.2

Hesaplama ve Hesaplamalı Düşünme

Bir hesap makinesi yalnız toplar ve çarpar. Bir dizüstü bilgisayar ise aynı gün müzik çalar, devre benzetimi yapar, Java derler. Aradaki fark donanımın gücünden çok, makineye verilen talimatların değiştirilebilmesinden gelir. Bu not, bilgisayarın ne yaptığı sorusunu temelden ele alır: hesaplama nedir, bir problem bilgisayarın çözebileceği biçime nasıl getirilir ve bu yolun sınırı neredediir? İlk kesimler programlama, hesaplama ve bilgisayar bilimi kavramlarını tanımlar. Ardından hesaplamalı düşünmenin dört becerisi, yani ayrıştırma, örüntü tanıma, soyutlama ve algoritma tasarımı tek bir örnek üzerinde işlenir. Tarih kesimi delikli kartlardan depolanan programlı bilgisayara uzanır. Turing makinesi hesaplamanın kesin tanımını verir ve her problemin çözülebilir olmadığını gösterir. Son kesimler bildirimsel ve buyurusal bilgiyi karekök ve EBOB örnekleriyle ayırır; algoritmayı, problem belirlemeyi ve sıralı, koşullu, döngülü işlemleri Java kodlarıyla anlatır.

ÖĞRENME ÇIKTILARI

Temel kavramlar
Programlama, hesaplama ve bilgisayar bilimini tanımlamak; bilgisayar biliminin teknolojiden ibaret olmadığını açıklamak.

Hesaplamalı düşünme
Ayrıştırma, örüntü tanıma, soyutlama ve algoritma tasarımı becerilerini bir probleme sırayla uygulamak.

Tarihsel gelişim
Sabit ve depolanan programlı bilgisayarı ayırt etmek; Babbage, Lovelace, ABC, Colossus, ENIAC, EDVAC ve von Neumann mimarisinin yerini açıklamak.

Turing makinesi
Bant, bant başlığı ve kural tablosuyla basit bir Turing makinesini adım adım çalıştırmak; hesaplanabilirlik ve durma problemi kavramlarını açıklamak.

Bilgi türleri
Bildirimsel (ne?) ve buyurusal (nasıl?) bilgiyi karekök ve EBOB örnekleriyle ayırt etmek; Babil yöntemini ve Öklid algoritmasını Java'da izlemek.

Algoritma ve işlemler
Algoritmanın özelliklerini saymak, bir problemin girdisini ve çıktısını belirlemek; sıralı, koşullu ve döngülü işlemleri tanımak, `repeat … until` ile `do … while` arasındaki ilişkiyi açıklamak.

1.2.1 Programlama ve hesaplama

Bilgisayar çok hızlı çalışan ama hiçbir şeyi kendiliğinden bilmeyen bir yardımcıdır. Saniyede milyarlarca işlem yapar, yine de hangi işlemi hangi sırayla yapacağına kendisi karar veremez. Bu kararı ona yazılı bir talimat listesi verir. O listeye **program**, listeyi yazma işine **programlama** denir. Programın yaptırdığı iş ise çoğu zaman bir **hesaplamadır**: veri alınır, işlenir ve bir sonuç üretilir. Bu kesim iki kavramı ayrı ayrı tanımlar ve küçük bir Java programında ikisinin nasıl birleştiğini gösterir.

1.2.1.1 Programlama

Tanım 1.2.1

Programlama (*programming*), bir bilgisayara yapacağı işi adım adım bildiren talimatları yazma etkinliğidir. Bu talimatların oluşturduğu metne **program** denir. Talimatlar bir **programlama dili** ile yazılır; Java, Python ve C bu dillere örnektir.

Programlama dili, insan ile makine arasında bir köprüdür. İnsan için okunaklı, makine için de kesin olmalıdır. Günlük konuşmada “biraz bekle” ya da “uygun bir sayı seç” demek yeterlidir; dinleyen kişi boşlukları kendi yorumuyla doldurur. Bilgisayar böyle bir yorum yapmaz. Ne kadar bekleyeceği, hangi sayıyı seçeceği açıkça yazılmamışsa ya hata verir ya da yanlış bir şey yapar. Bu yüzden programlama dillerinin kuralları sıkıdır: bir noktalı virgül eksikse Java programı derlenmez.

Programlanan bir bilgisayar karmaşık görevleri yerine getirebilir. Akıllı telefon, tablet, dizüstü bilgisayar, çamaşır makinesi ve otomobil, içlerinde çalışan programlar sayesinde iş görür. İnternet üzerinden alışveriş, haber, iletişim ve eğitim de programların ürünüdür. Bilgisayarlar evden iş yerine, eğlenceden eğitime her alana girmiştir.

Programlama öğrenmenin bir yararı da düşünme biçimine yansır. Bir işi bilgisayara yaptırabilmek için önce o işi tam olarak anlamak gerekir. “Ortalama al” gibi kısa bir isteğin arkasında kaç sayı olduğu, sayıların nereden geldiği ve sonucun nasıl yazılacağı gibi sorular yatar. Bu soruları sormayı ve yanıtlamayı öğrenen kişi, problem çözmede de daha düzenli düşünür.

1.2.1.2 Programlamanın rolü

Programlama, bilgisayarların istediğimiz görevleri yerine getirmesi için kullanılan bir araçtır. Programcılar bilgisayarları hayatın her alanında daha kullanışlı ve verimli hâle getirir. Üç alan bu rolü iyi örnekler:

- **Otomasyon:** Akıllı termostatlar sıcaklığı ölçüp ısıtıcıyı açar ya da kapatır. Aydınlatma ve güvenlik sistemleri belirli koşullarda kendiliğinden devreye girer. Fabrikalardaki robotlar aynı kaynağı ya da montajı binlerce kez aynı doğrulukla yapar.
- **Sağlık:** Kalp pili kalbin ritmini sürekli ölçer ve gerektiğinde uyarı verir. MR cihazı ölçülen sinyallerden görüntü üretir; bu dönüşüm baştan sona bir hesaplamadır. Hastane bilgi sistemleri ve randevu sistemleri de programlarla çalışır.
- **Eğitim:** Çevrim içi ders platformları, öğretim yazılımları, sınav ve not sistemleri programlanmış bilgisayarların ürünüdür.

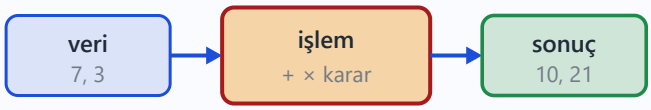
Elektrik-elektronik mühendisliği bu alanların hepsine dokunur. Bir mikrodenetleyiciye yüklenen gömülü yazılım, bir sensörden gelen sinyali süzen sayısal filtre ya da bir motorun hızını ayarlayan kontrol döngüsü programlamayla yazılır. Devre tasarımı ile yazılım arasındaki sınır, bu tür sistemlerde giderek incelir.

1.2.1.3 Hesaplama

Tanım 1.2.2

Hesaplama (*computation*), bir bilgisayarın ya da hesaplama cihazının bir görevi yerine getirmek için verileri işleme sürecidir. Hesaplama girdi olarak veri alır, bu veriyi matematiksel işlemler ve mantıksal kararlarla işler ve bir sonuç üretir.

Hesaplama bilgisayarların temel işlevidir. Ekranda bir harfin çizilmesi bile hesaplamadır: harfin biçimi, boyutu ve konumu sayılara çevrilir, bu sayılardan hangi piksellerin hangi renge boyanacağı hesaplanır. Hesaplama üç tür iş içerir. **Matematiksel işlemler** toplama ve çarpma gibi işlemlerdir. **Mantıksal kararlar** bir koşulun doğru olup olmadığına bakar. **Veri işleme** ise veriyi sıralamak, aramak ve biçimlendirmek gibi genel işleri kapsar.



Şekil 1.2.1: Hesaplamanın üç aşaması. Veri (7 ve 3) alınır, toplama, çarpma ve karar gibi işlemlerden geçer, sonuç (10 ve 21) üretilir.

Şekil 1.2.1'de görülen düzen bütün hesaplamalarda aynıdır. Değişen, verinin türü ve işlemin karmaşıklığıdır. Birkaç örnek:

- **Veri analizi:** Bir sıcaklık sensöründen bir gün boyunca alınan ölçümlerden ortalama, en büyük değer ve eğilim hesaplanır; sonuç bir grafik olarak çizilir.

- **Benzetim (simülasyon):** Bir devrenin ya da hava durumunun matematiksel modeli kurulur. Bilgisayar modeli küçük zaman adımlarıyla ilerletir ve sistemin nasıl davranacağını öngörür.
- **Oyun geliştirme:** Her karede nesnelerin konumları fizik kurallarıyla güncellenir, çarpışmalar denetlenir, bilgisayarın yönettiği karakterler karar verir.

Programlama ile hesaplama arasındaki ilişki şöyle özetlenebilir: hesaplama yapılan iştir, programlama o işin tarifini yazmaktır. İyi yazılmış bir program, aynı hesaplamayı daha az adımla, daha az bellekle ve daha az hatayla yaptırır. Bu yüzden programlama hesaplamayı daha etkili ve verimli hâle getirir.

1.2.1.4 Hesaplamanın yapı taşları

Hesaplamanın iki temel yapı taşı vardır. Birincisi **matematiktir**: toplama, çıkarma, çarpma ve bölme gibi işlemler. İkincisi **mantıktır**: kararlar, mantıksal ifadelerin değerlendirilmesiyle alınır. “Eğer hava güneşliyse dışarı çık” cümlesi bir mantık işlemidir. Önce “hava güneşli mi?” sorusunun yanıtı bulunur; yanıt doğru ya da yanlıştır. Ardından bu yanıta göre bir eylem seçilir.

Aşağıdaki Java programı iki yapı taşını birlikte kullanır. İki sayının toplamını ve çarpımını hesaplar, sonra bir koşula göre karar verir.

```
public class Hesaplama {
    public static void main(String[] args) {
        int a = 7, b = 3;
        int toplam = a + b;     // matematik
        int carpim = a * b;     // matematik
        boolean gunesli = true; // mantık
        System.out.println("Toplam: " + toplam);
        System.out.println("Çarpım: " + carpim);
        if (gunesli) {          // karar
            System.out.println("Dışarı çık");
        }
    }
}
```

Program çalıştırıldığında şu çıktıyı üretir:

```
Toplam: 10
Çarpım: 21
Dışarı çık
```

Program satır satır şöyle çalışır:

- `public class Hesaplama` satırı programın sınıfını tanımlar. Java'da her kod bir sınıfın içinde yazılır ve dosyanın adı sınıfın adıyla aynı olmalıdır: `Hesaplama.java`.

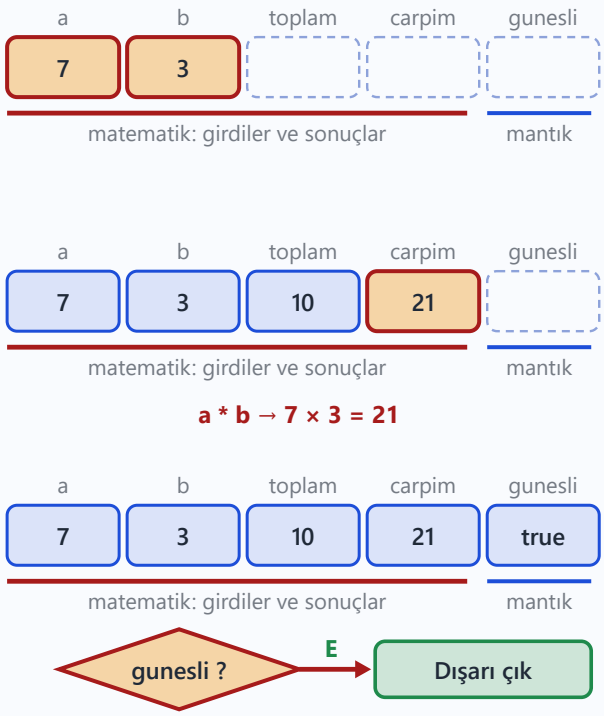
- `main` metodu programın giriş noktasıdır. Java sanal makinesi programı başlatınca ilk bu metodu çağırır.
- `int a = 7, b = 3;` satırı iki tam sayı değişkeni tanımlar ve onlara ilk değerlerini verir. Bunlar hesaplamanın girdileridir.
- `toplam` ve `carpim` satırları matematik yapı taşını kullanır. Sağ taraftaki ifade hesaplanır, sonuç soldaki değişkene yazılır.
- `boolean gunesli = true;` satırı mantık yapı taşının verisini hazırlar. `boolean` tipindeki bir değişken yalnız iki değerden birini alabilir: `true` ya da `false`.
- İki `println` satırı sonuçları yazdırır. `"Toplam: " + toplam` ifadesinde `+` işareti sayıyı metne ekler; buna dizgi birleştirme denir.
- `if (gunesli)` satırı kararı verir. Koşul `true` olduğu için süslü parantezlerin içindeki blok çalışır ve "Dışarı çık" yazılır. Koşul `false` olsaydı blok atlanır ve programın çıktısı yalnız iki satır olurdu.

1.2.1.5 Kod izleme: Hesaplama programı

Bir programın ne yaptığını anlamanın en güvenilir yolu onu elle izlemektir. **Kod izleme**, programı satır satır yürütür gibi okuyup her adımda değişkenlerin değerini ve ekrana yazılanları kaydetmektir. Tablo 1.2.1 `Hesaplama` programının izlemesidir. Satır numaraları kodun ilk satırından sayılır: 1. satır sınıf başlığı, 3. satır `int a = 7, b = 3;` satırıdır.

Satır	a	b	toplam	carpim	gunesli	Çıktı (yeni satır)
3	7	3				
4	7	3	**10**			
5	7	3	10	**21**		
6	7	3	10	21	**true**	
7	7	3	10	21	true	Toplam: 10
8	7	3	10	21	true	Çarpım: 21
9	7	3	10	21	true	(koşul doğru)
10	7	3	10	21	true	Dışarı çık

Tablo 1.2.1: Hesaplama programının izlemesi. Boş hücre, değişkenin henüz tanımlanmadığını gösterir.



Şekil 1.2.2: Kod izlemesinden üç an. (a) 3. satır: girdiler `a` = 7 ve `b` = 3 saklanır; öteki kutular henüz boştur. (b) 5. satır: `a * b` = 21 hesaplanır. (c) 10. satır: `gunesli` doğru olduğu için karar kutusunun "Evet" kolu seçilir ve "Dışarı çık" yazılır.

İzlemede iki ayrıntı dikkat çeker. Birincisi, bir değişken değer aldıktan sonra yeni bir atama gelmedikçe o değeri korur; tabloda `a` ve `b` baştan sona değişmez. İkincisi, 9. satırda ekrana bir şey yazılmaz. Bu satırda yalnız koşul değerlendirilir ve hangi yolun izleneceği belirlenir. Kararın sonucu bir sonraki satırda görünür. Program böylece hesaplamanın üç aşamasını sırayla gösterir: veri alındı, işlendi, sonuç üretildi.

Sık yapılan hata

Dizgi birleştirmede işlem sırasını unutmak. `System.out.println("Toplam: " + a + b);` satırı "Toplam: 10" değil "Toplam: 73" yazar. Java ifadeyi soldan sağa değerlendirir: önce `"Toplam: " + a` birleşir ve "Toplam: 7" dizgisi oluşur, ardından buna `b` eklenir. Sayıların önce toplanması için parantez gerekir: `"Toplam: " + (a + b)`.

1.2.2 Bilgisayar bilimi

"Bilgisayar bilimi" adı yanıltıcı olabilir. Ad, alanın bilgisayar denen makineyle ilgili olduğunu düşündürür. Oysa alanın asıl konusu makine değil, makinenin çözdüğü problemler ve bu problemleri çözme yollarıdır.

1.2.2.1 Teknolojiden fazlası

Tanım 1.2.3

Bilgisayar bilimi (*computer science*), bilginin nasıl işlendiğini, depolandığını ve iletildiğini inceleyen; problemleri tanımlama, analiz etme ve bu problemlere etkili çözümler üretme yollarını araştıran bilim dalıdır.

Bilgisayar bilimi yalnız bilgisayarlarla ilgili teknik bilgiden oluşmaz; aynı zamanda problem çözme becerileriyle ilgili bir disiplindir. Teknoloji bu disiplinde bir araçtır. Temel amaç sorunları çözmek ve bilgiyi daha iyi anlamaktır. Sık anılan bir benzetme bunu açıkça anlatır: astronomi teleskoplarla ne kadar ilgiliyse, bilgisayar bilimi de bilgisayarlarla o kadar ilgilidir. Teleskop gökyüzünü incelemenin aracıdır; bilgisayar da problem çözmenin.

Bu bakışın pratik bir sonucu vardır. Bir programlama dilinin ya da bir geliştirme ortamının ayrıntıları birkaç yılda değişir. Bir problemi parçalara ayırma, doğru veri yapısını seçme ve çözümün doğruluğunu sınama becerisi ise değişmez. Bu derste Java öğrenilir, ama Java'nın arkasındaki düşünme biçimi başka dillere de taşınır.

1.2.2.2 Üç dayanak: mantık, problem çözme, yaratıcılık

Bilgisayar bilimi üç beceriye dayanır:

- **Mantık:** Verileri değerlendirip doğru sonuçlar çıkarma yeteneğidir. Program yazma sürecinin temelini oluşturur. Bir `if` deyiminin koşulu, bir döngünün ne zaman duracağı, iki koşulun `&&` ile mi `||` ile mi bağlanacağı hep mantık sorularıdır.
- **Problem çözme:** Karmaşık problemleri tanımlama ve çözme becerisidir. Her problem, mantıklı bir adım sırasıyla çözülür. Bu adım sırasını bulmak programlamanın en zor ve en öğretici kısmıdır.
- **Yaratıcılık:** Yeni ve yenilikçi çözümlerin doğmasını sağlar. Aynı problemin birçok farklı çözümü olabilir. İki sayının EBOB'u bütün sayıları tek tek deneyerek de, Öklid algoritmasıyla da bulunur; ikisi aynı sonucu verir, ama büyük sayılarda biri diğerinden çok daha hızlıdır.

Bu üç beceri ayrı ayrı öğretilmez; programlama yaptıkça birlikte gelişir. Bir programdaki hata ayıklanırken mantık, programın parçaları tasarlanırken problem çözme, takılınan bir noktada başka bir yol aranırken yaratıcılık kullanılır.

1.2.2.3 Teknoloji ve sosyal sorumluluk

Bilgisayar bilimi, teknolojinin insanlara nasıl yardımcı olabileceğini ve toplumsal sorunların çözümüne nasıl katkı sağlayabileceğini düşünmeye yöneltir. Dünyayı daha iyi anlamaya ve geliştirmeye yardımcı olur. Teknolojiyi insanlığın hizmetine sunma sorumluluğunu da vurgular. Bu yüzden alan, yalnız teknolojiyle ilgilenenler için değil, problem çözme ve düşünme becerilerini geliştirmek isteyen herkes için ilginçtir.

Uygulama alanları çok geniştir: tıp, ekonomi, astronomi, biyoloji, enerji, ulaşım ve afet yönetimi bunlardan yalnız birkaçıdır. Bir deprem sonrasında hasar tespiti, uydu görüntülerini işleyen programlarla hızlanır. Bir elektrik şebekesinde yük dengelemesi, tüketimi öngören modellerle yapılır.

Bu genişliğin bir bedeli de vardır: yazılan programlar gerçek insanları etkiler. Bir hastane yazılımındaki hata yanlış ilaç dozuna, kişisel verilerin korunmaması da ciddi mağduriyetlere yol açabilir. Programcının sorumluluğu, kodun derlenmesiyle bitmez; kodun doğru çalıştığından ve kullanıcılarına zarar vermediğinden emin olmayı da kapsar.

1.2.2.4 Bilgisayar biliminin konuları

Bilgisayar bilimi gerçekliği anlamayı, modellemeyi ve analiz etmeyi amaçlar. Problemleri çözmek için algoritmalar ve kod kullanılır. Alanın içinde üç geniş başlık öne çıkar:

- **Soyutlama** (*abstraction*): Karmaşık bir sistemin gereksiz ayrıntılarını gizleyip özünü bırakmak. Metro haritası iyi bir örnektir. Harita sokakları, binaları ve gerçek uzaklıkları göstermez; yalnız durakları ve hatları gösterir. Yolcunun bilmesi gereken de budur: hangi duraktan binileceği, nerede aktarma yapılacağı.
- **Problem çözme:** Problemi parçalara ayırmak, her parçayı adım adım çözmek ve çözümleri birleştirmek.
- **Sanat ve tasarım:** Bilgisayar bilimi mantık ve analitik düşünceyle sınırlı değildir. Dijital medya, elektronik müzik, oyunlar ve animasyon da bu alanın ürünleridir.

Soyutlama ile problem çözme, hesaplamalı düşünmenin de iki ayağıdır. Örüntü tanıma ve algoritma tasarımı bu ikisini tamamlar ve dört beceri birlikte bir çalışma yöntemi oluşturur.

1.2.3 Hesaplamalı düşünme

Bir program yazmanın en zor kısmı çoğu zaman Java'nın kurallarını hatırlamak değildir. Asıl zorluk, kodun ilk satırı yazılmadan önce gelir. Problem ne istiyor ve hangi parçalardan oluşuyor? Hangi bilgi gerekli, hangisi gereksiz? Parçalar hangi sırayla çözülecek? Bu sorulara düzenli biçimde yanıt arama alışkanlığına **hesaplamalı düşünme** denir.

1.2.3.1 Tanım

Tanım 1.2.4

Hesaplamalı düşünme (*computational thinking*), bir problemi ve çözümünü, bir insanın ya da bilgisayarın adım adım yürütebileceği biçimde ifade etme yoludur. Dört beceriye dayanır: **ayrıştırma**, **örüntü tanıma**, **soyutlama** ve **algoritma tasarımı**.

Terim 2006'da Jeannette Wing'in kısa bir makalesiyle yaygınlaştı. Wing, bu düşünme biçiminin yalnız bilgisayar bilimcilerine değil herkese gerekli olduğunu savundu ve onu okuma, yazma ve aritmetik gibi temel bir beceri saydı. Dört becerili ayrım, bu fikri öğretmek için okullarda ve üniversitelerde yaygın olarak kullanılan bir çerçevedir.

Hesaplamalı düşünme "bilgisayar gibi düşünmek" demek değildir. Bilgisayar düşünmez; yalnız talimatları yürütür. Hesaplamalı düşünme, insanın bir problemi bilgisayara anlatılabilir hâle getirme becerisidir. Aynı beceri bilgisayar olmadan da işe yarar. Bir yemek tarifi yazmak, bir devreyi sınamak için ölçüm sırası belirlemek ya da bir sınava çalışma planı yapmak da aynı dört adımla düzenlenebilir.

Dört beceriyi tek bir problem üzerinde görmek, onları ayrı ayrı tanımlamaktan daha öğreticidir. Bu kesim boyunca örnek problem şudur: *bir sınıftaki öğrencilerin not ortalamasını bulmak*.

1.2.3.2 Ayrıştırma

Tanım 1.2.5

Ayrıştırma (*decomposition*), büyük ve karmaşık bir problemi, her biri ayrı ayrı çözülebilecek küçük alt problemlere bölmektir.

"Not ortalamasını bul" isteği tek bir adım gibi görünür, ama bilgisayar için dört ayrı işten oluşur:

1. Notları oku.
2. Notları topla.
3. Toplamı not sayısına böl.
4. Sonucu yaz.

Her alt problem tek başına kolaydır. Okuma için bir giriş komutu, toplama için bir döngü, bölme için tek bir işlem, yazma için `println` yeterlidir. Ayrıştırmanın gücü buradan gelir: zor bir problem, kolay problemlerin birleşimine dönüşür.

Ayrıştırma bir kez yapılıp bırakılmaz. Bir alt problem hâlâ büyükse o da bölünür. "Notları oku" adımı, notların klavyeden mi, bir dosyadan mı, bir veri tabanından mı geleceğine göre yeniden parçalanabilir. Java'da her alt problem çoğu zaman ayrı bir metoda karşılık gelir. Büyük programlar da böyle kurulur: sınıflar ve metotlar, ayrıştırmanın koddaki izleridir.

Ayrıştırmanın elektronikteki karşılığı blok diyagramdır. Bir ses yükselteci tasarlanırken önce ön yükselteç, ton denetimi ve güç katı gibi bloklar ayrılır; her blok ayrı tasarlanıp sınanır, sonra birleştirilir. Bir programdaki metotlar da aynı mantıkla ayrı ayrı yazılıp denenebilir.

1.2.3.3 Örüntü tanıma

Tanım 1.2.6

Örüntü tanıma (*pattern recognition*), alt problemler arasındaki ya da bir problemin bilinen başka problemlerle arasındaki benzerlikleri görmektir. Aynı biçimde tekrar eden bir iş bulunursa tek bir çözüm hepsine uygulanır.

Not ortalaması örneğinde örüntü açıktır. Birinci notu okuyup toplama eklemek ile ikinci notu okuyup toplama eklemek aynı iştir; yalnız not değişir. Otuz öğrencilik bir sınıf için bu adımı otuz kez ayrı ayrı yazmak gerekmez. Tekrar eden adım bir kez yazılır ve bir **döngüye** konur. Örüntü tanıma, programlamada çoğu zaman "burada bir döngü var" fikrine götürür.

Örüntü yalnız tek problemin içinde aranmaz; problemler arasında da aranır. Not ortalaması ile bir sensörün bir günlük sıcaklık ortalaması aynı problemdir: bir sayı listesinin toplamını alıp eleman sayısına bölmek. Biri için yazılan çözüm, küçük bir değişiklikle öbürüne de uyar. Babil yöntemi ve Öklid algoritmasında da tekrar eden bir adım vardır: biri tahmini iyileştirir, öbürü kalanı hesaplar. Her ikisi de bu adımı belirli bir koşul sağlanana kadar yineler.

Örüntü tanıma tecrübeyle güçlenir. Çok sayıda problem çözmüş bir programcı, yeni bir problemde "bu, en büyüğü bulma problemi" ya da "bu, iki listeyi birleştirme problemi" diyebilir. Problemi tanınan bir kalıba indirgemek, çözümü sıfırdan bulmaktan çok daha hızlıdır.

1.2.3.4 Soyutlama

Tanım 1.2.7

Soyutlama (*abstraction*), bir problemin çözümü için gerekli olmayan ayrıntıları atıp yalnız önemli özellikleri bırakmaktır.

Bir öğrenciyle ilgili çok sayıda bilgi vardır: adı, numarası, doğum tarihi, memleketi, devamsızlığı, notları. Ortalama hesabı için bunların çoğu gereksizdir. Öğrencinin adı ya da numarası ortalamayı etkilemez. Gerekli olan yalnız iki bilgidir: notlar ve not sayısı. Soyutlama yapıldıktan sonra problem "otuz öğrencinin ortalaması" olmaktan çıkar, "bir sayı listesinin ortalaması" olur. Bu daha genel bir problemdir ve çözümü başka yerlerde de kullanılabilir.

Metro haritası örneği aynı fikri gösterir: harita gerçek şehrin küçültülmüş bir fotoğrafı değildir; yolcuya gereken bilgi dışındaki her şey atılmıştır. İyi bir soyutlama, atılan ayrıntının gerçekten gereksiz olduğundan emin olmayı gerektirir. Notların ağırlıkları farklıysa, örneğin vize %40 ve final %60 ise, "yalnız notlar yeterli" soyutlaması yanlış olur; ağırlıklar da modele girmelidir.

Soyutlama nesne yönelimli programlamanın da temelidir. Java'da bir `Ogrenci` sınıfı tasarlanırken gerçek bir öğrencinin bütün özellikleri değil, programın ihtiyaç duyduğu özellikler sınıfa alan olarak konur. Bir kayıt sisteminde ad ve numara gerekir, bir not sisteminde notlar gerekir. Aynı gerçek varlık, farklı programlarda farklı biçimde soyutlanır.

1.2.3.5 Algoritma tasarımı

> **Tanım 1.2.8**
>
> **Algoritma tasarımı** (*algorithm design*), çözümü sonlu, açık ve sıralı adımlar olarak yazmaktır. Ortaya çıkan adım listesi bir insanın elle izleyebileceği ya da bir programlama diline çevrilebilecek kesinliktedir.

Önceki üç beceri problemi hazırlar; algoritma tasarımı çözümü yazar. Not ortalaması için algoritma, sunumdaki sözde kod biçimiyle şöyledir:

```
set toplam to 0
for each not in notlar do
    set toplam to toplam + not
set ort to toplam / n
print ort
```

Bu adımlar ayrıştırmanın dört alt problemini, örüntü tanımanın bulduğu döngüyü ve soyutlamanın bıraktığı iki bilgiyi (notlar ve n) bir araya getirir. Her adım açıktır: ne yapılacağı konusunda yoruma yer yoktur. Adım sayısı sonludur: döngü listedeki her not için bir kez çalışır ve liste bitince durur.

Algoritma yazılırken bazı ayrıntılar da kararlaştırılır. Toplam neden 0'dan başlar? Çünkü hiç not eklenmemişken toplam sıfırdır ve her not bu değere eklenir. Liste boşsa ne olur? $n = 0$ olur ve bölme tanımsız kalır. İyi tasarlanmış bir algoritma bu durumu da düşünür; örneğin “liste boşsa ‘not yok’ yaz” diye bir adım ekler.

1.2.3.6 Dört beceri bir arada: not ortalaması

Tablo 1.2.2 dört becerinin örnek problemde neye karşılık geldiğini özetler.

Beceri	Sorduğu soru	Not ortalamasındaki karşılığı
Ayrıştırma	Problem hangi parçalardan oluşuyor?	Oku, topla, böl, yaz.
Örüntü tanıma	Tekrar eden ya da tanıdık bir iş var mı?	Her notu toplama eklemek aynı adımdır: döngü.
Soyutlama	Hangi bilgi gerekli, hangisi gereksiz?	Ad ve numara gereksiz; notlar ve not sayısı gerekli.

Beceri	Sorduğu soru	Not ortalamasındaki karşılığı
Algoritma tasarımı	Çözüm hangi adımlarla, hangi sırayla yapılır?	`toplam ← 0`; her not için `toplam ← toplam + not`; `ort ← toplam / n`.

Tablo 1.2.2: Hesaplamalı düşünmenin dört becerisi ve not ortalaması problemindeki karşılıkları.

Algoritma hazır olunca Java'ya çevirmek kolaydır. Aşağıdaki program, notları bir dizide tutar ve ortalamayı hesaplar. Notlar sunumun örnek problemlerindeki 40, 75 ve 90'dır.

```
public class Ortalama {
    public static void main(String[] args) {
        int[] notlar = {40, 75, 90};   // girdi: yalnız notlar
        int toplam = 0;
        for (int not : notlar) {        // her not için aynı adım
            toplam = toplam + not;
        }
        double ort = (double) toplam / notlar.length;
        System.out.println("Ortalama: " + ort);
    }
}
```

Program “Ortalama: 68.33333333333333” yazar. Kodun her parçası bir beceriye karşılık gelir. `notlar` dizisi soyutlamanın sonucudur: öğrenciler değil, yalnız notlar tutulur. `for` döngüsü örüntü tanımanın bulduğu tekrar eden adımdır. Toplama ve bölme satırları ayrıştırmanın alt problemleridir. Bütün kod da algoritmanın Java'daki karşılığıdır.

Adım	`not`	`toplam`	Açıklama
Başlangıç	–	0	Hiç not eklenmedi.
1. tur	40	40	0 + 40
2. tur	75	115	40 + 75
3. tur	90	205	115 + 90; liste bitti, döngüden çıkılır.
Bölme	–	205	`ort` = 205 / 3 = 68,333…

Tablo 1.2.3: Ortalama programının izlemesi (notlar 40, 75, 90).

Bölme satırındaki `(double)` dönüşümü önemlidir. `toplam` ve `notlar.length` ikisi de `int` tipindedir. Java iki `int` değeri bölerken sonucu da `int` verir ve kesirli kısmı atar: 205 / 3 = 68. `(double) toplam` yazılınca bölünen ondalıklı sayıya çevrilir ve gerçek bölme yapılır. Bu ayrıntı soyutlamanın gözden kaçırabileceği türden bir şeydir: algoritma kâğıt üzerinde doğrudur, ama Java'nın sayı tipleri hesaba katılmazsa program yanlış sonuç verir.

Örnek 1.2.1

Bir laboratuvarda bir direncin üzerindeki gerilim bir saat boyunca her saniye ölçülüyor. Ölçümlerden en büyük değerin ve 5 V'u aşan ölçüm sayısının bulunması isteniyor. Dört beceri sırayla uygulanır.

Ayrıştırma: ölçümleri oku; en büyüğü bul; 5 V'u aşanları say; iki sonucu yaz. **Örüntü tanıma:** en büyüğü bulmak ve saymak, her ölçüm için tekrar eden birer adımdır; ikisi aynı döngüde yapılabilir. Ayrıca "en büyüğü bulma" tanınan bir kalıptır. **Soyutlama:** ölçümün saati, laboratuvarın adı ve direncin rengi gereksizdir; gereken yalnız 3600 sayılık bir gerilim listesidir. **Algoritma tasarımı:** `enBuyuk ← ilk ölçüm`, `sayac ← 0`; her ölçüm için, ölçüm `enBuyuk`'tan büyükse `enBuyuk ← ölçüm`, ölçüm 5'ten büyükse `sayac ← sayac + 1`; sonunda iki değeri yaz.

`enBuyuk`'un 0 ile değil ilk ölçümle başlatılması bilinçli bir seçimdir. Ölçümlerin hepsi negatif olsaydı, 0 ile başlayan bir algoritma yanlışlıkla 0 bulurdu.

Sık yapılan hata

Hesaplamalı düşünme adımlarını atlayıp doğrudan kod yazmaya başlamak. Problem ayrıştırılmadan yazılan kod genellikle tek bir uzun `main` metoduna dönüşür; hata çıktığında hangi parçanın sorumlu olduğu bulunamaz. Soyutlama yapılmadan yazılan kod da gereksiz veriyle uğraşır ya da gerekli bir bilgiyi (örneğin not ağırlıklarını) unutur.

1.2.4 Bilgisayarın tarihi

Bugünkü bilgisayar tek bir buluşun değil, birkaç yüzyıl süren bir gelişmenin ürünüdür. Bu gelişme iki fikrin etrafında döner. Birincisi hesabı makineye yaptırmaktır: dişli çarklardan vakum tüplerine, oradan transistörlere uzanan çizgi bu fikrin ürünüdür. İkincisi makinenin ne yapacağını değiştirebilmektir: sabit programlı makinelerden depolanan programlı makinelere geçiş bu fikrin ürünüdür. Tarihin ayrıntıları bu iki çizgiye yerleştirilince akılda daha kolay kalır.

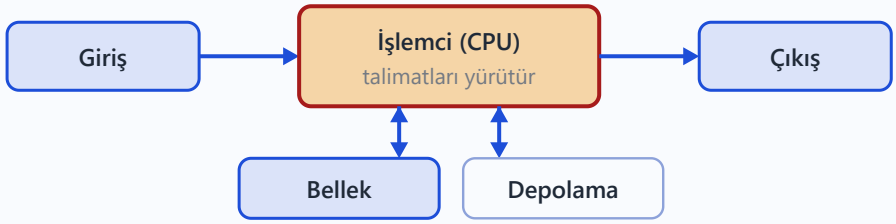
1.2.4.1 Bilgisayar nedir?

Tanım 1.2.9

Bilgisayar (*computer*), bir dizi hesaplama ve talimatı yürüten cihazdır. Talimatlar bilgisayarın bir görevi yerine getirmesini sağlar. Bilgisayar verileri işlemek için matematiksel hesaplamalar ve mantıksal talimatlar kullanır.

Günümüz bilgisayarları elektronik devrelerle çalışır ve sayıları dijital olarak temsil eder. Dört ana bileşenden oluşur:

- **İşlemci** (CPU, *central processing unit*, merkezî işlem birimi): Talimatları yürütür, hesapları yapar.
- **Bellek** (RAM, *random access memory*, rastgele erişimli bellek): Çalışan programı ve verisini tutar. Elektrik kesilince içeriği silinir.
- **Depolama:** Disk ve SSD gibi birimler veriyi kalıcı olarak saklar. Bilgisayar kapansa da veri kaybolmaz.
- **Giriş/çıkış birimleri:** Klavye, fare, ekran, yazıcı ve ağ bağlantısı bilgisayarın dış dünyayla iletişimini sağlar.



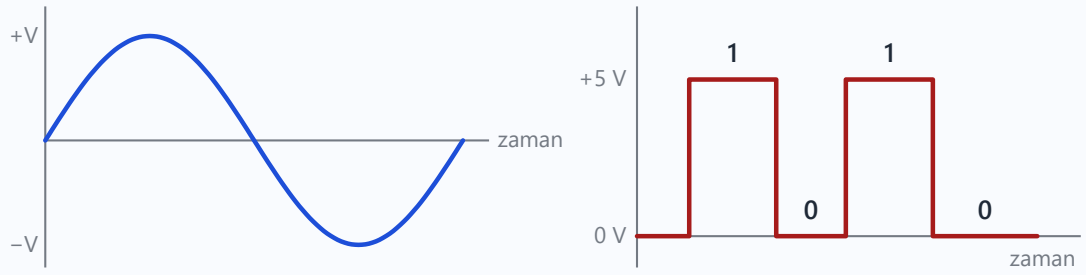
Şekil 1.2.3: Bilgisayarın bileşenleri. Giriş verisi işlemciye gelir; işlemci bellek ve depolamayla birlikte çalışır; sonuç çıkış birimine gider.

Bir Java programı çalışırken bu bileşenlerin hepsi devrededir. `.java` ve `.class` dosyaları diskte durur. Program başlatılınca belleğe yüklenir. İşlemci talimatları birer birer yürütür. `println` ise sonucu ekrana, yani bir çıkış birimine gönderir. Bellek ile depolama arasındaki fark programcı için de önemlidir: bir değişkende tutulan değer program bitince kaybolur, kalıcı olması gereken veri bir dosyaya yazılmalıdır.

1.2.4.2 Elektronik ve dijital

Bugünkü bilgisayarı tanımlayan iki sıfat vardır: elektronik ve dijital. İkisi farklı şeyler anlatır.

- **Elektronik:** Bilgisayar verileri elektrik akımıyla işler. Mekanik dişliler yerine transistörler anahtarlama yapar. Anahtarlamada hareketli parça olmadığı için işlemler çok hızlıdır.
- **Dijital (sayısal):** Veriler 0 ve 1 ile temsil edilir. Her 0 ya da 1 bir **bit**tir; 8 bit bir **bayt** eder. Bir bayt $2^8 = 256$ farklı değer alabilir.



Şekil 1.2.4: Analog ve dijital sinyal. Solda sürekli değişen ve her değeri alabilen analog bir sinüs sinyali, sağda yalnız 0 V ve 5 V düzeylerinde değişen dijital bir kare dalga görülür.

Dijital temsilin neden tercih edildiği Şekil 1.2.4'te görülür. Analog bir sinyalde her gerilim değeri farklı bir bilgi taşır. Bu yüzden sinyale karışan küçük bir gürültü bilgiyi değiştirir. Dijital sinyalde ise yalnız iki düzey vardır. 5 V'luk bir sistemde 3,9 V da 4,8 V da “1” sayılır. Gürültü sinyali biraz oynatsa da hangi düzeyde olduğu kolayca ayırt edilir. Dijital sistemlerin güvenilirliği buradan gelir. Elektronik dersinden tanıdık olan lojik kapılar, bu iki düzeyli sinyallerle çalışan devrelerdir.

1.2.4.3 Sabit programlı bilgisayar

Tanım 1.2.10

Sabit programlı bilgisayar (*fixed program computer*), belirli bir problemi çözmek için tasarlanmış ve önceden tanımlanmış tek bir programı çalıştıran makinedir. Program, makinenin yapısına gömülüdür; başka bir iş yaptırmak için makinenin kendisini değiştirmek gerekir.

Sabit programlı makinelerin kökleri çok eskilere dayanır:

- Abaküs ve Antikythera Mekanizması (MÖ 2. yüzyıl). Antikythera Mekanizması, gök cisimlerinin konumlarını dişli çarklarla hesaplayan antik bir Yunan aletidir.
- Pascaline (Blaise Pascal, 1642) ve Leibniz'in basamaklı hesap makinesi (1673).
- Babbage'ın Fark Makinesi (1822) ve Hollerith'in elektrikli çizelgeleme makinesi (1890). Hollerith'in makinesi ABD nüfus sayımının sonuçlarını delikli kartlardan saymak için yapıldı.
- Atanasoff–Berry Bilgisayarı (ABC) ve Turing Bombası. Bomba, İkinci Dünya Savaşı'nda Alman Enigma şifresini çözmek için kuruldu.



Şekil 1.2.5: Pascaline. Dişli çarklarla çalışan bu mekanik makine yalnız toplama ve çıkarma yapabilir.

Şekil 1.2.5'teki Pascaline sabit programlı makinenin tipik bir örneğidir. Çarkları toplama yapacak biçimde dizilmiştir; başka bir hesap, örneğin karekök, bu makineye yaptırılamaz. Bugün de sabit programlı cihazlar vardır: basit bir cep hesap makinesi ya da bir çamaşır makinesinin eski tip mekanik programlayıcısı aynı mantıkla çalışır. Sabit programlı makineler bilgisayar biliminin temellerini attı, teknolojik gelişmelere ilham verdi ve modern bilgisayarların öncüsü oldu.

1.2.4.4 Babbage ve Analitik Makine

Charles Babbage (1791–1871) İngiliz bir matematikçi ve mucitti. Fark Makinesi, çok terimli ifadelerin değer tablolarını hesaplamak için tasarlanmış sabit programlı bir makineydi. Babbage daha sonra çok daha iddialı bir tasarıma yöneldi. 1837'de **Analitik Makine**'yi (*Analytical Engine*) tasarladı. Bu, ilk genel amaçlı ve programlanabilir bilgisayar tasarımıdır.

Analitik Makine matematiksel hesaplama yapabilir ve sonuçları saklayabilirdi. Asıl yeniliği, farklı hesaplamalar için programlanabilmesiydi. Bugünkü bilgisayarın parçaları bu tasarımda vardı:

- **store** (depo): sayıları tutan bellek;
- **mill** (değirmen): işlemleri yapan, bugünkü işlemcinin karşılığı olan birim;
- delikli kartlarla giriş ve yazıcıyla çıkış.

Delikli kart fikri, dokuma tezgâhından alındı. Joseph Marie Jacquard 1804'te desenleri delikli kartlarla belirlenen bir dokuma tezgâhı geliştirmişti. Kart değiştirilince desen değişiyordu. Babbage aynı fikri hesaba uyguladı: kart değiştirilince hesap değişecekti. Makinenin ne yapacağını değiştirmek için makineyi değil, kartları değiştirmek yetiyordu.



Şekil 1.2.6: Analitik Makine'nin deneme modeli (makinenin bir parçası).

Makine Babbage'ın yaşamında tamamlanamadı. Dönemin metal işleme olanakları binlerce hassas parçayı üretmeye yetmiyordu, para da bulunamadı. Bugün yalnız Şekil 1.2.6'daki gibi deneme modelleri vardır. Yine de Analitik Makine modern bilgisayarların öncüsü kabul edilir; çünkü genel amaçlı bir makinenin hangi parçalardan oluşması gerektiğini ilk kez açıkça tanımladı.

1.2.4.5 Ada Lovelace ve ilk program

Ada Lovelace (1815–1852) İngiliz bir matematikçi ve yazardı. 1843'te İtalyan mühendis Luigi Menabrea'nın Analitik Makine üzerine yazdığı Fransızca bir makaleyi İngilizceye çevirdi ve çeviriye kendi notlarını ekledi. Notlar makalenin kendisinden uzundu.

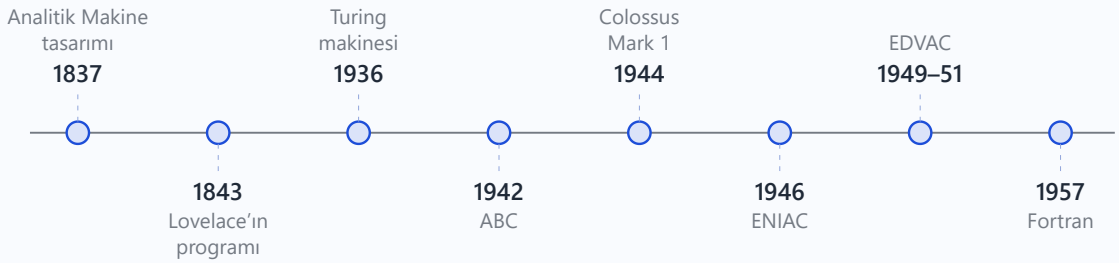
Notların en ünlüsü “Not G”dir. Lovelace bu notta makinenin Bernoulli sayılarını hesaplaması için adım adım bir yöntem verdi. Yöntem, hangi değerin hangi bellek hücresine yazılacağını ve hangi işlemin hangi sırayla yapılacağını gösteren bir tablodur. Bu tablo ilk bilgisayar programı kabul edilir ve Lovelace bilgisayar tarihinin ilk programcısı sayılır. Lovelace, bilimsel bir yaklaşımla matematiksel düşünceyi programlamayla birleştirdi.

Lovelace'ın öngörüsü programdan da ileri gider. Makinenin yalnız sayılarla değil, sembollerle de çalışabileceğini ilk fark edenlerdendir. Ona göre müzik notaları da sembollerle temsil edilirse makine müzik besteleyebilirdi. Bugünkü bilgisayarların metin, ses ve görüntü işlemesi bu fikrin gerçekleşmesidir. Lovelace'ın mirası bir programlama diline de adını verdi: 1980'de tanıtılan Ada dili onun adını taşır.

Bu öykünün bir dersi vardır: Lovelace'ın programı, makine henüz yokken yazıldı. Algoritma bir makineye bağlı değildir; kâğıt üzerinde tasarlanabilir ve elle izlenebilir. Bu not boyunca yapılan kod izlemeleri de aynı fikre dayanır.

1.2.4.6 Zaman çizelgesi ve depolanan program

Babbage ile ilk elektronik bilgisayarlar arasında yaklaşık bir yüzyıl vardır. Şekil 1.2.7 bu dönemin önemli tarihlerini bir arada gösterir.



Şekil 1.2.7: Bilgisayarın kısa tarihi: 1837 Analitik Makine tasarımı, 1843 Lovelace'ın programı, 1936 Turing makinesi, 1942 ABC, 1944 Colossus Mark 1, 1946 ENIAC, 1949–51 EDVAC, 1957 Fortran. Çizelgedeki aralıklar eşit çizilmiştir; yıllar arasındaki gerçek uzaklığı göstermez.

Çizelge üç döneme ayrılır:

- **Mekanik dönem:** Dişliler ve kartlar. Pascaline, Fark Makinesi ve Analitik Makine bu döneme aittir.
- **Kuramsal temel:** Alan Turing 1936'da "hesaplanabilir" kavramının ne demek olduğunu kesin olarak tanımladı ve evrensel makine fikrini ortaya koydu. Bu çalışma hiçbir makine yapmadı; bir makinenin neyi yapıp neyi yapamayacağını belirledi.
- **Elektronik dönem:** Vakum tüplü ABC, Colossus ve ENIAC; ardından depolanan programlı EDVAC.

Elektronik dönemin en önemli adımı **depolanan program** fikridir.

Tanım 1.2.11

Depolanan programlı bilgisayar (*stored-program computer*), programı da veri gibi bellekte saklayan bilgisayardır. Programı değiştirmek, belleğe yeni bir program yazmak demektir; makinenin yapısına dokunulmaz.

Fikrin 1945'teki önemi şuradadır: program ve veri aynı bellekte durduğu için program da bir veridir. Bir program, başka bir programı girdi olarak okuyabilir, değiştirebilir ya da üretebilir. Derleyiciler tam olarak bunu yapar: `javac`, girdi olarak bir Java kaynak dosyasını okur ve çıktı olarak başka bir program üretir. Java sanal makinesi de `.class` dosyasındaki programı veri gibi okuyup çalıştırır.

Tanım 1.2.12

Von Neumann mimarisi (*von Neumann architecture*), komutların ve verinin aynı bellekte tutulduğu bilgisayar düzenidir. İşlemci komutları bellekten sırayla okur ve yürütür. İşlemci, bellek ve giriş/çıkış birimleri adres, veri ve kontrol yollarıyla (*bus*) haberleşir.

Bugün kullanılan dizüstü bilgisayarlar, telefonlar ve sunucular temelde bu mimariyle çalışır. İşlemci bir döngü içinde sıradaki komutu bellekten getirir, çözer ve yürütür; sonra bir sonraki komuta geçer. Bir koşullu atlama komutu bu sırayı değiştirebilir. Java'daki `if` ve `while` deyimleri, derlendikten sonra bu tür atlama komutlarına dönüşür.

1.2.4.7 İlk elektronik bilgisayarlar: ABC ve Colossus

Atanasoff–Berry Bilgisayarı (ABC), John Atanasoff ve Clifford Berry'nin 1937–1942 yılları arasında Iowa State Üniversitesi'nde geliştirdiği makinedir. İlk elektronik dijital bilgisayar kabul edilir. ABC'nin özellikleri şunlardır:

- Mantık devreleri vakum tüpleriyle kurulmuştu.
- Sayıları ikili (0/1) sistemde işledi.
- Belleği, dönen tamburlar üzerindeki kondansatörlerden oluşuyordu. Bir kondansatörün yüklü ya da boş olması bir biti temsil ediyordu.
- Programlanamazdı. Yalnız doğrusal denklem sistemlerini çözüyordu; yani sabit programlı bir makineydi.

Colossus Mark 1, Tommy Flowers ve ekibinin 1944'te İngiltere'deki Bletchley Park'ta kurduğu makinedir. İlk programlanabilir elektronik dijital bilgisayar kabul edilir; anahtar ve fişlerle programlanıyordu. İkinci Dünya Savaşı'nda düşmanın şifreli mesajlarını çözmek için kullanıldı. Yaklaşık 1600 vakum tüpü içeriyordu ve mekanik cihazlardan çok daha hızlı ve esnekti. Colossus, bilgisayar teknolojisinin ilerlemesine büyük katkı sağladı; ancak savaştan sonra uzun yıllar gizli tutulduğu için etkisi geç anlaşıldı.

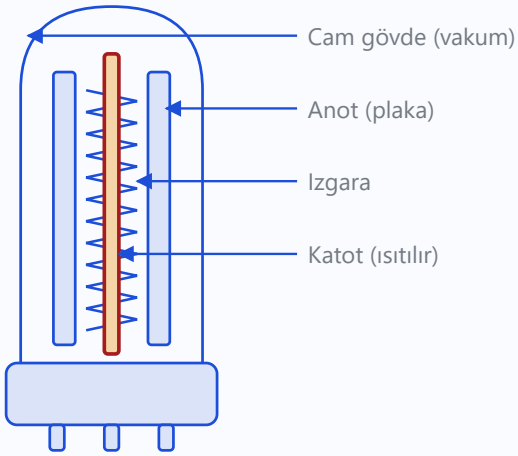
İki makine arasındaki fark, tarihin ana çizgisini özetler. ABC elektronik ve ikiliydi, ama tek bir işe bağlıydı. Colossus programlanabiliyordu, ama programı değiştirmek anahtarları ve fişleri elle değiştirmek demekti. Programın bellekte durması için birkaç yıl daha beklemek gerekti.

1.2.4.8 Vakum tüpü ve transistör

İlk elektronik bilgisayarların ortak parçası **vakum tüpü**dür. Vakum tüpü, havası boşaltılmış bir cam tüptür. İçinde üç ana eleman bulunur:

- **Katot:** Isıtılınca elektron yayar.
- **Anot** (plaka): Katodun yaydığı elektronları toplar.

- **Izgara:** Katot ile anot arasında durur. Izgaraya uygulanan küçük bir gerilim, elektron akışını açar ya da keser.



Şekil 1.2.8: Vakum tüpünün yapısı. Cam gövdenin ortasında ısıtılan katot, çevresinde ızgara ve en dışta anot plakası bulunur.

Izgara gerilimiyle akımın açılıp kapanabilmesi, tüpü bir elektronik anahtar yapar: akım geçiyorsa 1, geçmiyorsa 0. Bu anahtarlama hareketli parça içermediği için mekanik rölelerden binlerce kat hızlıdır. Bir röle milisaniyeler içinde açılıp kapanırken bir tüp mikrosaniyeler düzeyinde anahtarlama yapar.

Vakum tüpünün sakıncaları da büyüktü. Tüpler çok ısınır, çok enerji harcar ve sık arızalanır. Binlerce tüplü bir makinede her gün birkaç tüpün yanması olağandı. Çözüm 1947'de icat edilen **transistör** oldu. Transistör aynı anahtarlama işini çok daha küçük, serin ve güvenilir biçimde yapar. Elektronik derslerinde incelenen MOSFET'ler de bu işi görür. Bugünkü işlemcilerde milyarlarca transistör vardır.

1.2.4.9 ENIAC ve EDVAC

ENIAC (1946), John Mauchly ve J. Presper Eckert'in Pennsylvania Üniversitesi'nde (ABD) kurduğu makinedir. Genel amaçlı ilk büyük ölçekli elektronik bilgisayar kabul edilir. Mekanik parçalar yerine yaklaşık 17.500 vakum tüpü kullanıyordu; bu yüzden önceki makinelerden çok daha hızlıydı. Saniyede yaklaşık 5000 toplama yapabiliyordu. Topçu atış tabloları gibi büyük hesaplar için kullanıldı.

ENIAC'ın iki zayıf yanı vardı. Sayıları onlu sistemde tutuyordu; bu, devreleri karmaşıklaştırıyordu. Daha önemlisi, programı değiştirmek için kablolar elle yeniden takılıyordu. Yeni bir problem için makineyi hazırlamak günler sürebilirdi.

EDVAC (1949–1951) bu iki sorunu çözdü. Tasarımı, John von Neumann'ın 1945'te yazdığı bir raporda tanımlandı; makine 1951'de tam olarak çalıştı. EDVAC'ın yenilikleri şunlardır:

- ENIAC'tan farklı olarak ikili sayı sistemini kullandı.

- Programı ve veriyi aynı bellekte sakladı. Bu, depolanan program fikrinin uygulamasıdır.
- Komutlar sırayla çalışıyordu; koşula bağlı atlama yapılabiliyordu.

Bu tasarım, bugünkü bilgisayarların temeli olan von Neumann mimarisidir. ENIAC ile EDVAC'ı ayıran şey hız değil, programın nerede durduğudur. ENIAC'ta program kablolarda, EDVAC'ta bellekteydi.

1.2.4.10 Genel amaçlı bilgisayar

Tanım 1.2.13

Genel amaçlı bilgisayar (*general purpose computer*), kullanıcının ihtiyacına göre programlanabilen ve geniş bir uygulama yelpazesi için tasarlanmış bilgisayardır. İşlemci, bellek ve giriş/çıkış birimlerinden oluşur; işlemci verileri talimatlara göre işler ve sonuç üretir.

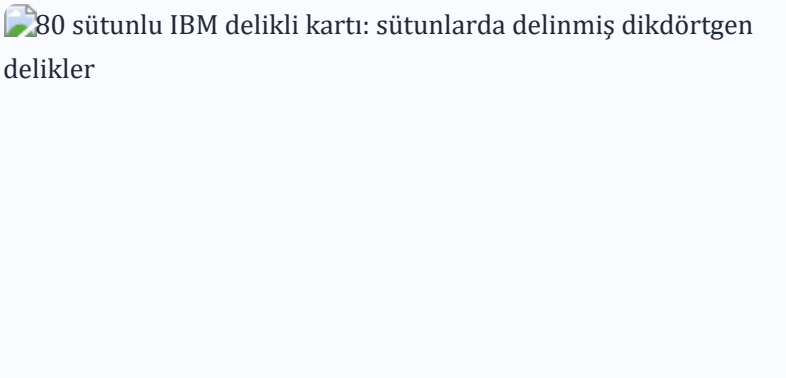
Genel amaçlı bilgisayar iş, eğitim ve eğlence gibi birçok alanda kullanılır. Program değiştirilerek farklı görevlere uyarlanır: aynı dizüstü bilgisayar bir dakika önce müzik çalar, bir dakika sonra Java derler. Sabit programlı makineyle farkı budur. Genel amaçlı bilgisayarlar iş süreçlerini otomatikleştirir, bilimsel hesaplamaları hızlandırır ve iletişimi kolaylaştırır.

Özellik	Sabit programlı	Depolanan programlı
Programın yeri	Makinenin yapısında (dişli, kablo, devre)	Bellekte, veri gibi
Program değiştirmek	Makineyi yeniden kurmak	Belleğe yeni program yüklemek
Yapabildiği iş	Tek bir iş	Programlanabilen her iş
Örnekler	Pascaline, Fark Makinesi, ABC	EDVAC, bugünkü bilgisayarlar

Tablo 1.2.4: Sabit programlı ve depolanan programlı bilgisayarların karşılaştırması.

1.2.4.11 İlk programlar ve Fortran

İlk programlar çoğunlukla matematiksel hesaplamalar içindi. Kullanımlar bilimsel ve askerî uygulamalara odaklanıyordu: atış tabloları, nükleer fizik hesapları, hava durumu tahmini. Programlar ve veriler delikli kartlarla girilirdi. Bir kartın her sütunundaki delik düzeni bir karakteri kodlardı.



Şekil 1.2.9: 80 sütunlu delikli kart. Bir kart yaklaşık bir satır programa karşılık gelir.

Şekil 1.2.9'daki gibi bir kart 80 karakter tutardı. Yüz satırlık bir program yüz kartlık bir desteydi. Deste düşüp kartlar karışırsa program bozulurdu. Bugün bazı kod düzenleyicilerinde görülen 80 karakterlik satır sınırı bu dönemden kalmadır.

İlk programlar makine diliyle ya da makine diline çok yakın biçimde yazılıyordu. Her işlemcinin komutları farklı olduğu için bir makinede yazılan program başka bir makinede çalışmazdı. İlk yaygın **yüksek düzeyli programlama dili** olan **Fortran** (*Formula Translation*) bu sorunu hafifletti. Fortran, John Backus önderliğinde IBM'de geliştirildi ve 1957'de kullanıma sunuldu. Bilimsel hesaplamalarda yaygın olarak kullanıldı ve bugün de bu alanda kullanılır.

Yüksek düzeyli dil, programcının `K = I + J` gibi matematiğe yakın ifadeler yazmasını sağlar. Bir **derleyici** bu ifadeyi makine komutlarına çevirir. Programcı artık bellek adresleriyle değil, değişken adlarıyla düşünür. Java da (1995) yüksek düzeyli bir dildir; `int toplam = a + b;` satırı bu geleneğin devamıdır.

1.2.5 Turing makineleri

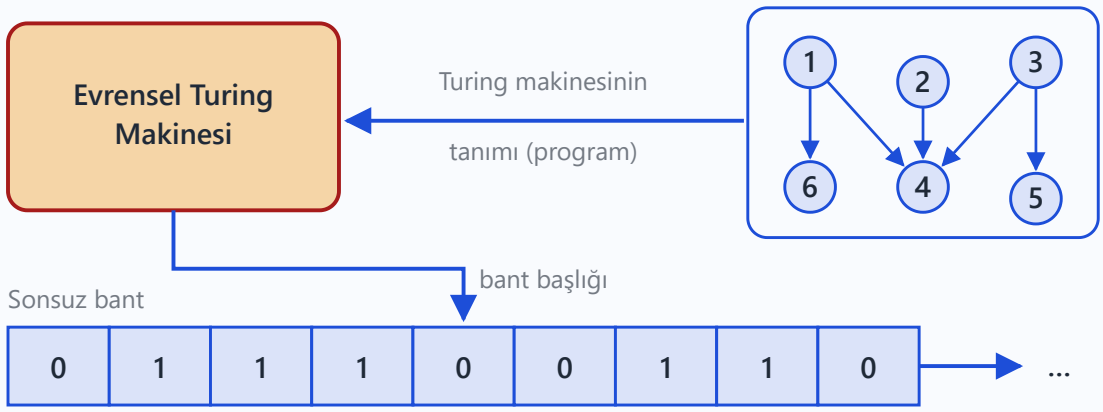
“Hesaplama” sözcüğü günlük dilde bulanık bir anlam taşır. Bir problem için “bilgisayarla çözülebilir” demek ne anlama gelir? Hangi bilgisayarla, ne kadar bellekle, hangi dille? Alan Turing 1936'da bu soruyu bütün makinelerden bağımsız olarak yanıtlamak için son derece basit bir soyut makine tanımladı. Bu makine gerçek bir bilgisayar gibi hızlı ya da kullanışlı değildir. Değeri, hesaplamanın ne olduğunu kesin olarak tanımlamasından gelir.

1.2.5.1 Evrensel Turing makinesi

Tanım 1.2.14

Evrensel Turing makinesi (*universal Turing machine*), girdi olarak başka bir Turing makinesinin tanımını ve o makinenin girdisini alan, sonra o makineyi taklit eden (benzeten) soyut bir Turing makinesidir.

Alan Turing'in 1936'da tanımladığı bu model, soyut ve genel amaçlı bir bilgisayar modelidir ve bilgisayar biliminin en temel modellerinden biridir. Evrensel makine, kendisine verilen her Turing makinesinin yaptığını yapabilir. Bu yüzden hesaplanabilir her problemi çözebilir.



Şekil 1.2.10: Evrensel Turing makinesi. Başka bir Turing makinesinin tanımını (programını) girdi olarak alır ve sonsuz bant üzerinde o makineyi taklit eder.

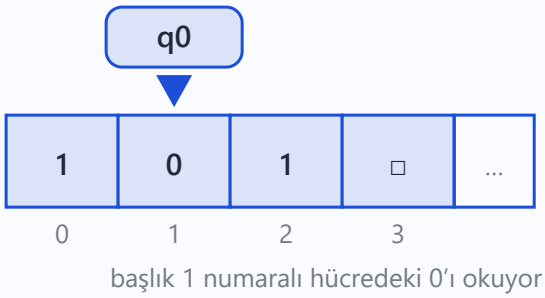
Evrensel makine fikri, depolanan program fikrinin kuramsal karşılığıdır. Şekil 1.2.10'da taklit edilecek makinenin tanımı, bant üzerindeki öteki veriler gibi okunur. Bugünkü bilgisayarlar da aynı biçimde çalışır: tek bir donanım, bellekteki programa göre metin düzenleyici, oyun ya da derleyici olur. Modern bilgisayarların evrensel olarak programlanabilmesinin kuramsal temeli budur. Depolanan programlı bilgisayar ve evrensel Turing makinesi, bilgisayar teknolojisinin iki önemli dönüm noktasıdır: biri makinenin nasıl yapılacağını, öbürü ne yapabileceğini belirledi.

1.2.5.2 Turing makinesinin parçaları

Tanım 1.2.15

Turing makinesi (*Turing machine*) üç parçadan oluşan soyut bir hesaplama modelidir:

- **Bant** (*tape*): Hücrelere bölünmüş, sonsuz uzunlukta bir şerit. Girdi, çıktı ve ara sonuçlar bantta saklanır. Her hücrede bir sembol (0, 1, boş □ ...) bulunur.
- **Bant başlığı** (*tape head*): Her an tek bir hücreyi gösterir. O hücredeki sembolü okur, yerine yeni bir sembol yazar ve bir hücre sola ya da sağa gider.
- **Durum ve kurallar:** Makinenin o anki durumu (q0, q1, ...) ile okunan sembol birlikte, ne yazılacağını, başlığın nereye gideceğini ve sonraki durumu belirler.



Şekil 1.2.11: Bir Turing makinesinin anlık görüntüsü. Bantta 1 0 1 □ yazılıdır; başlık 1 numaralı hücredeki 0'ı okuyor; makine q0 durumundadır.

Kural biçimi şöyledir:

> (durum, okunan) → (yazılacak, yön, yeni durum)

Örneğin (q0, 0) → (1, sağ, q0) kuralının anlamı şudur: makine q0 durumundayken başlığın altında 0 varsa o hücreye 1 yazılır, başlık bir sağa gider ve makine q0 durumunda kalır. Şekil 1.2.11'deki anlık görüntüde bu kural uygulanırsa 1 numaralı hücre 1 olur ve başlık 2 numaralı hücreye geçer.

Makinenin her adımı bu kadar basittir: bir sembol oku, bir sembol yaz, bir hücre git, durumu değiştir. Toplama, çarpma, karşılaştırma gibi bütün işlemler bu basit adımların uzun dizilerinden kurulur. Bant sonsuz olduğu için bellek sınırı yoktur. Gerçek bilgisayarlardan farkı da budur: gerçek bir bilgisayarın belleği sonludur.

Turing makinesinin bileşenleri bir Java programının bileşenlerine benzetilebilir. Bant, verinin tutulduğu bellektir; başlık, o an üzerinde çalışılan değişkeni gösterir; kural tablosu programdır; durum da programın o an hangi satırda olduğudur. Benzetme kusursuz değildir, ama bir Turing makinesinin de bir program yürüttüğünü görmeye yardım eder.

1.2.5.3 Örnek: bitleri ters çeviren makine

En basit Turing makinelerinden biri, banttaki her 0'ı 1, her 1'i 0 yapan makinedir. Makinenin yalnız bir çalışma durumu (q0) ve üç kuralı vardır. Kurallar Tablo 1.2.5'te verilmiştir.

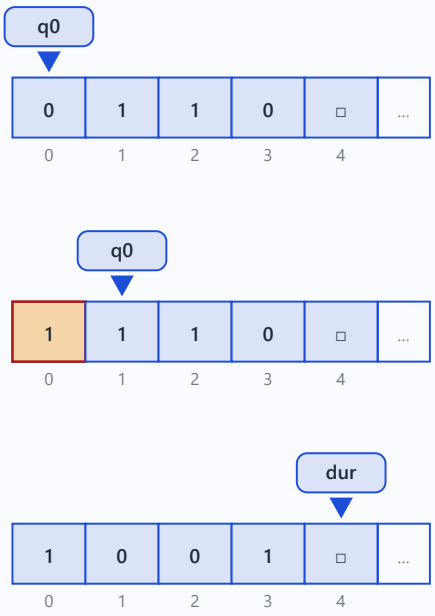
#	Durum, okunan	Yaz, git, durum
1	q0, 0	1, sağ, q0
2	q0, 1	0, sağ, q0
3	q0, □	dur

Tablo 1.2.5: Bitleri ters çeviren Turing makinesinin kural tablosu.

Kurallara göre 0 okunursa 1, 1 okunursa 0 yazılır ve her iki durumda başlık bir sağa gider. Boş hücreye gelince makine durur. Bantta 0110 yazılıyken makineyi çalıştıralım. Başlangıçta durum q0'dır ve başlık 0 numaralı hücrededir. Tablo 1.2.6 her adımı gösterir.

Adım	Bant (önce)	Başlık	Okunan	Kural	Bant (sonra)
1	**0**110□	0	0	1: 1 yaz, sağ	1110□
2	1**1**10□	1	1	2: 0 yaz, sağ	1010□
3	10**1**0□	2	1	2: 0 yaz, sağ	1000□
4	100**0**□	3	0	1: 1 yaz, sağ	1001□
5	1001□	4	□	3: dur	1001□

Tablo 1.2.6: Bitleri ters çeviren makinenin 0110 bandı üzerindeki çalışması. Başlığın okuduğu hücre kalın gösterilmiştir.



Şekil 1.2.12: Bitleri ters çeviren makineden üç an. (a) Başlangıç: bantta 0110, başlık 0. hücrede, durum q0. (b) 1. adımdan sonra: 0. hücreye 1 yazıldı, başlık 1. hücreye geçti. (c) Son: □ okundu, makine durdu; bantta 1001 kaldı.

Makine dört yazma adımı ve bir durma adımı, toplam beş adım çalışır. Bantta 1001 kalır. Adım sayısı girdinin uzunluğuna bağlıdır: *n* bitlik bir girdi için makine *n* yazma ve bir durma adımı, yani *n + 1* adım çalışır.

Bu küçük kural tablosu makinenin **programıdır**. Evrensel makine böyle bir tabloyu bantta okuyup çalıştırabilir. Aynı fikir Java'da bir döngüyle de yazılabilir: bir dizinin her öğesi için öğe 0 ise 1, 1 ise 0 yap ve bir sonraki öğeye geç. Turing makinesindeki “sağa git” adımı, döngüdeki indis artırımına karşılık gelir; boş hücrede durmak da dizinin sonuna gelince döngüden çıkmaya karşılık gelir.

Örnek 1.2.2

Aynı makine 1010 bandında çalıştırılsın. Adımlar: 0. hücrede 1 okunur, 0 yazılır (0010); 1. hücrede 0 okunur, 1 yazılır (0110); 2. hücrede 1 okunur, 0 yazılır (0100); 3. hücrede 0 okunur, 1 yazılır (0101); 4. hücrede □ okunur ve makine durur. Toplam 4 yazma adımı ve 1 durma adımı, yani 5 adım vardır. Bantta 0101 kalır.

Sık yapılan hata

Durma kuralını unutmak. Tablo yalnız 1. ve 2. kuralı içerseydi, makine boş hücreye geldiğinde hangi kuralı uygulayacağını bilemezdi. Bu tanımsız durumda makine takılır; doğru bir sonuç üretmeden kalır. Bir Java döngüsünde bitiş koşulunu unutmak da benzer bir hatadır: döngü dizinin sonunu aşar ve `ArrayIndexOutOfBoundsException` fırlatılır.

1.2.5.4 Hesaplanabilirlik ve durma problemi

Turing makinesi hesaplamanın ne olduğunu ve sınırlarını kesin olarak tanımlar. Bu tanımın kabul görmesini sağlayan iddia, Church–Turing tezidir.

Teorem 1.2.1 (Church–Turing tezi)

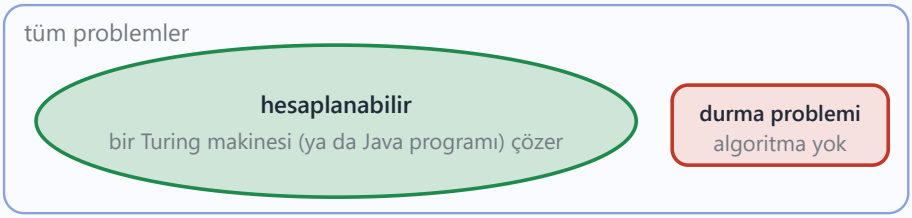
Adım adım bir yöntemle hesaplanabilen her şey bir Turing makinesiyle de hesaplanabilir.

Teorem kutusunda verilse de Church–Turing tezi kanıtlanmış bir matematik teoremi değildir. “Adım adım bir yöntem” sezgisel bir kavramdır ve kesin bir matematiksel tanımı yoktur. Tez, bu sezgisel kavramın Turing makinesiyle tam olarak yakalandığını ileri sürer. Bugüne kadar önerilen hesaplama modellerinin hiçbiri Turing makinesinden daha güçlü çıkmamıştır; tezin genel kabul görmesinin nedeni budur.

Tanım 1.2.16

Bir problem, her girdi için doğru yanıtı sonlu adımda veren bir algoritma (eşdeğer olarak bir Turing makinesi) varsa **hesaplanabilir** (*computable*) adını alır. **Hesaplanabilirlik** (*computability*), bir problemin herhangi bir algoritmayla çözülüp çözülemeyeceği sorusudur.

Turing'in 1936 makalesinin asıl şaşırtıcı sonucu, her problemin hesaplanabilir olmadığıdır. Bunun en ünlü örneği **durma problemi**dir (*halting problem*): verilen bir program, verilen bir girdiyle çalıştırılınca bir gün durur mu, yoksa sonsuza dek mi çalışır? Turing bu soruyu her program için doğru yanıtlayan bir algoritmanın olamayacağını kanıtladı.



Şekil 1.2.13: Tüm problemler içinde hesaplanabilir olanlar bir alt kümedir. Bu kümedeki problemleri bir Turing makinesi ya da bir Java programı çözer. Durma problemi kümenin dışındadır; onu çözen bir algoritma yoktur.

Kanıtın fikri, bir karşı örnek kurmaya dayanır ve birkaç cümleyle anlatılabilir. Varsayalım ki `durur(p, g)` adlı bir metot yazılabilsin. Bu metot, `p` programı `g` girdisiyle çalışınca durursa `true`, durmazsa `false` döndürsün ve kendisi her zaman dursun. Bu metodu kullanarak yeni bir program, `ters(p)`, yazılır. `ters`, `durur(p, p)` sorusunu sorar: yanıt `true` ise sonsuz bir döngüye girer, `false` ise hemen durur. Şimdi `ters` programı kendi kaynak koduyla çalıştırılsın. `ters(ters)` durursa, `durur(ters, ters)` `true` döner ve `ters` sonsuz döngüye girer; yani durmaz. Durmazsa, `durur` `false` döner ve `ters` hemen durur; yani durur. İki durum da çelişkiye varır. Çelişkinin tek kaynağı, `durur` metodunun yazılabileceği varsayımıdır; demek ki böyle bir metot yazılamaz.

Bu sonucun pratik bir anlamı vardır. Java derleyicisi bazı açık sonsuz döngüleri fark eder, ama her sonsuz döngüyü bulan bir derleyici yazılamaz. Bir programın her girdide durduğunu otomatik olarak gösteren genel bir araç da yoktur. Sınır, hızdan ya da bellekten gelmez; mantıksal bir sınırdır. Daha hızlı bir bilgisayar bu sınırı aşamaz. Bu sonuç 20. yüzyılın en önemli bilimsel sonuçlarından biri kabul edilir.

1.2.5.5 Turing-tamlık ve Java

Tanım 1.2.17

Bir Turing makinesinin yapabildiği her hesabı yapabilen bir programlama diline ya da sisteme **Turing-tam** (*Turing complete*) denir.

Java, Python ve C gibi diller Turing-tamdır. Bir Turing makinesinin yapabildiği her hesabı yapabilirler. Bunun için bir dilde koşullu dallanma ve sınırsız tekrar (döngü ya da özyineleme) ile istenen kadar veri tutma olanağının bulunması yeterlidir. Diller hız ve kolaylık bakımından birbirinden farklıdır, güç bakımından değil. Java'da yazılabilen her hesap Python'da da yazılabilir; yalnız biri daha kısa, öbürü daha hızlı olabilir.

Bu eşdeğerliğin bir sonucu da dil seçiminin bir “güç” sorunu olmamasıdır. Bir projeye Java'nın seçilmesinin nedenleri tip denetimi, taşınabilirlik ve kütüphanelerin zenginliği gibi mühendislik gerekçeleridir. Durma problemi de Turing-tam bütün diller için geçerlidir: Java'da da her programın durup durmadığına karar veren bir program yazılamaz.

1.2.6 Bilgisayarların sınırları

Durma problemi, hesaplamanın mantıksal sınırıdır: hiçbir bilgisayar onu aşamaz. Bunun yanında daha gündelik sınırlar da vardır. Hesaplanabilir bir problem bile, gereken zaman ya da bellek eldekinden fazlaysa pratikte çözülemez. Bu kesim fiziksel kaynakların sınırlarını ve bu sınırların bir Java programında nasıl göründüğünü ele alır.

1.2.6.1 Kaynak sınırları

Bilgisayarlar karmaşık hesaplamaları çok hızlı yapar; bu güç teknolojik gelişmelere dayanır. Yine de her kaynak sınırlıdır:

- **İşlemci hızı:** Hesapların ne kadar hızlı yapılacağını belirler. Bugünkü işlemciler saniyede milyarlarca işlem yapar, ama bu sayı da sonludur. Adım sayısı girdiyle çok hızlı büyüyen bir algoritma, en hızlı işlemcide bile yıllarca sürebilir.
- **Bellek:** Veri saklama kapasitesidir. Daha fazla bellek daha karmaşık görevler demektir, ama bellek de sınırlıdır. Turing makinesinin bandı sonsuzdur; gerçek bir bilgisayarın belleği değildir.
- **Veri işleme kapasitesi:** Büyük veri kümelerini işlemek zaman ve kaynak ister. Milyonlarca kaydı okumak bile diskin hızıyla sınırlıdır.
- **Enerji:** Bilgisayarlar sınırlı bir kaynak olan enerjiyi tüketir. Daha güçlü bilgisayar daha fazla enerji demektir. Büyük veri merkezleri küçük bir kentin tükettiği kadar elektrik harcayabilir.

Programcılar bu sınırları kaynakları etkin kullanarak aşmaya çalışır: aynı işi daha az adımda ve daha az bellekle yapan algoritmalar seçerler. EBOB'u bulmak için bütün sayıları tek tek denemek ile Öklid algoritmasını kullanmak arasındaki fark, bu seçimin iyi bir örneğidir.

1.2.6.2 Sınırlar Java'da da görünür

Bilgisayarın belleği sonlu olduğu için her sayı tipi de sonlu sayıda bitle saklanır. Bu sınır bazen beklenmedik sonuçlara yol açar. Aşağıdaki program üç tipik durumu gösterir.

```
public class Sinirlar {
    public static void main(String[] args) {
        int enBuyuk = Integer.MAX_VALUE;
        System.out.println(enBuyuk);
        System.out.println(enBuyuk + 1);  // taşma
        System.out.println(0.1 + 0.2);    // duyarlık
        System.out.println(5 / 2);        // tamsayı bölmesi
    }
}
```

Program şu çıktıyı üretir:

```
2147483647
-2147483648
0.30000000000000004
2
```

Dört satırı ayrı ayrı açıklamak gerekir.

Birinci ve ikinci satır: taşma. Java'da `int` tipi 32 bitlik bellek kullanır. Bitlerden biri işaret için ayrıldığı için en büyük değer $2^{31} - 1 = 2.147.483.647$'dir; bu değer `Integer.MAX_VALUE` sabitinde tutulur. Bu sayıya 1 eklenince sonuç 32 bite sığmaz. Java bu durumda hata vermez; sonuç sessizce en küçük değere, −2.147.483.648'e döner. Bu olaya **taşma** (*overflow*) denir.

> **Tanım 1.2.18**
>
> **Taşma** (*overflow*), bir işlemin sonucunun, sonucu tutan tipin değer aralığına sığmamasıdır. Java'da tam sayı tiplerinde taşma sessizdir: sonuç, aralığın öbür ucuna sarar ve hata fırlatılmaz.

Sararak dönmenin nedeni, negatif sayıların ikiye tümleyen (*two's complement*) biçiminde saklanmasıdır. 2.147.483.647 ikili tabanda `0111…1` (bir 0 ve otuz bir 1) olarak yazılır. Buna 1 eklenince elde zinciri bütün 1'leri 0 yapar ve en soldaki bit 1 olur: `1000…0`. İkiye tümleyende en soldaki bit işaret bitidir; bu bit örüntüsü $-2^{31} = -2.147.483.648$'i gösterir. Arabaların eski mekanik kilometre sayaçları da benzer davranır: 99.999'dan sonra 00 000'a döner.

Taşma, gerçek programlarda sinsi hatalara yol açar. `100000 * 100000` ifadesinin matematiksel değeri 10^{10}'dur, ama Java bu çarpımı `int` olarak yapar ve 1.410.065.408 sonucunu verir. İşlenenlerden biri `long` yapılırsa (`100000L * 100000`) sonuç doğru çıkar: 10.000.000.000. `long` 64 bittir ve en büyük değeri yaklaşık $9{,}2 \cdot 10^{18}$'dir. Taşmanın sessizce geçmesi istenmiyorsa `Math.addExact` ve `Math.multiplyExact` gibi metotlar kullanılır; bu metotlar taşma olduğunda `ArithmeticException` fırlatır ve "integer overflow" iletisini verir.

Üçüncü satır: duyarlık. `double` tipi ondalıklı sayıları ikili tabanda yaklaşık 15–16 anlamlı basamak duyarlıkla saklar. 0,1 sayısı ikili tabanda tam yazılamaz; tıpkı 1/3'ün onlu tabanda 0,333… diye sonsuza dek sürmesi gibi, 0,1 de ikili tabanda sonsuz bir kesirdir. Bilgisayar bu kesri bir yerde keser ve en yakın değeri saklar. 0,1 ile 0,2'nin yaklaşık değerleri toplanınca küçük hatalar birikir ve sonuç 0.30000000000000004 çıkar. Bu yüzden `0.1 + 0.2 == 0.3` ifadesi `false` verir. Ondalıklı sayılar `==` ile değil, aralarındaki farkın çok küçük olup olmadığına bakılarak karşılaştırılmalıdır: `Math.abs(a - b) < 1e-9`. Babil yöntemindeki döngü koşulu da bu fikirle yazılmıştır.

Dördüncü satır: tam sayı bölmesi. İki `int` değerin bölümü yine `int`'tir; kesirli kısım atılır. `5 / 2` ifadesi 2,5 değil 2 verir. Atma işlemi sıfıra doğru yapılır: `-7 / 2` ifadesi −3,5 değil −3 verir. Ondalıklı sonuç için işlenenlerden en az biri `double` olmalıdır: `5 / 2.0` ya da `5.0 / 2` ifadesi 2.5 verir.

Sıfıra bölme de iki tipte farklı davranır. `5 / 0` tam sayı bölmesidir ve `ArithmeticException: / by zero` fırlatır. `1.0 / 0` ise hata vermez; sonuç `Infinity` olur. `0.0 / 0` ifadesinin sonucu da "sayı değil" anlamına gelen `NaN` 'dır.

İfade	Sonuç	Neden
`Integer.MAX_VALUE + 1`	`-2147483648`	Taşma: en küçük değere sarar.
`100000 * 100000`	`1410065408`	Taşma: 10^{10}, 32 bite sığmaz.
`100000L * 100000`	`10000000000`	`long` 64 bittir.
`0.1 + 0.2`	`0.30000000000000004`	0,1 ikili tabanda tam yazılamaz.
`5 / 2`	`2`	Tam sayı bölmesi.
`-7 / 2`	`-3`	Sıfıra doğru kesilir.
`5 / 2.0`	`2.5`	Bir işlenen `double`.
`5 / 0`	`ArithmeticException`	Tam sayıda sıfıra bölme.
`1.0 / 0`	`Infinity`	`double` 'da sıfıra bölme hata değildir.

Tablo 1.2.7: Java'da sayı tiplerinin sınırlarına örnekler (JDK 21 ile çalıştırılarak doğrulanmıştır).

Sık yapılan hata

Sonucu `double` bir değişkene yazmanın bölmeyi ondalıklı yaptığını sanmak. `double ort = 205 / 3;` satırında önce `205 / 3` tam sayı olarak hesaplanır ve 68 bulunur; ardından bu 68, 68.0'a çevrilip değişkene yazılır. Kaybolan kesir geri gelmez. Doğrusu `205 / 3.0` ya da `(double) 205 / 3` yazmaktır.

1.2.6.3 Makineler ve teknoloji

Teknoloji insanların yaşamını etkileyen bir güçtür; makineler bu gelişmenin önemli bir parçasıdır. Makineler işleri hızlı ve verimli yapar. Bu güç sanayi devriminden bugüne sürekli arttı. Yine de inşa edilebilecek makinelerin sınırları vardır:

- **Fiziksel sınırlar:** Boyut, ağırlık ve dayanıklılık fizik yasalarına bağlıdır. Bir transistör birkaç atomdan küçük yapılamaz. Sinyal ışıktan hızlı gidemez; 3 GHz'lik bir işlemcide bir saat çevrimi sırasında ışık yalnız yaklaşık 10 cm yol alır. Yonganın boyutu bu yüzden hızı da sınırlar.
- **Enerji sınırları:** Makineler enerjiye ihtiyaç duyar ve enerji kaynakları sınırlıdır. Daha güçlü makine daha fazla enerji ve daha fazla soğutma ister. Harcanan elektrik enerjisinin büyük bölümü ısıya dönüşür; bu ısı uzaklaştırılamazsa yonga zarar görür.
- **Teknolojik sınırlar:** Yeni malzemeler, yeni tasarım fikirleri ve yeni algoritmalar sınırları zorlayabilir. Vakum tüpünden transistöre geçiş böyle bir sınırın aşılmasıydı.

Sınırları anlamak, daha iyi ve verimli makineler tasarlamayı sağlar. Yazılımda bunun karşılığı, problemi daha az kaynakla çözen algoritmalardır. Donanım belirli bir hızın ötesine geçemiyorsa, aynı işi daha az adımda yapan bir algoritma çoğu zaman daha hızlı bir makineden daha büyük kazanç sağlar.

1.2.7 Bildirimsel ve buyurusal bilgi

Bir şeyi bilmenin iki yolu vardır. Bir karekökün ne olduğunu bilmek ile bir sayının karekökünü bulabilmek aynı şey değildir. Birincisi tanımı bilmek, ikincisi yöntemi bilmektir. Bilgisayara bir iş yaptırmak için ikinci tür bilgi gerekir. Bu kesim iki bilgi türünü ayırır ve her ikisini iki klasik problem üzerinde gösterir: karekök ve en büyük ortak bölen.

1.2.7.1 İki bilgi türü

Bilgi, insanların ya da makinelerin dünyayı anlamasına ve sorunları çözmesine yardım eden bilinçli kavrayıştır. İki biçimde ifade edilebilir: bildirimsel ve buyurusal. Tablo 1.2.8 iki türü karşılaştırır.

	Bildirimsel bilgi	**Buyurusal bilgi**
Soru	Bir şey **nedir**?	Bir şey **nasıl yapılır**?
Biçim	Tanım, aksiyom, gerçek	Sıralı talimatlar, tarif
Sınanması	Doğru ya da yanlış olarak doğrulanır	Adımlar izlenir, sonuç elde edilir
Karekök	$y \cdot y = x$ ise y, x'in karekökü dür	Babil yöntemi: tahmin et, iyileştir, tekrarla
Günlük örnek	"Kek, un, şeker ve yumurtadan yapılan bir tatlıdır."	Kek tarifi: "Yumurtaları çırp, unu ekle, 180 °C'de 35 dk pişir."

Tablo 1.2.8: Bildirimsel ve buyurusal bilginin karşılaştırması.

Bilgisayara buyurusal bilgi gerekir: programlar "nasıl yapılır" bilgisini yazar. Kek tanımını bilen bir aşçı yamağı kek yapamaz; tarife ihtiyacı vardır. Bilgisayar da tanımdan yola çıkarak bir şey üretemez, ona adımlar verilmelidir. Bildirimsel bilgi ise boşa gitmez: sonucun doğru olup olmadığını sınamaya yarar. Tarifle yapılan şeyin gerçekten kek olup olmadığına tanıma bakılarak karar verilir.

1.2.7.2 Bildirimsel bilgi

Tanım 1.2.19

Bildirimsel bilgi (*declarative knowledge*), açıkça ifade edilen ve doğru ya da yanlış olarak doğrulanabilen bilgidir. Aksiyomlar ya da tanımlar yoluyla bir şeyin ne olduğunu belirtir. Gerçeklerin, yani dünya hakkında doğru ve doğrulanabilir ifadelerin dile getirilmesini de içerir.

“Su deniz düzeyinde 100 °C'de kaynar” bir gerçektir; bildirimsel bir bilgidir. Karekökün tanımı da bildirimseldir:

> y, x'in kareköküdür $\Leftrightarrow y \cdot y = x$ ve $y \geq 0$.

Bu tanımla bir tahmin sınanabilir. $4 \cdot 4 = 16$ ve $4 \geq 0$ olduğundan 4, 16'nın kareköküdür. $5 \cdot 5 = 25 \neq 16$ olduğundan 5 değildir. Tanımdaki $y \geq 0$ koşulu da gereklidir: $(-4) \cdot (-4) = 16$ olsa da -4, 16'nın karekökü sayılmaz.

Ama tanım, karekökün nasıl hesaplanacağını söylemez. $\sqrt{2}$'yi bu tanımdan doğrudan bulmak mümkün değildir. Tanıma göre $\sqrt{2}$, kendisiyle çarpılınca 2 eden sayıdır; bu sayının ne olduğunu bulmak için bir yöntem gerekir. Tanımın yapabildiği tek şey, önerilen bir adayı sınamaktır: $1{,}4 \cdot 1{,}4 = 1{,}96$ olduğu için $1{,}4$ $\sqrt{2}$ değildir, ama ona yakındır.

1.2.7.3 Buyurusal bilgi: Babil yöntemi

Tanım 1.2.20

Buyurusal bilgi (*imperative knowledge*), bir şeyin nasıl yapılacağına dair talimatlardan oluşan bilgidir. Sıralı adımlar biçiminde verilir; adımlar izlenince sonuç elde edilir.

Karekök için buyurusal bilgiye bir örnek **Babil yöntemi**dir. Yöntem, bir sayının karekökünü adım adım bulur:

1. x değerini al.
2. Bir başlangıç tahmini y_0 seç (ör. $y_0 = 1$).
3. $y_n^2 \approx x$ ise dur; cevap y_n'dir.
4. Değilse yeni tahmini hesapla: $y_{n+1} = (y_n + x / y_n) / 2$.
5. Adım 3'e dön.

Yöntemin neden çalıştığı basit bir gözlemle açıklanır. y gerçek kökten büyükse x / y ondan küçüktür; y küçükse x / y büyüktür. Çünkü $y \cdot (x / y) = x$'tir ve iki çarpanın ikisi birden kökten büyük ya da küçük olamaz. Gerçek kök her zaman y ile x / y arasındadır. İkisinin ortalaması bu aralığın ortasıdır ve köke iki uçtan da daha yakındır. Her tekrarda tahmin köke yaklaşır.

Yakınsama çok hızlıdır. İlk tahmin köke yeterince yakınsa, her adımda doğru basamak sayısı yaklaşık iki katına çıkar. Bunun nedeni, yeni tahminin hatasının eski hatanın karesiyle orantılı olmasıdır: hata 0,01 ise bir sonraki adımda yaklaşık 0,0001 düzeyine iner. Tablo 1.2.9 $x = 16$ ve $y_0 = 1$ için ilk beş tekrarı gösterir.

n	y_n	y_n^2
0	1,000000	1,000000

n	y_n	y_n^2
1	8,500000	72,250000
2	5,191176	26,948313
3	4,136665	17,111995
4	4,002258	16,018065
5	4,000001	16,000005

Tablo 1.2.9: Babil yöntemiyle $\sqrt{16}$'nın hesaplanması (y_0 = 1). Değerler altı basamağa yuvarlanmıştır.

Tablonun ilk satırları yöntemin bir özelliğini gösterir. Başlangıç tahmini 1, gerçek kökün (4) çok altındadır; bu yüzden $x / y = 16$ çok büyüktür ve ilk ortalama 8,5 olur. Tahmin bir anda kökün üstüne sıçrar. Bundan sonra bütün tahminler kökün üstünde kalır ve her adımda ona yaklaşır. Yakınsama ilk adımlarda yavaştır, köke yaklaştıkça hızlanır: 3. adımda hata yaklaşık 0,14, 4. adımda 0,002, 5. adımda 0,0000006'dır. Beş tekrarda 4'e çok yakın bir değer bulunur.

Örnek 1.2.3

$\sqrt{2}$, $y_0 = 1$ alınarak hesaplansın.

$y_1 = (1 + 2/1) / 2 = 1{,}5$; $y_1^2 = 2{,}25$.
$y_2 = (1{,}5 + 2/1{,}5) / 2 = (1{,}5 + 1{,}333333) / 2 \approx 1{,}416667$; $y_2^2 \approx 2{,}006944$.
$y_3 = (1{,}416667 + 2/1{,}416667) / 2 \approx 1{,}414216$; $y_3^2 \approx 2{,}000006$.

Gerçek değer $\sqrt{2} \approx 1{,}414214$'tür. Üç tekrarda beş ondalık basamak doğru bulunmuştur. Tanımın $\sqrt{2}$ için yapamadığını yöntem birkaç işlemle yapar.

1.2.7.4 Babil yöntemi Java ile

Babil yönteminin adımları doğrudan Java'ya çevrilebilir. "$y^2 \approx x$ ise dur" adımı, $|y \cdot y - x|$ farkı küçük bir eşikten (burada 0,001) büyük olduğu sürece devam eden bir `while` döngüsüne dönüşür.

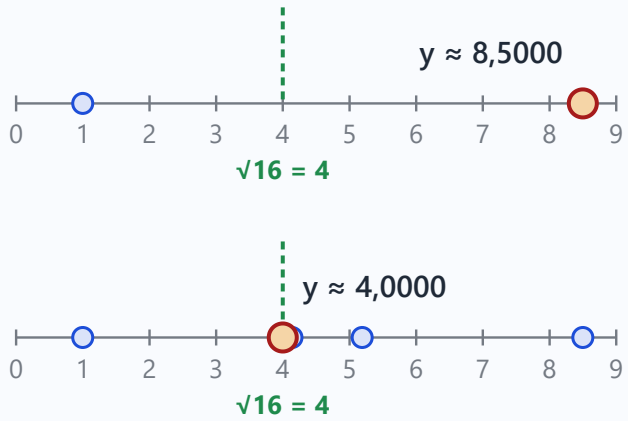
```
double x = 16;   // karekökü aranan sayı
double y = 1;    // başlangıç tahmini y0
while (Math.abs(y * y - x) > 0.001) {
    y = (y + x / y) / 2;   // yeni tahmin
    System.out.println(y);
}
System.out.println("Karekök ≈ " + y);
```

Kodun satırları algoritmanın adımlarına şöyle karşılık gelir:

- `double x = 16;` adım 1'dir. `x` ondalıklı tiptedir; Java 16 tam sayısını otomatik olarak 16.0'a çevirir.
- `double y = 1;` adım 2'dir: başlangıç tahmini.
- `while (Math.abs(y * y - x) > 0.001)` adım 3'ün tersidir. "Yaklaşık eşitse dur" yerine "yaklaşık eşit değilse devam et" yazılır. `Math.abs` mutlak değeri verir; fark negatif de olabileceği için gereklidir.
- `y = (y + x / y) / 2;` adım 4'tür. Eski `y` değeri sağ tarafta kullanılır, yenisi aynı değişkene yazılır.
- Döngünün sonuna gelince denetim yeniden `while` satırına döner; bu da adım 5'tir.

Tur	Koşuldaki $\lvert y \cdot y - x \rvert$	Koşul	Yeni `y` (yazdırılan)
1	15	doğru	`8.5`
2	56,25	doğru	`5.1911764705882355`
3	10,9483	doğru	`4.136664722546242`
4	1,112	doğru	`4.002257524798522`
5	0,0180653	doğru	`4.000000636692939`
–	$\approx 5{,}1 \cdot 10^{-6}$	yanlış	Döngü biter; `Karekök ≈ 4.000000636692939` yazılır.

Tablo 1.2.10: Babil yöntemi programının izlemesi (`x` = 16.0). Yazdırılan değerler Java'nın verdiği tam değerlerdir.



Şekil 1.2.14: Babil yönteminin tahminleri sayı doğrusu üzerinde. (a) İlk tahmin 1'den sonra yeni tahmin 8,5'e sıçrar. (b) Son durum: tahminler 1, 8,5, 5,191, 4,137, 4,002 ve 4,000 sırasıyla √16 = 4'e yaklaşır.

Program tam 4 değil, 4'e çok yakın bir yaklaşık değer yazar. Bunun iki nedeni vardır. Birincisi, döngü fark 0,001'in altına inince durur; kesin eşitlik beklenmez. İkincisi, `double` zaten sonlu duyarlıklıdır. Eşik daha küçük seçilirse (ör. `1e-12`) sonuç daha da yakın olur ve döngü yalnız bir iki tur daha döner.

Sık yapılan hata

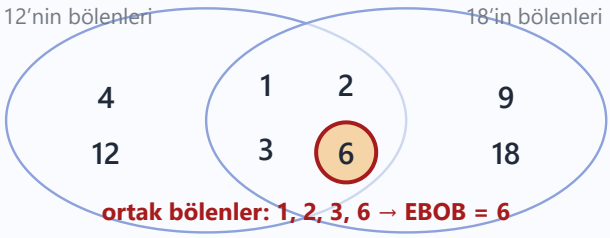
Döngü koşulunu `while (y * y != x)` biçiminde yazmak. `double` değerleri çoğu zaman tam eşitliğe ulaşmaz; $\sqrt{2}$ gibi bir kök ikili tabanda tam yazılamadığı için `y * y` hiçbir zaman tam olarak 2.0 olmayabilir. Bu durumda koşul hiç yanlış olmaz ve döngü sonsuza dek döner. Ondalıklı karşılaştırmada her zaman bir eşik kullanılır.

1.2.7.5 EBOB: iki tanım

Tanım 1.2.21

İki tam sayının **en büyük ortak böleni** (EBOB; *greatest common divisor, GCD*), ikisini de tam bölen en büyük pozitif tam sayıdır.

Örneğin 12'nin bölenleri 1, 2, 3, 4, 6, 12; 18'in bölenleri 1, 2, 3, 6, 9, 18'dir. Ortak bölenler 1, 2, 3 ve 6'dır; en büyüğü 6 olduğu için EBOB(12, 18) = 6'dır.



Şekil 1.2.15: 12'nin ve 18'in bölenleri. Kesişim kümesi 1, 2, 3, 6'dır; en büyüğü olan 6, EBOB'dur.

EBOB'un da iki tanımı vardır. **Bildirimsel tanım** EBOB'un ne olduğunu söyler. d, a ile b'nin ortak bölenidir; yani bazı x ve y tam sayıları için $a = d \cdot x$ ve $b = d \cdot y$'dir. Ayrıca bu koşulu sağlayan sayıların en büyüğüdür. Bu tanım bir adayı sınamaya yarar: 6, 12'yi de 18'i de böler ve daha büyük ortak bölen yoktur. Ama tanım EBOB'u bulma yolunu vermez. Şekil 1.2.15'teki gibi bütün bölenleri listelemek küçük sayılarda işe yarar, büyük sayılarda çok yavaştır.

Buyurusal tanım ise EBOB'u nasıl bulacağımızı söyler: Öklid algoritması. Bölme ve kalan almayı, kalan 0 olana kadar tekrarlar. Yaklaşık MÖ 300'de Öklid'in *Elemanlar* kitabında yer alan bu yöntem, bilinen en eski algoritmalardan biridir.

1.2.7.6 Öklid algoritması

Algoritmanın adımları şöyledir:

1. İki pozitif tam sayı a ve b'yi al ($a \geq b$).

2. a'yı b'ye böl, kalana r de: $r = a \bmod b$.
3. $r = 0$ ise dur: EBOB = b.
4. Değilse a'ya b'yi, b'ye r'yi ata.
5. Adım 2'ye dön.

Algoritmanın doğruluğu tek bir gözleme dayanır.

Teorem 1.2.2 (Öklid)

a ve b pozitif tam sayılar ve $r = a \bmod b$ ise EBOB(a, b) = EBOB(b, r)'dir.

İspat. Bölme işlemine göre bir k tam sayısı için $a = k \cdot b + r$, yani $r = a - k \cdot b$'dir. a'yı ve b'yi bölen her d sayısı, $a - k \cdot b$ farkını, yani r'yi de böler. Tersine, b'yi ve r'yi bölen her d, $k \cdot b + r$ toplamını, yani a'yı da böler. Demek ki (a, b) çiftinin ortak bölenleri ile (b, r) çiftinin ortak bölenleri aynı kümedir. Aynı kümenin en büyük elemanı da aynıdır. ■

Teorem, algoritmanın her adımda neyi koruduğunu söyler: sayılar küçülür ama EBOB değişmez. Kalan her zaman bölenden küçük olduğu için ($r < b$) sayılar her adımda kesin olarak küçülür ve bir gün kalan 0 olur. Kalan 0 olduğunda b, a'yı tam böler; bu durumda EBOB$(a, b) = b$'dir. Teorem sayesinde bu sonuç baştaki sayıların da EBOB'udur.

a	b	$r = a \bmod b$	Bölme
48	18	12	$48 = 2 \cdot 18 + 12$
18	12	6	$18 = 1 \cdot 12 + 6$
12	**6**	0	$12 = 2 \cdot 6 + 0$

Tablo 1.2.11: Öklid algoritmasıyla EBOB(48, 18).

Üç bölmeden sonra kalan 0 olur ve EBOB(48, 18) = 6 bulunur. Bölenleri listeleme yöntemiyle aynı sonuç, 48'in on, 18'in altı böleni yazılarak bulunurdu. Fark büyük sayılarda belirginleşir. Öklid algoritmasında ilk sayı en geç iki adımda bir yarıya iner. $b \le a/2$ ise kalan b'den, dolayısıyla $a/2$'den küçüktür. $b > a/2$ ise kalan $a - b$'dir ve bu da $a/2$'den küçüktür. Bu yüzden milyarlık sayılarda bile adım sayısı birkaç düzineyi geçmez.

Örnek 1.2.4

EBOB(1071, 462) bulunsun.

$1071 = 2 \cdot 462 + 147$, yani $r = 147$. $(a, b) = (462, 147)$ olur.
$462 = 3 \cdot 147 + 21$, yani $r = 21$. $(a, b) = (147, 21)$ olur.
$147 = 7 \cdot 21 + 0$, yani $r = 0$.

Algoritma durur ve EBOB = 21 bulunur. Üç bölme yapılmıştır. Sağlaması: $1071 = 21 \cdot 51$ ve $462 = 21 \cdot 22$'dir; 51 ile 22'nin 1'den büyük ortak böleni yoktur.

1.2.7.7 Öklid algoritması Java ile

```
int a = 48, b = 18;   // a >= b
int r = a % b;        // kalan
while (r != 0) {
    a = b;
    b = r;
    r = a % b;
}
System.out.println("EBOB = " + b);
```

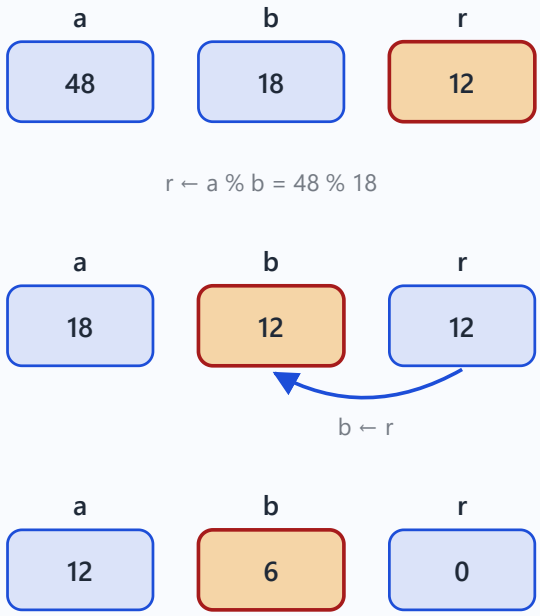
Java'daki `%` işleci kalanı verir; algoritmadaki "mod" işlemine karşılık gelir. Kod, algoritmanın adımlarını şöyle izler:

- İlk iki satır adım 1 ve 2'dir: sayılar alınır ve ilk kalan hesaplanır.
- `while (r != 0)` adım 3'ün tersidir: kalan 0 değilse devam edilir.
- Döngü gövdesindeki ilk iki satır adım 4'tür. Sıra önemlidir: önce `a = b`, sonra `b = r` yazılır. Ters sırada yazılsaydı `b`'nin eski değeri kaybolurdu.
- `r = a % b;` satırı adım 2'nin tekrarıdır; denetim yeniden `while` satırına döner.
- Döngü bittiğinde son bölen `b`'dedir ve sonuç olarak yazdırılır.

Adım	Deyim	a	b	r
1	`int a = 48, b = 18;`	**48**	**18**	–
2	`r = a % b;`	48	18	**12**
3	`r != 0` → doğru	48	18	12
4	`a = b;`	**18**	18	12
5	`b = r;`	18	**12**	12
6	`r = a % b;`	18	12	**6**
7	`r != 0` → doğru	18	12	6
8	`a = b;`	**12**	12	6
9	`b = r;`	12	**6**	6
10	`r = a % b;`	12	6	**0**
11	`r != 0` → yanlış	12	6	0
12	`println`	12	6	0

Tablo 1.2.12: Öklid programının izlemesi (`a` = 48, `b` = 18). Değişen hücre kalın gösterilmiştir.

Program “EBOB = 6” yazar. İzlemede 4. ve 8. adımlarda `a` ile `b`'nin bir an aynı değeri taşıdığı görülür (18 ve 18, 12 ve 12). Bu bir hata değildir; atamalar sırayla yapıldığı için ara bir durumdur ve bir sonraki satırda `b` güncellenir.



Şekil 1.2.16: Öklid programının izlemesinden üç an. (a) İlk kalan: 48 % 18 = 12. (b) Kaydırmadan sonra `a` = 18, `b` = 12. (c) Son: `r` = 0, döngü biter; son bölen `b` = 6 sonuçtur.

Kod *a ≥ b* koşulunu varsayar, ama bu koşul sağlanmasa da doğru çalışır. `a = 18, b = 48` ile başlanırsa ilk kalan 18 % 48 = 18 olur. İlk tur yalnız iki sayının yerini değiştirir (`a` = 48, `b` = 18) ve algoritma bir bölme fazlasıyla aynı sonucu bulur: dört bölme, EBOB = 6.

Sık yapılan hata

Sonuç olarak `a`'yı yazdırmak. Döngü bittiğinde `a` = 12'dir ve 12, EBOB değildir. EBOB, kalanı 0 yapan bölendir; yani döngü sonunda `b`'de durur. Bir başka hata da `b`'nin 0 olabileceğini unutmaktır: `b = 0` ile başlanırsa ilk satırdaki `a % b` işlemi `ArithmeticException: / by zero` fırlatır.

1.2.8 Algoritma ve problem

Babil yöntemi ve Öklid algoritması, buyurusal bilginin iki örneğidir. İkisinin ortak özelliği, bir problemi kesin adımlarla çözmeleridir. Bu tür adım dizilerine algoritma denir. Algoritma kavramı bilgisayar biliminin merkezindedir; programlama da büyük ölçüde algoritmaları bir dille ifade etmektir.

1.2.8.1 Algoritma

Tanım 1.2.22

Algoritma (*algorithm*), bir problemi adım adım çözme yöntemidir; izlenmesi gereken adımları belirleyen bir tariftir. Bir algoritma:

- adımları açıkça belirler,
- adımları mantıklı bir sıraya koyar,
- sonlu sayıda adımda biter,
- problemi çözme amacı güder.

Algoritma bir görevin nasıl yapılacağını belirler ve karmaşık problemleri basitleştirmeye yarar. Bilgisayar programlarının temelidir: **programlama**, algoritmaları bir programlama diliyle ifade etme sürecidir. Bilgisayar bilimi de büyük ölçüde algoritmaların tasarımı ve analizinden oluşur. İyi bir programcı, problem için uygun algoritmayı seçip uygulayabilen kişidir.

Tanımdaki dört özellik birer birer önemlidir.

- **Açıklık:** Her adım tek bir biçimde anlaşılmalıdır. “Sayıyı biraz büyüt” açık bir adım değildir; “sayıya 1 ekle” açıktır. Bilgisayar yorum yapmadığı için açık olmayan bir adımı yürütemez.
- **Sıra:** Adımların sırası sonucu değiştirir. Öklid algoritmasında `a = b` ile `b = r` atamalarının sırası değişirse algoritma yanlış sonuç verir. Kek tarifinde de pişirmeden önce un eklenir.
- **Sonluluk:** Algoritma her girdi için bir gün durmalıdır. “1 ekle ve yazdır; bunu hiç durmadan tekrarla” talimatı açık ve sıralıdır, ama bitmediği için algoritma değildir. Babil yönteminin durduğunu fark eşiği, Öklid algoritmasının durduğunu da kalanın her adımda küçülmesi güvence altına alır.
- **Amaç:** Algoritma belirli bir problemi çözmek için yazılır ve doğru sonucu vermelidir.

İyi tasarlanmış bir algoritma hem **doğru** hem **verimli** olmalıdır. Doğruluk, her geçerli girdi için doğru sonucu vermesidir. Verimlilik ise bu sonucu makul bir zamanda ve makul bir bellekle bulmasıdır. EBOB için bölenleri listeleme yöntemi doğrudur ama verimsizdir; Öklid algoritması hem doğru hem verimlidir.

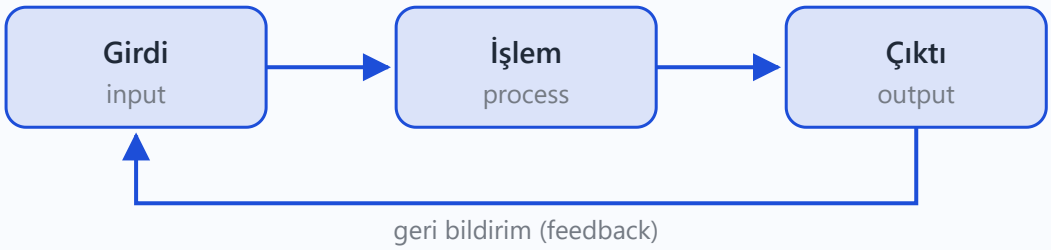
“Algoritma” sözcüğünün kökeni 9. yüzyılda Bağdat'ta yaşamış matematikçi el-Hârizmî'nin adıdır. Onun Hint-Arap rakamlarıyla hesap yapmayı anlatan kitabının Latince çevirisinde adı “Algoritmi” diye yazıldı. Sözcük zamanla, kuralları belli bir hesap yöntemi anlamını kazandı.

Algoritmalar çoğu zaman bir programlama dilinde değil, **sözde kodla** (*pseudocode*) yazılır. Sözde kod, algoritmayı belirli bir dile bağlı kalmadan, yarı doğal dille anlatma biçimidir. Noktalı virgül, tip bildirimi gibi ayrıntılarla uğraşmaz; adımların mantığına odaklanır. Bu notta kullanılan `while … do`, `set … to`, `repeat … until` gibi kalıplar sözde kodun yaygın biçimleridir.

1.2.8.2 Girdi, işlem, çıktı

Bir algoritma yazılmadan önce problemin kendisi açıkça tanımlanmalıdır. Programın ya da algoritmanın ne yapması gerektiği belirlenir; gerekli adımlar ve çıktı ortaya konur. Problem belirlemenin iki bileşeni vardır:

- **Girdi** (*input*): Problemin çözülmesi için gereken bilgi ya da veri.
- **Çıktı** (*output*): Problemin sonucu ya da istenen bilgi.



Şekil 1.2.17: Girdi, işlem ve çıktı kutuları soldan sağa oklarla bağlıdır. Çıktıdan girdiye giden ok geri bildirimi gösterir.

Şekil 1.2.17'deki düzen, hesaplamanın üç aşamasının problem tanımına uygulanmış biçimidir. Girdi ve çıktı belirlenince işlem, ikisini bağlayan algoritma olur. Öklid algoritması için girdi iki pozitif tam sayı, çıktı onların EBOB'udur. Babil yöntemi için girdi bir pozitif sayı ve bir başlangıç tahmini, çıktı karekökün yaklaşık değeridir.

Şekildeki geri dönüş oku **geri bildirimi** (*feedback*) gösterir: çıktıya bakarak girdiyi ya da işlemi düzeltmek. Termostat iyi bir örnektir. Sıcaklığı ölçer (girdi), istenen değerle karşılaştırır (işlem), ısıtıcıyı açar ya da kapatır (çıktı). Isıtıcının açılması odanın sıcaklığını, yani bir sonraki girdiyi değiştirir. Elektrik-elektronik mühendisliğindeki kapalı çevrim kontrol sistemleri de bu düzenle çalışır. Babil yöntemi de bir tür geri bildirimdir: her tahminin karesi x ile karşılaştırılır ve sonuç bir sonraki tahmini belirler.

1.2.8.3 Belirsiz bir isteği netleştirmek

Problemi tanımlarken altı soru yardımcı olur: *Ne? Kim? Nerede? Ne zaman? Nasıl? Neden?* Bu sorular özellikle günlük dille verilmiş belirsiz istekleri netleştirmeye yarar.

“Not ortalaması hesapla” isteği belirsizdir. Program yazılmadan önce en az üç sorunun yanıtı bilinmelidir:

- Kaç not var? Sayı sabit mi (üç not), yoksa kullanıcı mı girecek?
- Notların ağırlıkları eşit mi? Vize %40, final %60 ise basit ortalama yanlış sonuç verir.
- Sonuç tam sayı mı, ondalıklı mı olacak? Yuvarlama yapılacak mı?

Bu sorular yanıtlanmadan yazılan program yanlış problemi çözebilir. Program hatasız derlenir, hatasız çalışır, ama istenen şeyi yapmaz. Bu tür hataları bulmak, derleme hatalarını bulmaktan çok daha zordur; çünkü derleyici hiçbir uyarı vermez.

Örnek 1.2.5

Üç notun ortalamasını bulan programın girdisi, çıktısı ve sözde kodu yazılsın. Notlar 40, 75 ve 90 olsun.

Girdi: üç tam sayı (`n1` , `n2` , `n3`). **Çıktı:** bir ondalıklı sayı, ortalama. **Sözde kod:** üç notu oku → toplamı 3.0'a böl → sonucu yaz.

Toplam 40 + 75 + 90 = 205'tir. Java'da `205 / 3.0` ifadesi `68.33333333333333` verir. Bölen 3 yazılsaydı tam sayı bölmesi yapılır ve sonuç 68 olurdu. Çıktının "ondalıklı" olarak belirlenmesi, bu ayrıntıyı baştan doğru kurmayı sağlar.

Örnek 1.2.6

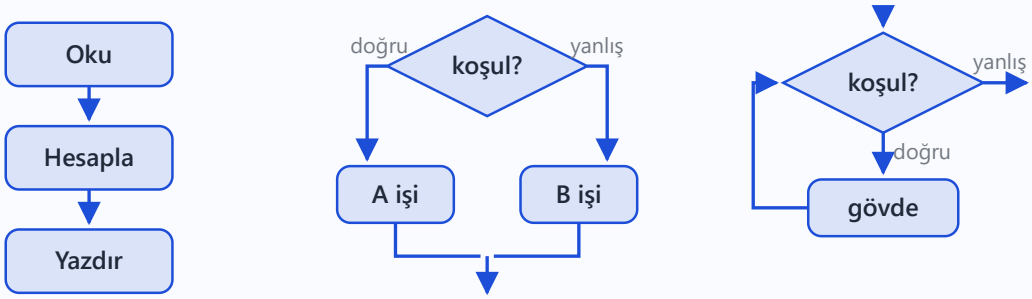
Ders notu "vize × 0,4 + final × 0,6" ile hesaplansın. Girdi iki tam sayıdır (vize ve final), çıktı bir ondalıklı sayıdır. Vize 50 ve final 80 için ortalama 50 · 0,4 + 80 · 0,6 = 20 + 48 = 68'dir. Java'da `50 * 0.4 + 80 * 0.6` ifadesi `68.0` verir. Aynı iki not için basit ortalama 65 olurdu; ağırlıkların sorulmaması 3 puanlık bir hataya yol açardı.

1.2.9 Algoritmik işlemler

Babil yöntemi, Öklid algoritması ve not ortalaması farklı problemleri çözer, ama hepsi aynı üç tür işlemden kurulur. Bu üç tür her programlama dilinde bulunur ve algoritmaların temel yapısını oluşturur.

1.2.9.1 Üç işlem türü

- **Sıralı işlem** (*sequential*): Her işlem bir öncekinden sonra, sırayla çalışır. Örneğin veriyi oku, işle, sonucu üret.
- **Koşullu işlem** (*conditional*): Koşul sağlanıyorsa bir işlem, sağlanmıyorsa başka bir işlem yapılır. Örneğin sayı pozitif mi?
- **Döngülü işlem** (*iterative*): Bir işlem, koşul sağlandığı sürece tekrarlanır. Örneğin bir listedeki bütün sayıları toplamak.



Şekil 1.2.18: Üç işlem türünün akış şemaları. (a) Sıralı: Oku, Hesapla, Yazdır kutuları alt alta. (b) Koşullu: koşul doğruysa A işi, yanlışsa B işi yapılır; sonra yollar birleşir. (c) Döngülü: koşul doğru oldukça gövde çalışır ve koşula geri dönülür; koşul yanlış olunca döngüden çıkılır.

Şekil 1.2.18'deki üç yapı birbirinin içine yerleştirilerek bütün algoritmalar kurulabilir. Öklid algoritmasının gövdesi sıralı üç atamadır; bu gövde bir döngünün içindedir; döngünün devam edip etmeyeceğine bir koşul karar verir. Java'da sıralı işlem deyimleri alt alta yazmakla, koşullu işlem `if` / `else` ile, döngülü işlem `while`, `do-while` ve `for` ile yazılır.

Bu üç türün karmaşıklığı denetlemeye yaraması da önemlidir. Bir programın akışı yalnız bu üç yapıyla kurulursa her parçanın tek bir girişi ve tek bir çıkışı olur. Program yukarıdan aşağıya okunabilir ve izlenebilir. Bu fikir, yapısal programlama adıyla 1960'lı ve 1970'li yıllarda yerleşti ve Java dahil bugünkü dillerin hepsini etkiledi.

1.2.9.2 while: önce koşul

Döngülü işlem, devam koşulu yanlış olana kadar belirli eylemleri tekrar eder. Listeleri gezmek ve verileri işlemek için kullanışlıdır. Döngünün iki parçası vardır: **devam koşulu** ve **gövde**. Koşulun gövdeden önce mi sonra mı sınandığı, iki farklı döngü türü ortaya çıkarır.

İlk türde koşul önce sınanır. Sözde kodda bu yapı `while … do` ile yazılır. Aşağıdaki örnek, *i > 0* olduğu sürece *s*'yi 1 artırır ve *i*'yi 1 azaltır.

```
while i > 0 do
    set s to s + 1
    set i to i - 1
```

Java karşılığı neredeyse birebirdir:

```
int s = 0, i = 3;
while (i > 0) {
    s = s + 1;
    i = i - 1;
}
```

Tur	Koşul `i > 0`	`s` (tur sonu)	`i` (tur sonu)
Başlangıç	–	0	3
1	3 > 0 doğru	1	2
2	2 > 0 doğru	2	1
3	1 > 0 doğru	3	0
–	0 > 0 yanlış	3	0

Tablo 1.2.13: `while` döngüsünün izlemesi (`s` = 0, `i` = 3).

Döngü üç tur döner ve sonunda `s` = 3, `i` = 0 olur. Koşul dört kez sınanır: üçünde doğru, sonuncusunda yanlış çıkar. Genel olarak `i` , *n* ile başlarsa döngü *n* tur döner ve `s` sonunda *n* olur. Koşul önce sınandığı için `i` baştan 0 ise gövde hiç çalışmaz.

1.2.9.3 repeat ... until ve do ... while

İkinci türde gövde önce çalışır, koşul sonra sınanır. Sözde kodda bu yapı `repeat … until` ile yazılır. Aşağıdaki örnek, *k* değeri *n*'den büyük olana kadar *k*'yi yazdırır ve 1 artırır.

```
repeat
    print k
    set k to k + 1
until k > n
```

Java'da `repeat … until` yoktur. Karşılığı `do … while`'dır, ama koşul **tersine çevrilir**. `until` , koşul doğru olunca durur; `while` ise koşul doğru oldukça devam eder. "*k* > *n* olana kadar" ifadesi, "*k* ≤ *n* olduğu sürece" ifadesiyle aynı anlama gelir.

```
int k = 1, n = 3;
do {
    System.out.println(k);
    k = k + 1;
} while (k <= n);
```

k = 1, *n* = 3 için program 1, 2 ve 3'ü alt alta yazar. Gövde üç kez çalışır; üçüncü turun sonunda *k* = 4 olur, `4 <= 3` yanlış çıkar ve döngü biter.

	Önce koşul	Önce gövde
Sözde kod	`while koşul do …`	`repeat … until koşul`
Java	`while (koşul) { … }`	`do { … } while (koşul);`
Koşulun anlamı	Doğru oldukça devam et	Sözde kodda: doğru olunca dur. Java'da: doğru oldukça devam et.
Gövde en az	0 kez çalışır	1 kez çalışır

Tablo 1.2.14: Önce koşul ve önce gövde sınayan döngülerin karşılaştırması.

İki yapının farkı, koşulun baştan yanlış olduğu durumda görülür. `k = 5, n = 3` ile başlanırsa `do … while` döngüsü 5'i bir kez yazar, sonra koşulu sınar ve durur. Aynı koşulla yazılmış bir `while (k <= n)` döngüsü ise gövdeyi hiç çalıştırmaz ve hiçbir şey yazmaz. Gövdenin en az bir kez çalışması gereken durumlarda `do … while` doğal bir seçimdir. Kullanıcı geçerli bir değer girene kadar ondan giriş istemek bunun tipik örneğidir: değer en az bir kez okunmalıdır.

Sık yapılan hata

Sözde koddaki `until` koşulunu Java'ya olduğu gibi taşımak. `until k > n` yerine `while (k > n);` yazılırsa anlam tersine döner. $k = 1$, $n = 3$ için gövde bir kez çalışır ve 1 yazılır. Ardından `2 > 3` yanlış çıkar ve döngü hemen biter. Beklenen çıktı 1, 2, 3 iken yalnız 1 yazılır. `until` koşulu Java'ya taşınırken her zaman değillenir.

1.2.9.4 Sonsuz döngü

Tanım 1.2.23

Sonsuz döngü (*infinite loop*), devam koşulu hiçbir zaman yanlış olmadığı için bitmeyen döngüdür.

Bir döngünün bitmesi için gövdenin koşulu etkileyen değişkeni değiştirmesi ve bu değişikliğin koşulu sonunda yanlış yapması gerekir. `while` örneğinde `i = i - 1;` satırı unutulursa `i` hep 3 kalır, `3 > 0` hep doğru olur ve döngü hiç bitmez. Benzer biçimde `i = i + 1;` yazılırsa `i` büyür ve koşul yine hep doğru kalır. Bu ikinci durumda `i` sonunda `int`'in sınırına ulaşıp taşar ve negatif olur; döngü belki biter ama çok uzun bir süre sonra ve yanlış bir sonuçla.

Derleyici sonsuz döngüleri genel olarak yakalayamaz. Durma probleminin bir sonucu da budur: bir döngünün her girdide biteceğini otomatik olarak gösteren genel bir yöntem yoktur. Takılan bir program, geliştirme ortamındaki (IDE) durdur düğmesiyle ya da konsolda Ctrl+C ile durdurulur.

Bir döngüyü yazarken üç soru sormak sonsuz döngüleri büyük ölçüde önler. Koşuldaki değişken döngüden önce doğru bir değer alıyor mu? Gövde bu değişkeni değiştiriyor mu? Değişiklik koşulu sonunda yanlış yapacak yönde mi? Üç sorunun yanıtı da “evet” ise döngü biter.

1.2.10 Sık yapılan hatalar

Sözde koddan Java'ya geçerken bazı hatalar çok sık yapılır. Bir kısmını derleyici yakalar ve bir hata iletisi verir; bir kısmı ise derlenir ama yanlış çalışır. Aşağıdaki iletiler JDK 21'in `javac` derleyicisiyle elde edilmiştir.

Sık yapılan hata

Noktalı virgülü unutmak. `int a = 5` satırı derleyiciden `error: ';' expected` iletisini alır. Sözde kodda satır sonları serbesttir, Java'da her deyim `;` ile biter. Derleyici, noktalı virgülün beklendiği yeri satırın sonunda bir imle gösterir.

Sık yapılan hata

`=` ile `==`'yi karıştırmak. `if (x = 5)` satırı `error: incompatible types: int cannot be converted to boolean` iletisini verir. `=` atama işlecidir ve `x = 5` ifadesinin değeri 5'tir, yani bir `int`'tir. `if` ise `boolean` bir koşul bekler. Karşılaştırma için `==` kullanılır.

Sık yapılan hata

İlk değer vermemek. `int toplam;` `System.out.println(toplam);` satırları `error: variable toplam might not have been initialized` iletisini verir. Java, yerel değişkenlerin kullanılmadan önce bir değer almasını zorunlu tutar. Toplam biriktiren bir değişken `int toplam = 0;` ile başlatılmalıdır.

Sık yapılan hata

Tam sayı bölmesi. `5 / 2` ifadesi 2, `9 / 5` ifadesi 1 verir. Ondalıklı sonuç için `5 / 2.0` ya da `5.0 / 2` yazılır. Santigrattan Fahrenhayta çevirmede `int f = 9 / 5 * c + 32;` satırı `c` = 25 için 77 yerine 57 verir, çünkü `9 / 5` 1'dir. Çarpmayı öne almak (`c * 9 / 5 + 32`) bu örnekte 77 verir, ama 37 °C için 98,6 yerine 98 bulur; doğru çözüm `c * 9.0 / 5 + 32`'dir.

Sık yapılan hata

Ondalık ayırıcı. Türkçe ayarlı bir bilgisayarda `Scanner` sınıfının `nextDouble()` metodu ondalık ayırıcı olarak virgül bekler: "1,75" girilirse 1.75 okunur. "1.75" yazılırsa `InputMismatchException` fırlatılır. Java kaynak kodunda ise ondalık ayırıcı her zaman noktadır (`1.75`); bu iki kural karıştırılmamalıdır.

Sık yapılan hata

Döngü mantık hataları. En büyüğü bulurken `enBuyuk` değişkenini 0 ile başlatmak, bütün öğeleri negatif olan bir listede yanlış sonuç verir: {−4, −2, −7} listesinde sonuç −2 yerine 0 çıkar. Doğrusu, listenin ilk öğesiyle başlamaktır. Koşul değişkenini güncellemeyi unutmak da

sonsuz döngüye yol açar. Derleyici bu iki hatayı da yakalamaz; program derlenir ve çalışır, ama yanlış sonuç verir ya da hiç bitmez.

1.2.11 Özet

- **Programlama** bilgisayara yapacağı işi adım adım bildiren talimatları yazmaktır. **Hesaplama** veriyi matematik ve mantıkla işleyip bir sonuç üretme sürecidir. Hesaplama yapılan iştir, programlama o işin tarifidir.
- **Bilgisayar bilimi** mantık, problem çözme ve yaratıcılığa dayanır; teknoloji bu bilimde yalnız bir araçtır. Yazılan programlar gerçek insanları etkiler ve bu da bir sorumluluk getirir.
- **Hesaplamalı düşünme** dört beceriden oluşur. Ayrıştırma problemi alt problemlere böler, örüntü tanıma tekrar eden işi görür. Soyutlama gereksiz ayrıntıyı atar, algoritma tasarımı çözümü sonlu ve açık adımlarla yazar.
- **Sabit programlı** makinede program donanıma gömülüdür. **Depolanan programlı** makinede program da veri gibi bellekte durur ve değiştirilebilir. EDVAC ve von Neumann mimarisi bugünkü bilgisayarların temelidir.
- Tarihin ana çizgisi: Babbage'ın Analitik Makinesi (1837) ve Lovelace'ın programı (1843); Turing'in kuramsal modeli (1936); vakum tüplü ABC, Colossus ve ENIAC; depolanan programlı EDVAC; transistör (1947) ve Fortran (1957).
- **Turing makinesi** bant, bant başlığı ve kural tablosundan oluşur. Evrensel Turing makinesi başka bir makinenin tanımını okuyup onu taklit eder. Church–Turing tezine göre adım adım hesaplanabilen her şey bir Turing makinesiyle hesaplanabilir.
- Her problem hesaplanabilir değildir: **durma problemi**ni her program için çözen bir algoritma yoktur. Java gibi Turing-tam diller güç bakımından eşittir; hız ve kolaylık bakımından farklıdır.
- Kaynaklar sınırlıdır ve bu sınırlar Java'da da görünür: `int` taşar, `double` yaklaşık değer tutar, iki `int`'in bölümü `int`'tir.
- **Bildirimsel bilgi** bir şeyin ne olduğunu söyler ve sınamaya yarar. **Buyurusal bilgi** nasıl yapılacağını söyler ve bilgisayara gereken budur. Babil yöntemi karekökü, Öklid algoritması EBOB'u buyurusal olarak bulur.
- **Algoritma** açık, sıralı, sonlu ve amaçlı adımlardan oluşur. Algoritma yazılmadan önce problemin **girdisi** ve **çıktısı** belirlenir; belirsiz istekler sorularla netleştirilir.
- Her algoritma **sıralı**, **koşullu** ve **döngülü** işlemlerden kurulur. `while` koşulu önce, `do … while` sonra sınar. Sözde koddaki `repeat … until` Java'da `do … while` ile ve koşul değillenerek yazılır. Gövde koşulu sonunda yanlış yapmazsa döngü sonsuza dek sürer.

1.2.12 Kavram listesi

hesaplama *computation* | hesaplamalı düşünme *computational thinking* | ayrıştırma *decomposition*

soyutlama *abstraction* · algoritma *algorithm* · sözde kod *pseudocode* · bit *bit* · bayt *byte* · depolanan programlı bilgisayar *stored-program computer* · von Neumann mimarisi *von Neumann architecture* · derleyici *compiler* · Turing makinesi *Turing machine* · hesaplanabilirlik *computability* · taşma *overflow* · kayan noktalı sayı *floating-point number* · tam sayı bölmesi *integer division* · bildirimsel / buyurusal bilgi *declarative / imperative knowledge* · en büyük ortak bölen (EBOB) *greatest common divisor (GCD)* · Öklid algoritması *Euclidean algorithm* · girdi / çıktı *input / output* · sıralı akış *sequential flow* · koşullu akış *conditional flow* · while döngüsü *while loop* · do-while döngüsü *do-while loop* · sonsuz döngü *infinite loop*

1.2.13 Kendini Dene

1. Bir otoparkın gün içindeki en yoğun saatini bulan bir program yazılacak. Giriş ve çıkış kapılarındaki sayaçlar her araç geçişinde saati kaydediyor. Aşağıdaki kararlardan her biri hesaplamalı düşünmenin hangi becerisine karşılık gelir? (a) Araçların plakası ve rengi göz ardı edilir. (b) Problem "kayıtları oku, her saat için içerideki araç sayısını hesapla, en büyüğü bul, yaz" diye bölünür. (c) "En büyüğü bulma" işinin not ortalamasındaki döngüye benzediği fark edilir.

   Cevap

   (a) Soyutlama: sonuç için yalnız geçiş saatleri ve yönü (giriş ya da çıkış) gerekir. (b) Ayrıştırma: büyük problem dört alt probleme bölünmüştür. (c) Örüntü tanıma: her kayıt için tekrar eden aynı adım bir döngüyle yazılabilir. Bu üç adımdan sonra algoritma tasarımı, alt problemleri sıralı adımlar olarak yazar.

2. ENIAC ile EDVAC arasındaki temel fark nedir? Hangisi depolanan programlı bir bilgisayardır?

   Cevap

   ENIAC'ta program kablo bağlantılarıyla kuruluyordu; yeni bir iş için kablolar elle yeniden takılıyordu. Sayıları da onlu sistemde tutuyordu. EDVAC ise programı ve veriyi aynı bellekte saklıyordu ve ikili sistemi kullanıyordu. Depolanan programlı bilgisayar EDVAC'tır; tasarımı von Neumann mimarisinin temelidir.

3. Bitleri ters çeviren Turing makinesi 111000 bandında kaç adım çalışır? Makine durduğunda bant-

ta ne kalır?

Cevap

Her hücre için bir yazma adımı vardır: 6 yazma adımı. Yedinci adımda □ okunur ve makine durur. Toplam 7 adım çalışır; bantta 000111 kalır. Genel olarak n bitlik bir bant için $n + 1$ adım gerekir.

4. Babil yöntemiyle $\sqrt{9}$, $y_0 = 1$ alınarak üç adım hesaplansın. Tahminler kökün hangi tarafında kalır?

Cevap

$y_1 = (1 + 9) / 2 = 5$; $y_2 = (5 + 9/5) / 2 = 3{,}4$; $y_3 = (3{,}4 + 9/3{,}4) / 2 \approx 3{,}023529$. Başlangıç tahmini kökün altındadır; ilk adımda tahmin kökün üstüne çıkar ve sonra hep üstte kalarak 3'e yaklaşır.

5. Öklid algoritmasıyla EBOB(252, 105) bulunsun. Kaç bölme yapılır? Sonucun sağlaması nasıl yapılır?

Cevap

252 = 2 · 105 + 42; 105 = 2 · 42 + 21; 42 = 2 · 21 + 0. Üç bölmeden sonra kalan 0 olur ve EBOB = 21 bulunur. Sağlama: 252 = 21 · 12 ve 105 = 21 · 5'tir; 12 ile 5'in 1'den büyük ortak böleni yoktur.

6. Aşağıdaki iki satır ne yazdırır?

```
System.out.println(Integer.MAX_VALUE + 1 == Integer.MIN_VALUE);
System.out.println(7 / 2 * 2.0);
```

Cevap

Birinci satır `true` yazar: en büyük `int` değere 1 eklenince taşma olur ve sonuç en küçük değere sarar. İkinci satır `6.0` yazar: işlemler soldan sağa yapılır, önce `7 / 2` tam sayı bölmesiyle 3 olur, sonra 3 · 2.0 = 6.0 bulunur.

7. `k` = 0 ile başlayan iki döngü veriliyor. Her birinin sonunda `k` kaçtır?

```
do { k = k - 3; } while (k > 0);
while (k > 0) { k = k - 3; }
```

Cevap

`do … while` gövdeyi koşuldan önce bir kez çalıştırır: `k` = −3 olur, `-3 > 0` yanlış çıkar ve döngü biter; sonuç −3. `while` ise koşulu önce sınar: `0 > 0` yanlış olduğu için gövde hiç çalışmaz; sonuç 0.

8. Ortalama programında bölme satırı `double ort = toplam / notlar.length;` biçiminde, yani `(double)` dönüşümü olmadan yazılsa, 40, 75 ve 90 notları için ne yazdırılır? Neden?

   Cevap

   "Ortalama: 68.0" yazdırılır. `toplam` (205) ve `notlar.length` (3) ikisi de `int`'tir; bölme tam sayı bölmesi olarak yapılır ve 68 bulunur. Bu değer ancak atamada `double`'a çevrilir; kaybolan kesir geri gelmez.

9. "Bir sayı, 1'den ve kendisinden başka pozitif böleni olmayan, 1'den büyük bir tam sayıysa asaldır" cümlesi hangi tür bilgidir? Aynı konuda öbür türe bir örnek verilsin.

   Cevap

   Bildirimseldir: asal sayının ne olduğunu tanımlar ve bir adayı sınamaya yarar. Buyurusal bir karşılığı şöyle olabilir: "$n > 1$ olsun. 2'den başlayarak $\sqrt{n}$'ye kadar her d için $n \bmod d$'ye bak; kalan 0 olan bir d bulunursa 'asal değil', bulunmazsa 'asal' yaz." Bu tarif, kararın nasıl verileceğini adım adım söyler.

1.2.14 İleri okuma

- J. M. Wing, "Computational Thinking", *Communications of the ACM* 49(3), 2006: hesaplamalı düşünme terimini yaygınlaştıran üç sayfalık makale. Bu düşünme biçiminin neden herkese gerekli olduğunu tartışır.
- J. V. Guttag, *Introduction to Computation and Programming Using Python* (3. baskı, 2021), Bölüm 1: bildirimsel ve buyurusal bilgi ayrımını ve karekök için tahmin et-iyileştir yöntemini bu notla aynı örnekler üzerinden anlatır.
- C. Petzold, *Code: The Hidden Language of Computer Hardware and Software* (2. baskı, 2022): rölelerden ve anahtarlardan başlayarak bir bilgisayarın parça parça nasıl kurulduğunu, ikili sayıları ve depolanan program fikrini teknik olmayan bir dille açıklar.
- R. Sedgewick ve K. Wayne, *Computer Science: An Interdisciplinary Approach* (2016), Turing makineleri ve hesaplanabilirlik bölümleri: Turing makinesi örneklerini, evrensel makineyi ve durma probleminin kanıtını Java programlarıyla ilişkilendirerek verir.

1.2.15 Kaynaklar

1. H. M. Deitel, P. J. Deitel, *Java: How to Program*, 11. baskı, Pearson, 2017.
2. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019.
3. J. V. Guttag, *Introduction to Computation and Programming Using Python*, 3. baskı, MIT Press, 2021.
4. Oracle, *The Java Tutorials*: "Language Basics", docs.oracle.com/javase/tutorial.

5. C. Petzold, *Code: The Hidden Language of Computer Hardware and Software*, 2. baskı, 2022.

6. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016.

7. A. M. Turing, "On Computable Numbers, with an Application to the Entscheidungsproblem", *Proceedings of the London Mathematical Society*, 1936.

8. J. M. Wing, "Computational Thinking", *Communications of the ACM* 49(3), 33–35, 2006.

KISIM I · BÖLÜM 1.3

Algoritmalar

Program yazmak klavyenin başında başlamaz. Önce problemin ne istediği netleşir, sonra çözüm adım adım kâğıda dökülür, en son bu adımlar bir programlama diline çevrilir. Bu not, bu yolun her durağını küçük ama eksiksiz örneklerle anlatır. Bir problemin girdi, çıktı ve test senaryolarıyla tanımlanması; çözümün sözde kodla yazılması ve değişken tablosuyla elle izlenmesi; sözde kodun Java'ya çevrilmesi ilk kesimlerin konusudur. Ardından iki ve üç sayıyı sıralama, faktöriyel, Fibonacci, üs alma, palindrom ve asal sayı algoritmaları gelir. Her biri aynı düzenle işlenir: sorun tanımı, sözde kod, elle izleme, Java kodu ve testler. Not, arama ve sıralamanın üç basit yöntemiyle biter: rastgele, doğrusal ve ikili arama; karıştırma, seçimli sıralama ve kabarcık sıralaması. Her kesimde Java'ya özgü tuzaklar, derleyicinin ve çalışma ortamının gerçek iletileriyle birlikte verilir.

ÖĞRENME ÇIKTILARI

Problem çözme
Bir problemi girdi, çıktı ve test senaryolarıyla tanımlamak; sınır durumları ayrıca düşünmek; çözümü önce algoritma, sonra kod olarak yazmak.

Sözde kod ve izleme
Algoritmayı sözde kodla ifade etmek ve değişken tablosuyla adım adım elle izlemek.

Java'ya dönüştürme
Okuma, yazma, karar ve döngü adımlarını Java karşılıklarına çevirmek; `Scanner`, `print` / `println`, tam sayı bölmesi, `long` ve `equals` gibi ayrıntıları doğru kullanmak.

Temel algoritmalar
Sıralı yazdırma, faktöriyel, Fibonacci, palindrom, asal sayı ve üs alma algoritmalarını yazmak, izlemek ve test etmek.

Arama
Rastgele, doğrusal ve ikili aramayı karşılaştırmak; ikili aramanın neden sıralı veri gerektirdiğini açıklamak; en kötü durumdaki karşılaştırma sayılarını hesaplamak.

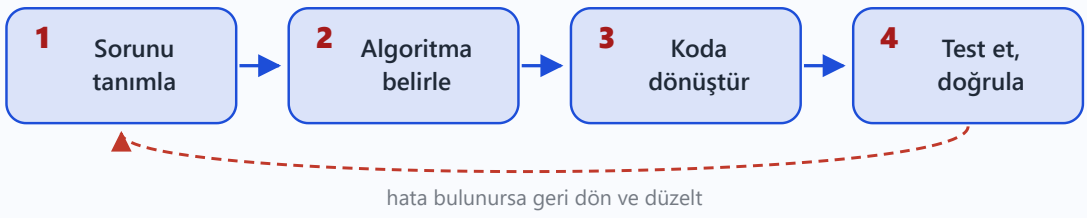
Sıralama
Karıştırmayı, seçimli sıralamayı ve kabarcık sıralamasını bir dizi üzerinde adım adım uygulamak; karşılaştırma ve yer değiştirme sayılarını bulmak.

1.3.1 Problem çözme

Programlamaya yeni başlayan biri çoğu zaman problemi okur okumaz kod yazmaya girişir. Küçük problemlerde bu yöntem bir süre işe yarar. Problem biraz büyüdüğünde ise yazılan kod ne yaptığı belli olmayan bir yamalar yığınına döner. Hata çıktığında nereden kaynaklandığı da bulunamaz, çünkü ortada "programın ne yapması gerektiğini" söyleyen yazılı bir şey yoktur. Deneyimli programcılar bu yüzden işi aşamalara böler. Her aşamanın çıktısı bir sonrakinin girdisi olur ve her aşama kendi içinde denetlenebilir.

1.3.1.1 Önce algoritma, sonra uygulama

Bir problemin bilgisayarla çözülmesi dört adımlı bir döngüdür (Şekil 1.3.1). İlk adım **sorunu tanımlamaktır**: ne verilecek, ne istenecek? Bu sorunun cevabı belirsiz kaldıkça sonraki adımların hepsi boşa gidebilir. İkinci adım **algoritmayı belirlemektir**. Algoritma, sorunu çözmek için izlenecek adımların listesidir. Aynı sorun için birden çok algoritma bulunabilir; doğru seçim programın hızını ve bellek kullanımını belirler. Üçüncü adım algoritmanın seçilen programlama dilinde, bu derste Java'da, **koda dönüştürülmesidir**. Dördüncü adımda program önceden hazırlanmış **test senaryolarıyla** denenir. Bir test beklenen sonucu vermezse önceki adımlara dönülür. Hata kodlamada olabilir, algoritmada olabilir, hatta sorunun yanlış anlaşılmasından da kaynaklanabilir.



Şekil 1.3.1: Problem çözme döngüsü. Test bir hata gösterirse kesikli ok boyunca geri dönülür ve hatanın kaynağı olan adım düzeltilir.

Bu düzenin en önemli kazancı, hataların ucuz oldukları anda yakalanmasıdır. Kâğıt üzerindeki bir algoritmada bulunan yanlış birkaç satırlık düzeltmeyle giderilir. Aynı yanlış yüzlerce satırlık bir programın içine gömüldükten sonra bulunursa hem bulmak hem düzeltmek çok daha pahalıdır. Test senaryolarının kod yazılmadan önce hazırlanması da aynı nedene dayanır. Program yazıldıktan sonra hazırlanan testler, farkında olmadan programın yaptığını "doğru" sayma eğilimindedir.

1.3.1.2 Algoritma nedir?

> **Tanım 1.3.1**
>
> **Algoritma** (*algorithm*), bir problemi çözmek için izlenen, sırası belli, açık ve sonlu sayıda adımdan oluşan yönergeler dizisidir.

Sözcük, 9. yüzyılda Bağdat'ta yaşamış matematikçi el-Hârizmî'nin adından gelir. Adının Latince yazılışı Algoritmi idi; onun ondalık sayılarla hesap üzerine yazdığı kitabın Latince çevirisi Avrupa'da bu adla tanındı. Zamanla “algoritma” sözcüğü, hesap yapmanın adım adım tarif edilmiş her yöntemi için kullanılır oldu.

Bilgisayar bir algoritmayı doğrudan anlamaz. Algoritma insanın düşünme biçimiyle yazılır; bilgisayar ise yalnız kendi makine dilindeki komutları çalıştırır. Aradaki köprü programlama dilidir: algoritma Java gibi bir dile çevrilir, derleyici bu metni makinenin çalıştırabileceği biçime dönüştürür. Bir algoritmanın birden çok programla yazılabileceği gibi, bir problemin de birden çok algoritması olabilir. Bu algoritmalar genellikle iki ölçüyle karşılaştırılır: ne kadar sürede bittikleri ve ne kadar bellek kullandıkları.

Gündelik hayattan en tanıdık benzetme yemek tarifidir. Malzemeler girdidir, yapılış adımları algoritmadır, ortaya çıkan yemek de çıktıdır. Ancak tarifler insan için yazılır ve “biraz tuz ekle”, “kıvamını bulana dek karıştır” gibi belirsiz adımlar içerir. Deneyimli bir aşçı bu boşlukları kendi bilgisiyle doldurur. Bilgisayar ise “biraz” sözcüğüne bir sayı karşılık getiremez; ona “5 gram tuz ekle” demek gerekir. Bir yönergeler listesinin algoritma sayılması için beş özelliği taşıması beklenir (Tablo 1.3.1).

Özellik	Anlamı	Tarifte bozulduğu yer
Girdi (*input*)	Sıfır ya da daha fazla başlangıç verisi alır.	—
Çıktı (*output*)	En az bir sonuç üretir.	—
Kesinlik	Her adım açık ve tek anlamlıdır.	“Biraz tuz ekle.”
Sonluluk	Sonlu sayıda adımda biter.	“Göz kararı olana dek karıştır.”
Etkinlik	Her adım gerçekten yapılabilir.	“Sonsuz incelikte doğra.”

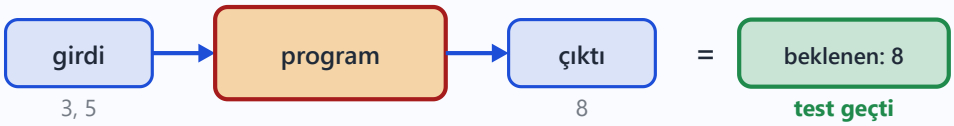
Tablo 1.3.1: Bir algoritmanın beş özelliği ve yemek tarifindeki karşılıkları.

Girdi özelliğinin “sıfır ya da daha fazla” diye tanımlanması bilinçlidir. Ekrana ilk on asal sayıyı yazan bir algoritmanın hiç girdisi yoktur ama yine de bir algoritmadır. Çıktısı olmayan bir yöntem ise hiçbir şey üretmediği için işe yaramaz; bu yüzden en az bir çıktı istenir. Sonluluk, algoritmanın her girdi için bir noktada durmasıdır. Kullanıcı yanlış değer girdiğinde sonsuza dek dönen bir döngü bu özelliği bozar. Etkinlik ise adımların gerçekten yapılabilir olmasını ister: “bütün asal sayıları listele” adımı açık ve tek anlamlıdır ama sonsuz sayıda asal sayı olduğu için yapılamaz.

1.3.1.3 Sorunu tanımlama ve test senaryoları

Sorunu tanımlamak, programın girdilerini ve çıktılarını doğal dille, bu derste Türkçe, açıkça yazmak demektir. “İki sayıyı topla” cümlesi ilk bakışta yeterli görünür. Biraz düşününce sorular çıkar: sayılar tam sayı mı, ondalıklı da olabilir mi? Negatif sayı girilebilir mi? Sonuç ekrana mı yazılacak, başka bir hesapta mı kullanılacak? Bu soruların cevapları tanıma eklendikçe programın ne yapacağı netleşir.

Tanımın ikinci parçası test senaryolarıdır. **Test senaryosu** (*test case*), belirli bir girdi için beklenen çıktının önceden yazılmasıdır. Program çalıştırılır, ürettiği çıktı beklenenle karşılaştırılır; ikisi aynıysa test geçer (Şekil 1.3.2). Beklenen çıktı programdan bağımsız olarak, elle ya da başka bir güvenilir yolla bulunmalıdır. Yoksa test, programı kendisiyle karşılaştırmaktan öteye geçmez.



Şekil 1.3.2: Bir testin işleyişi: 3 ve 5 girdisi programa verilir, üretilen 8 beklenen 8 ile karşılaştırılır ve test geçer.

Tablo 1.3.2 iki küçük problem için hazırlanmış test senaryolarını gösterir. Her problemde ilk iki satır sıradan girdilerdir. Üçüncü satırlar ise bilinçli seçilmiş **sınır durumlardır** (*edge case*). Sıfır, boş metin, negatif sayı, dizinin ilk ya da son öğesi bu tür uç girdilerdir. Programcılar sıradan girdileri doğal olarak düşünür; uçlar ise kolayca unutulur. Bu yüzden hatalar en çok uçlarda saklanır. Kelime sayan bir program “Merhaba, dünya!” için 2 verirken boş metinde 1 verebilir. Metni boşluklardan bölen basit bir yaklaşım, hiç boşluk olmayan boş metni bile tek parça sayar.

Sorun	Test	Girdi	Beklenen çıktı
İki sayının toplamı	1	3, 5	8
	2	−2, 7	5
	3 (sınır)	0, 0	0
Metindeki kelime sayısı	1	"Merhaba, dünya!"	2
	2	"Bu bir denemedir."	3
	3 (sınır)	"" (boş metin)	0

Tablo 1.3.2: İki problem için test senaryoları. Üçüncü satırlar sınır durumdur.

İyi bir test kümesi az sayıda ama birbirinden farklı durumları kapsar. Toplama için 3 + 5, 4 + 6 ve 10 + 20 testleri aynı şeyi üç kez sınar: üçü de pozitif tam sayıdır. Bunların yerine bir pozitif çift, bir negatifli çift ve bir sıfır çifti seçmek, aynı emekle çok daha fazla hata yakalama şansı verir. Bu notun bütün algoritmalarında test senaryoları bu ilkeyle seçilmiştir: her biri için hem sıradan hem uç girdiler vardır.

1.3.1.4 Algoritma belirleme ve elle izleme

Sorun tanımlandıktan sonra çözümün adımları sırayla yazılır. İki sayının toplamı için algoritma dört adımdır:

1. İlk sayıyı al.
2. İkinci sayıyı al.
3. İki sayıyı topla.
4. Toplamı ekrana yazdır.

Algoritma yazılınca iş bitmez; kod yazılmadan önce algoritmanın kendisi test edilir. Bunun yolu **elle izlemedir** (*dry run*). Bir test senaryosu seçilir ve algoritma kâğıt üzerinde adım adım çalıştırılır. Her adımdan sonra değişkenlerin değeri bir tabloya yazılır. Bu tabloya **izleme tablosu** (*trace table*) denir. Tablonun sütunları değişkenler ve ekran, satırları adımlardır. Bir hücre boş bırakılırsa o değişken o anda henüz bir değer almamıştır.

Adım	sayı1	sayı2	toplam	Ekran
1. İlk sayıyı al	3			
2. İkinci sayıyı al	3	5		
3. İki sayıyı topla	3	5	8	
4. Toplamı yazdır	3	5	8	8

Tablo 1.3.3: Toplama algoritmasının 1. test (girdi 3, 5) için elle izlenmesi. Ekrana 8 yazıldı; beklenen de 8.

Bu örnekte izleme çok kolaydır ve gereksiz görünebilir. Ama aynı alışkanlık döngülü algoritmalarda büyük fark yaratır. Döngünün kaç kez döndüğü, bir değişkenin hangi turda hangi değeri aldığı, döngünün bir tur erken mi geç mi bittiği ancak böyle bir tabloyla güvenle görülür. Elle izlemede küçük ama anlamlı test senaryoları seçilir. Faktöriyel için 5 iyi bir seçimdir: beş tur elle izlenebilir ve sonuç (120) ezbere bilinir. 20 ise kötü bir seçimdir; hem tablo uzar hem de sonucu doğrulamak zorlaşır.

1.3.1.5 Sözde kod

Algoritma dört biçimde yazılabilir: doğal dille, sözde kodla, akış diyagramıyla ya da doğrudan bir programlama diliyle. Doğal dil herkesin anladığı ama belirsizliğe açık bir araçtır. Programlama dili ise kesindir ama ayrıntı yükü ağırdır: noktalı virgüller, tip bildirimleri, kütüphane adları asıl düşünceyi gölgeler. **Sözde kod** (*pseudocode*) bu ikisinin arasında durur.

> **Tanım 1.3.2**
>
> **Sözde kod**, programlama diline benzeyen ama derleyici kuralları olmayan, insan için yazılmış bir algoritma gösterimidir. Okuma, yazma, atama, karar ve döngü gibi temel yapılar için sabit anahtar sözcükler kullanır; geri kalan her şey doğal dille yazılabilir.

Sözde kodun iki büyük yararı vardır. Birincisi, mantık hatalarını kod yazmadan önce bulmayı kolaylaştırır: dikkat söz dizimine değil, adımların sırasına ve koşullara yönelir. İkincisi, farklı diller kullanan programcıların aynı algoritma üzerinde konuşmasını sağlar. Java bilen biriyle Python bilen biri aynı sözde kodu okuyup kendi dillerine çevirebilir. Bu notta kullanılan sözde kod anahtar sözcükleri ve Java karşılıkları Tablo 1.3.4'te toplanmıştır.

Sözde kod	Anlamı	Java karşılığı
Başla … Bitir	Algoritmanın başı ve sonu	`main` metodunun `{ … }` gövdesi
Oku x	Kullanıcıdan değer al	`x = tarayici.nextInt();`
Yaz "…", x	Ekrana yazdır	`System.out.println("…" + x);`
x = 5	Değişkene değer ata	`x = 5;`
Eğer x = y ise	Eşitliği karşılaştır	`if (x == y)` (iki eşittir)
Değilse	Koşul yanlışsa	`else`
Döngü i = 1'den n'ye kadar	Tekrarla	`for (int i = 1; i <= n; i++)`
Döngüyü Kır	Döngüden hemen çık	`break;`
%, ve, veya	Kalan, mantıksal ve/veya	`%`, `&&`, `||`

Tablo 1.3.4: Sözde kod anahtar sözcükleri ve Java karşılıkları.

Tablonun en sinsi satırı eşitlik satırıdır. Sözde kodda ve matematikte `=` işareti hem atama hem karşılaştırma için kullanılır; anlam bağlamdan çıkarılır. Java'da ise bu iki iş farklı işaretlerle yapılır. Tek eşittir (`=`) sağdaki değeri soldaki değişkene koyar. Çift eşittir (`==`) iki değeri karşılaştırır ve `true` ya da `false` üretir. Sözde koddan Java'ya çeviri yapılırken her `Eğer … = … ise` satırındaki işaretin `==` 'ye dönüştürülmesi gerekir.

Sık yapılan hata

Koşulda tek eşittir kullanmak. `if (sayi % bolen = 0)` satırı derlenmez. Sol taraf bir değişken değil, hesaplanmış bir değer olduğu için derleyici şu iletiyi verir: `error: unexpected type … required: variable, found: value` . Sol taraf bir değişken olsaydı, örneğin `if (s = 5)` , ileti değişirdi: `error: incompatible types: int cannot be converted to boolean` . Java'da `if` yalnız `boolean` kabul ettiği için bu hata derleme aşamasında yakalanır. C gibi dillerde aynı satır sessizce derlenir ve programın mantığını bozar.

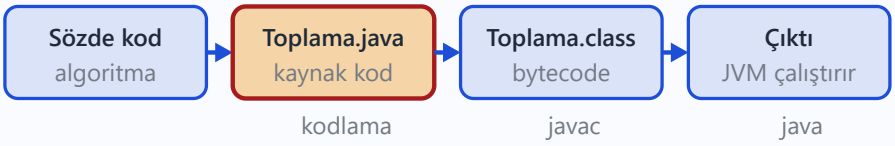
Mantıksal işleçlerde de benzer bir ayrım vardır. Sözde koddaki “ve” Java'da `&&` , “veya” ise `||` ile yazılır. Bu işleçler kısa devre çalışır. `a && b` ifadesinde `a` yanlışsa sonuç zaten yanlış olacağı için `b` hiç hesaplanmaz. `a || b` ifadesinde `a` doğruysa `b` 'ye bakılmaz.

1.3.2 Koda dönüştürme

Algoritma hazırlanıp elle izlendikten sonraki adım kodlamadır (*implementation*). Programlama dili, algoritmayı bilgisayarın derleyip çalıştırabileceği biçimde ifade eder. Kod yazılırken program mantıksal parçalara ayrılır: girdi alma, hesaplama ve çıktı verme. Daha büyük programlarda her parça ayrı bir metoda taşınır. Sözde koddaki her satır genellikle bir ya da birkaç Java satırına karşılık gelir.

1.3.2.1 Sözde koddan çalışan programa

Bir algoritmanın ekranda sonuç üretmesine kadar dört durak vardır (Şekil 1.3.3). Sözde kod Java kaynak koduna çevrilir ve `Toplama.java` gibi bir dosyaya yazılır. Java derleyicisi `javac` bu dosyayı okur. Kodda yazım hatası yoksa `Toplama.class` adlı bir bayt kodu (*bytecode*) dosyası üretir. Bayt kodu belirli bir işlemcinin makine dili değildir. Java sanal makinesinin (*Java Virtual Machine*, JVM) anladığı ara bir dildir. `java Toplama` komutu JVM'i başlatır; JVM bayt kodunu çalıştırır ve çıktı ekrana gelir. Aynı `.class` dosyası Windows'ta, Linux'ta ya da macOS'ta, JVM kurulu her makinede çalışır.



Şekil 1.3.3: Sözde koddan çıktıya: kodlama, `javac` ile derleme ve `java` komutuyla JVM'de çalıştırma.

Hatalar iki farklı anda ortaya çıkabilir. **Derleme hataları** `javac` çalışırken bulunur: eksik noktalı virgül, yanlış yazılmış anahtar sözcük, tanımlanmamış değişken, uyumsuz tip. Derleme hatası olan bir programdan `.class` dosyası üretilmez. **Çalışma zamanı hataları** ise program çalışırken çıkar: dizinin olmayan bir hücresine erişmek, sıfıra bölmek, kullanıcının sayı yerine harf girmesi. Üçüncü ve en tehlikeli tür **mantık hatalarıdır**. Program derlenir, çalışır, hiçbir ileti vermez ama yanlış sonuç üretir. Bu tür hataları ancak testler yakalar.

Algoritma adımlarıyla Java satırları arasındaki eşleşme Tablo 1.3.5'te görülür. Her adım tek bir satıra dönüşmüştür. Daha karmaşık algoritmalarda bir adım birkaç satır tutabilir; ama eşleşmenin izi her zaman sürülebilir olmalıdır.

Algoritma adımı	Java
İlk sayıyı al	`sayi1 = tarayici.nextDouble();`
İkinci sayıyı al	`sayi2 = tarayici.nextDouble();`
İki sayıyı topla	`toplam = sayi1 + sayi2;`
Toplamı yazdır	`System.out.println(toplam);`

Tablo 1.3.5: Toplama algoritmasının adımları ve Java karşılıkları.

Adlandırma konusunda bir not: Java kütüphaneleri İngilizce adlar kullanır (`Scanner`, `nextDouble`, `println`). Bu derste kendi değişken ve metot adlarımız Türkçe seçilir, ama Türkçe karakter kullanılmaz: `sayı1` yerine `sayi1`, `üçüncü` yerine `ucuncu`. Java aslında Türkçe harfli adları kabul eder. Yine de kaynak dosya farklı bir karakter kodlamasıyla açıldığında bu harfler bozulabilir ve derleme hatasına yol açabilir. Sözde kodda böyle bir sorun yoktur; orada `birinciTamsayı` gibi adlar rahatça kullanılır.

1.3.2.2 Toplama programı

Toplama algoritmasının tam Java programı aşağıdadır. Sunumda yalnız `main` gövdesi gösterilen kod burada `import` satırı ve sınıf tanımıyla birlikte verilmiştir; bu hâliyle `Toplama.java` adlı bir dosyaya yazılıp derlenebilir.

```
import java.util.Scanner;

public class Toplama {
    public static void main(String[] args) {
        Scanner tarayici = new Scanner(System.in);
        System.out.print("Birinci sayıyı giriniz: ");
        double sayi1 = tarayici.nextDouble();
        System.out.print("İkinci sayıyı giriniz: ");
        double sayi2 = tarayici.nextDouble();
        double toplam = sayi1 + sayi2;
        System.out.println(sayi1 + " ile " + sayi2
                + " toplamı: " + toplam);
        tarayici.close();
    }
}
```

İlk satırdaki `import java.util.Scanner;` derleyiciye `Scanner` sınıfının nerede bulunacağını söyler. Java'nın standart kütüphanesi paketlere bölünmüştür ve `Scanner`, `java.util` paketindedir. `System` ve `String` gibi birkaç temel sınıf `java.lang` paketindedir ve bu paket her programa kendiliğinden eklenir. Geri kalan her sınıf için `import` gerekir. Satır unutulursa derleyici `Scanner` adını tanımaz:

```
Toplama.java:4: error: cannot find symbol
        Scanner tarayici = new Scanner(System.in);
        ^
  symbol:   class Scanner
  location: class Toplama
```

Aynı satırda `Scanner` iki kez geçtiği için derleyici bu hatayı iki kez bildirir ve “2 errors” der. Tek bir eksik satır birden çok hata iletisine yol açabilir. Bu yüzden derleme hataları her zaman ilkinden başlanarak okunur.

`public class Toplama` satırı sınıfı tanımlar. Herkese açık bir sınıfın adı dosya adıyla aynı olmak zorundadır: `Toplama` sınıfı `Toplama.java` dosyasında durur. `public static void main(String[] args)` ise programın başlangıç noktasıdır; JVM çalıştırmaya bu metottan başlar. Sözde koddaki `Başla … Bitir` çiftinin karşılığı bu metodun süslü parantezleridir.

`Scanner tarayici = new Scanner(System.in);` klavyeden okuma yapacak bir nesne oluşturur. `System.in` standart girdi akışıdır, yani klavye. `tarayici` nesnesinin `nextDouble()` metodu bir sonraki sayıyı `double` olarak, `nextInt()` metodu `int` olarak, `nextLine()` metodu ise satırın tamamını metin olarak okur. Program sonundaki `tarayici.close();` nesnenin kullandığı kaynağı serbest bırakır.

`double toplam = sayi1 + sayi2;` satırında önce sağ taraf hesaplanır, sonra sonuç soldaki değişkene konur. Atama işlemi her zaman sağdan sola işler. Bu yüzden `toplam = toplam + 1` gibi matematikte saçma görünen bir satır programlamada anlamlıdır: “toplamın şimdiki değerine 1 ekle ve sonucu yine toplama koy”.

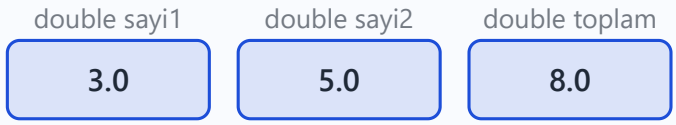
1.3.2.3 Toplama programının izlenmesi

Kullanıcının 3 ve 5 girdiği bir çalıştırmanın satır satır izlemesi Tablo 1.3.6'dadır. Satır numaraları `main` gövdesinin satırlarıdır: 1. satır `Scanner`'ın oluşturulduğu satır, 9. satır `close` çağrısıdır.

Satır	sayi1	sayi2	toplam	Ekran	Açıklama
1	—	—	—		Klavyeden okumak için `Scanner` oluşturulur.
2	—	—	—	Birinci sayıyı giriniz:	`print` alt satıra geçmez; imleç iletinin yanında bekler.
3	3.0	—	—	… giriniz: 3↵	Kullanıcı 3 yazıp Enter'a basar; `nextDouble()` 3.0 okur.
4	3.0	—	—	İkinci sayıyı giriniz:	İkinci ileti yazılır.
5	3.0	5.0	—	… giriniz: 5↵	Kullanıcı 5 girer.
6	3.0	5.0	8.0		Sağ taraf hesaplanır: 3.0 + 5.0 = 8.0.
7–8	3.0	5.0	8.0	3.0 ile 5.0 toplamı: 8.0↵	`+` metinleri birleştirir.
9	3.0	5.0	8.0		Tarayıcı kapatılır, program biter.

Tablo 1.3.6: Toplama programının izlemesi (girdi 3 ve 5). “Ekran” sütunu o satırdan sonra ekranda biriken metindir; ↵ satır sonudur.

Programın ekrana yazdığı son satır “3.0 ile 5.0 toplamı: 8.0”dır. Test senaryosu 8 bekliyordu; program 8.0 yazdı. Değer aynıdır, yazım biçimi farklıdır. `double` tipindeki bir sayı metne çevrilirken tam sayı olsa bile ondalık kısmıyla, “8.0” biçiminde yazılır. Test senaryoları yazılırken bu tür biçim farkları da düşünülmelidir. Testi otomatik yapan bir program “8” ile “8.0” metinlerini farklı sayar.



Şekil 1.3.4: Program bitmeden hemen önce bellekteki üç `double` değişken.

1.3.2.4 Okuma ve yazmanın ayrıntıları

`print` ve `println`. İki metot da kendisine verilen değeri ekrana yazar. Fark, yazdıktan sonra ne olduğundadır. `println` (*print line*) değeri yazıp alt satıra geçer; `print` geçmez. Kullanıcıdan girdi isterken `print` kullanılması bu yüzdendir: imleç "Birinci sayıyı giriniz: " iletisinin hemen yanında bekler ve kullanıcı sayıyı aynı satıra yazar. Bir dizi değeri aynı satıra yan yana yazmak için de `print` kullanılır; son değerden sonra bir `println()` satırı tamamlar.

Metin birleştirme ve işlem sırası. `+` işleci iki sayı arasında toplama, bir metinle herhangi bir değer arasında birleştirme yapar. İfade soldan sağa hesaplanır ve bu, şaşırtıcı sonuçlara yol açabilir. Tablo 1.3.7 JDK 21'de çalıştırılarak elde edilmiş çıktıları gösterir.

İfade	Çıktı	Neden
`3 + 5 + " toplam"`	8 toplam	Önce 3 + 5 = 8 toplanır, sonra metne eklenir.
`"toplam " + 3 + 5`	toplam 35	Metin + 3 = "toplam 3"; ardından + 5 yine birleştirmedir.
`"toplam " + (3 + 5)`	toplam 8	Parantez toplamayı öne alır.
`7 / 2`	3	İki `int` arasında tam sayı bölmesi; kesir atılır.
`7 % 2`	1	Kalan.
`-7 / 2` ve `-7 % 2`	−3 ve −1	Bölüm sıfıra doğru kesilir; kalan bölünenin işaretini alır.
`7.0 / 2`	3.5	İşlenenlerden biri `double` ise gerçek bölme yapılır.

Tablo 1.3.7: `+` işlecinin sayı ve metinle davranışı; bölme ve kalan.

Tam sayı bölmesi bu bölümün algoritmalarında sık kullanılır. Asal sayı kontrolündeki `sayi / 2` ve ikili aramadaki `(alt + ust) / 2` hep tam sayı bölmesidir ve kesirli kısmı bilerek atar. Kalan işleci `%` da en az onun kadar önemlidir. `sayi % bolen == 0` "tam bölünür" demektir; `sayi % 10` son basamağı, `sayi % 2` ise çift mi tek mi olduğunu verir.

Ondalık ayırıcı ve yerel ayar. `Scanner` sayıları okurken bilgisayarın yerel ayarını (*locale*) kullanır. Türkçe ayarlı bir bilgisayarda ondalık ayırıcı virgüldür. Bu yüzden toplama programına "3,5" yazmak çalışır, "3.5" yazmak ise programı durdurur:

```
Birinci sayıyı giriniz: 3.5
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:947)
        at java.base/java.util.Scanner.next(Scanner.java:1602)
        at java.base/java.util.Scanner.nextDouble(Scanner.java:2573)
        at Toplama.main(Toplama.java:7)
```

İngilizce ayarlı bir bilgisayarda durum tersinedir: “3.5” kabul edilir, “3,5” reddedilir. Programın çıktısı ise her iki ayarda da “3.5” biçimindedir, çünkü `double` değerin metne çevrilmesi yerel ayara bakmaz. Aynı programın farklı bilgisayarlarda farklı davranması öğrencileri sık şaşırtır. Okumanın her yerde aynı olması isteniyorsa tarayıcı `new Scanner(System.in).useLocale(Locale.US)` biçiminde oluşturulur. Kullanıcı sayı yerine harf girdiğinde de aynı `InputMismatchException` fırlatılır. Bu istisnanın yakalanıp kullanıcıya yeniden sorulması daha ileri bir konudur. Bu notun programları kullanıcının geçerli bir sayı girdiğini varsayar.

1.3.3 Karar yapısıyla algoritmalar

Bu kesimden başlayarak her algoritma aynı düzenle işlenir. Önce sorun ve test senaryoları tanımlanır, sonra sözde kod yazılır ve değişken tablosuyla elle izlenir. En son Java kodu verilir ve testler çalıştırılır. İlk iki örnek döngü içermez. Algoritmanın izleyeceği yol, girdilerin karşılaştırılmasıyla verilen kararlara bağlıdır.

1.3.3.1 İki tam sayıyı sıralı yazdırma

Sorun tanımı. Girdi iki tam sayıdır. Çıktı aynı iki sayıdır, küçük olan önce yazılır. Test senaryoları Tablo 1.3.8'de görülür. Üçüncü test sınır durumdur: iki sayı eşittir.

Girdi	Beklenen	Sınanan durum
4, 11	4 11	Sayılar zaten sıralı
13, 11	11 13	Sayılar ters sırada
7, 7	7 7	Eşit sayılar (sınır)

Tablo 1.3.8: İki tam sayıyı sıralı yazdırma için test senaryoları.

Sözde kod. Algoritma iki sayıyı okur ve tek bir karşılaştırma yapar. Birinci sayı ikinciden küçükse sayılar okunduğu sırayla, değilse ters sırayla yazılır.

```
Başla
    Yaz "Birinci tamsayıyı girin: "
    Oku birinciTamsayı
    Yaz "İkinci tamsayıyı girin: "
    Oku ikinciTamsayı
    Eğer birinciTamsayı < ikinciTamsayı ise
        Yaz birinciTamsayı, " ", ikinciTamsayı
    Değilse
        Yaz ikinciTamsayı, " ", birinciTamsayı
    Eğer Sonu
Bitir
```

`Eğer … ise … Değilse … Eğer Sonu` yapısı algoritmayı iki kola ayırır. Koşul doğruysa yalnız birinci kol, yanlışsa yalnız ikinci kol çalışır; iki kol hiçbir zaman birlikte çalışmaz. `Eğer Sonu` satırından sonra iki yol yeniden birleşir.

Elle izleme. Tablo 1.3.9 ilk iki testin izlemesidir. (4, 11) girdisinde koşul doğrudur ve `ise` kolu çalışır. (13, 11) girdisinde koşul yanlıştır; `Değilse` kolu önce ikinci sayıyı, yani küçük olanı yazar.

Test	Adım	birinciTamsayı	ikinciTamsayı	Koşul	Ekran
1	Oku birinciTamsayı	4			
	Oku ikinciTamsayı	4	11		
	Eğer 4 < 11	4	11	doğru	
	`ise` kolu	4	11		4 11
2	Oku birinciTamsayı	13			
	Oku ikinciTamsayı	13	11		
	Eğer 13 < 11	13	11	yanlış	
	`Değilse` kolu	13	11		11 13

Tablo 1.3.9: İki tam sayı algoritmasının (4, 11) ve (13, 11) için elle izlenmesi.

Eşit sayılar testi izlenmeden de sonuçlandırılabilir. (7, 7) için 7 < 7 koşulu yanlıştır ve `Değilse` kolu çalışır. Bu kol “ikinci, sonra birinci” yazar; iki sayı aynı olduğu için çıktı yine “7 7”dir. Koşulda `<` yerine `≤` kullanılsaydı eşit sayılarda `ise` kolu çalışırdı ve sonuç değişmezdi. Sınır durum testi burada bir hata yakalamaz ama algoritmanın eşit sayılarda da doğru olduğunu güvenceye alır.

Java kodu. Sözde kodun Java karşılığı aşağıdadır. Her `Yaz` bir `System.out.print` ya da `println`, her `Oku` bir `nextInt()` olmuştur.

```java
import java.util.Scanner;

public class IkiSayiSirala {
    public static void main(String[] args) {
        Scanner tarayici = new Scanner(System.in);
        System.out.print("Birinci tamsayıyı girin: ");
        int birinci = tarayici.nextInt();
        System.out.print("İkinci tamsayıyı girin: ");
        int ikinci = tarayici.nextInt();
        if (birinci < ikinci) {
            System.out.println(birinci + " " + ikinci);
        } else {
            System.out.println(ikinci + " " + birinci);
        }
        tarayici.close();
    }
}
```

Program JDK 21 ile derlenip üç testle çalıştırıldığında sırasıyla “4 11”, “11 13” ve “7 7” yazar; üç test de geçer. Çıktının başında “Birinci tam sayıyı girin: İkinci tam sayıyı girin: ” iletileri de görünür. Kullanıcı bir terminalde sayıları yazdığında her sayının ardından Enter'a bastığı için bu iletiler ayrı satırlara düşer.

Java'da `if` ve `else` kollarının her biri tek bir deyim ya da süslü parantezlerle sarılmış bir bloktur. Yukarıdaki kodda tek satırlık kollar da süslü parantez içine alınmıştır. Bu bir zorunluluk değildir ama iyi bir alışkanlıktır. Parantezsiz bir kola sonradan ikinci bir satır eklenirse o satır koşuldan bağımsız olarak her zaman çalışır ve girintisi bunu gizler.

1.3.3.2 Üç tam sayıyı sıralı yazdırma

Sorun tanımı. Girdi üç tam sayıdır; çıktı aynı sayıların küçükten büyüğe yazılmış hâlidir. Bu problem test senaryosu seçimini düşündürür. Üç sayı 3! = 6 farklı sırada gelebilir: 1 2 3, 1 3 2, 2 1 3, 2 3 1, 3 1 2 ve 3 2 1. Her sıra algoritmada farklı bir yoldan geçer; altı sıranın hepsi denenirse algoritmanın her kolu en az bir kez çalışmış olur. Bunlara eşit sayılı girdiler eklenir: 5 5 2 için beklenen çıktı 2 5 5'tir.

Fikir. Önce üç sayının en küçüğü bulunur ve yazılır. Geriye kalan iki sayı, iki tam sayı algoritmasıyla sıralanıp yazılır. En küçük sayı birinci, ikinci ya da üçüncü olabileceği için algoritmada üç ana kol vardır. Her ana kolun içinde de iki sayılık küçük bir karar bulunur.

```
Başla
    // İlk tamsayıyı kullanıcıdan al
    Yaz "Birinci tamsayıyı girin: "
    Oku birinciTamsayı
    // İkinci tamsayıyı kullanıcıdan al
    Yaz "İkinci tamsayıyı girin: "
    Oku ikinciTamsayı
    // Üçüncü tamsayıyı kullanıcıdan al
    Yaz "Üçüncü tamsayıyı girin: "
    Oku üçüncüTamsayı
    // Küçükten büyüğe sırala ve yazdır
    Eğer birinciTamsayı ≤ ikinciTamsayı ve
         birinciTamsayı ≤ üçüncüTamsayı ise
        Yaz birinciTamsayı
        Eğer ikinciTamsayı ≤ üçüncüTamsayı ise
            Yaz ikinciTamsayı, " ", üçüncüTamsayı
        Değilse
            Yaz üçüncüTamsayı, " ", ikinciTamsayı
        Eğer Sonu
    Değilse Eğer ikinciTamsayı ≤ birinciTamsayı ve
         ikinciTamsayı ≤ üçüncüTamsayı ise
        Yaz ikinciTamsayı
        Eğer birinciTamsayı ≤ üçüncüTamsayı ise
            Yaz birinciTamsayı, " ", üçüncüTamsayı
        Değilse
            Yaz üçüncüTamsayı, " ", birinciTamsayı
        Eğer Sonu
    Değilse
        Yaz üçüncüTamsayı
        Eğer birinciTamsayı ≤ ikinciTamsayı ise
            Yaz birinciTamsayı, " ", ikinciTamsayı
        Değilse
            Yaz ikinciTamsayı, " ", birinciTamsayı
        Eğer Sonu
    Eğer Sonu
Bitir
```

İlk koşul, birinci sayının iki sayıdan da küçük ya da onlara eşit olup olmadığını sorar. İki karşılaştırma "ve" bağlacıyla birleştirilmiştir: ikisi birden doğru olmalıdır. Bu kol çalışırsa birinci sayı yazılır, kalan ikinci ve üçüncü sayı iç içe bir kararla sıralanır. İlk koşul yanlışsa ikinci sayının en küçük olup olmadığına bakılır. O da yanlışsa en küçük sayı üçüncü sayıdır ve son kol çalışır. Son kolda ayrıca bir koşul yazılmasına gerek yoktur: ilk ikisi elendiyse geriye tek seçenek kalır.

Karşılaştırmalarda `<` yerine `≤` kullanılması eşit sayılar için gereklidir. Koşullar katı `<` ile yazılsın ve girdi 5 5 7 olsun. Birinci sayı için 5 < 5 yanlıştır; ikinci sayı için de 5 < 5 yanlıştır. Algoritma son kola düşer, en küçük sayı diye önce 7'yi yazar ve çıktı "7 5 5" olur. En küçük değer iki sayıda birden bulunduğunda hiçbir sayı "ötekilerden kesin küçük" değildir. `≤` ise "en küçük ya da en küçüklerden biri"

anlamını doğrudan ifade eder ve bu girdide birinci kolu çalıştırır. 5 5 2 testi bu hatayı yakalamaz, çünkü orada en küçük sayı tektir. Hatayı ancak en küçük değerin tekrarlandığı 5 5 7 gibi bir test ortaya çıkarır.

Sözde kodda bir gözlem saklıdır. Aynı "iki sayıyı sırala ve yaz" işi üç ayrı yerde, yalnız değişken adları değiştirilerek tekrarlanmıştır. Tekrarlanan bir iş, bir fonksiyona (Java'da metoda) dönüştürülmeye çok uygundur. `ikiSayiYaz(int a, int b)` gibi bir metot yazılırsa üç kolun her biri tek satıra iner. Metotlar ayrı bir konudur; burada önemli olan, tekrarı fark etme alışkanlığıdır.

1.3.3.3 Üç tam sayının Java kodu ve izlemesi

Sunumdaki tam Java programı aşağıdadır. Sözde koddaki üç ana kol `if … else if … else` zincirine, iç kararlar da tek satırlık `if … else` deyimlerine dönüşmüştür. "ve" bağlacı `&&` olmuştur.

```
import java.util.Scanner;

public class UcSayiSirala {
    public static void main(String[] args) {
        Scanner tarayici = new Scanner(System.in);
        System.out.print("Üç tamsayı girin: ");
        int birinci = tarayici.nextInt();
        int ikinci = tarayici.nextInt();
        int ucuncu = tarayici.nextInt();
        if (birinci <= ikinci && birinci <= ucuncu) {
            System.out.print(birinci + " ");
            if (ikinci <= ucuncu) System.out.println(ikinci + " " + ucuncu);
            else                  System.out.println(ucuncu + " " + ikinci);
        } else if (ikinci <= birinci && ikinci <= ucuncu) {
            System.out.print(ikinci + " ");
            if (birinci <= ucuncu) System.out.println(birinci + " " + ucuncu);
            else                   System.out.println(ucuncu + " " + birinci);
        } else {
            System.out.print(ucuncu + " ");
            if (birinci <= ikinci) System.out.println(birinci + " " + ikinci);
            else                   System.out.println(ikinci + " " + birinci);
        }
        tarayici.close();
    }
}
```

Bu programda bir ayrıntı daha vardır. `System.out.print("Üç tamsayı girin: ")` tek bir ileti yazar ve üç `nextInt()` çağrısı art arda gelir. `Scanner` sayıları boşlukla ya da satır sonuyla ayrılmış parçalar olarak okur. Kullanıcı "7 3 5" yazıp bir kez Enter'a basabilir ya da her sayıdan sonra Enter'a basabilir; iki durumda da üç sayı doğru okunur.

Kod izleme. Girdi 7, 3, 5 olsun. Tablo 1.3.10 her koşulun değerini ve ekrana yazılanları adım adım verir.

Adım	İncelenen ifade	Değer	Ekran
1	`birinci <= ikinci` → 7 ≤ 3	yanlış; `&&` 'nin sağına bakılmaz	
2	`ikinci <= birinci && ikinci <= ucuncu` → 3 ≤ 7 ve 3 ≤ 5	doğru; en küçük ikinci	
3	`System.out.print(ikinci + " ")`	—	3
4	`birinci <= ucuncu` → 7 ≤ 5	yanlış	3
5	`else` kolu: `println(ucuncu + " " + birinci)`	—	3 5 7

Tablo 1.3.10: `UcSayiSirala` programının (7, 3, 5) için izlenmesi. birinci = 7, ikinci = 3, ucuncu = 5.

Birinci adımda kısa devre davranışı görülür. `birinci <= ikinci` yanlış çıktığı anda `&&` ifadesinin tamamı yanlış olacağı kesinleşir; Java `birinci <= ucuncu` karşılaştırmasını hiç yapmaz. Bu örnekte atlanan karşılaştırma zararsızdır. Ama sağ taraf bir metot çağrısı ya da sıfıra bölme riski taşıyan bir ifade olduğunda kısa devre bilinçli olarak kullanılır. `b != 0 && a / b > 2` ifadesi `b` sıfırken bölmeyi hiç denemez.



Şekil 1.3.5: İzlemenin sonunda üç değişken ve üretilen sıra: 3 5 7.

Testler. Program altı sıranın her biri için “1 2 3” yazar. 5 5 2 için “2 5 5”, 7 3 5 için “3 5 7” yazar. Bu sonuçlar programın kendisi JDK 21'de çalıştırılarak elde edilmiştir. Ayrıca 5 2 5, 2 5 5, 5 5 7 ve 4 4 4 girdileri de denenmiş, hepsinde doğru sıra bulunmuştur. Aynı program katı `<` ile yeniden yazılıp denendiğinde 5 5 7 için “7 5 5” yazmıştır.

Sık yapılan hata

Üç sayı problemini bütün olasılıkları tek tek yazarak çözmeye çalışmak. Altı sıranın her biri için ayrı bir koşul yazılabilir: `if (a < b && b < c) … else if (a < c && c < b) …`. Sonuç altı kollu, okunması zor bir yapıdır. Eşit sayılar için koşulların bazıları hiçbir kola girmeyebilir ve program hiçbir şey yazmadan biter. “Önce en küçüğü bul, sonra kalanı sırala” fikri problemi küçük parçalara böler ve her parçayı ayrı ayrı doğrulamayı mümkün kılar.

1.3.4 Döngüyle algoritmalar

Karar yapıları algoritmanın hangi yoldan gideceğini belirler. Döngüler ise bir işi tekrar ettirir. Bu kesimdeki faktöriyel, Fibonacci ve üs alma algoritmaları aynı temel kalıba dayanır. Bir ya da iki değişken döngüden önce başlangıç değeri alır. Döngünün her turunda bu değişkenler güncellenir. Döngü bitin-

ce sonuç bu değişkenlerdedir. Kalıbın ayrıntıları algoritmadan algoritmaya değişir ve hatalar tam da bu ayrıntılarda çıkar.

1.3.4.1 Faktöriyel

> **Tanım 1.3.3**
>
> Negatif olmayan bir *n* tam sayısının **faktöriyeli**, 1'den *n*'ye kadar bütün tam sayıların çarpımıdır: *n! = 1 × 2 × 3 × … × n*. Ayrıca *0! = 1* kabul edilir.

0! = 1 kuralı keyfî görünebilir. Ama hiç sayı çarpılmadığında elde kalan değer, çarpmanın etkisiz elemanı olan 1'dir. Hiç sayı toplanmadığında sonucun 0 olması gibi. Faktöriyel, *n* farklı nesnenin kaç farklı sırada dizilebileceğini verir. Üç sayıyı sıralama probleminde 3! = 6 sıranın denenmesi bu yüzdendir. Tablo 1.3.11 test senaryolarını verir.

n	*n!*
0	1
1	1
3	6
5	120
10	3.628.800

Tablo 1.3.11: Faktöriyel için test senaryoları. 0 ve 1 sınır durumdur.

Sözde kod.

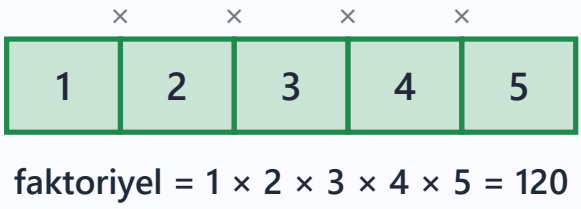
```
Başla
    // Bir sayıyı gir
    Oku sayı
    // Sonucu saklamak için bir değişkeni başlat
    faktoriyel = 1
    // Faktöriyel hesabı için döngü
    Döngü i = 1'den başlayarak sayı'ya kadar
        faktoriyel = faktoriyel * i
    Döngü Sonu
    Yaz "Girilen sayının faktöriyeli: ", faktoriyel
Bitir
```

`faktoriyel` değişkeni bir **biriktiricidir** (*accumulator*). Döngünün her turunda sonucun bir parçası onun üzerine eklenir; burada "eklemek" çarpmak demektir. Biriktiricinin başlangıç değeri, yapılan işlemin etkisiz elemanıdır: çarpma için 1, toplama için 0. Biriktirici 0 ile başlatılsaydı her çarpım 0 olurdu ve sonuç hep 0 çıkardı.

Elle izleme. *sayı = 5* için izleme Tablo 1.3.12'de görülür. Döngü değişkeni `i` 1'den başlar ve her turda bir artar. `i` 6 olduğunda *6 ≤ 5* koşulu yanlış çıkar ve döngü biter.

Tur	`i`	`i ≤ 5`	İşlem	`faktoriyel`
başlangıç	—	—	`faktoriyel = 1`	1
1	1	doğru	1 × 1	1
2	2	doğru	1 × 2	2
3	3	doğru	2 × 3	6
4	4	doğru	6 × 4	24
5	5	doğru	24 × 5	120
çıkış	6	yanlış	döngü biter; 120 yazılır	120

Tablo 1.3.12: Faktöriyel algoritmasının *sayı = 5* için izlenmesi.



Şekil 1.3.6: Döngü bitince beş çarpanın hepsi kullanılmıştır: 1 × 2 × 3 × 4 × 5 = 120.

sayı = 0 sınır durumunda döngü koşulu ilk denemede yanlıştır (*1 ≤ 0*); gövde hiç çalışmaz ve biriktiricinin başlangıç değeri 1 yazılır. Bu, *0! = 1* kuralıyla uyuşur. Algoritmanın 0 için ayrı bir kola ihtiyacı yoktur; doğru başlangıç değeri sınır durumu kendiliğinden çözer.

Java kodu.

```
import java.util.Scanner;

public class Faktoriyel {
    public static void main(String[] args) {
        Scanner tarayici = new Scanner(System.in);
        System.out.print("Bir sayı girin: ");
        int sayi = tarayici.nextInt();
        long faktoriyel = 1;
        for (int i = 1; i <= sayi; i++) {
            faktoriyel = faktoriyel * i;
        }
        System.out.println("Girilen sayının faktöriyeli: " + faktoriyel);
        tarayici.close();
    }
}
```

Sözde koddaki `Döngü i = 1'den başlayarak sayı'ya kadar` satırı Java'da `for (int i = 1; i <= sayi; i++)` olmuştur. `for` başlığının üç parçası vardır: başlangıç (`int i = 1`), devam koşulu (`i <= sayi`) ve her turun sonunda yapılacak güncelleme (`i++`). Sözde koddaki "sayı'ya kadar" ifadesi sayının kendisini de kapsar; bu yüzden koşul `<` değil `<=` ile yazılır.

Neden `long`? Faktöriyel çok hızlı büyür. Java'da `int` 32 bitlik bir tam sayıdır ve en çok 2.147.483.647 değerini tutar. 12! = 479.001.600 bu sınırın altındadır; 13! = 6.227.020.800 ise üstündedir. Biriktirici `int` tanımlansaydı program 13 için hiçbir hata iletisi vermeden 1.932.053.504 yazardı. Bu sayı 6.227.020.800'den $2^{32} = 4.294.967.296$ çıkarılarak elde edilen değerdir: `int` sınırı aşıldığında sayı başa sarar. Bu olaya **taşma** (*overflow*) denir. Java taşmayı denetlemez ve istisna fırlatmaz. `long` 64 bitliktir ve en çok 9.223.372.036.854.775.807 değerini tutar; 20! = 2.432.902.008.176.640.000 buna sığar. 21! için `long` da taşar ve program −4.249.290.049.419.214.848 gibi anlamsız bir negatif sayı yazar. Daha büyük faktöriyeller için `java.math.BigInteger` sınıfı kullanılır.

Sık yapılan hata

Biriktiriciyi döngünün içinde başlatmak. `long faktoriyel = 1;` satırı `for` gövdesinin içine yazılırsa her turda değişken yeniden 1 olur. Döngü sonunda değişkene erişilemez bile, çünkü gövdede tanımlanan bir değişken yalnız o gövdede geçerlidir. Derleyici `System.out.println` satırında `cannot find symbol` hatası verir. Biriktirici her zaman döngüden önce, döngünün dışında tanımlanır.

1.3.4.2 Fibonacci serisi

Tanım 1.3.4

Fibonacci serisinin ilk iki terimi 0 ve 1'dir; sonraki her terim kendisinden önceki iki terimin toplamıdır: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, …

Problem, kullanıcının girdiği *n* sayısı kadar terimi yazdırmaktır. Test senaryoları Tablo 1.3.13'te görülür. *n = 1* ve *n = 2* sınır durumlardır: bu durumlarda döngü hiç çalışmamalı, yalnız ilk terimler yazılmalıdır.

n (terim sayısı)	Beklenen çıktı
1	0
2	0 1
5	0 1 1 2 3
7	0 1 1 2 3 5 8

Tablo 1.3.13: Fibonacci için test senaryoları.

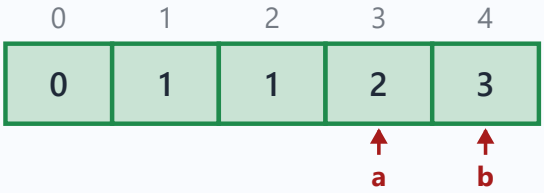
```
Başla
    // Kaç terim istendiğini gir
    Oku n
    // İlk iki Fibonacci sayısını başlat
    a = 0
    b = 1
    Eğer n ≥ 1 ise Yaz a
    Eğer n ≥ 2 ise Yaz b
    // Kalan terimleri hesapla
    Döngü i = 2'den başlayarak n − 1'e kadar
        sonuc = a + b
        a = b
        b = sonuc
        Yaz sonuc
    Döngü Sonu
Bitir
```

Algoritma bütün seriyi saklamaz; yalnız son iki terimi tutar. `a` sondan bir önceki, `b` son terimdir. Yeni terim `sonuc = a + b` ile hesaplanır. Ardından iki atama değerleri bir adım kaydırır: `a = b` eski son terimi "bir önceki" yapar, `b = sonuc` yeni terimi "son" yapar. İlk iki terim döngüden önce, *n*'nin yeterince büyük olup olmadığı denetlenerek yazılır. Döngü *i = 2*'den *n − 1*'e kadar döner, yani *n − 2* tur atar. Her tur bir terim yazdığı için toplam terim sayısı *2 + (n − 2) = n* olur.

Elle izleme. *n = 5* için izleme Tablo 1.3.14'te satır satır verilmiştir.

Aşama	`i`	`sonuc`	`a`	`b`	Ekran
başlangıç	—	—	0	1	
5 ≥ 1, 5 ≥ 2	—	—	0	1	0 1
tur 1	2	0 + 1 = 1	1	1	0 1 1
tur 2	3	1 + 1 = 2	1	2	0 1 1 2
tur 3	4	1 + 2 = 3	2	3	0 1 1 2 3
çıkış	5	5 ≤ 4 yanlış; döngü biter			0 1 1 2 3

Tablo 1.3.14: Fibonacci algoritmasının *n = 5* için izlenmesi.



Şekil 1.3.7: Döngü bittiğinde hesaplanan beş terim; `a` ve `b` son iki terimi gösterir.

Kaydırmanın sırası. `a = b;` ve `b = sonuc;` satırlarının sırası önemlidir. Sıra değiştirilip önce `b = sonuc` yazılırsa `b`'nin eski değeri kaybolur ve `a = b` satırı `a`'ya yeni terimi koyar. Bu durumda `a` ile `b` eşitlenir ve seri 0 1 1 2 4 8 16 … gibi ikiye katlanan bir diziye döner. Atamalarda değeri ezilecek değişken her zaman en son güncellenir.

Java kodu.

```java
import java.util.Scanner;

public class Fibonacci {
    public static void main(String[] args) {
        Scanner tarayici = new Scanner(System.in);
        System.out.print("Kaç terim? ");
        int n = tarayici.nextInt();
        long a = 0, b = 1, sonuc;
        if (n >= 1) System.out.print(a + " ");
        if (n >= 2) System.out.print(b + " ");
        for (int i = 2; i <= n - 1; i++) {
            sonuc = a + b;
            a = b;
            b = sonuc;
            System.out.print(sonuc + " ");
        }
        System.out.println();
        tarayici.close();
    }
}
```

Java kodu sözde kodla satır satır örtüşür. Tek fark, terim değişkenlerinin `long` seçilmesidir. Fibonacci terimleri de üstel hızla büyür. `int` kullanılırsa F(46) = 1.836.311.903 son doğru terim olur. F(47)'nin gerçek değeri 2.971.215.073'tür ve `int` sınırını aşar; program bunun yerine −1.323.752.223 yazar. Seriyi 0'dan sayarak F(0) = 0, F(1) = 1 kabul edildiğinde F(47), serinin 48. terimidir. Yani `int` 'li program *n = 47*'ye kadar doğru çalışır; *n = 48* olduğunda yazdığı son terim bozuktur. `long` ile F(92)'ye kadar doğru değerler elde edilir. Program *n = 8* için “0 1 1 2 3 5 8 13”, *n = 10* için “0 1 1 2 3 5 8 13 21 34” yazar.

Sık yapılan hata

Döngü sınırında bir fazla ya da bir eksik (*off-by-one*). Fibonacci döngüsü `i <= n - 1` yerine `i <= n` ile yazılırsa döngü bir tur fazla döner ve *n + 1* terim yazılır: *n = 5* için 0 1 1 2 3 5. Bu tür hatalar en kolay küçük bir testle elle izlenerek bulunur. *n = 1* ve *n = 2* için de ayrıca bakılmalıdır: bu iki durumda döngü hiç çalışmamalıdır.

1.3.4.3 Üs alma

Bir sayının kuvveti, aynı sayının kendisiyle tekrar tekrar çarpılmasıdır: $taban^{üs} = taban \times taban \times \ldots \times taban$ (üs kez). Ayrıca $taban^{0} = 1$'dir. Problem, negatif olmayan bir tam sayı üs için bu değeri hesaplamaktır. Tablo 1.3.15 test senaryolarını verir; üssün 0 olduğu satır sınır durumdur.

taban	üs	Beklenen
3	4	81

taban	üs	Beklenen
2	10	1024
5	0	1

Tablo 1.3.15: Üs alma için test senaryoları.

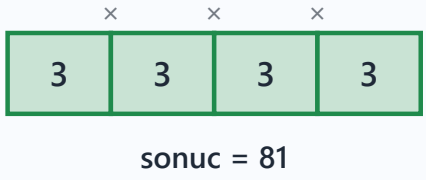
```
Başla
    // Bir sayıyı gir
    Oku taban
    // Bir üssü gir
    Oku üs
    // Sonucu saklamak için bir değişkeni başlat
    sonuc = 1
    // Üssü kullanarak sonucu hesapla
    Döngü i = 1'den başlayarak üs'e kadar
        sonuc = sonuc * taban
    Döngü Sonu
    Yaz "Sonuç: ", sonuc
Bitir
```

Bu algoritma faktöriyelle aynı kalıbı izler: biriktirici 1 ile başlar ve döngü her turda bir çarpma yapar. Fark, çarpanın her turda aynı olmasıdır. Faktöriyelde çarpan döngü değişkeni `i`'dir; burada hep `taban`'dır. `i` yalnız tur saymak için kullanılır. Üs 0 olduğunda döngü koşulu ($1 \le 0$) baştan yanlıştır, gövde hiç çalışmaz ve sonuç 1 kalır.

Tur	`i`	İşlem	`sonuc`
başlangıç	—	`sonuc = 1`	1
1	1	1 × 3	3
2	2	3 × 3	9
3	3	9 × 3	27
4	4	27 × 3	81
çıkış	5	5 ≤ 4 yanlış; “Sonuç: 81” yazılır	81

Tablo 1.3.16: Üs alma algoritmasının taban = 3, üs = 4 için izlenmesi.

Tablo 1.3.16 taban = 3, üs = 4 için dört turu gösterir. Döngü tam üs kadar döner; yani algoritmanın yaptığı çarpma sayısı üsse eşittir. 2^{10} için 10 çarpma yapılır ve sonuç 1024'tür.



Şekil 1.3.8: 3^4: dört kez 3 ile çarpma, sonuç 81.

Java kodu.

```
import java.util.Scanner;

public class UsAl {
    public static void main(String[] args) {
        Scanner tarayici = new Scanner(System.in);
        System.out.print("Taban: ");
        int taban = tarayici.nextInt();
        System.out.print("Üs: ");
        int us = tarayici.nextInt();
        long sonuc = 1;
        for (int i = 1; i <= us; i++) {
            sonuc = sonuc * taban;
        }
        System.out.println("Sonuç: " + sonuc);
        tarayici.close();
    }
}
```

Java'da üs işleci yoktur. `2 ^ 10` yazmak derlenir ama üs almaz: `^` bit düzeyinde "özel veya" işlecidir ve `2 ^ 10` ifadesinin değeri 8'dir. Hazır bir yol olarak `Math.pow(taban, us)` metodu vardır; ancak bu metot `double` döndürür. `Math.pow(3, 4)` 81 değil 81.0 verir ve çok büyük tam sayılarda `double`'ın sınırlı duyarlılığı yüzünden son basamaklar yanlış olabilir. Tam sayı kuvvetleri için yukarıdaki döngü güvenilir bir yoldur. Taşma burada da geçerlidir: `sonuc` `int` seçilseydi 2^{31} için −2.147.483.648, 2^{32} için 0 elde edilirdi. Program `long` ile 2^{31} için doğru değeri, 2.147.483.648'i yazar.

1.3.5 Metin işleme ve bayrak

Bu kesimin iki algoritması yeni araçlar getirir. Palindrom kontrolü sayılarla değil metinle çalışır ve Java'da metinlerin nasıl karşılaştırıldığı sorusunu gündeme getirir. Asal sayı kontrolü ise bir döngünün sonucunu tek bir mantıksal değişkende toplayan bayrak kalıbını ve döngüden erken çıkmayı tanıtır.

1.3.5.1 Palindrom kontrolü

Tanım 1.3.5

Palindrom, tersten okunuşu düz okunuşuyla aynı olan kelime ya da cümledir: kayak, abcba, "Ey Edip Adana'da pide ye". Cümlelerde büyük/küçük harf farkı, boşluklar ve noktalama işaretleri dikkate alınmaz.

Tanımdaki son cümle algoritmaya bir ön işlem ekler. "Ey Edip Adana'da pide ye" cümlesi harf harf tersine çevrildiğinde "ey edip ad'anadA pidE yE" olur ve ilk hâliyle aynı değildir. Büyük harfler, boşluklar ve kesme işareti karşılaştırmayı bozar. Bu yüzden metin önce **temizlenir**: bütün harfler küçük harfe çevrilir, harf ve rakam dışındaki karakterler atılır. Temizlenmiş metin "eyedipadanadapideye" olur ve bu metin gerçekten tersine eşittir.

Girdi	Temiz metin	Beklenen
abcba	abcba	palindrom
Kayak	kayak	palindrom
Kitap	kitap	değil
Ey Edip Adana'da pide ye	eyedipadanadapideye	palindrom

Tablo 1.3.17: Palindrom kontrolü için test senaryoları.

"Kayak" testi büyük harfin temizlendiğini, "Kitap" testi palindrom olmayan bir girdinin doğru reddedildiğini, son test de boşluk ve noktalamanın atıldığını sınar (Tablo 1.3.17). Sözde kod üç adımdan oluşur: temizle, ters çevir, karşılaştır.

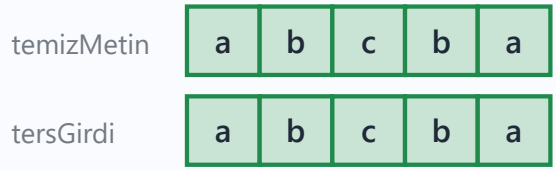
```
Başla
    // Bir kelime veya cümlenin girilmesini iste
    Oku girdiMetin
    // Girdiyi temizle: küçük harf, boşluk ve noktalama yok
    temizMetin = Temizle(girdiMetin)
    // Temizlenmiş girdiyi ters çevir
    tersGirdi = TersÇevir(temizMetin)
    // Temizlenmiş girdi ile ters girdiyi karşılaştır
    Eğer temizMetin = tersGirdi ise
        Yaz "Girilen metin bir palindromdur."
    Değilse
        Yaz "Girilen metin bir palindrom değildir."
    Eğer Sonu
Bitir
```

`Temizle` ve `TersÇevir` sözde kodda tek satırlık işlemler olarak yazılmıştır. Her biri aslında kendi başına küçük bir algoritmadır. Sözde kodun bu esnekliği yararlıdır: ana fikir önce kurulur, alt işlerin ayrıntısı sonraya bırakılır.

Elle izleme. “abcba” ve “Kitap” girdileri için izleme Tablo 1.3.18'de görülür.

Adım	“abcba”	“Kitap”
Oku girdiMetin	abcba	Kitap
temizMetin = Temizle(…)	abcba (değişiklik yok)	kitap
tersGirdi = TersÇevir(…)	abcba	patik
temizMetin = tersGirdi ?	doğru	yanlış (ilk harfte k ≠ p)
Ekran	Girilen metin bir palindromdur.	Girilen metin bir palindrom değildir.

Tablo 1.3.18: Palindrom algoritmasının iki test için izlenmesi.



Şekil 1.3.9: “abcba” için temiz metin ve tersi harf harf eşleşir.

Java kodu. Aşağıdaki programda temizleme işi `temizle` adlı ayrı bir metoda taşınmıştır. Ters çevirme için Java'nın hazır `StringBuilder` sınıfı kullanılır.

```
import java.util.Locale;
import java.util.Scanner;

public class Palindrom {
    public static void main(String[] args) {
        Scanner tarayici = new Scanner(System.in);
        System.out.print("Bir kelime veya cümle girin: ");
        String girdiMetin = tarayici.nextLine();
        String temizMetin = temizle(girdiMetin);
        String tersGirdi = new StringBuilder(temizMetin).reverse().toString();
        if (temizMetin.equals(tersGirdi)) {
            System.out.println("Girilen metin bir palindromdur.");
        } else {
            System.out.println("Girilen metin bir palindrom değildir.");
        }
        tarayici.close();
    }

    static String temizle(String metin) {
        String kucuk = metin.toLowerCase(Locale.forLanguageTag("tr"));
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < kucuk.length(); i++) {
            char c = kucuk.charAt(i);
            if (Character.isLetterOrDigit(c)) sb.append(c);
        }
        return sb.toString();
    }
}
```

`nextLine()` satırın tamamını okur. Burada `next()` kullanılsaydı yalnız ilk kelime, “Ey”, okunurdu; çünkü `next()` boşlukta durur. Cümle girilebilecek her yerde `nextLine()` gerekir.

`new StringBuilder(temizMetin).reverse().toString()` üç işi zincir hâlinde yapar. Önce metnin değiştirilebilir bir kopyası oluşturulur, sonra bu kopya yerinde ters çevrilir, en son yeniden `String`'e dönüştürülür. Java'da `String` nesneleri değiştirilemez; bir metni ters çevirmek, birleştirmek ya da harflerini değiştirmek her zaman yeni bir nesne üretir. `StringBuilder` ise değiştirilebilir bir karakter dizisidir ve bu tür işler için tasarlanmıştır.

`temizle` metodu metni küçük harfe çevirir, sonra karakterler üzerinde tek tek gezinir. `charAt(i)` metodu `i`. konumdaki karakteri verir; konumlar 0'dan başlar ve `length() - 1`'de biter. `Character.isLetterOrDigit(c)` karakterin bir harf ya da rakam olup olmadığını söyler. Türkçe harfler de harf sayılır; boşluk, virgül ve kesme işareti sayılmaz. Uygun karakterler bir `StringBuilder`'a eklenir ve sonuç döndürülür.

Metin karşılaştırma: `equals` ve `==`. Programın en önemli satırı `if (temizMetin.equals(tersGirdi))` satırıdır. Bu satır `temizMetin == tersGirdi` biçiminde yazılırsa program derlenir, çalışır ama “abcba” için bile “palindrom değildir” der. Sebep, `==` işlecinin nesneler için ne anlama geldiğidir. `int`, `double`, `char` gibi temel tiplerde `==` değerleri karşılaştırır. `String`

ise bir nesnedir ve `String` tipindeki bir değişken nesnenin kendisini değil, bellekteki yerini, yani nesneye bir başvuruyu (*reference*) tutar. `==` iki başvurunun aynı nesneyi gösterip göstermediğine bakar. `tersGirdi`, `StringBuilder`'ın yeni ürettiği ayrı bir nesnedir; içeriği `temizMetin`'le aynı olsa da başka bir nesnedir ve `==` `false` verir. İçerik karşılaştırması `equals` metoduyla yapılır.

Bu hata bir de sinsi bir biçimde ortaya çıkar. Kaynak kodda doğrudan yazılmış iki aynı metin, `String a = "abc"; String b = "abc";`, Java'da tek bir nesne olarak saklanır ve `a == b` `true` verir. Programcı küçük bir denemede `==`'nin çalıştığını görür ve yanlış bir güvene kapılır. Kullanıcıdan okunan ya da çalışma sırasında üretilen metinlerde ise `==` hemen her zaman `false` verir. JDK 21'de yapılan denemede iki sabit metin için `true`, "abcba" ile `StringBuilder`'ın ürettiği tersi için `false` elde edilmiştir. Kural basittir: metinler her zaman `equals` ile karşılaştırılır.

Sık yapılan hata

Türkçe metni yerel ayar belirtmeden küçültmek. `toLowerCase()` metodu parametresiz çağrıldığında bilgisayarın yerel ayarını kullanır. Büyük "I" harfi Türkçe ayarda "ı"ya, İngilizce ayarda "i"ye dönüşür. Aynı program farklı bilgisayarlarda farklı sonuç verebilir. Yukarıdaki kodda `toLowerCase(Locale.forLanguageTag("tr"))` yazılarak Türkçe kurallar sabitlenmiştir. Bu çağrıyla "Ilık" "ılık"a dönüşür; `Locale.ROOT` ile aynı kelime "ilık" olurdu. Kelimeler İngilizce ise `Locale.ROOT` seçilir.

Testler. Program JDK 21'de dört testle çalıştırılmıştır: "abcba", "Kayak" ve "Ey Edip Adana'da pide ye" için "Girilen metin bir palindromdur.", "Kitap" için "Girilen metin bir palindrom değildir." yazar.

1.3.5.2 Asal sayı kontrolü

Tanım 1.3.6

Asal sayı, 1'den büyük olup yalnız 1'e ve kendisine bölünebilen doğal sayıdır: 2, 3, 5, 7, 11, 13, … 1'den büyük olup asal olmayan sayılara **bileşik sayı** denir.

Tanım iki sınır durumu açıkça dışarıda bırakır: 0 ve 1 asal değildir. 2 ise hem en küçük hem tek çift asal sayıdır. Test senaryoları bu durumları ve hem asal hem bileşik sıradan sayıları içerir (Tablo 1.3.19).

Girdi	Beklenen	Sınanan durum
0, 1	asal değil	Sınır: döngü hiç çalışmaz
2	asal	Sınır: en küçük asal, döngü hiç çalışmaz
7	asal	Döngü sonuna kadar döner
21	asal değil	Bölen erken bulunur

Girdi	Beklenen	Sınanan durum
25	asal değil	Bir asalın karesi

Tablo 1.3.19: Asal sayı kontrolü için test senaryoları.

```
Başla
    // Bir sayıyı gir
    Oku sayı
    // Bayrak (flag): 0 ve 1 asal değildir
    Eğer sayı < 2 ise asalMi = False
    Değilse asalMi = True
    // 2'den sayının yarısına kadar bölenleri dene
    Döngü bölen = 2'den başlayarak sayı / 2'ye kadar
        Eğer sayı % bölen = 0 ise
            // Bölünüyorsa sayı asal değildir
            asalMi = False
            Döngüyü Kır
        Eğer Sonu
    Döngü Sonu
    // Bayrağa göre sonucu yazdır
    Eğer asalMi = True ise
        Yaz sayı, " bir asal sayıdır."
    Değilse
        Yaz sayı, " bir asal sayı değildir."
    Eğer Sonu
Bitir
```

Algoritma üç fikre dayanır.

Bayrak. `asalMi` değişkeni bir **bayraktır** (*flag*): yalnız doğru ya da yanlış değer alan ve bir durumun gerçekleşip gerçekleşmediğini kaydeden değişken. Başta sayının asal olduğu varsayılır ve bayrak kaldırılır. Döngü bir bölen bulursa bayrak indirilir. Döngü bittiğinde bayrağın durumu sonucu söyler. Bayrak kalıbı “hepsi için doğru mu?” türündeki soruların genel çözümüdür: iyimser bir varsayımla başlanır, tek bir karşı örnek varsayımı çürütür.

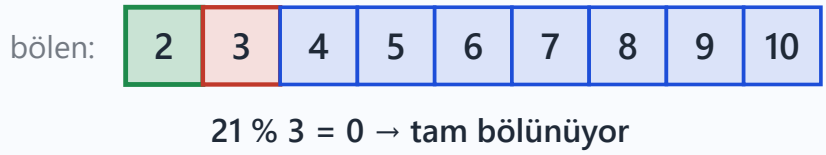
Kalan ve erken çıkış. `sayı % bölen = 0` koşulu bölmenin kalansız olup olmadığını sınar: 21 % 3 = 0 olduğu için 3, 21'i tam böler. Bir bölen bulunduğu anda sayının asal olmadığı kesinleşir; geri kalan bölenleri denemek gereksizdir. `Döngüyü Kır` (Java'da `break`) döngüden hemen çıkar.

Döngünün sınırı. Bölenler 2'den `sayı / 2`'ye kadar denenir. Sayının kendisi hariç hiçbir böleni sayının yarısından büyük olamaz. Bir *d* böleni *sayı / 2*'den büyükse *sayı / d* bölümü 2'den küçük olur, yani 1 olur; bu da *d = sayı* demektir. `sayı / 2` tam sayı bölmesidir: 7 / 2 = 3. 0 ve 1 için sınır 0'dır; döngü hiç çalışmaz ve sonucu ilk satırdaki `Eğer sayı < 2` kararı belirler. 2 ve 3 için sınır 1'dir; döngü yine çalışmaz, bayrak doğru kalır ve iki sayı da doğru biçimde asal bulunur.

Elle izleme. Tablo 1.3.20 7 ve 21 için izlemeyi verir.

Sayı	Adım	`bölen`	Kalan	`asalMi`
7 (sınır 7 / 2 = 3)	7 ≥ 2, bayrak kalkar	—	—	True
	tur 1	2	7 % 2 = 1	True
	tur 2	3	7 % 3 = 1	True
	çıkış: 4 ≤ 3 yanlış	4	—	True
	“7 bir asal sayıdır.”			
21 (sınır 21 / 2 = 10)	21 ≥ 2, bayrak kalkar	—	—	True
	tur 1	2	21 % 2 = 1	True
	tur 2	3	21 % 3 = 0	False
	Döngüyü Kır	3	4 … 10 denenmez	False
	“21 bir asal sayı değildir.”			

Tablo 1.3.20: Asal sayı algoritmasının 7 ve 21 için izlenmesi.



Şekil 1.3.10: 21 için denenebilecek bölenler 2 … 10'dur. 2 bölmez, 3 böler; döngü kırılır ve 4 … 10 hiç denenmez.

Java kodu.

```
import java.util.Scanner;

public class AsalMi {
    public static void main(String[] args) {
        Scanner tarayici = new Scanner(System.in);
        System.out.print("Bir sayı girin: ");
        int sayi = tarayici.nextInt();
        boolean asalMi = true;
        if (sayi < 2) asalMi = false;
        for (int bolen = 2; bolen <= sayi / 2; bolen++) {
            if (sayi % bolen == 0) {
                asalMi = false;
                break;
            }
        }
        if (asalMi) {
            System.out.println(sayi + " bir asal sayıdır.");
        } else {
            System.out.println(sayi + " bir asal sayı değildir.");
        }
        tarayici.close();
    }
}
```

Java'da bayrak `boolean` tipinde bir değişkendir ve değerleri `true` ile `false`'tur. Sözde koddaki `Eğer asalMi = True ise` satırı Java'da `if (asalMi)` biçiminde yazılmıştır. `asalMi` zaten bir `boolean` olduğu için onu `true` ile karşılaştırmaya gerek yoktur. `if (asalMi == true)` de doğru çalışır ama gereksiz uzundur. `if (asalMi = true)` ise bir atamadır: derlenir, `asalMi`'yi `true` yapar ve her zaman "asal" sonucunu verir. `boolean` değişkenlerde tek eşittir hatası derleyiciden kaçar.

`break` deyimi içinde bulunduğu en yakın döngüyü sonlandırır. Program akışı döngüden sonraki ilk satıra, burada `if (asalMi)` satırına geçer. `break` kullanılmasaydı sonuç değişmezdi; bayrak bir kez indirildikten sonra yeniden kaldırılmaz. Ama döngü gereksiz yere 4'ten 10'a kadar yedi bölen daha denerdi.

Kaç bölen denenir? Programın denediği bölen sayıları sayaçlı bir kopyayla JDK 21'de ölçülmüştür. 7 için 2 bölen (2, 3) denenir ve sayı asal bulunur. 21 için 2 bölen denenir; 3'te döngü kırılır. 25 için 4 bölen (2, 3, 4, 5) denenir. 49 için 6 bölen denenir ve 7'de bölen bulunur. 97 asaldır; döngü 2'den 48'e kadar 47 bölenin hepsini dener. Asal sayılar algoritmanın en kötü durumudur, çünkü erken çıkış hiç gerçekleşmez.

Daha iyi bir sınır. Sınır `sayı / 2` yerine $\sqrt{sayı}$ alınabilir. Bir sayı $a \times b$ biçiminde iki çarpana ayrılıyorsa çarpanlardan biri $\sqrt{sayı}$'dan büyük değildir. Bu yüzden $\sqrt{sayı}$'ya kadar bölen bulunamazsa sayı asaldır. 97 için bu sınır 9'dur ve 47 yerine 8 bölen denenir. Java'da koşul `bolen * bolen <= sayi` biçiminde yazılır; böylece karekök almaya ve ondalıklı sayılara gerek kalmaz. Büyük sayılarda fark çarpıcıdır: bir milyar civarındaki bir asal için `sayı / 2` sınırı yaklaşık 500 milyon, $\sqrt{sayı}$ sınırı yaklaşık 31.600 deneme demektir.

Sık yapılan hata

Sınır durumları unutmak. `if (sayi < 2) asalMi = false;` satırı çıkarılırsa 0 ve 1 için döngü hiç çalışmaz, bayrak `true` kalır ve program iki sayıyı da asal ilan eder. Faktöriyelde 0!, Fibonacci'de $n = 1$, asal kontrolünde 0 ve 1 ayrıca test edilmelidir. Bu girdiler algoritmanın döngüye hiç girmediği durumlardır ve doğru sonuç tamamen başlangıç değerlerine bağlıdır.

1.3.6 Arama algoritmaları

1.3.6.1 Arama problemi

Arama, bir koleksiyonun içinde belirli bir öğenin bulunup bulunmadığını ve bulunuyorsa nerede olduğunu sorgular. Veri işleyen hemen her programın temel işlemlerinden biridir. Kütüphanede bir kitabın yeri, bir web sitesinde belirli bir sayfa, veri tabanında bir kayıt, telefon rehberinde bir isim hep bir aramayla bulunur. Problemi soyut olarak tanımlamak için bir sepet ve bir elma düşünülsün.

Tanım 1.3.7

Arama problemi. Girdi, nesnelerden oluşan bir koleksiyon ("Sepet") ve belirli bir nesnedir ("Elma"). Çıktı, elma sepetin içindeyse `true`, değilse `false`'tur. Programlarda genellikle bunun yerine bulunan öğenin **indisi**, yani dizideki konumu döndürülür; öğe bulunamazsa −1 döndürülür.

İndis döndürmek, `true` / `false` döndürmekten daha fazla bilgi verir: öğenin varlığı da konumu da tek bir sayıyla bildirilir. Java dizilerinde geçerli indisler 0'dan `length - 1`'e kadardır; −1 hiçbir zaman geçerli bir indis olamayacağı için "bulunamadı" anlamında güvenle kullanılır. Bu notta üç arama yöntemi incelenir (Şekil 1.3.11). Rastgele arama her adımda rastgele bir öğeye bakar. Doğrusal arama öğelere baştan sona sırayla bakar. İkili arama ortadaki öğeye bakar ve her adımda arama alanının yarısını eler.



Şekil 1.3.11: Üç arama yöntemi. Rastgele arama: sıra gerekmez, bulma garantisi yoktur. Doğrusal arama: sıra gerekmez, en fazla n karşılaştırma. İkili arama: dizi sıralı olmalıdır, en fazla yaklaşık $\log_2 n$ karşılaştırma.

Yöntemleri karşılaştırmanın ölçüsü **karşılaştırma sayısıdır**: aranan öğenin dizideki bir öğeyle kaç kez karşılaştırıldığı. Bu sayı makineden, dilden ve derleyiciden bağımsızdır. Bir yöntemin bir milyon öğede kaç karşılaştırma yaptığı biliniyorsa hangi bilgisayarda çalışırsa çalışsın ötekilerden ne kadar hızlı olacağı da kestirilebilir.

1.3.6.2 Rastgele arama

Rastgele arama (*random search*) koleksiyondaki öğeleri rastgele bir sırayla kontrol eder. Her adımda rastgele bir öğe seçilir ve aranan öğeyle karşılaştırılır. Eşleşme varsa arama biter; yoksa yeni bir rastgele öğe seçilir. Sözde kod dört adımdır:

1. Rastgele bir öğe seç.
2. Seçilen öğeyi hedefle karşılaştır.
3. Hedef bulunduysa işlemi sonlandır.
4. Bulunamadıysa adımları tekrarla.

Şekil 1.3.12 dokuz öğeli sırasız bir dizide 48'in aranmasını gösterir. Rastgele seçilen öğeler sırasıyla 23, 42, 7 ve 48'dir; arama dördüncü denemede biter. Aynı arama yeniden çalıştırılsa başka öğeler seçilir ve deneme sayısı da değişir.



Şekil 1.3.12: Rastgele arama, hedef 48. Kırmızı çerçeveli öğeler (23, 42, 7) daha önce seçilip eşleşmeyenlerdir; dördüncü seçim 48'dir.

Java kodu. Aşağıdaki metot `java.util.Random` sınıfıyla rastgele indisler üretir. Metodun derlenmesi için dosyanın başında `import java.util.Random;` satırı bulunmalıdır.

```
static int rastgeleAra(int[] dizi, int hedef, int denemeSiniri) {
    Random rastgele = new Random();
    for (int deneme = 1; deneme <= denemeSiniri; deneme++) {
        int i = rastgele.nextInt(dizi.length);   // 0 … length - 1
        if (dizi[i] == hedef) {
            return i;
        }
    }
    return -1;   // sınır doldu; öğe yok ya da şans yaver gitmedi
}
```

`rastgele.nextInt(dizi.length)` 0 ile `dizi.length - 1` arasında, iki uç dahil, rastgele bir tam sayı üretir; bu aralık tam olarak dizinin geçerli indisleridir. Metodun üçüncü parametresi `denemeSiniri`'dir. Rastgele aramanın en büyük zaafı, hedef dizide yoksa durmamasıdır: her deneme başarısız olur ve döngü sonsuza dek döner. Bu, algoritmanın sonluluk özelliğini bozar. Deneme sınırı bu sorunu çözer ama yeni bir belirsizlik getirir. Sınır dolduğunda −1 döner. Bu −1'in "öğe yok" mu, yoksa "öğe var ama şans yaver gitmedi" mi demek olduğu bilinemez.

Kaç deneme gerekir? Dizide n öğe ve tek bir hedef varsa her denemenin başarı olasılığı $1/n$'dir. Denemeler birbirinden bağımsız olduğu için ortalama deneme sayısı n'dir. Dokuz öğeli dizide 48'in aranması JDK 21'de 200.000 kez tekrarlanmış ve ortalama 9,02 deneme ölçülmüştür. n denemede hiç bulamama olasılığı $(1 - 1/n)^n$'dir. $n = 9$ için bu değer yaklaşık 0,346'dır: her üç aramadan biri, öğe sayısı kadar denemeden sonra hâlâ sonuçsuzdur. Aynı öğe tekrar tekrar seçilebildiği için doğrusal aramanın en kötü durumundan bile daha fazla deneme gerekebilir.

Rastgele aramanın avantajları basitliği ve verinin sırasından bağımsız olmasıdır. Dezavantajları ağırdır: kesin bir sonuç garantisi yoktur, aynı öğeyi defalarca seçebilir ve hedef yoksa ancak bir deneme sınırıyla durdurulabilir. Bu yöntem pratikte tek başına bir arama aracı olarak pek kullanılmaz. Değeri başka bir yerdedir: arama uzayı denenemeyecek kadar büyük olduğunda, örneğin bir optimizasyon probleminde, rastgele örnekleme işe yarar bir başlangıç noktasıdır.

1.3.6.3 Doğrusal arama

Doğrusal arama (*linear search*) bir koleksiyonda öğe aramanın en temel yoludur. Öğeler ilkinden başlanarak sırayla hedefle karşılaştırılır. Eşleşme bulunursa konum döndürülür. Listenin sonuna gelinmişse öğe yoktur. Öğelerin sıralı olması gerekmez.

1. Öğeleri bir diziye koy.
2. İlk öğeden başlayarak sırayla hedef öğeyle karşılaştır.
3. Hedef bulunursa işlemi sonlandır ve konumunu döndür.
4. Bulunmazsa bir sonraki öğeye geç.
5. Listenin sonuna kadar devam et; sona gelindiyse öğe yoktur.

```
static int dogrusalAra(int[] dizi, int hedef) {
    for (int i = 0; i < dizi.length; i++) {
        if (dizi[i] == hedef) {
            return i;
        }
    }
    return -1;
}
```

Metot dizinin her indisini sırayla dener. `dizi[i] == hedef` doğru olduğunda `return i;` hem metodu hem döngüyü anında bitirir; `break`'e gerek yoktur. Döngü hiçbir eşleşme bulmadan biterse akış son satıra ulaşır ve −1 döner. `i < dizi.length` koşulu dizinin son geçerli indisi olan `length - 1`'de durur.

İzleme. [42, 17, 8, 23, 31] dizisinde 23 aranırken 42, 17 ve 8 eşleşmez; dördüncü karşılaştırmada `dizi[3] = 23` bulunur ve 3 döner (Şekil 1.3.13). Tablo 1.3.21 aynı dizide başka hedefler için karşılaştırma sayılarını verir. Değerler sayaçlı bir kopya JDK 21'de çalıştırılarak doğrulanmıştır.



Şekil 1.3.13: Doğrusal arama, hedef 23. Solda ilk karşılaştırma (`i = 0`, 42); sağda dördüncü karşılaştırmada eşleşme (`i = 3`).

Hedef	Bakılan öğeler	Karşılaştırma	Dönen değer
42	42	1 (en iyi durum)	0
23	42, 17, 8, 23	4	3
31	42, 17, 8, 23, 31	5	4
50	42, 17, 8, 23, 31	5 (en kötü durum)	−1

Tablo 1.3.21: [42, 17, 8, 23, 31] dizisinde doğrusal arama.

Maliyet. *n* öğeli bir dizide en iyi durum hedefin ilk öğe olmasıdır: 1 karşılaştırma. En kötü durum hedefin son öğe olması ya da hiç olmamasıdır: *n* karşılaştırma. Hedef dizide varsa ve her konumda bulunma olasılığı eşitse ortalama *(1 + 2 + … + n) / n = (n + 1) / 2* karşılaştırma yapılır. İş *n* ile doğru orantılı büyür: dizi iki katına çıkarsa en kötü durumdaki iş de iki katına çıkar. Algoritmanın adı bu doğrusal büyümeden gelir.

Doğrusal arama basit ve anlaşılırdır, sırasız veride çalışır ve sonucu kesindir: öğe varsa mutlaka bulunur. Küçük koleksiyonlarda en iyi seçimdir. Dezavantajı büyük koleksiyonlarda yavaş kalmasıdır. Bir milyon öğeli bir dizide olmayan bir öğe için bir milyon karşılaştırma gerekir.

1.3.6.4 İkili arama

İkili arama (*binary search*), sıralı veride öğe bulmanın hızlı ve etkili bir yoludur. Fikir, bir sözlükte kelime aramaya benzer. Sözlük ortadan açılır; aranan kelime açılan sayfadan önce geliyorsa sol yarıya, sonra geliyorsa sağ yarıya geçilir. Her adımda bakılacak sayfa sayısı yarıya iner.

1. Ortadaki öğeyi seç ve hedefle karşılaştır.
2. Hedef bulunursa işlemi sonlandır.
3. Hedef ortadakinden küçükse sol yarıya odaklan.
4. Hedef ortadakinden büyükse sağ yarıya odaklan.
5. Hedef bulunana ya da arama alanı tükenene kadar tekrarla.

Üçüncü ve dördüncü adımlar dizinin sıralı olmasına dayanır. Hedef ortadaki öğeden küçükse ve dizi küçükten büyüğe sıralıysa, sağ yarıdaki her öğe ortadakinden büyük ya da ona eşittir; hedef orada olamaz. Bu çıkarım ancak sıralı dizide geçerlidir.

```
static int ikiliAra(int[] dizi, int hedef) {
    int alt = 0, ust = dizi.length - 1;
    while (alt <= ust) {
        int orta = (alt + ust) / 2;
        if (dizi[orta] == hedef) {
            return orta;
        } else if (hedef < dizi[orta]) {
            ust = orta - 1;
        } else {
            alt = orta + 1;
        }
    }
    return -1;
}
```

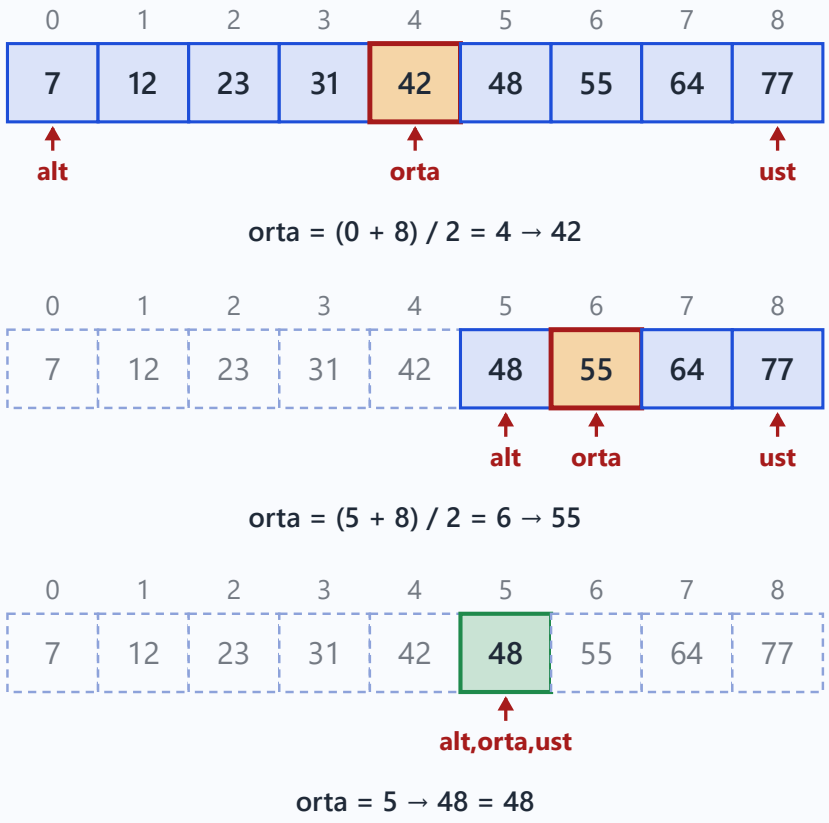
Arama alanı `alt` ve `ust` indisleri arasındaki bölgedir, iki uç dahil. Başta bütün dizidir. Her turda `orta = (alt + ust) / 2` hesaplanır. Bu tam sayı bölmesidir; alanın öğe sayısı çiftse iki ortadan soldakini seçer. Hedef ortadaki öğeye eşitse indis döner. Küçükse `ust = orta - 1` ile sağ yarı ve orta

öğe elenir. Büyükse `alt = orta + 1` ile sol yarı ve orta öğe elenir. Orta öğe iki durumda da yeniden incelenmez; zaten hedefle karşılaştırılmıştır. Döngü koşulu `alt <= ust`'tür. `alt`, `ust`'ü geçtiğinde arama alanı boşalmıştır ve öğe yoktur.

İzleme. Sıralı [7, 12, 23, 31, 42, 48, 55, 64, 77] dizisinde 48 aransın. Tablo 1.3.22 her turdaki değişkenleri, Şekil 1.3.14 üç anı gösterir.

Tur	`alt`	`ust`	`orta`	`dizi[orta]`	Karar
1	0	8	(0 + 8) / 2 = 4	42	48 > 42 → `alt = 5`
2	5	8	(5 + 8) / 2 = 6	55	48 < 55 → `ust = 5`
3	5	5	(5 + 5) / 2 = 5	48	eşit → `return 5`

Tablo 1.3.22: İkili arama, [7, 12, 23, 31, 42, 48, 55, 64, 77] dizisinde hedef 48.



Şekil 1.3.14: İkili aramanın üç turu. Kesikli çerçeveli öğeler elenmiştir. İkinci turda arama alanında dört öğe vardır (48, 55, 64, 77); iki ortadan soldaki, 55 seçilir. Dokuz öğeli dizide yalnız 3 karşılaştırma yapılmıştır.

Örnek 1.3.1

Soru. Aynı dizide 50 aranırsa hangi öğelere bakılır ve metot ne döndürür?

Çözüm. İlk iki tur 48 aramasıyla aynıdır: `orta = 4` (42) ve 50 > 42 olduğu için `alt = 5`; `orta = 6` (55) ve 50 < 55 olduğu için `ust = 5`. Üçüncü turda `orta = 5`, `dizi[5] = 48`'dir ve 50 > 48 olduğu için `alt = 6` olur. Şimdi `alt = 6`, `ust = 5`'tir; `alt <= ust` yanlıştır ve döngü biter. Metot 3 karşılaştırmadan sonra −1 döndürür. Son durumda `alt`, 50'nin dizide bulunması gereken konumu gösterir: 48 ile 55'in arası, indis 6.

Aynı dizide 12 için iki karşılaştırma yeter (42 → 12). 77 için dört karşılaştırma gerekir (42 → 55 → 64 → 77). Dokuz öğeli bu dizide hiçbir arama dört karşılaştırmayı aşmaz. Bu sayılar sayaçlı bir kopyanın JDK 21'de çalıştırılmasıyla doğrulanmıştır.

Sırasız dizide ikili arama. İkili aramanın tehlikeli yanı, sıralı olmayan bir dizide hata vermeden yanlış sonuç üretmesidir. [77, 42, 64, 12, 23, 55, 7, 48, 31] dizisinde 77 aransın. İlk ortadaki öğe 23'tür; 77 > 23 olduğu için sağ yarıya geçilir. Sonra 7'ye, 48'e ve 31'e bakılır; 77 hepsinden büyük olduğu için hep sağa gidilir. Sonunda alan boşalır ve −1 döner. Oysa 77 dizinin ilk öğesidir. Metot dizinin sıralı olup olmadığını denetlemez, uyarı da vermez. Bu dizinin dokuz öğesinin her biri ayrı ayrı arandığında yalnız ikisi (23 ve 48) bulunur. Bulunanlar da şans eseri orta konumlara denk gelmiştir. Sıralı olmayan veride sonuç rastlantıya kalır; bu yüzden "sırasız dizide yanlış sonuç verebilir" demek daha doğrudur.

Kaç karşılaştırma? Her tur arama alanını yarıya indirir. n öğeli bir alan bir turdan sonra en çok $n/2$, iki turdan sonra en çok $n/4$ öğeye iner. Alan tek öğeye inene kadar yaklaşık $\log_2 n$ tur geçer; son öğeye de bir kez bakılır. En kötü durumdaki karşılaştırma sayısı $\lfloor \log_2 n \rfloor + 1$'dir. 1.000.000 öğe için $\log_2 1.000.000 \approx 19{,}93$; tabanı alınıp bir eklenince 20 karşılaştırma bulunur. Doğrusal arama aynı dizide bir milyon karşılaştırma yapabilir.

Sık yapılan hata

Sınırlarda bir fazla ya da eksik. Döngü koşulu `alt < ust` yazılırsa alanda tek öğe kaldığında döngü durur ve o öğeye hiç bakılmaz: [7, 12, …, 77] dizisinde 48 ararken üçüncü tur hiç çalışmaz ve −1 döner. `ust = orta` ya da `alt = orta` yazılırsa alan bazı durumlarda küçülmez ve döngü sonsuza dek döner. Doğru kalıp şudur: koşul `alt <= ust`, güncellemeler `orta - 1` ve `orta + 1`.

Java'nın hazır ikili araması. `java.util.Arrays` sınıfındaki `Arrays.binarySearch(dizi, hedef)` metodu aynı algoritmayı uygular. Öğe bulunursa indisini döndürür. Bulunamazsa −1 yerine `-(eklemeNoktası) - 1` biçiminde negatif bir sayı döndürür. Ekleme noktası, öğenin sırayı bozmadan diziye gireceği indistir. [7, 12, …, 77] dizisinde 50 için ekleme noktası 6 olduğundan metot −7 döndürür. Sonucun negatif olması "bulunamadı" demektir. Bu metot da dizinin sıralı olduğunu varsayar; belgesi sırasız dizi için sonucun tanımsız olduğunu açıkça yazar.

1.3.6.5 Doğrusal ve ikili aramanın karşılaştırması

İki yöntemin farkı küçük dizilerde önemsizdir, büyük dizilerde ise çarpıcıdır (Tablo 1.3.23).

Öğe sayısı n	Doğrusal (n)	İkili ($\lfloor \log_2 n \rfloor + 1$)
9	9	4
100	100	7
1.000	1.000	10
1.000.000	1.000.000	20
2.000.000	2.000.000	21
1.000.000.000	1.000.000.000	30

Tablo 1.3.23: En kötü durumda karşılaştırma sayısı. İkili arama sütunu $\lfloor \log_2 n \rfloor + 1$ formülüyle hesaplanmış, $n \le 2.000.000$ için bütün hedefler denenerek de doğrulanmıştır.

Tablonun en öğretici satır çifti 1.000.000 ile 2.000.000'dur. Dizi iki katına çıkınca doğrusal aramanın işi de iki katına çıkar. İkili aramanın işi ise yalnız bir artar, çünkü fazladan bir tur iki katına çıkan alanı yeniden eski boyutuna indirir.

Bu üstünlüğün bir bedeli vardır: veri sıralı olmalıdır ve sıralamanın da bir maliyeti vardır. Tek bir arama için bir diziyi önce sıralayıp sonra ikili aramak, doğrudan doğrusal aramaktan daha pahalıdır. İkili arama, aynı veri üzerinde çok sayıda arama yapılacaksa, ya da veri zaten sıralı tutuluyorsa kazançlıdır. Telefon rehberi ve sözlük bu yüzden alfabetik sıradadır: bir kez sıralanırlar, milyonlarca kez aranırlar.

	Rastgele	Doğrusal	İkili
Sıralı veri gerekir mi?	Hayır	Hayır	Evet
Öğe varsa bulunur mu?	Garanti yok	Evet	Evet (sıralı veride)
Öğe yoksa durur mu?	Yalnız deneme sınırıyla	Evet, n karşılaştırmada	Evet, $\lfloor \log_2 n \rfloor + 1$ karşılaştırmada
Ortalama karşılaştırma (öğe varken)	n	$(n + 1)/2$	yaklaşık $\log_2 n$

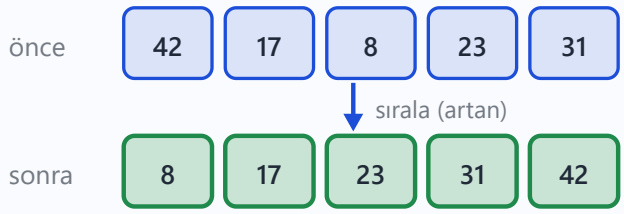
Tablo 1.3.24: Üç arama yönteminin karşılaştırması.

1.3.7 Sıralama algoritmaları

1.3.7.1 Sıralama problemi

Sıralama, bir koleksiyondaki öğeleri belirli bir kurala göre düzenler. Sayılar için kural genellikle küçükten büyüğe (artan) ya da büyükten küçüğe (azalan) sıradır; metinler için alfabetik sıradır. Sıralama, veri analizinin ve aramanın temel bir ön adımıdır. Sıralı bir dizide en küçük ve en büyük öğe

uçlardadır, ortanca değer ortadadır, tekrar eden öğeler yan yanadır. En önemlisi, ikili arama yalnız sıralı dizide çalışır.



Şekil 1.3.15: Artan sıralama: [42, 17, 8, 23, 31] dizisi [8, 17, 23, 31, 42] olur.

Bu notta üç basit yöntem incelenir. **Rastgele sıralama** (karıştırma) öğeleri rastgele bir düzene sokar ve sıralamayı garanti etmez; sıralamanın tersi bir iştir ama aynı takas işlemini kullanır. **Seçimli sıralama** en küçük öğeyi seçip başa koyar. **Kabarcık sıralaması** komşu öğeleri karşılaştırıp büyüğü sona taşır. Daha hızlı yöntemler de vardır: birleştirmeli sıralama, hızlı sıralama. Java'nın hazır `Arrays.sort` metodu bunların geliştirilmiş biçimlerini kullanır.

Üç yöntemin ortak yapı taşı **takastır** (*swap*): dizideki iki öğenin yer değiştirmesi. Java'da iki değişkenin değerini doğrudan değiştiren tek bir işlem yoktur; takas geçici bir değişkenle üç adımda yapılır.

```
int gecici = dizi[i];
dizi[i] = dizi[j];
dizi[j] = gecici;
```

İlk satır `dizi[i]`'nin değerini güvenli bir yere kopyalar. İkinci satır `dizi[i]`'nin üzerine `dizi[j]`'yi yazar; `dizi[i]`'nin eski değeri artık yalnız `gecici`'dedir. Üçüncü satır bu değeri `dizi[j]`'ye koyar.

Sık yapılan hata

Takası geçici değişkensiz yapmak. `a = b; b = a;` satırları iki değişkeni de `b`'nin değeri yapar. `a = 3, b = 5` ile başlanırsa birinci satırdan sonra `a = 5` olur ve 3 kaybolur. İkinci satır `b`'ye yine 5 koyar. JDK 21'de çalıştırılan deneme “5 5” yazar; geçici değişkenli doğru takas ise “5 3” yazar.

1.3.7.2 Rastgele sıralama (karıştırma)

Rastgele sıralama ya da **karıştırma** (*shuffle*), bir koleksiyonun öğelerini rastgele bir düzene sokar. Kart destesini karıştırmak, sınav sorularının sırasını her öğrenci için değiştirmek, bir program için rastgele test verisi üretmek tipik kullanım alanlarıdır. Bazı algoritmalar da girdiyi önce karıştırarak kötü durumlardan kaçınır. Hızlı sıralama, girdisi önceden karıştırıldığında özel olarak hazırlanmış kötü girdilerden etkilenmez.

Şekil 1.3.16 [42, 17, 8, 23, 31] dizisinin art arda üç kez karıştırılmasının ilk ve son durumunu gösterir. Birinci karıştırma [31, 17, 42, 8, 23], ikincisi [8, 23, 31, 42, 17], üçüncüsü [8, 23, 17, 31, 42] üretmiştir. Son dizi sıraya yaklaşmıştır ama hâlâ sıralı değildir. Karıştırma sıralamayı garanti etmez; amacı da bu değildir.



Şekil 1.3.16: Karıştırma. Solda başlangıç dizisi, sağda üçüncü karıştırmanın sonucu: [8, 23, 17, 31, 42]. Her karıştırma farklı bir sonuç üretir.

Doğru karıştırma. Karıştırmanın iyi yapılmış sayılması için her dizilişin eşit olasılıkla çıkması gerekir. *n* öğenin *n!* farklı dizilişi vardır; her birinin olasılığı *1/n!* olmalıdır. Bu koşulu sağlayan klasik yöntem **Fisher–Yates karıştırmasıdır**. Dizinin sonundan başlanır. Her `i` konumu için 0 ile `i` arasından, iki uç dahil, rastgele bir `j` seçilir ve `dizi[i]` ile `dizi[j]` takas edilir. Sonra bir sola geçilir.

```
static void karistir(int[] dizi) {
    Random rastgele = new Random();
    for (int i = dizi.length - 1; i > 0; i--) {
        int j = rastgele.nextInt(i + 1);   // 0 … i
        int gecici = dizi[i];
        dizi[i] = dizi[j];
        dizi[j] = gecici;
    }
}
```

Son konuma yerleşecek öğe *n* öğenin her birinden eşit olasılıkla seçilir. Sondan ikinci konuma kalan *n − 1* öğeden biri gelir ve bu böyle sürer. Toplam seçenek sayısı *n × (n − 1) × … × 2 = n!*'dir ve her seçim dizisi farklı bir dizilişe karşılık gelir. Yöntem tam *n − 1* takas yapar. Kod JDK 21'de [1, 2, 3] dizisiyle 600.000 kez çalıştırılmıştır. Altı dizilişin her biri 99.651 ile 100.509 arasında gözlenmiştir; beklenen değer 100.000'dir. Java'da aynı iş hazır olarak `Collections.shuffle(liste)` metoduyla yapılır; bu metot da Fisher–Yates yöntemini kullanır.

Sık yapılan hata

`j`'yi her adımda bütün diziden seçmek. `rastgele.nextInt(dizi.length)` ile seçim yapan bir döngü de diziyi karıştırır ama dizilişlerin olasılıkları eşit olmaz. Bu döngü n^n eşit olasılıklı seçim dizisi üretir. Bu sayı $n!$'e tam bölünmediği için bazı dizilişler ötekilerden daha sık çıkar. Üç öğe için $3^3 = 27$ seçim dizisi altı dizilişe eşit dağıtılamaz: [1, 2, 3] dizisinde [1, 3, 2], [2, 1, 3] ve [2, 3, 1] dizilişleri 5/27, ötekiler 4/27 olasılıkla çıkar. JDK 21'de 270.000 denemede ilk üç diziliş yaklaşık 50.000'er, ötekiler yaklaşık 40.000'er kez gözlenmiştir.

Karıştırarak sıralamak. "Dizi sıralanana kadar karıştır" diye bir sıralama yöntemi tasarlanabilir. Bu yönteme bogosort denir ve umutsuz derecede yavaştır. 10 öğenin 10! = 3.628.800 farklı dizilişinden yalnız biri sıralıdır. Her karıştırmanın sıralı diziyi üretme olasılığı yaklaşık üç buçuk milyonda birdir. Ortalama olarak 3,6 milyon karıştırma ve her birinden sonra bir sıralılık denetimi gerekir. 20 öğe için bu sayı $2{,}4 \times 10^{18}$'e çıkar.

1.3.7.3 Seçimli sıralama

Seçimli sıralama (*selection sort*) öğeleri karşılaştırarak sıralayan basit bir algoritmadır. Dizinin sıralanmamış kısmındaki en küçük öğe bulunur ve bu kısmın başına taşınır. Sonra sıralanmamış kısım bir öğe kısalır ve işlem tekrarlanır.

1. Koleksiyondaki en küçük öğeyi bul.
2. Bu öğeyi koleksiyonun başına taşı (ilk öğeyle yer değiştir).
3. Geri kalan kısımdaki en küçük öğeyi bul.
4. Onu ikinci konuma taşı.
5. Bu adımları koleksiyonun sonuna kadar tekrarla.

```
static void secimliSirala(int[] dizi) {
    int n = dizi.length;
    for (int i = 0; i < n - 1; i++) {
        int enKucuk = i;
        for (int j = i + 1; j < n; j++) {
            if (dizi[j] < dizi[enKucuk]) enKucuk = j;
        }
        int gecici = dizi[i];
        dizi[i] = dizi[enKucuk];
        dizi[enKucuk] = gecici;
    }
}
```

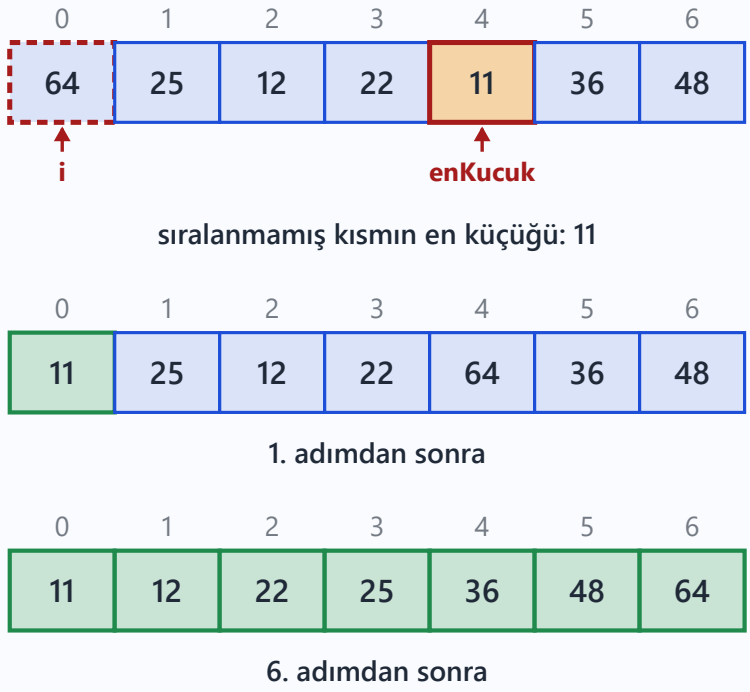
Dış döngünün `i` değişkeni, en küçük öğenin yerleşeceği konumdur. `i`'nin solundaki öğeler sıralıdır ve kesin yerlerindedir; `i` ve sağı henüz sıralanmamıştır. İç döngü `i + 1`'den dizinin sonuna kadar gezinir ve şimdiye kadar görülen en küçük öğenin indisini `enKucuk`'ta tutar. Bu, bir dizide en küçük

öğeyi bulmanın klasik kalıbıdır: ilk öğe aday kabul edilir, daha küçük bir öğe görüldükçe aday değiştirilir. İç döngü bitince `dizi[i]` ile `dizi[enKucuk]` takas edilir. Dış döngü `n - 1`'de durur, çünkü son öğe tek başına kaldığında zaten yerindedir.

Adım adım. [64, 25, 12, 22, 11, 36, 48] dizisi sıralansın. Tablo 1.3.25 her dış turu, Şekil 1.3.17 ilk adımın öncesini ve sonrasını ve son durumu gösterir.

Adım (`i`)	Sıralanmamış kısım	En küçük (`enKucuk`)	Takas	Karş.	Adımdan sonra dizi
1 (0)	indis 0–6	11 (4)	64 ↔ 11	6	[11, 25, 12, 22, 64, 36, 48]
2 (1)	indis 1–6	12 (2)	25 ↔ 12	5	[11, 12, 25, 22, 64, 36, 48]
3 (2)	indis 2–6	22 (3)	25 ↔ 22	4	[11, 12, 22, 25, 64, 36, 48]
4 (3)	indis 3–6	25 (3)	yok (zaten yerinde)	3	[11, 12, 22, 25, 64, 36, 48]
5 (4)	indis 4–6	36 (5)	64 ↔ 36	2	[11, 12, 22, 25, 36, 64, 48]
6 (5)	indis 5–6	48 (6)	64 ↔ 48	1	[11, 12, 22, 25, 36, 48, 64]

Tablo 1.3.25: Seçimli sıralama, [64, 25, 12, 22, 11, 36, 48]. "Karş." iç döngünün o turdaki karşılaştırma sayısıdır.



Şekil 1.3.17: Seçimli sıralama. (a) 1. adım: sıralanmamış kısmın en küçüğü 11 (indis 4), 0. konumdaki 64 ile yer değiştirecek. (b) 1. adımdan sonra 11 yerindedir (yeşil). (c) 6. adımdan sonra dizi sıralıdır; son öğe kendiliğinden yerine oturmuştur.

Dördüncü adımda en küçük öğe zaten sıralanmamış kısmın başındadır; `enKucuk` ile `i` eşittir. Kod bu durumda da takas satırlarını çalıştırır: öğe kendisiyle yer değiştirir ve dizi değişmez. Bu yüzden kodun yaptığı takas sayısı her zaman *n* − *1*'dir; bu örnekte 6. Gerçekten yer değiştiren takasların sayısı ise 5'tir. İstenirse takastan önce `if (enKucuk != i)` denetimi eklenebilir.

Maliyet. İç döngü birinci turda $n - 1$, ikinci turda $n - 2$, son turda 1 karşılaştırma yapar. Toplam karşılaştırma sayısı

$$(n - 1) + (n - 2) + \ldots + 2 + 1 = n(n - 1)/2$$

olur. 7 öğe için $7 \cdot 6 / 2 = 21$; tablodaki karşılaştırmaların toplamı da 6 + 5 + 4 + 3 + 2 + 1 = 21'dir. Bu sayı dizinin başlangıç düzenine bağlı değildir. Dizi zaten sıralı olsa bile seçimli sıralama 21 karşılaştırma yapar, çünkü en küçüğü bulmak için kalan her öğeye bakmak zorundadır. Algoritmanın dizinin sıralı olduğunu fark etmesinin bir yolu yoktur. Takas sayısı ise en fazla $n - 1$'dir. Seçimli sıralamanın güçlü yanı budur: yazma işleminin pahalı olduğu durumlarda, örneğin her yazmanın yıprattığı bir bellekte, az takas yapması bir üstünlüktür.

Örnek 1.3.2

Soru. [29, 10, 14, 37, 13] dizisi seçimli sıralamayla sıralanırken her adımdan sonra dizi nasıldır? Kaç karşılaştırma ve kaç gerçek takas yapılır?

Çözüm. 1. adım: en küçük 10'dur (indis 1); 29 ile takas edilir → [10, 29, 14, 37, 13]. 2. adım: indis 1–4 arasında en küçük 13'tür (indis 4); 29 ile takas edilir → [10, 13, 14, 37, 29]. 3. adım: indis 2–4 arasında en küçük 14'tür ve zaten yerindedir → [10, 13, 14, 37, 29]. 4. adım: 37 ile 29'dan küçüğü 29'dur; takas edilir → [10, 13, 14, 29, 37]. Karşılaştırma sayısı 4 + 3 + 2 + 1 = 10 = $5 \cdot 4 / 2$'dir. Gerçek takas sayısı 3'tür; kodun yaptığı takas sayısı ise kendisiyle takas dahil 4'tür.

1.3.7.4 Kabarcık sıralaması

Kabarcık sıralaması (*bubble sort*) yan yana duran öğeleri karşılaştırır. Soldaki öğe sağdakinden büyükse ikisi yer değiştirir. Dizinin başından sonuna kadar yapılan böyle bir geçişe **tur** denir. Bir tur boyunca en büyük öğe, suyun yüzeyine çıkan bir kabarcık gibi adım adım sağa taşınır ve tur sonunda dizinin son konumuna yerleşir.

1. Koleksiyonun başından başla.
2. Yan yana iki öğeyi karşılaştır; gerekiyorsa yer değiştirip büyük olanı sağa taşı.
3. Koleksiyonun sonuna kadar bu adımı tekrarla (bir tur).
4. En büyük öğe sona geldiği için son öğeyi sıralanmış kabul et.
5. İlk dört adımı kalan kısım için tekrarla; bir turda hiç yer değiştirme olmazsa dur.

```
static void kabarcikSirala(int[] dizi) {
    int n = dizi.length;
    for (int tur = 0; tur < n - 1; tur++) {
        boolean degisti = false;
        for (int j = 0; j < n - 1 - tur; j++) {
            if (dizi[j] > dizi[j + 1]) {
                int gecici = dizi[j];
                dizi[j] = dizi[j + 1];
                dizi[j + 1] = gecici;
                degisti = true;
            }
        }
        if (!degisti) break;
    }
}
```

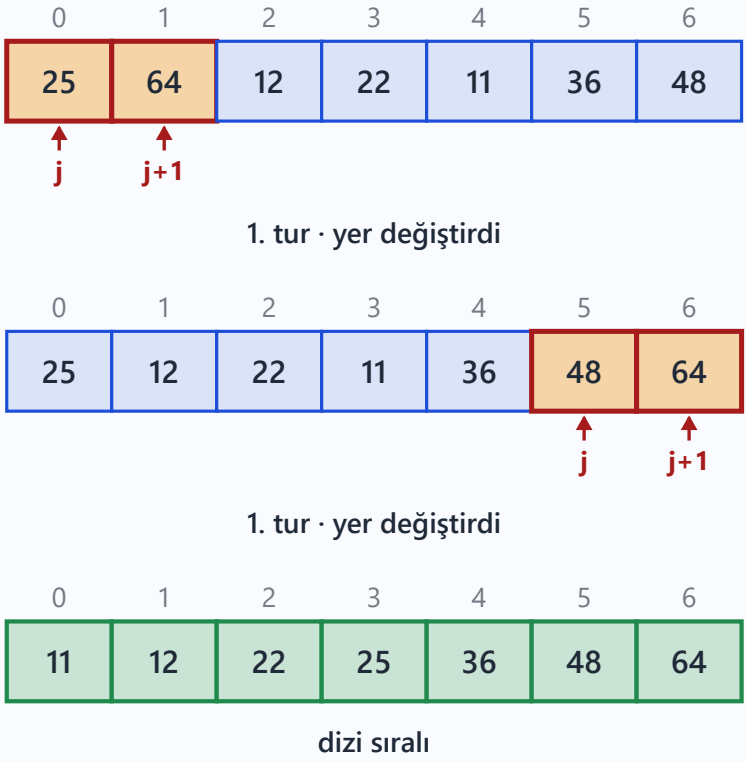
Dış döngünün `tur` değişkeni tamamlanmış tur sayısıdır. Her turdan sonra dizinin sonunda bir öğe daha kesin yerine oturur. Bu yüzden iç döngü `j < n - 1 - tur` koşuluyla her turda bir öğe daha erken durur; yerine oturmuş öğeleri yeniden karşılaştırmanın anlamı yoktur. İç döngünün sınırında `n - 1` kullanılmasının nedeni `dizi[j + 1]` erişimidir. `j` en çok `n - 2` olabilir; aksi hâlde `j + 1` dizinin dışına taşar.

`degisti` değişkeni yine bir bayraktır. Her turun başında indirilir, bir takas yapıldığında kaldırılır. Tur sonunda bayrak hâlâ inikse turda hiç takas olmamış demektir. Bu da her komşu çiftin zaten doğru sırada olduğunu, yani dizinin sıralı olduğunu gösterir. `break` kalan turları atlar. Bu **erken durma** özelliği kabarcık sıralamasını sıralı ya da neredeyse sıralı dizilerde hızlı yapar.

Adım adım. Seçimli sıralamadaki diziyle aynı dizi, [64, 25, 12, 22, 11, 36, 48], kabarcık sıralamasıyla sıralansın. Tablo 1.3.26 her turun sonucunu, Şekil 1.3.18 birinci turun ilk ve son karşılaştırmasıyla son durumu gösterir.

Tur	Adımlar	Karş.	Takas	Turdan sonra dizi	Yerine oturan
1	1–6	6	6	[25, 12, 22, 11, 36, 48, 64]	64
2	7–11	5	3	[12, 22, 11, 25, 36, 48, 64]	48
3	12–15	4	1	[12, 11, 22, 25, 36, 48, 64]	36
4	16–18	3	1	[11, 12, 22, 25, 36, 48, 64]	25
5	19–20	2	0	[11, 12, 22, 25, 36, 48, 64]	takas yok → dur
Toplam		20	11		

Tablo 1.3.26: Kabarcık sıralaması, [64, 25, 12, 22, 11, 36, 48]. Sunumdaki adım numaraları her karşılaştırmayı bir adım sayar.



Şekil 1.3.18: Kabarcık sıralaması. (a) 1. adım: 64 > 25, yer değiştirdiler. (b) 6. adım: 64 son konuma ulaştı. (c) 5. turda hiç takas olmadı; dizi sıralıdır.

Birinci turda 64 her karşılaştırmayı kazanır ve altı takasla en sona gider. İkinci turda 25 önce 12'yi ve 22'yi, sonra 11'i geçer; 36'ya takılır. Bundan sonra kabarcık 36'dır ama 36 < 48 olduğu için takas olmaz; 48 bu turun sonunda tesadüfen zaten kendi yerindedir. Üçüncü ve dördüncü turlar 11'i adım adım sola taşır: küçük öğeler kabarcık sıralamasında her turda yalnız bir konum sola ilerleyebilir. Dizinin başlarındaki büyük öğeler hızla sona gider, sonlardaki küçük öğeler ise yavaşça başa gelir. Beşinci turda hiç takas olmaz ve algoritma durur. Toplam 20 karşılaştırma ve 11 takas yapılmıştır. Değerler sayaçlı bir kopyanın JDK 21'de çalıştırılmasıyla doğrulanmıştır.

Takas sayısı ve ters çiftler. Kabarcık sıralamasındaki takas sayısının özel bir anlamı vardır. Dizideki her **ters çift**, yani solda duran büyük öğe ile sağda duran küçük öğeden oluşan çift, tam olarak bir takasla düzelir. Komşu iki öğenin takası tam olarak bir ters çifti ortadan kaldırır. Bu yüzden takas sayısı dizideki ters çift sayısına eşittir. [64, 25, 12, 22, 11, 36, 48] dizisinde 64'ün sağında ondan küçük 6 öğe vardır; 25'in sağında 3 (12, 22, 11), 12'nin sağında 1 (11), 22'nin sağında 1 (11) öğe vardır. Toplam 6 + 3 + 1 + 1 = 11 ters çift ve 11 takas.

Maliyet. En kötü durum dizinin ters sıralı olmasıdır. [64, 48, 36, 25, 22, 12, 11] dizisinde her karşılaştırma bir takasla biter ve erken durma hiç devreye girmez. 6 + 5 + 4 + 3 + 2 + 1 = 21 karşılaştırma ve 21 takas yapılır; genel olarak $n(n - 1)/2$ karşılaştırma ve aynı sayıda takas. En iyi durum dizinin zaten sıralı olmasıdır. İlk turda $n - 1$ karşılaştırma yapılır, hiç takas olmaz ve algoritma durur. 7 öğe için 6 karşılaştırma yeter. Seçimli sıralama aynı sıralı dizide yine 21 karşılaştırma yapar.

Örnek 1.3.3

Soru. [5, 1, 4, 2, 8] dizisine kabarcık sıralaması uygulanıyor. Birinci turdan sonra dizi nasıldır? Erken durmayla kaç tur, kaç karşılaştırma ve kaç takas yapılır?

Çözüm. 1. tur: 5 > 1 takas → [1, 5, 4, 2, 8]; 5 > 4 takas → [1, 4, 5, 2, 8]; 5 > 2 takas → [1, 4, 2, 5, 8]; 5 < 8 takas yok. Turdan sonra dizi [1, 4, 2, 5, 8]'dir (4 karşılaştırma, 3 takas). 2. tur: 1 < 4; 4 > 2 takas → [1, 2, 4, 5, 8]; 4 < 5 (3 karşılaştırma, 1 takas). 3. tur: 1 < 2, 2 < 4; takas yok ve algoritma durur (2 karşılaştırma). Toplam 3 tur, 9 karşılaştırma ve 4 takas. Dizideki ters çiftler (5, 1), (5, 4), (5, 2) ve (4, 2)'dir; sayıları da 4'tür.

Sık yapılan hata

İç döngüyü `j < n` ile yazmak. İç döngünün son adımında `j = n - 1` olur ve `dizi[j + 1]`, yani `dizi[n]` okunmaya çalışılır. Program `ArrayIndexOutOfBoundsException: Index 7 out of bounds for length 7` gibi bir çalışma zamanı hatasıyla durur. `dizi[j + 1]` erişimi olan her döngüde üst sınır bir eksik alınır.

1.3.7.5 Sıralama yöntemlerinin karşılaştırması

Üç yöntemin özellikleri Tablo 1.3.27'de yan yanadır. Son satır aynı yedi öğeli dizi için ölçülen değerleri verir.

	Rastgele (karıştırma)	Seçimli	Kabarcık
Fikir	Rastgele diziliş üret	En küçüğü seç, başa koy	Komşuları karşılaştır, büyüğü sona taşı
Sonuç sıralı mı?	Garanti yok	Evet	Evet
Karşılaştırma (n öğe)	—	her zaman $n(n-1)/2$	en kötü $n(n-1)/2$, sıralı dizide $n-1$
Yer değiştirme	$n-1$	en fazla $n-1$	en kötü $n(n-1)/2$ (ters çift sayısı kadar)
Örnek dizi ($n = 7$)	—	21 karşılaştırma, 5 gerçek takas	20 karşılaştırma, 11 takas

Tablo 1.3.27: Karıştırma, seçimli sıralama ve kabarcık sıralaması.

Seçimli sıralamanın ve kabarcık sıralamasının işi n^2 ile büyür. 1.000 öğe için $1000 \cdot 999 / 2 = 499.500$, yani yaklaşık 500 bin karşılaştırma yapılır. 1.000.000 öğe için bu sayı yaklaşık 500 milyara çıkar. Saniyede bir milyar karşılaştırma yapan bir bilgisayar bile bu iş için sekiz dakikadan fazla harcar. Büyük veride daha akıllı algoritmalar kullanılır. Birleştirmeli sıralama gibi yöntemlerin işi $n \cdot \log_2 n$ ile büyür; bir milyon öğe için yaklaşık 20 milyon karşılaştırma demektir. Fark yaklaşık 25.000 kattır.

Algoritma	Fikir	Ortalama	En kötü	Not
Kabarcık	Komşu iki öğe karşılaştırılır, gerekirse yer değiştirir	$O(n^2)$	$O(n^2)$	Basit; büyük verilerde etkisiz
Seçimli	En küçük bulunur, başa yerleşir; kalanla tekrarlanır	$O(n^2)$	$O(n^2)$	Büyük verilerde yavaş; az takas
Eklemeli	Sıradaki öğe sıralı kısımda uygun yere eklenir	$O(n^2)$	$O(n^2)$	Küçük ya da neredeyse sıralı veride etkili
Birleştirmeli	İkiye böl, her yarıyı sırala, birleştir	$O(n \log n)$	$O(n \log n)$	Büyük verilerde etkili; ek bellek ister
Hızlı	Bir pivot seçilir; küçükler sola, büyükler sağa	$O(n \log n)$	$O(n^2)$	Ortalamada çok hızlı

Tablo 1.3.28: Beş sıralama algoritmasının karşılaştırması. $O(\ldots)$ gösterimi işin n ile nasıl büyüdüğünü, sabit çarpanları atarak verir.

Bu tabloda iki gösterim kullanılır: $O(n^2)$ ve $O(n \log n)$. "Büyük O" gösterimi bir algoritmanın işinin girdi boyutuyla nasıl büyüdüğünü kabaca anlatır. Kesin sayılar yerine büyümenin biçimi verilir. $n(n-1)/2$ karşılaştırma yapan bir algoritma $O(n^2)$'dir. n büyüdükçe bu ifadenin değerini belirleyen terim $n^2/2$'dir; 1/2 çarpanı ise büyüme biçimini değiştirmez. Doğrusal arama $O(n)$, ikili arama $O(\log n)$'dir. Bu gösterim algoritmaların karşılaştırılmasında ortak dil olarak kullanılır.

Java'da bir diziyi sıralamak için bu algoritmaları her seferinde yeniden yazmak gerekmez. `Arrays.sort(dizi)` çağrısı `int` gibi temel tiplerden oluşan dizileri hızlı sıralamanın geliştirilmiş bir biçimiyle, nesne dizilerini ise birleştirmeli sıralamaya dayanan bir yöntemle sıralar. Basit algoritmaları öğrenmenin amacı onları üretimde kullanmak değildir. Amaç döngü, karşılaştırma, takas ve sayma alışkanlıklarını kazanmak ve bir algoritmanın maliyetini kendi başına hesaplayabilmektir.

1.3.8 Sık yapılan hatalar

Bu bölümdeki algoritmalarda en sık görülen hatalar iki gruba ayrılır. Birinci grup algoritmanın kendisindeki hatalardır: dil ne olursa olsun ortaya çıkarlar ve elle izlemeyle bulunurlar (Tablo 1.3.29). İkinci grup Java'ya özgüdür: derleyicinin ya da JVM'in bir iletisiyle, bazen de hiçbir ileti olmadan kendini gösterir (Tablo 1.3.30).

Hata	Sonuç	Doğrusu
Biriktiriciyi yanlış başlatmak: `faktoriyel = 0`	Her çarpım 0 olur; sonuç hep 0	Toplama için 0, çarpma için 1 ile başlanır
Döngü sınırında bir fazla ya da eksik: Fibonacci'de `i = 2 … n`	$n + 1$ terim yazılır	Sınırlar küçük bir testle elle izlenir
İkili aramayı sırasız dizide kullanmak	[77, 42, 64, 12, 23, 55, 7, 48, 31] içinde 77 için −1; hata iletisi yok	Önce sıralanır ya da doğrusal arama kullanılır
Sınır durumları unutmak	Asal kontrolünde 0 ve 1 asal çıkar	0, 1, boş girdi, $n = 1$ ayrıca test edilir

Hata	Sonuç	Doğrusu
Rastgele aramada durma koşulu koymamak	Hedef yoksa döngü sonsuza dek sürer	Bir deneme sınırı konur
Takası geçici değişkensiz yapmak: `a = b; b = a;`	İki değişken de `b`'nin değerini alır	Önce `gecici = a` saklanır
Fibonacci'de kaydırma sırasını ters yazmak	0 1 1 2 4 8 … (ikiye katlanan dizi)	Önce `a = b`, sonra `b = sonuc`
Üç sayı karşılaştırmalarında katı `<`	5 5 7 için 7 5 5	`<=` kullanılır

Tablo 1.3.29: Algoritma düzeyindeki sık hatalar.

Hata	Ne olur?
`import java.util.Scanner;` unutuldu	Derleme hatası: `error: cannot find symbol … symbol: class Scanner`
Satır sonunda `;` yok: `int toplam = 0`	Derleme hatası: `error: ';' expected`
Karşılaştırmada tek `=`: `if (sayi % bolen = 0)`	Derleme hatası: `error: unexpected type … required: variable, found: value`
`boolean` ile tek `=`: `if (asalMi = true)`	Derlenir; koşul her zaman doğru çıkar
Metinleri `==` ile karşılaştırmak	Derlenir ama “abcba” ile tersi için `false` verir; `equals` kullanılmalı
`i <= dizi.length` ile döngü	Çalışma hatası: `ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 5`
13! için `int` kullanmak	Hata iletisi yok, sonuç yanlış: 1932053504 (doğrusu 6227020800); `long` kullanılmalı
Türkçe ayarlı makinede “3.5” girmek	Çalışma hatası: `java.util.InputMismatchException`
Üs için `^` yazmak: `2 ^ 10`	Derlenir; sonuç 8 (bit düzeyinde özel veya)

Tablo 1.3.30: Java'ya özgü sık hatalar. İletiler JDK 21'de üretilmiştir.

Tablolardaki hataların önemli bir kısmı hiçbir ileti üretmez. Taşma, metinlerin `==` ile karşılaştırılması, sırasız dizide ikili arama ve `boolean` atama hatası programı durdurmaz, yalnız sonucu bozar. Bu hatalara karşı tek güvence, sorun tanımlanırken hazırlanmış ve sınır durumları da içeren test senaryolarıdır. Derleme hatalarında ise ileti çoğu zaman hatanın yerini doğru gösterir; iletiler ilkinden başlanarak okunur, çünkü tek bir eksik çoğu zaman ardından birkaç ileti daha getirir.

1.3.9 Özet

- **Problem çözme döngüsü:** sorunu tanımla, algoritmayı belirle, koda dönüştür, test et; hata varsa geri dön. Test senaryoları kod yazılmadan önce hazırlanır ve sınır durumları (0, boş metin, eşit sayılar, *n = 1*) içerir.
- **Algoritma:** sırası belli, açık ve sonlu adımlardan oluşan yönergeler dizisi. Beş özelliği girdi, çıktı, kesinlik, sonluluk ve etkinliktir.

- **Sözde kod ve elle izleme:** sözde kod dil kurallarından bağımsız, insan için yazılmış bir gösterimdir. İzleme tablosu her adımdaki değişken değerlerini gösterir ve döngü sınırı hatalarını kod yazmadan yakalar.
- **Java'ya dönüştürme:** `Oku` → `Scanner` ile `nextInt` / `nextDouble` / `nextLine`; `Yaz` → `print` / `println`; eşitlik `==`, atama `=`; "ve" `&&`, "veya" `||`; `Döngüyü Kır` → `break`. İki `int` arasında `/` tam sayı bölmesidir.
- **Karar algoritmaları:** iki sayıyı sıralamak tek bir `if … else` ister. Üç sayıda önce en küçük bulunur, kalan ikisi sıralanır; eşitliklerin doğru işlenmesi için `<=` gerekir; 3! = 6 sıranın hepsi test edilir.
- **Döngü algoritmaları:** faktöriyel ve üs alma 1 ile başlayan bir biriktiriciyle çalışır; Fibonacci son iki terimi tutar ve her turda kaydırır. `int` 12!'ye, F(46)'ya ve 2^{30}'a kadar yeter; ötesinde sessizce taşar. `long` 20!'ye ve F(92)'ye kadar yeter.
- **Palindrom ve asal sayı:** metin önce temizlenir, ters çevrilir, `equals` ile karşılaştırılır. Asal kontrolü bayrak ve `break` kullanır; sınır `sayı / 2`, daha iyisi `bolen * bolen <= sayi`'dir.
- **Arama:** rastgele arama ortalama n deneme yapar ve garanti vermez. Doğrusal arama en kötü n karşılaştırma yapar ve sırasız veride çalışır. İkili arama sıralı veri ister ve en kötü $\lfloor \log_2 n \rfloor + 1$ karşılaştırma yapar: 1.000.000 öğede 20.
- **Sıralama:** takas geçici değişkenle yapılır. Fisher–Yates karıştırması her dizilişi eşit olasılıkla üretir. Seçimli sıralama her zaman $n(n-1)/2$ karşılaştırma ve en çok $n-1$ takas yapar. Kabarcık sıralaması en kötü $n(n-1)/2$ karşılaştırma yapar; takas sayısı ters çift sayısına eşittir, erken durmayla sıralı dizide $n-1$ karşılaştırmada biter. İkisi de $O(n^2)$'dir.

1.3.10 Kavram listesi

algoritma *algorithm* · sözde kod *pseudocode* · izleme tablosu *trace table* · test senaryosu *test case* · sınır durum *edge case* · elle izleme *dry run* · biriktirici *accumulator* · bayrak *flag* · taşma *overflow* · derleyici *compiler* · bayt kodu *bytecode* · çalışma zamanı hatası *runtime error* · tam sayı bölmesi *integer division* · palindrom *palindrome* · asal sayı *prime number* · doğrusal arama *linear search* · ikili arama *binary search* · rastgele arama *random search* · karıştırma *shuffle* · seçimli sıralama *selection sort* · kabarcık sıralaması *bubble sort* · takas *swap* · ters çift *inversion*

1.3.11 Kendini Dene

1. Faktöriyel programı `sayi = 6` ile çalıştırılıyor. Döngü kaç tur döner ve sonuç nedir? Döngü koşu-

lu yanlışlıkla `i < sayi` yazılsaydı ne yazılırdı?

Cevap

Döngü `i` = 1 … 6 için 6 tur döner; `faktoriyel` sırasıyla 1, 2, 6, 24, 120, 720 olur ve 720 yazılır. `i < sayi` ile son çarpan 6 atlanır ve 5! = 120 yazılır.

2. Fibonacci programı *n = 10* için ne yazar ve döngü kaç tur döner? Döngü gövdesinde `a = b;` ile `b = sonuc;` satırlarının yeri değiştirilirse *n = 6* için çıktı ne olur?

Cevap

0 1 1 2 3 5 8 13 21 34; döngü `i` = 2 … 9 için $n - 2 = 8$ tur döner. Satırların yeri değişirse önce `b = sonuc` olur, ardından `a = b` ile `a` da yeni terimi alır ve iki değişken eşitlenir: 0 1 1 2 4 8.

3. Asal sayı programı 49 için kaç bölen dener? Döngü koşulu `bolen * bolen <= sayi` olarak değiştirilirse 97 için kaç bölen denenir?

Cevap

49 için sınır 49 / 2 = 24'tür; 2, 3, 4, 5, 6 bölmez, 7 böler ve döngü kırılır: 6 deneme. Yeni koşulla 97 için $b^2 \le 97$ olan bölenler 2 … 9'dur; hiçbiri bölmez ve 8 denemeden sonra 97 asal bulunur. Eski koşul 47 deneme yapıyordu.

4. Şu üç satır ne yazar: `System.out.println("toplam " + 2 + 3);`, `System.out.println(2 + 3 + " toplam");`, `System.out.println(7 / 2 * 2);`?

Cevap

“toplam 23”, “5 toplam” ve “6”. İlk satırda metin önce geldiği için iki `+` da birleştirmedir. İkincide önce 2 + 3 = 5 toplanır. Üçüncüde 7 / 2 tam sayı bölmesiyle 3 olur, 3 * 2 = 6.

5. Sıralı {2, 5, 8, 12, 16, 23, 38, 56, 72, 91} dizisinde ikili arama 23 ve 4 için hangi öğelere bakar, ne döndürür?

Cevap

23: `orta` = 4 (16) → sağ, `orta` = 7 (56) → sol, `orta` = 5 (23) → bulundu; 3 karşılaştırma, dönen değer 5. 4: 16 → sol, `orta` = 1 (5) → sol, `orta` = 0 (2) → sağ; `alt` = 1, `ust` = 0 olur ve 3 karşılaştırmadan sonra −1 döner.

6. [9, 4, 7, 1, 3] dizisi seçimli sıralamayla sıralanıyor. Her adımdan sonra dizi nasıldır? Kaç karşılaştır-

ma ve kaç gerçek takas yapılır?

Cevap

[1, 4, 7, 9, 3] → [1, 3, 7, 9, 4] → [1, 3, 4, 9, 7] → [1, 3, 4, 7, 9]. Karşılaştırma 4 + 3 + 2 + 1 = 10; her adımda en küçük öğe başka bir konumda olduğu için 4 gerçek takas.

7. Kabarcık sıralaması (erken durmalı) [2, 3, 4, 5, 1] ve [5, 1, 2, 3, 4] dizilerinde kaçar tur, karşılaştırma ve takas yapar? İki dizinin ters çift sayısı aynı olduğu hâlde tur sayıları neden farklıdır?

Cevap

[2, 3, 4, 5, 1]: 4 tur, 4 + 3 + 2 + 1 = 10 karşılaştırma, 4 takas. [5, 1, 2, 3, 4]: 2 tur, 4 + 3 = 7 karşılaştırma, 4 takas. İki dizide de 4 ters çift vardır, takas sayısı bu yüzden eşittir. Ama büyük öğe bir turda en sona kadar gidebilirken küçük öğe her turda yalnız bir konum sola ilerler. Sondaki 1'in başa gelmesi dört tur sürer.

8. Sıralı bir dizide en kötü durumda doğrusal ve ikili arama 2.000.000 öğe için kaç karşılaştırma yapar? Öğe sayısı 1.000.000 iken sayılar neydi?

Cevap

Doğrusal arama 2.000.000, ikili arama $\lfloor \log_2 2.000.000 \rfloor + 1 = 20 + 1 = 21$. 1.000.000 öğede sayılar 1.000.000 ve 20'ydi. Dizi iki katına çıkınca doğrusal aramanın işi iki katına çıkar, ikili aramanınki bir artar.

9. Üs alma programında `sonuc` değişkeni `long` yerine `int` tanımlansaydı 2^{31} ve 2^{32} için ne yazılırdı? Neden hata iletisi çıkmaz?

Cevap

−2147483648 ve 0. `int` en çok $2^{31} - 1 = 2.147.483.647$ tutar; bir fazlası başa sararak en küçük `int` değerine, bir çarpma daha 0'a dönüşür. Java tam sayı taşmasını denetlemez ve istisna fırlatmaz; sonuç sessizce yanlıştır.

1.3.12 İleri okuma

- A. B. Downey ve C. Mayfield, *Think Java* (2. baskı, 2019): koşul, döngü, metin ve dizi bölümleri bu notun Java ayrıntılarını sade bir dille tamamlar. Kitap çevrim içi ücretsiz okunabilir.
- R. Sedgewick ve K. Wayne, *Algorithms* (4. baskı, 2011), Kesim 1.1 ve 2.1: ikili aramayı ve temel sıralama algoritmalarını Java koduyla ve deneysel ölçümlerle işler.

- P. Deitel ve H. Deitel, *Java: How to Program (Early Objects)* (11. baskı, 2017), Bölüm 4 ve 19: algoritma geliştirmeyi, sözde kodu ve arama-sıralama algoritmalarını çok sayıda tam programla anlatır.
- D. E. Knuth, *The Art of Computer Programming, Vol. 2* (3. baskı, 1997), Kesim 3.4.2: Fisher–Yates karıştırmasının ve rastgele dizilişlerin klasik incelemesini içerir.

1.3.13 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
2. P. Deitel, H. Deitel, *Java: How to Program (Early Objects)*, 11. baskı, Pearson, 2017 (Bölüm 4: Algoritmalar ve sözde kod; Bölüm 19: Arama ve sıralama).
3. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019.
4. D. E. Knuth, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, 3. baskı, Addison-Wesley, 1997.
5. D. E. Knuth, *The Art of Computer Programming, Vol. 3: Sorting and Searching*, 2. baskı, Addison-Wesley, 1998.
6. Oracle, *The Java Tutorials: Language Basics*, docs.oracle.com/javase/tutorial.
7. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011 (Kesim 1.1: ikili arama; 2.1: temel sıralamalar).

KISIM I · BÖLÜM 1.4

Akış Diyagramları

Bir program yazılmadan önce çözümün kâğıt üzerinde planlanması gerekir. Akış diyagramı bu planı kutular ve oklarla çizer. Not, akış diyagramlarının sembollerini ve çizim kurallarını tanıtır; her sembolün Java'daki karşılığını gösterir. Ardından bütün algoritmaların yapı taşı olan üç akış türü gelir: sıralı, koşullu ve tekrarlı akış. Her tür bir diyagram, o diyagramdan yazılmış bir Java programı ve programın adım adım izlenmesiyle anlatılır. Döngülerde sık kullanılan sayaç, biriktirici, bayrak ve bekçi değer kalıpları, erken çıkış ve iç içe döngüler ayrıca ele alınır. Son kesimler alt programı, yani Java'daki metodu, diyagramın masa başında izlenmesini ve test durumlarının seçimini işler. Örneklerdeki bütün Java programları JDK 21 ile derlenip çalıştırılmıştır; izleme tablolarındaki değerler bu çalıştırmalarla denetlenmiştir.

ÖĞRENME ÇIKTILARI

Kavram
Algoritma ile akış diyagramı arasındaki ilişkiyi açıklamak; akış diyagramının ne işe yaradığını ve nerede yetersiz kaldığını söylemek.

Semboller
Başla/Bitir, giriş, işlem, karar, döngü, yazdır, bağlantı noktası ve alt program sembollerini tanımak; her adım için doğru sembolü seçmek; çizim kurallarını uygulamak.

Kontrol yapıları
Sıralı, koşullu ve tekrarlı akışı ayırt etmek; her biri için diyagram çizmek; yapısal diyagramı yapısal olmayandan ayırmak.

İzleme
Bir diyagramı ya da programı verilen girdiyle izleme tablosu kurarak adım adım yürütmek; sınır hatalarını ve sonsuz döngüleri fark etmek.

Diyagramdan Java'ya
Her sembolü Java karşılığına çevirmek: `nextInt()`, atama, `if` / `else`, `while`, `for`, `println`, metot çağrısı; derleyicinin yakaladığı ve yakalayamadığı hataları ayırmak.

Tasarım ve sınama
Basit bir problemi önce diyagramla planlayıp sonra programa dönüştürmek; her yolu ve sınır değerlerini deneyen test durumları seçmek.

1.4.1 Algoritma ve akış diyagramı

Bir Java programı, bilgisayarın sırayla yürüttüğü deyimlerden oluşur. Ancak program yazmaya doğrudan klavyenin başında başlamak çoğu zaman hatalı bir sonuç verir. Önce problemin nasıl çözüleceği düşünülür, çözümün adımları belirlenir, sonra bu adımlar programlama diline çevrilir. Akış diyagramı, adımları belirleme aşamasında kullanılan en eski ve en yaygın araçlardan biridir.

1.4.1.1 Algoritmadan çizime

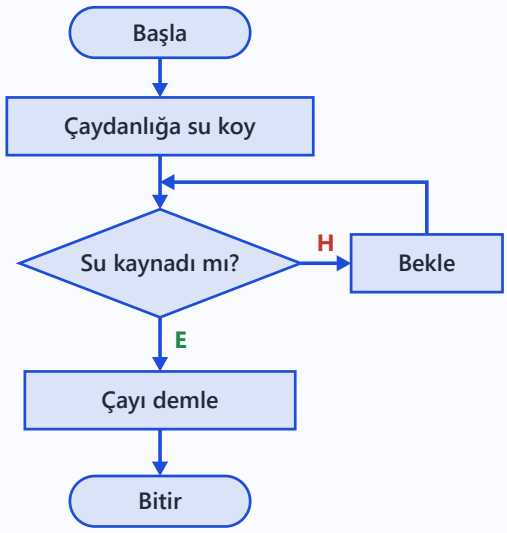
Tanım 1.4.1

Algoritma (*algorithm*), bir problemi çözmek ya da bir görevi yerine getirmek için izlenecek adımların sıralı ve sonlu bir planıdır. Her adım açıktır; aynı girdiyle her seferinde aynı sonuca ulaşılır.

Tanım 1.4.2

Akış diyagramı (*flowchart*), bir algoritmanın adımlarını, adımlar arasındaki bağlantıları ve kontrol mekanizmalarını görsel olarak gösteren çizimdir. Her adım bir sembolle çizilir. Adımların çalışma sırası oklarla, yani **akış çizgileriyle** gösterilir.

İki tanım arasındaki ilişki bir yemek tarifiyle açıklanabilir. Tarif bir algoritmadır: malzemeleri hazırla, suyu kaynat, çayı demle. Aynı tarifin kutular ve oklarla çizilmiş hâli ise akış diyagramıdır. Tarifi yazıyla anlatmak da mümkündür. Ama “su kaynamadıysa bekle, kaynadıysa demle” gibi bir karar ya da “kaynayana kadar bekle” gibi bir tekrar yazıda kolayca gözden kaçar. Çizimde ise karar bir eşkenar dörtgen, tekrar da geriye dönen bir ok olarak hemen göze çarpar. Şekil 1.4.1 bu tarifin diyagramıdır.



Şekil 1.4.1: Çay demleme algoritmasının akış diyagramı. Karar kutusunun Hayır (H) çıkışı "Bekle" kutusuna gider; oradan çıkan ok yeniden karar kutusunun üstüne döner. Su kaynayınca Evet (E) çıkışından "Çayı demle" adımına geçilir.

Diyagram yukarıdan aşağıya okunur. Yürütme yuvarlak uçlu *Başla* kutusunda başlar ve oklar boyunca ilerler. "Çaydanlığa su koy" bir işlemdir ve dikdörtgenle çizilir. "Su kaynadı mı?" bir sorudur; cevabı Evet ya da Hayır olduğu için eşkenar dörtgenle çizilir ve iki çıkışı vardır. Hayır çıkışı "Bekle" kutusuna, oradan da geriye, sorunun kendisine döner. Böylece soru, cevap Evet olana kadar tekrar tekrar sorulur. Bu küçük çizimde üç akış türünün üçü de vardır: arka arkaya gelen adımlar, bir karar ve bir tekrar.

Akış diyagramı bir programlama diline bağlı değildir. Aynı diyagramdan Java, C ya da Python programı yazılabilir. Bu yüzden diyagram, programın kendisi değil, programın planıdır. Planın doğru olduğu kâğıt üzerinde denetlenir; ancak ondan sonra kod yazılır.

1.4.1.2 Faydaları ve sınırları

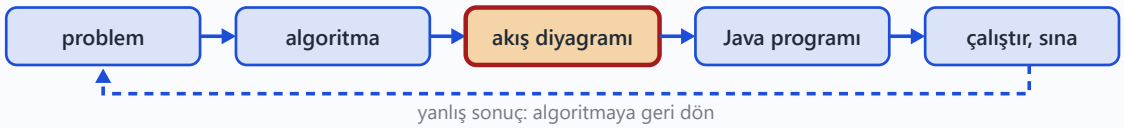
Akış diyagramının en önemli faydası, karmaşık bir süreci tek bakışta görülebilir hâle getirmesidir. Bir programda hangi adımın hangisinden sonra geldiği, hangi durumda bir adımın atlandığı ya da tekrarlandığı oklarla açıkça görünür. Bu görünürlük hataların kod yazılmadan önce bulunmasını sağlar. Örneğin bir karar kutusunun yalnızca Evet çıkışı çizilmişse, Hayır durumunda ne olacağının düşünülmediği hemen anlaşılır.

Diyagramın ikinci faydası belgelemedir. Bir süreç, örneğin bir kayıt işlemi ya da bir makinenin çalışma adımları, raporlarda ve teknik belgelerde akış diyagramıyla anlatılır. Üçüncü faydası ortak bir dil sağlamasıdır. Bir ekipte herkes aynı programlama dilini bilmeyebilir, ama akış diyagramını herkes okuyabilir.

Diyagramın sınırları da vardır. Büyük bir programın diyagramı birkaç sayfaya taşar ve okunması zorlaşır. Algoritma değiştiğinde diyagramın yeniden çizilmesi gerekir; bu da zaman alır. Ayrıntı arttıkça sembol sayısı da artar. Bu durumda algoritmayı yarı doğal dille yazan **sözde kod** (*pseudocode*) daha

pratiktir. Uygulamada akış diyagramı küçük ve orta büyüklükteki algoritmalar için, ya da büyük bir programın ana hatlarını göstermek için kullanılır.

Bir programın geliştirilmesi genellikle Şekil 1.4.2 sırasıyla ilerler. Problem tanımlanır, çözüm adımları yani algoritma belirlenir, algoritma akış diyagramına dökülür, diyagram Java programına çevrilir ve program çalıştırılıp sınanır. Sınama yanlış bir sonuç verirse geri dönülür ve algoritma düzeltilir.



Şekil 1.4.2: Tipik çalışma sırası: problem → algoritma → akış diyagramı → Java programı → çalıştırma ve sınama. Sınamada yanlış sonuç çıkarsa kesikli ok algoritmaya geri döner.

1.4.2 Semboller ve çizim kuralları

Akış diyagramındaki her sembolün biçimi, o adımın türünü söyler. Okuyucu kutunun içindeki yazıyı okumadan önce bile bir adımın okuma mı, hesaplama mı, karar mı olduğunu biçiminden anlar. Bu yüzden sembollerin doğru seçilmesi diyagramın anlaşılır olmasının ilk koşuludur.

1.4.2.1 Temel semboller

Sembollerin biçimleri uluslararası bir standarda, ISO 5807'ye dayanır. Şekil 1.4.3 en sık kullanılan sekiz sembolü gösterir.



Küçük çember. Çizgilerin birleştiği nokta.	Ok. Adımların sırası; okun yönünde ilerlenir.

Şekil 1.4.3: Temel akış diyagramı sembolleri ve görevleri.

Başlangıç/Bitiş sembolü kenarları yuvarlatılmış bir dikdörtgen, yani ovaldir. Her diyagramda bir Başla sembolü bulunur; yürütme oradan başlar. Bitir sembolü algoritmanın sona erdiği yerdir.

Giriş sembolü paralelkenardır ve dışarıdan veri alınan adımı gösterir. "Oku sayı" yazılı bir paralelkenar, kullanıcının klavyeden bir sayı gireceği ve bu sayının *sayı* adlı değişkene konacağı anlamına gelir. Bazı kaynaklar çıkışı, yani ekrana yazmayı da paralelkenarla çizer. Bu notta ekrana yazma için ayrı bir sembol, alt kenarı dalgalı **Yazdır** sembolü kullanılır. Dalgalı kenar bir kâğıt yaprağını andırır; sembolün eski adı da "belge"dir.

İşlem sembolü düz bir dikdörtgendir. İçine bir hesaplama ya da atama yazılır: "alan = a × b", "sayaç = sayaç + 1" gibi. İşlem kutusunun tek girişi ve tek çıkışı vardır.

Karar sembolü eşkenar dörtgendir. İçine cevabı Evet ya da Hayır olan bir soru yazılır: "sayı > 0 ?". Karar kutusu, iki çıkışı olan tek semboldür. Çıkışlardan biri Evet (E), öteki Hayır (H) diye etiketlenir.

Döngü sembolü uzunlamasına bir altıgendir. Belirli sayıda tekrarlanan bir işi tek kutuda özetler: "i = 1; i ≤ 10; i++". Altıgenin içindeki üç parça başlangıç, koşul ve güncellemedir. **Bağlantı noktası** küçük bir çemberdir. Birden çok okun birleştiği yeri gösterir; örneğin bir kararın iki kolu işini bitirince bağlantı noktasında buluşur. **Akış çizgisi** ise sembolleri birbirine bağlayan oktur. Yürütme her zaman okun gösterdiği yönde ilerler.

1.4.2.2 Sembollerin Java karşılıkları

Bir diyagram Java programına çevrilirken her sembol belirli bir Java yapısına karşılık gelir. Tablo 1.4.1 bu eşlemeyi verir. Tablo, diyagramdan koda geçişte bir sözlük gibi kullanılabilir.

Sembol	Diyagramdaki yazı	Java karşılığı
Başla / Bitir	Başla … Bitir	`public static void main(String[] args) { … }`
Giriş	Oku sayı	`int sayi = giris.nextInt();`
İşlem	sonuç = sayı1 + sayı2	`int sonuc = sayi1 + sayi2;`
Karar	sonuç = 0 ?	`if (sonuc == 0) { … } else { … }`
Karar + geri ok	sayaç ≤ 10 ?	`while (sayac <= 10) { … }`
Döngü	i = 1; i ≤ 10; i++	`for (int i = 1; i <= 10; i++) { … }`
Yazdır	Yaz sonuç	`System.out.println(sonuc);`

Tablo 1.4.1: Akış diyagramı sembolleri, kutuya yazılan ifade ve Java karşılığı.

Tablonun ilk satırı, diyagramın Başla ve Bitir sembolleri arasındaki her şeyin Java'da `main` metodunun gövdesine yazılacağını söyler. Program `main` metodunun ilk satırından başlar, son satırına gelince biter. Giriş sembolü için Java'da `java.util.Scanner` sınıfı kullanılır. Program başında `Scanner giris = new Scanner(System.in);` satırıyla klavyeden okuyan bir nesne oluşturulur. Ardından her "Oku" kutusu için `giris.nextInt()` (tam sayı), `giris.nextDouble()` (ondalıklı sayı) ya da `giris.next()` (sözcük) çağrılır.

Karar sembolünün iki farklı Java karşılığı vardır. Geri dönen ok yoksa karar bir `if` deyimidir: soru bir kez sorulur, cevaba göre iki koldan biri çalışır. Kararın kollarından birinden çıkan ok karar kutusunun üstüne geri dönüyorsa yapı bir döngüdür ve `while` ile yazılır. Yani aynı eşkenar dörtgen, okların nereye gittiğine göre `if` de olabilir, `while` da. Altıgen döngü sembolü ise doğrudan `for` deyimine karşılık gelir.

Çeviride en çok hata yapılan nokta eşittir işaretidir. Diyagramdaki işlem kutusunda "=" **atama** demektir: sağdaki değer hesaplanır, soldaki değişkene konur. Karar kutusunda ise "=" **karşılaştırma** demektir: iki değer eşit mi diye sorulur. Java bu iki anlamı farklı işaretlerle yazar: atama `=`, karşılaştırma `==` ile. Benzer biçimde diyagramdaki "mod" (bölümden kalan) Java'da `%` işlecidir; ≤ ve ≥ işaretleri `<=` ve `>=`, ≠ işareti `!=` olarak yazılır.

Sık yapılan hata

Karar kutusundaki "sonuç = 0 ?" sorusu Java'ya `if (sonuc = 0)` diye çevrilir. Bu satır derlenmez. JDK 21'in derleyicisi şu iletiyi verir: `error: incompatible types: int cannot be converted to boolean`. `sonuc = 0` bir atamadır ve değeri `int` türünde 0'dır; `if` ise `boolean` bir koşul bekler. Doğrusu `if (sonuc == 0)` 'dır. C dilinde aynı hata derlenir ve sessizce yanlış çalışır; Java'nın tür denetimi bu hatayı derleme sırasında yakalar.

1.4.2.3 Diğer semboller

Temel semboller çoğu diyagram için yeterlidir. Uzun diyagramları bölmek ya da verinin nereye yazıldığını göstermek için ISO 5807'de başka semboller de vardır. Şekil 1.4.4 bunlardan altısını gösterir.



Elle giriş	Açıklama	Hazırlık
Üst kenarı eğik dörtgen. Bazı kaynaklar klavyeden girişi bununla çizer.	Açık köşeli parantez. Akışa katılmayan not. Java: `//` yorumu.	Altıgen. ISO 5807'deki adı hazırlıktır; döngünün başlangıç ayarı için kullanılır.

Şekil 1.4.4: Ek akış diyagramı sembolleri.

Alt program sembolü, iki yanına dikey çizgi eklenmiş bir dikdörtgendir. Ayrı bir yerde çizilmiş adım dizisini çağırır; Java'daki karşılığı bir metot çağrısıdır. Uzun bir diyagramın bir parçası ayrı çizilir ve ana diyagramda bu sembolle çağrılır. **Sayfa dışı bağlantı**, bir diyagram tek sayfaya sığmadığında kullanılır. Birinci sayfanın sonundaki beşgene bir harf yazılır, örneğin B. İkinci sayfanın başındaki beşgende de aynı harf bulunur; okuyucu akışı bu harfle izler.

Veri deposu sembolü bir silindirdir ve kalıcı bir yere, örneğin bir dosyaya ya da veri tabanına yazılan veriyi gösterir. **Elle giriş** sembolü klavyeden yapılan girişi ayrıca vurgulamak için kullanılır; birçok kaynak bunun yerine paralelkenarı tercih eder. **Açıklama** sembolü akışın parçası değildir. Bir kutunun yanına not düşmeye yarar; Java'daki `//` yorumuna benzer. Derleyici yorumları nasıl yok sayıyorsa, diyagramı izleyen kişi de açıklamaları yürütmeye katmaz.

Hazırlık sembolünün biçimi döngü sembolüyle aynıdır. ISO 5807'de bu altıgenin adı “hazırlık”tır ve bir işe başlamadan önceki ayarı gösterir. Döngünün başlangıç değerini, koşulunu ve güncellemesini tek kutuda toplamak için bu sembolün kullanılması da buradan gelir.

1.4.2.4 Çizim kuralları

Bir akış diyagramı birkaç adımda oluşturulur. Önce işlemler belirlenir: hangi girdiler okunacak, hangi hesaplar yapılacak, hangi çıktılar yazılacak? Sonra adımlar mantıklı bir sıraya dizilir. Ardından bütün semboller akış çizgileriyle birbirine bağlanır. Son olarak koşula bağlı adımlar için karar noktaları eklenir.

Çizimin okunabilir olması için şu kurallara uyulur:

- Diyagramda tek bir Başla ve en az bir Bitir bulunur. İki ayrı Bitir sembolü kurala aykırı değildir; ama iki Başla sembolü, yürütmenin nereden başlayacağını belirsiz bırakır.
- Akış yukarıdan aşağıya ve soldan sağa okunur. Bu yönün dışına çıkan oklar, örneğin bir döngünün geri oku, okun ucuyla açıkça gösterilir.
- Karar dışındaki her sembolün tek bir çıkışı vardır. Yalnızca karar sembolünün iki çıkışı olur.
- Karar çıkışları mutlaka Evet/Hayır (E/H) diye etiketlenir. Etiketsiz bir çıkış hangi cevaba ait olduğunu söylemez.
- Çizgiler birbirini kesmez. Kesişme kaçınılmazsa çizgiler bir bağlantı noktasında birleştirilir.
- Kutulara kısa ve açık ifadeler yazılır: “Oku a, b”, “alan = a × b”, “Yaz alan”. Uzun cümleler diyagramı kalabalıklaştırır.

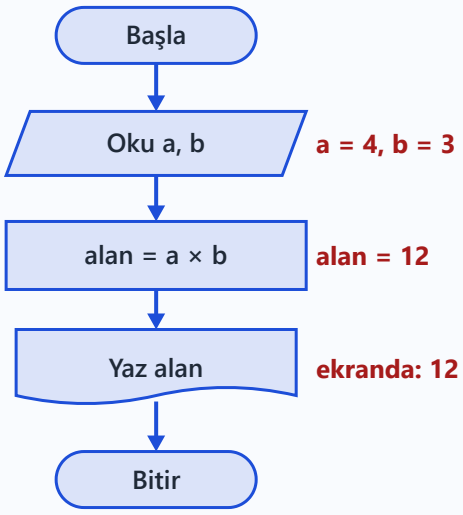
1.4.2.5 Adım adım çizim: dikdörtgenin alanı

Kurallar küçük bir problem üzerinde uygulanabilir. Problem, kenar uzunlukları klavyeden girilen bir dikdörtgenin alanını bulmaktır.

Örnek 1.4.1

Problem: klavyeden okunan iki kenar uzunluğuna göre dikdörtgenin alanını ekrana yazmak.

1. **İskelet.** Önce bir Başla ve bir Bitir sembolü çizilir. Bütün adımlar bu ikisinin arasına girecektir.
2. **Girdiler.** Alanı hesaplamak için iki kenar uzunluğu gerekir. Bunlar klavyeden okunur: "Oku a, b" yazılı bir paralelkenar eklenir.
3. **İşlem.** Alan, iki kenarın çarpımıdır. "alan = a × b" yazılı bir dikdörtgen eklenir. Bu kutu çarpımı hesaplar ve sonucu *alan* değişkenine atar.
4. **Çıktı.** Hesaplanan alan ekrana yazılır: "Yaz alan" yazılı bir Yazdır sembolü eklenir.
5. **Bağlama.** Semboller mantıklı sırayla akış çizgileriyle bağlanır: Başla → Oku → İşlem → Yaz → Bitir. Bu problemde karar noktası yoktur; akış sıralıdır.
6. **Sınama.** Diyagram bir girdiyle denenir. a = 4 ve b = 3 girilirse alan = 4 × 3 = 12 hesaplanır ve ekranda 12 görülür. Beklenen sonuç da 12'dir; diyagram doğru çalışır.



Şekil 1.4.5: Dikdörtgenin alanını bulan diyagram. Sağdaki kırmızı notlar a = 4, b = 3 girdisiyle yapılan sınamada değişkenlerin aldığı değerleri gösterir.

Diyagram hazır olduğunda Java'ya çevirmek mekanik bir iştir. Her sembol Tablo 1.4.1 yardımıyla bir satıra dönüşür:

```java
import java.util.Scanner;

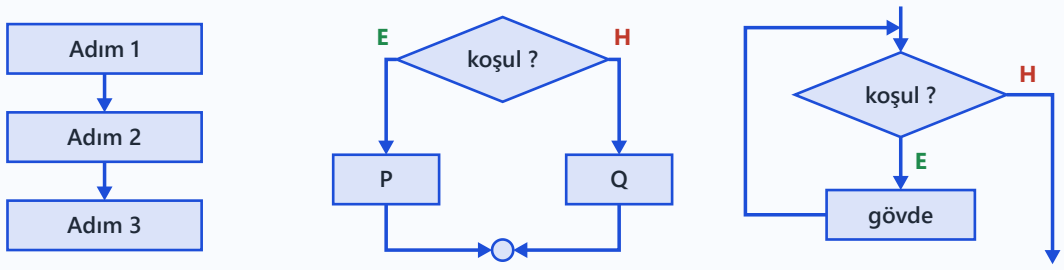
public class DikdortgenAlani {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int a = giris.nextInt();
        int b = giris.nextInt();
        int alan = a * b;
        System.out.println(alan);
        giris.close();
    }
}
```

Paralelkenardaki “Oku a, b” iki ayrı `nextInt()` çağrısı olur. İşlem kutusu `int alan = a * b;` satırına dönüşür; diyagramdaki × işareti Java'da `*` ile yazılır. Yazdır sembolü `System.out.println(alan);` satırıdır. Program 4 ve 3 girdisiyle çalıştırıldığında ekrana 12 yazar; bu, diyagramın sınamasındaki sonuçla aynıdır. Her kutu yalnızca bir kez çalışır, hiçbir adım atlanmaz ve hiçbir adıma geri dönülmez. Bu özellikler sıralı akışın tanımını oluşturur.

1.4.3 Kontrol yapıları

Bir programın hangi deyimi hangi sırayla çalıştıracağına **kontrol akışı** denir. Akış diyagramında kontrol akışı okların kendisidir. Oklar ne kadar karışık görünürse görünsün, iyi çizilmiş her diyagram yalnızca üç temel yapıdan kurulur.

1.4.3.1 Üç akış türü



Şekil 1.4.6: Üç kontrol yapısı. (a) Sıralı akış: adımlar sırayla, birer kez çalışır. (b) Koşullu akış: koşula göre P ya da Q çalışır, yollar bağlantı noktasında birleşir. (c) Tekrarlı akış: koşul sağlandığı sürece gövde tekrarlanır; koşul yanlış olunca H çıkışından devam edilir.

Tanım 1.4.3

Sıralı akış (*sequential flow*), adımların yazıldıkları sırayla ve her birinin tam bir kez çalıştığı akıştır. **Koşullu akış** (*conditional flow*), bir koşulun sonucuna göre iki ya da daha çok yoldan birinin seçildiği akıştır. **Tekrarlı akış** ya da **döngü** (*loop*), bir koşul sağlandığı sürece bazı adımların tekrar tekrar çalıştığı akıştır.

Şekil 1.4.6 üç yapıyı yan yana gösterir. Sıralı akışta oklar hep aşağı iner, hiçbir ok ayrılmaz ya da geri dönmez. Koşullu akışta bir karar kutusundan iki ok çıkar; bu iki yol daha sonra bir bağlantı noktasında yeniden birleşir. Tekrarlı akışta ise bir ok geriye, karar kutusunun üstüne döner. Geri dönen ok bir döngünün kesin işaretidir.

Bu üç yapının yeterli olduğu matematiksel olarak gösterilmiştir. Corrado Böhm ve Giuseppe Jacopini 1966'da yayımladıkları makalede şu sonucu kanıtladılar:

Teorem 1.4.1 (Yapısal programlama teoremi, Böhm–Jacopini)

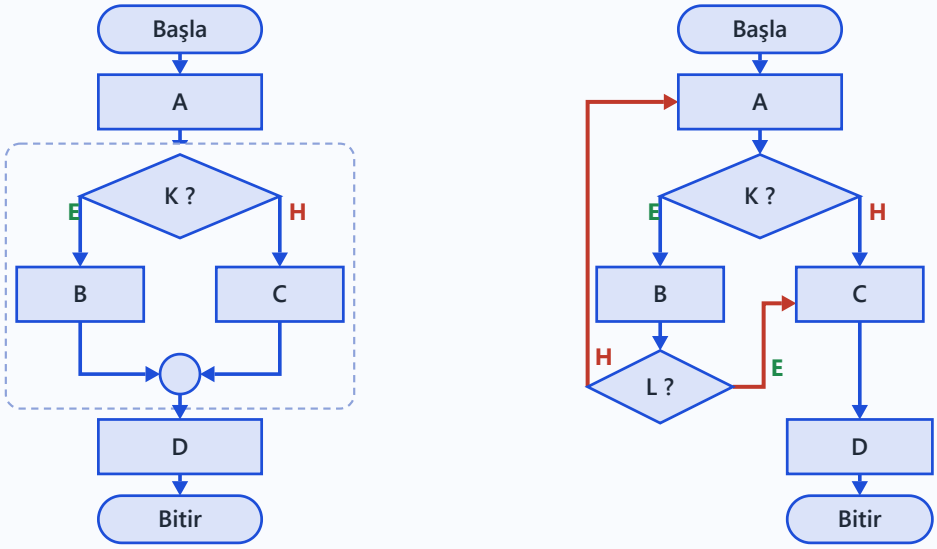
Hesaplanabilen her algoritma yalnızca üç yapı kullanılarak yazılabilir: sıra, seçim ve tekrar. Bu yapılar ya art arda dizilir ya da birbirinin içine yerleştirilir.

Teoremin Java'daki karşılığı açıktır. Sıra, art arda yazılmış deyimlerdir. Seçim `if` / `else`, tekrar `while` ve `for` deyimleridir. Java'nın öteki kontrol deyimleri, örneğin `switch` ya da `do-while`, bu üç yapının kısa yazılışlarıdır. Teorem, karmaşık bir problemin bile bu üç parçayla çözülebileceğini garanti eder. Bir diyagram tasarlanırken de aynı fikir izlenir: problem sıra, seçim ve tekrar parçalarına bölünür ve parçalar birleştirilir.

1.4.3.2 Yapısal ve yapısal olmayan diyagram

Tanım 1.4.4

Yapısal diyagram (*structured flowchart*), her kontrol yapısının tek girişi ve tek çıkışı olan akış diyagramıdır. Yapılar ya art arda dizilir ya da birbirinin içine yerleştirilir; bir yapının ortasına dışarıdan ok girmez.



Şekil 1.4.7: (a) Yapısal diyagram: K kararı ile B ve C kolları kesikli çerçeve içinde tek girişli, tek çıkışlı bir blok oluşturur. (b) Yapısal olmayan diyagram: L kararının Evet oku öteki kolun ortasına (C) girer, Hayır oku geriye, A'ya döner.

Şekil 1.4.7'nin solundaki diyagramda K kararıyla başlayan blok kesikli bir çerçeveyle gösterilmiştir. Bloğa üstten tek bir ok girer, alttan tek bir ok çıkar. B ve C kolları blok içinde kalır ve bağlantı noktasında birleşir. Bu bloğun içinde ne olduğunu bilmeden onu tek bir kutu gibi düşünmek mümkündür. Diyagramı okuyan kişi önce A'yı, sonra bloğu, sonra D'yi görür.

Sağdaki diyagramda ise L kararının Evet oku öteki kolun ortasına, C kutusuna girer. Hayır oku ise geriye, A'ya döner. Bu durumda C kutusuna iki farklı yoldan gelinebilir; A'ya da hem Başla'dan hem de L'den ulaşılır. Bir kutuya hangi yoldan gelindiğini anlamak için bütün diyagramı izlemek gerekir. Oklar spagetti gibi birbirine dolanır; bu yüzden böyle akışa “spagetti akış” denir. Spagetti akışı izlemek ve içindeki hatayı bulmak zordur.

Eski programlama dillerinde bu tür atlamalar `goto` deyimiyle yazılırdı. `goto etiket;` deyimi yürütmeyi programın herhangi bir yerine götürürdü. Java'da `goto` ayrılmış bir sözcüktür ama hiçbir anlamı yoktur; kullanılamaz. `goto basa;` yazılan bir program JDK 21'de `error: illegal start of expression` iletisiyle derlenmez. Java'nın tasarımcıları bu sözcüğü bilinçli olarak dilin dışında tutmuştur.

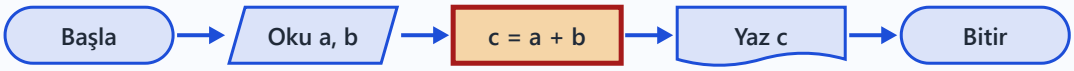
Java'da sınırlı iki atlama vardır: `break` ve `continue`. `break` içinde bulunduğu döngüyü hemen bitirir. `continue` döngünün o turunun geri kalanını atlar ve sonraki tura geçer. Bu iki deyim yürütmeyi programın rastgele bir yerine değil, yalnızca bulundukları döngünün sonuna ya da başına götürür. Böylece spagetti akış oluşmaz.

Böhm–Jacopini teoreminin pratik bir sonucu da şudur: yapısal olmayan her diyagram, gerekirse `boolean` bir **bayrak** değişkeni eklenerek yapısal biçime dönüştürülebilir. Bayrak, bir olayın olup olmadığını tutar; atlamanın yerine bu değişken sınanır.

1.4.4 Sıralı akış

Sıralı akış üç yapının en basitidir. Yine de her program sıralı parçalar içerir: değişkenlere başlangıç değeri vermek, girdi okumak, bir formülü hesaplamak, sonucu yazmak. Koşullu ve tekrarlı yapılar bile kendi içlerinde sıralı adımlardan oluşur.

1.4.4.1 Sıralı akışın özellikleri



Şekil 1.4.8: Sıralı akış: Başla → Oku a, b → c = a + b → Yaz c → Bitir. Akış dallanmadan ve geri dönmeden ilerler.

Sıralı akış, yapılacak işlerin art arda ve mantıklı bir sırayla dizilmesidir. Şekil 1.4.8 böyle bir akışı soldan sağa çizer; aynı akış yukarıdan aşağıya da çizilebilir. Sıralı akışın dört özelliği vardır.

Basittir. Dallanma ve tekrar yoktur. Akışı anlamak için okları sırayla izlemek yeterlidir.

Bakımı kolaydır. Adımlar yazıldıkları sırayla çalıştığı için bir hata ararken programın hangi noktada olduğunu bulmak kolaydır. Hata ayıklama, adımların birer birer kontrol edilmesinden ibarettir.

Sıra önemlidir. Bir adım öncekinin sonucunu kullanıyorsa yer değiştiremez. “c = a + b” adımı a ve b okunmadan önce yapılamaz. Ama birbirinden bağımsız adımların sırası değiştirilebilir. “Oku a” ile “Oku b” adımlarının yeri değişirse sonuç değişmez; yalnızca kullanıcıdan istenen sıra değişir.

Geri dönüş yoktur. Her adım tam olarak bir kez çalışır. Hiçbir adım atlanmaz ve önceki bir adıma geri dönülmez. N adımlı bir sıralı akış, girdiden bağımsız olarak her zaman N adımda biter.

Sık yapılan hata

Toplama adımı sayılar okunmadan önce yazılırsa toplama henüz bilinmeyen değerlerle yapılmaya çalışılır. Diyagramda bu bir mantık hatasıdır. Java'da ise değişkenler tanımlanmış ama henüz değer almamışsa derleyici hatayı yakalar:

```
int sayi1, sayi2;
int sonuc = sayi1 + sayi2;     // sayılar henüz okunmadı
sayi1 = giris.nextInt();
sayi2 = giris.nextInt();
```

JDK 21 bu parça için iki hata verir: `error: variable sayi1 might not have been initialized` ve aynısı `sayi2` için. Java, değer almamış bir yerel değişkenin okunmasına hiçbir koşulda izin vermez. Bu kurala *kesin atama* denir. Derleyici her değişken için programın

bütün yollarını inceler; değişkenin kullanıldığı noktaya değer almadan ulaşan bir yol varsa programı derlemez.

1.4.4.2 İki sayının toplamı

Sıralı akışın klasik örneği, klavyeden girilen iki sayının toplamını yazdırmaktır. Problemin algoritması altı adımdır:

1. Başla.
2. İlk sayıyı oku (sayı1).
3. İkinci sayıyı oku (sayı2).
4. İki sayıyı topla, sonucu sonuç değişkenine ata.
5. Sonucu ekrana yaz.
6. Bitir.



Şekil 1.4.9: İki sayının toplamını yazan akış diyagramı. Algoritmanın her adımı bir sembole karşılık gelir.

Şekil 1.4.9 bu algoritmanın diyagramıdır. Algoritmanın her adımı bir sembole dönüşmüştür: Başla ve Bitir ovaldir, iki okuma adımı paralelkenardır, toplama işlemi dikdörtgendir, ekrana yazma Yazdır sembolüdür. Algoritmada altı adım, diyagramda altı sembol vardır. Bu birebir eşleme sıralı akışın tipik bir özelliğidir.

Okuma adımları iki ayrı paralelkenar olarak çizilmiştir. Tek bir “Oku sayı1, sayı2” kutusu da doğru olurdu. İki ayrı kutu, Java programında da iki ayrı okuma ve iki ayrı soru mesajı olacağını açıkça gösterir.

1.4.4.3 Diyagramdan Java'ya: toplam programı

Diyagram aşağıdaki Java programına çevrilir:

```
import java.util.Scanner;

public class IkiSayiToplami {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        System.out.print("İlk sayı: ");
        int sayi1 = giris.nextInt();
        System.out.print("İkinci sayı: ");
        int sayi2 = giris.nextInt();
        int sonuc = sayi1 + sayi2;
        System.out.println("Sonuç: " + sonuc);
        giris.close();
    }
}
```

Program satır satır şöyle çalışır. 1. satırdaki `import` deyimi, `Scanner` sınıfını `java.util` paketinden programa getirir. Bu satır olmadan derleyici `Scanner` adını tanımaz. 3. satır `IkiSayiToplami` adlı sınıfı tanımlar; Java'da her program en az bir sınıfın içinde yazılır ve dosyanın adı `IkiSayiToplami.java` olmalıdır.

4. satırdaki `main` metodu diyagramdaki Başla sembolüdür; program buradan başlar. 5. satırda klavyeden okuyacak `Scanner` nesnesi oluşturulur. `System.in` standart girdi, yani klavyedir.

6. ve 7. satırlar birinci paralelkenardır. `print` önce kullanıcıya ne girmesi gerektiğini söyler. `println`'dan farklı olarak satır sonu eklemez; böylece kullanıcının yazdığı sayı mesajla aynı satırda görünür. `giris.nextInt()` kullanıcının bir tam sayı yazıp Enter'a basmasını bekler. Okunan değer `sayi1` değişkenine atanır. 8. ve 9. satırlar aynı işi ikinci sayı için yapar.

10. satır işlem kutusudur: `int sonuc = sayi1 + sayi2;`. 11. satır Yazdır sembolüdür. `"Sonuç: " + sonuc` ifadesinde `+` işleci bir dizgiyle bir sayıyı birleştirir; sayı dizgiye çevrilir ve sonuna eklenir. 12. satır `Scanner` nesnesini kapatır. 13. satırda `main` biter; bu da diyagramdaki Bitir sembolüdür.

Tablo 1.4.2 programın kullanıcı önce 7, sonra 5 girdiğindeki izlemesidir. Tabloda her satır bir diyagram kutusuna ve o kutuyu oluşturan kod satırlarına karşılık gelir. “—” işareti değişkenin henüz değer almadığını gösterir.

Satır	Diyagram kutusu	`sayi1`	`sayi2`	`sonuc`	Ekrana eklenen
4–5	Başla; `Scanner` oluşturulur	—	—	—	
6–7	Oku sayı1	7	—	—	İlk sayı: 7
8–9	Oku sayı2	7	5	—	İkinci sayı: 5
10	sonuç = sayı1 + sayı2	7	5	12	
11	Yaz sonuç	7	5	12	Sonuç: 12

Satır	Diyagram kutusu	`sayi1`	`sayi2`	`sonuc`	Ekrana eklenen
12–13	Bitir	7	5	12	

Tablo 1.4.2: `IkiSayiToplami` programının 7 ve 5 girdisiyle izlenmesi.

Ekranda görülen konuşma şöyledir; kullanıcının yazdığı sayılar mesajların hemen arkasındadır:

```
İlk sayı: 7
İkinci sayı: 5
Sonuç: 12
```

İzleme tablosu sıralı akışın özelliklerini sayılarla gösterir. Tabloda altı satır vardır ve her diyagram kutusu tam bir kez görünür. Hiçbir satır atlanmamış, hiçbir satıra geri dönülmemiştir. `sonuc` değişkeni ancak 10. satırda değer alır; ondan önceki satırlarda "—" görünür. Girdi değişse de tablonun satır sayısı değişmez; değişen yalnızca hücrelerdeki değerlerdir. Örneğin −3 ve 10 girilirse program `Sonuç: 7` yazar.

Sık yapılan hata

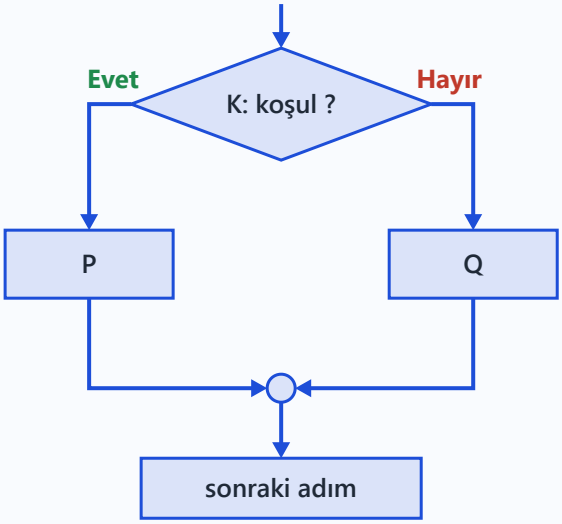
`nextInt()` yalnızca tam sayı okur. Kullanıcı "abc" ya da "7,5" yazarsa program bir hata iletisiyle durur: `Exception in thread "main" java.util.InputMismatchException`. Diyagramdaki "Oku sayı" kutusu kullanıcının doğru türde veri gireceğini varsayar. Bu varsayım programda da geçerlidir; geçersiz girdiyi yakalamak için ayrıca bir karar ya da istisna yakalama gerekir. Ondalıklı sayılar için `nextDouble()` kullanılır; bu metot ondalık ayırıcı olarak bilgisayarın bölge ayarındaki işareti bekler.

1.4.5 Koşullu akış

Sıralı bir program her çalıştırmada aynı adımları izler. Oysa çoğu problemde yapılacak iş duruma göre değişir. Not 50'nin altındaysa öğrenci kalır, değilse geçer. Sayı çiftse bir mesaj, tekse başka bir mesaj yazılır. Koşullu akış programa bu esnekliği verir.

1.4.5.1 Karar yapısı

Koşullu akışta iki ya da daha çok ihtimalden biri seçilir. Yapının üç parçası vardır. **Karar noktası** (K) bir koşul sorusu içerir; sorunun cevabı Evet ya da Hayır'dır. Cevap Evet ise akış **P** işlemi yönünde ilerler. Cevap Hayır ise akış **Q** işlemi yönünde ilerler. İki yol işini bitirince bir bağlantı noktasında birleşir ve akış sonraki adımla sürer.



Şekil 1.4.10: Koşullu akışın genel biçimi. K koşulu doğruysa Evet kolundaki P, yanlışsa Hayır kolundaki Q çalışır. İki kol bağlantı noktasında birleşir.

Bu yapının Java karşılığı `if` / `else` deyimidir:

```
if (K) {
    P;   // koşul doğru
} else {
    Q;   // koşul yanlış
}
```

Parantez içindeki K, değeri `true` ya da `false` olan bir `boolean` ifadedir. Java'da koşul mutlaka `boolean` olmalıdır; C dilindeki gibi sıfır ya da sıfırdan farklı bir tam sayı koşul yerine geçmez. Süslü parantezler bir kolda çalışacak deyimleri bir **blok** içinde toplar. Bir kolda tek deyim varsa süslü parantezler yazılmayabilir. Ama o kola sonradan ikinci bir deyim eklenince hata kolayca yapılır; bu yüzden parantezleri her zaman yazmak iyi bir alışkanlıktır.

Koşullu akışın programa kazandırdığı üç şey vardır. **Esneklik:** program farklı girdilere farklı davranır. **Karar verme:** bir işlem yalnızca gerektiğinde yapılır; örneğin sıfıra bölme ancak bölen sıfır değilse denenir. **Kod kontrolü:** program bir durumu denetler ve ona göre davranır; örneğin geçersiz bir girdiyi fark edip uyarı yazar.

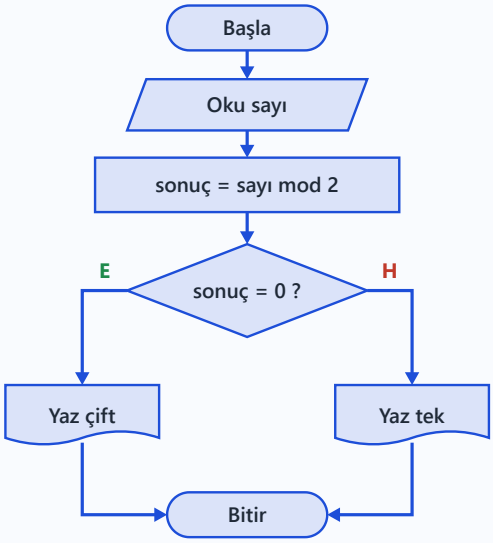
Karar yapısının bir de tek kollu biçimi vardır. Hayır kolunda yapılacak bir iş yoksa o kol doğrudan bağlantı noktasına gider. Bu **tek yönlü karardır** ve Java'da `else`'siz bir `if` ile yazılır: `if (x > 0) System.out.println(x);`. Koşul yanlışsa hiçbir şey yapılmaz, akış bir sonraki deyimle sürer.

1.4.5.2 Sayı tek mi çift mi?

Problem, girilen bir tam sayının tek mi çift mi olduğunu ekrana yazmaktır. Bir sayının çift olması, 2'ye bölümünden kalanın 0 olması demektir. Bölümden kalan **mod** işlemiyle bulunur: 7 mod 2 = 1, çünkü 7 = 3 × 2 + 1'dir. 8 mod 2 = 0'dır, çünkü 8 = 4 × 2 + 0'dır. Algoritma yedi adımdır:

1. Başla.
2. Sayıyı oku.
3. Sayı mod 2'yi al, sonuç değişkenine ata.
4. Sonuç 0 ise 5'e git, değilse 6'ya git.
5. Ekrana “çift” yaz, 7'ye git.
6. Ekrana “tek” yaz.
7. Bitir.

Algoritmada “... 'e git” ifadeleri dikkat çeker. Yazıyla anlatılan bir algoritmada dallanma ancak böyle adım numaralarıyla gösterilebilir. Diyagramda ise aynı dallanma bir karar kutusu ve iki okla çizilir; adım numaralarına gerek kalmaz.



Şekil 1.4.11: Tek/çift diyagramı. İşlem kutusu kalanı hesaplar, karar kutusu kalanın 0 olup olmadığını sorar. E kolu “çift”, H kolu “tek” yazar; iki kol Bitir'de birleşir.

Diyagramın Java karşılığı şöyledir:

```
import java.util.Scanner;

public class TekMiCiftMi {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        System.out.print("Sayı: ");
        int sayi = giris.nextInt();
        int sonuc = sayi % 2;
        if (sonuc == 0) {
            System.out.println("çift");
        } else {
            System.out.println("tek");
        }
        giris.close();
    }
}
```

Diyagramdaki “mod” Java'da `%` işlecidir. 8. satır `sonuc = sayi % 2` hesabını yapar. 9. satırdaki `if` karar kutusudur ve `sonuc == 0` sorusunu sorar. Koşul doğruysa 10. satır, yanlışsa 12. satır çalışır; ikisi birden asla çalışmaz. 11. satırdaki `else` Hayır kolunun başlangıcıdır.

Tablo 1.4.3 programın 7 ve 8 girdileriyle iki ayrı çalıştırmasını özetler. 7 girildiğinde `sonuc = 7 % 2 = 1` olur. `1 == 0` yanlıştır, akış Hayır koluna, yani `else` bloğuna gider ve “tek” yazılır. Bu çalıştırmada `if` bloğundaki 10. satır hiç çalışmaz. 8 girildiğinde `sonuc = 0` olur, Evet kolundan “çift” yazılır ve bu kez 12. satır atlanır. Her çalıştırmada iki koldan yalnızca biri çalışır; iki yol Bitir'de birleşir.

Girdi	İşlem kutusu	Karar	Çıkış	Çalışan satır	Çıktı
7	7 % 2 = 1	1 == 0 → yanlış	Hayır	12	tek
8	8 % 2 = 0	0 == 0 → doğru	Evet	10	çift

Tablo 1.4.3: `TekMiCiftMi` programının iki çalıştırması.

Negatif sayılar ayrı bir dikkat ister. Java'da `%` işleminin sonucu bölünenin işaretini alır. Bu yüzden `-3 % 2` değeri 1 değil, −1'dir; `-4 % 2` ise 0'dır. Sunumdaki diyagram koşulu “sonuç = 0 ?” diye kurduğu için bu ayrıntıdan etkilenmez: −1 ≠ 0 olduğundan −3 için doğru olarak “tek” yazılır. Koşul “sonuç = 1 ?” diye kurulsaydı durum değişirdi. Tablo 1.4.4 iki koşulu −3, 7 ve 0 girdileriyle karşılaştırır.

Girdi	`sayi % 2`	Koşul `== 0` (sunumdaki)	Koşul `== 1`
−3	−1	Hayır → tek	Hayır → çift
7	1	Hayır → tek	Evet → tek
0	0	Evet → çift	Hayır → çift

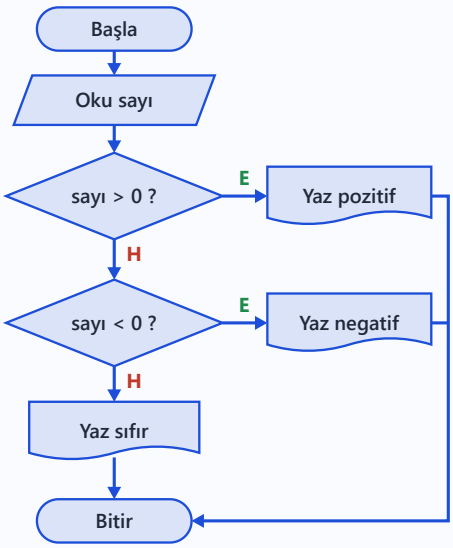
Tablo 1.4.4: Karar kutusundaki koşulun negatif girdide sonuca etkisi (Java'da kalan bölünenin işaretini alır).

Sık yapılan hata

Tek sayı denetimini `if (sayi % 2 == 1)` diye yazmak negatif tek sayılarda yanlış sonuç verir. −3 için `-3 % 2` değeri −1 olduğundan koşul yanlış olur ve program `-3 çift` yazar. Derleyici bu hatayı yakalamaz, çünkü kod Java kurallarına uygundur; hata mantıktadır. Güvenli yol, çiftliği sınamaktır: `sayi % 2 == 0`. Kalan −1 de olsa 1 de olsa sayı "tek" koluna düşer. Matematikteki gibi her zaman negatif olmayan bir kalan gerektiğinde `Math.floorMod(sayi, 2)` kullanılır; `Math.floorMod(-3, 2)` değeri 1'dir.

1.4.5.3 İç içe karar: pozitif, negatif, sıfır

Bir karar kutusunun yalnızca iki çıkışı vardır. Üç ihtimal olduğunda iki karar kutusu art arda bağlanır: ilk sorunun Hayır çıkışı ikinci soruya gider. Bir sayının pozitif, negatif ya da sıfır olduğunu bulan Şekil 1.4.12 bu yapıyı gösterir.



Şekil 1.4.12: İç içe karar. Birinci kararın (sayı > 0 ?) Hayır çıkışı ikinci karara (sayı < 0 ?) gider. Üç Yaz kutusundan her çalıştırmada yalnızca biri çalışır.

Birinci karar "sayı > 0 ?" sorusunu sorar. Cevap Evet ise "pozitif" yazılır ve akış Bitir'e gider; ikinci soru hiç sorulmaz. Cevap Hayır ise sayı ya negatiftir ya da sıfırdır. Bunu ayırmak için ikinci karar "sayı < 0 ?" sorusunu sorar. Evet ise "negatif", Hayır ise "sıfır" yazılır. İkinci kararın Hayır çıkışına ulaşan bir sayı ne pozitiftir ne negatif; geriye tek ihtimal kalır, o da sıfırdır. Bu yüzden üçüncü bir soruya gerek yoktur.

Java'da iç içe karar `else if` zinciriyle yazılır:

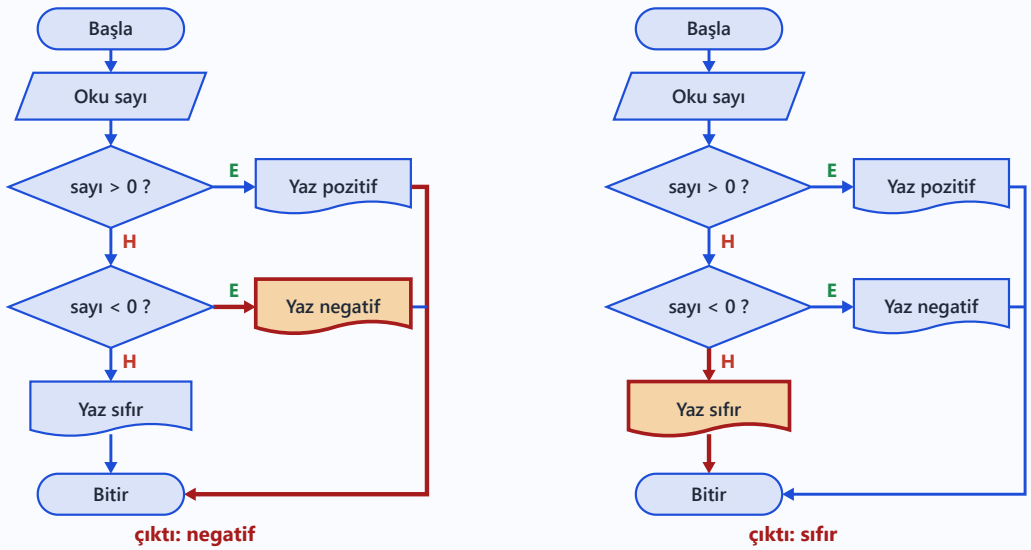
```
Scanner giris = new Scanner(System.in);
int sayi = giris.nextInt();
if (sayi > 0) {
    System.out.println("pozitif");
} else if (sayi < 0) {
    System.out.println("negatif");
} else {
    System.out.println("sıfır");
}
```

`else if`, Java'da ayrı bir deyim değildir. Birinci `if`'in `else` kolunda ikinci bir `if` deyimi bulunur; süslü parantezler bu yapıyı tek düzeyde yazmaya izin verir. Diyagramda ikinci kararın birinci kararın Hayır çıkışına bağlı olması da tam olarak bu iç içeliği gösterir.

Tablo 1.4.5 programın üç çalıştırmasını izler. −4 girildiğinde birinci koşul `-4 > 0` yanlıştır; akış Hayır çıkışından ikinci karara gider. `-4 < 0` doğrudur ve "negatif" yazılır; son `else` kolu atlanır. 0 girildiğinde `0 > 0` yanlıştır, çünkü 0 pozitif değildir. `0 < 0` da yanlıştır; iki koşul da yanlış olduğu için son `else` çalışır ve "sıfır" yazılır. 5 girildiğinde ise birinci koşul doğrudur ve ikinci koşul hiç sınanmaz.

Girdi	1. karar	2. karar	Çıktı
−4	−4 > 0 → yanlış	−4 < 0 → doğru	negatif
0	0 > 0 → yanlış	0 < 0 → yanlış	sıfır
5	5 > 0 → doğru	sorulmaz	pozitif

Tablo 1.4.5: İç içe karar programının üç çalıştırması. Sınanan karar sayısı girdiye göre 1 ya da 2'dir.



Şekil 1.4.13: İç içe kararın iki çalıştırması. (a) Girdi −4: birinci kararın Hayır, ikinci kararın Evet çıkışı izlenir ve "negatif" yazılır. (b) Girdi 0: iki kararın da Hayır çıkışı izlenir ve "sıfır" yazılır. Vurgulu oklar izlenen yolu gösterir.

Üç kol olduğu için diyagramı tam olarak denemek en az üç girdi gerektirir. Her girdi farklı bir Yaz kutusunu çalıştırır. Bu, test durumlarının seçiminde kullanılan *yol kapsama* fikrinin ilk örneğidir: her karar kutusunun her çıkışı en az bir kez denenmelidir.

Örnek 1.4.2

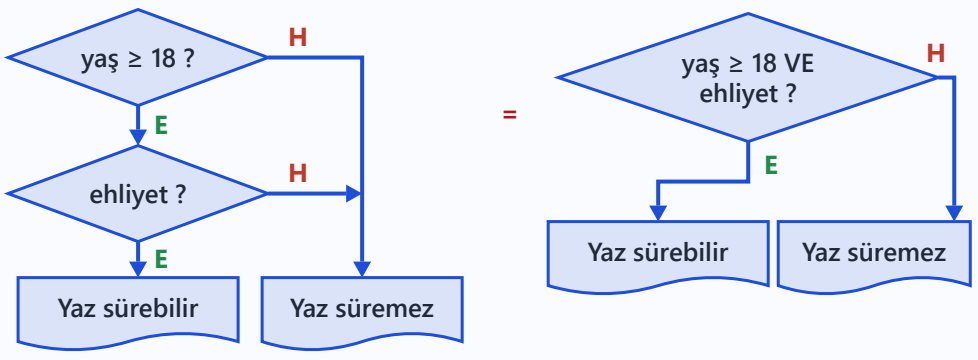
0–100 arasındaki bir notu harf notuna çeviren diyagramda dört ihtimal vardır: not ≥ 85 ise A, ≥ 70 ise B, ≥ 50 ise C, değilse F. Dört ihtimal için üç karar kutusu art arda bağlanır. Java karşılığı bir `else if` zinciridir:

```
if (not >= 85) {
    System.out.println("A");
} else if (not >= 70) {
    System.out.println("B");
} else if (not >= 50) {
    System.out.println("C");
} else {
    System.out.println("F");
}
```

72 girildiğinde `72 >= 85` yanlış, `72 >= 70` doğrudur; "B" yazılır ve üçüncü karar hiç sınanmaz. 45 girildiğinde üç koşul da yanlıştır ve "F" yazılır. 72 için iki, 45 için üç karar sınanır. İkinci koşulda `not < 85` diye ayrıca sormaya gerek yoktur: o karara yalnızca 85'ten küçük notlar ulaşır. Sınır değerler de doğru kola düşer: 85 → A, 84 → B, 70 → B, 50 → C, 49 → F.

1.4.5.4 Birleşik koşullar: VE, VEYA

Bazı kararlar iki koşula birden bağlıdır. Bir kişinin araba sürebilmesi için hem 18 yaşında ya da daha büyük olması hem de ehliyetinin olması gerekir. Bu bir **VE** koşuludur. Bir indirimden yararlanmak için öğrenci olmak ya da 65 yaşını geçmiş olmak yetiyorsa bu bir **VEYA** koşuludur.



Şekil 1.4.14: VE koşulunun iki çizimi. Solda iki ayrı karar art arda bağlanmıştır; ilk kararın Hayır çıkışı doğrudan "süremez"e gider. Sağda iki koşul tek bir karar kutusunda "yaş ≥ 18 VE ehliyet ?" olarak yazılmıştır. İki

çizim aynı sonucu verir.

Şekil 1.4.14 VE koşulunu iki biçimde çizer. Soldaki çizimde önce “yaş ≥ 18 ?” sorulur. Cevap Hayır ise ehliyetin olup olmadığına bakmaya gerek yoktur; kişi zaten süremez ve ilk kararın Hayır çıkışı doğrudan “Yaz süremez” kutusuna gider. Cevap Evet ise ikinci kararda “ehliyet ?” sorulur. Sağdaki çizim aynı mantığı tek bir karar kutusuna sığdırır. VEYA koşulunda durum tersine döner: iki kararla çizildiğinde ilk kararın Evet çıkışı doğrudan sonuca gider, çünkü koşullardan birinin doğru olması yeter.

Java'da VE `&&`, VEYA `||` ile yazılır: `if (yas >= 18 && ehliyet)`. Burada `ehliyet` bir `boolean` değişkendir. Tablo 1.4.6 dört olası durumu iki koşul için de verir; tablo bir Java programıyla üretilmiştir.

`yas`	`ehliyet`	`yas >= 18`	`yas >= 18 && ehliyet`	`yas >= 18 || ehliyet`
20	var	doğru	doğru	doğru (ehliyete bakılmaz)
20	yok	doğru	yanlış	doğru (ehliyete bakılmaz)
16	var	yanlış	yanlış (ehliyete bakılmaz)	doğru
16	yok	yanlış	yanlış (ehliyete bakılmaz)	yanlış

Tablo 1.4.6: VE ve VEYA koşullarının doğruluk tablosu. Parantez içindeki not, Java'nın ikinci koşula hiç bakmadığı durumları gösterir.

Tanım 1.4.5

Kısa devre değerlendirme (*short-circuit evaluation*), `&&` ve `||` işleçlerinde sonuç sol taraftan belli olunca sağ tarafın hiç hesaplanmamasıdır. `A && B` ifadesinde A yanlışsa sonuç yanlıştır ve B'ye bakılmaz. `A || B` ifadesinde A doğruysa sonuç doğrudur ve B'ye bakılmaz.

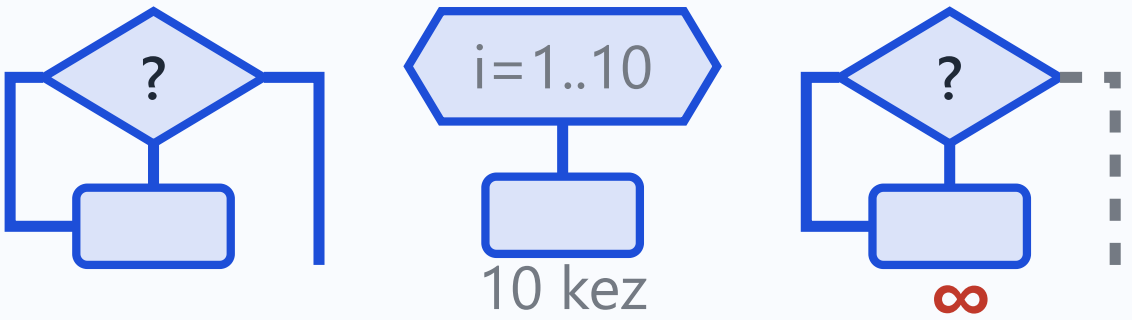
Kısa devre, iki kararla çizilen diyagramın Java'daki karşılığıdır. Diyagramda ilk kararın Hayır çıkışı ikinci kararı atlıyordu; Java da ikinci koşula yalnızca gerekirse bakar. Bu davranış bir deneyle görülebilir. Sağ taraftaki koşul, her çağrıldığında bir sayacı artıran bir metotla yazılırsa, `yas = 16` için `yas >= 18 && ehliyetVarMi(true)` ifadesi `false` verir ve sayaç 0'da kalır; metot hiç çağrılmamıştır. `yas = 20` için `yas >= 18 || ehliyetVarMi(false)` ifadesi `true` verir ve sayaç yine 0'dır.

Kısa devrenin pratik bir yararı vardır: soldaki koşul sağdakinin güvenle hesaplanabileceğini garanti edebilir. `adet != 0 && toplam / adet > 2` ifadesinde `adet` 0 ise bölme hiç yapılmaz. Sıra ters yazılsaydı, yani bölme sola konsaydı, `adet = 0` için program `ArithmeticException` ile dururdu.

1.4.6 Tekrarlı akış

Bilgisayarı güçlü kılan özelliklerden biri, aynı işi yorulmadan binlerce kez yapabilmesidir. Tekrarlı akış, işlemlerin birden çok kez tekrarlandığı akış türüdür. Diyagramda tekrar, geriye doğru giden bir okla gösterilir. Bu ok bir karar kutusunun üstüne döner; karar kutusu da tekrarın ne zaman biteceğine karar verir.

1.4.6.1 Döngü türleri



Şekil 1.4.15: Üç döngü türünün simgeleri. (a) Koşula bağlı döngü. (b) Belirli sayıda tekrar. (c) Sonsuz döngü.

Koşula bağlı döngü, bir koşul sağlandığı sürece tekrar eder. Kaç kez döneceği önceden bilinmeyebilir. “Kullanıcı 0 girene kadar sayı oku” böyle bir döngüdür; kullanıcının kaç sayı gireceği program yazılırken bilinmez. Java'da bu tür döngü genellikle `while` ile yazılır.

Belirli sayıda tekrar eden döngüde tekrar sayısı baştan bellidir. “1'den 10'a kadar sayıları yaz” işi tam 10 tur sürer. Java'da bu tür döngü genellikle `for` ile yazılır. `for` ile `while` arasındaki seçim bir zorunluluk değil, okunabilirlik meselesidir; her `for` döngüsü `while` ile, her `while` döngüsü de `for` ile yazılabilir.

Sonsuz döngüde koşul hep sağlandığı için işlemler durmadan tekrarlanır. Sonsuz döngü çoğunlukla bir hatadır. Döngü değişkenini güncellemeyi unutmak, koşulu yanlış kurmak ya da geri oku yanlış yere bağlamak sonsuz döngü üretir. Bazı programlar ise bilinçli olarak sonsuz döngüyle çalışır. Bir web sunucusu kapatılana kadar gelen istekleri bekler; bir gömülü sistemin ana döngüsü cihaz açık kaldıkça döner. Java'da bu tür döngü `while (true)` ile yazılır.

Döngülerin programa kazandırdığı üç şey vardır. Döngü **verimlidir**: aynı işlem, kod tekrar tekrar yazılmadan istenen sayıda yapılır. Döngü **dinamiktir**: tekrar sayısı girdiye göre değişebilir ve program farklı durumlara uyum sağlar. Döngü **kod tekrarını önler**: 1000 sayıyı yazdırmak için 1000 satır değil, 3–4 satır yeterlidir.

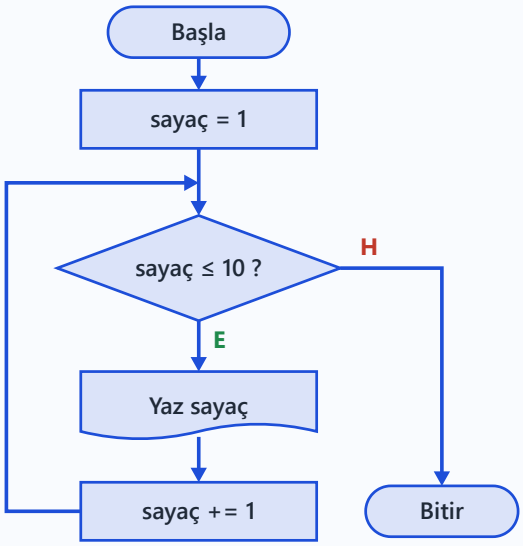
Her döngünün üç parçası vardır. **Başlangıç**, döngü değişkenine ilk değerini verir ve yalnızca bir kez çalışır. **Koşul**, her turdan önce sınanır ve döngünün sürüp sürmeyeceğini belirler. **Güncelleme**, döngü değişkenini değiştirir ve her turun sonunda çalışır. Bu üç parçadan biri eksik ya da yanlışsa döngü ya hiç çalışmaz, ya eksik ya da fazla döner, ya da hiç bitmez.

1.4.6.2 1'den 10'a kadar sayıları yazdırma

Tekrarlı akışın ilk örneği 1'den 10'a kadar sayıları ekrana yazmaktır. Algoritma yedi adımdır:

1. Başla.
2. Sayaç = 1.

3. Sayaç ≤ 10 ise 4'e git, değilse 7'ye git.
4. Sayacı yaz.
5. Sayacı 1 artır.
6. 3'e git.
7. Bitir.



Şekil 1.4.16: 1'den 10'a kadar sayıları yazan diyagram. “sayaç += 1” kutusundan çıkan ok sola, sonra yukarı gider ve karar kutusunun üstüne döner. Karar kutusunun Hayır çıkışı döngünün tek çıkış kapısıdır.

Algoritmanın 6. adımındaki “3'e git”, diyagramda karar kutusunun üstüne dönen geri oktur. Bu ok döngüyü oluşturur. Karar kutusu ise döngüden çıkış kapısıdır: sayaç 10'u aştığı anda Hayır çıkışından Bitir'e gidilir. Geri okun karar kutusuna, “sayaç = 1” kutusunun altına bağlandığına dikkat edilmelidir. Başlangıç değeri yalnızca bir kez, döngüden önce verilir.

Diyagramın Java karşılığı bir `while` döngüsüdür:

```
public class BirdenOna {
    public static void main(String[] args) {
        int sayac = 1;
        while (sayac <= 10) {
            System.out.print(sayac + " ");
            sayac += 1;
        }
    }
}
```

3. satır başlangıçtır: `sayac` 1 değerini alır. 4. satırdaki `while (sayac <= 10)` karar kutusudur. Koşul doğru olduğu sürece süslü parantezler arasındaki gövde çalışır. 5. satır sayacı yazar; `print` satır sonu eklemediği için sayılar aynı satırda, aralarında birer boşlukla görünür. 6. satır güncellemedir: `sayac`

`+= 1`, yani `sayac = sayac + 1`. Gövde bitince yürütme 4. satıra, yani koşula geri döner. Bu dönüş diyagramdaki geri oktur; Java'da ayrıca bir "git" deyimi yazılmaz, `while`'ın kapanan süslü parantezi bu işi görür.

Tablo 1.4.7 döngünün izlemesidir. Tablonun ortasındaki 4.–8. turlar aynı kalıbı izlediği için kısaltılmıştır.

Tur	`sayac` (tur başı)	Koşul	Yazılan	`sayac` (tur sonu)	Çıktı
1	1	1 <= 10 → doğru	1	2	1
2	2	2 <= 10 → doğru	2	3	1 2
3	3	3 <= 10 → doğru	3	4	1 2 3
…	…	…	…	…	…
9	9	9 <= 10 → doğru	9	10	1 2 3 4 5 6 7 8 9
10	10	10 <= 10 → doğru	10	11	1 2 3 4 5 6 7 8 9 10
—	11	11 <= 10 → yanlış	—	11	1 2 3 4 5 6 7 8 9 10

Tablo 1.4.7: `BirdenOna` programının izlenmesi. Son satır, koşulun yanlış olduğu ve döngüden çıkılan sınamadır.

```
1 2 3 4 5 6 7 8 9 10
```

Tablodan iki önemli sayı okunur. Gövde 10 kez çalışmıştır; her turda bir sayı yazılmıştır. Koşul ise 11 kez sınanmıştır: on kez doğru, son kez yanlış çıkmıştır. Koşulun gövdeden bir fazla sınanması, karar kutusu başta olan her döngünün özelliğidir. Döngüden çıkıldığında `sayac` 10 değil, 11'dir. Çünkü 10 yazıldıktan sonra sayaç bir kez daha artırılmış, sonra koşul yanlış çıkmıştır.

1.4.6.3 Sınır koşulu: < mı, ≤ mü?

Döngü koşulundaki tek bir karakter tekrar sayısını değiştirir. Sayaç 1'den başlayıp her turda 1 artarsa koşulun dört farklı yazılışı Tablo 1.4.8'deki sonuçları verir.

Koşul	Gövde	Koşul sınaması	Çıkışta `sayac`	Son yazılan
`sayac < 10`	9	10	10	9
`sayac <= 10`	10	11	11	10
`sayac < 11`	10	11	11	10
`sayac <= 11`	11	12	12	11

Tablo 1.4.8: Sayaç 1'den başlayıp 1'er artarken koşulun yazılışına göre gövdenin çalışma sayısı (Java ile ölçüldü).

`sayac < 10` koşulu 10'u yazmaz; çünkü sayaç 10 olduğunda `10 < 10` yanlıştır. Gövde 9 kez çalışır ve 1'den 9'a kadar sayılar yazılır. `sayac <= 10` ve `sayac < 11` koşulları aynı anlamdadır; ikisi de 10 tur döner ve 10'u da yazar. `sayac <= 11` ise bir tur fazla döner ve 11'i de yazar.

Tanım 1.4.6

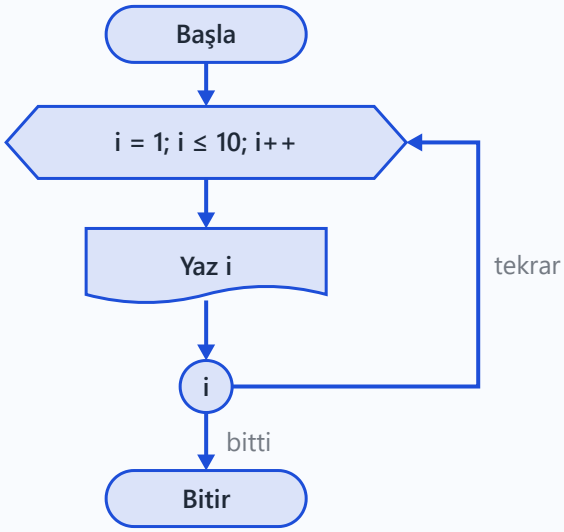
Bir eksik / bir fazla hatası (*off-by-one error*), bir döngünün sınır koşulunun yanlış yazılması yüzünden gerekenden bir tur eksik ya da bir tur fazla dönmesidir.

Bu hata programlamanın en yaygın hatalarından biridir ve derleyici onu yakalamaz; çünkü yanlış yazılan koşul da geçerli bir Java ifadesidir. Hatayı önlemenin basit bir yolu vardır: koşul *son değerle* sınanır. "1'den 10'a kadar" ifadesi 10'u da kapsar. O hâlde sayaç 10 iken koşulun doğru olması gerekir. `10 < 10` yanlış, `10 <= 10` doğrudur; doğru koşul `sayac <= 10`'dur. Aynı yoklama ilk değerle de yapılır: sayaç 1 iken koşul doğru olmalıdır, yoksa döngü hiç başlamaz.

1.4.6.4 Döngü sembolü ve for

Aynı problem, yani 1'den 10'a kadar sayıları yazdırmak, döngüye özel altıgen sembolle daha kısa çizilebilir. Algoritma beş adıma iner:

1. Başla.
2. Döngü: 1'den 10'a kadar.
3. Döngü değişkenini yaz.
4. Döngü sonu.
5. Bitir.



Şekil 1.4.17: Döngü sembolüyle 1'den 10'a kadar yazdırma. Altıgen başlangıcı, koşulu ve güncellemeyi tek kutuda toplar. İçinde "i" yazan küçük çember döngü sonudur: akış oradan "tekrar" okuyla altıgene döner, döngü bitince "bitti" okuyla Bitir'e geçer.

Altıgen sembol döngünün üç parçasını tek kutuda toplar. **Başlangıç** `i = 1`'dir ve bir kez çalışır. **Koşul** `i ≤ 10`'dur ve her turdan önce sınanır. **Güncelleme** `i++`'dır, yani i'yi 1 artırır ve her turdan sonra çalışır. İçinde "i" yazan küçük çember döngünün sonudur. Gövde bitince akış bu çemberden altıgene döner; altıgen güncellemeyi yapar ve koşulu yeniden sınar. Koşul yanlış olunca akış çemberin altındaki "bitti" okundan Bitir'e geçer.

Altıgenin Java karşılığı `for` deyimidir. Üç parça `for`'un parantezi içinde noktalı virgülle ayrılarak aynı sırayla yazılır:

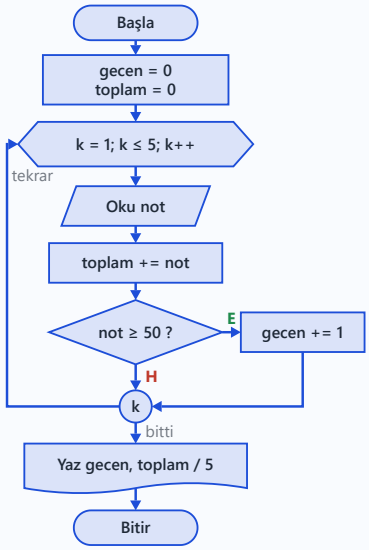
```
public class ForBirdenOna {
    public static void main(String[] args) {
        for (int i = 1; i <= 10; i++) {
            System.out.print(i + " ");
        }
        System.out.println();
    }
}
```

Bu program ile `while` sürümü aynı çıktıyı verir: `1 2 3 4 5 6 7 8 9 10`. Fark yalnızca yazılıştadır. `for`, döngünün üç parçasını tek satırda topladığı için okuyan kişi tekrar sayısını bir bakışta görür. `while` sürümünde başlangıç döngüden önce, güncelleme ise gövdenin sonunda yazılır ve gözden kaçabilir.

`for` ile `while` arasında küçük ama önemli bir fark daha vardır. `for`'un başlangıç kısmında tanımlanan `int i` değişkeni yalnızca döngü içinde geçerlidir. Döngüden sonra `i` kullanılmak istenirse derleyici `cannot find symbol` hatası verir. `while` sürümünde ise `sayac` döngüden önce tanımlandığı için döngüden sonra da kullanılabilir ve değeri 11'dir.

1.4.6.5 Döngü içinde karar: sayaç ve biriktirici

Döngünün gövdesinde her türlü adım bulunabilir; bir karar kutusu da. Örnek problem şöyledir: beş öğrencinin notu okunur; kaç öğrencinin geçtiği (not ≥ 50) ve sınıf ortalaması yazılır.



Şekil 1.4.18: Döngü içinde karar. Döngü sembolü k = 1'den 5'e kadar döner. Her turda bir not okunur ve toplama eklenir. Karar kutusunun Evet kolu sayacı (gecen) 1 artırır; Hayır kolu sayacı atlayıp döngü sonuna gider.

```
int gecen = 0, toplam = 0;
for (int k = 1; k <= 5; k++) {
  int not = giris.nextInt();
  toplam += not;
  if (not >= 50) {
    gecen++;
  }
}
System.out.println(gecen + " geçti, ortalama "
    + toplam / 5.0);
```

Bu program iki önemli döngü kalıbını birlikte kullanır.

Tanım 1.4.7

Sayaç (*counter*), bir koşul sağlandığında 1 artan ve böylece bir olayın kaç kez olduğunu sayan değişkendir. **Biriktirici** (*accumulator*), döngünün her turunda bir değer eklenerek sonucu toplayan değişkendir. İkisi de döngüden önce başlangıç değeri alır.

Programda `gecen` bir sayaçtır: yalnızca `not >= 50` doğru olduğunda `gecen++` ile 1 artar. `toplam` bir biriktiricidir: koşul ne olursa olsun her turda okunan not ona eklenir. İkisinin de başlangıç değeri 0'dır; toplama ve saymaya sıfırdan başlanır. Karar kutusu döngünün içindedir ve her turda yeni okunan notla yeniden sınanır. Tablo 1.4.9 programın 45, 70, 50, 30 ve 88 girdileriyle izlenmesidir.

k	not	toplam	Karar	gecen
1	45	45	45 >= 50 → yanlış	0
2	70	115	70 >= 50 → doğru	1
3	50	165	50 >= 50 → doğru	2
4	30	195	30 >= 50 → yanlış	2
5	88	283	88 >= 50 → doğru	3

Tablo 1.4.9: Sayaç ve biriktiricinin izlenmesi. Her satır bir turun sonundaki durumu gösterir.

Döngü bittiğinde `gecen = 3` ve `toplam = 283` olur. Program şunu yazar:

```
3 geçti, ortalama 56.6
```

Bu kısa programda iki Java ayrıntısı saklıdır. Birincisi sınırdaki nottur. 50 alan öğrenci geçer, çünkü koşul `>=` ile yazılmıştır. Koşul `not > 50` yazılsaydı 50 alan öğrenci kalırdı ve sonuç 3 değil 2 olurdu. Sınır değeri koşulu tek karakterle değiştirir; bir eksik/bir fazla hatasının karar kutusundaki biçimi budur.

İkincisi bölmedir. `toplam / 5.0` ifadesinde bölen ondalıklı bir sayı olduğu için Java bölmeyi ondalıklı yapar: 283 / 5,0 = 56,6. Bölen `5` yazılsaydı iki tam sayı bölünürdü. Java'da iki tam sayının bölümü yine tam sayıdır; ondalık kısım atılır ve 283 / 5 = 56 olur. Diyagramdaki “toplam / 5” ifadesi bu ayrımı göstermez; Java'ya çevirirken türlere dikkat etmek gerekir. Ekranda ondalık ayırıcının nokta olarak (56.6) göründüğüne de dikkat edilmelidir; `println` sayıları her zaman noktayla yazar.

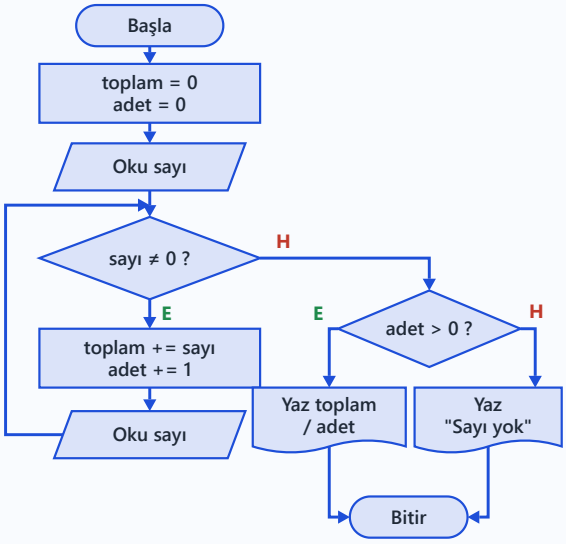
Sayaç ve biriktiricinin yanında üçüncü bir kalıp daha vardır: **bayrak**. Bayrak, bir olayın olup olmadığını tutan `boolean` bir değişkendir. Başlangıçta `false`'tur; olay gerçekleşince `true` yapılır, örneğin `bulundu = true`. Döngü bittikten sonra bayrağa bakılarak olayın en az bir kez olup olmadığı anlaşılır.

1.4.6.6 Bekçi değerli döngü

Önceki örnekte not sayısı baştan belliydi: beş. Çoğu zaman ise kaç değer geleceği önceden bilinmez. Bir öğretmen bazen 3, bazen 40 not girer. Bu durumda kullanıcı, girişin bittiğini özel bir değerle bildirir.

Tanım 1.4.8

Bekçi değer (*sentinel value*), girdinin bittiğini bildiren ve veri olarak sayılmayan özel bir değerdir. 0, −1 ya da “son” gibi bir değer seçilir.



Şekil 1.4.19: Bekçi değerli döngü. İlk sayı döngüden önce okunur. Gövdenin sonunda sıradaki sayı okunur ve geri okla karara dönülür. Döngüden sonraki “adet > 0 ?” kararı hiç sayı girilmeyen durumu ayırır.

```
int toplam = 0, adet = 0;
int sayi = giris.nextInt();
while (sayi != 0) {
  toplam += sayi;
  adet++;
  sayi = giris.nextInt();
}
if (adet > 0) {
  System.out.println("Ortalama: "
      + (double) toplam / adet);
} else {
  System.out.println("Sayı yok");
}
```

Kaç değer geleceği bilinmediği için döngü koşula bağlıdır ve `while` ile yazılır. Diyagramın en önemli özelliği **önce okuma** düzenidir. İlk değer döngüden önce, 2. satırda okunur. Sıradaki değer ise gövdenin sonunda, 6. satırda okunur. Böylece okunan her değer, gövdeye girmeden önce koşulda sınanır. Bekçi geldiğinde koşul hemen yanlış olur ve bekçi toplama hiç katılmaz.

Tablo 1.4.10 programın 12, 7, 5, 0 girdisiyle izlenmesidir. Her tur satırında `sayi` sütunu, o turda toplama eklenen değeri ve gövdenin sonunda okunan yeni değeri gösterir.

Adım	`sayi`	Koşul	`toplam`	`adet`
önce okuma	12	—	0	0
1. tur	12 → 7	12 != 0 → doğru	12	1
2. tur	7 → 5	7 != 0 → doğru	19	2

Adım	`sayi`	Koşul	`toplam`	`adet`
3. tur	5 → 0	5 != 0 → doğru	24	3
karar	0	0 != 0 → yanlış	24	3

Tablo 1.4.10: Bekçi değerli döngünün 12, 7, 5, 0 girdisiyle izlenmesi. 0 bekçidir ve toplama katılmaz.

Döngüden çıkıldığında `toplam = 24` ve `adet = 3` 'tür. Döngüden sonraki karar `adet > 0` koşulunu sınar; koşul doğrudur ve ortalama yazılır:

```
Ortalama: 8.0
```

Ortalama hesabındaki `(double) toplam / adet` ifadesine dikkat edilmelidir. `(double)` bir **tür dönüşümüdür**: `toplam` 'ı bölmeden önce ondalıklı sayıya çevirir. Böylece bölme ondalıklı yapılır ve 24 / 3 = 8.0 olur. Dönüşüm yazılmasaydı iki tam sayı bölünürdü; 24 / 3 için sonuç yine 8 olurdu, ama 25 / 3 için 8,33… yerine 8 çıkardı.

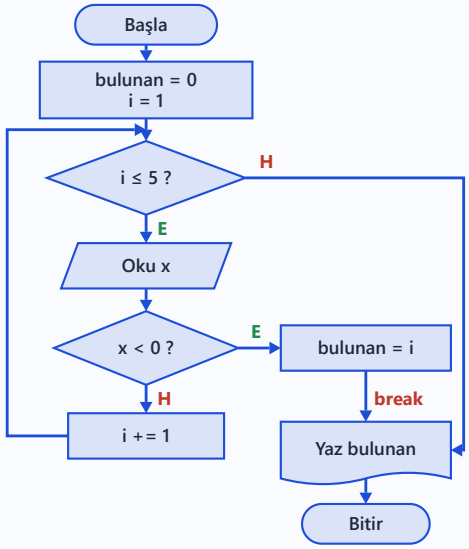
Bekçi değer, gerçek veride görülemeyecek bir değer olmalıdır. Öğrenci notları için −1 uygun bir bekçidir; 0 değildir, çünkü sınavdan 0 alan bir öğrenci olabilir. Bu örnekteki gibi sıradan tam sayılar okunuyorsa 0 bekçi seçilebilir, ama o zaman 0 hiçbir zaman veri olarak girilemez.

Döngüden sonraki `adet > 0` denetimi gereksiz değildir. Kullanıcı ilk sayı olarak 0 girerse döngü hiç çalışmaz; `toplam` ve `adet` 0'da kalır. Denetim olmasaydı 0 / 0 hesaplanırdı. Java'da bu bölmenin sonucu türe göre değişir. `(double) 0 / 0` ifadesi hata vermez, `NaN` (*Not a Number*, sayı değil) değerini üretir ve ekrana `Ortalama: NaN` yazılırdı. Tam sayılarda ise `0 / 0` programı `java.lang.ArithmeticException: / by zero` hatasıyla durdurur. Denetim sayesinde program bu durumda `Sayı yok` yazar.

> **Sık yapılan hata**
>
> Okuma yalnızca gövdenin başına konursa, yani “önce okuma” yerine “döngü içinde okuma” yapılırsa bekçi de toplama eklenir. Bekçi 0 ise toplam değişmez ama `adet` bir fazla sayılır ve ortalama yanlış çıkar. Bekçi −1 ise hem toplam hem adet yanlış olur. Önce okuma düzeni bu yüzden vardır: her değer kullanılmadan önce sınanır.

1.4.6.7 Döngüden erken çıkış

Bir aramada aranan bulununca kalan turlara gerek kalmaz. Örnek problem şöyledir: beş sayı sırayla okunur; ilk negatif sayının kaçıncı sırada girildiği yazılır. Negatif sayı bulununca geri kalan sayıları okumanın anlamı yoktur.



Şekil 1.4.20: Döngüden erken çıkış. Döngünün iki çıkışı vardır: “i ≤ 5 ?” kararının Hayır çıkışı (aranan bulunamadı) ve “x < 0 ?” kararının Evet kolu (aranan bulundu, break). İki yol “Yaz bulunan” kutusunda birleşir.

Diyagramda döngünün içinde ikinci bir karar vardır: “x < 0 ?”. Bu kararın Evet kolu, `bulunan = i` atamasından sonra akışı doğrudan döngünün dışına çıkarır. Böylece döngünün iki çıkışı olur. Birinci çıkış olağan çıkıştır: `i` 5'i aşınca koşul yanlış olur; bu, negatif sayının bulunamadığı anlamına gelir. İkinci çıkış erken çıkıştır: negatif sayı bulununca döngü hemen biter. İki yol “Yaz bulunan” kutusunda birleşir. Java'da ikinci çıkış `break` deyimiyle yazılır:

```
int bulunan = 0;
for (int i = 1; i <= 5; i++) {
  int x = giris.nextInt();
  if (x < 0) {
    bulunan = i;
    break;
  }
}
System.out.println("Sıra: " + bulunan);
```

`break` yalnızca içinde bulunduğu döngüyü bitirir. Yürütme döngüden sonraki ilk deyimle, burada `println` ile sürer. `bulunan` değişkeninin başlangıç değeri 0'dır ve “bulunamadı” anlamına gelir. Sıra 1'den başladığı için 0 hiçbir gerçek sırayla karışmaz. Tablo 1.4.11 programın 4, 9, −2, 6, 1 girdisiyle izlenmesidir.

`i`	**Döngü koşulu**	`x`	**İç karar**	`bulunan`
1	1 <= 5 → doğru	4	4 < 0 → yanlış	0
2	2 <= 5 → doğru	9	9 < 0 → yanlış	0

`i`	**Döngü koşulu**	`x`	**İç karar**	`bulunan`
3	3 <= 5 → doğru	−2	−2 < 0 → doğru; `break`	3

Tablo 1.4.11: `break`'li aramanın 4, 9, −2, 6, 1 girdisiyle izlenmesi. Üçüncü turda döngü biter; 6 ve 1 hiç okunmaz.

Üçüncü turda `x = -2` okunur ve `-2 < 0` doğrudur. `bulunan = 3` olur ve `break` çalışır. `break`, `for`'un güncelleme kısmını (`i++`) ve koşulu da atlar; döngü hemen biter. Program `Sıra: 3` yazar. Negatif sayı hiç girilmeseydi, örneğin 4, 9, 2, 6, 1 girdisinde, döngü beş tur döner, `i = 6` olunca koşulun Hayır çıkışından çıkar ve `Sıra: 0` yazılırdı.

Bu diyagram yapısal değildir: döngü bloğunun iki çıkışı vardır. Böhm–Jacopini teoremi, aynı işin tek çıkışlı bir yapıyla da yapılabileceğini söyler. Yapısal karşılık, `break` yerine bir bayrak kullanır:

```
int bulunan = 0;
boolean bulundu = false;
for (int i = 1; i <= 5 && !bulundu; i++) {
  int x = giris.nextInt();
  if (x < 0) {
    bulunan = i;
    bulundu = true;
  }
}
```

Burada `bulundu` bir bayraktır. Negatif sayı bulunduğunda `true` yapılır. Döngü koşulu artık iki parçalıdır: `i <= 5 && !bulundu`. `!` işleci "değil" demektir; `!bulundu`, "henüz bulunamadı" anlamına gelir. Bayrak `true` olunca koşul yanlış olur ve döngü olağan çıkışından biter. Tablo 1.4.12 aynı girdiyle bu sürümün izlenmesidir.

`i`	**Döngü koşulu**	`x`	`bulundu`	`bulunan`
1	1 <= 5 && !false → doğru	4	false	0
2	2 <= 5 && !false → doğru	9	false	0
3	3 <= 5 && !false → doğru	−2	true	3
4	4 <= 5 && !true → yanlış	—	true	3

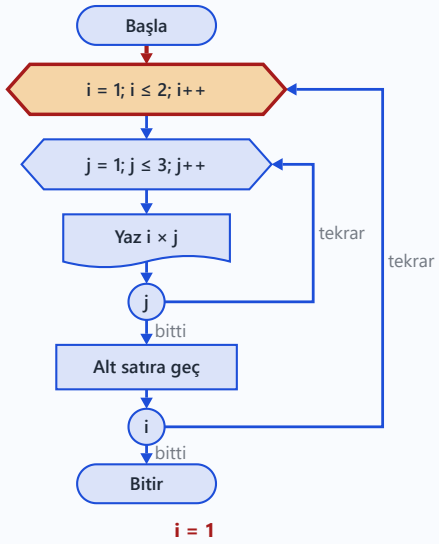
Tablo 1.4.12: Bayraklı sürümün aynı girdiyle izlenmesi. Döngünün tek çıkışı koşulun yanlış olmasıdır.

İki sürüm aynı sonucu verir: `Sıra: 3`. Aralarındaki fark, döngü bittiğinde `i`'nin değerindedir. `break`'li sürümde döngü `i = 3` iken biter. Bayraklı sürümde ise üçüncü turun sonunda `i++` çalışır, `i = 4` olur ve koşul bu değerle yanlış çıkar. Bu yüzden bayraklı sürümde sıra `i`'den değil, `bulunan`'dan okunur. Hangi sürümün seçileceği bir üslup meselesidir. Kısa döngülerde `break` okunaklıdır; uzun döngülerde ise döngünün tek çıkışlı olması izlemeyi kolaylaştırır.

1.4.6.8 İç içe döngü

Bir döngünün gövdesinde başka bir döngü bulunabilir. Bu yapıya **iç içe döngü** denir. Dış döngünün her turunda iç döngü baştan sona bütün turlarını döner. Aşağıdaki program küçük bir çarpım tablosu yazar:

```
for (int i = 1; i <= 2; i++) {
  for (int j = 1; j <= 3; j++) {
    System.out.print(i * j + " ");
  }
  System.out.println();
}
```



Şekil 1.4.21: İç içe döngünün diyagramı. Dış altıgen i'yi, iç altıgen j'yi yönetir. İç döngünün "j" çemberi iç altıgene, dış döngünün "i" çemberi dış altıgene döner. "Alt satıra geç" kutusu iç döngünün dışında, dış döngünün içindedir.

Dış döngü `i` = 1 ve 2 için iki tur döner. İç döngü her dış turda `j` = 1, 2, 3 için üç tur döner ve `i * j` değerini yazar. İç döngü bitince `println()` alt satıra geçer. Tablo 1.4.13 programın bütün adımlarıdır.

`i`	`j`	Koşul	Yazılan
1	1	1 <= 3 → doğru	1 × 1 = 1
1	2	2 <= 3 → doğru	1 × 2 = 2
1	3	3 <= 3 → doğru	1 × 3 = 3
1	4	4 <= 3 → yanlış: `println()`	(alt satır)
2	1	1 <= 3 → doğru	2 × 1 = 2
2	2	2 <= 3 → doğru	2 × 2 = 4
2	3	3 <= 3 → doğru	2 × 3 = 6

`i`	`j`	Koşul	Yazılan
2	4	4 <= 3 → yanlış: `println()`	(alt satır)
3	—	3 <= 2 → yanlış: dış döngü biter	—

Tablo 1.4.13: İç içe döngünün izlenmesi. İç döngü her dış turda `j = 1`'den yeniden başlar.

```
1 2 3
2 4 6
```

Tabloda dikkat edilecek nokta, ikinci dış turda `j`'nin yeniden 1'den başlamasıdır. Bunun nedeni, `int j = 1` başlangıcının iç `for`'un parçası olmasıdır. İç `for` deyimi dış döngünün her turunda baştan çalıştığı için başlangıç da her seferinde yeniden yapılır. Diyagramda bu, dış döngünün her turunda akışın iç altıgene yeniden üstten girmesiyle gösterilir.

Çalışma sayıları çarpımla bulunur. İç gövde, yani `print`, 2 × 3 = 6 kez çalışır. `println()` ise dış gövdenin parçası olduğu için yalnızca 2 kez çalışır. Genel olarak dış döngü m, iç döngü n tur dönüyorsa iç gövde m × n kez çalışır. Bu yüzden iç içe döngüler büyük girdilerde hızla yavaşlar: 1000 × 1000'lik bir iç içe döngü bir milyon kez döner.

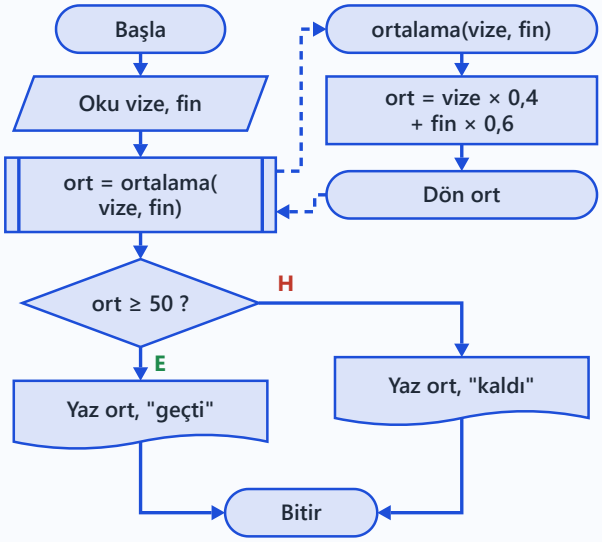
1.4.7 Alt program

Diyagramlar büyüdükçe tek sayfaya sığmaz ve okunması zorlaşır. Bazı adım dizileri de diyagramın farklı yerlerinde tekrar tekrar gerekir. Her iki sorunun çözümü aynıdır: diyagram anlamlı parçalara bölünür.

1.4.7.1 Alt program sembolü ve metot

Tanım 1.4.9

Alt program (*subprogram, predefined process*), ayrı bir diyagram olarak çizilen ve ana diyagramdan alt program sembolüyle çağrılan adım dizisidir. Java'daki karşılığı **metottur**.



Şekil 1.4.22: Alt program. Solda ana diyagram, sağda "ortalama(vize, fin)" alt programı. Ana diyagramdaki iki yanı çizgili kutu alt programı çağırır. Kesikli oklar akışın alt programa geçişini ve "Dön ort" kutusundan çağrının hemen sonrasına dönüşünü gösterir.

Şekil 1.4.22 bir öğrencinin dönem ortalamasını hesaplayan diyagramı iki parçaya böler. Sağdaki parça alt programdır. Başla kutusunun yerinde alt programın adı ve girdileri yazar: "ortalama(vize, fin)". Bu girdilere **parametre** denir. Alt program vize notunun %40'ını ve final notunun %60'ını toplar. Sonunda Bitir yerine bir **Dön** kutusu bulunur: "Dön ort". Dön kutusu hesaplanan değeri alt programı çağıran yere geri verir.

Soldaki ana diyagram önce vize ve final notlarını okur. Sonra iki yanı çizgili alt program sembolüyle alt programı çağırır: "ort = ortalama(vize, fin)". Çağrı anında akış alt programa geçer. Alt program Dön kutusuna gelince sonuç çağırana verilir ve akış çağrının hemen sonrasından sürer. Ana diyagram bu sonucu `ort` değişkenine atar ve "ort ≥ 50 ?" kararıyla öğrencinin geçip geçmediğini yazar.

Alt programın üç kazancı vardır. Aynı adımlar her ihtiyaç duyulduğunda yeniden çizilmez; bir kez çizilir, defalarca çağrılır. Her parça ayrı ayrı sınanabilir; ortalama hesabı doğruysa ana diyagramda yalnızca okuma ve karar denetlenir. Ana diyagram kısa ve okunaklı kalır; okuyucu ayrıntıya girmeden programın ne yaptığını görür.

Java karşılığı şöyledir:

```
static double ortalama(int vize, int fin) {
  double ort = vize * 0.4 + fin * 0.6;
  return ort;
}
public static void main(String[] args) {
  Scanner giris = new Scanner(System.in);
  int vize = giris.nextInt();
  int fin = giris.nextInt();
  double ort = ortalama(vize, fin);
  if (ort >= 50) System.out.println(ort + " geçti");
  else System.out.println(ort + " kaldı");
}
```

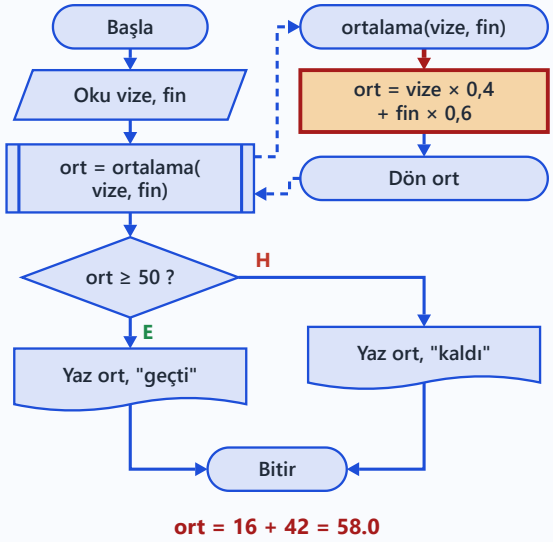
1.–4. satırlar alt programdır. `static double ortalama(int vize, int fin)` başlığı metodun adını (`ortalama`), iki `int` parametresini ve döndürdüğü değerin türünü (`double`) belirtir. `return ort;` deyimi diyagramdaki Dön kutusudur. 9. satırdaki `ortalama(vize, fin)` ifadesi alt program sembolüdür; çağrıdır.

Parametre adı olarak `final` yerine `fin` seçilmiştir. Bunun nedeni, `final`'ın Java'da ayrılmış bir anahtar sözcük olmasıdır; değişken adı olarak kullanılamaz. `int final = 70;` yazılırsa derleyici `error: not a statement` başta olmak üzere bir dizi hata verir. Diyagramda "final" yazmak serbesttir, ama Java'ya çevirirken anahtar sözcükler değişken adı olamaz.

1.4.7.2 Alt program çağrısını izlemek

Kullanıcı aynı satıra 40 ve 70 girdiğinde program şu adımlarla çalışır.

1. Program `main`'den başlar. 7. ve 8. satırlar iki notu okur: `vize = 40`, `fin = 70`.
2. 9. satırda `ortalama` çağrılır. 40 ve 70 değerleri metodun parametrelerine **kopyalanır**. Metodun `vize` ve `fin` değişkenleri, `main`'deki aynı adlı değişkenlerden ayrıdır; yalnızca değerleri aynıdır.
3. Alt programın kendi diyagramı çalışır: `ort = 40 × 0,4 + 70 × 0,6 = 16 + 42 = 58.0`. Bu `ort`, metodun yerel değişkenidir; `main`'deki `ort`'tan ayrıdır.
4. `return ort;` çalışır: 58.0 değeri çağırana verilir. Metot için ayrılan bellek alanı, yani metodun çerçevesi, çağrı yığınından kalkar; metodun yerel değişkenleri de onunla birlikte yok olur.
5. Akış 9. satırdan sürer: dönen değer `main`'deki `ort`'a atanır.
6. 10. satırdaki karar: `58.0 >= 50` doğrudur; Evet kolu `58.0 geçti` yazar.
7. `main` biter (Bitir).



Şekil 1.4.23: Alt programın çalıştığı an. Ana diyagramda çağrı kutusu bekler; alt programda "ort = vize × 0,4 + fin × 0,6" kutusu çalışır ve 16 + 42 = 58.0 hesaplanır.

Aynı alt program başka girdilerle yeniden çağrılabilir. Tablo 1.4.14 iki çalıştırmayı karşılaştırır. İkinci çalıştırmada 45 ve 30 girilir; ortalama 36.0 çıkar ve öğrenci kalır.

`vize`	`fin`	Alt programdaki işlem	Dönen değer	Karar	Çıktı
40	70	40 × 0,4 + 70 × 0,6 = 16,0 + 42,0	58,0	58,0 >= 50 → doğru	58.0 geçti
45	30	45 × 0,4 + 30 × 0,6 = 18,0 + 18,0	36,0	36,0 >= 50 → yanlış	36.0 kaldı

Tablo 1.4.14: Alt programın iki çağrısı (Java ile çalıştırıldı).

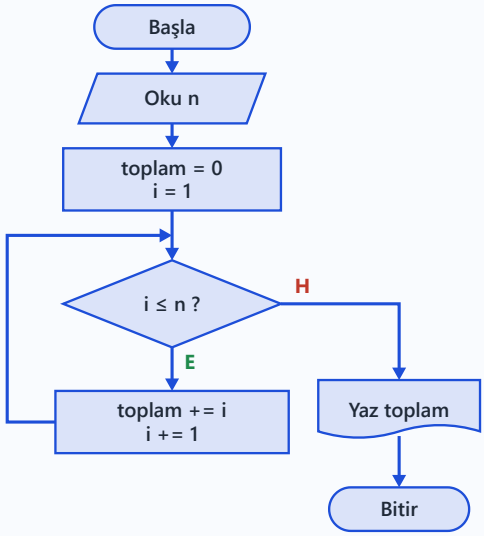
Çıktıda ortalamanın `58.0` biçiminde, sonunda `.0` ile yazıldığına dikkat edilmelidir. Metot `double` döndürdüğü için Java sayıyı ondalıklı biçimde yazar; değer tam sayı olsa bile ondalık kısmı gösterir. `40 * 0.4` çarpımında `int` bir değer `double` bir değerle çarpıldığı için sonuç da `double`'dır.

1.4.8 Uygulama ve sınama

Bir diyagram ya da program yazıldıktan sonra doğru çalıştığından emin olmak gerekir. Bunun iki yolu vardır: diyagramı ya da programı kâğıt üzerinde elle izlemek ve onu iyi seçilmiş girdilerle denemek.

1.4.8.1 1'den n'ye kadar toplam

Problem, kullanıcının girdiği bir n sayısı için 1 + 2 + … + n toplamını bulmaktır. Bu problemde üç akış türü bir aradadır: n'nin okunması ve başlangıç değerleri sıralı, toplama tekrarlı, sonucun yazılması yine sıralı akıştır.



Şekil 1.4.24: 1'den n'ye kadar toplam. “toplam = 0, i = 1” başlangıçtır. “i ≤ n ?” kararı Evet oldukça “toplam += i, i += 1” kutusu çalışır ve geri ok karara döner. Hayır çıkışından toplam yazılır.

```
System.out.print("n: ");
int n = giris.nextInt();
int toplam = 0;
int i = 1;
while (i <= n) {
    toplam += i;
    i += 1;
}
System.out.println("Toplam: " + toplam);
```

`toplam` bir biriktiricidir: 0'dan başlar ve her turda o anki `i` kadar büyür. `i` ise bir sayaçtır: 1'den başlar ve her turda 1 artar. Tablo 1.4.15 kullanıcının 4 girdiği çalıştırmadır.

Adım	Koşul ve işlem	`toplam`	`i`
başlangıç	—	0	1
1. tur	1 <= 4 → doğru; 0 + 1 = 1	1	2
2. tur	2 <= 4 → doğru; 1 + 2 = 3	3	3
3. tur	3 <= 4 → doğru; 3 + 3 = 6	6	4
4. tur	4 <= 4 → doğru; 6 + 4 = 10	10	5
karar	5 <= 4 → yanlış	10	5

Tablo 1.4.15: 1'den n'ye toplamın n = 4 için izlenmesi.

Gövde 4 kez, koşul 5 kez çalışır. Program `Toplam: 10` yazar; 1 + 2 + 3 + 4 = 10'dur. n = 5 için aynı diyagram bir tur daha döner ve `Toplam: 15` yazar. Bu sonuç Gauss'un formülüyle de denetlenebilir: 1'den n'ye toplam n(n + 1)/2'dir. n = 4 için 4 · 5 / 2 = 10, n = 5 için 5 · 6 / 2 = 15 bulunur.

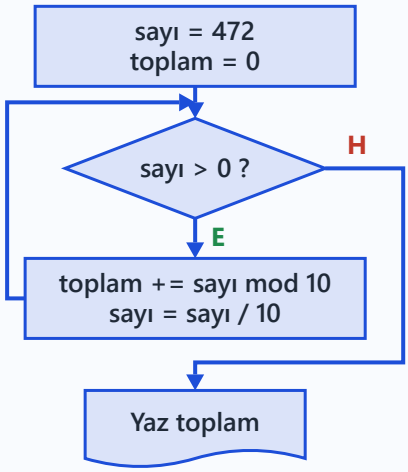
Biriktiricinin başlangıç değeri sonucu doğrudan etkiler. “toplam = 0” kutusu yanlışlıkla “toplam = 1” yazılırsa döngü yine 1 + 2 + 3 + 4 = 10 ekler, ama başlangıç 1 olduğu için sonuç 11 çıkar. Toplama için biriktirici 0 ile, çarpma için 1 ile başlatılır. Örneğin n! hesaplanırken çarpım `f = 1` ile başlar; `f = 0` ile başlasaydı her çarpım 0 verirdi. Derleyici bu mantık hatalarını yakalamaz.

1.4.8.2 Masa başı kontrol ve izleme tablosu

Tanım 1.4.10

Masa başı kontrol (*desk check*), bir diyagramın ya da kodun bilgisayar kullanılmadan, bir girdiyle kâğıt üzerinde izlenmesidir. **İzleme tablosu** (*trace table*), bu izlemede değişkenlerin her adımdaki değerlerini gösteren tablodur: her değişkene bir sütun, her tura bir satır ayrılır.

Örnek olarak bir sayının basamaklarını toplayan küçük bir program alınabilir. 472 sayısının basamakları 4, 7 ve 2'dir; toplamları 13'tür.



Şekil 1.4.25: Basamak toplamı diyagramı. Her turda son basamak (sayı mod 10) toplama eklenir ve sayı 10'a bölünerek son basamağı atılır.

```
int sayi = 472, toplam = 0;
while (sayi > 0) {
  toplam += sayi % 10;
  sayi /= 10;
}
System.out.println(toplam);
```

Program iki tam sayı işlemine dayanır. `sayi % 10` sayının son basamağını verir: 472 % 10 = 2. `sayi / 10` ise tam sayı bölmesidir ve son basamağı atar: 472 / 10 = 47. Basamaklar böylece sağdan sola, birer birer alınır. `toplam += sayi % 10`, `toplam = toplam + sayi % 10`'un kısa yazılışıdır; `sayi /= 10` de `sayi = sayi / 10` demektir. Tablo 1.4.16 masa başı kontrolün izleme tablosudur.

Adım	`sayi % 10`	`toplam`	`sayi`	Koşul
başlangıç	—	0	472	472 > 0 → doğru
1. tur	2	2	47	47 > 0 → doğru
2. tur	7	9	4	4 > 0 → doğru
3. tur	4	13	0	0 > 0 → yanlış

Tablo 1.4.16: Basamak toplamının 472 için izleme tablosu.
Koşul sütunu, o satırdaki değerle yapılan bir sonraki sınamadır.

Tablo döngünün kaç kez döndüğünü de gösterir: 3 tur ve 4 sınama. Program `13` yazar. Tablonun asıl değeri, hataları program çalıştırılmadan göstermesidir. “sayı = sayı / 10” kutusu unutulsaydı `sayi` sütunu her satırda 472 kalırdı. Koşul hep doğru olurdu ve `toplam` her turda 2 artardı: 2, 4, 6, … Tablonun ikinci satırında bu tuhaflık hemen görülür: döngü hiç bitmeyecektir. Program çalıştırılsaydı bu hata ancak ekran donduğunda fark edilirdi.

İzleme tablosu kurarken birkaç alışkanlık işe yarar. Her değişkene bir sütun ayrılır; koşul için de ayrı bir sütun açılır. Bir değişken değiştiğinde yeni değer yeni satıra yazılır, eskisi silinmez; böylece değişimin geçmişi görünür. Döngü bittiğinde değişkenlerin son değerleri ayrıca not edilir; döngüden sonraki adımlar bu değerleri kullanır.

1.4.8.3 Test durumları

Bir diyagramın tek bir girdiyle doğru çalışması, her girdiyle doğru çalışacağını göstermez. İç içe karar diyagramı 5 girdisiyle denenirse yalnızca “pozitif” kolu çalışır; öteki iki kolun doğru olup olmadığı anlaşılmaz. Bu yüzden girdiler bilinçli olarak seçilir.

Tanım 1.4.11

Yol kapsama (*path coverage*), sınamada her karar kutusunun her çıkışının en az bir kez çalıştırılmasıdır. **Sınır değerleri**, koşulun sonucunun değiştiği noktanın hemen iki yanındaki değerlerdir.

Tablo 1.4.17 pozitif/negatif/sıfır diyagramı için seçilmiş test durumlarıdır. Her girdinin yanında neden seçildiği ve beklenen çıktı yazılıdır.

Girdi	Neden seçildi?	Beklenen
5	1. kararın Evet yolu	pozitif
−2	2. kararın Evet yolu	negatif

Girdi	Neden seçildi?	Beklenen
0	iki kararın Hayır yolu	sıfır
1 ve −1	sınırın hemen iki yanı	pozitif, negatif

Tablo 1.4.17: Pozitif/negatif/sıfır diyagramının test durumları. Bütün beklenen çıktılar Java programıyla doğrulandı.

İlk üç girdi yol kapsamayı sağlar: üç Yaz kutusunun her biri bir kez çalışır. Son satırdaki 1 ve −1 sınır değerleridir. Koşul `sayi > 0` yerine yanlışlıkla `sayi >= 0` yazılmış olsaydı 5 ve −2 yine doğru sonuç verirdi; hata ancak 0 girildiğinde "pozitif" yazılmasıyla ortaya çıkardı. Bir eksik/bir fazla hataları tam olarak sınır değerlerinde görünür. `not >= 50` gibi bir koşul için sınır değerleri 49 ve 50'dir.

Döngüler için ayrı bir kural vardır: döngü hiç dönmeyen, bir kez dönen ve çok kez dönen girdilerle denenir. Tablo 1.4.18 1'den n'ye toplam diyagramı için bu durumları verir.

n	Neden seçildi?	Beklenen
0	gövde hiç çalışmaz	Toplam: 0
1	tek tur	Toplam: 1
4	olağan durum	Toplam: 10
−3	geçersiz girdi	Toplam: 0 (istenen bu mu?)

Tablo 1.4.18: 1'den n'ye toplam diyagramının test durumları (Java ile çalıştırıldı).

n = 0 için `1 <= 0` yanlıştır ve gövde hiç çalışmaz; program `Toplam: 0` yazar. Bu doğru bir sonuçtur: hiç sayı toplanmamıştır. n = −3 için de aynı şey olur. Ama burada sonuç tartışmalıdır. Negatif bir n anlamsız bir girdidir; programın 0 yazması yerine bir uyarı vermesi daha doğru olabilir. Test durumu bu soruyu açığa çıkarır; cevabı ise problemin tanımı verir.

Sınamanın son kuralı, beklenen çıktının denemeden *önce* elle bulunmasıdır. Önce n = 4 için sonucun 10 olması gerektiği hesaplanır, sonra program çalıştırılır ve gerçek çıktıyla karşılaştırılır. Program çalıştırıldıktan sonra çıktıya bakıp "doğru görünüyor" demek, hatayı gözden kaçırmanın en kolay yoludur. Fark çıkarsa hata ya diyagramda, ya kodda ya da beklentinin kendisindedir.

1.4.8.4 Algoritmayı göstermenin yolları

Akış diyagramı bir algoritmayı göstermenin tek yolu değildir. Tablo 1.4.19 dört gösterimi tek/çift örneği üzerinden karşılaştırır.

Gösterim	Tek/çift örneği	Güçlü yanı	Zayıf yanı
Doğal dil	"Sayıyı oku; 2'ye bölümünden kalan 0 ise çift, değilse tek yaz."	Herkes okuyabilir	Belirsiz olabilir
Akış diyagramı	Oku → İşlem → Karar → Yaz çift / Yaz tek	Akışı görsel gösterir; dallanma ve döngü kolay görülür	Büyük problemlerde hantal

Gösterim	Tek/çift örneği	Güçlü yanı	Zayıf yanı
Sözde kod	`OKU sayı; EĞER sayı mod 2 = 0 İSE YAZ "çift" DEĞİLSE YAZ "tek"`	Hızlı yazılır, koda yakın	Standart bir söz dizimi yok
Java	`if (sayi % 2 == 0) … else …`	Bilgisayar çalıştırabilir	Söz dizimi kurallarına uymak gerekir

Tablo 1.4.19: Bir algoritmayı göstermenin dört yolu.

Doğal dil en kolay okunan gösterimdir, ama belirsizdir. “Sayı büyükse” gibi bir ifade neyden büyük olduğunu söylemeyebilir. Sözde kod doğal dille programlama dili arasında durur. Anahtar sözcükleri (OKU, EĞER, İSE, YAZ) programlama diline benzer, ama derleyicisi yoktur ve herkes kendi biçimini kullanır. Java en kesin gösterimdir; bilgisayar onu çalıştırabilir. Buna karşılık noktalı virgülden süslü paranteze kadar bütün kurallara uymak gerekir. Akış diyagramı bu yelpazede görselliğiyle öne çıkar: dallanmalar ve döngüler okların biçiminden hemen anlaşılır.

Yazılım mühendisliğinde akış diyagramının bugünkü karşılığı UML (*Unified Modeling Language*, Birleşik Modelleme Dili) **etkinlik diyagramıdır** (*activity diagram*). Fikir aynıdır, semboller biraz farklıdır. Başlangıç dolu bir daireyle, karar yine bir eşkenar dörtgenle, etkinlikler yuvarlatılmış kutularla çizilir. Nesne yönelimli programların tasarımında etkinlik diyagramları UML'nin öteki diyagramlarıyla birlikte kullanılır.

1.4.8.5 Çizim araçları

Akış diyagramı kâğıt ve kalemle çizilebileceği gibi bilgisayarda da çizilebilir. Tablo 1.4.20 yaygın araçları özetler.

Araç	Ne yapar?	Not
Kâğıt ve kalem	En hızlı taslak; sınavda tek seçenek	Düzeltmesi zahmetli
draw.io (diagrams.net)	Tarayıcıda çalışan ücretsiz çizim aracı; hazır akış diyagramı sembolleri	Çizer, çalıştırmaz
Microsoft Visio	Ticari diyagram programı; standart sembol kütüphaneleri	Ücretli
Flowgorithm	Diyagramı çalıştırır, değişkenleri izler; Java dâhil birçok dile kod üretir	Ücretsiz, eğitim amaçlı
Mermaid	Metinden diyagram çizer; GitHub ve birçok Markdown aracı doğrudan gösterir	Kodla birlikte saklanır

Tablo 1.4.20: Akış diyagramı çizim araçları.

Mermaid ilginç bir yaklaşım izler: diyagram çizilmez, yazılır. Tek/çift diyagramının Mermaid metni şöyledir:

```
flowchart TD
  A([Başla]) --> B[/Oku sayı/]
  B --> C{sayı mod 2 = 0 ?}
  C -- Evet --> D[Yaz çift]
  C -- Hayır --> E[Yaz tek]
  D --> F([Bitir])
  E --> F
```

İlk satırdaki `flowchart TD` bir akış diyagramı başlatır; `TD` (*top-down*) yukarıdan aşağıya yerleşim demektir. Her düğümün bir harf adı ve köşeli parantezler içinde bir yazısı vardır. Parantezlerin biçimi sembolü belirler: `([ ])` Başla/Bitir ovali, `[/ /]` giriş paralelkenarı, `{ }` karar eşkenar dörtgeni, `[ ]` işlem dikdörtgenidir. `-->` bir akış çizgisi, `-- Evet -->` etiketli bir karar çıkışıdır. Kutuların sayfadaki yerini araç kendisi belirler. Metin biçimi, diyagramın program koduyla aynı dosyalarda saklanmasını ve kodla birlikte güncellenmesini kolaylaştırır.

Çizim aracı ne olursa olsun kurallar aynıdır: tek Başla, etiketli karar çıkışları, kesişmeyen oklar. Araç yalnızca çizimi kolaylaştırır; diyagramın doğruluğu yine onu tasarlayan kişiye bağlıdır.

1.4.9 Sık yapılan hatalar

Akış diyagramı çizerken ve diyagramı Java'ya çevirirken bazı hatalar çok sık görülür. Bir kısmını derleyici yakalar; bir kısmı ise derlenen ama yanlış çalışan programlara yol açar. İkinci gruptaki hatalar daha tehlikelidir, çünkü program hiçbir uyarı vermeden yanlış sonuç üretir.

Sık yapılan hata

Karar kutusunda tek çıkış. Karar sembolünden iki ok çıkmalı ve ikisi de E/H diye etiketlenmelidir. Yalnızca Evet oku çizilirse “hayır” durumunda ne olacağı belirsiz kalır. Hayır kolunda yapılacak iş yoksa bile ok çizilir ve doğrudan birleşme noktasına bağlanır.

Sık yapılan hata

Yanlış sembol. Klavyeden okuma için dikdörtgen, hesaplama için paralelkenar kullanmak diyagramı yanıltıcı yapar. Kural basittir: girdi paralelkenar, işlem dikdörtgen, soru eşkenar dörtgendir.

Sık yapılan hata

Karşılaştırmada = kullanmak. `if (sonuc = 0)` derlenmez: `error: incompatible types: int cannot be converted to boolean`. Karşılaştırma `==` ile yazılır: `if (sonuc == 0)`.

Sık yapılan hata

Başlangıç değerini unutmak. Diyagramda "sayaç = 1" kutusu yoksa Java da hata verir. `int sayac;` diye tanımlanıp değer verilmeden `while (sayac <= 10)` yazılırsa derleyici `error: variable sayac might not have been initialized` iletisini verir.

Sık yapılan hata

Güncellemeyi unutmak. "sayaç += 1" kutusu yoksa sayaç hiç değişmez ve koşul hep doğru kalır: sonsuz döngü. Derleyici bu hatayı yakalamaz; program ekrana durmadan 1 yazar.

Sık yapılan hata

Sınır hatası. "1'den 10'a kadar" için `sayac < 10` yazmak 10'u atlar. Koşul son değerle sınanmalıdır: sayaç 10 iken koşul doğru olmalıdır.

Sık yapılan hata

Döngü başlığına noktalı virgül. `while (sayac <= 10);` satırı derlenir, ama sondaki noktalı virgül boş bir deyimdir ve döngünün gövdesi olur. Altındaki süslü parantezli blok döngünün parçası değildir. Sayaç hiç artmadığı için koşul hep doğru kalır ve program hiçbir şey yazmadan takılır. JDK 21 bu kodu uyarısız derler.

Sık yapılan hata

Geri oku yanlış yere bağlamak. 1'den 10'a kadar yazan diyagramda geri ok karar kutusuna değil, "sayaç = 1" kutusunun üstüne dönerse sayaç her turda yeniden 1 olur. Koşul hep doğru kalır ve ekrana durmadan 1 yazılır: yine sonsuz döngü. Geri ok, başlangıç değerinden sonraya, karar kutusuna dönmelidir. Java'da bu hata, başlangıç atamasının döngünün içine yazılmasına karşılık gelir.

Sık yapılan hata

Tam sayı bölmesi. Diyagramdaki "toplam / 5" ifadesi Java'da iki `int` arasında yazılırsa ondalık kısım atılır: 283 / 5 = 56. Ondalıklı sonuç için bölenlerden biri `double` olmalıdır: `toplam / 5.0` ya da `(double) toplam / adet`.

1.4.10 Özet

- **Algoritma** bir problemi çözen sıralı ve sonlu adımlardır. **Akış diyagramı** algoritmanın semboller ve oklarla çizimidir; programlama dilinden bağımsız bir plandır. Büyük programlarda hantallaşır; o zaman sözde kod daha pratiktir.
- Temel semboller: oval Başla/Bitir, paralelkenar giriş, dikdörtgen işlem, eşkenar dörtgen karar, altıgen döngü, dalgalı kutu yazdır, çember bağlantı noktası, ok akış çizgisi. Ek semboller: alt program, sayfa dışı bağlantı, veri deposu, elle giriş, açıklama, hazırlık.
- Çizim kuralları: tek Başla, en az bir Bitir; yukarıdan aşağıya ve soldan sağa akış; yalnızca karar kutusunun iki çıkışı vardır ve çıkışlar E/H diye etiketlenir; çizgiler kesişmez.
- Java karşılıkları: Başla/Bitir → `main`; giriş → `nextInt()`; işlem → atama; karar → `if` / `else`; karar + geri ok → `while`; altıgen → `for`; yazdır → `println`; alt program → metot. Diyagramdaki "=" işlem kutusunda atama (`=`), karar kutusunda karşılaştırmadır (`==`).
- **Böhm–Jacopini teoremi:** her algoritma sıra, seçim ve tekrar yapılarıyla yazılabilir. **Yapısal diyagramda** her yapının tek girişi ve tek çıkışı vardır. Java'da `goto` kullanılamaz; `break` ve `continue` yalnızca içinde bulundukları döngüye etki eden sınırlı atlamalardır.
- **Sıralı akışta** her adım tam bir kez çalışır; birbirine bağlı adımların sırası değiştirilemez. Java, değer almamış bir değişkenin okunmasını derleme sırasında yakalar.
- **Koşullu akışta** karar kutusu K'ye göre P ya da Q çalışır. Üç ve daha çok ihtimal iç içe kararlarla (`else if`) çizilir. Java'da `%` işleminin sonucu bölünenin işaretini alır: `-3 % 2 == -1`. VE `&&`, VEYA `||` ile yazılır ve kısa devre değerlendirilir.
- **Tekrarlı akışta** geri ok karar kutusuna döner. Döngünün üç parçası başlangıç, koşul ve güncellemedir. Karar başta ise koşul gövdeden bir fazla sınanır (1'den 10'a: gövde 10, koşul 11 kez).
- Döngü kalıpları: **sayaç** koşul sağlanınca 1 artar; **biriktirici** her turda değer toplar (toplama için 0, çarpma için 1 ile başlar); **bayrak** bir olayın olup olmadığını tutar. **Bekçi değerli döngüde** önce okuma yapılır; bekçi toplama katılmaz; döngüden sonra adet = 0 durumu ayrıca denetlenir.
- **Erken çıkış** `break` ile yazılır; yapısal karşılığı bir bayrakla döngü koşulunu genişletmektir. **İç içe döngüde** iç gövde m × n kez çalışır.
- **Alt program** ayrı çizilen ve çağrılan adım dizisidir; parametre alır, Dön kutusuyla değer döndürür. Java'da metottur; parametreler kopyalanır, yerel değişkenler çağrı bitince yok olur.
- **Masa başı kontrolde** diyagram ya da kod bir izleme tablosuyla kâğıt üzerinde yürütülür. **Test durumları** her kararın her çıkışını (yol kapsama) ve sınır değerlerini içerir. Döngü hiç dönmeyen, bir kez ve çok kez dönen girdilerle denenir; beklenen çıktı denemeden önce elle bulunur.
- Tuzaklar: < / ≤ sınır hatası; güncellemeyi unutmak ya da geri oku başlangıç kutusuna bağlamak (sonsuz döngü); döngü başlığına noktalı virgül; tam sayı bölmesi.

1.4.11 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

algoritma *algorithm* · akış diyagramı *flowchart* · sözde kod *pseudocode* · sıralı akış *sequential flow* · koşullu akış *conditional flow* · yapısal diyagram *structured flowchart* · iç içe karar *nested decision* · karşılaştırma işleci *comparison operator* · mantıksal işleç *logical operator* · kısa devre değerlendirme *short-circuit evaluation* · döngü *loop* · while döngüsü *while loop* · for döngüsü *for loop* · sonsuz döngü *infinite loop* · biriktirici *accumulator* · bekçi değer *sentinel value* · break / continue *break / continue* · iç içe döngü *nested loop* · alt program *subprogram (predefined process)* · masa başı kontrol *desk check* · izleme tablosu *trace table* · yol kapsama *path coverage*

1.4.12 Kendini Dene

1. “Sonuçları dosyaya kaydet”, “toplamı ekrana yaz” ve “i = 1; i ≤ 5; i++” adımları hangi sembollerle çizilir?

   Cevap

   Dosyaya kaydetme kalıcı bir yere yazmaktır: veri deposu (silindir). Ekrana yazma Yazdır sembolüyle (dalgalı alt kenarlı kutu) çizilir; bazı kaynaklar çıkış paralelkenarı da kullanır. Başlangıç, koşul ve güncellemeyi bir arada içeren ifade döngü sembolüdür (altıgen); Java karşılığı `for (int i = 1; i <= 5; i++)` 'dır.

2. `int x = 50, adim = 0; while (x > 1) { x = x / 2; adim++; }` parçası çalıştıktan sonra `x` ve `adim` kaçtır?

   Cevap

   Tam sayı bölmesiyle x şu değerleri alır: 50 → 25 → 12 → 6 → 3 → 1. Her bölmede `adim` 1 artar; 5 bölme yapılır. x = 1 olunca `1 > 1` yanlıştır ve döngü biter. Sonuç: `x = 1`, `adim = 5`. 25 / 2 ve 3 / 2 bölmelerinde ondalık kısım atılmıştır.

3. `for (int i = 3; i <= 15; i += 4) toplam += i;` döngüsü kaç kez döner? `toplam` 0'dan başlı-

yorsa sonunda kaç olur?

Cevap

`i` 3, 7, 11 ve 15 değerlerini alır; 15 de koşulu sağlar çünkü koşul `<=` ile yazılmıştır. Bir sonraki değer 19'dur ve `19 <= 15` yanlıştır. Döngü 4 kez döner; `toplam = 3 + 7 + 11 + 15 = 36`. Koşul 5 kez sınanır.

4. Bekçi değeri 0 olan ortalama programına kullanıcı sırasıyla 8, 0, 5, 0 giriyor. Program ne yazar? 5 değeri ne olur?

Cevap

İlk okunan 8'dir; `8 != 0` doğrudur, `toplam = 8`, `adet = 1` olur. Gövdenin sonunda 0 okunur; 0 bekçidir ve döngü biter. `adet > 0` doğru olduğu için `Ortalama: 8.0` yazılır. 5 ve ikinci 0 hiç okunmaz; program onları beklemeden biter. Bekçi, girdinin nerede bittiğini kesin olarak belirler.

5. İlk negatif sayının sırasını bulan program 3, −1, −4, 2, 7 girdisiyle çalışıyor. `break`'li sürüm ve bayraklı sürüm ne yazar? Döngü bittiğinde iki sürümde `i` kaçtır?

Cevap

İkinci sayı −1 negatiftir; iki sürüm de `Sıra: 2` yazar. −4 hiç dikkate alınmaz, çünkü aranan *ilk* negatif sayıdır. `break`'li sürümde döngü `i = 2` iken biter. Bayraklı sürümde ikinci turun sonunda `i++` çalışır; `i = 3` olur ve `3 <= 5 && !true` yanlış çıkar. (Sorudaki karşılaştırma için `i`'nin döngüden önce tanımlandığı varsayılır.)

6. Dış döngü `a` = 1'den 3'e kadar dönüyor. İç döngü `b` = 1'den `a`'ya kadar dönüyor ve gövdesi bir `*` yazıyor; iç döngüden sonra `println()` çalışıyor. Ekranda ne görülür? Kaç yıldız yazılır?

Cevap

İç döngünün tur sayısı dış değişkene bağlıdır: a = 1 için 1, a = 2 için 2, a = 3 için 3 tur. Ekranda üç satır görülür: `*`, `**`, `***`. Toplam 1 + 2 + 3 = 6 yıldız yazılır. Burada iç gövdenin çalışma sayısı m × n değil, tur sayılarının toplamıdır.

7. "1'den 100'e kadar 5'in katlarını yaz" işi için `for (int i = 5; i < 100; i += 5)` yazılmış. Kaç sayı yazılır? Hata nedir?

Cevap

5, 10, …, 95 yazılır: 19 sayı. i = 100 olduğunda `100 < 100` yanlıştır ve 100 yazılmaz. Bu bir eksik hatasıdır. Koşul son değerle sınanırsa hata görülür: 100 de katlardan biridir, o hâlde koşul `i <= 100` olmalıdır. Düzeltilmiş döngü 20 sayı yazar.

8. Tek/çift programında koşul `sayi % 2 == 1` diye yazılmış. −7 girildiğinde ne yazılır? Koşul `sayi % 2 == 0` olsaydı ne yazılırdı?

   Cevap

   Java'da kalan bölünenin işaretini alır: `-7 % 2 == -1`. Birinci koşulda `-1 == 1` yanlıştır ve −7 "çift" sanılır; bu yanlıştır. İkinci koşulda `-1 == 0` yanlıştır, Hayır kolundan "tek" yazılır; bu doğrudur. Çiftliği sınamak negatif sayılarda da güvenlidir.

9. "Not ≥ 50 ise 'geçti', değilse 'kaldı' yaz" diyagramını sınamak için en az hangi girdiler seçilmelidir? Neden?

   Cevap

   Yol kapsama için her iki kol da çalışmalıdır: biri ≥ 50, biri < 50 olan iki not. Sınır değerleri için koşulun sonucunun değiştiği noktanın iki yanı seçilir: 49 (kaldı) ve 50 (geçti). Bu iki girdi aynı zamanda iki kolu da kapsar. Geçerli aralığın uçları (0 ve 100) da eklenebilir. Koşul yanlışlıkla `> 50` yazılmışsa hatayı yalnızca 50 girdisi ortaya çıkarır.

1.4.13 İleri okuma

- P. Deitel ve H. Deitel, *Java: How to Program* (11. baskı, 2017), Bölüm 4 ve 5: Java'nın kontrol deyimlerini UML etkinlik diyagramlarıyla birlikte anlatır; sayaç, bekçi değer ve iç içe kontrol yapılarını çok sayıda örnekle işler.
- A. B. Downey ve C. Mayfield, *Think Java* (2. baskı, 2019): koşullar, döngüler ve metotlar için kısa, sade bir giriş sunar; kitabın tamamı yazarların sitesinde ücretsiz okunabilir.
- C. Böhm ve G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules" (1966): yapısal programlama teoreminin özgün makalesidir; akış diyagramlarının üç yapıya indirgenebileceğini kanıtlar.
- Oracle, *The Java Tutorials*, "Control Flow Statements": `if`, `switch`, `while`, `do-while`, `for`, `break` ve `continue` deyimlerinin resmî açıklaması ve kısa örnekleri.

1.4.14 Kaynaklar

1. ISO 5807:1985, *Information processing — Documentation symbols and conventions for data, program and system flowcharts, program network charts and system resources charts*.
2. C. Böhm, G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", *Communications of the ACM* 9(5), 1966.
3. P. Deitel, H. Deitel, *Java: How to Program*, 11. baskı, Pearson, 2017.
4. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019.

5. Oracle, *The Java Tutorials*: “Control Flow Statements” (docs.oracle.com/javase/tutorial).

6. K. Sierra, B. Bates, T. Gee, *Head First Java*, 3. baskı, O'Reilly, 2022.

KISIM I · BÖLÜM 2

Programlama ve Java Söz Dizimi

Bir programlama dili, belirli söz dizimi ve anlam kurallarına uyan ifadelerden oluşur. Bölümde programlama dillerinin gelişimi, söz dizimi ile anlam arasındaki fark, ifadeler, değişkenler ve tipler genel bir çerçevede tanıtılır. Java tarafında kaynak kodun derlenip çalıştırılması, ilkel veri tipleri, tanımlayıcılar ve anahtar sözcükler, atama, sabitler ve tip dönüşümleri ile işleçler ayrıntılı olarak işlenir. Kontrol akışına kısa bir girişle bölüm tamamlanır.

Bu bölümün kodu: `github.com/sercankulcu/object-oriented-programming-java/Bolum02/src`

BU BÖLÜMÜN NOTLARI

2.1 Programlama

2.2 Söz Dizimi Kuralları

KISIM I · BÖLÜM 2.1

Programlama

Bir algoritma kâğıt üzerinde ne kadar açık yazılmış olursa olsun, bilgisayar onu ancak bir programlama diliyle yazıldığında çalıştırabilir. Programlama dili, insanın düşündüğü çözüm ile makinenin yaptığı işlem arasındaki köprüdür. Bu not o köprünün nasıl kurulduğunu anlatır. İlk kesimler dilleri soyutlama düzeyine, uygulama alanına, yürütme biçimine ve paradigmaya göre sınıflandırır. Java'nın kaynak kodu nasıl bytecode'a derlediği ve sanal makinede nasıl çalıştırdığı da burada yer alır. Ardından her dilin üç katmanı ele alınır: söz dizimi, statik anlam ve anlam. Bu katmanlar derleme, çalışma zamanı ve mantık hatalarını ayırmanın da temelidir. Sonraki kesimler ifadelerin işleç önceliğiyle nasıl değerlendirildiğini, tam sayı bölmesini, statik ve dinamik tiplemeyi, tipler arası işlemleri ve dizgi birleştirmeyi işler. Son kesim bir algoritmanın Java programına dönüşmesini, ifade ile deyim ayrımını ve deyim sırasının sonucu nasıl belirlediğini gösterir.

ÖĞRENME ÇIKTILARI

Programlama dilleri
Programlama dilini tanımlamak; dilleri soyutlama düzeyi, uygulama alanı, yürütme biçimi ve paradigmaya göre sınıflandırmak.

Java'nın çalışması
Kaynak kodun `javac` ile bytecode'a derlenip JVM'de çalıştırılmasını açıklamak.

Söz dizimi ve anlam
Söz dizimi, statik anlam ve anlam kavramlarını ayırmak; derleme, çalışma zamanı ve mantık hatalarını tanımak.

İfadeler
İşleç önceliği, soldan sağa değerlendirme ve tam sayı bölmesiyle bir ifadenin değerini bulmak.

Değişkenler ve tipler
Değişken tanımlamak, atamayı iki adımda izlemek; statik ve dinamik tiplemeyi, tipler arası işlemleri açıklamak.

Algoritmadan programa
İfade ile deyimi ayırt etmek; bir algoritmayı Java programına çevirmek ve deyim sırasının önemini göstermek.

2.1.1 Programlama dilleri

İnsanlar birbirleriyle doğal dillerle anlaşır. Türkçe, İngilizce ya da Almanca zengin dillerdir; aynı düşünce pek çok biçimde söylenebilir ve dinleyen kişi eksik kalan yerleri bağlamdan tamamlar. Bilgisayar böyle bir tamamlama yapmaz. Ona verilen her talimat tek bir biçimde okunmalı ve tek bir anlama gelmelidir. Programlama dilleri bu ihtiyaçtan doğmuştur. Bu kesim programlama dilini tanımlar ve dilleri dört ölçüte göre sınıflandırır: soyutlama düzeyi, uygulama alanı, yürütme biçimi ve paradigma. Paradigmalar ayrı bir kesimde işlenir.

2.1.1.1 Programlama dili nedir?

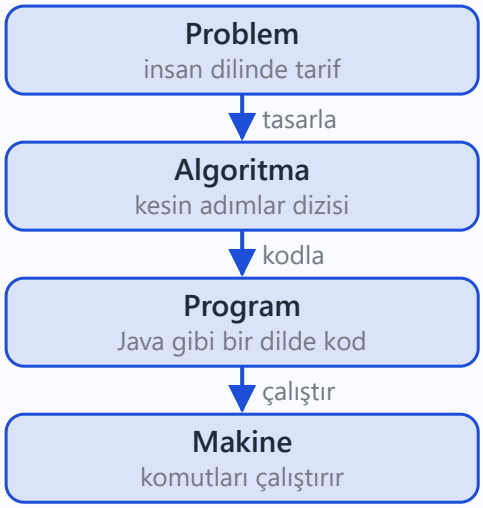
Tanım 2.1.1

Programlama dili (*programming language*), bir makinenin yapabileceği hesaplamaları ifade etmek için tasarlanmış yapay bir dildir. Bu dille yazılan metne **program**, programın yazıldığı hâline de **kaynak kod** (*source code*) denir.

Tanımdaki “yapay” sözcüğü önemlidir. Doğal diller yüzyıllar içinde kendiliğinden gelişir; kuralları sonradan, gözlemle yazılır. Programlama dilleri ise baştan tasarlanır. Hangi sözcüklerin kullanılacağı, bu sözcüklerin hangi sırayla dizileceği ve her dizilişin ne anlama geldiği bir belgeyle kesin olarak belirlenir. Java için bu belge *Java Dil Belirtimi*'dir (*The Java Language Specification*).

Programlama dili iki amaçla kullanılır. Birincisi, makinenin davranışını denetleyen programlar yazmaktır. Bunun için algoritmaların açık ve kesin biçimde ifade edilmesi gerekir. İkincisi daha az fark edilir: program, programcılar arasında bir iletişim aracıdır. Yazılan kodu yıllar sonra başka biri okuyacak, değiştirecek ya da hatasını arayacaktır. İyi bir program yalnız doğru çalışmaz, aynı zamanda kolay okunur.

Bilgisayarlar, telefonlar ve gömülü cihazlar işlevlerini programlardan alır. Bir çamaşır makinesinin yıkama adımları, bir arabadaki fren denetim sisteminin kararları ya da bir telefonun ekranındaki her dokunuşa verilen yanıt, bir programlama diliyle yazılmış talimatlardır. Programlama dili bu açıdan algoritmaların ve mantıksal süreçlerin yazılı ifadesidir.



Şekil 2.1.1: Problemden makineye giden yol. İnsan dilindeki tarif bir algoritmaya, algoritma bir programa dönüşür; program da makinede çalışır.

Şekil 2.1.1 bu yolun dört durağını gösterir. Her geçişin kendi zorluğu vardır. Problemden algoritmaya geçişte belirsizlikler giderilir ve çözüm kesin adımlara bölünür. Algoritmadan programa geçişte bu adımlar bir dilin kurallarına göre yazılır. Programdan makineye geçişi ise derleyici ya da yorumlayıcı denen yardımcı programlar üstlenir. Bu notun büyük bölümü ikinci geçişle ilgilidir: bir algoritma Java'da nasıl doğru yazılır ve yazılan kod gerçekte ne anlama gelir?

Örnek dil sayısı çok fazladır: C, C++, Java, Python, Prolog, Haskell, Scala ve yüzlercesi. Hiçbiri her iş için en iyisi değildir. Dil seçimi projenin ihtiyaçlarına göre yapılır. Hız, güvenilirlik, geliştirme süresi, çalışılacak donanım ve ekibin deneyimi bu seçimi etkiler. Sonraki alt kesimler bu seçimde işe yarayan sınıflandırmaları verir.

2.1.1.2 Soyutlama düzeyleri

İşlemci yalnız kendi komut kümesini anlar. Bu komutlar ikili sayılarla kodlanmıştır. Her biri çok küçük bir iş yapar: bir yazmaçtan ötekine değer taşır, iki yazmacı toplar ya da bellekten bir sözcük okur. İşlemcinin doğrudan çalıştırdığı bu biçime **makine dili** (*machine language*) ya da **makine kodu** denir. Makine dili bir işlemci ailesine özgüdür; bir Intel işlemcinin makine kodu bir ARM işlemcide çalışmaz.

Tanım 2.1.2

Düşük düzeyli dil (*low-level language*), donanıma yakın dildir. Bu dilde yazılan her satır işlemcinin bir ya da birkaç komutuna karşılık gelir. Makine dili ve **assembly dili** (*assembly language*) bu gruptadır. **Yüksek düzeyli dil** (*high-level language*) ise donanımın ayrıntısını gizler; programcı değişkenler, ifadeler, döngüler ve metotlarla düşünür. Java, Python ve C# yüksek düzeyli dillerdir.

Assembly dili makine komutlarına okunabilir kısaltmalar verir. `ADD`, `MOV`, `LOAD` gibi adlar ikili kodların yerini alır. Yine de her satır hâlâ tek bir makine komutuna karşılık gelir. `a = b + c` gibi basit bir işlem assembly'de kabaca üç satırdır: `b`'yi bir yazmaca yükle, `c`'yi ekle, sonucu `a`'nın bellek adresine yaz. Java'da ise aynı iş tek bir satırdır ve hangi yazmacın kullanılacağı programcıyı ilgilendirmez.

Soyutlama düzeyi yükseldikçe program kısalır, okunur hâle gelir ve başka donanımlara taşınması kolaylaşır. Buna karşılık programcı donanım üzerindeki denetimin bir kısmını bırakır. Elektrik-elektronik mühendisliğinde iki ucun da yeri vardır. Bir mikrodenetleyicinin kesme yordamı ya da çok sıkı zaman kısıtı olan bir sürücü bazen assembly ya da C ile yazılır. Ölçüm verisini işleyen, arayüzü olan ya da ağda haberleşen bir uygulama ise çoğunlukla Java ya da Python ile yazılır.

Düzey	Örnek	Bir satır neye karşılık gelir?	Taşınabilirlik
Makine dili	ikili komut kodları	tek bir işlemci komutu	yalnız o işlemci ailesi
Assembly	`ADD`, `MOV`	tek bir işlemci komutu (okunur adla)	yalnız o işlemci ailesi
Yüksek düzey	Java, Python, C#	çok sayıda işlemci komutu	derleyici ya da sanal makine olan her yer

Tablo 2.1.1: Soyutlama düzeylerinin karşılaştırması.

C dili bu sınıflandırmada ara bir yerde durur. Yüksek düzeyli bir dildir ama belleğe adresle erişmeye ve donanıma yakın kod yazmaya izin verir. Bu yüzden gömülü sistemlerde çok yaygındır. Java ise programcıyı bellek adreslerinden tümüyle uzak tutar; bellek yönetimini de çöp toplayıcı denen bir bileşene bırakır.

2.1.1.3 Genel amaçlı ve hedefe odaklı diller

Dilleri sınıflandırmanın ikinci yolu uygulama alanına bakmaktır.

> **Tanım 2.1.3**
>
> **Genel amaçlı dil** (*general-purpose language*), çok çeşitli uygulamaları destekleyen geniş bir temel komut kümesi sunar. **Hedefe odaklı dil** (*domain-specific language*) ise belirli bir uygulama türüne odaklanır ve o türdeki işlemleri kolaylaştırır.

Java, C, C++ ve Python genel amaçlı dillerdir. Aynı dille bir masaüstü programı, bir web sunucusu, bir mobil uygulama ya da bir gömülü sistem yazılabilir. Hedefe odaklı dillerin en tanıdık iki örneği MATLAB ve SQL'dir. MATLAB sayısal hesaplama için tasarlanmıştır; matris ve vektör işlemleri dile gömülüdür ve hızlı çalışacak biçimde eniyilenmiştir. SQL (*Structured Query Language*, yapılandırılmış sorgu dili) veri tabanından veri istemek ve veri tabanını güncellemek içindir.

Farkı en iyi aynı iş gösterir. İki $n \times n$ matrisin çarpımı MATLAB'de tek satırdır:

```
C = A * B;   % tek satır
```

Java'da aynı iş elle yazılan üç iç içe döngüyle yapılır:

```
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
        for (int k = 0; k < n; k++)
            C[i][j] += A[i][k] * B[k][j];
```

Java kodunda en içteki satır $C_{ij} = \Sigma_k A_{ik} \cdot B_{kj}$ toplamını adım adım kurar. Dış iki döngü sonuç matrisinin her hücresi üzerinde gezinir, en içteki döngü o hücre için çarpımları toplar. `C[i][j] += …` yazımı "`C[i][j]`'ye ekle" anlamına gelir; `C` dizisinin başlangıçta sıfırlarla dolu olduğu varsayılır. Java'da yeni oluşturulan bir `double` dizisinin öğeleri zaten 0.0'dır.

İki yazımın maliyeti de farklıdır. Üç iç içe döngü her biri n tur döndüğü için toplam n^3 çarpma ve toplama yapar. MATLAB'deki `A * B` da sonuçta benzer sayıda işlem yapar, ama bu işlemleri önbelleği iyi kullanan, çoğu zaman paralel çalışan hazır bir kitaplığa bırakır. Hedefe odaklı dil bu yüzden hem daha kısa hem de o alanda çoğu zaman daha hızlıdır. Bedeli esnekliktir: MATLAB ile bir web sunucusu yazmak, Java ile yazmaktan çok daha zordur.

Sık yapılan hata

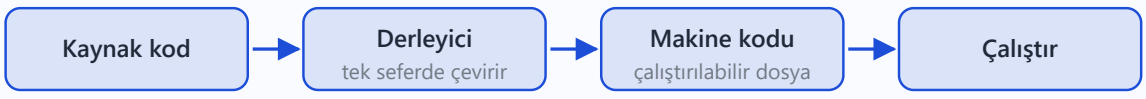
MATLAB alışkanlığıyla Java'da `C = A * B;` yazmak. Java'da `*` işleci yalnız sayılar için tanımlıdır; iki dizinin çarpımını bilmez. Derleyici `bad operand types for binary operator '*'` iletisiyle satırı reddeder. Java'da matris çarpımı ya döngülerle ya da bir kitaplık metoduyla yapılır.

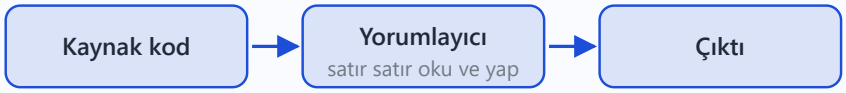
2.1.1.4 Derlenen ve yorumlanan diller

Üçüncü sınıflandırma, kaynak kodun nasıl çalıştırıldığına bakar. İşlemci kaynak kodu doğrudan anlamaz; kodun önce makine diline çevrilmesi gerekir. Bu çeviri iki temel yolla yapılır.

Tanım 2.1.4

Derleyici (*compiler*), kaynak kodun tamamını program çalıştırılmadan önce başka bir dile, çoğunlukla makine koduna çeviren programdır. **Yorumlayıcı** (*interpreter*) ise kaynak kodu okur ve komutları sırayla, o anda çalıştırır; önceden bütün bir çeviri üretmez. Bu iki yola göre diller **derlenen** (*compiled*) ve **yorumlanan** (*interpreted*) diye ayrılır.





Şekil 2.1.2: Yürütme biçimleri. Üstte derlenen dil (C, C++, Go, Rust): kaynak kod derleyiciyle tek seferde makine koduna çevrilir, oluşan çalıştırılabilir dosya sonra çalıştırılır. Altta yorumlanan dil (Python, JavaScript): yorumlayıcı kaynak kodu satır satır okur ve doğrudan çalıştırır.

Şekil 2.1.2'de iki yol yan yana durur. Derlenen bir dilde çeviri bir kez yapılır ve sonucu bir dosyaya yazılır. Windows'taki bir `.exe` dosyası böyle bir çıktıdır. Program her çalıştırıldığında yeniden çevrilmez; makine kodu doğrudan işlemciye verilir. Derleyici bütün programı birden gördüğü için kodu eniyileyebilir: gereksiz işlemleri atar, sık kullanılan değerleri yazmaçlarda tutar. Bu yüzden derlenen programlar genellikle hızlıdır.

Yorumlanan bir dilde ayrı bir çeviri dosyası yoktur. Yorumlayıcı programın yanında sürekli çalışır, her komutu sırası geldiğinde çözümler ve yerine getirir. Bu yaklaşım esneklik sağlar: kod yazılır yazılmaz denenebilir, program çalışırken bile yeni komutlar verilebilir. Bedeli hızdır. Aynı döngü bin kez dönüyorsa yorumlayıcı aynı satırı bin kez çözümler.

	Derlenen	**Yorumlanan**
Nasıl?	Tüm kaynak kod, çalıştırılmadan önce derleyiciyle nesne kodu (*object code*) ya da makine koduna çevrilir.	Yorumlayıcı kaynak kodu okur ve komutları sırayla o anda çalıştırır.
Hız	Hızlı ve eniyilenmiş yürütme	Genellikle daha yavaş
Hatalar	Birçoğu derleme sırasında bulunur	Çoğu ancak o satır çalışınca görülür
Dağıtım	Her işletim sistemi ve işlemci için ayrı derleme gerekir	Yorumlayıcısı olan her yerde aynı kaynak kod çalışır

Tablo 2.1.2: Derlenen ve yorumlanan dillerin karşılaştırması.

Tablonun üçüncü satırı programcı için belki de en önemli farktır. Derleyici programı çalıştırmadan önce bütünüyle okur. Yanlış yazılmış bir sözcük, eksik bir parantez ya da uyumsuz bir işlem bu okumada yakalanır ve program hiç çalışmaz. Yorumlanan bir dilde ise hatalı bir satır, programın akışı o satıra ulaşana kadar fark edilmez. Bir Python programı bir saat sorunsuz çalışıp ancak nadiren girilen bir `if` dalında yazım hatası yüzünden durabilir.

2.1.1.5 Java'nın yolu: bytecode ve JVM

Java iki yaklaşımı birleştirir. Kaynak kod önce derlenir, ama doğrudan bir işlemcinin makine koduna değil, **bytecode** denen ara bir koda. Bu ara kodu da **Java Sanal Makinesi** (*Java Virtual Machine*, JVM) çalıştırır.

Tanım 2.1.5

Bytecode (ara kod), Java derleyicisi `javac`'ın ürettiği ve `.class` dosyalarında saklanan, platformdan bağımsız komutlardır. **JVM**, bytecode'u yorumlayan ve sık çalışan kısımlarını **JIT derleyici** (*just-in-time compiler*, tam zamanında derleyici) ile çalışma sırasında makine koduna çeviren sanal makinedir.

Bir Java programı iki komutla çalıştırılır. İlki derleme, ikincisi çalıştırmadır:

```
> javac Paradigma.java      (Paradigma.class oluşur)
> java Paradigma
Buyruksal: 15
Fonksiyonel: 15
```

`javac Paradigma.java` komutu kaynak dosyayı okur. Hata yoksa aynı klasörde `Paradigma.class` dosyasını oluşturur. `java Paradigma` komutu ise JVM'i başlatır ve ona bu sınıfı yükletir. Dikkat edilirse ikinci komutta dosya uzantısı yazılmaz; JVM'e bir dosya adı değil, bir sınıf adı verilir.

Bytecode'un neye benzediği JDK ile gelen `javap -c` aracıyla görülebilir. İki tam sayıyı toplayan küçük bir metot:

```
static int topla(int a, int b) {
    return a + b;
}
```

`javac` bu metodu şu dört bytecode komutuna çevirir:

```
static int topla(int, int);
  Code:
     0: iload_0
     1: iload_1
     2: iadd
     3: ireturn
```

Komutlar bir yığın üzerinde çalışır. `iload_0` birinci parametreyi (`a`), `iload_1` ikinci parametreyi (`b`) yığına koyar. `iadd` yığının üstündeki iki tam sayıyı alır, toplar ve sonucu yığına geri koyar. `ireturn` yığının üstündeki değeri metodun sonucu olarak döndürür. Baştaki `i` harfi her komutun `int` ile çalıştığını gösterir. Bu komutlar hiçbir gerçek işlemciye ait değildir; JVM için tanımlanmış soyut bir komut kümesidir.

Bu ara katmanın büyük bir kazancı vardır. Aynı `.class` dosyası Windows'ta, Linux'ta ya da macOS'ta, Intel ya da ARM işlemcide değiştirilmeden çalışır. Her platform için ayrı bir JVM yazılmıştır; program ise tek bir kez derlenir. Java'nın tanıtımında kullanılan "bir kez yaz, her yerde çalıştır" (*write once, run anywhere*) sözü bu özelliği anlatır.

Hız sorununu JIT derleyici çözer. JVM programı önce yorumlayarak çalıştırır ve hangi metotların sık çağrıldığını sayar. Sık çalışan bir metot, çalışma sırasında o makinenin gerçek makine koduna derlenir; sonraki çağrılar yorumlama olmadan doğrudan işlemcide yürür. Bu yüzden uzun süre çalışan Java programları derlenen dillere yakın bir hıza ulaşır. Kısa programlarda ise JVM'in açılış süresi hissedilir.

Java'nın derleme adımı, derlenen dillerin en önemli avantajını da korur. `javac` bütün programı okur, tipleri denetler ve hatalı programın `.class` dosyasını hiç üretmez. Yanlış yazılmış bir değişken adı ya da bir sayıya bir dizginin atanması program çalışmadan yakalanır. Kotlin ve Scala gibi başka diller de JVM bytecode'una derlenir; bu diller aynı sanal makineyi ve Java'nın kitaplıklarını paylaşır.

Sık yapılan hata

Programı `java Paradigma.class` ya da sınıf adından farklı adlı bir dosyayla çalıştırmaya çalışmak. `java` komutu bir sınıf adı bekler; `.class` uzantısı yazılırsa JVM `Paradigma.class` adlı bir sınıf arar ve bulamaz. Ayrıca `public` bir sınıfın kaynak dosyası sınıfla aynı adı taşımalıdır: `public class Paradigma` yalnız `Paradigma.java` adlı dosyada derlenir.

2.1.2 Programlama paradigmaları

İki programcı aynı problemi aynı dille bile çok farklı biçimlerde çözebilir. Dillerin kendisi de belli bir çözüm biçimini kolaylaştıracak şekilde tasarlanır. Bir dilin hangi düşünme biçimini öne çıkardığı, o dilin paradigmasıyla anlatılır. Bu kesim dört temel paradigmayı tanıtır ve her birini kısa bir programla gösterir.

2.1.2.1 Paradigma kavramı

Tanım 2.1.6

Programlama paradigması (*programming paradigm*), bir problemi çözerken izlenen temel yaklaşımdır. Paradigma, programın neyden oluştuğunu (komutlar, fonksiyonlar, kurallar ya da nesneler) ve hesaplamanın nasıl ilerlediğini belirler.

Dört paradigma bu derste sık anılır. Tablo 2.1.3 bunları özetler.

Paradigma	Temel fikir	Örnek diller
Buyruksal (*imperative*)	Programcı makineye, durumunu *nasıl* değiştireceğini adım adım söyler.	FORTRAN, BASIC, Pascal, C
Fonksiyonel (*functional*)	Hesaplama, matematiksel fonksiyonların değerlendirilmesidir; durum değiştirmekten kaçınılır.	Lisp, Scheme, Haskell
Mantıksal (*logic*)	Hesaplamanın mantığı olgular ve kurallarla açıklanır; kontrol akışı ayrıntılandırılmaz.	Prolog
Nesne yönelimli (*object-oriented*)	İlişkili veri ve metotlar bir nesne altında toplanır; komutlar, üzerinde çalıştıkları durumla birlikte gruplanır.	C++, Java, C#, Python

Tablo 2.1.3: Dört temel programlama paradigması.

Tablodaki “durum” (*state*) sözcüğü paradigmaları ayırmanın anahtarıdır. Durum, programın belli bir andaki değişkenlerinin değerleridir. Buyruksal bir programda bu değerler her adımda değişir ve sonuç bu değişikliklerin toplamıdır. Fonksiyonel bir programda ise değerler bir kez hesaplanır ve değiştirilmez; sonuç, fonksiyonların birbirine bağlanmasıyla elde edilir. Mantıksal programda programcı ne adımları ne de fonksiyonları yazar; yalnız neyin doğru olduğunu söyler. Nesne yönelimli programda durum ortadan kalkmaz, nesnelerin içine yerleşir.

Birden çok paradigmayı destekleyen dillere **çok paradigmalı** diller denir. Java temelde nesne yönelimli ve buyruksaldır. Java 8'den bu yana lambda ifadeleri ve akışlar (*streams*) gibi fonksiyonel öğeler de içerir. Python ve C# da çok paradigmalıdır. Paradigma bu yüzden bir dilin değil, çoğu zaman bir programın ya da programın bir parçasının özelliğidir.

2.1.2.2 Buyruksal ve fonksiyonel: aynı problem

Problem basittir: 1'den 5'e kadar sayıların toplamı bulunsun. Aşağıdaki Java programı bu toplamı iki yolla hesaplar.

```
import java.util.stream.IntStream;

public class Paradigma {
    public static void main(String[] args) {
        // Buyruksal: adım adım durum değiştir
        int toplam = 0;
        for (int i = 1; i <= 5; i++) {
            toplam = toplam + i;
        }
        System.out.println("Buyruksal: " + toplam);

        // Fonksiyonel (Java 8+): ne istediğini söyle
        int sonuc = IntStream.rangeClosed(1, 5).sum();
        System.out.println("Fonksiyonel: " + sonuc);
    }
}
```

Program çalıştırıldığında iki yol da aynı sonucu verir:

```
Buyruksal: 15
Fonksiyonel: 15
```

Buyruksal parça, toplamın *nasıl* bulunacağını yazar. `toplam` adlı bir değişken 0 ile başlar. `for` döngüsü `i`'yi 1'den 5'e kadar sayar ve her turda `toplam = toplam + i` deyimiyle `toplam`'ı günceller. Sonuç, beş güncellemenin ardından değişkende kalan değerdir. Programcı burada makinenin yerine düşünür: hangi değişken, hangi sırayla, nasıl değişecek?

Fonksiyonel parça ise *ne* istendiğini yazar. `IntStream.rangeClosed(1, 5)` 1, 2, 3, 4 ve 5 sayılarından oluşan bir akış üretir; `closed` sözcüğü üst sınırın da dahil olduğunu belirtir. `.sum()` bu akıştaki bütün sayıları toplar. Kodda adım adım güncellenen bir değişken yoktur; `sonuc` değişkeni yalnız bir kez, son değeriyle atanır. Toplamanın hangi sırayla yapılacağı kitaplığa bırakılmıştır.

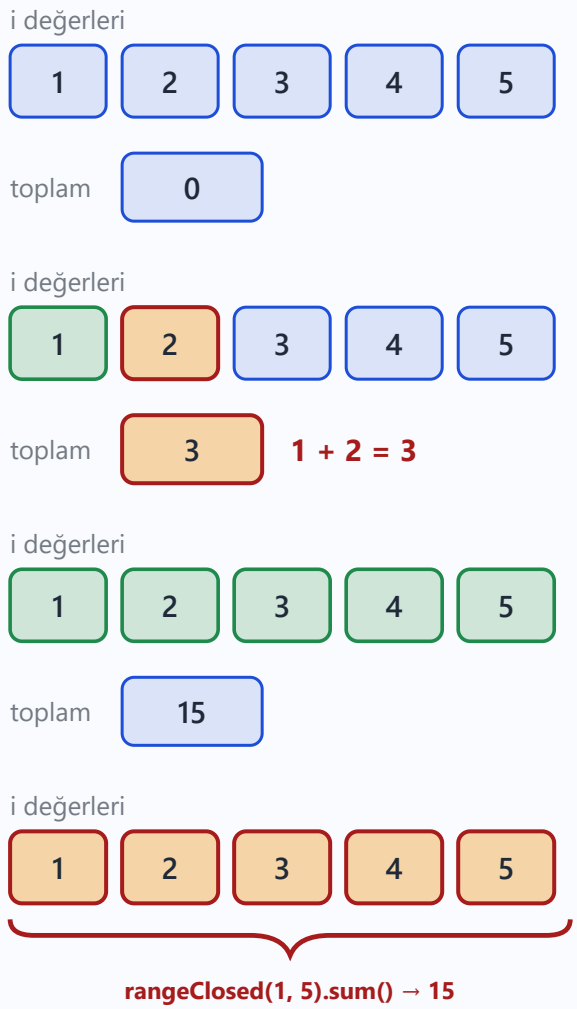
Programın ilk satırındaki `import java.util.stream.IntStream;` satırı derleyiciye `IntStream` sınıfının nerede olduğunu söyler. Bu satır olmasa derleyici `IntStream` adını tanımaz ve `cannot find symbol` hatası verir. `System` ve `String` gibi sınıflar `java.lang` paketinde olduğu için onları ayrıca içe aktarmak gerekmez.

2.1.2.3 Kod izleme: buyruksal toplam

Buyruksal yaklaşımda durumun nasıl değiştiğini görmenin en iyi yolu kodu satır satır izlemektir. Tablo 2.1.4 döngünün her turunda koşulun değerini ve `toplam`'ın yeni değerini verir.

`i`	`i <= 5`	**Gövde**	`toplam`
—	—	başlangıç	0
1	doğru	0 + 1	1
2	doğru	1 + 2	3
3	doğru	3 + 3	6
4	doğru	6 + 4	10
5	doğru	10 + 5	15
6	yanlış	döngü biter	15

Tablo 2.1.4: Buyruksal toplamın izlemesi. Her satır bir koşul denetimini ve ardından gelen gövde çalışmasını gösterir.



Şekil 2.1.3: Toplam programının izlemesinden dört an. (a) `toplam` durum değişkeni 0 ile başlar. (b) `i` = 2: 1 + 2 = 3; birinci tur yeşille işaretlenmiştir. (c) `i` 6 olunca 6 <= 5 yanlıştır ve döngü biter; beş turda biriken toplam 15'tir. (d) Fonksiyonel yol: `rangeClosed(1, 5)` beş sayıyı birlikte üretir, `sum()` hepsini toplar; hiçbir değişken adım adım güncellenmez.

İzlemede üç nokta dikkat çeker. Birincisi, koşul altı kez denetlenir ama gövde yalnız beş kez çalışır; son denetim döngüyü bitiren denetimdir. İkincisi, döngü bittiğinde `i` 6'dır, 5 değil. `i` döngünün başlığında tanımlandığı için döngüden sonra zaten kullanılamaz; yine de bu ayrıntı sınır hatalarını anlamak için önemlidir. Üçüncüsü, `toplam`'ın her ara değeri bir sonraki turun girdisidir. Bir turdaki hata sonraki bütün turlara taşınır.

İki yolun maliyeti aynıdır. Her ikisi de beş toplama yapar; *n* sayı için *n* toplama gerekir. Fark yazımdadır, iş miktarında değil. Şekil 2.1.3'ün son karesi bu farkı özetler. İki yaklaşım aynı sonucu verir. Fark, nasıl yapılacağının mı yoksa ne istendiğinin mi yazıldığındadır.

Örnek 2.1.1

3'ten 6'ya kadar sayıların toplamı iki yolla yazılsın.

Buyruksal: `int t = 0; for (int i = 3; i <= 6; i++) t = t + i;` . Turlar sırasıyla `t` 'yi 3, 7, 12 ve 18 yapar; `i` 7 olunca koşul yanlış çıkar.

Fonksiyonel: `IntStream.rangeClosed(3, 6).sum()` . Akış 3, 4, 5, 6 sayılarını üretir; toplamları 18'dir.

İki yol da 18 verir. `rangeClosed(3, 6)` yerine `range(3, 6)` yazılsaydı üst sınır dahil olmaz, sonuç 3 + 4 + 5 = 12 olurdu.

Sık yapılan hata

Toplam değişkenine ilk değer vermeyi unutmak. `int toplam;` yazıp döngüde `toplam = toplam + i;` denirse Java `variable toplam might not have been initialized` iletisiyle derlemeyi durdurur. Yerel değişkenlerin varsayılan değeri yoktur. Başka dillerde böyle bir değişken rastgele bir değerle başlayabilir ve yanlış bir toplam sessizce üretilir; Java bu hatayı derlemede yakalar.

2.1.2.4 Mantıksal programlama: Prolog

Mantıksal paradigmanın bilinen örneği Prolog'dur. Prolog programı adımlardan değil, **olgulardan** (*facts*) ve **kurallardan** (*rules*) oluşur. Programcı neyin doğru olduğunu yazar; bir soru sorulduğunda cevabı Prolog kendisi arar.

```
% olgular
ebeveyn(ali, ayse).
ebeveyn(ayse, can).
% kural
dede(X, Z) :- ebeveyn(X, Y),
              ebeveyn(Y, Z).
% sorgu
?- dede(ali, can).
true.
```

İlk iki satır iki olgu bildirir: Ali, Ayşe'nin ebeveynidir; Ayşe de Can'ın ebeveynidir. Prolog'da küçük harfle başlayan adlar sabittir, büyük harfle başlayanlar değişkendir. Kural satırı “X, Z'nin dedesidir, eğer X bir Y'nin ebeveyniyse ve Y de Z'nin ebeveyniyse” diye okunur. `:-` işareti “eğer”, virgül “ve” anlamındadır.

Sorgu `?- dede(ali, can).` “Ali, Can'ın dedesi midir?” sorusudur. Prolog kuralı kullanır ve X yerine `ali`, Z yerine `can` koyar. Ardından `ebeveyn(ali, Y)` olgusunu sağlayan bir Y arar ve `ayse`'yi bulur. Son olarak `ebeveyn(ayse, can)` olgusunun var olup olmadığına bakar. Olgu vardır, cevap `true` olur.

Bu arama sırasının hiçbiri programda yazılı değildir. Programcı döngü kurmamış, hangi olgunun önce deneneceğini söylememiştir. Mantıksal paradigmada kontrol akışı dilin kendi arama yöntemine bırakılır. Aynı program başka sorulara da cevap verir: `?- dede(ali, X).` sorgusu "Ali kimin dedesidir?" anlamına gelir ve Prolog `X = can` cevabını bulur. Buyruksal bir dilde bu iki soru için iki ayrı kod yazmak gerekirdi.

2.1.2.5 Nesne yönelimli programlama: Dikdortgen

Nesne yönelimli paradigma, bu dersin asıl konusudur. Temel fikir, birbirine ait veri ile o veri üzerinde çalışan işlemleri tek bir birimde toplamaktır. Bu birime **nesne** denir; nesnelerin kalıbı da **sınıf**tır.

> **Tanım 2.1.7**
>
> **Nesne yönelimli programlama** (*object-oriented programming*, OOP), programı durumu (veri) ve davranışı (metotlar) birleştiren nesneler üzerine kurar. Nesnenin verisini tutan değişkenlere **alan** (*field*), davranışını tanımlayan işlemlere **metot** (*method*) denir.

Aşağıdaki sınıf bir dikdörtgeni modeller. Java'da tanımlayıcılarda Türkçe karakter kullanılabilir, ama derste dosya adlarıyla sorun çıkmasın diye `Dikdortgen` yazımı seçilmiştir.

```
public class Dikdortgen {
    double en, boy;          // veri (durum)

    Dikdortgen(double en, double boy) {
        this.en = en;
        this.boy = boy;
    }

    double alan() {          // davranış
        return en * boy;
    }

    public static void main(String[] args) {
        Dikdortgen d = new Dikdortgen(3, 4);
        System.out.println("Alan: " + d.alan());
    }
}
```

```
Alan: 12.0
```

Sınıfın dört parçası vardır:

- `double en, boy;` satırı iki alan tanımlar. Her `Dikdortgen` nesnesi kendi `en` ve `boy` değerlerini taşır; bu iki değer nesnenin durumudur.

- `Dikdortgen(double en, double boy)` bir **yapıcıdır** (*constructor*). Yapıcının adı sınıfın adıyla aynıdır ve dönüş tipi yoktur. Nesne oluşturulurken bir kez çağrılır ve alanlara ilk değerlerini verir.
- `double alan()` bir metottur. Nesnenin kendi alanlarını kullanarak alanı hesaplar ve `double` bir değer döndürür.
- `main` metodu programın giriş noktasıdır. Bir nesne oluşturur, nesnenin `alan()` metodunu çağırır ve sonucu yazdırır.

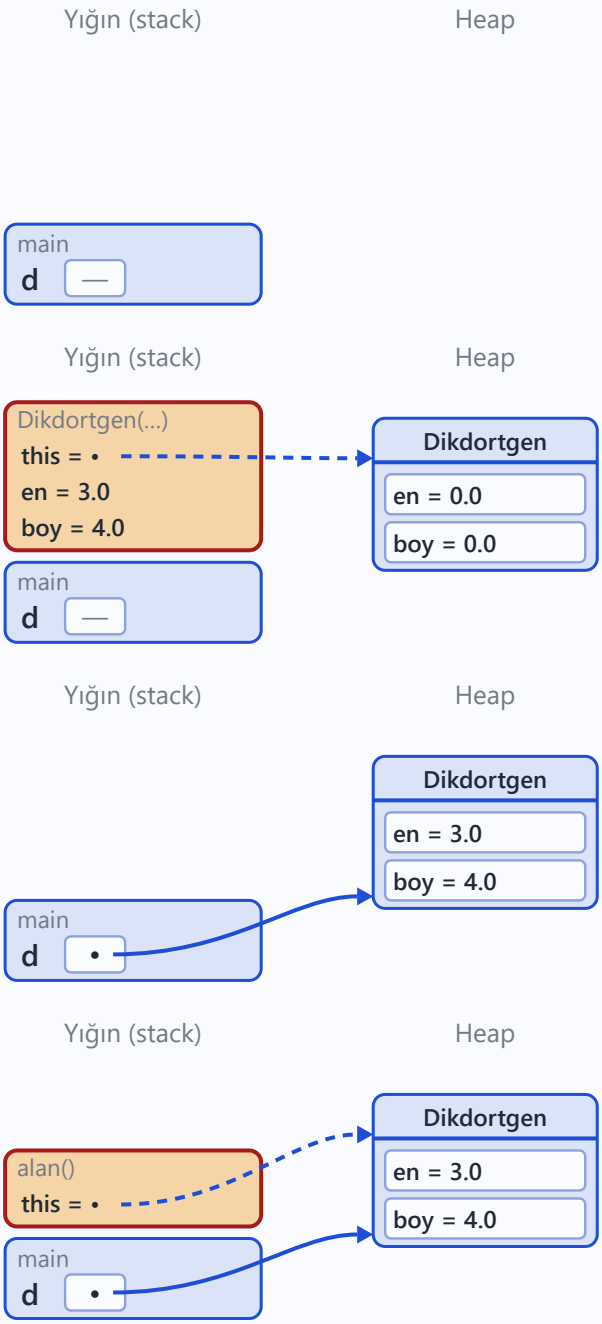
Yapıcının içindeki `this.en = en;` satırı ilk bakışta tuhaf görünür. Burada iki ayrı `en` vardır. Sağdaki `en`, yapıcının parametresidir ve çağrıda verilen değeri taşır. Soldaki `this.en` ise nesnenin alanıdır. `this` anahtar sözcüğü "şu anda üzerinde çalışılan nesne" anlamına gelir. Parametre ile alan aynı adı taşıdığında parametre alanı gölgeler; alana ulaşmak için `this.` yazmak gerekir.

`alan()` metodunda ise `this` yazılmamıştır: `return en * boy;`. Metodun içinde `en` adında bir parametre ya da yerel değişken olmadığı için bu ad doğrudan alana gider. `return this.en * this.boy;` yazmak da aynı anlama gelir.

> **Sık yapılan hata**
>
> Yapıcıda `this` yazmayı unutmak: `en = en;`. Bu satır derlenir, ama parametreyi kendisine atar; alan değişmez. Nesnenin `en` ve `boy` alanları varsayılan 0.0 değerinde kalır ve program `Alan: 0.0` yazar. Derleyici bu durumda hata vermez, çünkü satır dil açısından geçerlidir. Bu bir mantık hatasıdır.

2.1.2.6 Kod izleme: Dikdortgen nesnesi

Nesne yönelimli bir programı izlerken bellekte iki bölge birlikte düşünülür. **Yığın** (*stack*), metot çağrılarının çerçevelerini ve yerel değişkenleri tutar. **Heap** ise `new` ile oluşturulan nesnelerin yaşadığı bölgedir. Yerel değişken `d` yığındadır; gösterdiği `Dikdortgen` nesnesi heap'tedir.



Şekil 2.1.4: Dikdortgen programının izlemesinden dört an. (a) `main` başlar; yığında `main` çerçevesi vardır, `d` henüz bir nesneyi göstermez. (b) `new` heap'te nesneye yer ayırır, alanlar varsayılan 0.0'dır; yapıcı çerçevesinde `this` nesneyi gösterir, parametreler 3.0 ve 4.0'dır. (c) Yapıcı bitti, alanlar 3.0 ve 4.0 oldu; `d` artık nesneyi gösterir. (d) `d.alan()` çağrısında `this` ile `d` aynı nesneyi gösterir.

Şekil 2.1.4 programın akışını adım adım verir. Ayrıntılar şöyledir:

1. `main` çalışmaya başlar. Yığında `main` çerçevesi açılır, `d` değişkenine yer ayrılır ama henüz bir nesne yoktur.

2. `new Dikdortgen(3, 4)` ifadesi değerlendirilir. `new` heap'te nesneye yer ayırır ve alanlara varsayılan değer 0.0'ı verir. Ardından yapıcı çağrılır. Çağrıdaki 3 ve 4 tam sayıdır, parametreler ise `double`'dır; bu yüzden 3 ve 4 otomatik olarak 3.0 ve 4.0'a genişletilir.
3. `this.en = en;` çalışır: nesnenin `en` alanı 3.0 olur.
4. `this.boy = boy;` çalışır: `boy` alanı 4.0 olur. Yapıcı biter ve çerçevesi yığından kalkar.
5. `new` ifadesinin değeri nesnenin **referansıdır**, yani heap'teki yerini gösteren değerdir. Bu referans `d`'ye atanır.
6. `d.alan()` çağrılır. Yeni çerçevede `this`, `d` ile aynı nesneyi gösterir. `en * boy` = 3.0 × 4.0 = 12.0 hesaplanır ve döndürülür.
7. Dönen 12.0, `"Alan: "` dizgisiyle birleştirilir ve `Alan: 12.0` yazılır.

Çıktıda 12 değil 12.0 yazmasının nedeni tiptir. `alan()` bir `double` döndürür. Java bir `double` değeri dizgiye çevirirken, değer tam sayı olsa bile en az bir ondalık basamak yazar.

Bu küçük program nesne yönelimli paradigmanın özünü taşır. Veri (`en`, `boy`) ve davranış (`alan`) aynı nesnede durur. `main` metodu alanın nasıl hesaplandığını bilmez; yalnız nesneden alanını ister. Hesaplama yöntemi değişse bile, örneğin alanlar santimetre yerine milimetre tutulsa, `main`'deki çağrı aynı kalır.

Örnek 2.1.2

`main`'e şu iki satır eklensin: `Dikdortgen k = new Dikdortgen(2, 5);` ve `System.out.println(k.alan() + k.en);`.

Heap'te ikinci bir nesne oluşur; alanları 2.0 ve 5.0'dır. İlk nesne bundan etkilenmez, çünkü her nesnenin alanları ayrıdır. `k.alan()` 10.0 döndürür; `k.en` 2.0'dır. İki `double` toplanır ve `12.0` yazılır.

Satır `"Alan: " + k.alan() + k.en` olsaydı sonuç farklı olurdu. Soldaki işlenen dizgi olduğu için iki `+` da birleştirme yapar ve `Alan: 10.02.0` yazılır.

2.1.3 Söz dizimi ve anlam

Programlama dilleri ile doğal diller birçok ortak yön paylaşır. Her ikisinde de temel yapılar vardır, bu yapıların nasıl dizileceğini belirleyen kurallar vardır ve doğru dizilmiş bir metnin bir anlamı vardır. Bir dili incelemenin üç katmanı bu gözleme dayanır: söz dizimi, statik anlam ve anlam. Bu ayrım yalnız kuramsal değildir. Programdaki bir hatanın katmanı, hatayı kimin ve ne zaman bulacağını belirler: derleyici mi, JVM mi, yoksa test eden kişi mi?

2.1.3.1 Temel yapılar ve söz dizimi

Türkçede temel yapılar kelimeler ve noktalama işaretleridir. Java'da ise sayılar, dizgiler, basit işleçler (`+ - * /`) ve anahtar sözcüklerdir. `int` , `class` , `if` ve `return` anahtar sözcüklere örnektir; dil bu sözcükleri kendine ayırmıştır.

> **Tanım 2.1.8**
>
> **Söz dizimi** (*syntax*), bir dilde hangi karakter ve sembol dizilerinin düzgün olduğunu belirleyen biçim kurallarıdır. Bu kurallara uymayan bir metin, içindeki sözcükler tek tek doğru olsa bile geçersizdir.

"kedi köpek çocuk" dizisini oluşturan kelimelerin her biri Türkçedir, ama ortada bir cümle yoktur; özne, nesne ve yüklem kurulmamıştır. Java'da da durum aynıdır. `int x = 5;` geçerli bir satırdır: önce tip, sonra ad, sonra atama işareti, sonra değer, en sonda noktalı virgül gelir. Aynı parçalar başka bir sırayla dizilirse satır geçersiz olur:

```
int = x 5;
```

Derleyici bu satırı reddeder. JDK 21'in tam iletisi şudur:

```
Hata.java:3: error: not a statement
        int = x 5;
        ^
Hata.java:3: error: ';' expected
        int = x 5;
           ^
Hata.java:3: error: not a statement
        int = x 5;
              ^
Hata.java:3: error: ';' expected
        int = x 5;
               ^
4 errors
```

Tek bir satırdaki tek bir sıralama yanlışı dört hata iletisine yol açmıştır. Bunun nedeni derleyicinin hatadan sonra durmaması, satırın geri kalanını anlamlandırmaya çalışmasıdır. `int`'ten sonra bir ad beklerken `=` görür ve "bu bir deyim değil" der. Ardından bir `;` bekler, bulamaz. Kalan `x` ve `5` parçalarını da ayrı ayrı yorumlamaya çalışır ve iki ileti daha üretir. Sonraki iletiler ilk hatanın yan etkisidir. Bu yüzden birden çok hata listelendiğinde düzeltmeye **ilkinden** başlanır; ilk hata düzelince çoğu zaman ötekiler de kaybolur.

Söz dizimi dilin anlaşılırlığını da etkiler. Kuralları kesin olan bir dilde bir satırın nasıl okunacağı konusunda tartışma çıkmaz. Söz dizimi hataları bu yüzden en kolay bulunan hatalardır: kurallar açıkça tanımlanmıştır ve derleyici her satırı bu kurallara göre denetler.

2.1.3.2 Statik anlam

Söz dizimi doğru olan her metin anlamlı değildir. İngilizcedeki "I are big" cümlesi biçim olarak `<isim> <fiil> <isim>` kalıbına uyar. Ama "I" tekil, "are" çoğuldur; cümle dil açısından geçersizdir. Bu tür bir uyumsuzluk cümlenin dizilişinde değil, parçaların birbirine uygunluğundadır.

> **Tanım 2.1.9**
>
> **Statik anlam** (*static semantics*), söz dizimi doğru olan bir ifadenin anlamlı olup olmadığını program çalışmadan önce denetleyen kurallardır. Java'da bu kuralların en önemlisi tip uyumudur.

Java'da `<değer> <işleç> <değer>` geçerli bir kalıptır. `2.3 / 4` bu kalıba uyar ve anlamlıdır. Aşağıdaki satır da aynı kalıba uyar, ama bir sayıyı bir dizgiye bölmenin anlamı yoktur:

```
double sonuc = 2.3 / "abc";
```

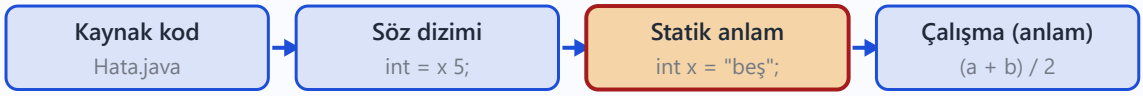
```
error: bad operand types for binary operator '/'
  first type:  double
  second type: String
```

İleti, `/` işlecinin bu iki tiple kullanılamayacağını söyler ve iki işlenenin tipini ayrı ayrı yazar: birincisi `double`, ikincisi `String`. Aynı türden bir başka hata, bir değişkene tipine uymayan bir değer atamaktır:

```
int x = "beş";
```

```
error: incompatible types:
  String cannot be converted to int
```

Bu iki satırın söz diziminde bir kusur yoktur. Hata, derleyicinin tip denetimi adımında bulunur. Java **statik tipli** bir dildir: derleyici her değişkenin ve her ifadenin tipini bilir, her işlecin işlenenlerinin tipini denetler. Bu yüzden bu tür hatalar program hiç çalışmadan yakalanır.



Şekil 2.1.5: Kaynak kodun geçtiği denetimler. Kod önce söz dizimi (`int = x 5;` burada takılır), sonra statik anlam denetiminden (`int x = "beş";` burada takılır) geçer; ikisi de derleme zamanındadır. Anlam hataları (`(a + b) / 2` ile ortalama) ise ancak program çalışırken ortaya çıkar.

Şekil 2.1.5 üç katmanı bir boru hattı gibi gösterir. Her katman bir öncekinden geçmiş kodu alır. Söz dizimi hatası olan bir satır tip denetimine hiç ulaşmaz. Tip denetiminden geçen kod derlenir ve çalıştırılabilir. Son kutudaki hatalar ise derleyicinin göremeyeceği türdendir.

Statik anlam denetiminin gücü dilden dile değişir. Bu denetimi zayıf olan dillerde yakalanmayan hatalar programın davranışını öngörülemez kılar. C dilinde bir dizinin sınırı dışına yazmak derlemede yakalanmaz ve program belleğin başka bir yerini sessizce bozabilir. Java bu tür hataların bir kısmını derlemede, kalanını çalışma zamanında bir istisnayla yakalar; hiçbirini sessiz bırakmaz.

2.1.3.3 Anlam

Tanım 2.1.10

Anlam (*semantics*), söz dizimi doğru bir ifadenin ne anlama geldiğini, yani çalıştığında ne yaptığını belirler.

Doğal dillerde bir cümlenin birden fazla anlamı olabilir. “They saw the man with the telescope.” cümlesi iki biçimde okunur: adamı teleskopla gördüler, ya da teleskobu olan adamı gördüler. Dinleyen kişi doğru anlamı bağlamdan çıkarır. Programlama dillerinde böyle bir çok anlamlılık yoktur. Her geçerli ifadenin tek bir anlamı vardır ve bu anlam dil belirtiminde yazılıdır.

Ancak bu tek anlam, programcının beklediği şey olmayabilir. Aşağıdaki kod 7 ile 8'in ortalamasını üç farklı biçimde hesaplamaya çalışır:

```
int a = 7, b = 8;
int ortalama = (a + b) / 2;
System.out.println(ortalama);       // 7.5 beklenirken
System.out.println(a + b / 2);      // öncelik
System.out.println((a + b) / 2.0);  // doğru
```

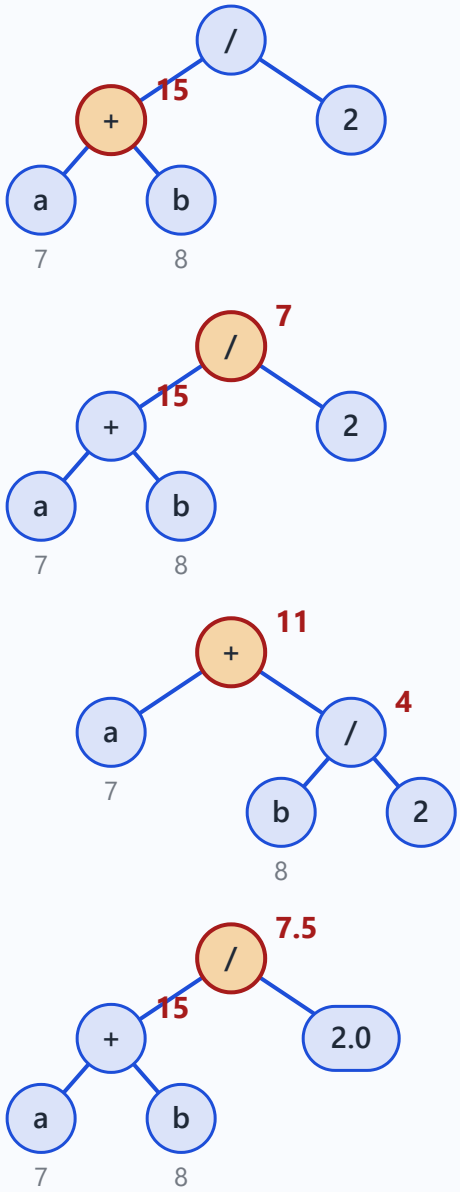
```
7
11
7.5
```

7 ile 8'in ortalaması 7,5'tir. Yalnız üçüncü satır bu değeri yazar. İlk iki satır derlenir, çalışır ve bir sayı yazar; ama sayı yanlıştır. Derleyici bu satırlarda bir hata görmez, çünkü dil açısından ortada bir hata yoktur. Her satır Java'nın kurallarına göre tam olarak ne yazılmışsa onu yapar.

- `(a + b) / 2`: iki işlenen de `int` olduğu için Java **tam sayı bölmesi** yapar. 15 / 2'nin kesirli kısmı atılır ve 7 bulunur.
- `a + b / 2`: parantez yoktur ve `/` işleminin önceliği `+`'dan yüksektir. Önce `b / 2` = 4 hesaplanır, sonra 7 + 4 = 11 bulunur.
- `(a + b) / 2.0`: `2.0` bir `double` sabittir. 15 de `double`'a genişletilir ve gerçek bölme yapılır: 7.5.

2.1.3.4 Kod izleme: tam sayı bölmesi

Bir ifadenin nasıl değerlendirildiği, **ifade ağacıyla** açıkça görülür. Ağaçta her iç düğüm bir işleç, her yaprak bir değer ya da değişkendir. Bir işleç ancak alt düğümlerinin değerleri bulunduktan sonra uygulanır; bu yüzden değerlendirme yapraklardan köke doğru ilerler.



Şekil 2.1.6: Ortalama programının ifade ağaçları. (a) `(a + b) / 2` : önce parantez, `a + b` = 15. (b) `15 / 2` : iki işlenen de `int` , tam sayı bölmesiyle 7. (c) `a + b / 2` : parantez olmayınca bölme alta iner; önce `b / 2` = 4, sonra 7 + 4 = 11. (d) `(a + b) / 2.0` : 2.0 `double` olduğu için sonuç 7.5.

Şekil 2.1.6'daki (a) ve (c) ağaçları aynı dört parçadan (`a` , `b` , `+` , `/`) ve aynı 2 sabitinden oluşur. Yalnız biçimleri farklıdır. Parantez `+` düğümünü alta, yani önce yapılacak yere taşır. Parantez olmadığında öncelik kuralı `/` 'yi alta koyar. Aynı karakterlerle yazılan iki ifadenin farklı sonuç vermesinin nedeni budur.

Satır	Hesap	Değişkenler	Çıktı
1	—	`a` = 7, `b` = 8	
2	(7 + 8) / 2 = 15 / 2 = 7	`ortalama` = 7	
3	—		7

Satır	Hesap	Değişkenler	Çıktı
4	7 + 8 / 2 = 7 + 4 = 11		11
5	(7 + 8) / 2.0 = 15.0 / 2.0 = 7.5		7.5

Tablo 2.1.5: Ortalama programının satır satır izlemesi.

Programın ikinci satırı bu notun en sık rastlanacak tuzağını gösterir. `ortalama` değişkeni `int` olarak tanımlanmıştır. Sağ taraf 7.5 üretseydi bile Java bu değeri `int`'e atamaya izin vermezdi. Değişken `double` yapılsa da sorun çözülmez; sağ taraf yine tam sayı bölmesiyle 7 üretir, atama yalnız 7'yi 7.0'a çevirir.

Sık yapılan hata

Sonucu `double` bir değişkene yazmanın bölmeyi ondalıklı yaptığını sanmak. `double ort = (a + b) / 2;` satırı 7.5 değil 7.0 verir. Java önce sağ tarafı kendi tipleriyle hesaplar: iki `int`'in bölümü `int`'tir. Atama ancak bu hesap bittikten sonra yapılır ve kaybolan kesir geri gelmez. Doğrusu işlenenlerden birini `double` yapmaktır: `(a + b) / 2.0`.

2.1.4 Hata türleri ve hata iletileri

Program yazmak, hata yapmayı ve hatayı bulmayı da içerir. Deneyimli programcılar bile ilk denemede hatasız kod yazmaz. Fark, hatanın türünü hızla tanımak ve onu nerede arayacağını bilmektir. Söz dizimi, statik anlam ve anlam ayrımı burada doğrudan işe yarar.

2.1.4.1 Dört hata türü

Hata türü	Ne zaman fark edilir?	Java örneği	Belirti
Söz dizimi	derlemede (derleyici)	`System.out.println("Merhaba")`	`';' expected`
Statik anlam	derlemede (tip denetimi)	`int x = "beş";`	`incompatible types`
Çalışma zamanı	program çalışırken	`a / b` (`b` sıfırken)	çökme: `ArithmeticException`
Mantık	derleyici fark etmez; ancak testle	`(a + b) / 2` ile ortalama	yanlış sonuç, sonsuz döngü

Tablo 2.1.6: Hata türleri, fark edildikleri an ve Java'daki belirtileri.

İlk iki tür **derleme hatası** diye birlikte anılır. İkisi de program çalışmadan bulunur ve hata varken `.class` dosyası oluşmaz. Söz dizimi hataları yazım hataları gibidir: açıkça tanımlanmış kurallara aykırılıktır ve derleyici bunları kolayca bulur. Statik anlam hataları ise dile özgü denetimlerle aranır; Java'da bu denetimin merkezinde tipler vardır.

Tanım 2.1.11

Çalışma zamanı hatası (*runtime error*), derlenmiş bir program çalışırken ortaya çıkan ve Java'da bir **istisnaya** (*exception*) yol açan hatadır. **Mantık hatası** (*logic error*) ise programın çalışıp yanlış sonuç üretmesidir; ne derleyici ne de JVM bunu fark eder.

Tablonun son satırı en tehlikeli durumu anlatır. Söz dizimi ve statik anlam açısından kusursuz bir program bile beklenen sonucu üretmeyebilir. Program çökebilir, sonsuz döngüye girebilir ya da hiçbir uyarı vermeden yanlış bir sonuç verebilir. Bu sonuncusu en zor bulunanıdır, çünkü bir şeylerin yanlış gittiğini gösteren bir belirti yoktur.

2.1.4.2 Derleme hatası iletisini okumak

Derleyicinin hata iletisi korkutucu görünebilir, ama düzenli bir yapısı vardır. Aşağıdaki programda bir noktalı virgül eksiktir:

```
public class Selam {
    public static void main(String[] args) {
        System.out.println("Merhaba")
    }
}
```

```
Selam.java:3: error: ';' expected
        System.out.println("Merhaba")
                                    ^
1 error
```

İleti dört parçadan oluşur:

- `Selam.java:3` : hatanın bulunduğu dosya ve satır numarası.
- `error: ';' expected` : derleyicinin ne beklediği. Burada bir noktalı virgül bekleniyordu.
- Satırın kopyası ve altındaki `^` işareti: hatanın satır içindeki konumu. İşaret, kapanan parantezden hemen sonrasını, yani `;` 'ün gelmesi gereken yeri gösterir.
- `1 error` : toplam hata sayısı.

Hata varsa `.class` dosyası oluşmaz ve program hiç çalışmaz. Önceki bir derlemeden kalmış eski bir `.class` dosyası varsa, `java` komutu yanlışlıkla o eski sürümü çalıştırabilir. Kodda yapılan değişikliğin etkisinin görülmemesi çoğu zaman bu yüzdendir: derleme başarısız olmuştur ama ileti gözden kaçmıştır.

Noktalı virgül eksikse `javac` hatayı, `;` 'ün eksik olduğu satırın sonunda gösterir. Aşağıdaki parçada ilk satırın sonundaki `;` unutulmuştur:

```
int x = 5
int y = 6;
System.out.println(x + y);
```

```
Semi.java:3: error: ';' expected
        int x = 5
                ^
1 error
```

İşaret ikinci satıra değil, `5`'in hemen arkasına konmuştur. Derleyici `5`'ten sonra gelen sözcüğü (`int`) görünce deyimin bitmesi gerektiğini anlar ve eksikliği bir önceki parçanın sonuna yazar. Dosyanın 3. satırı, kod parçasının ilk satırıdır; parçanın üstünde sınıf ve `main` başlıkları vardır.

Örnek 2.1.3

Bir programda `int toplam = 0;` tanımlanmış, yedinci satırda ise `System.out.println(Toplam);` yazılmıştır. Derleyici şu iletiyi verir: `Hesap.java:7: error: cannot find symbol`, altında da `symbol: variable Toplam`.

İleti okunur: hata 7. satırdadır; derleyici `Toplam` adında bir değişken bulamamıştır. Tanımlanan ad `toplam`'dır. Java büyük ve küçük harfi ayırır; bu iki ad farklıdır. Bu bir derleme (statik anlam) hatasıdır. `Toplam` yerine `toplam` yazılınca program derlenir.

2.1.4.3 Çalışma zamanı hatası ve istisna

Aşağıdaki program sorunsuz derlenir. Hata ancak çalışırken ortaya çıkar:

```
public class Bolme {
    public static void main(String[] args) {
        int a = 10, b = 0;
        System.out.println(a / b);
    }
}
```

```
Exception in thread "main"
java.lang.ArithmeticException: / by zero
        at Bolme.main(Bolme.java:4)
```

Derleyici `a / b` ifadesinde bir sorun görmez: iki `int`'in bölümü geçerli bir ifadedir. `b`'nin o anda 0 olduğu ancak program çalışırken bilinir. Tam sayılar sıfıra bölünemez; JVM bu durumda bir **istisna** fırlatır. İletinin parçaları şöyledir:

- `Exception in thread "main"` : istisna `main` iş parçacığında oluştu.
- `java.lang.ArithmeticException` : istisnanın türü, yani sınıfının tam adı.
- `/ by zero` : istisnanın açıklaması; tam sayı sıfıra bölünmeye çalışıldı.
- `at Bolme.main(Bolme.java:4)` : istisnanın oluştuğu metot, dosya ve satır.

Gerçek çıktıda ilk iki parça tek satırda yazılır; burada sayfaya sığsın diye bölünmüştür. İstisna bir `try` - `catch` bloğuyla yakalanmazsa program bu satırda sonlanır ve sonraki satırlar hiç çalışmaz.

Sıfıra bölmenin davranışı tipe bağlıdır. `10 / 0` istisna fırlatır. `10.0 / 0` ise istisna fırlatmaz ve `Infinity` değerini üretir; `0.0 / 0` da `NaN` (*not a number*, sayı değil) verir. Kayan noktalı sayıların bu özel değerleri vardır, tam sayıların yoktur. Bu yüzden `double` ile yapılan bir sıfıra bölme programı çökertmez, ama sonucu sessizce anlamsız bir değere dönüştürür.

Bir başka sık çalışma zamanı hatası, dizinin sınırı dışına çıkmaktır. `int[] a = new int[3];` üç öğeli bir dizi oluşturur; geçerli indisler 0, 1 ve 2'dir. `a[3] = 1;` derlenir, ama çalışırken `ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3` istisnasını fırlatır. İndisin değeri çoğu zaman bir hesabın sonucudur ve derleyici onu önceden bilemez.

2.1.4.4 Mantık hatası ve test

Mantık hatasında program derlenir, çalışır ve bir sonuç verir; yalnız sonuç yanlıştır. Ortalamayı `(a + b) / 2` ile almak, dairenin alanı için `2 * pi * r` yazmak (bu çevredir), bir döngünün sınırını bir eksik ya da bir fazla kurmak bu türdendir. Dil açısından bu satırların hiçbirinde kusur yoktur.

Mantık hatalarını bulmanın yolu **testtir**: programı, sonucu önceden bilinen girdilerle çalıştırıp çıktıyı beklenen değerle karşılaştırmak. 7 ile 8'in ortalamasının 7,5 olduğu bilinir; program 7 yazıyorsa bir hata vardır. İyi seçilmiş birkaç test, hatanın yerini de daraltır. 6 ile 8 için program doğru sonucu (7) verir, 7 ile 8 için yanlış sonucu verir; bu fark, kesirli sonuçların kaybolduğunu düşündürür.

İstisna mekanizması mantık hatalarını yakalamaz. İstisna, geçersiz bir işlem yapıldığında fırlatılır; mantık hatasında ise yapılan her işlem geçerlidir. Bu ayrım önemlidir: programın istisna fırlatmadan çalışması, doğru çalıştığını göstermez.

Örnek 2.1.4

Aşağıdaki dört satırın her biri ayrı bir programda olsun. Hata türleri bulunsun.

(a) `String s = 5;` Statik anlam hatası. Derleyici `incompatible types: int cannot be converted to String` der. Satırın biçimi doğrudur, tipler uyuşmaz.

(b) `System.out.println("Merhaba";` Söz dizimi hatası. Kapanan parantez eksiktir; derleyici `')' or ',' expected` der.

(c) `int k = 0; System.out.println(10 % k);` Çalışma zamanı hatası. Kalan işlemi de bölme gibi sıfıra yapılamaz; `ArithmeticException: / by zero` fırlatılır.

(d) `double cevre = pi * r * r;` Mantık hatası. Kod derlenir ve çalışır, ama çevre yerine alanı hesaplar. Doğrusu `2 * pi * r` 'dir.

2.1.5 İfadeler ve işleç önceliği

Bir programın hesap yapan her parçası bir ifadedir. Bir ifadenin değerini doğru bulmak için iki şey bilinmelidir: işlemlerin hangi sırayla yapıldığı ve her işlemin hangi tiple sonuç verdiği. Bu kesim birinciyi, yani öncelik kurallarını işler; tipler de bu kuralların içine yer yer girer.

2.1.5.1 İfade, işleç, işlenen

Tanım 2.1.12

İfade (*expression*), değerlendirilince bir değer üreten yapıdır: `3 + 4`, `x > 0`, `pi * r * r`. İfadedeki işlem işaretlerine **işleç** (*operator*), işlecin üzerinde çalıştığı değerlere **işlenen** (*operand*) denir.

`3 + 4` ifadesinde `+` işleç, 3 ve 4 işlenendir; ifadenin değeri 7'dir. İşlenenler de birer ifade olabilir. `(a + b) / 2` ifadesinde `/` işlecinin sol işleneni `(a + b)` ifadesidir. İfadeler böylece iç içe geçer ve bir ağaç oluşturur.

Java'daki temel aritmetik işleçler beştir: toplama `+`, çıkarma `-`, çarpma `*`, bölme `/` ve kalan `%`. Kalan işleci, bölmenin kalanını verir: `7 % 3` = 1, çünkü 7 = 2 · 3 + 1'dir. Bunlara ek olarak karşılaştırma işleçleri (`<`, `>`, `<=`, `>=`, `==`, `!=`) iki değeri karşılaştırır ve `boolean` bir değer, yani `true` ya da `false` üretir.

2.1.5.2 Öncelik ve soldan sağa kuralı

Tanım 2.1.13

İşleç önceliği (*operator precedence*), bir ifadede işleçlerin hangi sırayla uygulanacağını belirleyen kuraldır. Aynı öncelikteki aritmetik işleçler **soldan sağa** uygulanır.

Java'nın aritmetik öncelikleri okulda öğrenilen matematik kurallarıyla aynıdır. Tablo 2.1.7 bu notta kullanılan işleçleri yüksek öncelikten düşüğe doğru sıralar.

Düzey	İşleçler	Yön
1	parantez `( )`	içten dışa
2	`*` `/` `%`	soldan sağa

Düzey	İşleçler	Yön
3	`+` `-` (toplama, çıkarma, dizgi birleştirme)	soldan sağa
4	`<` `>` `<=` `>=`	soldan sağa
5	`==` `!=`	soldan sağa
6	`=` (atama)	sağdan sola

Tablo 2.1.7: Bu notta kullanılan işleçlerin önceliği (yukarıdan aşağıya azalır).

Tablodaki iki ayrıntı sık gözden kaçar. Birincisi, `*`, `/` ve `%` aynı düzeydedir. Biri ötekinden önce gelmez; hangisi soldaysa o önce yapılır. `12 / 3 * 2` ifadesi 12 / 6 = 2 değil, (12 / 3) · 2 = 8 verir. İkincisi, atama en düşük önceliklidir ve sağdan sola işler. Bu yüzden `x = y = 7;` önce `y`'ye 7 atar, sonra bu atamanın değerini (7) `x`'e atar.

Aynı öncelikteki işlemlerin soldan sağa yapılması, çıkarmada sonucu doğrudan etkiler. `10 - 4 - 3` ifadesi (10 − 4) − 3 = 3 diye hesaplanır. Sağdan başlansaydı 10 − (4 − 3) = 9 bulunurdu. Toplama ve çarpmada sıra sonucu değiştirmez, ama tam sayı bölmesi devreye girince değiştirir: `7 / 2 * 2` = 3 · 2 = 6, oysa `7 * 2 / 2` = 14 / 2 = 7'dir.

Örnek 2.1.5

`10 - 2 * 3 + 8 / 4 % 3` ifadesinin değeri bulunsun.

En yüksek öncelikli işleçler `*`, `/` ve `%`'dir. Soldan sağa: `2 * 3` = 6. Sonra `8 / 4` = 2, ardından `2 % 3` = 2. İfade `10 - 6 + 2` olur.

`-` ile `+` aynı düzeydedir, soldan sağa: 10 − 6 = 4, 4 + 2 = 6. Sonuç **6**'dır ve tipi `int`'tir.

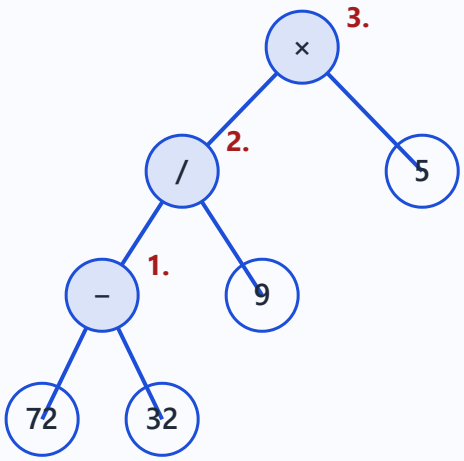
2.1.5.3 İfade değerlendirme adım adım

Bilgisayar bir ifadeyi tek seferde değil, öncelik kurallarına göre adım adım sadeleştirir. Her adımda en yüksek öncelikli işlemlerden en soldakini yapar ve onun yerine sonucunu koyar. Fahrenhayttan Celsius'a dönüşüm formülünden alınan şu ifade bunu iyi gösterir: `(72 - 32) / 9 * 5`. Matematikte bu ifadenin değeri 40 · 5 / 9 ≈ 22,22'dir.

Adım	İfade	Yapılan işlem
1	`(72 - 32) / 9 * 5`	önce parantez: 72 − 32 = 40
2	`40 / 9 * 5`	`/` ile `*` aynı düzeyde, soldan sağa: önce 40 / 9
3	`4 * 5`	iki işlenen `int`: tam sayı bölmesi, 4,44… değil 4 (kalan 4 atıldı)
4	`20`	sonuç 20; beklenen 22,22… idi
5	`(72.0 - 32) / 9 * 5`	`double` − `int`: 32, 32.0'a genişletilir; sonuç 40.0
6	`40.0 / 9 * 5`	gerçek bölme: 4.444444444444445
7	`4.444444444444445 * 5`	`double` × `int` → `double`

Adım	İfade	Yapılan işlem
8	`22.22222222222222`	doğru sonuç

Tablo 2.1.8: `(72 - 32) / 9 * 5` ve `(72.0 - 32) / 9 * 5` ifadelerinin adım adım değerlendirilmesi.



Şekil 2.1.7: `(72 - 32) / 9 * 5` ifadesinin ağacı. Alttaki işlem önce yapılır; numaralar işlem sırasıdır: 1. çıkarma, 2. bölme, 3. çarpma.

Şekil 2.1.7 ifadenin yapısını gösterir. Kökte çarpma vardır, çünkü en son o yapılır. Çarpmanın sol işleneni bölmedir, bölmenin sol işleneni de çıkarmadır. Soldan sağa kuralı ağaçta şu biçimde görünür: `/` ile `*` aynı öncelikte olduğu için soldaki (`/`) alta iner ve önce yapılır.

Tablo 2.1.8'in ilk dört adımı, öncelik kurallarının doğru uygulandığı hâlde sonucun yanlış çıkabileceğini gösterir. Hata öncelikte değil, tiptedir. Üçüncü adımda iki `int` bölünür ve kesir kaybolur. Bu kayıp sonraki çarpmayla büyür: 0,44'lük kayıp beşle çarpılır ve sonuç 2,22 eksik çıkar. Bu bir anlam (mantık) hatasıdır; derleyici uyarmaz.

Düzeltme için tek bir sabitin `double` yapılması yeterlidir. `72.0` yazılınca çıkarmanın sonucu `double` olur ve bu tip ifadenin geri kalanına yayılır. Bir işlemin işlenenlerinden biri `double` ise öteki de `double`'a genişletilir ve sonuç `double` olur. Ondalıklı sonuç gerekiyorsa işlenenlerden en az biri `double` olmalıdır; bu genişletme de bölmeden *önce* gerçekleşmelidir.

Son adımdaki 22.22222222222222'nin son basamağı da dikkat çekebilir. Matematikte 200 / 9 = 22,222… sonsuza kadar sürer. `double` yaklaşık 15–17 anlamlı basamak tutar; Java bu değeri yazarken onu tek başına belirleyen en kısa basamak dizisini seçer. Kayan noktalı sayılar bu yüzden her zaman yaklaşık değerdir.

Örnek 2.1.6

Aynı hesap işlem sırası değiştirilerek `(72 - 32) * 5 / 9` diye yazılsın.

Parantez: 40. Soldan sağa, önce çarpma: 40 · 5 = 200. Sonra bölme: 200 / 9 = 22 (tam sayı bölmesi, kalan 2 atılır). Sonuç **22**'dir.

Kayıp küçülmüştür: tam sayı bölmesi en sona kaldığı için yalnız son kesir (0,22…) atılır. Çarpmayı bölmeden önce yapmak tam sayılarla çalışırken hatayı azaltır, ama ortadan kaldırmaz. Tam sonuç için yine `double` gerekir: `200 / 9.0` = 22.22222222222222.

2.1.5.4 Tam sayı bölmesi ve kalan

Tanım 2.1.14

Tam sayı bölmesi (*integer division*), iki tam sayının bölümünde kesirli kısmın atılmasıdır. Java'da `/` işlecinin iki işleneni de tam sayı tipindeyse (`int` , `long` gibi) sonuç da tam sayıdır. Sonuç sıfıra doğru kesilir: `7 / 2` = 3, `-7 / 2` = −3.

"Sıfıra doğru kesilir" ifadesi negatif sayılarda önem kazanır. −7 / 2 = −3,5'tir. Kesir atılınca −3 kalır; sonuç sıfıra yaklaşır. Bazı diller (örneğin Python'un `//` işleci) aşağı yuvarlar ve −4 verir. Java her zaman sıfıra doğru keser.

Kalan işleci `%` , bölmeyle tutarlı çalışır. Her `a` ve sıfırdan farklı `b` için şu eşitlik sağlanır:

> (a / b) * b + a % b == a

Bu eşitlikten kalanın işaretinin bölünenin işaretiyle aynı olduğu çıkar. `-7 % 2` = −1'dir, çünkü (−3) · 2 + (−1) = −7'dir. `-17 % 5` = −2'dir. Buna karşılık `7 % -2` = 1'dir; bölenin işareti kalanı etkilemez. Bir sayının çift olup olmadığı `n % 2 == 0` ile sınanır. Negatif tek sayılarda kalan −1 olduğu için tek sayı sınaması `n % 2 == 1` değil, `n % 2 != 0` ile yapılmalıdır.

İfade	Sonuç	Tip	Açıklama
`7 % 3`	`1`	`int`	7 = 2 · 3 + 1
`17 / 5`	`3`	`int`	3,4'ün kesri atılır
`17 % 5`	`2`	`int`	17 = 3 · 5 + 2
`-7 / 2`	`-3`	`int`	sıfıra doğru kesilir
`-7 % 2`	`-1`	`int`	kalan bölünenin işaretini alır
`1 / 3 * 3`	`0`	`int`	1 / 3 = 0, sonra 0 · 3
`1.0 / 3 * 3`	`1.0`	`double`	gerçek bölme
`17 / 5.0`	`3.4`	`double`	bir işlenen `double`

Tablo 2.1.9: Tam sayı bölmesi ve kalan örnekleri (JDK 21 ile çalıştırılarak doğrulanmıştır).

Tam sayı bölmesi her zaman bir hata değildir. Birçok algoritma onu bilerek kullanır. Saniyeyi dakika ve saniyeye ayırmak bunun tipik örneğidir: 135 saniye için `135 / 60` = 2 dakika, `135 % 60` = 15 saniye. Bir sayının basamaklarını ayırmak da aynı yolla yapılır: `n % 10` son basamağı verir, `n / 10` son basamağı atar. Sorun, ondalıklı bir sonuç beklenen yerde tam sayı bölmesinin farkında olmadan yapılmasıdır.

Örnek 2.1.7

Bir öğrencinin üç sınav notu 70, 85 ve 90'dır. Ortalama üç biçimde hesaplansın:

```
int n1 = 70, n2 = 85, n3 = 90;
System.out.println((n1 + n2 + n3) / 3);     // 81
System.out.println((n1 + n2 + n3) / 3.0);   // 81.66666666666667
double ort = (n1 + n2 + n3) / 3;
System.out.println(ort);                    // 81.0
```

Toplam 245'tir. İlk satır 245 / 3'ü tam sayı bölmesiyle 81 bulur. İkinci satırda 3.0 bir `double` olduğu için gerçek bölme yapılır ve 81.66666666666667 yazılır. Üçüncü satır en yanıltıcı olanıdır: değişken `double` olsa da sağ taraf önce 81 olarak hesaplanır; atama bunu 81.0'a çevirir.

2.1.5.5 Üs alma

Java'da üs alma işleci yoktur. Matematikte 2^3 diye yazılan işlem, `Math.pow(2, 3)` metoduyla yapılır ve sonuç `double`'dır: `8.0`. `Math.pow(2, 10)` = 1024.0, `Math.pow(2, 0.5)` = 1.4142135623730951'dir. Küçük tam sayı kuvvetleri için çarpma da kullanılır: `x * x`, x^2'dir.

Başka dillerden gelen iki alışkanlık burada sorun çıkarır. Python'daki `**` işleci Java'da yoktur; `2 ** 3` yazılırsa derleyici `illegal start of expression` der. Daha sinsi olanı `^` işaretidir. Java'da `^` geçerli bir işleçtir, ama üs değil, bit düzeyinde dışlayıcı veya (XOR) işlemidir:

```
System.out.println(2 ^ 3);          // 1: XOR
System.out.println(Math.pow(2, 3)); // 8.0
```

2'nin ikili gösterimi 010, 3'ünki 011'dir. XOR, iki sayının farklı olduğu bitlerde 1 üretir: 001, yani 1. Kod derlenir ve çalışır; yalnız sonuç yanlıştır. Derleme hatası vermediği için `^` ile üs almaya çalışmak bir mantık hatasıdır.

2.1.6 Değişkenler, atama ve tipleme

Bir programın hesapladığı değerlerin bir yerde saklanması gerekir. Bu yer değişkendir. Değişkenin tipi, içinde ne tür bir değer saklanacağını ve o değerle hangi işlemlerin yapılabileceğini belirler.

2.1.6.1 Değişken ve atama

Tanım 2.1.15

Değişken (*variable*), bir ada bağlı, değeri değişebilen bellek alanıdır. Java'da bir değişken kullanılmadan önce tipi ve adıyla **bildirilir**: `int sayac;`. **Atama** (*assignment*) ise `=` işaretiyle, sağdaki ifadenin değerini soldaki değişkene yazmaktır.

Atama iki adımda gerçekleşir ve bu iki adımın sırası önemlidir:

1. Sağ taraftaki ifade, değişkenlerin *şu anki* değerleriyle hesaplanır.
2. Bulunan değer soldaki değişkene yazılır; değişkenin eski değeri silinir.

Bu sıra, matematikte anlamsız görünen `x = x + 1` satırını anlamlı kılar. Matematikte bu bir denklemdir ve çözümü yoktur. Java'da ise bir komuttur: "`x`'in şimdiki değerine 1 ekle, sonucu `x`'e yaz." `x` 5 ise önce 5 + 1 = 6 hesaplanır, sonra 6 yazılır.

Örnek 2.1.8

Şu dört satır çalıştıktan sonra `a` ve `b` kaçtır? `int a = 4; int b = a * 3; a = b - a; b = b / a;`

1. `a` = 4. 2. Sağ taraf 4 · 3 = 12, `b` = 12. 3. Sağ taraf `a`'nın eski değeriyle hesaplanır: 12 − 4 = 8; sonra `a` = 8 olur. 4. 12 / 8 = 1 (tam sayı bölmesi, 0,5 atılır); `b` = 1.

Sonuç: `a` = 8, `b` = 1. Üçüncü satırda `a` sağda da solda da geçer. Sağdaki `a` okunurken değeri hâlâ 4'tür; yeni değer ancak hesap bittikten sonra yazılır.

Değişken adlarına **tanımlayıcı** (*identifier*) denir ve birkaç kurala uymaları gerekir. Ad bir harf, alt çizgi (`_`) ya da dolar işaretiyle (`$`) başlar; sonraki karakterler bunlara ek olarak rakam olabilir. Ad boşluk içeremez ve bir anahtar sözcük olamaz. `toplam2`, `_sayac`, `$fiyat` ve `ortalamaNot` geçerlidir. `2toplam` rakamla başladığı, `ogrenci sayisi` boşluk içerdiği, `int` de anahtar sözcük olduğu için geçersizdir. Java geleneğinde değişken adları küçük harfle başlar ve sonraki sözcükler büyük harfle bitiştirilir: `ortalamaNot`, `ogrenciSayisi`.

Java büyük ve küçük harfi ayırır. `Sayi` ile `sayi` iki farklı addır:

```
int Sayi = 4;
System.out.println(sayi);
```

```
Harf.java:4: error: cannot find symbol
        System.out.println(sayi);
                           ^
  symbol:   variable sayi
  location: class Harf
1 error
```

İleti, derleyicinin `sayi` adlı bir değişken bulamadığını söyler. `symbol` satırı aranan adı, `location` satırı aramanın yapıldığı sınıfı verir. Aynı ileti, değişkeni hiç tanımlamamak ya da adını yanlış yazmak durumunda da çıkar.

Bir yerel değişken, değer verilmeden okunamaz:

```
int s;
System.out.println(s);
```

```
IlkDeger.java:4: error: variable s might not have been initialized
        System.out.println(s);
                           ^
1 error
```

Derleyici, `s`'ye ulaşan bütün yolları inceler ve hiçbirinde atama bulamaz. Bu denetim, rastgele bir bellek değerinin sessizce kullanılmasını önler. Bir nesnenin alanları ise (`Dikdortgen`'in `en` ve `boy`'u gibi) otomatik olarak varsayılan değer alır: sayılar için 0, `boolean` için `false`. Yerel değişkenler için böyle bir varsayılan değer yoktur.

2.1.6.2 Atama ile karşılaştırma

Java'da `=` atamadır, `==` karşılaştırmadır. Matematikteki eşittir işareti her iki anlamda da kullanıldığı için bu iki işleç sık karıştırılır. Bir koşulun içine yanlışlıkla `=` yazmak tipik bir hatadır:

```
int x = 100;
if (x = 100) {
    System.out.println("yüz");
}
```

```
EsitMi.java:4: error: incompatible types: int cannot be converted to boolean
        if (x = 100) {
              ^
1 error
```

`if` deyiminin parantezi içinde bir `boolean` değer beklenir. `x = 100` ise bir atamadır ve değeri atanan değer, yani `int` türünde 100'dür. Java bir `int`'i `boolean`'a çevirmez, bu yüzden satır derlenmez. Doğrusu `if (x == 100)`'dür. C dilinde tam sayılar koşul olarak kullanılabildiği için aynı hata derlenir ve koşul her zaman doğru çıkar; Java'nın tip denetimi bu hatayı derlemede yakalar.

Karşılaştırma işleçlerinin ürettiği değerler `boolean` değişkenlerde saklanabilir: `boolean gecti = not >= 50;`. Bu satırda önce `not >= 50` karşılaştırması yapılır (öncelik tablosunda `>=`, `=`'den yüksektir), sonra sonucu `gecti`'ye atanır. Böyle bir değişken bir `if` deyiminde doğrudan kullanılabilir: `if (gecti)`.

2.1.6.3 Statik ve dinamik tipleme

Her programlama dili değerlerin tipleriyle ilgilenir; diller arasındaki fark, tiplerin ne zaman denetlendiğindedir.

> **Tanım 2.1.16**
>
> **Statik tipli** dillerde (Java, C, C#) değişkenin tipi açıkça belirtilir, değişmez ve tip uyumu derleme zamanında denetlenir. **Dinamik tipli** dillerde (Python, JavaScript) değişkenin tipi önceden belirtilmez; değer atanınca belirlenir ve çalışma zamanında değişebilir. Bu iki yaklaşıma **statik tipleme** ve **dinamik tipleme** (*static / dynamic typing*) denir.

Farkı aynı fikrin iki dildeki yazımı gösterir. Python'da bir değişkene önce sayı, sonra dizgi atanabilir:

```
x = 5          # x şimdi int
x = "beş"      # artık str: sorun yok
print(x + 1)   # çalışırken:
# TypeError: can only concatenate
#   str (not "int") to str
```

İlk iki satır sorunsuz çalışır. Hata ancak üçüncü satır çalıştırılınca, bir dizgiye sayı eklenmeye çalışılınca ortaya çıkar. Bu satır programın sonunda ya da nadiren girilen bir dalda olsaydı, hata belki haftalarca fark edilmezdi. Java'da aynı fikir derlenmez:

```
int x = 5;
x = "beş";     // derlenmez:
// error: incompatible types:
//   String cannot be converted to int
```

`x`, `int` olarak bildirilmiştir ve bu tip değişmez. İkinci satırdaki atama derlemede reddedilir; program hiç çalışmaz.

	Dinamik tipli (Python, JavaScript)	Statik tipli (Java, C, C#)
Tip bildirimi	Önceden belirtilmez; değer atanınca belirlenir	Açıkça belirtilir ve değişmez
Tip değişebilir mi?	Çalışma zamanında değişebilir	Hayır
Uyumsuz işlem	Çalışırken hata (`TypeError`)	Derlemede hata
Kazanç	Esneklik, kısa kod	Güvenlik; derleyici hızlı kod üretebilir

Tablo 2.1.10: Dinamik ve statik tiplemenin karşılaştırması.

Statik tiplemenin bir kazancı da hızdır. Derleyici `a + b` ifadesindeki iki değişkenin `int` olduğunu bildiği için doğrudan bir tam sayı toplama komutu üretir. Dinamik tipli bir dilde ise her toplamada işlenenlerin o anki tipine bakılması gerekir: iki sayı mı, iki dizgi mi, yoksa uyumsuz bir çift mi?

Java 10'dan bu yana yerel değişkenler `var` ile de bildirilebilir. Bu, Java'yı dinamik tipli yapmaz:

```
var adet = 3;          // tip: int
var oran = adet / 2;   // tip: int, değer 1
var ad = "Ali";        // tip: String
adet = "üç";           // derlenmez: String cannot be converted to int
```

`var` ile tip yazılmaz, ama derleyici tipi sağ taraftan çıkarır ve derlemede sabitler. `adet`'in tipi `int`'tir; ona daha sonra bir dizgi atanamaz. `var` yalnız yazımı kısaltır; tip denetimi aynen sürer. Bu yüzden `var` kullanılan bir değişkene ilk değer vermek zorunludur: `var x;` derlenmez, çünkü derleyici tipi çıkaracağı bir değer bulamaz.

Sık yapılan hata

`var`'ın değişkeni "her tipten değer alabilir" yaptığını sanmak. `var oran = adet / 2;` satırında `adet` 3 ise `oran`'ın tipi `int`, değeri 1'dir. `var` ondalıklı bir tip seçmez; sağ taraf ne üretiyorsa onu alır. Ondalıklı sonuç isteniyorsa ya `double oran = adet / 2.0;` yazılır ya da sağ taraf `double` üretecek biçimde düzenlenir.

2.1.7 Dizgiler ve tipler

Bir program yalnız sayılarla çalışmaz. Kullanıcıya gösterilen her ileti, okunan her ad, ekrana yazılan her sonuç bir dizgidir. Sayılar ile dizgiler aynı ifadede buluştuğunda Java'nın kuralları kesin ama ilk bakışta şaşırtıcıdır. Bu kesim o kuralları işler.

Tanım 2.1.17

Dizgi (*string*), karakterlerden oluşan bir metindir. Java'da dizgiler `String` sınıfının nesneleridir ve kaynak kodda çift tırnak içinde yazılır: `"Merhaba"`. Tek tırnak içindeki tek bir karakter (`'a'`) ise dizgi değil, `char` tipinde bir değerdir.

2.1.7.1 Farklı tipler karşılaştırılmaz

Farklı veri tiplerini doğrudan karşılaştırmak hataya yol açar. Bir tam sayı ile bir dizgi karşılaştırılamaz; `5` ile `"5"` insan gözüne aynı görünse de biri bir sayı, öteki bir karakter dizisidir.

```
int sayi = 5;
String metin = "5";
boolean b = sayi == metin;
```

```
error: bad operand types for binary operator '=='
  first type:  int
  second type: String
```

Diller, tiplerin karşılaştırılması için kurallar ve **tip dönüşümü** (*type conversion*) yolları tanımlar. Karşılaştırma ancak iki değer aynı tipe getirildikten sonra anlamlıdır. Bunun iki yönü vardır: dizgi sayıya çevrilir ya da sayı dizgiye çevrilir.

```
int sayi = 5;
String metin = "5";
System.out.println(Integer.parseInt(metin) == sayi);
System.out.println(metin.equals(String.valueOf(sayi)));
System.out.println(metin.equals(sayi));
```

```
true
true
false
```

- `Integer.parseInt(metin)`, `"5"` dizgisini `int` türündeki 5'e çevirir. İki `int` artık `==` ile karşılaştırılabilir: `true`.
- `String.valueOf(sayi)`, 5 sayısını `"5"` dizgisine çevirir. İki dizgi `equals` metoduyla karşılaştırılır: `true`.
- `metin.equals(sayi)` derlenir, ama her zaman `false` döndürür. `equals` her türden nesneyi parametre olarak kabul eder; `sayi` burada otomatik olarak bir `Integer` nesnesine sarılır. Bir `String` hiçbir zaman bir `Integer`'a eşit değildir.

Üçüncü satır en tehlikeli olanıdır, çünkü derleyici uyarmaz. Program çalışır ve yanlış bir cevap verir; bu bir mantık hatasıdır. `parseInt` da dikkat ister: dizgi bir tam sayı değilse, örneğin `"5a"` ya da `"5.0"` ise, çalışma zamanında `NumberFormatException` fırlatır.

Dizgilerin karşılaştırılmasında bir kural daha vardır: dizgiler `==` ile değil `equals` ile karşılaştırılır. `==` iki referansın aynı nesneyi gösterip göstermediğine bakar; `equals` ise iki dizginin karakterlerinin aynı olup olmadığına. İki ayrı nesne aynı metni taşıyabilir. Bu durumda `==` `false`, `equals` `true` verir. Kaynak kodda yazılan aynı sabit dizgiler Java'da tek bir nesne olarak saklandığı için `==` bazen yanlışlıkla doğru sonuç da verir. Bu tesadüfe güvenilmemelidir.

2.1.7.2 Farklı tiplerde işlemler

Aynı işleç farklı tipler üzerinde farklı davranır. En belirgin örnek `+`'dır: sayılar üzerinde toplama, dizgiler üzerinde **birleştirme**dir.

Tanım 2.1.18

Dizgi birleştirme (*string concatenation*), `+` ile dizgileri uç uca eklemektir. `+`'nın işlenenlerinden biri dizgiyse öteki de dizgiye çevrilir ve sonuç bir dizgidir. **Genişletme dönüşümü** (*widening conversion*) ise küçük bir sayı tipinin daha büyük bir tipe otomatik çevrilmesidir: `int` → `double`.

İfade	Java sonucu	Tip	Açıklama
`3.0 + 4.0`	`7.0`	`double`	iki ondalıklı sayı
`3 + 4`	`7`	`int`	iki tam sayı
`3 + 4.0`	`7.0`	`double`	`int` otomatik olarak `double`'a genişletilir
`"3" + "4"`	`"34"`	`String`	birleştirme
`3 + "4"`	`"34"`	`String`	bir taraf dizgiyse öteki dizgiye çevrilir; hata değildir
`3 + true`	derlenmez	—	`bad operand types for binary operator '+'`
`'a' + 1`	`98`	`int`	`char` bir sayısal koddur (`'a'` = 97)

Tablo 2.1.11: Farklı tiplerde `+` işlemi (JDK 21 ile çalıştırılarak doğrulanmıştır).

Tablonun her satırı bir kuralı gösterir. İki `int`'in toplamı `int`'tir. Bir `int` ile bir `double` toplanırken `int` önce `double`'a genişletilir; genişletmede bilgi kaybolmaz, çünkü her `int` değeri bir `double`'da tam olarak yazılabilir. Ters yön otomatik değildir. `int n = 4.5;` satırı derlenmez ve derleyici `possible lossy conversion from double to int` der: `double`'dan `int`'e geçişte kesir kaybolabilir ve Java bu kaybı programcının açıkça istemesini bekler.

`3 + "4"` satırı dillerin farklı tercihlerini gösterir. Java bunu hata saymaz ve 3'ü `"3"`'e çevirip birleştirir. Python'da `3 + "4"` bir `TypeError` fırlatır; Python sayıyı kendiliğinden dizgiye çevirmez. JavaScript ise Java gibi birleştirir. Hiçbiri yanlış değildir; her dilin kuralı farklıdır ve programcı kullan-

dığı dilin kuralını bilmelidir.

`3 + true` satırı Java'nın bir başka tercihidir: `boolean` bir sayı değildir. C'de doğru değeri 1 sayılır ve bu toplam 4 verir; Java ise satırı derlemez. Son satırdaki `'a' + 1` ise şaşırtıcı görünse de kurala uyar. `char` tipi bir karakteri Unicode kod numarasıyla saklar; `'a'`'nın kodu 97'dir. Aritmetik işlemde `char`, `int`'e genişletilir ve 97 + 1 = 98 bulunur. Sonucu karakter olarak görmek için açık dönüşüm gerekir: `(char) ('a' + 1)` ifadesi `'b'` verir.

Farklı tipler arasında işlem, dilin kuralları bilinmiyorsa beklenmeyen sonuçlara ya da hatalara yol açar. Java'da bu kurallar kesindir ve derleyici onları her ifadeye aynı biçimde uygular. Kuralları bilen bir programcı için Java'nın davranışı her zaman öngörülebilirdir.

2.1.7.3 Birleştirme soldan sağa

`+` işleci soldan sağa işler ve her adımda yalnız o adımın iki işlenenine bakar. Bir ifadede hem sayılar hem dizgiler varsa sonuç, dizginin hangi konumda olduğuna bağlıdır:

İfade	Adımlar	Çıktı
`"Toplam: " + 2 + 3`	`"Toplam: 2"` + 3	`Toplam: 23`
`"Toplam: " + (2 + 3)`	parantez önce: 5; `"Toplam: " + 5`	`Toplam: 5`
`2 + 3 + " toplam"`	2 + 3 = 5; `5 + " toplam"`	`5 toplam`
`1 + 2 + "3" + 4 + 5`	3; `"33"`; `"334"`; `"3345"`	`3345`
`"" + 'A' + 'B'`	`"A"`; `"AB"`	`AB`
`'A' + 'B'`	65 + 66	`131`

Tablo 2.1.12: Sayı ve dizgi karışık ifadeler.

İlk satır, sık yapılan bir hatanın kaynağıdır. Programcı “Toplam: 5” yazdırmak ister, ama Java soldan başlar: `"Toplam: " + 2` bir dizgidir. Bu dizgiye 3 eklenince yine birleştirme yapılır ve “Toplam: 23” çıkar. Sayıların önce toplanması için parantez gerekir. Üçüncü satırda ise dizgi en sondadır. Soldaki iki `int` toplanır ve 5 bulunur; birleştirme ancak son adımda yapılır.

Dördüncü satır kuralın genel hâlini gösterir. İlk dizgi görülene kadar sayılar toplanır; ilk dizgiden sonra her `+` birleştirmedir. Son iki satır `char` için aynı kuralı uygular. `'A' + 'B'` ifadesinde dizgi yoktur; iki karakter sayı olarak toplanır ve 65 + 66 = 131 bulunur. Başa boş bir dizgi (`""`) eklemek ifadeyi en baştan birleştirmeye çevirir.

2.1.7.4 Girdide ondalık ayırıcı

Tipler arası dönüşümün bir yüzü de kullanıcıdan okunan girdidir. Klavyeden gelen her şey önce metindir; `Scanner` sınıfı bu metni istenen tipe çevirir. `nextDouble()` metodu bir ondalıklı sayı okurken bilgisayarın **yerel ayarını** (*locale*) kullanır. Türkçe yerel ayarda ondalık ayırıcı virgüldür.

```
Scanner giris = new Scanner(System.in);   // yerel ayar: Türkçe
double boy = giris.nextDouble();
// girdi "1,75" -> 1.75
// girdi "1.75" -> java.util.InputMismatchException
```

Türkçe bölge ayarlı bir bilgisayarda kullanıcı `1,75` yazarsa değer 1.75 olarak okunur. `1.75` yazarsa `Scanner` bunu bir ondalıklı sayı olarak tanımaz ve `InputMismatchException` fırlatır; program durur. Kaynak koddaki sabitler (`65.2` gibi) ve `println` 'ın çıktısı ise yerel ayardan etkilenmez; her zaman nokta kullanılır. Bu yüzden aynı program girdide virgül ister, çıktıda nokta yazar.

Sorunu çözmenin iki yolu vardır. Birincisi, kullanıcıya gösterilen istemdeki örneği yerel ayara uydurmaktır: "Boy (örn: 1,75)". İkincisi, `Scanner` 'ın yerel ayarını sabitlemektir: `giris.useLocale(Locale.US);` satırından sonra girdi her bilgisayarda noktayla okunur. Bu ayrıntı, aynı programın bir bilgisayarda çalışıp ötekinde çökmesinin sık nedenlerinden biridir.

2.1.8 Algoritmadan programa

Önceki kesimler bir programın yapı taşlarını tek tek ele aldı: değerler, tipler, ifadeler, değişkenler. Bu kesim onları bir araya getirir. Bir program bu yapı taşlarından kurulmuş bir talimatlar dizisidir; talimatların sırası da en az kendileri kadar önemlidir.

2.1.8.1 Program bir talimatlar dizisidir

Bilgisayar talimatları birer birer ve sırayla işler. Bir talimat bittikten sonra bir sonraki başlar ve her talimat bir öncekinin bıraktığı durumdan devam eder. Bu yüzden işlem sırasını doğru belirlemek önemlidir. Klasik bir örnek, iki değişkenin değerlerini yer değiştirmektir; buna **takas** denir.

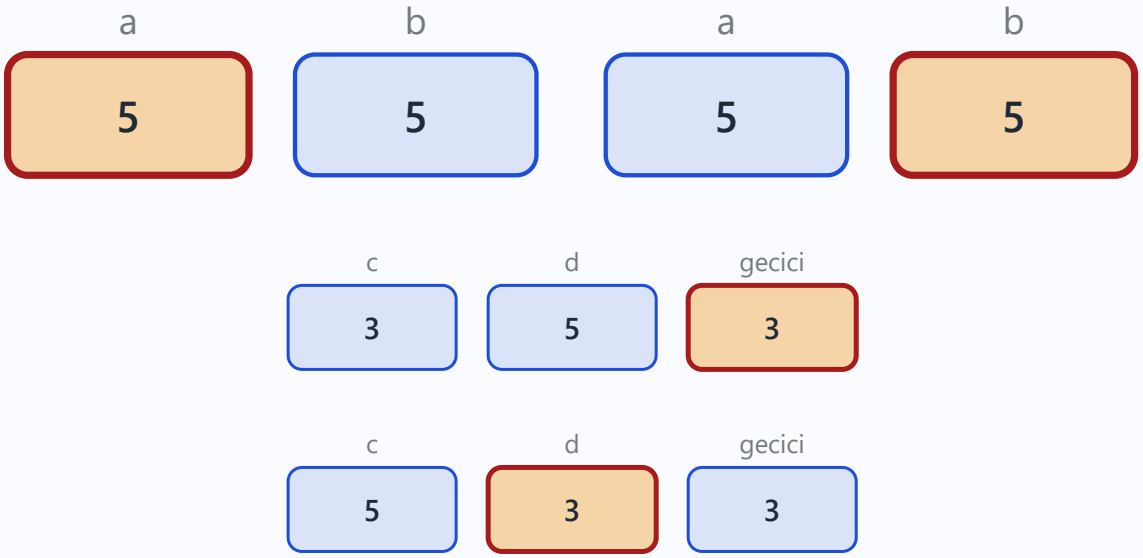
```
int a = 3, b = 5;
a = b;
b = a;
System.out.println(a + " " + b);

int c = 3, d = 5;
int gecici = c;
c = d;
d = gecici;
System.out.println(c + " " + d);
```

Program iki satır yazar: `5 5` ve `5 3` . İlk takas başarısızdır, ikincisi başarılıdır. Tablo 2.1.13 iki parçayı satır satır izler.

Satır	Deyim	Değişkenler	Çıktı
1	`int a = 3, b = 5;`	`a` = 3, `b` = 5	
2	`a = b;`	`a` = 5, `b` = 5 (eski 3 kayboldu)	
3	`b = a;`	`a` = 5, `b` = 5	
4	`println`		`5 5`
6	`int c = 3, d = 5;`	`c` = 3, `d` = 5	
7	`int gecici = c;`	`c` = 3, `d` = 5, `gecici` = 3	
8	`c = d;`	`c` = 5, `d` = 5, `gecici` = 3	
9	`d = gecici;`	`c` = 5, `d` = 3, `gecici` = 3	
10	`println`		`5 3`

Tablo 2.1.13: Takas programının izlemesi.



Şekil 2.1.8: Takas izlemesinden dört an. (a) `a = b` : `a` 'ya 5 yazılır, eski değer 3 kaybolur. (b) `b = a` : `b` 'ye `a` 'nın şimdiki değeri (5) yazılır; takas başarısızdır. (c) Doğru yöntem: `c` 'nin değeri önce `gecici` 'de saklanır. (d) `d` 'ye saklanan eski `c` değeri (3) yazılır; takas başarılıdır.

İlk parçanın hatası, atamanın eski değeri silmesindedir. `a = b;` çalıştığında `a` 'nın 3 değeri kaybolur ve hiçbir yerde saklanmamıştır. Sonraki `b = a;` satırı `a` 'nın şimdiki değerini, yani 5'i kopyalar. Şekil 2.1.8'in (a) ve (b) kareleri bu kaybı gösterir. Satırların sırası değiştirilse de sonuç düzelmez: önce `b = a;` yazılırsa bu kez `b` 'nin değeri kaybolur ve çıktı `3 3` olur.

Doğru yöntem, üzerine yazılacak değeri önceden üçüncü bir **geçici** değişkende saklamaktır. `gecici = c` satırı 3'ü güvene alır. Ardından `c` 'nin üzerine güvenle yazılabilir. Son satır, saklanan değeri `d` 'ye taşır. Takas üç atamayla yapılır ve sıraları değiştirilemez. Maliyeti sabittir: kaç değer olursa olsun bir takas üç atama ve bir ek değişken gerektirir.

Örnek 2.1.9

Geçici değişken olmadan da tam sayılar takas edilebilir. `a` = 4, `b` = 9 için şu üç satır izlensin: `a = a + b; b = a - b; a = a - b;`

1. `a` = 4 + 9 = 13, `b` = 9. 2. `b` = 13 − 9 = 4. 3. `a` = 13 − 4 = 9.

Sonuçta `a` = 9, `b` = 4'tür. Yöntem çalışır, ama iki sakıncası vardır. Okunması zordur ve `a + b` toplamı `int` 'in sınırını aşarsa taşma olur. Geçici değişkenli yöntem her tip için çalışır ve niyeti açıkça gösterir.

2.1.8.2 İfadeler, deyimler ve programlar

Bir programın parçaları iki türlüdür: değer üretenler ve etki yaratanlar.

Tanım 2.1.19

İfade bir değeri hesaplar ve sonucunu döndürür: `3 + 4` ifadesinin değeri 7'dir. **Deyim** (*statement*) ise bir etki yaratır: `pi = 3.14159;` bir değişkene değer atar, `System.out.println(pi);` ekrana yazar. Java'da deyimler genellikle `;` ile biter. Bir program, sırayla çalışan deyimlerden oluşur.

İfadeler iç içe geçebilir ve deyimlerin içinde yer alabilir. `double alan = pi * r * r;` bir deyimdir; içinde `pi * r * r` ifadesi vardır, o ifade de `pi * r` ifadesini içerir. Tersi geçerli değildir: bir deyim, bir ifadenin içinde yer alamaz. `println` ekrana yazar, ama bir değer döndürmez; dönüş tipi `void` 'dir. Bu yüzden onu bir hesabın içinde kullanmak anlamsızdır:

```
double d = 3 + System.out.println(pi);
```

```
error: 'void' type not allowed here
```

Tersine, değer üreten ama hiçbir etki yaratmayan bir ifade de tek başına deyim olamaz:

```
3 + 4;      // değer üretir ama etkisi yok
```

```
error: not a statement
```

Her ifade sonuna `;` konarak deyime dönüşmez. Java buna yalnız dört ifade türünde izin verir: atama (`x = 5;`), artırma ve azaltma (`i++;`), metot çağrısı (`println(x);`) ve nesne oluşturma (`new Dikdortgen(1, 2);`). Bunların ortak yanı bir etki yaratmalarıdır. `3 + 4;`, `x * 2;` ya da `x > 3;` hesaplanır ve sonuç hemen atılır; bu boşa yapılmış bir işlemdir ve derleyici onu bir yazım yanlışı sayar.

Burada ince bir nokta vardır. Java'da atama da bir değer üretir: atanan değeri. Bu yüzden şu satır derlenir:

> `int y = (x = 4) + 1;` → `x` = 4, `y` = 5

Parantezdeki atama önce yapılır ve değeri 4 olur; buna 1 eklenir. Atamanın değer üretmesi, `x = y = 7;` gibi zincirleme atamaları da mümkün kılar. Yine de atamayı bir ifadenin ortasına gömmek okunmayı zorlaştırır ve tercih edilmez. Programın her satırında tek bir etki olması, izlemeyi de hata aramayı da kolaylaştırır.

Sık yapılan hata

`println`'ın yazdırdığı değeri bir değişkene almaya çalışmak: `int s = System.out.println(5);`. `println` ekrana yazar ama bir değer döndürmez; derleyici `incompatible types: void cannot be converted to int` iletisini verir. Bir hesabın sonucu hem kullanılacak hem yazdırılacaksa önce değişkene atanır, sonra yazdırılır.

2.1.8.3 Fahrenhayttan Celsius'a

Bir program yazmanın sırası bellidir: önce algoritma, sonra kod. Algoritma problemi kesin adımlara ayırır; kod bu adımları dilin kurallarıyla yazar. Fahrenhayt cinsinden verilen bir sıcaklığı Celsius'a çeviren bir program bu sırayla kurulsun. Dönüşüm formülü $C = (F - 32) \cdot 5 / 9$'dur.

Algoritma üç adımdır:

```
1. Fahrenhayt değeri f'yi oku.
2. c = (f − 32) · 5 / 9 değerini hesapla.
3. c'yi yazdır.
```

Her adımın Java'da bir karşılığı vardır. Okuma için `Scanner`, hesap için bir atama deyimi, yazdırma için `println` kullanılır:

```
import java.util.Scanner;

public class Fahrenhayt {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        System.out.print("Fahrenhayt: ");
        double f = giris.nextDouble();
        double c = (f - 32) * 5 / 9;
        System.out.println("Celsius: " + c);
    }
}
```

Kullanıcı 100 girdiğinde ekranda şu görünür:

```
Fahrenhayt: 100
Celsius: 37.77777777777778
```

Kod satır satır şöyle çalışır. `import java.util.Scanner;` satırı `Scanner` sınıfını kullanılabilir kılar. `new Scanner(System.in)` klavyeden okuyan bir nesne oluşturur. `print`, `println`'dan farklı olarak satır sonuna geçmez; bu yüzden kullanıcının girdiği 100 aynı satırda görünür. `giris.nextDouble()` bir sayı okur ve `f`'ye atar. Hesap satırında önce `f - 32` = 68.0 bulunur; `f` `double` olduğu için bu sonuç da `double`'dır. Ardından 68.0 · 5 = 340.0 ve 340.0 / 9 = 37.77777777777778 hesaplanır.

Formülün yazılış biçimi burada da önemlidir. Aynı formül `5 / 9 * (f - 32)` diye yazılırsa sonuç 0.0 olur. Öncelik kuralları değişmemiştir: `5 / 9` soldadır ve önce yapılır. İki işlenen de `int` olduğu için tam sayı bölmesi 0 verir ve 0 · 68.0 = 0.0 bulunur. Derleyici bunu bir hata olarak görmez; program 100 °F için 0 °C yazar. Düzeltmenin iki yolu vardır: bölmeyi ondalıklı yapmak (`5.0 / 9 * (f - 32)`) ya da çarpmayı öne almak (`(f - 32) * 5 / 9`). İkincisi, `f` `double` olduğu sürece doğru çalışır; `f - 32` `double` ürettiği için sonraki bölme de gerçek bölmedir.

Yazılış	`f` `double`	`f` `int`
`(f - 32) * 5 / 9`	`37.77777777777778`	`37` (340 / 9, kesir atılır)
`5 / 9 * (f - 32)`	`0.0`	`0`
`5.0 / 9 * (f - 32)`	`37.77777777777778`	`37.77777777777778`

Tablo 2.1.14: Fahrenhayt-Celsius formülünün farklı yazılışları, $F = 100$ için.

Tablonun son sütunu, tipin sonucu nasıl değiştirdiğini gösterir. `f` `int` olsa bile `5.0` ifadeye bir `double` soktuğu için sonuç doğru çıkar. Program yazmanın sırası bu küçük örnekte bile görünür: algoritma doğrudur, Java koduna çeviri sırasında yapılan küçük bir yazım farkı sonucu bozar. Bu tür hataları bulmanın yolu, sonucu bilinen bir girdiyle denemektir. 212 °F'nin 100 °C, 32 °F'nin 0 °C olduğu bilinir; program bu iki değeri doğru çeviriyorsa formül büyük olasılıkla doğru yazılmıştır.

2.1.9 Popüler programlama dilleri

Programlama dilleri farklı görevler için farklı avantajlar sunar ve programcılar projenin ihtiyacına göre en uygun dili seçer. Dillerin ne kadar kullanıldığı her yıl anketler ve dizinlerle izlenir. Stack Overflow'un geliştirici anketi, TIOBE dizini ve IEEE Spectrum'un sıralaması bunların bilinenleridir. Her biri farklı bir ölçüt kullanır: ankete katılanların beyanı, arama motoru sonuçları ya da iş ilanları. Bu yüzden sıralamalar birbirinden biraz farklıdır, ama ilk sıralarda hep aynı diller görülür.

Dil	Öne çıktığı alan
Python	veri analizi, yapay zekâ, web geliştirme
Java	büyük ölçekli kurumsal uygulamalar, Android
JavaScript	web tarayıcıları, web geliştirme
C#	Windows uygulamaları, oyun geliştirme
C++	oyun, sistem programlama, performans odaklı yazılım
Rust	güvenli ve hızlı sistem programlama
Swift	Apple platformlarında uygulama geliştirme
Kotlin	Android uygulama geliştirme (JVM üzerinde çalışır)
Go (Golang)	basit ve etkili; ağ ve bulut uygulamaları

Tablo 2.1.15: 2023'te öne çıkan dillerden bazıları ve öne çıktıkları alanlar.

Tablo önceki kesimlerdeki sınıflandırmalarla birlikte okunabilir. C, C++, Rust ve Go doğrudan makine koduna derlenir ve performansın önemli olduğu yerlerde öne çıkar. Java, Kotlin ve C# bir ara koda derlenip bir sanal makinede çalışır; taşınabilirlik ile hızı dengeler. Python ve JavaScript genellikle yorumlanır ve hızlı geliştirme gereken yerlerde tercih edilir. Kotlin'in Java ile aynı JVM'de çalışması, iki dilin aynı projede birlikte kullanılabilmesini sağlar.

Mühendislik açısından tek bir “en iyi dil” yoktur. Bir mikrodenetleyicinin yazılımı için C, bir sinyal işleme prototipi için MATLAB ya da Python, bir masaüstü ölçüm uygulaması için Java uygun düşebilir. Bir dilde öğrenilen kavramlar, yani tipler, ifadeler, öncelik, atama ve deyim sırası, öteki dillere büyük ölçüde taşınır. Yeni bir dil öğrenmek çoğu zaman bu kavramların o dildeki yazılışını ve küçük kural farklarını öğrenmektir.

2.1.10 Sık yapılan hatalar

Programlamaya yeni başlayanların hataları büyük ölçüde aynı birkaç kaynaktan gelir. Aşağıdaki sekiz hata, bu notta ayrıntılı ele alınan kuralların özetidir. Her birinde derleyicinin ya da JVM'in tepkisi JDK 21 ile alınmıştır.

Hata	Örnek	Belirti	Düzeltme
Noktalı virgülü unutmak	`System.out.println("Merhaba")`	`';' expected` ; `^` satırın sonunu gösterir	deyimin sonuna `;`
`=` ile `==`'yi karıştırmak	`if (x = 100)`	`int cannot be converted to boolean`	`if (x == 100)`
Tam sayı bölmesi	`(a + b) / 2` ile 7 ve 8'in ortalaması	7 (beklenen 7,5); derleyici uyarmaz	`(a + b) / 2.0`
Üs için `**` ya da `^`	`2 ** 3` ; `2 ^ 3`	`illegal start of expression` ; `^` derlenir ama XOR'dur (1)	`Math.pow(2, 3)`
Dizgi + sayı sırası	`"Toplam: " + 2 + 3`	`Toplam: 23`	`"Toplam: " + (2 + 3)`
Büyük/küçük harf	`Sayi` tanımlayıp `sayi` kullanmak	`cannot find symbol`	adı tanımlandığı gibi yazmak
Değer atamadan kullanmak	`int s;` `System.out.println(s);`	`variable s might not have been initialized`	bildirimde ilk değer: `int s = 0;`
Ondalık ayırıcı	Türkçe yerel ayarda `nextDouble()`'a `7.5` girmek	`InputMismatchException`	`7,5` girmek ya da `useLocale(Locale.US)`

Tablo 2.1.16: Sık yapılan hatalar, belirtileri ve düzeltmeleri.

Tablodaki hatalar üç gruba ayrılır. Noktalı virgül, büyük/küçük harf, ilk değer ve `=` / `==` hataları derleme hatasıdır; derleyici yakalar ve iletisi hatanın yerini gösterir. Ondalık ayırıcı hatası bir çalışma zamanı hatasıdır; program derlenir ama belli bir girdiyle çöker. Tam sayı bölmesi, `^` ile üs ve birleştirme sırası ise mantık hatasıdır. Bu sonuncular en tehlikeli olanlardır, çünkü program hiçbir belirti göstermeden yanlış bir sonuç üretir. Onları bulmanın tek yolu, sonucu bilinen girdilerle denemektir.

Derleme hatalarıyla ilgili genel bir alışkanlık da işe yarar: ileti dikkatle okunur ve düzeltmeye ilk hatadan başlanır. `int = x 5;` örneğinde olduğu gibi, tek bir yanlış bir dizi iletiye yol açabilir. İlk hata düzelince ötekiler çoğu zaman kendiliğinden kaybolur. Hata iletisinin gösterdiği satırda bir sorun görünmüyorsa bir önceki satıra da bakılır; özellikle kapanmamış bir parantez ya da tırnak, hatayı birkaç satır ileriye taşıyabilir.

2.1.11 Özet

- Programlama dili, bir makinenin yapabileceği hesaplamaları ifade etmek için tasarlanmış yapay bir dildir. Program hem makineye talimat hem de programcılar arasında bir iletişim aracıdır.
- Diller dört ölçüte göre sınıflandırılır: soyutlama düzeyi (düşük ya da yüksek düzey), uygulama alanı (genel amaçlı ya da hedefe odaklı), yürütme biçimi (derlenen ya da yorumlanan) ve paradigma.
- Java'da `javac` kaynak kodu bytecode'a (`.class`) derler; JVM bytecode'u yorumlar ve sık çalışan kısımları JIT ile makine koduna çevirir. Aynı `.class` dosyası JVM olan her platformda çalışır.
- Dört temel paradigma buyruksal, fonksiyonel, mantıksal ve nesne yönelimlidir. Java temelde nesne yönelimli ve buyruksaldır; Java 8'den bu yana fonksiyonel öğeler de içerir.

- Söz dizimi biçim kurallarıdır. Statik anlam, söz dizimi doğru kodun derlemede denetlenen anlam kurallarıdır (tip uyumu). Anlam ise kodun ne yaptığıdır; tek ve kesindir, ama programcının beklediği olmayabilir.
- Hatalar dört türlüdür: söz dizimi ve statik anlam hataları derlemede, çalışma zamanı hataları çalışırken istisnayla, mantık hataları ancak testle bulunur.
- İfadeler önce parantez, sonra `* / %`, sonra `+ -` sırasıyla, aynı düzeyde soldan sağa değerlendirilir. İki tam sayının bölümü tam sayıdır ve sıfıra doğru kesilir. Java'da üs işleci yoktur; `Math.pow` kullanılır.
- Atama iki adımdır: önce sağ taraf şu anki değerlerle hesaplanır, sonra soldaki değişkene yazılır. `=` atama, `==` karşılaştırmadır.
- Java statik tiplidir: tipler derlemede denetlenir ve değişmez. `var` bile tipi derlemede sabitler. `int` ile `double`'ın işlemi `double`'dır. Bir işlenen dizgiyse `+` birleştirmedir ve soldan sağa işler.
- Dizgiler `equals` ile karşılaştırılır. Farklı tipler önce aynı tipe dönüştürülür: `Integer.parseInt`, `String.valueOf`.
- Program sırayla çalışan deyimlerden oluşur. İfade değer üretir, deyim etki yaratır. Deyimlerin sırası sonucu belirler; takas üçüncü bir geçici değişkenle yapılır.
- Bir program önce algoritma olarak tasarlanır, sonra koda çevrilir. Çeviride tip ve öncelik ayrıntıları sonucu değiştirebilir.

2.1.12 Kavram listesi

programlama dili *programming language* · kaynak kod *source code* · assembly dili *assembly language* · derleyici *compiler* · yorumlayıcı *interpreter* · bytecode (ara kod) *bytecode* · Java Sanal Makinesi (JVM) *Java Virtual Machine* · JIT (tam zamanında) derleyici *just-in-time compiler* · programlama paradigması *programming paradigm* · nesne yönelimli programlama *object-oriented programming* · alan *field* · yapıcı *constructor* · referans *reference* · söz dizimi *syntax* · statik anlam *static semantics* · anlam *semantics* · çalışma zamanı hatası *runtime error* · mantık hatası *logic error* · istisna *exception* · ifade *expression* · işlenen *operand* · işleç önceliği *operator precedence* · tam sayı bölmesi *integer division* · değişken *variable* · atama *assignment* · tanımlayıcı *identifier* · statik / dinamik tipleme *static / dynamic typing* · genişletme dönüşümü *widening conversion* · tip dönüşümü *type conversion* · dizgi *string* · dizgi birleştirme *string concatenation* · deyim *statement*

2.1.13 Kendini Dene

1. `17 / 5 * 5 + 17 % 5` ve `17 / 5.0` ifadelerinin değerleri ve tipleri nedir?

   Cevap

   Birincisi: `17 / 5` = 3 (tam sayı bölmesi), 3 · 5 = 15, `17 % 5` = 2, toplam **17**, tipi `int` 'tir. Bu sonuç rastlantı değildir: `(a / b) * b + a % b` her zaman `a` 'ya eşittir. İkincisi: 5.0 `double` olduğu için gerçek bölme yapılır; **3.4**, tipi `double` 'dır.

2. `System.out.println(2 * 3 + "x" + 2 * 3);` ne yazdırır?

   Cevap

   `6x6`. `*` işlecinin önceliği `+` 'dan yüksektir; iki çarpma, birleştirmelerden önce yapılır ve ifade `6 + "x" + 6` olur. Soldan sağa: `6 + "x"` = `"6x"`, ardından `"6x" + 6` = `"6x6"`. Çarpmalar parantezsiz de doğru çalışır; sorun yalnız `+` ile toplama yapılmak istendiğinde çıkar.

3. Her satır ayrı bir programda olsun. Hata türü (söz dizimi, statik anlam, çalışma zamanı, mantık) nedir? (a) `boolean b = 1;` (b) `int n = 5; int y = 10 / (n - 5);` (c) Dairenin çevresi için `double cevre = 2 * pi + r;`

   Cevap

   (a) Statik anlam: `incompatible types: int cannot be converted to boolean`. Java'da `boolean` bir sayı değildir. (b) Çalışma zamanı: kod derlenir, ama `n - 5` = 0 olduğu için `ArithmeticException: / by zero` fırlatılır. (c) Mantık: kod derlenir ve çalışır, ama formül yanlıştır; doğrusu `2 * pi * r` 'dir.

4. Şu satırlar çalıştıktan sonra `x` ve `y` kaçtır? `int x = 6; int y = x / 4; x = x * y + x % 4; y = x - y;`

   Cevap

   `y` = 6 / 4 = 1 (tam sayı bölmesi). Üçüncü satırın sağ tarafı eski `x` (6) ile hesaplanır: 6 · 1 + 6 % 4 = 6 + 2 = 8; `x` = 8. Son satır: 8 − 1 = 7. Sonuç `x` **= 8,** `y` **= 7**.

5. `a` = 1, `b` = 2, `c` = 3 iken değerler sağa döndürülmek isteniyor: `a` ← `c`, `b` ← `a`, `c` ← `b`. Geçici

bir değişkenle hangi sırayla yazılır? Sonuç ne olur?

Cevap

Önce üzerine ilk yazılacak değer saklanır. `int g = c; c = b; b = a; a = g;` sırası doğrudur: `g` = 3, `c` = 2, `b` = 1, `a` = 3. Sonuç **3 1 2**'dir. Her atama, değeri daha önce başka bir yere kopyalanmış bir değişkenin üzerine yazar; hiçbir değer kaybolmaz.

6. `IntStream.range(2, 6).sum()` ne verir? Aynı hesabın buyruksal karşılığı nasıl yazılır?

Cevap

`range` üst sınırı dahil etmez: 2, 3, 4, 5 sayıları toplanır ve **14** bulunur. Buyruksal karşılığı: `int t = 0; for (int i = 2; i < 6; i++) t = t + i;`. Döngü koşulunda `<=` değil `<` kullanılır; `rangeClosed(2, 6)` olsaydı 6 da dahil olur ve 20 bulunurdu.

7. `int c = 37;` iken `c * 9 / 5 + 32`, `9 / 5 * c + 32` ve `c * 9 / 5.0 + 32` ifadeleri ne verir?

Cevap

Birincisi: 37 · 9 = 333, 333 / 5 = 66 (kesir atılır), 66 + 32 = **98**. İkincisi: 9 / 5 = 1 (tam sayı bölmesi), 1 · 37 + 32 = **69**; büyük bir hata. Üçüncüsü: 333 / 5.0 = 66.6, 66.6 + 32 = **98.6** (`double`), doğru değer budur.

8. `x` bir `int` değişken olsun. Aşağıdakilerden hangileri tek başına bir deyim olarak derlenir? `x++;`, `x + 1;`, `Math.pow(x, 2);`, `x == 3;`

Cevap

`x++;` derlenir: `x`'i bir artırır, yani bir etki yaratır. `Math.pow(x, 2);` de derlenir: metot çağrıları deyim olabilir, döndürülen değer atılır. `x + 1;` ve `x == 3;` derlenmez ve ikisi de `not a statement` iletisini verir; yalnız bir değer üretirler, etkileri yoktur.

2.1.14 İleri okuma

- Oracle, *The Java Tutorials*, "Language Basics" bölümü: değişkenleri, işleçleri, öncelik tablosunu, ifadeleri ve deyimleri kısa örneklerle anlatır. Java'nın resmî ve ücretsiz başlangıç kaynağıdır.
- A. B. Downey ve C. Mayfield, *Think Java* (2. baskı, 2019), Bölüm 1–2: programlama dilini, derleme ve yorumlamayı, hata türlerini ve hata iletilerini okumayı anlatır. Anlatım yeni başlayan okura göredir.
- J. V. Guttag, *Introduction to Computation and Programming Using Python* (3. baskı, 2021), Bölüm 2: söz dizimi, statik anlam ve anlam ayrımının kaynağıdır. Ayrımı Python örnekleriyle verdiği için dinamik tiplemeyle karşılaştırmaya da elverişlidir.

- J. Gosling ve diğerleri, *The Java Language Specification, Java SE 21 Edition*, Bölüm 15: ifadelerin değerlendirme sırasını, tam sayı bölmesini ve dizgi birleştirmeyi kesin kurallarla tanımlar. Bir ifadenin değeri konusunda son söz bu belgededir.

2.1.15 Kaynaklar

1. P. Deitel ve H. Deitel, *Java: How to Program, Early Objects*, 11. baskı, Pearson, 2017 (Bölüm 1–2).
2. A. B. Downey ve C. Mayfield, *Think Java: How to Think Like a Computer Scientist*, 2. baskı, O'Reilly, 2019 (Bölüm 1–2).
3. J. Gosling ve diğerleri, *The Java Language Specification, Java SE 21 Edition*, Oracle, 2023 (Bölüm 3: Sözcük yapısı, Bölüm 15: İfadeler).
4. J. Guttag, *Introduction to Computation and Programming Using Python*, 3. baskı, MIT Press, 2021 (Bölüm 2: söz dizimi, statik anlam ve anlam).
5. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson, 2022 (Bölüm 2–3).
6. Oracle, *The Java Tutorials*: “Language Basics” (değişkenler, işleçler, ifadeler, deyimler ve bloklar).
7. K. Sierra ve B. Bates, *Head First Java*, 3. baskı, O'Reilly, 2022.

KISIM I · BÖLÜM 2.2

Söz Dizimi Kuralları

Bir Java programı derleyiciye bir metin olarak verilir. Derleyici bu metni harf harf okur ve dilin kurallarına uyup uymadığına bakar. Tek bir noktalı virgül eksikse ya da bir harf büyük yerine küçük yazılmışsa program çalışmaz. Bu not, Java'nın yazım kurallarını ve bu kuralların arkasındaki çalışma düzenini anlatır. İlk kesim bir `.java` dosyasının `javac` ile bytecode'a, `java` komutuyla da çalışan bir programa dönüşmesini izler. JVM'in yığın üzerinde nasıl çalıştığı ve aynı `.class` dosyasının neden her işletim sisteminde çalıştığı burada açıklanır. Ardından değişkenler, sekiz ilkel tip ve bu tiplerin sınırları gelir. Tanımlayıcılar, anahtar sözcükler ve adlandırma kuralları adların nasıl seçileceğini belirler. Atama, sabit değerler ve tip dönüşümü kesimi, sayıların bir tipten ötekine geçerken nasıl değiştiğini gösterir. Son kesimler işleçleri, işleç önceliğini, kısa devre değerlendirmeyi, üçlü işleci ve kontrol deyimlerinin söz dizimini işler. Her kesimde sunumdaki kodlar satır satır anlatılır ve derleyicinin gerçek hata iletileri verilir.

ÖĞRENME ÇIKTILARI

Derleme ve çalıştırma

Bir `.java` dosyasının `javac` ile bytecode'a, `java` ile çalışan programa dönüşmesini anlatmak; soyutlama düzeylerini ve JVM yığınında bytecode'un yürüyüşünü izlemek; JVM'in platform bağımsızlığını nasıl sağladığını açıklamak.

Değişkenler ve tipler

Sekiz ilkel veri tipini boyut ve aralıklarıyla tanımak; değişken bildirmek, değer atamak, taşma ve ondalık hassasiyet sorunlarını fark etmek.

Kurallar ve adlandırma

Geçerli tanımlayıcıyı geçersizden ayırmak, anahtar sözcükleri tanımak, camelCase, PascalCase ve BÜYÜK_HARF kurallarına uymak.

İfadeler ve akış

İşleç önceliğine göre ifadeleri hesaplamak, tip dönüşümü yapmak; `?:` ile koşula göre değer seçmek; `if`, `switch`, `for`, `while`, `do-while`, `break` ve `continue` deyimlerinin söz dizimini tanımak.

2.2.1 Derleme ve çalıştırma

Java ile yazılan bir program, ekranda bir sonuç görünene kadar birkaç aşamadan geçer. Programcı kodu bir metin dosyasına yazar. Derleyici bu metni denetler ve ara bir koda çevirir. Java Sanal Makinesi ara kodu alır ve bilgisayarın işlemcisinde yürütür. Bu kesim aşamaları sırayla ele alır. Hata iletilerini doğru okumak ve "Çalıştır" düğmesinin arkasında ne olduğunu bilmek, ancak bu zinciri tanıyınca mümkündür.

2.2.1.1 Kaynak kod dosyası

Tanım 2.2.1

Kaynak kod (*source code*), programcının bir programlama dilinin kurallarına göre yazdığı, insanın okuyabildiği program metnidir. Java kaynak kodu **düz metin** (*plain text*) dosyalarında durur ve bu dosyaların uzantısı `.java`'dır.

Düz metin, yalnız karakterlerden oluşan bir dosyadır. İçinde yazı tipi, renk, kalın yazı ya da sayfa düzeni bilgisi yoktur. Word gibi bir kelime işlemcinin kaydettiği belge ise metnin yanında bu biçim bilgilerini de saklar. Derleyici bu ek bilgiyi anlamaz. Bu yüzden Java kodu Not Defteri, Notepad++ ya da VS Code gibi bir metin düzenleyiciyle ya da bir geliştirme ortamıyla yazılır.

Düz metin üç açıdan elverişlidir:

- **Okunaklıdır.** İnsan dosyayı açınca doğrudan kodu görür. Kod kolayca anlaşılır, e-postayla gönderilir, bir sürüm denetim sisteminde saklanır.
- **Taşınabilirdir.** Aynı `.java` dosyası her işletim sisteminde ve her geliştirme ortamında açılır.
- **Kolay düzeltilir.** Bir hatayı düzeltmek ya da kodu güncellemek için herhangi bir metin düzenleyici yeterlidir.

Java bir dosya adı kuralı da koyar. Bir dosyada `public` bir sınıf varsa dosyanın adı sınıfın adıyla birebir aynı olmalıdır. `public class MerhabaDunya` diye başlayan bir kod `MerhabaDunya.java` adlı dosyaya kaydedilir. Bu eşleşmede büyük ve küçük harf de önemlidir. Aynı kod `merhabadunya.java` adıyla kaydedilirse derleyici şu iletiyi verir:

```
merhabadunya.java:1: error: class MerhabaDunya is public, should be declared in a file
named MerhabaDunya.java
public class MerhabaDunya {
       ^
1 error
```

Windows'un dosya sistemi büyük ve küçük harfi ayırmasa da `javac` ayırır. İleti tam olarak neyin yanlış olduğunu söyler: `MerhabaDunya` sınıfı public'tir ve `MerhabaDunya.java` adlı bir dosyada bildirilmelidir.

2.2.1.2 Programlama araçları

Java programı yazmak için birkaç araç birlikte kullanılır. Tablo 2.2.1 bu araçları ve görevlerini özetler.

Araç	Ne işe yarar?	Örnekler
Metin düzenleyici	Kaynak kodu düz metin olarak yazmak	Notepad++, VS Code
JDK (*Java Development Kit*)	Derleyici, çalıştırıcı ve öteki geliştirme araçları	`javac`, `java`, `javap`, `jshell`
IDE (tümleşik geliştirme ortamı)	Düzenleme, derleme, çalıştırma ve hata ayıklama tek pencerede; hatalar yazılırken altı çizilerek gösterilir	Eclipse, IntelliJ IDEA, NetBeans, Java eklentileriyle VS Code
JShell (Java 9 ve sonrası)	Tek satırlık Java ifadelerini sınıf yazmadan denemek	`jshell> 7 / 2` → `3`

Tablo 2.2.1: Java ile program geliştirmede kullanılan araçlar.

JDK (*Java Development Kit*, Java geliştirme kiti), Java ile program geliştirmek için gereken her şeyi içeren pakettir. İçinde derleyici `javac`, programı başlatan `java`, bir `.class` dosyasının içini gösteren `javap` ve etkileşimli deneme ortamı `jshell` bulunur. **IDE** (*Integrated Development Environment*, tümleşik geliştirme ortamı) ise bu araçları bir düzenleyiciyle birleştirir. IDE'deki “Çalıştır” düğmesine basılınca arka planda yine `javac` ve `java` çalışır.

JShell küçük denemeler için çok kullanışlıdır. `7 / 2` ifadesinin neden 3.5 değil de 3 verdiğini merak eden biri, bir sınıf ve `main` metodu yazmadan ifadeyi JShell'e yazıp sonucu hemen görür. Başlangıçta bir kez komut satırında `javac` ve `java` kullanmak da yararlıdır. IDE'nin gizlediği adımlar böylece görünür hâle gelir ve bir hata çıktığında hangi aşamada çıktığı anlaşılır.

2.2.1.3 İlk program: MerhabaDunya

Aşağıdaki program, Java'da yazılabilecek en kısa tam programlardan biridir. Ekrana tek bir satır yazar.

```
public class MerhabaDunya {
    public static void main(String[] args) {
        System.out.println("Merhaba, Dünya!");
    }
}
```

Kısa olmasına karşın programın her parçası bir kurala bağlıdır:

- `public class MerhabaDunya` bir sınıf tanımlar. Her Java programı en az bir sınıf içerir. Sınıfın gövdesi, bu satırdaki `{` ile son satırdaki `}` arasında kalan kısımdır. Sınıf public olduğu için dosyanın adı `MerhabaDunya.java` olmalıdır.

- `public static void main(String[] args)` programın **giriş noktasıdır**. JVM programı başlatınca çalışmaya bu metottan başlar. Başlığın yazımı birebir böyle olmalıdır. `Main` ya da `main(String args)` yazılırsa program derlenir, ama JVM giriş noktasını bulamaz ve çalıştırmayı reddeder.
- `System.out.println(...)` parantez içindeki değeri konsola yazar ve alt satıra geçer. Metinler çift tırnak içinde yazılır: `"Merhaba, Dünya!"`.
- Satırın sonundaki **noktalı virgül**, deyimin bittiğini gösterir. Her deyim (*statement*) `;` ile biter. Süslü parantezle kapanan bloklardan sonra `;` gerekmez; bu yüzden sınıfın ve metodun kapanış parantezinden sonra noktalı virgül yoktur.

Programın gövdesi iç içe iki bloktan oluşur. Dıştaki blok sınıfın, içteki blok `main` metodunun gövdesidir. Girinti, yani satır başlarındaki boşluklar, bu iç içeliği göze görünür kılar. Derleyici girintiye bakmaz; yalnız süslü parantezleri sayar.

2.2.1.4 Temel söz dizimi kuralları

> **Tanım 2.2.2**
>
> **Söz dizimi** (*syntax*), bir dilin yazım kurallarıdır: hangi sözcüklerin, işaretlerin ve sıralamaların geçerli olduğunu belirler. Söz dizimi kurallarına uymayan Java kodu derlenmez.

Türkçede “kitabı okudum ben” cümlesi alışılmadık bir sıradadır ama anlaşılır. Derleyici bu kadar hoşgörülü değildir. Kurala uymayan bir satırı tahmin yoluyla düzeltmez, hata verip durur. Java'nın temel yazım kuralları şunlardır:

- Her deyim `;` ile biter.
- **Bloklar** (*block*) `{` ile açılır, `}` ile kapanır. Açılan her blok kapanmalıdır.
- Java **büyük-küçük harfe duyarlıdır** (*case-sensitive*): `sayi` ile `Sayi` iki ayrı addır; `System` yerine `system` yazılamaz.
- Boşluk, sekme ve satır sonu derleyici için önemsizdir. Bir deyim birkaç satıra bölünebilir. Okunabilirlik için girinti kullanılır.
- **Açıklamalar** (*comments*) iki biçimde yazılır: `//` satırın sonuna kadar, `/* … */` ise birden çok satır boyunca sürer. Derleyici açıklamaları yok sayar.

Aşağıdaki program bu kuralların hepsini bir arada gösterir.

```
/* Çok satırlı açıklama:
   derleyici bunu yok sayar. */
public class Kurallar {
    // Tek satırlık açıklama
    public static void main(String[] args) {
        int sayi = 5;  // ; ile biter
        int Sayi = 7;  // sayi ≠ Sayi
        System.out.println(sayi
                + Sayi);   // satır bölünebilir
    }
}
```

Program şu çıktıyı verir:

```
12
```

İlk üç satır çok satırlı bir açıklamadır ve derlenen koda hiçbir şey katmaz. `sayi` ile `Sayi` büyük-küçük harf farkı yüzünden iki ayrı değişkendir: biri 5, öteki 7 değerini tutar. `println` çağrısı iki satıra bölünmüştür. Derleyici için bu bölünme önemsizdir, çünkü deyim ancak `;` görülünce biter. Çıktıdaki 12, iki değişkenin toplamıdır. İki ayrı değişkenin yan yana bulunabilmesi, adlardaki tek bir harf farkının ne kadar önemli olduğunu gösterir. Gerçek programlarda böyle benzer adlar karışıklığa yol açar ve kaçınılması gerekir; buradaki amaç yalnız kuralı göstermektir.

Sık yapılan hata

Büyük-küçük harfi karıştırmak. Aşağıdaki üç satır, Türkçe yazarken alışılan küçük harf kullanımından kaynaklanan tipik hatalardır. Derleyici her biri için ayrı bir ileti verir:

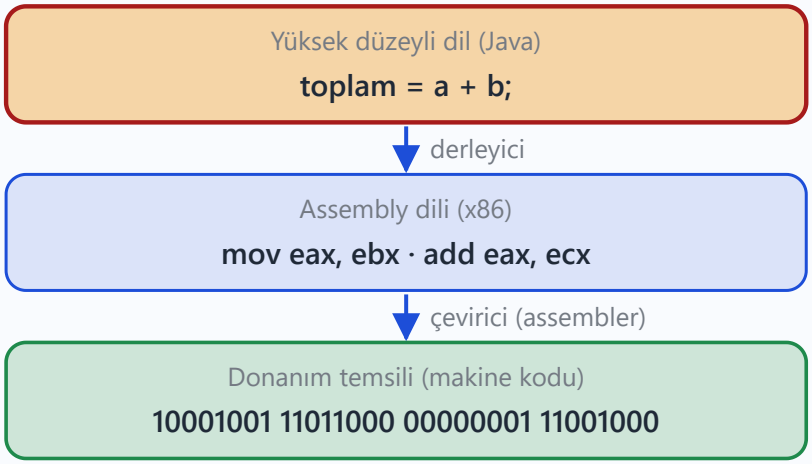
```
Harf.java:3: error: package system does not exist
        system.out.println("a");
              ^
Harf.java:4: error: cannot find symbol
        string s = "b";
        ^
  symbol:   class string
  location: class Harf
Harf.java:5: error: cannot find symbol
        Int n = 3;
        ^
  symbol:   class Int
  location: class Harf
3 errors
```

`system.out` yazılınca derleyici `system`'i bir paket adı sanar ve bu paketi bulamaz. `string` ve `Int` ise tanımlı olmayan sınıf adları gibi görülür. Doğru yazımlar `System`, `String` ve `int`'tir. `String` bir sınıf olduğu için büyük harfle, `int` bir anahtar sözcük olduğu için küçük harfle başlar.

2.2.1.5 Soyutlama düzeyleri

Bir işlemci yalnız ikili sayılarla kodlanmış komutları anlar. İnsan ise "a ile b'yi topla, sonucu toplama yaz" gibi düşünür. İnsandan işlemciye uzanan bu yolda üç düzey vardır ve her düzey bir altındakinin ayrıntılarını gizler.

- **Yüksek düzeyli dil** (*high-level language*) problem alanına yakındır. Java, C ve Python bu düzeydedir. Kod kısa, okunur ve anlaşılırdır; programcı daha üretken çalışır.
- **Assembly dili** (*assembly language*) işlemci komutlarını `MOV` ve `ADD` gibi kısa adlarla yazar. Her satır çoğu zaman tek bir makine komutuna karşılık gelir. Daha düşük düzeyli bir soyutlamadır.
- **Donanım temsili** (*hardware representation*) yalnız bitlerden oluşur. Komutlar ve veriler bu biçimde doğrudan donanıma gider.



Şekil 2.2.1: Aynı toplama işleminin üç düzeyi. Java'daki tek satır, assembly dilinde iki komuta, donanımda iki baytlık iki makine koduna karşılık gelir.

Şekil 2.2.1 `toplam = a + b` satırının üç düzeydeki karşılığını gösterir. Örnekte `a`, `b` ve `toplam` değişkenlerinin sırasıyla `ebx`, `ecx` ve `eax` yazmaçlarında durduğu varsayılmıştır. Assembly dilinde önce `a`'nın değeri `eax`'e kopyalanır (`mov eax, ebx`), sonra `b` bunun üstüne eklenir (`add eax, ecx`). Makine kodu x86 işlemcilerinin gerçek kodlamasıdır: birinci komut onaltılık `89 D8`, ikinci komut `01 C8` baytlarıdır. Soyutlama arttıkça kod kısalır ve okunur hâle gelir. Alt düzeye çevirme işini bir derleyici ya da bir assembler yapar.

Java bu tabloya bir ara katman ekler. Java derleyicisi kaynak kodu doğrudan x86 makine koduna çevirmez. Önce bytecode denilen ve hiçbir gerçek işlemciye ait olmayan bir koda çevirir. Bu ara kod, aynı derlenmiş programın farklı işlemcilerde çalışabilmesini sağlar.

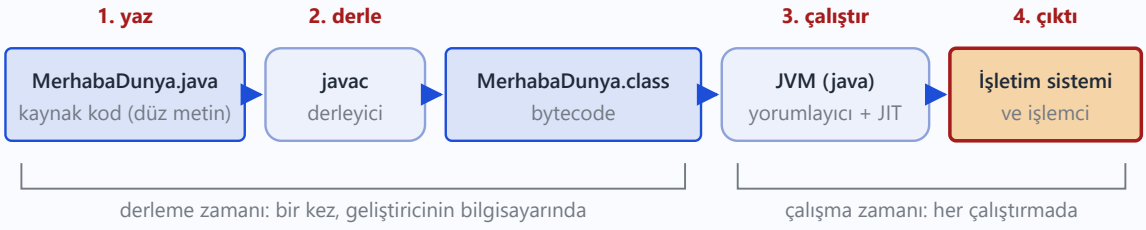
2.2.1.6 Derleme

Tanım 2.2.3

Derleme (*compilation*), kaynak kodun bütünüyle başka bir dile çevrilmesidir. Bu işi yapan programa **derleyici** (*compiler*) denir. Java derleyicisi `javac`, kaynak kodu işlemcinin makine diline değil, Java Sanal Makinesi'nin anladığı **bytecode**'a çevirir ve sonucu `.class` uzantılı bir dosyaya yazar.

Bilgisayar `.java` metnini doğrudan çalıştıramaz. Metnin önce derlenmesi gerekir. Kaynak koddan ekrandaki çıktıya giden yol dört adımda tamamlanır:

1. Programcı kaynak kodu düz metin olarak yazar ve `MerhabaDunya.java` adıyla kaydeder.
2. `javac MerhabaDunya.java` komutu derleyiciyi çalıştırır. Derleyici söz dizimini ve tipleri denetler. Hata yoksa aynı klasörde `MerhabaDunya.class` dosyası oluşur.
3. `java MerhabaDunya` komutu **başlatıcıyı** (*launcher*) çalıştırır. Başlatıcı JVM'i açar ve sınıfı yükler. JVM bytecode'u yorumlar, sık çalışan kısımları JIT derleyicisiyle makine koduna çevirir.
4. İşletim sistemi ve işlemci asıl komutları yürütür; ekranda `Merhaba, Dünya!` görünür.



Şekil 2.2.2: Derleme ve çalıştırma hattı. `.java` dosyası `javac` ile `.class` dosyasına, `.class` dosyası da JVM ile çalışan programa dönüşür.

Şekil 2.2.2'de iki ayrı komut ve iki ayrı zaman dilimi vardır. **Derleme zamanı**, `javac`'ın çalıştığı süredir; bu sürede program henüz çalışmaz, yalnız denetlenir ve çevrilir. **Çalışma zamanı** ise `java` komutuyla başlar. Bir hatanın hangi zamanda ortaya çıktığı önemlidir. Derleme hatası programın hiç çalışmamasına yol açar. Çalışma zamanı hatası ise program çalışırken belirir; sıfıra bölme bunun örneğidir.

Derleme aşamasının en büyük yararı, hataların program çalışmadan önce yakalanmasıdır. Derleyici eksik noktalı virgülü, kapanmamış parantezi, yanlış yazılmış bir adı ve uyumsuz tipleri bulur. Hata varsa `.class` dosyası oluşmaz. İletideki `dosya adı:satır` bilgisi hatanın yerini gösterir. `MerhabaDunya` programının 3. satırındaki noktalı virgül unutulursa satır şöyle olur:

```
System.out.println("Merhaba, Dünya!")
```

Derleyici bu satır için şu iletiyi verir:

```
MerhabaDunya.java:3: error: ';' expected
        System.out.println("Merhaba, Dünya!")
                                              ^
1 error
```

İletinin ilk satırı dosyayı (`MerhabaDunya.java`), satır numarasını (3) ve hatanın türünü (`';' expected`, yani noktalı virgül bekleniyordu) verir. İkinci satır hatalı kodu yineler. Üçüncü satırdaki `^` işareti, derleyicinin noktalı virgülü beklediği yeri gösterir: kapanış parantezinin hemen sağı. Son satır toplam hata sayısını bildirir. Derleme aşaması programın hızına da katkı sağlar. Derleyici tip denetimi gibi işleri bir kez yapar; çalışma sırasında bu işlerin tekrarlanmasına gerek kalmaz. JVM'in çalışma sırasındaki iyileştirmeleri de buna eklenir.

Sık yapılan hata

Hatayı iletinin gösterdiği satırdan önce aramamak. Noktalı virgül eksik olduğunda derleyici bunu çoğu zaman bir sonraki simgeyi okuyunca fark eder. Bu yüzden ileti bazen bir alt satırı gösterir. Bir satırda anlamsız görünen bir `';' expected` iletisi çıkarsa önce bir önceki satırın sonuna bakılır.

2.2.1.7 Komut satırında derleme ve çalıştırma

Java ile temel işlem adımları üç tanedir. **Derleme** adımında `.java` dosyası `.class` dosyasına dönüştürülür: `javac MerhabaDunya.java`. **Çalıştırma** adımında `java` komutuna uzantısız sınıf adı verilir: `java MerhabaDunya`. Üçüncü adım **çıktıdır**: program ekrana `Merhaba, Dünya!` yazar. Bir komut satırı oturumu şöyle görünür:

```
> javac MerhabaDunya.java
> dir /b
MerhabaDunya.class
MerhabaDunya.java
> java MerhabaDunya
Merhaba, Dünya!
```

`javac` hiçbir şey yazmadıysa derleme başarılıdır. Ardından gelen `dir /b` komutu (Linux ve macOS'ta `ls`) klasördeki dosyaları listeler. Listede kaynak dosyanın yanında yeni oluşan `MerhabaDunya.class` görülür. Son komut programı çalıştırır.



Şekil 2.2.3: İki komut, iki dosya: `javac` kaynak dosyayı `.class` dosyasına çevirir; `java` JVM'i başlatır ve program çıktı verir.

Sık yapılan hata

Çalıştırırken uzantı yazmak: `java MerhabaDunya.class`. JVM bir dosya adı değil, bir **sınıf adı** bekler. Bu komut `MerhabaDunya.class` adlı bir sınıf arar ve "Could not find or load main class" iletisiyle durur. Doğrusu uzantısız yazmaktır: `java MerhabaDunya`.

Java 11 ile tek dosyalık programlar için bir kısayol geldi. `java MerhabaDunya.java` komutu dosyayı bellekte derler ve hemen çalıştırır. Bu yolda diske `.class` dosyası yazılmaz. Kısayol küçük denemeler için pratiktir. Ancak birden çok dosyadan oluşan programlarda iki aşamalı yol kullanılır.

2.2.1.8 Bytecode

Tanım 2.2.4

Bytecode (ara kod), `javac`'ın ürettiği ve Java Sanal Makinesi'nin çalıştırdığı, işlemciden ve işletim sisteminden bağımsız komutlardır. `.class` dosyaları işlemcinin doğrudan anlayabileceği makine kodu içermez; içlerinde bytecode bulunur. Bytecode, JVM'in "makine dili"dir.

Adı komutların yapısından gelir: her komutun **işlem kodu** (*opcode*) tek bir bayttır. Komutun gerektirdiği ek bilgi, örneğin bir sabit ya da bir tablo kaydının numarası, işlem kodundan sonraki baytlarda durur. Tek baytlık işlem kodu en çok 256 farklı komut demektir ve bu sayı JVM için yeterlidir.

Bytecode bir işlemciye ya da işletim sistemine bağlı değildir. Aynı `.class` dosyası Windows'ta, Linux'ta ve macOS'ta çalışır. Bir benzetme bu ilişkiyi iyi anlatır: bytecode her ülkede aynı okunan bir nota gibidir. Her bilgisayardaki JVM ise o notayı kendi çalgısında çalan müzisyendir. Nota değişmez; çalgı ve müzisyen değişir.

Bir `.class` dosyası ikili bir dosyadır ve metin düzenleyicide anlamsız karakterler olarak görünür. JDK'daki `javap -c Sinif` komutu bu dosyadaki bytecode komutlarını okunur biçimde listeler. İki satırlık şu kodu ele alalım: `int x = 5; System.out.println(x);`. Kod bir `main` metodunun içine yazı-

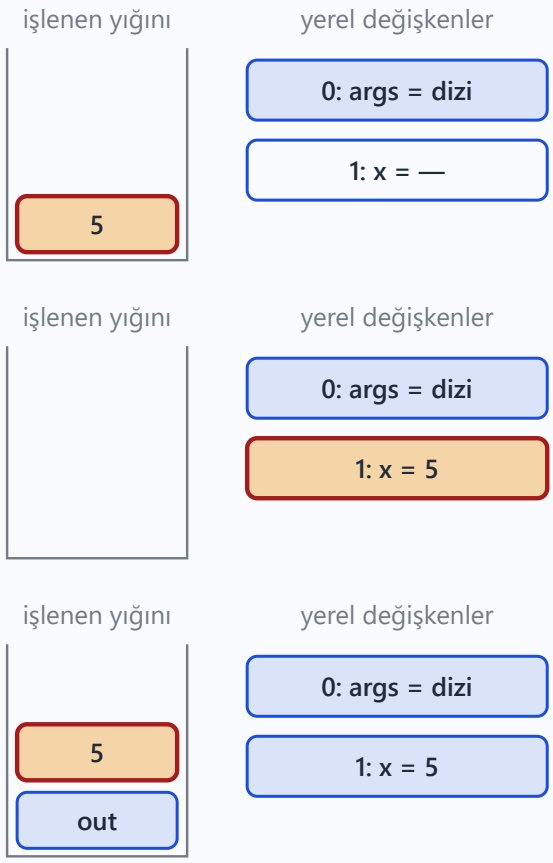
lıp JDK 21 ile derlendiğinde `javap -c` şu listeyi verir:

```
0: iconst_5
1: istore_1
2: getstatic     #7    // System.out
5: iload_1
6: invokevirtual #13   // println(int)
9: return
```

Her satırın başındaki sayı, komutun metot içindeki bayt konumudur. Konumlar 0, 1, 2, 5, 6, 9 diye ilerler. `iconst_5` ve `istore_1` tek baytlık komutlardır. `getstatic` ve `invokevirtual` ise işlem kodundan sonra iki baytlık bir tablo numarası taşır; bu yüzden üç bayt yer kaplar.

JVM bir **yığın makinesidir**. Gerçek işlemciler değerleri yazmaçlarda tutar. JVM'in komutları ise değerleri bir **işlenen yığınına** (*operand stack*) iter ve oradan alır. Yığın, üst üste konmuş tabaklar gibidir: en son konan en önce alınır. Komutlar sırayla şunu yapar:

1. `iconst_5` : sabit 5 işlenen yığınına itilir. Kaynaktaki karşılığı `int x = 5;` satırının sağ tarafıdır.
2. `istore_1` : yığının üstündeki değer alınır ve 1 numaralı yerel değişkene, yani `x` 'e yazılır. Yığın yeniden boşalır.
3. `getstatic #7` : `System.out` nesnesine bir başvuru yığına itilir. `#7` , sınıf dosyasındaki sabitler tablosunun 7 numaralı kaydıdır.
4. `iload_1` : `x` 'in değeri (5) yığına kopyalanır. Kaynaktaki karşılığı `System.out.println(x);` satırıdır.
5. `invokevirtual #13` : `println(int)` metodu çağrılır. Metot iki değeri yığından alır: hangi nesne üzerinde çalışacağını (`System.out`) ve yazacağı sayıyı (5). Ekrana 5 yazılır.
6. `return` : `main` biter, program sonlanır.



Şekil 2.2.4: Bytecode yürürken işlenen yığını ve yerel değişkenler. (a) `iconst_5` sonrası yığında 5 vardır, `x` henüz boştur. (b) `istore_1` sonrası yığın boşalmış, `x` = 5 olmuştur. (c) `iload_1` sonrası yığında `System.out` başvurusu ve 5 durur; `println` bu ikisini alacaktır.

Şekil 2.2.4 üç anı gösterir. (b) ile (c) arasındaki fark önemlidir: `iload_1` değeri yerel değişkenden yığına **kopyalar**. `x` yerinde kalır ve değeri değişmez. Bir değişkeni okumak onu tüketmez; bu durum bytecode düzeyinde de geçerlidir.

2.2.1.9 JVM yığınında bir toplama

Yığın makinesi fikri bir toplama işleminde daha açık görülür. Şu üç satırı ele alalım:

```
int a = 5;
int b = 7;
int toplam = a + b;
System.out.println(toplam);
```

Bu satırların derlenmiş hâli on iki komuttur. Açıklamalar, her komutun kaynak koddaki karşılığını verir:

```
iconst_5        // 5'i yığına koy
istore_1        // a = 5
bipush 7        // 7'yi yığına koy
istore_2        // b = 7
iload_1         // a'yı yükle
iload_2         // b'yi yükle
iadd            // topla
istore_3        // toplam = 12
getstatic       // System.out
iload_3         // toplam'ı yükle
invokevirtual   // println(int)
return
```

JVM toplama yapmak için yazmaç kullanmaz. İki sayıyı yığına iter, `iadd` komutu üstteki iki değeri çıkarıp toplar ve sonucu yeniden yığına koyar. Tablo 2.2.2 her komuttan sonra işlenen yığınını ve yerel değişken tablosunu verir. Yerel değişken tablosunun 0 numaralı yuvasında `main`'in parametresi `args` durur; tabloda "…" ile gösterilmiştir. `a`, `b` ve `toplam` sırasıyla 1, 2 ve 3 numaralı yuvalara yerleşir.

Komut	İşlenen yığını	1: `a`	2: `b`	3: `toplam`	Açıklama
`iconst_5`	5				sabit 5 itilir
`istore_1`	boş	5			`a = 5`
`bipush 7`	7	5			bir bayta sığan sabit 7 itilir
`istore_2`	boş	5	7		`b = 7`
`iload_1`	5	5	7		`a` kopyalanır
`iload_2`	5, 7	5	7		`b` kopyalanır
`iadd`	**12**	5	7		iki değer çıkar, toplam girer
`istore_3`	boş	5	7	**12**	`toplam = 12`
`getstatic`	out	5	7	12	`System.out` başvurusu
`iload_3`	out, 12	5	7	12	`println`'ın argümanı
`invokevirtual`	boş	5	7	12	ekrana 12 yazılır
`return`	boş	5	7	12	`main` biter

Tablo 2.2.2: `int toplam = a + b;` örneğinde her komuttan sonra JVM'in durumu. Yığının üstü sağdadır.



Şekil 2.2.5: Toplamanın üç anı. (a) `iload_2` sonrası yığında 5 ve 7 vardır. (b) `iadd` ikisini çıkarır ve 12'yi iter. (c) `istore_3` 12'yi `toplam`'ın yuvasına yazar, yığın boşalır.

Tablodan üç gözlem çıkar. Birincisi, 5 için `iconst_5`, 7 için ise `bipush 7` kullanılır. JVM'de −1 ile 5 arasındaki küçük sabitlerin her birine ayrı, tek baytlık bir komut ayrılmıştır. 7 bu aralıkta olmadığı için "bir bayt it" anlamına gelen `bipush` ve ardından 7 değerini taşıyan bir bayt yazılır. İkincisi, yığın her deyimin sonunda boşalır. Yığın yalnız bir ifadenin hesaplandığı kısa süre boyunca dolu kalır. Üçüncüsü, `iload` komutları değişkenleri tüketmez; `a` ve `b` toplamadan sonra da 5 ve 7 olarak durur.

Komut adlarındaki `i` harfi `int` tipini belirtir. `long` için `ladd`, `double` için `dadd` gibi ayrı komutlar vardır. JVM her işlemde hangi tiple çalıştığını bilir. Tiplerin derleme sırasında denetlenmesinin bir nedeni de budur: derleyici hangi komutu üreteceğine tipe bakarak karar verir. Aynı bytecode her işletim sistemindeki JVM'de bu sırayla çalışır.

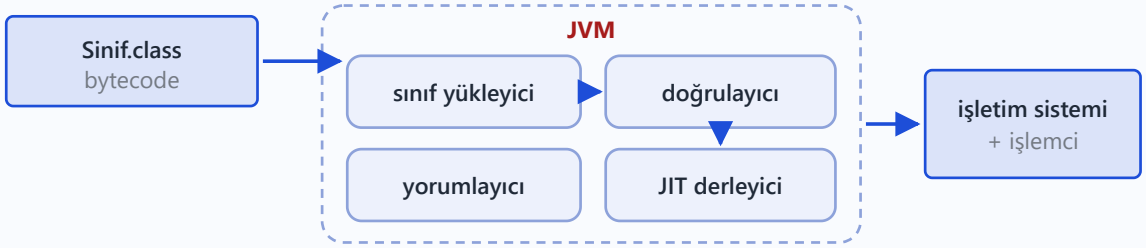
2.2.1.10 Java Sanal Makinesi ve JIT

Tanım 2.2.5

Java Sanal Makinesi (*Java Virtual Machine*, JVM), Java uygulamalarını çalıştıran yazılımdır. Bytecode'u yorumlar ve bulunduğu işlemciye uygun biçimde yürütür. Sık çalışan bytecode parçalarını **JIT** (*just-in-time*, tam zamanında) derleyicisiyle çalışma sırasında makine koduna çevirir.

"Sanal makine" adı, JVM'in kendi komut kümesi olan bir bilgisayar gibi davranmasından gelir. Bu bilgisayar donanım olarak yoktur; bir program olarak gerçek bilgisayarın üzerinde çalışır. JVM'in iç yapısı dört ana parçadan oluşur:

- **Sınıf yükleyici**, gerekli `.class` dosyalarını bulur ve belleğe alır.
- **Doğrulayıcı**, yüklenen bytecode'un kurallara uyduğunu denetler. Örneğin bir komutun yığında bulunmayan bir değeri almaya çalışmadığından emin olur. Böylece bozuk ya da kötü niyetli bir dosya JVM'i çökertemez.
- **Yorumlayıcı**, bytecode komutlarını tek tek okur ve her birinin karşılığı olan işi yapar.
- **JIT derleyicisi**, çok kez çalışan kod parçalarını fark eder ve onları doğrudan makine koduna çevirir. Sonraki çalışmalarda yorumlama yerine bu makine kodu kullanılır.



Şekil 2.2.6: JVM'in iç yapısı: sınıf yükleyici, doğrulayıcı, yorumlayıcı ve JIT derleyici. Üretilen işler işletim sistemine gider.

Yorumlama esnektir ama yavaştır: her komut her çalıştırılışında yeniden çözümlenir. Makine kodu ise hızlıdır ama üretilmesi zaman alır. JIT bu ikisi arasında bir denge kurar. Bir kez çalışan kod yorumlanır, binlerce kez çalışan bir döngü gövdesi derlenir. Bu yüzden bir Java programı zamanla hızlanır. Uzun süre çalışan sunucu programlarında bu etki belirgindir.

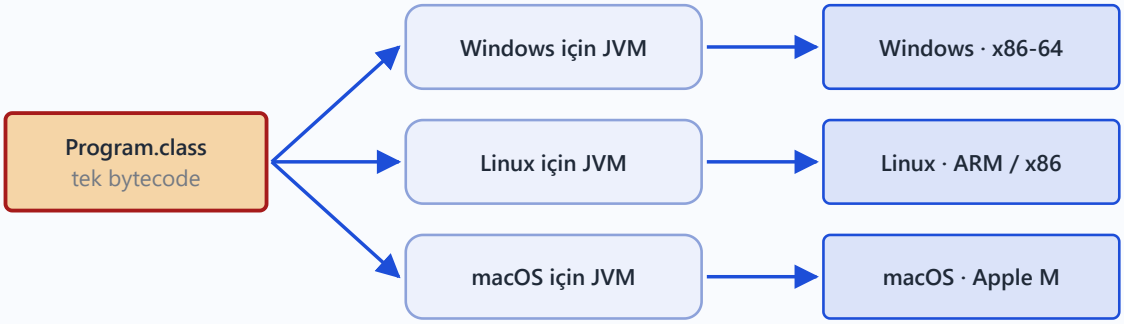
Programı başlatan `java` komutu, JDK belgelerinde **Java başlatıcı aracı** (*launcher tool*) diye geçer. Başlatıcı JVM'i başlatır, adı verilen sınıfı yükler ve o sınıfın `main` metodunu çağırır. `main` bitince JVM de kapanır.

2.2.1.11 Platform bağımsızlığı ve Java platformu

Tanım 2.2.6

Platform, bir programın çalıştığı donanım ya da yazılım ortamıdır: işlemci ile işletim sisteminin birleşimi. **Platform bağımsızlığı** (*platform independence*), aynı derlenmiş programın JVM bulunan her platformda yeniden derlenmeden çalışmasıdır.

Platform bağımsızlığı Java'nın temel özelliklerinden biridir ve "Bir kez yaz, her yerde çalıştır" (*Write Once, Run Anywhere*) sloganıyla anılır. Program bir kez derlenir; aynı bytecode farklı işletim sistemlerinde çalıştırılır. Bunu mümkün kılan, JVM'in birçok işletim sistemi ve işlemci için ayrı ayrı yazılmış olmasıdır. Windows için yazılmış JVM bytecode'u x86 makine koduna çevirir. ARM işlemcili bir Mac'teki JVM aynı bytecode'u ARM makine koduna çevirir. Farklılık bytecode'da değil, JVM'dedir.



Şekil 2.2.7: Platform bağımsızlığı. Aynı `.class` dosyası Windows, Linux ve macOS için yazılmış JVM'lerde çalışır.

C gibi dillerde durum farklıdır. C derleyicisi doğrudan makine kodu üretir ve bu kod derlendiği platforma bağlıdır. Windows'ta derlenen bir C programı Linux'ta çalışmaz; her platform için ayrı derleme gerekir. Java'da bu yük JVM'i yazanlara geçer. Programcı tek bir `.class` dosyası dağıtır, kullanıcının bilgisayarında JVM bulunması yeterlidir.

Java platformunun iki ana bileşeni vardır:

- **JVM** uygulamaları çalıştırır ve farklı donanımlara taşınabilirlik sağlar.
- **API** (*Application Programming Interface*, uygulama programlama arayüzü) hazır sınıf kütüphanelerinden oluşur. Grafik arayüz, veri tabanı, ağ iletişimi ve dosya işlemleri gibi birçok işlevi hazır sunar ve geliştirmeyi hızlandırır. `System`, `String`, `Math` ve `Scanner` bu kütüphanelerdeki sınıflardır.



Şekil 2.2.8: Java platformunun katmanları. JDK çalışma ortamını, çalışma ortamı da API kütüphanelerini ve JVM'i içerir.

Şekil 2.2.8 katmanların iç içe olduğunu gösterir. Çalışma ortamı (JRE, *Java Runtime Environment*) bir Java programını çalıştırmak için gerekenleri, yani JVM'i ve API kütüphanelerini içerir. JDK bunların üstüne derleyiciyi ve öteki geliştirme araçlarını ekler. Program geliştirmek için JDK kurulur. Oracle, Java 11'den beri JRE'yi ayrı bir paket olarak dağıtmaz; JDK hepsini içerir.

2.2.2 Değişkenler ve veri tipleri

Bir program veri alır, veriyi işler ve sonuç üretir. Bu sürecin her anında verinin bir yerde durması gerekir. Değişkenler bu işi görür. Java her değişkenin hangi türden veri tuttuğunu baştan bilmek ister; bu bilgiye tip denir. Bu kesim değişkenleri, Java'nın sekiz ilkel tipini ve her tipin sınırlarını ele alır.

2.2.2.1 Temel programlama unsurları

Her program, hangi dille yazılmış olursa olsun, birkaç temel unsurdan kurulur. **Değişkenler** verileri saklar. Her değişkenin bir **tipi** vardır. Değerler **ifadelerle** (*expressions*) hesaplanır: `sayi > 0` ve `toplam + 1` birer ifadedir. **Kontrol akışı** (*control flow*) ise programın hangi sırayla ve nasıl çalışacağını belirler. Kontrol akışının iki temel biçimi vardır:

- **Dallanma** (*branching*), bir koşula göre programın farklı kısımlarını çalıştırır. `if-else` buna örnektir.
- **Döngüler** (*loops*), bir işlemi tekrarlar. `for` ve `while` buna örnektir.

Aşağıdaki kod parçası bir değişkeni, bir koşulu ve çıktıyı bir araya getirir:

```
int sayi = 5;
if (sayi > 0) {
    System.out.println("Sayı pozitif.");
} else {
    System.out.println("Sayı negatif veya sıfır.");
}
```

```
Sayı pozitif.
```

`sayi` 5 olduğu için `sayi > 0` ifadesi doğrudur ve ilk blok çalışır. `else` bloğu atlanır. `sayi` 0 ya da negatif olsaydı ilk blok atlanır, ikinci blok çalışırdı. Burada `sayi > 0` ifadesinin değeri bir sayı değil, doğru ya da yanlış bilgisidir. Java bu bilgi için ayrı bir tip kullanır: `boolean`.

2.2.2.2 Değişken ve bildirim

Tanım 2.2.7

Değişken (*variable*), bellekte bir değeri saklayan, adı ve tipi olan bir alandır. Değişkenin kullanılmadan önce tanıtılmasına **bildirim** (*declaration*) denir. Bildirimde önce tip, sonra ad yazılır ve bildirim `;` ile biter.

Değişken, üzerinde ad yazan bir kutu gibi düşünülebilir. Kutunun etiketinde iki bilgi vardır: adı ve içine ne tür bir şey konabileceği. `int` etiketli bir kutuya yalnız tam sayı, `double` etiketli bir kutuya yalnız ondalıklı sayı konur. Kutunun içindeki değer program çalışırken değişebilir; adı ve tipi ise değişmez.



Şekil 2.2.9: Değişkenler etiketli kutular gibidir: `numara` (`int`) 1042, `faizOrani` (`double`) 0.45, `cevap` (`char`) `'E'`.

Java'da her değişken kullanılmadan önce bildirilmelidir. Aynı tipten birden çok değişken virgülle ayrılarak tek satırda bildirilebilir:

```
int numara, cekSayisi, mevcutSayisi;
double miktar, faizOrani;
char cevap;
```

Birinci satır üç `int` değişken, ikinci satır iki `double` değişken, üçüncü satır bir `char` değişken bildirir. Bildirimin üç biçimi vardır: `tip ad;`, `tip ad1, ad2, ad3;` ve `tip ad = ilkDeğer;`. Sonuncusu bildirimle birlikte değişkene ilk değerini de verir.

Adlar açıklayıcı ve anlamlı seçilmelidir. `x` adı değişkenin ne tuttuğu hakkında hiçbir şey söylemez. `cekSayisi` ise bir hesaptaki çek sayısını tuttuğunu hemen belli eder. Birkaç ay sonra kodu yeniden okuyan kişi, çoğu zaman kodu yazan kişinin kendisidir. Anlamlı adlar bu kişinin işini kolaylaştırır.

2.2.2.3 Bildirim, atama, güncelleme

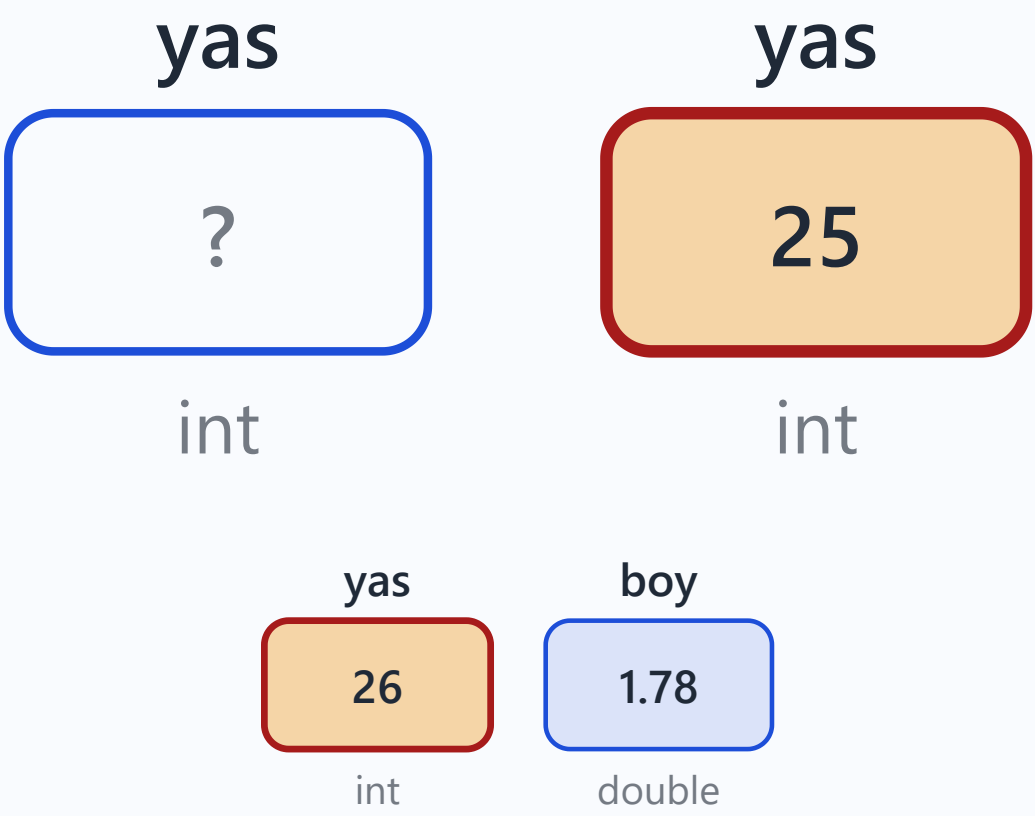
Bir değişkenin yaşamı üç olayla ilerler: bildirilir, bir değer alır ve bu değer güncellenir. Aşağıdaki kod üç olayı da içerir:

```
int yas;
yas = 25;
double boy = 1.78;
yas = yas + 1;
char harf = 'K';
System.out.println(yas + " " + boy + " " + harf);
```

Program `26 1.78 K` yazar. Tablo 2.2.3 her satırdan sonra bellekteki kutuları gösterir.

Satır	Kod	`yas`	`boy`	`harf`	Açıklama
1	`int yas;`	?			kutu ayrılır, içi boştur
2	`yas = 25;`	**25**			atama: kutuya 25 konur
3	`double boy = 1.78;`	25	**1.78**		bildirim ve ilk değer tek satırda
4	`yas = yas + 1;`	**26**	1.78		önce 25 + 1 = 26 hesaplanır
5	`char harf = 'K';`	26	1.78	**'K'**	karakter tek tırnakla yazılır
6	`println(…)`	26	1.78	'K'	çıktı: `26 1.78 K`

Tablo 2.2.3: Bildirim, atama ve güncelleme izlemesi. “?” değer atanmamış kutuyu, boş hücre henüz bildirilmemiş değişkeni gösterir.



Şekil 2.2.10: Değişkenin yaşamından üç an. (a) `int yas;` kutuyu ayırır, içi boştur. (b) `yas = 25;` kutuya 25'i koyar. (c) `yas = yas + 1;` sonrası kutuda 26 vardır; eski değer kaybolmuştur.

1. satırdan sonra `yas` adlı kutu vardır ama içinde değer yoktur. Bu durumda `yas` okunamaz; okumaya çalışan bir kod derlenmez. 4. satır bir güncellemedir. Önce sağ taraf hesaplanır: kutudaki 25 okunur ve 1 eklenir. Sonuç olan 26 aynı kutuya yazılır ve eski değer kaybolur. 6. satırdaki `yas + " " + boy + " " + harf` ifadesinde `+` işareti sayıları toplamaz. İşlenenlerden biri metin olduğu için değerleri metne ekler. Bu işleme **dizgi birleştirme** denir.

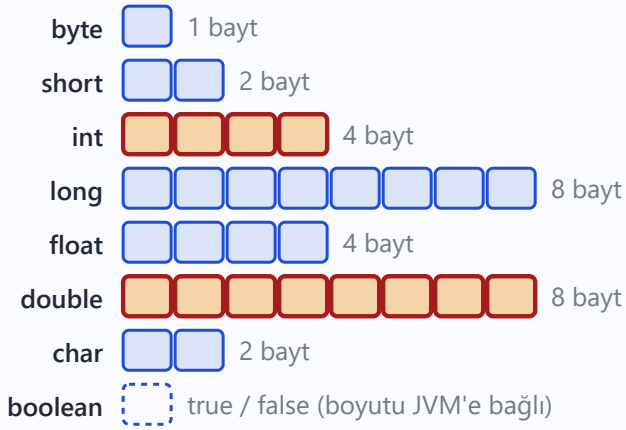
2.2.2.4 Sekiz ilkel tip

Tanım 2.2.8

Veri tipi (*data type*), bir değişkenin hangi değerleri alabileceğini, bellekte ne kadar yer kapladığını ve üzerinde hangi işlemlerin yapılabileceğini belirler. Java'nın dile gömülü sekiz temel tipine **ilkel tip** (*primitive type*) denir: `byte`, `short`, `int`, `long`, `float`, `double`, `char` ve `boolean`.

Sekiz tip dört gruba ayrılır:

- **Tam sayı:** `byte`, `short`, `int`, `long`.
- **Kesirli (ondalıklı) sayı:** `float`, `double`.
- **Karakter:** `char`.
- **Mantıksal:** `boolean`.



Şekil 2.2.11: İlkel tiplerin bellekte kapladığı yer: `byte` 1, `short` 2, `int` 4, `long` 8, `float` 4, `double` 8, `char` 2 bayt. `boolean`'ın boyutunu JVM belirler.

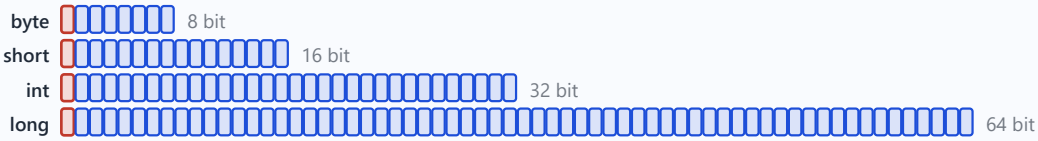
Her tipin bellekte kendine özgü bir boyutu ve buna bağlı bir değer aralığı vardır. Bit sayısı arttıkça tip daha büyük sayıları tutar, ama bellekte daha çok yer kaplar. C dilinde `int`'in boyutu derleyiciye ve işlemciye göre değişebilir. Java'da ise boyutlar dilin tanımında sabittir: `int` her platformda 32 bittir. Bu sabitlik de platform bağımsızlığının bir parçasıdır.

Günlük kullanımda tam sayılar için `int`, ondalıklı sayılar için `double` varsayılan seçimdir. `byte` ve `short` çoğunlukla büyük dizilerde bellek kazanmak ya da dosyadan bayt okumak gibi özel durumlarda kullanılır. `float` ise grafik ve bazı gömülü sistemler dışında pek tercih edilmez.

2.2.2.5 Tam sayı tipleri

Tip	Bellek	En küçük değer	En büyük değer	Örnek
`byte`	1 bayt (8 bit)	−128	127	`byte b = 100;`
`short`	2 bayt (16 bit)	−32.768	32.767	`short s = 32000;`
`int`	4 bayt (32 bit)	−2.147.483.648	2.147.483.647	`int n = 1_000_000;`
`long`	8 bayt (64 bit)	−9.223.372.036.854.775.808	9.223.372.036.854.775.807	`long l = 3000000000L;`

Tablo 2.2.4: Tam sayı tipleri, bellek boyutları ve değer aralıkları.



Şekil 2.2.12: Tam sayı tiplerinin bit genişlikleri: `byte` 8, `short` 16, `int` 32, `long` 64 bit.

Dört tipin hepsi **işaretlidir** (*signed*): hem pozitif hem negatif değer alır. n bitlik bir tipin aralığı -2^{n-1} ile $2^{n-1} - 1$ arasıdır. `byte` için $n = 8$'dir ve aralık $-2^7 = -128$ ile $2^7 - 1 = 127$ arasında kalır. Pozitif tarafın bir eksik olmasının nedeni, sıfırın da pozitif yarıda yer kaplamasıdır: 8 bit 256 farklı değer verir; bunların 128'i negatif, biri sıfır, 127'si pozitiftir.

Sınırları ezberlemek gerekmez. Java her tip için sabitler sunar: `Byte.MIN_VALUE`, `Byte.MAX_VALUE`, `Integer.MAX_VALUE`, `Long.MIN_VALUE` gibi. Yine de `int`'in üst sınırının yaklaşık 2,1 milyar olduğunu bilmek yararlıdır. Dünya nüfusu, milisaniye cinsinden bir ay ya da bir ülkenin bütçesi bu sınırı aşar.

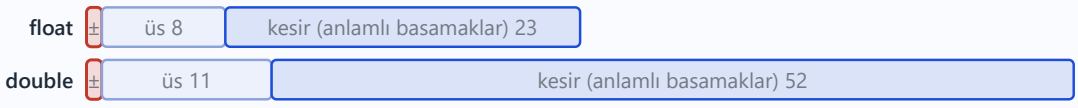
Kodda doğrudan yazılan bir tam sayı, örneğin `42`, `int` kabul edilir. `int` aralığını aşan bir sayı `long` olarak yazılacaksa sonuna `L` eklenir: `3000000000L`. Küçük harf `l` de geçerlidir ama 1 rakamına çok benzediği için büyük harf tercih edilir. `L` unutulursa derleyici `integer number too large` iletisi verir: sayı `long` değişkene atanmadan önce `int` olarak okunmaya çalışılır ve sığmaz.

2.2.2.6 Ondalıklı sayı tipleri

Ondalıklı sayılar bilgisayarda **kayan noktalı** (*floating-point*) biçimde saklanır. Bu biçim, bilimsel gösterimin ikili tabandaki karşılığıdır. Sayı bir işaret, bir üs ve bir kesirden oluşur: $6{,}02 \times 10^{23}$ sayısında 6,02 kesir, 23 üstür. Kesir anlamlı basamakları, üs ise virgülün yerini belirler. Virgül sayının büyüklüğüne göre “kayar”; ad buradan gelir.

Tip	Bellek	En küçük pozitif	En büyük (±)	Anlamlı basamak
`float`	4 bayt (32 bit)	$\approx 1{,}4 \times 10^{-45}$	$\approx 3{,}4028235 \times 10^{38}$	≈ 7
`double`	8 bayt (64 bit)	$\approx 4{,}9 \times 10^{-324}$	$\approx 1{,}7976931348623157 \times 10^{308}$	≈ 15–16

Tablo 2.2.5: Ondalıklı sayı tipleri. “En küçük pozitif”, sıfırdan farklı en küçük pozitif değerdir.



Şekil 2.2.13: Kayan noktalı sayının bitleri. `float`: 1 işaret, 8 üs, 23 kesir biti. `double`: 1 işaret, 11 üs, 52 kesir biti.

Şekil 2.2.13 iki tipin farkını gösterir. `double`'ın kesir kısmı 52 bittir; bu, yaklaşık 15–16 onluk basamağa karşılık gelir. `float`'ın 23 bitlik kesri ise yaklaşık 7 basamak tutar. `double` **çift duyarlıklıdır** (*double precision*): `float`'tan hem daha geniş bir aralığı hem daha fazla basamağı vardır. Tablodaki

negatif sınırlar, en büyük değerlerin eksi işaretlisidir.

Kodda yazılan ondalıklı bir sayı, örneğin `3.14`, `double` kabul edilir. `float` için sonuna `f` eklenir: `3.14f`. Aradaki basamak farkı küçük bir hesapta bile görülür:

```
System.out.println(1.1f * 3);   // 3.3000002
System.out.println(1.1 * 3);    // 3.3000000000000003
```

İki sonuç da tam 3,3 değildir. `float`'taki hata yedinci basamakta, `double`'daki hata on altıncı basamakta görünür. Hatanın kaynağı sayının kendisidir: 1,1 ikili tabanda sonlu bir kesirle yazılamaz ve bellekte en yakın değere yuvarlanmış olarak durur.

2.2.2.7 char ve boolean

`char` tek bir karakter tutar. Bellekte 2 bayt (16 bit) yer kaplar ve **işaretsizdir**: değer aralığı 0 ile 65.535 arasındadır. Her değer bir Unicode karakter kodudur. Unicode, dünyadaki yazı sistemlerinin karakterlerine birer numara veren uluslararası bir standarttır. `char` sabitleri tek tırnakla yazılır: `'A'`, `'ş'`, `'7'`. Tek tırnak ile çift tırnak arasındaki fark önemlidir: `'A'` bir `char`, `"A"` ise tek karakterlik bir `String`'dir.

Bir karakterin kodunu bilmek, karakterlerle hesap yapmayı mümkün kılar. Üç kod sık kullanılır: `'A'` = 65, `'a'` = 97, `'0'` = 48. İngilizce alfabedeki harflerin ve rakamların kodları ASCII tablosundakilerle aynıdır ve ardışık dizilir: `'B'` = 66, `'C'` = 67, `'1'` = 49. `'7'` karakterinin kodu 7 değil, 55'tir. Rakam karakteri ile rakamın sayı değeri ayrı şeylerdir.

`boolean` mantıksal bir değer tutar. Yalnız iki değeri vardır: `true` (doğru) ve `false` (yanlış). Taşıdığı bilgi bir bittir; bellekte kaç bayt tutacağını JVM belirler. C dilinin aksine Java'da `boolean` bir sayı değildir. `boolean bitti = 0;` derlenmez, `if (1)` de yazılamaz. `boolean` değerler koşullarda ve karşılaştırma sonuçlarında kullanılır:

```
boolean gectiMi = puan >= 50;
```

Bu satırda `puan >= 50` karşılaştırmasının sonucu, yani `true` ya da `false`, `gectiMi` değişkenine yazılır. `puan` 72 ise `gectiMi` `true` olur.

```
char harf = 'A';
boolean dogruMu = true;
System.out.println(harf);
System.out.println(dogruMu);
System.out.println(!dogruMu);
```

```
A
true
false
```

`println` bir `char` aldığında karakterin kendisini yazar: `A`. Kodunu, yani 65'i yazmaz. `boolean` değerler `true` ve `false` sözcükleriyle yazılır. Üçüncü satırdaki `!` işareti mantıksal DEĞİL işlemidir ve `true`'yu `false`'a çevirir.

2.2.2.8 Sınırları program ile görmek

Tabloları ezberlemek yerine sınırları bir programla yazdırmak mümkündür. Her ilkel tipin bir **sarmalayıcı sınıfı** (*wrapper class*) vardır: `byte` için `Byte`, `int` için `Integer`, `double` için `Double`, `char` için `Character`. Bu sınıflar tipin sınırlarını `MIN_VALUE` ve `MAX_VALUE` sabitleriyle sunar.

```
System.out.println("byte  : " + Byte.MIN_VALUE + " .. " + Byte.MAX_VALUE);
System.out.println("short : " + Short.MIN_VALUE + " .. " + Short.MAX_VALUE);
System.out.println("int   : " + Integer.MIN_VALUE + " .. " + Integer.MAX_VALUE);
System.out.println("long  : " + Long.MIN_VALUE + " .. " + Long.MAX_VALUE);
System.out.println("float : " + Float.MIN_VALUE + " .. " + Float.MAX_VALUE);
System.out.println("double: " + Double.MIN_VALUE + " .. " + Double.MAX_VALUE);
System.out.println("char  : " + (int) Character.MIN_VALUE + " .. "
                              + (int) Character.MAX_VALUE);
```

```
byte  : -128 .. 127
short : -32768 .. 32767
int   : -2147483648 .. 2147483647
long  : -9223372036854775808 .. 9223372036854775807
float : 1.4E-45 .. 3.4028235E38
double: 4.9E-324 .. 1.7976931348623157E308
char  : 0 .. 65535
```

Çıktıdaki `4.9E-324` bilimsel gösterimdir ve $4{,}9 \times 10^{-324}$ anlamına gelir. `E` harfi "çarpı 10 üzeri" diye okunur. `float` ve `double` satırlarında soldaki değer en küçük *negatif* sayı değil, sıfırdan büyük en küçük sayıdır. Bu durum tam sayı tiplerinden farklıdır ve sık karıştırılır. `char` satırında `(int)` dönüşümü kullanılmıştır. Dönüşüm olmasaydı `println` bu iki değeri karakter olarak yazmaya çalışırdı. Kod 0 görünmeyen bir denetim karakteridir, 65.535 de yazdırılabilir bir karakter değildir.

2.2.2.9 Taşma

Tanım 2.2.9

Taşma (*overflow*), bir işlemin sonucunun tipin değer aralığına sığmamasıdır. Java'da tam sayı taşması hata vermez: değer aralığın öbür ucuna **sarar**.

Taşmayı saatle düşünmek kolaydır. Saat 12'yi gösterirken bir saat ilerletilirse 13'e değil, 1'e döner. Tam sayı tipleri de böyle davranır: en büyük değere 1 eklenince en küçük değere dönülür. Önemli olan, bu dönüşün sessizce olmasıdır. Program ne derleme hatası ne de çalışma hatası verir; yanlış sonuçla yoluna devam eder.

```
int enBuyuk = Integer.MAX_VALUE;
System.out.println(enBuyuk);
System.out.println(enBuyuk + 1);   // taşma!
byte sayi1 = 80;
byte sayi2 = 50;
byte toplam = (byte) (sayi1 + sayi2);
System.out.println(toplam);        // 130 değil
long dogru = (long) enBuyuk + 1;
System.out.println(dogru);
```

```
2147483647
-2147483648
-126
2147483648
```

Satırlar şöyle çalışır:

- `enBuyuk` `int`'in en büyük değerini, 2.147.483.647'yi tutar ve ilk satırda bu değer yazılır.
- `enBuyuk + 1` iki `int` arasındaki bir toplamadır ve sonucu da `int`'tir. Sonuç sığmaz ve en küçük değere sarar: `-2147483648`.
- `sayi1 + sayi2` 80 + 50 = 130 verir. 130, `byte` sınırı olan 127'yi aşar. `(byte)` dönüşümü sonucun son 8 bitini alır ve değer 130 − 256 = −126 olur.
- `(long) enBuyuk + 1` ifadesinde dönüşüm toplamadan önce yapılır. İşlem `long` tipinde yürür; 64 bitlik aralık 2.147.483.648'i rahatça tutar.



Şekil 2.2.14: Taşma. `byte` tipinde 127'ye 1 eklenince değer −128 olur; sayı ekseni bir çember gibi kapanır.

Çözüm daha geniş bir tip kullanmaktır. Ancak dönüşümün yeri önemlidir. `long yanlis = enBuyuk + 1;` satırı yine −2147483648 verir. Sağ taraf iki `int` arasında hesaplanır ve önce taşar; `long` değişkene ancak taşmış sonuç atanır. Dönüşüm işlemden önce yapılmalıdır: `(long) enBuyuk + 1` ya da `enBuyuk + 1L`.

Örnek 2.2.1

Bir milyonun karesi `int` ile hesaplanırsa ne çıkar? `int milyon = 1_000_000;` iken `milyon * milyon` ifadesinin gerçek değeri 10^{12}'dir; bu sayı `int` 'in yaklaşık 2,1 milyarlık sınırını çok aşar. Java'nın verdiği sonuç `-727379968`'dir. Bu değer, 10^{12}'nin ikili gösteriminin son 32 bitinin işaretli okunuşudur. Aynı hesap `(long) milyon * milyon` diye yazılınca doğru sonuç, `1000000000000` elde edilir. Sonucun negatif çıkması burada taşmayı ele verir. Her taşma bu kadar belirgin değildir; sonuç pozitif ama yanlış da olabilir.

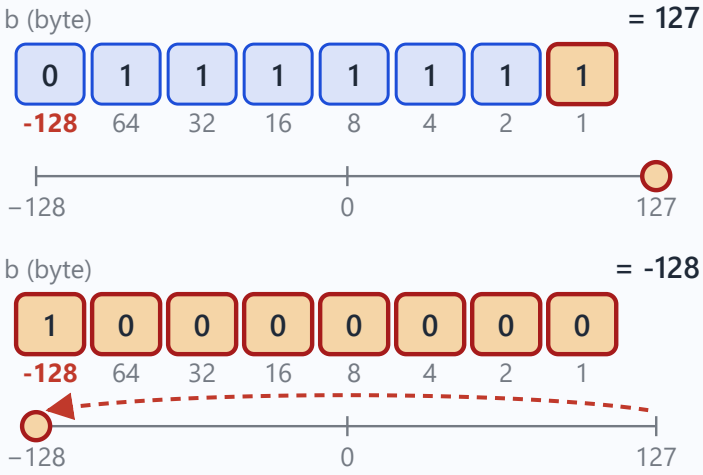
Taşmanın bit düzeyinde nasıl oluştuğunu bir `byte` sayacı üzerinde adım adım izleyelim:

```
byte b = 125;
for (int i = 1; i <= 4; i++) {
    b++;
    System.out.println(b);
}
```

`b` 125 ile başlar ve döngü dört kez çalışır. Tablo 2.2.6 her turdaki değeri ve ikili gösterimi verir.

`i`	`i <= 4`	`b++` sonrası	İkili	Çıktı
—	—	125 (başlangıç)	`01111101`	
1	doğru	126	`01111110`	126
2	doğru	127	`01111111`	127
3	doğru	**−128**	`10000000`	-128
4	doğru	−127	`10000001`	-127
5	yanlış	—	—	döngü biter

Tablo 2.2.6: `byte` sayacının izlemesi. En soldaki bit işaret bitidir ve ağırlığı −128'dir.



Şekil 2.2.15: Taşma anı. (a) `b` = 127, ikili `01111111` : `byte` 'ın en büyük değeri. (b) 1 eklenince ikili toplama `10000000` verir; işaret biti 1 olur ve değer −128'e sarar.

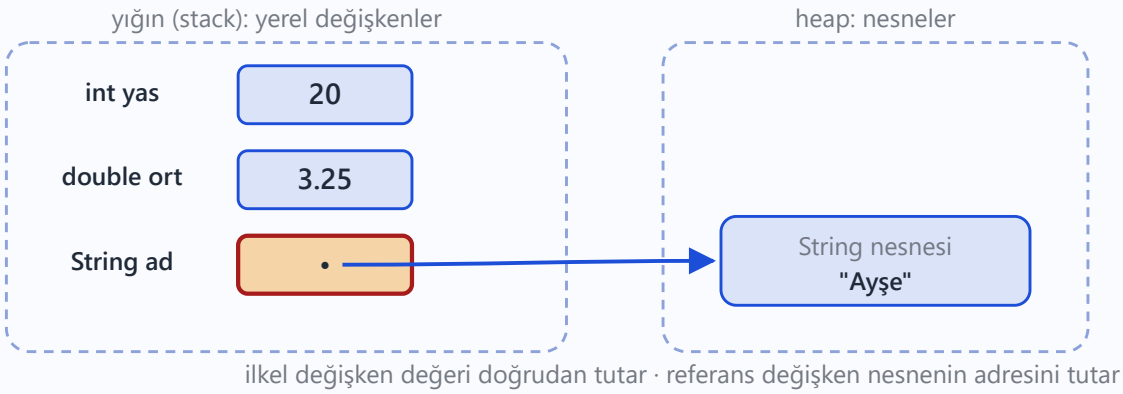
Taşmanın nedeni ikili toplamanın kendisidir. 127'nin ikili gösterimi `01111111` 'dir. Buna 1 eklenince bütün 1'ler sıfırlanır ve elde en soldaki bite geçer: `10000000` . Bu bit işaret bitidir ve ağırlığı +128 değil, −128'dir. Diğer bitler sıfır olduğu için değer −128 olur. Bu gösterime **ikiye tümleyen** (*two's complement*) gösterim denir. İşlemci aynı toplama devresini hem pozitif hem negatif sayılar için kullanabildiği için bu gösterim seçilmiştir. Program 126, 127, −128, −127 yazar ve hiçbir uyarı vermez.

Tablodaki `b++` satırı ayrıca dikkat ister. `b = b + 1;` yazılsaydı kod derlenmezdi, çünkü `b + 1` bir `int` ifadesidir ve `byte` 'a ancak açık dönüşümle atanabilir. `++` ve bileşik atama işleçleri ise dönüşümü kendileri yapar. Taşma da bu gizli dönüşümün sonucudur.

2.2.2.10 İlkel tipler ve referans tipleri

Tanım 2.2.10

İlkel tipteki bir değişken değerin kendisini saklar. **Referans tipindeki** (*reference type*) bir değişken ise bir nesneyi gösterir: değişkende nesnenin kendisi değil, bellekteki yeri, yani nesneye bir **başvuru** tutulur.



Şekil 2.2.16: İlkel ve referans tipleri. `int` ve `double` değişkenleri değeri doğrudan tutar; `String` değişkeni heap bellekteki nesneye bir okla başvurur.

İlkel tipler sekiz tanedir ve adları küçük harfle başlar. Geri kalan bütün tipler referans tipidir. `String` (karakter dizisi), diziler, sınıflar, arayüzler ve `enum`'lar bu gruptadır. Referans tiplerinin adları büyük harfle başlar. Nesneler genellikle `new` ile oluşturulur. `String` bu kuralın sık görülen bir istisnasıdır: `"Ali"` gibi bir dizgi sabiti, `new` yazılmadan da bir nesne oluşturur.

Hiçbir nesneyi göstermeyen bir referansın değeri `null`'dır. `null` ile boş dizgi karıştırılmamalıdır. Boş dizgi `""` gerçek bir nesnedir ve uzunluğu sıfırdır: `"".length()` 0 verir. `null` ise "burada nesne yok" demektir. `null` bir referans üzerinde metot çağrılırsa program `NullPointerException` ile durur. Yerel bir `String` değişkeni için üç durum birbirinden ayrıdır. Değer atanmamış değişkeni yazdıran kod derlenmez. `null` atanmış değişken `null` yazar. `""` atanmış değişken ise boş bir satır yazar.

İki tip grubunun farkı atamada da görülür. `int a = 5; int b = a;` satırlarından sonra `b`, 5'in ayrı bir kopyasını tutar; `b` değişince `a` etkilenmez. İki referans değişkeni ise aynı nesneyi gösterebilir. Bu ayrım nesne yönelimli programlamanın temel konularından biridir.

2.2.3 Tanımlayıcılar ve anahtar sözcükler

Bir programdaki değişkenlerin, metotların ve sınıfların her birinin bir adı vardır. Programcı bu adları seçmekte özgürdür, ama bu özgürlüğün sınırları bellidir. Bazı karakterler adlarda kullanılamaz, bazı sözcükler ise dile ayrılmıştır. Ayrıca Java topluluğunun uyduğu yazım alışkanlıkları vardır. Bu kesim üç düzeyi sırayla ele alır: derleyicinin zorunlu kuralları, ayrılmış sözcükler ve uzlaşımlar.

2.2.3.1 Tanımlayıcı kuralları

Tanım 2.2.11

Tanımlayıcı (*identifier*), değişkenlere, sınıflara, metotlara ve öteki program öğelerine verilen addır.

Tanımlayıcılar şu kurallara uyar:

- Yalnız **harf**, **rakam** (0–9), alt çizgi `_` ve dolar işareti `$` içerebilir. Boşluk, `-`, `@`, `.` gibi karakterler bulunamaz.
- İlk karakter harf, `_` ya da `$` olmalıdır. Tanımlayıcı **rakamla başlayamaz**.
- Anahtar sözcükler ve `true`, `false`, `null` sabitleri tanımlayıcı olamaz.
- Uzunluk sınırı yoktur.
- Büyük ve küçük harf ayrıdır: `kisiAdi` ile `kisiadi` farklı değişkenlerdir. Bildirimdeki yazımdan farklı yazılan bir ad `cannot find symbol` hatası verir.

Rakamla başlama yasağının bir nedeni vardır. Derleyici bir rakam gördüğünde sayı okumaya başlar. `2yas` ile karşılaşan derleyici önce 2 sayısını okur, ardından gelen `yas`'ı ayrı bir sözcük sayar. Aynı nedenle `ad-soyad`, "ad eksi soyad" diye okunur; çünkü `-` çıkarma işlecidir.

Java harf olarak yalnız İngilizce harfleri değil, bütün Unicode harflerini kabul eder. `öğrenci` adlı bir değişken derlenir ve çalışır. Yine de adlarda yalnız İngilizce harfler kullanılması önerilir. Dosyalar farklı karakter kodlamasıyla açılınca Türkçe harfler bozulabilir ve kod başka sistemlerde derlenmez hâle gelebilir. Bu yüzden adlar `öğrenciNo` yerine `ogrenciNo`, `sayaç` yerine `sayac` biçiminde yazılır. Geçerli örnekler: `kullaniciAdi`, `toplamPuan`, `ogrenci_adi`, `veri1`, `_sayac`.

2.2.3.2 Geçerli ve geçersiz adlar

Ad	Durum	Neden?
`ogrenciNo`	✓	harf ile başlıyor
`sayi_2`	✓	harf, rakam ve `_`
`$fiyat`	✓	`$` izinli, ama önerilmez
`2yas`	✗	rakamla başlıyor
`maas hesabi`	✗	boşluk içeriyor
`ad-soyad`	✗	`-` çıkarma işlecidir
`@isim`	✗	`@` izinli değil
`class`	✗	anahtar sözcük
`Class`	✓	büyük C: anahtar sözcük değil

Tablo 2.2.7: Tanımlayıcı örnekleri. ✓ geçerli, ✗ geçersiz.

`$` işareti geçerlidir ama derleyicinin kendi ürettiği sınıf adlarında kullanılır. Programcının bu işareti kullanmaması bu yüzden önerilir. `Class` derlenir, çünkü Java büyük-küçük harfe duyarlıdır ve anahtar sözcüklerin hepsi küçük harflidir. Ancak `Class`, Java API'sinde var olan bir sınıfın adıdır ve okuyanı şaşırtır.

Geçersiz bir ad her zaman açık bir ileti üretmez. Rakamla başlayan bir ad derleyiciyi şaşırtır; ileti adı değil, deyimi işaret eder:

```
int 2yas = 20;
```

```
Tanim.java:3: error: not a statement
        int 2yas = 20;
        ^
Tanim.java:3: error: ';' expected
        int 2yas = 20;
           ^
2 errors
```

Derleyici `int` sözcüğünden sonra bir ad bekler. `2`'yi görünce burada bir ad olmadığını anlar ve `int`'i tek başına anlamsız bir deyim sayar (`not a statement`). Ardından `2yas` kısmını yeni bir deyimin başı gibi okumaya çalışır ve noktalı virgül bekler. İki ileti de asıl sorunu, yani adın rakamla başlamasını söylemez. İletiler kafa karıştırıcı göründüğünde ilk bakılacak yer, satırdaki adların kurallara uyup uymadığıdır.

2.2.3.3 Anahtar sözcükler

Tanım 2.2.12

Anahtar sözcük (*keyword*, *reserved word*), dilde belirli bir görevi olan ve bu yüzden ayrılmış sözcüktür. Anahtar sözcükler değişken, sınıf ya da metot adı olarak kullanılamaz. Hepsi küçük harfle yazılır.

Java 21'de 51 anahtar sözcük vardır. Tablo 2.2.8 bunları görevlerine göre gruplar.

Grup	Anahtar sözcükler	Sayı
Tipler	`boolean byte char short int long float double void`	9
Kontrol akışı	`if else switch case default for while do break continue return`	11
Sınıf ve nesne	`class interface enum extends implements new this super instanceof package import`	11

Grup	Anahtar sözcükler	Sayı
Düzenleyiciler	`public protected private static final abstract native synchronized volatile transient strictfp`	11
İstisnalar	`try catch finally throw throws assert`	6
Ayrılmış, kullanılmıyor	`const goto _`	3

Tablo 2.2.8: Java 21'in 51 anahtar sözcüğü, görev gruplarına göre.

Üç sözcük sık sık anahtar sözcük sanılır: `true`, `false` ve `null`. Bunlar anahtar sözcük değil, **sabit değerdir** (*literal*). Yine de onlar da ad olarak kullanılamaz. Bir başka grup **bağlamsal kelimelerdir**. `var` (Java 10 ile), `yield` (Java 14 ile) ve `record` (Java 16 ile) gibi sözcükler yalnız belirli yerlerde özel anlam taşır. Dile sonradan eklendikleri için, eski programlarda değişken adı olarak kullanılmış olabilirler. Bu kodların bozulmaması için tam anahtar sözcük yapılmamışlardır. `String`, `System` ve `main` ise hiçbir gruba girmez. Bunlar API'deki sınıfların ve metotların adlarıdır. `int String = 3;` satırı bu yüzden derlenir, ama okuyanı şaşırtacağı için asla yazılmamalıdır.

Bir anahtar sözcük ad olarak kullanılınca derleyici, o sözcüğün başlattığı yapıyı bekler. `new` bir nesne oluşturma ifadesini başlatır; ardından bir sınıf adı ve parantez gelmelidir:

```
int new = 5;
```

```
Anahtar.java:3: error: not a statement
        int new = 5;
        ^
... (toplam 4 hata)
```

Tek bir hatalı sözcük dört ileti üretir. Derleyici `new`'den sonra bir sınıf adı (`<identifier> expected`) ve ardından `(` ya da `[` bekler. İletilerin hiçbiri "`new` anahtar sözcüktür" demez. Tanımlayıcı hatalarında olduğu gibi, anlamsız görünen bir ileti dizisi çoğu zaman adın kendisinde bir sorun olduğunu gösterir.

2.2.3.4 Anahtar sözcük grupları

Anahtar sözcüklerin çoğu ancak ilgili yapılar yazılırken tam anlamını kazanır. Başlangıçta her birinin ne iş gördüğünü kısaca bilmek yeterlidir. Tip anahtar sözcükleri ilkel tiplerin adlarıdır; onlara `void` eklenir. `void` bir tip değil, bir metodun değer döndürmediğini belirten işarettir: `public static void main`.

Sözcük	Görevi
`if` / `else`	Koşulu denetler; doğruysa `if` bloğu, değilse `else` bloğu çalışır.
`switch`	Bir değişkenin farklı değerlerine göre farklı işlemler yapar.

Sözcük	Görevi
`case`	`switch` içinde bir durumu (değeri) temsil eder.
`default`	`switch` içinde hiçbir durum eşleşmediğinde yapılacak işlem.
`for`	Bir işlemi bir sayaçla, koşul sağlandıkça tekrarlar.
`while`	Koşul sağlandığı sürece işlemi tekrarlar.
`do`	`do-while` döngüsü: gövde en az bir kez çalışır, koşul sonda denetlenir.
`break`	Döngüden ya da `switch` bloğundan hemen çıkar.
`continue`	Döngünün o turunun kalanını atlar, bir sonraki tura geçer.
`return`	Metodu bitirir; gerekiyorsa bir sonuç döndürür.

Tablo 2.2.9: Kontrol akışı anahtar sözcükleri.

Sözcük	Görevi
`class`	Sınıf tanımlar. Sınıflar nesnelerin şablonudur ve Java programlarının temel yapı taşıdır.
`interface`	Arayüz tanımlar: bir sınıfın sağlaması gereken metotları belirleyen sözleşme.
`enum`	Sabit değerlerden oluşan bir tip: `enum Renk { KIRMIZI, YESIL }`.
`extends`	Kalıtım: bir sınıfın başka bir sınıfın alan ve metotlarını alması.
`implements`	Bir sınıfın bir arayüzü uygulaması, yani arayüzün metotlarını sağlaması.
`new`	Nesne oluşturur: bellekte yer ayırır ve ilk değerleri verir.
`this`	Nesnenin kendisine başvuru; aynı adlı alanı yerel değişkenden ayırt eder.
`super`	Üst sınıfın metotlarına, alanlarına ve yapıcısına erişir.
`instanceof`	Bir nesnenin belirli bir sınıf ya da arayüz tipinden olup olmadığını denetler.
`package` / `import`	`package` sınıfın paketini (ad alanını) belirler; `import` başka paketteki sınıfları kullanıma açar.

Tablo 2.2.10: Sınıf ve nesne anahtar sözcükleri.

Sözcük	Görevi
`public`	Her yerden, bütün sınıflardan erişilebilir.
`protected`	Aynı paketten ve alt sınıflardan erişilebilir.
`private`	Yalnız aynı sınıfın içinden erişilebilir.
`static`	Üye nesneye değil sınıfa aittir; bütün nesneler arasında paylaşılır.
`final`	Değişken: değeri değişmez. Metot: alt sınıfta yeniden yazılamaz. Sınıf: kalıtılamaz.
`abstract`	Soyut sınıf (nesnesi oluşturulamaz) ya da gövdesiz metot; alt sınıf metodun gövdesini yazmak zorundadır.
`native`	Gövdesi Java dışında, platforma özgü kodla (C/C++) yazılmış metot; JNI ile bağlanır.
`synchronized`	Çok iş parçacıklı programda bir bloğa aynı anda tek iş parçacığının girmesini sağlar.
`volatile`	Değişkenin her iş parçacığında her zaman güncel değeriyle okunmasını sağlar.
`transient`	Nesne serileştirilirken (kaydedilirken) bu alan dikkate alınmaz.

Sözcük	Görevi
`strictfp`	Ondalıklı işlemlerin her platformda aynı sonucu vermesi. Java 17'den beri bu zaten varsayılandır; kelimenin etkisi kalmadı.

Tablo 2.2.11: Düzenleyiciler (*modifiers*).

İstisna grubundaki kelimeler program çalışırken oluşan hatalarla ilgilidir. `try` hata verebilecek kodu çevreler. `catch` bu blokta oluşan istisnayı yakalar ve işler. `finally` bloğu, istisna olsa da olmasa da en sonda çalışır; açılan bir dosyanın kapatılması bu bloğa yazılır. `throw` bir istisna nesnesi fırlatır ve normal akışı keser. `throws` metot başlığında, metodun fırlatabileceği istisna türlerini listeler. `assert` bir koşulun doğru olduğunu denetler. Bu denetim varsayılan olarak kapalıdır ve `java -ea` ile açılır.

Son grup ilginçtir. `const` ve `goto` ayrılmış ama hiçbir işlevi olmayan sözcüklerdir. C ve C++'tan gelen programcılar bu sözcükleri yanlışlıkla kullanmasın diye ayrılmışlardır. Tek başına alt çizgi `_` ise Java 9'dan beri ad olamaz. Java 21'de `int _ = 3;` yazılınca derleyici, bu kullanımın bir önizleme özelliği olduğunu ve kapalı olduğunu bildirir. Java 22 ile `_` "kullanılmayan değişken" anlamını kesin olarak kazandı.

2.2.3.5 Adlandırma kuralları

Derleyicinin zorunlu kurallarının yanında, Java topluluğunun uyduğu yazım alışkanlıkları da vardır. Bunlara **adlandırma kuralları** (*naming conventions*) denir.

Ne?	Kural	Örnekler
Sınıf ve arayüz	PascalCase, büyük harfle başlar	`String`, `Ogrenci`, `MerhabaDunya`
Değişken, alan, metot	camelCase, küçük harfle başlar	`firstName`, `classAverage`, `hesapla()`
İlkel tipler	küçük harf (anahtar sözcüktür)	`int`, `float`, `boolean`
İsimlendirilmiş sabit	BÜYÜK_HARF, kelimeler `_` ile ayrılır	`PI`, `GUNLER_HAFTADA`, `MAX_SAYI`
Paket	tamamen küçük harf	`syntax`, `tr.edu.giresun`

Tablo 2.2.12: Java adlandırma kuralları.

camelCase yazımında ilk kelime küçük, sonraki kelimeler büyük harfle başlar. Büyük harfler devenin hörgüçlerine benzetilir ve ad buradan gelir. **PascalCase** yazımında ise ilk kelime de dahil her kelime büyük harfle başlar. Bu ayrım kodu okuyan kişiye anında bilgi verir: `Ogrenci` bir sınıf, `ogrenci` bir değişken, `OGRENCI_SAYISI` bir sabittir.

```
class OgrenciKaydi { }    // sınıf
String isimSoyisim;       // değişken
boolean evliMi;           // evet/hayır
final int MAX_SAYI = 100; // sabit
```

Bunlar derleyici kuralı değil, **uzlaşımdır**. Uyulmazsa kod yine derlenir. `class ogrenci_kaydi` ya da `int ToplamPuan` hatasız çalışır. Ancak böyle bir kodu başka bir Java programcısının okuması zorlaşır. Uzlaşımlar ekip çalışmasının ortak dilidir. Java API'sinin kendisi de bu kurallara uyar: `String` bir sınıf, `length()` bir metot, `Integer.MAX_VALUE` bir sabittir.

2.2.4 Atama, sabitler ve dönüşüm

Değişkenler bildirildikten sonra değer alır ve bu değerler değişir. Bu kesim, değerin değişkene nasıl konduğunu, kodda doğrudan yazılan değerlerin nasıl okunduğunu ve bir değerin bir tipten ötekine nasıl geçtiğini anlatır. Tip dönüşümü, başlangıç düzeyindeki Java programlarında en çok hata çıkan konulardan biridir.

2.2.4.1 Atama ifadeleri

Tanım 2.2.13

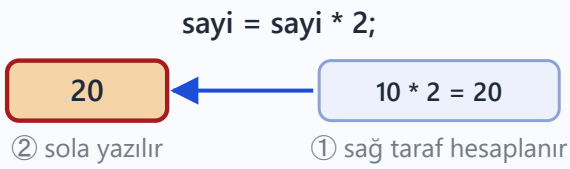
Atama (*assignment*), bir değişkene değer vermektir. `=` atama işleciyle yazılır: `değişken = ifade;` . Önce sağ taraftaki ifade hesaplanır, sonra sonuç sol taraftaki değişkene yazılır.

Sağ taraftaki ifade bir sabit değer, başka bir değişken ya da bir işlem olabilir:

```
int sayi, sonuc;
int sayi1 = 4, sayi2 = 6;
String isim;

sayi = 10;             // sabit değer
sonuc = sayi1 + sayi2; // toplam: 10
isim = "Ahmet";        // metin
sayi = sayi * 2;       // 10 → 20
```

İlk üç satır bildirimdir. `sayi1` ve `sayi2` bildirimle birlikte değer alır; `sayi` , `sonuc` ve `isim` ise henüz boştur. Ardından dört atama gelir ve her birinin sağ tarafı farklıdır. Birincisi bir sabittir (10), ikincisi iki değişkenin toplamıdır (4 + 6 = 10). Üçüncüsü bir metin (`"Ahmet"`), dördüncüsü değişkenin kendi değerini kullanan bir işlemdir. Son satırdan sonra `sayi` 20 olur.



Şekil 2.2.17: `sayi = sayi * 2` : önce sağ taraf 10 * 2 = 20 hesaplanır, sonra sonuç `sayi` kutusuna yazılır.

`=` işareti matematikteki eşitlik değildir. Matematikte *x = x + 1* çelişkili bir denklemdir; hiçbir sayı kendisinin bir fazlasına eşit olamaz. Java'da ise `x = x + 1` "`x`'in yeni değeri, eski değerinin bir fazlası olsun" demektir. Okurken "eşittir" yerine "alır" demek bu yanılgıyı önler: "`x`, `x + 1`'i alır."

Atamanın sol tarafında yalnız bir değişken bulunabilir. `5 = sayi;` satırı derlenmez:

```
Sol.java:4: error: unexpected type
        5 = sayi;
        ^
  required: variable
  found:    value
1 error
```

İleti, sol tarafta bir değişken beklendiğini (`required: variable`) ama bir değer bulunduğunu (`found: value`) söyler. Atamada tipler de uyumlu olmalıdır. `String` bir değişkene sayı, `int` bir değişkene metin atanamaz. `int x = 3; String s = x;` satırları `incompatible types: int cannot be converted to String` iletisini verir.

Atama aynı zamanda bir ifadedir ve bir değer üretir: atanan değeri. Bu yüzden `a = b = 0;` yazılabilir. Atama sağdan sola gruplanır; önce `b = 0` yapılır, bu ifadenin değeri olan 0 da `a`'ya atanır. İki değişken birden sıfırlanmış olur.

2.2.4.2 Bileşik atama

Tanım 2.2.14

Bileşik atama (*compound assignment*) işleçleri bir işlemi ve atamayı birleştirir: `+=`, `-=`, `*=`, `/=`, `%=`. `x op= y` yazımı, `x = x op y` ile aynı işi yapar.

```
int x = 5;
int y = 10;
x = x + y;
x -= y;
x *= 2;
x /= 4;
x %= 3;
System.out.println("x = " + x + ", y = " + y);
```

Program `x = 2, y = 10` yazar. Tablo 2.2.13 her satırdan sonraki değerleri verir.

Satır	Açık yazımı	Hesap	x	y
`int x = 5;`	—	—	5	
`int y = 10;`	—	—	5	10
`x = x + y;`	—	5 + 10	**15**	10
`x -= y;`	`x = x - y`	15 − 10	**5**	10
`x *= 2;`	`x = x * 2`	5 × 2	**10**	10
`x /= 4;`	`x = x / 4`	10 / 4 (tam sayı bölmesi)	**2**	10
`x %= 3;`	`x = x % 3`	2'nin 3'e bölümünden kalan	**2**	10

Tablo 2.2.13: Bileşik atama izlemesi.



Şekil 2.2.18: Bileşik atamanın üç anı. (a) `x = x + y` sonrası `x` = 15. (b) `x *= 2` sonrası `x` = 10. (c) `x /= 4` sonrası `x` = 2; `y` baştan sona 10'dur.

İki adım dikkat ister. `x /= 4` satırında 10 / 4 iki tam sayı arasında yapılır ve sonuç 2,5 değil 2'dir; kesirli kısım atılır. `x %= 3` satırında 2, 3'ten küçük olduğu için bölüm 0, kalan 2'dir. `y` hiçbir satırda sol tarafta olmadığı için değişmez.

Bileşik atamanın iki inceliği vardır. Birincisi, sağ taraf bütün olarak hesaplanır, sonra işlem yapılır. `k *= 2 + 3` ifadesi `k = k * 2 + 3` değil, `k = k * (2 + 3)` demektir; `k` 7 ise sonuç 35'tir. İkincisi, bileşik atama sonucu sol taraftaki değişkenin tipine kendisi dönüştürür. `byte b = 10; b += 5;` derlenir ve `b` 15 olur. Oysa `b = b + 5;` derlenmez, çünkü `b + 5` bir `int` 'tir. Bu kolaylığın bir bedeli de vardır: `int m = 10; m /= 0.3;` hata vermeden derlenir ve `m` 33 olur. 10 / 0,3 = 33,33… hesaplanır, ardından kesir sessizce atılır.

2.2.4.3 İlk değer ve varsayılan değerler

Bir değişken değer atanmadan bildirilebilir. İlk değer bildirimle birlikte de verilebilir: `int yas = 25;` . Bu işleme **ilk değer verme** (*initialization*) denir. Değerin ne zaman verildiği, değişkenin nerede bildirildiğine göre iki farklı kurala bağlıdır.

Yerel değişkenler, yani bir metodun içinde bildirilen değişkenler, varsayılan değer almaz. Değer atanmadan okunan yerel değişken programın derlenmesini engeller:

```
int sayac;
System.out.println(sayac);
```

```
Ilk.java:4: error: variable sayac
  might not have been initialized
```

Derleyici programın her olası yolunu inceler ve `sayac`'ın okunduğu noktaya değer almadan ulaşılabildiğini görür. Bu denetim, rastgele bir bellek içeriğinin sessizce kullanılmasını önler. C dilinde aynı hata derlenir ve program her çalıştırılışında farklı, anlamsız bir sayı yazabilir.

Sınıf alanları ise otomatik olarak **varsayılan değer** (*default value*) alır. Tablo 2.2.14 bu değerleri verir.

Alan tipi	Varsayılan
`int`, `long`, `byte`, `short`	0
`double`, `float`	0.0
`char`	`'\u0000'` (kod 0)
`boolean`	`false`
`String` ve öteki referanslar	`null`

Tablo 2.2.14: Sınıf alanlarının varsayılan değerleri.

Bir sınıfın değer verilmemiş `int`, `long`, `double`, `float`, `char` (kodu), `boolean` ve `String` alanları sırayla yazdırılınca `0 0 0.0 0.0 0 false null` çıkar. Yine de varsayılan değerlere güvenmemek ve değeri açıkça vermek iyi bir alışkanlıktır. Açıkça yazılan `int sayac = 0;`, okuyan kişiye sıfırın bilinçli bir seçim olduğunu söyler.

2.2.4.4 Sabit değerler

Tanım 2.2.15

Sabit değer (*literal*), kaynak kodda doğrudan yazılan değerdir: `2`, `3.7`, `'y'`, `"Ali"`, `true` gibi.

Her sabit değerin bir tipi vardır ve bu tip yazılış biçiminden anlaşılır:

- **Tam sayı:** `+` ya da `-` ile başlayabilir ama **virgül içeremez**: `+42`, `-17`. Okunabilirlik için basamaklar arasına alt çizgi konabilir (Java 7 ile geldi): `1_000_000`. Onaltılık sayılar `0x`, ikilik sayılar `0b` ile başlar: `0x1F`, `0b1010`.
- **Ondalıklı:** ondalık ayırıcı noktadır. Bilimsel gösterim `e` ile yazılır: `2.5e3` = 2,5 × 10^{3}, `7.68e-2` = 7,68 × 10^{-2}.
- **Karakter:** tek tırnak içinde: `'y'`. Özel karakterler ters bölü ile yazılır: `'\n'` yeni satır, `'\t'` sekme, `'\''` tek tırnak, `'\\'` ters bölü.
- **Metin:** çift tırnak içinde: `"Merhaba"`. **Mantıksal:** `true`, `false`.

```
System.out.println(2.5e3);
System.out.println(7.68e-2);
System.out.println(1_000_000);
System.out.println(0x1F);
System.out.println(0b1010);
System.out.println(3000000000L);
System.out.println("Ali\tVeli\n\"Java\"");
```

```
2500.0
0.0768
1000000
31
10
3000000000
Ali     Veli
"Java"
```

Çıktı birkaç şeyi gösterir. `2.5e3` bir `double`'dır ve `2500.0` diye yazılır; tam sayı değerli olsa da sonundaki `.0` tipini belli eder. Alt çizgiler yalnız kaynak kodda okunurluk içindir, çıktıda görünmez. `0x1F` onaltılıkta 1 × 16 + 15 = 31, `0b1010` ikilikte 8 + 2 = 10'dur; `println` ikisini de onluk tabanda yazar. Son satırdaki metinde `\t` bir sekme, `\n` bir satır sonu üretir; `\"` ise metnin içine çift tırnak koyar. Bu yüzden tek bir `println` iki satır yazar.

Sık yapılan hata

Türkçedeki ondalık virgülü koda taşımak. `double oran = 0,45;` derlenmez; Java'da ondalık ayırıcı noktadır: `0.45`. Benzer biçimde binlik ayırıcı nokta da yazılamaz: `1.000.000` anlamsızdır. Bin, `1000` ya da `1_000` diye yazılır. Bir başka tuzak da başa sıfır koymaktır: `010` sekizlik tabanda okunur ve değeri 10 değil, 8'dir.

2.2.4.5 Ondalık sayıların hassaslığı

Ondalık sayılar sınırlı sayıda bitle ve ikili tabanda saklanır. Onluk tabanda 1/3 = 0,333… sonsuza dek sürer ve sonlu basamakla tam yazılamaz. İkili tabanda da benzer bir durum vardır: 1/10 = $0{,}000110011\ldots_2$ diye sonsuz tekrar eder. Onluk tabanda masum görünen 0,1 bu yüzden bilgisayarda yaklaşık saklanır. `double`'daki 0,1'in tam değeri 0,1000000000000000055511151231257827… gibi bir sayıdır.

```
System.out.println(1.0 / 3.0);
System.out.println(1.0/3 + 1.0/3 + 1.0/3);
System.out.println(0.1 + 0.2);
System.out.println(0.1 + 0.2 == 0.3);
double t = 0.0;
for (int i = 0; i < 10; i++) t += 0.1;
System.out.println(t);
double fark = (0.1 + 0.2) - 0.3;
System.out.println(Math.abs(fark) < 1e-9);
```

```
0.3333333333333333
1.0
0.30000000000000004
false
0.9999999999999999
true
```

Sonuçlar satır satır şöyle okunur:

- `1.0 / 3.0` yaklaşık bir değerdir ve 16 basamakla yazılır.
- Üç tane `1.0/3`'ün toplamı tam `1.0` çıkar. Burada yuvarlama hataları birbirini götürmüştür. Bu bir şanstır, kural değildir.
- `0.1 + 0.2` sonucu `0.30000000000000004`'tür. İki sayının küçük hataları toplanır ve sonuç `0.3` sabitinin saklanan değerinden farklı olur. Bu yüzden `0.1 + 0.2 == 0.3` `false` verir.
- 0,1'in on kez toplanması tam 1 etmez: `0.9999999999999999`. Küçük hatalar her toplamada birikir.
- Son satırda iki sayının farkının mutlak değeri çok küçük bir sınırla, 10^{-9} ile karşılaştırılır. Fark yaklaşık $5{,}55 \times 10^{-17}$'dir; sınırdan küçük olduğu için sonuç `true`'dur.

Bu davranış Java'ya özgü değildir. Bütün modern işlemciler kayan noktalı sayıları IEEE 754 standardına göre saklar ve C, Python, JavaScript aynı sonuçları verir. Bundan iki kural çıkar. Birincisi, ondalıklı sayılar `==` ile karşılaştırılmaz; farkın çok küçük olup olmadığına bakılır: `Math.abs(a - b) < 1e-9`. İkincisi, kuruşu kuruşuna doğru olması gereken para hesaplarında `double` kullanılmaz. Bunun için `BigDecimal` sınıfı vardır; `new BigDecimal("0.1")` ile `new BigDecimal("0.2")`'nin toplamı tam olarak 0,3'tür. Bir başka yol, tutarları kuruş cinsinden `long` tam sayılarla tutmaktır.

Sık yapılan hata

Ondalıklı bir sayaçla döngü kurmak. `for (double t = 0; t != 1.0; t += 0.1)` döngüsü hiç bitmez. `t` 1,0'a tam olarak hiç ulaşmaz; 0,9999999999999999'dan sonra 1,0999999999999999'a geçer ve `t != 1.0` hep doğru kalır. Döngü sayacı tam sayı olmalıdır; ondalıklı değer gerekiyorsa sayaçtan hesaplanır: `double t = i * 0.1;` .

2.2.4.6 İsimlendirilmiş sabitler

Tanım 2.2.16

İsimlendirilmiş sabit (*named constant*), `final` ile bildirilen ve değeri bir kez verildikten sonra değiştirilemeyen değişkendir. Sınıf düzeyindeki kalıbı şudur: `public static final Tip SABIT_ADI = değer;`.

Kodun içine gömülü, anlamı belirtilmemiş sayılara **sihirli sayı** denir. `gun = 4 * 7;` satırındaki 7'nin haftanın gün sayısı olduğu, ancak bağlamdan tahmin edilebilir. Aynı satır `gun = 4 * GUNLER_HAFTADA;` diye yazılınca anlam kendiliğinden ortaya çıkar. Değer değişirse de tek bir yerden güncellenir. Sabit adları büyük harfle yazılır, kelimeler alt çizgiyle ayrılır.

```
public class Sabitler {
    public static final double PI = 3.14159;
    public static final int GUNLER_HAFTADA = 7;

    public static void main(String[] args) {
        double r = 2.0;
        double alan = PI * r * r;
        System.out.println("Alan: " + alan);
        int gun = 4 * GUNLER_HAFTADA;
        System.out.println(gun + " gün");
    }
}
```

```
Alan: 12.56636
28 gün
```

`PI` ve `GUNLER_HAFTADA` sınıf düzeyinde, `main`'in dışında bildirilmiştir. `static` olmaları, nesne oluşturulmadan `main` içinde kullanılabilmelerini sağlar. Alan 3,14159 × 2 × 2 = 12,56636 bulunur. Java'nın kendisi de daha duyarlı bir değer sunar: `Math.PI`. Aynı yarıçapla `Math.PI * r * r` 12,566370614359172 verir.

Sabiti değiştirmeye çalışmak ya da `double` bir sonucu `float`'a atamak derlenmez:

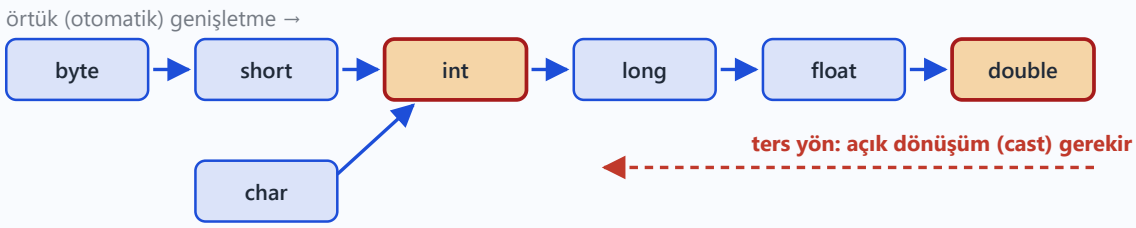
```
PI = 3.14;
float alan = PI * r * r;
```

```
error: cannot assign a value to
  static final variable PI
error: incompatible types: possible
  lossy conversion from double to float
```

Birinci ileti `final` anahtar sözcüğünün işini yaptığını gösterir. Değişmemesi gereken bir değer yanlışlıkla değiştirilmek istenince hata derleme sırasında yakalanır. İkinci ileti bir tip dönüşümü sorunudur. `PI * r * r` bir `double` ifadesidir ve `float` daha dar bir tiptir. Derleyici "olası kayıplı dönüşüm" (`possible lossy conversion`) uyarısıyla atamayı reddeder.

2.2.4.7 Atama uyumluluğu

Bir değer, başka tipte bir değişkene her zaman atanamaz. Java'nın kuralı basittir: değer kayıpsız sığıyorsa atama kendiliğinden yapılır, kayıp olabilecekse programcının açıkça istemesi gerekir.



Şekil 2.2.19: Atama uyumluluğu. `byte` , `short` , `int` , `long` , `float` , `double` zincirinde sağa doğru atama otomatik yapılır; `char` `int` 'e atanabilir. Ters yön dönüşüm işareti ister.

Tanım 2.2.17

Genişletme dönüşümü (*widening conversion*), bir değerin daha geniş aralıklı bir tipe geçmesidir ve otomatik yapılır. **Daraltma dönüşümü** (*narrowing conversion*) ise daha dar aralıklı bir tipe geçiştir; değer kaybı olabileceği için açık dönüşüm ister.

Şekil 2.2.19 zinciri gösterir: `byte → short → int → long → float → double` . Bir değer, zincirde daha sağdaki bir tipin değişkenine doğrudan atanabilir. `int` bir değer `long` 'a, `long` bir değer `double` 'a sorunsuz geçer. `char` zincire `int` 'ten girer: bir `char` değeri `int` 'e ve onun sağındaki tiplere atanabilir, tersi derlenmez.

```
char karakter = 'A';
int tamsayi;
tamsayi = karakter;   // mümkün: 65
karakter = tamsayi;   // hata
```

```
Uyum.java:5: error: incompatible types:
  possible lossy conversion from int to char
        karakter = tamsayi;
                   ^
```

`tamsayi = karakter;` satırında `'A'`'nın kodu olan 65, `int` değişkene yazılır. Ters yöndeki atama ise reddedilir. `tamsayi`'nın o anda 65 olduğu bilinse bile derleyici değere değil, tipe bakar. `int` bir değişken −5 ya da 100.000 gibi `char`'a sığmayan bir değer de taşıyabilir.

Zincirin bir inceliği vardır. `long` 64 bit, `float` ise 32 bittir; yine de `long → float` otomatik izinlidir. Bunun nedeni, `float`'ın aralığının `long`'unkinden çok daha geniş olmasıdır. Ancak `float` yalnız 7 basamak kadar duyarlıdır. Çok büyük bir sayıda son basamaklar yuvarlanır. `long buyuk = 123456789123L;` değeri `float`'a atanıp geri okununca `123456790528` elde edilir. Aynı durum `int → float` için de geçerlidir: 16.777.217 `float`'a atanınca 16.777.216 olur. Genişletme aralık bakımından güvenlidir, ama duyarlık bakımından her zaman değil.

2.2.4.8 Örtük ve açık dönüşüm

Tanım 2.2.18

Tip dönüşümü (*type conversion*), bir değeri başka bir tipe çevirmektir. Java'nın kendiliğinden yaptığı dönüşüme **örtük dönüşüm** (*implicit*), programcının hedef tipi parantez içinde yazarak istediği dönüşüme **açık dönüşüm** (*explicit conversion*, *cast*) denir: `(int) d`.

Küçük tipten büyük tipe dönüşümü Java kendiliğinden yapar. Büyük tipten küçük tipe dönüşümde ise hedef tip parantez içinde yazılır. Programcı bu işaretle olası kaybı kabul ettiğini derleyiciye bildirmiş olur. Kayıp iki biçimde ortaya çıkar:

- Ondalıklı sayıdan tam sayıya geçerken kesirli kısım **atılır**; sayı yuvarlanmaz. `(int) 3.99` 3, `(int) -3.99` −3 verir. Kesir sıfıra doğru kesilir.
- Sığmayan bir tam sayı taşar. `(byte) 130` değerin son 8 bitini alır ve −126 verir.

```
double ondalikDegisken = 5;     // örtük: 5.0
System.out.println(ondalikDegisken);
double d = 5.0;
int tamSayiDegisken = (int) d;   // açık: 5
System.out.println(tamSayiDegisken);
System.out.println((int) 3.99);
System.out.println((int) -3.99);
System.out.println(Math.round(3.99));
System.out.println((byte) 130);
```

```
5.0
5
3
-3
4
-126
```

`double ondalikDegisken = 5;` satırında `int` sabiti 5 örtük olarak 5.0'a çevrilir; `println` bu yüzden `5.0` yazar. `(int) d` açık dönüşümdür ve 5.0'ı 5'e çevirir. Bu değerde kayıp yoktur, ama derleyici değere değil tipe baktığı için dönüşüm işareti yine gerekir. `(int) 3.99` ile `(int) -3.99` kesirin atıldığını gösterir. Yuvarlama isteniyorsa `Math.round` kullanılır: `Math.round(3.99)` 4 verir. `(byte) 130` taşmanın açık dönüşümle de oluştuğunu gösterir. Dönüşüm işareti hatayı önlemez; yalnız derleyicinin itirazını susturur.

Sık yapılan hata

`Math.round`'un dönüş tipini unutmak. `Math.round(double)` `int` değil, `long` döndürür. `int n = Math.round(3.99);` bu yüzden `possible lossy conversion from long to int` iletisiyle derlenmez; `int n = (int) Math.round(3.99);` yazılır. Bir başka ayrıntı da negatif yarımların yuvarlanmasıdır: `Math.round(2.5)` 3, `Math.round(-2.5)` ise −2 verir. Yöntem, sayıya 0,5 ekleyip aşağı yuvarlar.

2.2.4.9 Tam sayı bölmesi ve dönüşümün yeri

Tanım 2.2.19

Tam sayı bölmesi (*integer division*), iki tam sayının bölümünün yine tam sayı olarak hesaplanmasıdır: kesirli kısım atılır. **Sayısal yükseltme** (*numeric promotion*) ise bir işlemdeki işlenenlerin ortak bir tipe çevrilmesidir: bir işlenen `double` ise öteki de `double`'a yükseltilir.

Java bir işlemin tipine işlenenlerin tipine bakarak karar verir. Sonucun atanacağı değişkenin tipi bu kararı etkilemez. Tablo 2.2.15 aynı 7 ve 2 sayılarıyla yazılmış altı ifadeyi karşılaştırır.

İfade	Sonuç	Neden?
`7 / 2`	3	iki `int` → tam sayı bölmesi, kesir atılır
`7 / 2.0`	3.5	`2.0` `double` → işlem `double` yapılır
`(double) 7 / 2`	3.5	önce 7 → 7.0, sonra bölme
`(double) (7 / 2)`	3.0	önce tam sayı bölmesi (3), sonra 3 → 3.0: geç kalınmış dönüşüm
`(int) 3.99`	3	kesirli kısım atılır

İfade	Sonuç	Neden?
`Math.round(3.99)`	4	en yakın tam sayıya yuvarlar (`long` döndürür)

Tablo 2.2.15: Tam sayı bölmesi ve dönüşümün yeri.

Üçüncü ve dördüncü satırlar arasındaki fark, dönüşüm işaretinin önceliğinden gelir. `(double)` tekli bir işleçtir ve bölmeden önce uygulanır. `(double) 7 / 2` ifadesinde dönüşüm yalnız 7'ye uygulanır; bölme `7.0 / 2` olarak yapılır. `(double) (7 / 2)` ifadesinde ise parantez önce hesaplanır. Tam sayı bölmesi 3'ü verir ve dönüşüm, kesir çoktan kaybolduktan sonra gelir.

Yükseltme kuralının bir sonucu daha vardır. `byte` ve `short` değerler bir işleme girerken önce `int`'e yükseltilir. `byte + byte` işleminin sonucu bu yüzden `int`'tir:

```
Toplam.java:4: error: incompatible types: possible lossy conversion from int to byte
        byte c = a + b;
                   ^
1 error
```

`a` ve `b` küçük sayılardır ve toplamları 30 da `byte`'a sığar. Yine de derleyici, iki `byte`'ın toplamının en çok 254 olabileceğini ve sığmayabileceğini bilir. Çözüm `byte c = (byte) (a + b);` yazmak ya da sonucu `int` bir değişkende tutmaktır.

Dönüşümün yerini adım adım izlemek için şu kodu ele alalım:

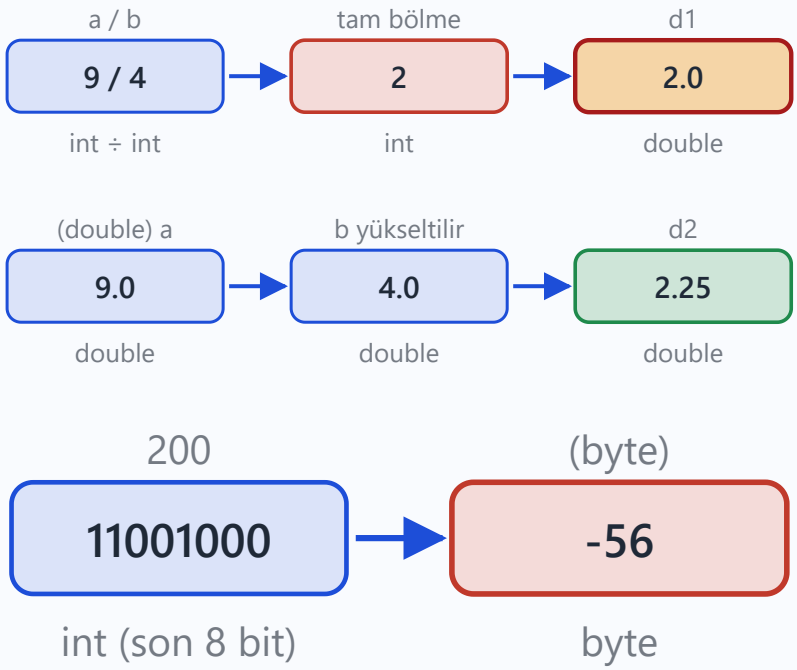
```
int a = 9, b = 4;
double d1 = a / b;
double d2 = (double) a / b;
double d3 = (double) (a / b);
int k = (int) 9.75;
long r = Math.round(9.75);
byte y = (byte) 200;
System.out.println(d1 + " " + d2 + " " + d3);
System.out.println(k + " " + r + " " + y);
```

Program iki satır yazar: `2.0 2.25 2.0` ve `9 10 -56`. Tablo 2.2.16 her değişkenin nasıl hesaplandığını verir.

Değişken	İfade	Ara adımlar	Değer
`d1`	`a / b`	9 / 4 = 2 (tam sayı bölmesi) → atamada 2.0	2.0
`d2`	`(double) a / b`	9 → 9.0; `b` 4.0'a yükseltilir; 9.0 / 4.0	2.25
`d3`	`(double) (a / b)`	9 / 4 = 2 → 2.0	2.0
`k`	`(int) 9.75`	kesir atılır	9
`r`	`Math.round(9.75)`	en yakın tam sayı (`long`)	10

Değişken	İfade	Ara adımlar	Değer
`y`	`(byte) 200`	200 = `11001000` ; işaret biti 1 → 200 − 256	−56

Tablo 2.2.16: Tip dönüşümü izlemesi (`a` = 9, `b` = 4).



Şekil 2.2.20: Dönüşümün yeri. (a) `d1` : önce tam sayı bölmesi 2'yi verir, sonra 2 → 2.0. (b) `d2` : önce 9 → 9.0, bölme `double` yapılır ve 2.25 bulunur. (c) `y` : 200'ün son 8 biti `11001000` , işaretli okununca −56'dır.

`d1` satırı en sık yapılan hatayı gösterir. Hedefin `double` olması bölmeyi kurtarmaz. Java önce sağ tarafı hesaplar; o sırada sonucun nereye atanacağına bakmaz. İki `int` bölünür, 2 bulunur ve ancak atamada 2.0'a çevrilir. `y` satırında 200 ikili tabanda `11001000` 'dir. Bu 8 bit `byte` olarak okunduğunda en soldaki 1, −128 ağırlığını taşır: −128 + 64 + 8 = −56.

Örnek 2.2.2

Santigrat dereceyi Fahrenhayta çeviren formül *F = C × 9 / 5 + 32*'dir. `int c = 37;` için üç farklı yazımın sonucu nedir?

- `c * 9.0 / 5 + 32` : 9.0 bir `double` olduğu için işlem baştan `double` yürür. 37 × 9,0 = 333,0; 333,0 / 5 = 66,6; 66,6 + 32 = **98,6**.
- `c * 9 / 5 + 32` : her şey `int` 'tir. 37 × 9 = 333; 333 / 5 = 66 (kesir atılır); 66 + 32 = **98**. Sonuç 0,6 eksiktir.
- `9 / 5 * c + 32` : işlemler soldan sağa yapılır. 9 / 5 = 1 (tam sayı bölmesi); 1 × 37 = 37; 37 + 32 = **69**. Sonuç neredeyse 30 derece yanlıştır.

> Matematikte bu üç yazım aynıdır. Java'da ise işlemin sırası ve işlenenlerin tipi sonucu değiştirir. Tam sayı bölmesinin ne kadar erken yapıldığı, kaybın büyüklüğünü belirler.

2.2.4.10 char ile int arasında dönüşüm

Her `char` aslında bir sayıdır: karakterin Unicode kodu. `'A'` = 65, `'B'` = 66, `'a'` = 97. Bu yüzden karakterler arasında hesap yapılabilir. Kurallar öteki sayısal tiplerle aynıdır:

- `char → int` örtüktür. `int → char` için `(char)` gerekir.
- İki `char` `+` ile toplanırsa ikisi de `int`'e yükseltilir ve bir sayı çıkar. Metin elde etmek için ifadenin bir `String` ile başlaması gerekir.
- İngilizce alfabede büyük ve küçük harflerin kodları arasında 32 fark vardır: `'a' - 'A'` = 97 − 65 = 32.
- Rakamların kodları 48'den başlar ve ardışık dizilir. Bu yüzden `'7' - '0'` rakamın sayı değerini, yani 7'yi verir.

Karakter kodlarının tamamı ASCII tablosunda görülebilir.

```
char karakter = 'A';
int kod = karakter;               // 65
System.out.println(kod);
char sonraki = (char) (kod + 1);
System.out.println(sonraki);
System.out.println('B' + 'C');
System.out.println("" + 'B' + 'C');
System.out.println((char) (karakter + 32));
System.out.println((int) 'ş');
```

```
65
B
133
BC
a
351
```

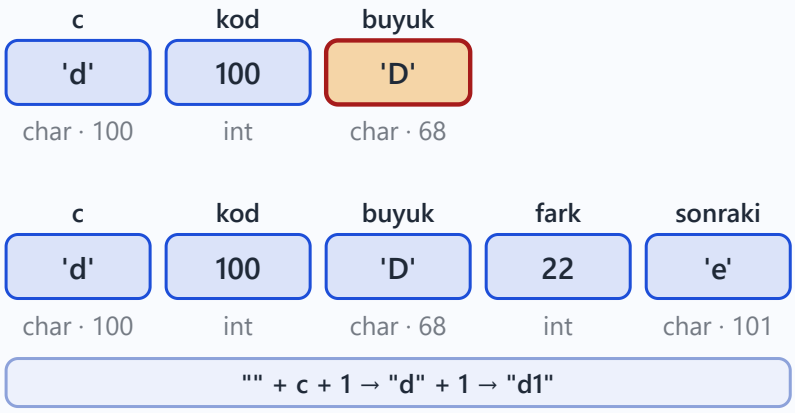
`'B' + 'C'` 66 + 67 = 133 verir; burada hiçbir metin yoktur. `"" + 'B' + 'C'` ifadesi ise boş bir dizgiyle başlar. Soldan sağa önce `"" + 'B'` hesaplanır ve `"B"` metni oluşur. Ardından `'C'` bu metne eklenir: BC. `(char) (karakter + 32)` büyük A'yı küçük a'ya çevirir. Son satır Türkçe `'ş'`'nin kodunu verir: 351. Türkçe harfler İngilizce harflerin arasında değil, Unicode tablosunun daha ileri bir bölümündedir. Bu yüzden 32 farkı kuralı onlar için geçerli değildir: `'Ş'` 350, `'ı'` 305, `'İ'` 304 kodludur.

Bu kuralları bir izleme üzerinde birleştirelim:

```
char c = 'd';
int kod = c;
char buyuk = (char) (c - 32);
int fark = 'z' - c;
char sonraki = c;
sonraki++;
System.out.println(kod + " " + buyuk + " " + fark + " " + sonraki);
System.out.println(c + 1);
System.out.println("" + c + 1);
```

Satır	Hesap	Sonuç
`char c = 'd';`	`'d'` 'nin kodu saklanır	`c` = `'d'` (100)
`int kod = c;`	örtük `char → int`	`kod` = 100
`char buyuk = (char) (c - 32);`	100 − 32 = 68 (`int`), `(char)` ile 68 = `'D'`	`buyuk` = `'D'`
`int fark = 'z' - c;`	122 − 100	`fark` = 22
`char sonraki = c; sonraki++;`	100 → 101; `++` dönüşüm işareti istemez	`sonraki` = `'e'`
`println(kod + " " + … )`	`" "` metni olduğu için birleştirme	`100 D 22 e`
`println(c + 1)`	`char` sayıya yükseltilir: 100 + 1	`101`
`println("" + c + 1)`	önce `"d"` , sonra `+ 1` metne eklenir	`d1`

Tablo 2.2.17: `char ⇔ int` izlemesi.



Şekil 2.2.21: `char ⇔ int` izlemesinden iki an. (a) `c` = `'d'` , `kod` = 100 ve `buyuk` = `'D'` hesaplanmıştır. (b) Bütün değişkenler hazırdır; son iki satır `101` ve `d1` yazar.

Tablodaki son iki satır aynı değerlerle çok farklı sonuçlar üretir. `c + 1` ifadesinde metin yoktur. `char` sayıya yükseltilir ve `101` yazılır; harf yazılmaz. Harf isteniyorsa `(char) (c + 1)` yazılır ve `e` çıkar. `"" + c + 1` ifadesinde ise soldaki boş dizgi bütün ifadeyi metin işlemine çevirir. `c` metne `d` olarak, 1 de `1` olarak eklenir. `sonraki++` satırı da dikkat çekicidir. `sonraki = sonraki + 1;` derlenmezdi, çünkü sağ taraf `int` 'tir. `++` ise bileşik atama gibi dönüşümü kendisi yapar.

2.2.5 İşleçler

İşleçler değerler üzerinde işlem yapan simgelerdir. `+`, `<` ve `&&` birer **işleçtir** (*operator*); üzerinde işlem yaptıkları değerlere **işlenen** (*operand*) denir. Java'nın işleçleri dört gruba ayrılır: aritmetik, karşılaştırma, mantıksal ve atama işleçleri. Bir ifadede birden çok işleç bulununca hangisinin önce uygulanacağını öncelik kuralları belirler.

2.2.5.1 Aritmetik işleçler

İşleç	Anlamı	İfade	Sonuç
`+`	toplama	`x + y`	22
`-`	çıkarma	`x - y`	12
`*`	çarpma	`x * y`	85
`/`	bölme	`x / y`	3
`%`	kalan (mod)	`x % y`	2
		`-17 / 5`	−3
		`-17 % 5`	−2
		`17.0 / 5`	3.4

Tablo 2.2.18: Aritmetik işleçler (`x` = 17, `y` = 5).



Şekil 2.2.22: 17 nesne 5'erli gruplara ayrılınca 3 grup oluşur ve 2 nesne artar: `17 / 5` = 3, `17 % 5` = 2.

`/` iki tam sayı arasında tam sayı bölmesi yapar. `%` ise bölümden kalanı verir. Şekil 2.2.22 ikisini birlikte gösterir: 17 = 5 × 3 + 2. Bölüm ve kalan her zaman bu eşitliği sağlar: `(x / y) * y + x % y` ifadesi `x`'e eşittir. Negatif sayılarda Java bölümü sıfıra doğru keser: −17 / 5 = −3,4 değil −3'tür. Eşitliğin korunması için kalan −2 olur: 5 × (−3) + (−2) = −17. Kural şudur: kalanın işareti bölünenin işaretidir. `17 % -5` bu yüzden 2 verir.

`%` işlecinin birkaç yaygın kullanımı vardır:

- **Tek-çift testi:** `n % 2 == 0` ise `n` çifttir. Tek sayı testi için `n % 2 != 0` yazılır. `n % 2 == 1` negatif tek sayılarda yanlış sonuç verir, çünkü `-7 % 2` −1'dir.
- **Son basamak:** `n % 10` son basamağı, `n / 10` son basamağı atılmış sayıyı verir. `n` = 1234 için 4 ve 123 bulunur.

- **Birim ayırma:** saniyeyi saat, dakika ve saniyeye ayırmak gibi işler `/` ve `%` ile yapılır.

Örnek 2.2.3

3725 saniye kaç saat, kaç dakika, kaç saniyedir? `int s = 3725;` için:

- `s / 3600` = 1. Bir saat 3600 saniyedir ve 3725'in içinde bir kez vardır.
- `s % 3600 / 60` = 125 / 60 = 2. Önce saatten artan 125 saniye bulunur, sonra bunun içindeki tam dakikalar sayılır. `%` ve `/` eşit önceliklidir ve soldan sağa uygulanır.
- `s % 60` = 5. Tam dakikalardan artan saniyelerdir.

Sonuç: 1 saat 2 dakika 5 saniye. Sağlama: 1 × 3600 + 2 × 60 + 5 = 3725.

Java'da üs işleci yoktur. `2 ** 3` derlenmez, `2 ^ 3` ise derlenir ama üs almaz: `^` bitlerin özel veyasıdır ve 1 verir. Üs için `Math.pow(2, 3)` kullanılır; sonuç `double` tipinde 8.0'dır. `+` işlecinin de ikinci bir anlamı vardır. İşlenenlerinden biri `String` ise **birleştirme** (*concatenation*) yapar: `"Yaş: " + 20` ifadesi `Yaş: 20` metnini verir.

Tam sayı bölmesinde bölen 0 ise program `ArithmeticException: / by zero` ile durur. Ondalıklı bölmede ise istisna oluşmaz: `5.0 / 0` `Infinity`, `0.0 / 0` `NaN` (sayı değil) verir. Bu fark da işlenenlerin tipinden gelir.

2.2.5.2 Artırma ve azaltma

Tanım 2.2.20

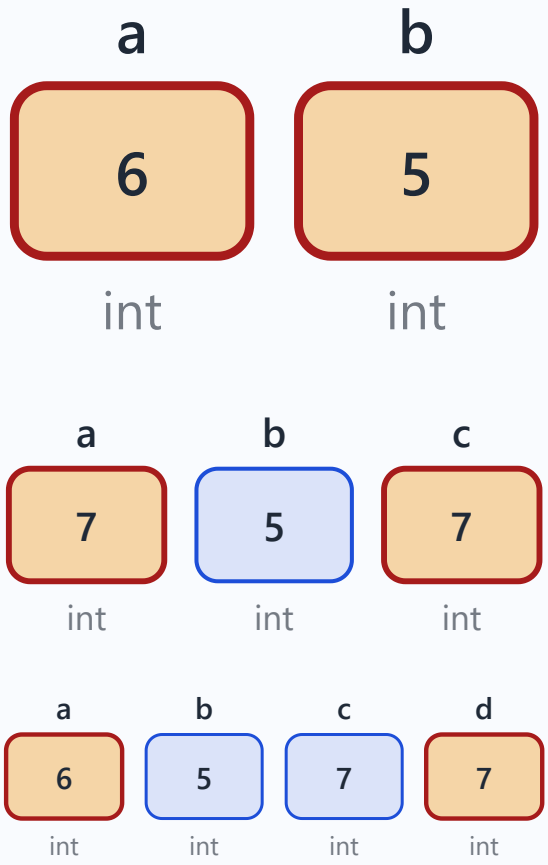
Artırma (`++`) ve **azaltma** (`--`) işleçleri bir değişkeni 1 artırır ya da azaltır (*increment / decrement*). İşleç değişkenden sonra yazılırsa **sonek** (`x++`), önce yazılırsa **önek** (`++x`) biçimidir.

`x++` ve `++x` ikisi de `x`'i 1 artırır. Fark, ifadenin **değerindedir**. Sonek biçim eski değeri, önek biçim yeni değeri verir. `i++;` tek başına bir deyim olarak yazıldığında ifadenin değeri kullanılmaz ve iki biçim arasında fark yoktur. Fark yalnız ifade başka bir işlemin parçası olunca ortaya çıkar.

```
int a = 5;
int b = a++;
System.out.println(a + " " + b);
int c = ++a;
System.out.println(a + " " + c);
int d = a--;
System.out.println(a + " " + d);
```

Satır	Ne olur?	`a`	Yeni değişken	Çıktı
`int a = 5;`	—	5		
`int b = a++;`	önce eski değer (5) kullanılır, sonra `a` artar	6	`b` = 5	
`println(a + " " + b)`		6		`6 5`
`int c = ++a;`	önce `a` artar, sonra yeni değer (7) kullanılır	7	`c` = 7	
`println(a + " " + c)`		7		`7 7`
`int d = a--;`	önce eski değer (7), sonra `a` azalır	6	`d` = 7	
`println(a + " " + d)`		6		`6 7`

Tablo 2.2.19: Artırma ve azaltma izlemesi.



Şekil 2.2.23: (a) `b = a++` sonrası: `a` = 6, `b` = 5. (b) `c = ++a` sonrası: `a` = 7, `c` = 7. (c) `d = a--` sonrası: `a` = 6, `d` = 7.

Sık yapılan hata

`i = i++;` yazmak. Bu satır `i` 'yi artırmaz. Sağ taraf önce hesaplanır: `i++` eski değeri, örneğin 5'i verir ve `i` 'yi 6 yapar. Ardından atama gelir ve `i` 'ye o eski değer, 5 yazılır. Artış böylece silinir. Bir değişkeni artırmak için `i++;` ya da `i += 1;` yeterlidir. Genel kural da buradan

çıkar: aynı değişkeni bir ifadede hem değiştirip hem okumak, kodu okunmaz hâle getirir.

2.2.5.3 İşleç önceliği

Tanım 2.2.21

İşleç önceliği (*operator precedence*), bir ifadede birden çok işleç bulunduğunda hangisinin önce uygulanacağını belirleyen sıradır. Eşit öncelikli işleçlerde **birleşme yönü** (*associativity*) karar verir: soldan sağa ya da sağdan sola.

Öncelik	İşleçler	Yön
1 (en yüksek)	sonek: `x++` `x--`	→
2	tekli: `+x` `-x` `!` `++x` `--x` · tip dönüşümü `(int)`	←
3	çarpma, bölme, mod: `*` `/` `%`	→
4	toplama, çıkarma: `+` `-`	→
5	karşılaştırma: `<` `>` `<=` `>=`	→
6	eşitlik: `==` `!=`	→
7	mantıksal VE: `&&`	→
8	mantıksal VEYA: `||`	→
9	üçlü: `? :`	←
10 (en düşük)	atama: `=` `+=` `-=` `*=` …	←

Tablo 2.2.20: İşleçlerin öncelik sırası. → soldan sağa, ← sağdan sola gruplanır.

Tablo matematikteki işlem sırasını genişletir. Çarpma ve bölme toplamadan önce gelir; karşılaştırmalar aritmetikten sonra, mantıksal bağlaçlar karşılaştırmalardan sonra, atama en sonda yapılır. Bu sıra günlük yazımla uyumludur: `x + 1 < y * 2 && z == 0` ifadesi parantezsiz de beklendiği gibi `((x + 1) < (y * 2)) && (z == 0)` diye okunur.

Eşit öncelikli ikili işleçlerde soldaki önce uygulanır: `10 - 4 - 3` = (10 − 4) − 3 = 3'tür, 10 − (4 − 3) = 9 değil. Tekli işleçlerde ve atamada ise sağdaki önce uygulanır: `a = b = 0` önce `b`'yi, sonra `a`'yı sıfırlar. Sırayı değiştirmek ya da okunurluğu artırmak için parantez kullanılır. Gereksiz görünen bir parantez de çoğu zaman yararlıdır; okuyanın öncelik tablosunu hatırlamasına gerek bırakmaz.

Önceliğin nasıl uygulandığını bir ifade üzerinde adım adım görelim: `System.out.println(2 + 3 * 4 - 6 / 2);`

1. `2 + 3 * 4 - 6 / 2`: `*` ve `/`, `+` ve `-`'den önce gelir. İkisinden soldaki önce: 3 * 4 = 12.
2. `2 + 12 - 6 / 2`: sırada kalan yüksek öncelikli işlem 6 / 2 = 3.
3. `2 + 12 - 3`: kalan `+` ve `-` eşit önceliklidir ve soldan sağa yapılır: 2 + 12 = 14.

4. `14 - 3` = 11.

Sonuç `11`'dir. Parantez sırayı değiştirir: `(2 + 3) * 4 - 6 / 2` = 5 × 4 − 3 = 20 − 3 = 17. Birkaç kısa örnek daha: `20 / 4 * 2` soldan hesaplanır ve 5 × 2 = 10 verir, 20 / 8 değil. `-2 * -3` ifadesinde tekli eksiler önce uygulanır ve sonuç 6'dır.

Dizgi birleştirme de aynı soldan sağa kuralına uyar ve bu yüzden şaşırtıcı sonuçlar verebilir. `1 + 2 + "3" + 4 + 5` ifadesinde önce 1 + 2 = 3 hesaplanır; henüz metin yoktur. Sonra 3 + `"3"` metin olur: `"33"`. Bundan sonraki her `+` birleştirmedir: `"334"`, `"3345"`. Sonuç `3345`'tir.

2.2.5.4 Karşılaştırma işleçleri

Tanım 2.2.22

Karşılaştırma işleçleri (*comparison*, ilişkisel işleçler) iki değer arasındaki ilişkiyi değerlendirir. Sonuç her zaman bir `boolean`'dır: `true` ya da `false`.

İşleç	Anlamı	Örnek	Sonuç
`==`	eşit mi?	`x == y`	`false`
`!=`	eşit değil mi?	`x != y`	`true`
`>`	soldaki büyük mü?	`y > x`	`true`
`<`	soldaki küçük mü?	`x < y`	`true`
`>=`	büyük veya eşit mi?	`x >= 30`	`false`
`<=`	küçük veya eşit mi?	`x <= y`	`true`

Tablo 2.2.21: Karşılaştırma işleçleri (`x` = 10, `y` = 20).

Karşılaştırmanın sonucu bir değişkende saklanabilir: `boolean buyukMu = y > x;`. Bu satırdan sonra `buyukMu` `true` olur. İki karakterli işleçlerin yazımı sabittir: `>=` doğrudur, `=>` yazılamaz. İşlecin içine boşluk da konamaz: `> =` derlenmez.

Matematikteki *0 < x < 10* yazımı Java'da çalışmaz. `0 < x < 10` ifadesinde önce `0 < x` hesaplanır ve bir `boolean` çıkar. Ardından bu `boolean` 10 ile karşılaştırılmaya çalışılır ve derleyici hata verir. Doğru yazım iki karşılaştırmayı mantıksal VE ile bağlar: `0 < x && x < 10`.

Sık yapılan hata

`=` ile `==`'yi karıştırmak. `=` atamadır, `==` karşılaştırmadır. `if (x = 5)` yazılınca koşulun yerine bir atama girer. Atamanın değeri 5'tir, yani bir `int`. Java koşul olarak yalnız `boolean` kabul ettiği için kod derlenmez:

```
Esit.java:4: error: incompatible types: int cannot be converted to boolean
        if (x = 5) { }
              ^
1 error
```

C dilinde aynı satır derlenir ve koşul her zaman doğru sayılır; hata gizli kalır. Java'nın katı tip denetimi bu hatayı derleme sırasında yakalar. Bir istisna vardır: `boolean bitti; … if (bitti = true)` derlenir, çünkü atamanın değeri bir `boolean` 'dır. `boolean` değişkenlerde en güvenli yazım karşılaştırmayı hiç kullanmamaktır: `if (bitti)` ve `if (!bitti)`.

2.2.5.5 Mantıksal işleçler

Tanım 2.2.23

Mantıksal işleçler (*logical operators*) `boolean` değerleri birleştirir. VE (*AND*) `&&` : iki koşul da `true` ise `true` . VEYA (*OR*) `||` : en az biri `true` ise `true` . DEĞİL (*NOT*) `!` : değeri tersine çevirir.

`A`	`B`	`A && B`	`A || B`	`!A`
`true`	`true`	`true`	`true`	`false`
`true`	`false`	`false`	`true`	`false`
`false`	`true`	`false`	`true`	`true`
`false`	`false`	`false`	`false`	`true`

Tablo 2.2.22: Mantıksal işleçlerin doğruluk tablosu.

Tablo kısaca şöyle okunur: `A && B` yalnız ikisi de doğruyken doğrudur; `A || B` yalnız ikisi de yanlışken yanlıştır. Günlük dildeki “veya” bazen “ikisinden yalnız biri” anlamına gelir (“çay veya kahve”). Java'daki `||` ise ikisi de doğru olduğunda da doğru verir.

```
int x = 10, y = 20, z = 30;
System.out.println(x < y && y < z);
System.out.println(x > y || y < z);
System.out.println(!(x < y));
int yas = 20;
boolean ehliyet = true;
System.out.println(yas >= 18 && ehliyet);
```

```
true
true
false
true
```

`x < y && y < z` ifadesinde iki karşılaştırma da doğrudur (10 < 20 ve 20 < 30), sonuç `true`'dur. `x > y || y < z` ifadesinde ilk karşılaştırma yanlış, ikincisi doğrudur; VEYA için biri yeterlidir. `!(x < y)` doğru olan `x < y`'yi tersine çevirir ve `false` verir. Son satır bir kuralı koda döker: 18 yaşında ya da daha büyük olan ve ehliyeti bulunan kişi araç kullanabilir. İki koşul da sağlandığı için sonuç `true`'dur.

`!` işlecinin önceliği yüksektir. `!x < y` yazılırsa `!` yalnız `x`'e uygulanmaya çalışılır ve `int` bir değer tersine çevrilemeyeceği için kod derlenmez. Bu yüzden karşılaştırmanın tersi parantezle yazılır: `!(x < y)`. Aynı ifade daha sade biçimde `x >= y` diye de yazılabilir.

2.2.5.6 Kısa devre değerlendirme

Tanım 2.2.24

Kısa devre değerlendirme (*short-circuit evaluation*), `&&` ve `||` işleçlerinde sonuç sol taraftan belli olunca sağ tarafın hiç hesaplanmamasıdır.

`&&`'de sol taraf `false` ise sonuç zaten `false`'tur; sağ taraf ne olursa olsun bunu değiştiremez. Java bu durumda sağ tarafı **hiç hesaplamaz**. `||`'de sol taraf `true` ise sonuç zaten `true`'dur ve sağ taraf yine atlanır. `gun == 6 || gun == 7` ifadesinde `gun` 6 ise ikinci karşılaştırma hiç yapılmaz. Bu ifade "hafta sonu" anlamına gelir. Tersi `!(gun == 6 || gun == 7)` ise "hafta içi" demektir.

Kısa devre yalnız hız kazandırmaz; tehlikeli bir işlemi korumak için de kullanılır. Önce "bölen sıfır değil mi?" diye bakılır, ancak öyleyse bölme yapılır:

```java
int bolen = 0;
int sayi = 10;
if (bolen != 0 && sayi / bolen > 1) {
    System.out.println("Oran 1'den büyük");
} else {
    System.out.println("Bölünemez ya da küçük");
}
```

```
Bölünemez ya da küçük
```

`bolen` 0'dır. `bolen != 0` yanlış çıkar ve `&&` sağ tarafa hiç geçmez. Böylece `sayi / bolen` bölmesi yapılmaz ve program `else` bloğuna geçer. Sıra önemlidir. Koşul `sayi / bolen > 1 && bolen != 0` diye yazılsaydı bölme önce yapılırdı. Program sıfıra bölme hatasıyla, `java.lang.ArithmeticException: / by zero` iletisiyle dururdu. Koruyucu koşul her zaman sola yazılır.

Java'da kısa devre yapmayan iki işleç daha vardır: tek `&` ve tek `|`. Bunlar `boolean` değerlerle kullanıldığında iki tarafı da her zaman hesaplar. `a != 0 & 10 / a > 1` ifadesi `a` = 0 iken sol taraf yanlış olsa bile bölmeyi yapar ve istisna fırlatır. Koşullarda neredeyse her zaman çift karakterli `&&` ve `||` kullanılır.

2.2.5.7 Üçlü işleç

Tanım 2.2.25

Üçlü işleç (*ternary operator*, koşul işleci) bir koşula göre iki değerden birini seçen kısa bir ifadedir: koşul doğruysa `değer1`, değilse `değer2` sonuç olur. Üç işleni olan tek Java işlecidir.

```
sonuç = (koşul) ? değer1 : değer2;
```

Bir sayının pozitif ya da negatif olduğunu belirlemek için:

```
int sayi = -5;
String sonuc = (sayi > 0) ? "Pozitif" : "Negatif";
System.out.println("Sayı " + sonuc);
```

```
Sayı Negatif
```

`sayi` −5 olduğu için `sayi > 0` yanlıştır ve iki nokta üst üsteden sonraki değer, `"Negatif"` seçilir. Aynı işi yapan `if-else` şöyledir:

```
String sonuc;
if (sayi > 0) {
    sonuc = "Pozitif";
} else {
    sonuc = "Negatif";
}
```

İki yazım arasındaki temel fark şudur: `if-else` bir **deyimdir** ve bir iş yapar, `?:` ise bir **ifadedir** ve bir değer üretir. Bu yüzden `?:` atamanın sağında ya da bir `println`'ın içinde kullanılır. Tek başına bir deyim olarak yazılamaz:

```
Tern.java:4: error: not a statement
        x > 0 ? System.out.println("a") : System.out.println("b");
              ^
1 error
```

İki seçeneğin tipi uyumlu olmalıdır. Birkaç örnek: `double mutlak = (sayi < 0) ? -sayi : sayi;` mutlak değeri verir ve −4.5 için 4.5 bulunur. `int buyuk = (a > b) ? a : b;` iki sayının büyüğünü seçer; 8 ve 13 için 13 bulunur. Seçenekler farklı sayısal tiplerdeyse sonuç geniş olana yükseltilir: `true ? 1 : 2.0` ifadesinin değeri 1 değil, `1.0`'dır.

Üçlü işleçler iç içe kullanılabilir. Bir sayının sıfır mı, pozitif mi, negatif mi olduğu tek satırda bulunur:

```
int sayi = -5;
String sonuc = (sayi == 0) ? "Sıfır" : (sayi > 0) ? "Pozitif" : "Negatif";
System.out.println("Sayı " + sonuc);
```

`sayi`	`sayi == 0`	`sayi > 0`	**Sonuç**
0	`true`	hesaplanmaz	Sıfır
7	`false`	`true`	Pozitif
−5	`false`	`false`	Negatif

Tablo 2.2.23: İç içe üçlü işlecin sonuçları.

İfade şöyle okunur: “sıfır mı? evet → Sıfır; hayır → pozitif mi? evet → Pozitif; hayır → Negatif.” İkinci `?:`, birincinin `değer2`'sidir; üçlü işleç sağdan gruplanır. İkiden fazla iç içe `?:` okunurluğu bozar. Bu durumda `if-else if` zinciri tercih edilir.

2.2.6 Kontrol deyimlerinin söz dizimi

Kontrol deyimleri programın akışını yönetir. Bir koşula göre dallanmayı ya da bir işin tekrarlanmasını sağlarlar. Ayrıntılı kullanımları kendi başına bir konudur. Burada her birinin yazım kuralı ve en sık rastlanan söz dizimi tuzakları verilir.

Deyim	Söz dizimi	Kural
`if` / `else`	`if (koşul) { … } else { … }`	koşul `boolean` olmalı; `if (…);` sonundaki `;` boş deyimdir
`switch`	`switch (d) { case 1: … break; default: … }`	`int`, `char`, `String`, `enum`; `break` yoksa alttaki durumlara düşer

Deyim	Söz dizimi	Kural
`switch` (Java 14 ve sonrası)	`x = switch (d) { case 1 -> "P"; default -> "?"; };`	aşağı düşme yok; değer üreten ifade
`for`	`for (int i = 0; i < n; i++) { … }`	başlangıç bir kez; koşul turdan önce, güncelleme sonra
`while`	`while (koşul) { … }`	gövde hiç çalışmayabilir
`do-while`	`do { … } while (koşul);`	gövde en az bir kez çalışır; sondaki `;` zorunlu
`break` / `continue`	`break;` · `continue;`	`break` çıkar, `continue` turun kalanını atlar

Tablo 2.2.24: Kontrol deyimlerinin söz dizimi.

`if` **deyiminde** koşul her zaman parantez içinde yazılır ve `boolean` olmalıdır. Gövde tek bir deyimse süslü parantez yazılmayabilir, ama bu alışkanlık hataya açıktır. Parantezsiz bir `if`'e yalnız ilk deyim bağlıdır; altına girintili yazılan ikinci satır koşuldan bağımsız çalışır. Derleyici girintiye bakmaz. Bu yüzden gövde tek satır olsa bile süslü parantez kullanmak güvenlidir.

`switch` **deyiminde** denetlenen değer `int` (ve `byte`, `short`), `char`, `String` ya da `enum` olabilir. `long`, `double` ve `boolean` kullanılamaz; derleyici `selector type double is not allowed` gibi bir ileti verir. Klasik biçimde her `case`'in sonuna `break` yazılır. `break` unutulursa eşleşen durumdan sonraki durumlar da sırayla çalışır. Bu davranışa **aşağı düşme** (*fall-through*) denir. Aşağı düşme bazen bilinçli kullanılır: `case "Cmt": case "Paz":` yazımı iki durumu tek bir koda bağlar. Java 14 ile gelen ok (`->`) biçiminde ise aşağı düşme yoktur ve `switch` bir değer üretebilir. İki biçim aynı `switch` içinde karıştırılamaz (`different case kinds used in the switch`).

`for` **döngüsünün** başlığı noktalı virgülle ayrılmış üç bölümden oluşur. Başlangıç bölümü yalnız bir kez, döngüye girerken çalışır. Koşul her turdan önce denetlenir; yanlışsa döngü biter. Güncelleme her turun sonunda çalışır. Her `for` döngüsü bir `while` döngüsüyle de yazılabilir: başlangıç döngüden önceye, güncelleme gövdenin sonuna taşınır.

`while` **ile** `do-while` arasındaki fark koşulun yeridir. `while` koşulu önce denetler; koşul baştan yanlışsa gövde hiç çalışmaz. `do-while` ise gövdeyi önce çalıştırır, koşulu sonra denetler; gövde en az bir kez çalışır. `do-while`'ın sonundaki noktalı virgül zorunludur ve unutulursa `';' expected` hatası alınır.

`break` içinde bulunduğu en yakın döngüden ya da `switch`'ten hemen çıkar. `continue` ise yalnız o turun kalanını atlar ve bir sonraki tura geçer. `for` döngüsünde `continue`'dan sonra güncelleme bölümü yine çalışır. `switch` bir döngünün içindeyse, `switch` içindeki `break` yalnız `switch`'ten çıkar; döngü sürer.

Örnek 2.2.4

1'den 100'e kadar olan sayılardan 3'ün katı olmayanların toplamı `continue` ile şöyle bulunur:

```java
int toplam = 0;
for (int i = 1; i <= 100; i++) {
    if (i % 3 == 0) continue;   // 3'ün katlarını atla
    toplam += i;
}
```

3'ün katı olan bir `i` için `continue` çalışır, `toplam += i` atlanır ve döngü `i++` ile sürer. Sağlama: 1'den 100'e kadar sayıların toplamı 5050'dir. 3'ün katları 3, 6, …, 99'dur; 33 tanedir ve toplamları 3 × (1 + 2 + … + 33) = 3 × 561 = 1683'tür. Sonuç 5050 − 1683 = **3367**'dir.

Sık yapılan hata

Sonsuz döngü. Koşulu yanlışa götüren bir adımı olmayan döngü hiç bitmez. `for (int i = 1; i <= 10; i--)` döngüsünde `i` azaldıkça koşul hep doğru kalır. `int i = 1; while (i <= 3) { System.out.println(i); }` döngüsünde ise `i` gövdede hiç güncellenmez ve ekrana durmadan 1 yazılır. Her döngü yazılırken "koşul hangi adımda yanlış olacak?" sorusu sorulmalıdır.

2.2.7 Sık yapılan hatalar

Söz dizimi hataları iki gruba ayrılır. Birinci gruptakileri derleyici yakalar; program hiç çalışmaz ve ileti hatanın yerini gösterir. İkinci gruptakiler derlenir ama yanlış çalışır. Bunlar daha tehlikelidir, çünkü ancak çıktı dikkatle incelenince fark edilir.

Hata	Örnek	İleti	Doğrusu
Eksik noktalı virgül	`System.out.println("a")`	`';' expected`	satır sonuna `;` ; hata çoğu zaman bir önceki satırın sonundadır
Dosya adı ≠ sınıf adı	`merhabadunya.java` içinde `public class MerhabaDunya`	`class MerhabaDunya is public, should be declared in a file named MerhabaDunya.java`	dosyayı sınıf adıyla kaydetmek
Büyük-küçük harf	`string`, `Int`, `system.out`	`cannot find symbol`; `package system does not exist`	`String`, `int`, `System`
Değersiz yerel değişken	`int sayac; println(sayac);`	`variable sayac might not have been initialized`	okumadan önce değer vermek
Soneksiz sabit	`float f = 3.14;` · `long l = 3000000000;`	`possible lossy conversion from double to float` · `integer number too large`	`3.14f` · `3000000000L`
Daraltmada dönüşümü unutmak	`char k = i;` · `int n = d;`	`incompatible types: possible lossy conversion`	`(char) i` · `(int) d`
`=` ile `==`	`if (x = 5)`	`int cannot be converted to boolean`	`if (x == 5)`

Tablo 2.2.25: Derleyicinin yakaladığı hatalar ve iletileri.

Hata	Örnek	Sonuç	Doğrusu
Tam sayı bölmesi	`double ort = (3 + 4) / 2;`	3.0	`(3 + 4) / 2.0` → 3.5
Geç kalınmış dönüşüm	`(double) (a / b)`	kesir önceden kaybolmuş	`(double) a / b`
Taşma	`long l = Integer.MAX_VALUE + 1;`	−2147483648	`Integer.MAX_VALUE + 1L`
`switch`'te `break` yok	`case 1: … case 2: …`	sonraki durumlar da çalışır	her `case` sonuna `break` ya da `->` biçimi
`if`'ten sonra `;`	`if (x > 0); { … }`	boş deyim koşula bağlanır, blok her zaman çalışır	`;`'ü silmek
Sonsuz döngü	`for (int i = 1; i <= 10; i--)`	döngü bitmez	koşulu yanlışa götüren güncelleme
Ondalıkta `==`	`0.1 + 0.2 == 0.3`	`false`	`Math.abs(a - b) < 1e-9`
Birleştirme sırası	`"Toplam: " + a + b`	sayılar yan yana yazılır	`"Toplam: " + (a + b)`

Tablo 2.2.26: Derlenen ama yanlış çalışan kodlar.

`if (x > 0); { … }` hatası özellikle sinsidir. `if`'in hemen arkasındaki noktalı virgül, hiçbir şey yapmayan bir **boş deyimdir**. Koşula bağlanan budur. Alttaki süslü parantezli blok ise koşuldan bağımsız, sıradan bir bloktur ve her zaman çalışır. `x` = −3 iken bile blok çalışır. Derleyici bunu hata saymaz, çünkü iki satır da ayrı ayrı geçerlidir.

2.2.8 Özet

- **Derleme:** `javac Ad.java` kaynak kodu denetler ve `Ad.class` dosyasına bytecode olarak yazar. `java Ad` JVM'i başlatır, sınıfı yükler ve `main`'i çağırır. Aynı bytecode JVM bulunan her platformda çalışır.
- **JVM** bir yığın makinesidir: komutlar değerleri işlenen yığınına iter ve oradan alır. Yorumlayıcı bytecode'u çalıştırır, JIT derleyicisi sık çalışan kodu makine koduna çevirir.
- **Söz dizimi:** deyimler `;` ile biter, bloklar `{ }` ile açılıp kapanır, Java büyük-küçük harfe duyarlıdır. Açıklamalar `//` ve `/* */` ile yazılır.
- **İlkel tipler:** `byte` 1, `short` 2, `int` 4, `long` 8 bayt; `float` 4, `double` 8 bayt; `char` 2 bayt; `boolean`. Sınırı aşan tam sayı hata vermeden taşar ve aralığın öbür ucuna sarar.
- **Ondalıklı sayılar** ikili tabanda yaklaşık saklanır: `0.1 + 0.2` 0,3 değildir. Ondalıklı sayılar `==` ile karşılaştırılmaz.
- **Adlar** harf, rakam, `_` ve `$` içerir; rakamla başlamaz; anahtar sözcük olmaz. Sınıflar PascalCase, değişkenler camelCase, sabitler BÜYÜK_HARF yazılır.
- **Atama:** sağ taraf önce hesaplanır. `+=` gibi bileşik atamalar dönüşümü kendileri yapar. Yerel değişkene okunmadan önce değer verilmelidir. `final` değişken bir kez değer alır.

- **Dönüşüm:** `byte → short → int → long → float → double` yönü otomatiktir. Ters yön `(tip)` ile yapılır; kesir atılır, sığmayan değer taşar. `int / int` tam sayı bölmesidir ve dönüşüm bölmeden önce yapılmalıdır.
- **İşleçler:** öncelik sırası tekli > `* / %` > `+ -` > karşılaştırma > `&&` > `||` > `?:` > atama. `&&` ve `||` kısa devre yapar. `?:` bir değer üreten ifadedir.
- **Akış:** `if/else`, `switch` (+ `break`), `for`, `while` (0 ya da daha çok tur), `do-while` (en az 1 tur). `break` çıkar, `continue` turun kalanını atlar.

2.2.9 Kavram listesi

kaynak kod *source code* · açıklama *comment* · blok *block* · JDK (geliştirme kiti) *Java Development Kit* · bytecode (ara kod) *bytecode* · Java Sanal Makinesi (JVM) *Java Virtual Machine* · JIT (tam zamanında) derleyici *JIT compiler* · platform bağımsızlığı *platform independence* · değişken *variable* · bildirim *declaration* · ilk değer verme *initialization* · veri tipi *data type* · ilkel tip *primitive type* · referans tipi *reference type* · tam sayı *integer* · karakter (char) *character* · boolean (mantıksal tip) *boolean* · taşma *overflow* · tanımlayıcı *identifier* · anahtar sözcük *keyword* · adlandırma kuralı *naming convention* · atama *assignment* · bileşik atama *compound assignment* · varsayılan değer *default value* · sabit değer *literal* · isimlendirilmiş sabit *named constant* · tip dönüşümü *type conversion* · açık dönüşüm (cast) *cast* · genişletme / daraltma dönüşümü *widening / narrowing conversion* · tam sayı bölmesi *integer division* · ifade *expression* · işleç *operator* · artırma / azaltma *increment / decrement* · işleç önceliği *operator precedence* · karşılaştırma işleci *comparison operator* · mantıksal işleç *logical operator* · kısa devre değerlendirme *short-circuit evaluation* · koşul işleci *conditional (ternary) operator*

2.2.10 Kendini Dene

1. `Deneme.java` adlı bir dosyada `public class Test` tanımlı. Dosya `javac Deneme.java` ile derlenmek isteniyor. Ne olur? Program nasıl çalıştırılır hâle getirilir?

> **Cevap**
>
> Derleme başarısız olur: `class Test is public, should be declared in a file named Test.java`. Public bir sınıfın adı dosya adıyla birebir aynı olmalıdır. Dosya `Test.java` adıyla kaydedilir, `javac Test.java` ile derlenir ve `java Test` ile (uzantısız) çalıştırılır.

2. `int a = 5, b = 7; int c = a * b;` satırları derlendiğinde `a * b` hesaplanırken JVM'in işlenen yığınında hangi değerler bulunur? `imul` komutundan hemen sonra yığında ne kalır?

> **Cevap**
>
> Önce `iload_1` 5'i, sonra `iload_2` 7'yi yığına iter; yığında alttan üste 5, 7 durur. `imul` bu iki değeri çıkarır, çarpar ve sonucu iter: yığında yalnız 35 kalır. Ardından `istore_3` 35'i `c`'ye yazar ve yığın boşalır. Toplamadaki `iadd` ile aynı düzen geçerlidir.

3. Aşağıdaki satır neden yanlış sonuç verir? Doğru değer nasıl elde edilir?
`long mikrosaniye = 24 * 60 * 60 * 1000 * 1000;`

> **Cevap**
>
> Bütün sabitler `int`'tir ve çarpma `int` tipinde yapılır. Gerçek sonuç 86.400.000.000'dir ve `int` sınırını aşar; taşan değer `500654080` olur. `long` değişkene ancak bu taşmış değer atanır. İlk sabite `L` eklenince çarpma baştan `long` yürür: `24L * 60 * 60 * 1000 * 1000` = `86400000000`.

4. `byte b = 120; b += 10;` satırlarından sonra `b` kaçtır? Aynı iş `b = b + 10;` diye yazılsa ne olur?

> **Cevap**
>
> `b` = −126 olur. 130, `byte` sınırı 127'yi aşar; bileşik atama sonucu kendisi `byte`'a çevirir ve 130 − 256 = −126 kalır. `b = b + 10;` ise derlenmez: `b + 10` bir `int` ifadesidir ve `byte`'a ancak `(byte)` ile atanabilir (`possible lossy conversion from int to byte`).

5. Şu adlardan hangileri geçerli tanımlayıcıdır? `_2x`, `x-2`, `int`, `Integer`, `toplam$`, `9lu`

> **Cevap**
>
> Geçerli: `_2x` (alt çizgiyle başlar), `Integer` (bir sınıf adıdır, anahtar sözcük değildir; ama ad olarak kullanılması okuyanı şaşırtır), `toplam$` (`$` izinlidir). Geçersiz: `x-2` (`-` çıkarma işlecidir), `int` (anahtar sözcük), `9lu` (rakamla başlar).

6. `char c = '5';` iken `System.out.println(c + 1);` ve `System.out.println(c - '0' + 1);` ne yazdırır?

> **Cevap**
>
> Birinci satır 54 yazar: `'5'`'in kodu 53'tür ve `c + 1` bir `int` toplamıdır. İkinci satır 6 yazar: `c - '0'` = 53 − 48 = 5, yani rakamın sayı değeridir; buna 1 eklenir.

7. `int x = 7; int y = x-- - --x;` satırlarından sonra `x` ve `y` kaçtır?

   Cevap

   İşlenenler soldan sağa hesaplanır. `x--` eski değeri, 7'yi verir ve `x` 6 olur. `--x` önce `x`'i 5 yapar ve 5'i verir. `y` = 7 − 5 = 2, `x` = 5 olur.

8. `int a = 0;` iken `if (a != 0 & 10 / a > 1)` ile `if (a != 0 && 10 / a > 1)` koşulları arasında ne fark vardır?

   Cevap

   Tek `&` kısa devre yapmaz ve iki tarafı da hesaplar. Sol taraf yanlış olsa da `10 / a` bölmesi yapılır ve program `ArithmeticException: / by zero` ile durur. `&&` ise sol taraf yanlış olunca sağ tarafa hiç bakmaz; koşul yanlış sayılır ve program güvenle sürer.

9. `int k = 3;` ile başlayan iki döngünün sonunda `k` kaçtır?
   `do { k -= 4; } while (k > 5);`
   `while (k > 5) { k -= 4; }` (yine `k` = 3 ile başlayarak)

   Cevap

   `do-while` gövdeyi bir kez çalıştırır: `k` = −1 olur, `-1 > 5` yanlıştır ve döngü biter; sonuç −1. `while` koşulu önce denetler: `3 > 5` yanlış olduğu için gövde hiç çalışmaz; sonuç 3.

2.2.11 İleri okuma

- Oracle, *The Java Tutorials*, "Language Basics" kesimi: değişkenleri, işleçleri, ifadeleri ve kontrol akışı deyimlerini kısa örneklerle anlatan resmî öğretici.
- J. Gosling vd., *The Java Language Specification*, Java SE 21: anahtar sözcüklerin, sabit değerlerin (Bölüm 3), tiplerin (Bölüm 4) ve dönüşüm kurallarının (Bölüm 5) kesin tanımı. Bir derleyici iletisinin nedeni merak edildiğinde başvurulacak kaynak.
- T. Lindholm vd., *The Java Virtual Machine Specification*, Java SE 21: bytecode komutlarını, işlenen yığınını ve `.class` dosyasının yapısını ayrıntılı tanımlar; `javap -c` çıktısını okumayı kolaylaştırır.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 13. baskı, Bölüm 3: ilkel tipleri, dönüşümleri ve işleçleri deneyimli bir programcının gözünden, sık yapılan hatalara dikkat çekerek işler.

2.2.12 Kaynaklar

1. Oracle, *The Java Tutorials*: “Language Basics” (değişkenler, işleçler, ifadeler, kontrol akışı), docs.oracle.com/javase/tutorial.
2. J. Gosling vd., *The Java Language Specification*, Java SE 21 Edition (Bölüm 3: söz dizimi, anahtar sözcükler, sabit değerler; Bölüm 4: tipler; Bölüm 5: dönüşümler).
3. T. Lindholm vd., *The Java Virtual Machine Specification*, Java SE 21 Edition (bytecode ve JVM).
4. P. Deitel, H. Deitel, *Java: How to Program*, 11. baskı, Pearson, 2017 (Bölüm 2–5).
5. K. Sierra, B. Bates, *Head First Java*, 3. baskı, O'Reilly, 2022.
6. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 13. baskı, 2024 (Bölüm 3).
7. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019.

KISIM I · BÖLÜM 3

Kontrol Yapıları, Fonksiyonlar, Diziler ve Dizgiler

Programın akışı karar yapıları ve döngülerle yönlendirilir. Bölümde if ve switch yapıları, iç içe döngüler, break ve continue deyimleri ile metotların tanımlanması, çağrılması, kapsam kuralları ve soyutlamadaki rolü anlatılır. Dizilerin bellekteki yerleşimi, sık görülen dizi hataları, temel dizi algoritmaları ve çok boyutlu diziler Tic Tac Toe uygulamasıyla pekiştirilir. Dizgilerde değiştirilemezlik, karşılaştırma ve biçimli çıktı incelenir; özyineleme ise faktöriyel ve seri örnekleriyle başlayıp verimlilik, kuyruk özyinelemesi ve bellekleme konularıyla tamamlanır.

Bu bölümün kodu: `github.com/sercankulcu/object-oriented-programming-java/Bolum03/src`

BU BÖLÜMÜN NOTLARI

3.1 Kontrol Yapıları

3.2 Fonksiyonlar

3.3 Diziler

3.4 Dizgiler

3.5 Özyineleme

KISIM I · BÖLÜM 3.1

Kontrol Yapıları

Bir program yalnız yukarıdan aşağıya akan satırlardan oluşsaydı her çalıştırmada aynı işi aynı sırayla yapardı. Kullanıcının girdiğine göre farklı davranamaz, bir işi bin kez yapmak için de bin satır yazmak gerekirdi. Kontrol yapıları bu iki eksiği giderir. Karar yapıları bir koşula bakarak iki ya da daha çok yoldan birini seçer. Döngüler bir deyim grubunu koşul sağlandıkça tekrarlar. Önce sıralı akış, ardından `if`, `if-else`, `else if` zinciri, iç içe kararlar ve `switch` yapısı işlenir. Sonra `for`, `while` ve `do-while` döngüleri, iç içe döngüler ve akışı döngünün ortasında değiştiren `break` ile `continue` gelir. Her yapı için kod satır satır izlenir, değişkenlerin değerleri tabloyla gösterilir ve Java derleyicisinin gerçek hata iletileri verilir. Kodların hepsi JDK 21 ile çalıştırılmış, çıktılar bu çalıştırmalardan alınmıştır.

ÖĞRENME ÇIKTILARI

Akış kontrolü
Sıralı işlem, karar ve döngü yapılarının programın akışını nasıl yönlendirdiğini açıklamak.

Karar yapıları
`if`, `if-else`, `else if` zinciri, iç içe kararlar ve `&&`, `||` ile birleşik koşullarla koşula bağlı kod yazmak; kısa devre değerlendirmenin sonucunu önceden bulmak.

switch-case
Bir değeri sabitlerle karşılaştırmak; `break`'in görevini ve aşağı düşme davranışını açıklamak; klasik `switch` ile ok biçimli `switch` ifadesini ayırt etmek.

Döngüler
`for`, `while` ve `do-while` döngülerini doğru yerde seçmek; sonsuz döngü ve sınır hatalarından kaçınmak.

İç içe döngüler
Desen, tablo ve basamak örneklerinde iç içe döngülerin kaç kez döndüğünü hesaplamak.

İzleme
Bir programı adım adım izleyerek değişken değerlerini ve ekran çıktısını önceden bulmak; `break` ve `continue`'nun etkisini göstermek.

3.1.1 Kontrol yapıları

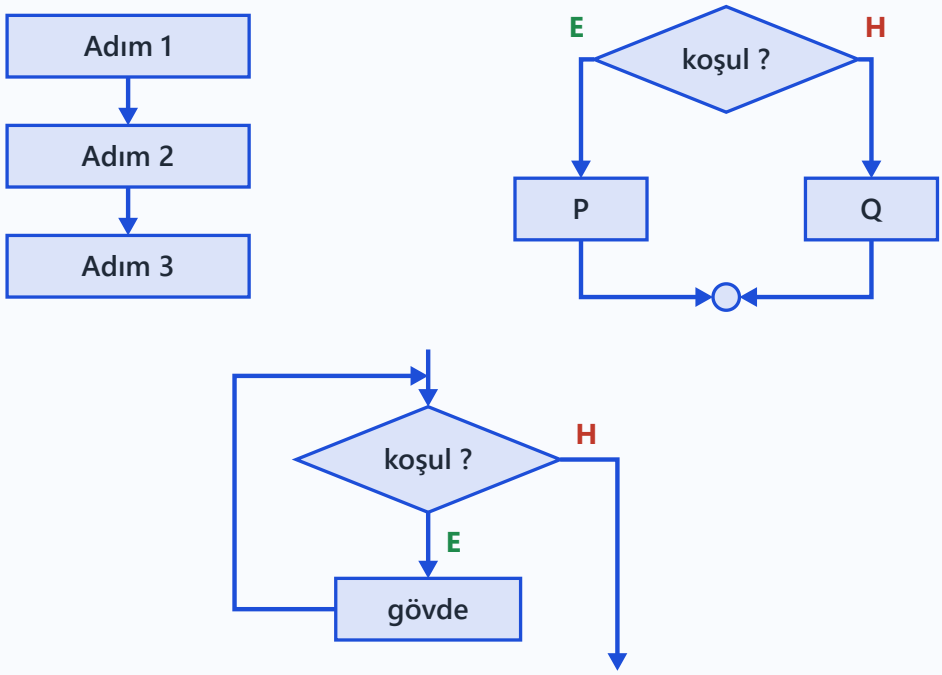
Java kaynak kodu deyimlerden oluşur. Bir değişken tanımı, bir atama ya da bir `println` çağrısı birer deyimdir ve sonları noktalı virgülle biter. Derleyici bu deyimleri yazıldıkları sırayla bayt koduna çevirir. Çalışma sırasında ise hangi deyimin ne zaman, kaç kez çalışacağını kontrol yapıları belirler.

Tanım 3.1.1

Kontrol yapısı (*control structure*), deyimlerin hangi sırayla ve kaç kez çalışacağını belirleyen dil yapısıdır. Programın bu sıraya göre izlediği yola **akış** denir.

3.1.1.1 Üç temel yapı

Java'da üç temel kontrol yapısı vardır (Şekil 3.1.1). **Sıralı işlemlerde** deyimler yukarıdan aşağıya, birer kez çalışır. **Karar yapılarında** bir koşula bakılır ve iki ya da daha çok yoldan biri seçilir; Java'daki karşılıkları `if-else` ve `switch`'tir. **Döngülerde** koşul sağlandıkça aynı deyim grubu tekrarlanır; Java'da `for`, `while` ve `do-while` döngüleri bulunur.



Şekil 3.1.1: Üç temel kontrol yapısı. Solda sıralı akış: adımlar art arda çalışır. Ortada karar: koşul doğruysa (E) P, yanlışsa (H) Q yapılır ve iki yol yeniden birleşir. Sağda döngü: koşul doğru oldukça gövde çalışır ve akış koşula geri döner; koşul yanlış olunca döngüden çıkılır.

Bu üç yapı küçük görünse de bilinen her algoritmayı yazmaya yeter. Bu sonuç, Böhm ile Jacopini'nin 1966'da gösterdiği yapısal program kuramına dayanır: sıra, seçim ve tekrar bir araya getirilerek her hesaplanabilir iş ifade edilebilir. Yapılar birbirinin içine de yerleştirilebilir. Bir döngünün gövdesinde karar, bir kararın kolunda döngü bulunabilir. Gerçek programlar bu iç içe yerleşimin sonucudur.

Kararlar programa esneklik kazandırır. Aynı program farklı girdilere farklı cevap verir: negatif sayıyı pozitife çevirir, 85 alan öğrenciye B, 45 alana F yazar. Döngüler ise tekrar eden işleri kısaltır. 1'den 1.000.000'a kadar sayıları toplamak için bir milyon satır değil, üç satırlık bir döngü yeter. Kod hem kısalır hem de değiştirmesi kolaylaşır: sınırı 1.000.000'dan 2.000.000'a çıkarmak için yalnız bir sayı değişir.

3.1.1.2 Sıralı işlemler

Kontrol yapısı içermeyen her program sıralıdır. Kod yukarıdan aşağıya satır satır çalışır ve bir satır bitmeden sonraki başlamaz. Aşağıdaki dört satır bu düzenin en küçük örneğidir:

```
int x = 5;
int y = 10;
int sonuc = x + y;
System.out.println(sonuc);
```

İlk satır `x` adlı bir `int` değişken ayırır ve içine 5 yazar. İkinci satır `y` için aynısını 10 ile yapar. Üçüncü satır önceki iki satırın sonuçlarını okur, 5 + 10 = 15'i hesaplar ve `sonuc` değişkenine yazar. Son satır bu değeri ekrana basar. Tablo 3.1.1, her satırdan sonra bellekte hangi değişkenlerin bulunduğunu gösterir. Boş hücre, değişkenin henüz tanımlanmadığı anlamına gelir.

Satır	Deyim	x	y	sonuc	Ekran
1	`int x = 5;`	5			
2	`int y = 10;`	5	10		
3	`int sonuc = x + y;`	5	10	15	
4	`System.out.println(sonuc);`	5	10	15	15

Tablo 3.1.1: Sıralı programın satır satır izlenmesi.



Şekil 3.1.2: Üçüncü satırdan sonra bellekteki üç değişken. `sonuc` kutusundaki 15, önceki iki satırın yazdığı değerlerden hesaplanmıştır.

Sıranın önemi bir deneyle görülür. Üçüncü satır en başa alınsın: `int sonuc = x + y;` satırı çalıştığında `x` ve `y` henüz tanımlı değildir. Java bu durumda programı hiç derlemez ve `cannot find symbol` iletisini verir. Sıralı yapıda her satır, kendisinden önceki satırların bıraktığı duruma dayanır.

> **Sık yapılan hata**
>
> Değişkene değer vermeden okumak. `int sonuc;` satırından hemen sonra `System.out.println(sonuc);` yazılırsa derleyici `variable sonuc might not have been initialized` hatası verir. Java, yerel bir değişkenin okunmadan önce mutlaka bir değer aldığını derleme sırasında denetler. Bu denetim, değişkene yalnız bazı kollarda değer veren karar yapılarında özellikle önem kazanır.

3.1.2 if ile karar verme

Karar verme, programın hangi yolu izleyeceğini çalışma sırasında seçmesidir. Seçim derleme sırasında yapılamaz, çünkü çoğu zaman girdiye bağlıdır: kullanıcının yazdığı sayı, okunan dosyanın içeriği, bir sensörden gelen ölçüm. Gündelik hayattaki karşılığı yol ayrımındaki bir tabeladır: “Hava yağmurluysa şemsiye al, değilse alma.” Java'da kararlar `if-else` ve `switch` ile yazılır. Bu kesim `if` ailesini, sonraki kesimler çok yollu kararları ve `switch`'i işler.

3.1.2.1 Koşul ve karşılaştırma işleçleri

> **Tanım 3.1.2**
>
> **Koşul** (*condition*), sonucu `true` ya da `false` olan mantıksal bir ifadedir (*boolean expression*). Java'da koşulun tipi her zaman `boolean`'dır.

Koşulların çoğu iki değerin karşılaştırılmasıyla kurulur. Java'nın altı karşılaştırma işleci Tablo 3.1.2'de, `x = 7` için örnekleriyle verilmiştir.

İşleç	Anlamı	Örnek	Sonuç
`==`	eşit mi	`x == 7`	`true`
`!=`	eşit değil mi	`x != 7`	`false`
`<`	küçük mü	`x < 10`	`true`
`>`	büyük mü	`x > 10`	`false`
`<=`	küçük ya da eşit mi	`x <= 7`	`true`
`>=`	büyük ya da eşit mi	`x >= 8`	`false`

Tablo 3.1.2: Karşılaştırma işleçleri (`x = 7`).

İki karakterli işleçlerin arasına boşluk girmez: `< =` yazılırsa derleyici bunu iki ayrı simge olarak okur ve hata verir. Matematikteki ≤, ≥ ve ≠ işaretlerinin Java'da karşılığı yoktur; yerlerine `<=`, `>=` ve `!=` kullanılır.

En önemli ayrım `=` ile `==` arasındadır. Tek eşittir **atamadır**: sağdaki değeri soldaki değişkene yazar. Çift eşittir **karşılaştırmadır**: iki değerin eşit olup olmadığını sorar ve bir `boolean` üretir. C dilinde `if (x = 7)` derlenir ve her zaman doğru sayılır; bu yüzden C'de çok sinsi bir hata kaynağıdır. Java ise koşulun `boolean` olmasını şart koşar. `x = 7` ifadesinin değeri `int` tipinde 7 olduğu için program derlenmez:

```
error: incompatible types: int cannot be converted to boolean
        if (x = 7) {
              ^
```

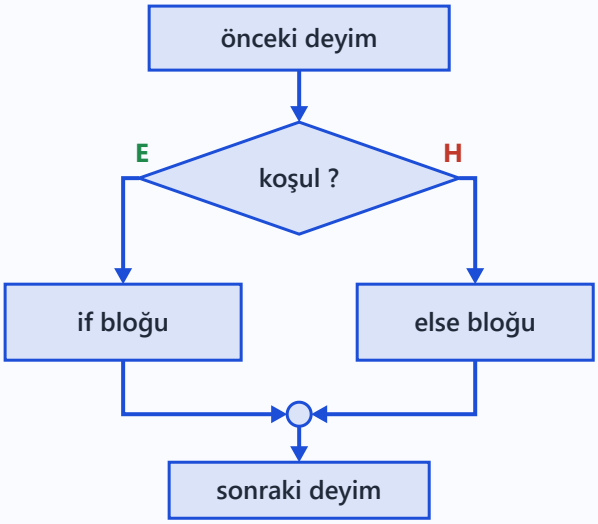
Bu kural yalnız `int` gibi sayısal değişkenleri korur. Değişkenin kendisi `boolean` ise `if (bitti = true)` derlenir, `bitti`'ye `true` yazar ve koşul hep doğru çıkar. Bu tuzaktan kaçmanın en kolay yolu `boolean` bir değişkeni hiçbir şeyle karşılaştırmamaktır: `if (bitti)` ve `if (!bitti)` yeterlidir.

3.1.2.2 if-else yapısı

> **Tanım 3.1.3**
>
> **if deyimi** (*if statement*), koşul doğruysa bir bloğu çalıştıran karar deyimidir. İsteğe bağlı `else` kolu, koşul yanlış olduğunda çalışacak bloğu verir.

```
if (koşul) {
    // Koşul doğruysa burası çalışır
} else {
    // Koşul yanlışsa burası çalışır
}
```

Yapının işleyişi üç adımdır. Önce parantez içindeki koşul değerlendirilir. Koşul `true` ise `if` bloğu çalışır ve `else` bloğu atlanır. Koşul `false` ise `if` bloğu atlanır ve `else` bloğu çalışır. Her iki durumda da akış yapının sonunda birleşir ve sonraki deyimle devam eder (Şekil 3.1.3). İki bloktan en fazla biri çalışır; `else` varsa tam olarak biri çalışır.



Şekil 3.1.3: `if-else` akış diyagramı. Koşul doğruysa (E) `if` bloğu, yanlışsa (H) `else` bloğu çalışır; iki yol birleşir ve sonraki deyime geçilir.

Yazımda iki kural vardır. Koşul her zaman parantez içinde yazılır: `if koşul` derlenmez, `if (koşul)` derlenir. Bloklar süslü parantezle açılıp kapanır. Süslü parantezler arasındaki deyim grubuna **blok** denir. Bir blok içinde tanımlanan değişken yalnız o blokta yaşar; bloktan çıkınca adı da kaybolur.

3.1.2.3 Yalnız if ve süslü parantezler

`else` kolu isteğe bağlıdır. Koşul yanlış olduğunda yapılacak özel bir iş yoksa yalnız `if` yazılır:

```
int sicaklik = 35;
if (sicaklik > 30) {
    System.out.println("Hava sıcak!");
}
System.out.println("Program devam ediyor.");
```

```
Hava sıcak!
Program devam ediyor.
```

`sicaklik` 35 olduğu için `35 > 30` doğrudur ve blok çalışır. Ardından `if`'ten sonraki deyime geçilir. Sıcaklık 20 olsaydı koşul yanlış olur, blok atlanır ve ekranda yalnız “Program devam ediyor.” görünürdü. `if` yapısının dışındaki deyim, koşulun sonucundan bağımsız olarak her durumda çalışır.

Süslü parantezler, blokta tek deyim olduğunda yazılmayabilir. Java'nın kuralı şudur: parantez yoksa `if`'e yalnız ilk deyim bağlıdır. Girinti, yani satır başındaki boşluklar, Java için hiçbir anlam taşımaz. Python gibi girintiyle blok kuran dillere alışmış biri için bu durum aşağıdaki tuzağı doğurur:

```
int sicaklik = 35;
if (sicaklik > 40)
    System.out.println("Çok sıcak!");
    System.out.println("Bol su için.");
```

```
Bol su için.
```

`35 > 40` yanlıştır, bu yüzden "Çok sıcak!" yazılmaz. "Bol su için." satırı ise iki satırın aynı hizada durmasına rağmen `if`'in parçası değildir. Derleyicinin gördüğü yapı şudur: `if (sicaklik > 40) println("Çok sıcak!");` ve ardından ayrı, koşulsuz bir `println("Bol su için.");`. Programcının niyeti iki satırı birlikte koşula bağlamaksa bunu yalnız süslü parantezler sağlar. Bu nedenle tek satırlık bloklarda bile `{ }` kullanmak iyi bir alışkanlıktır. Sonradan bloğa ikinci bir satır eklendiğinde aynı hata yeniden doğmaz.

Sık yapılan hata

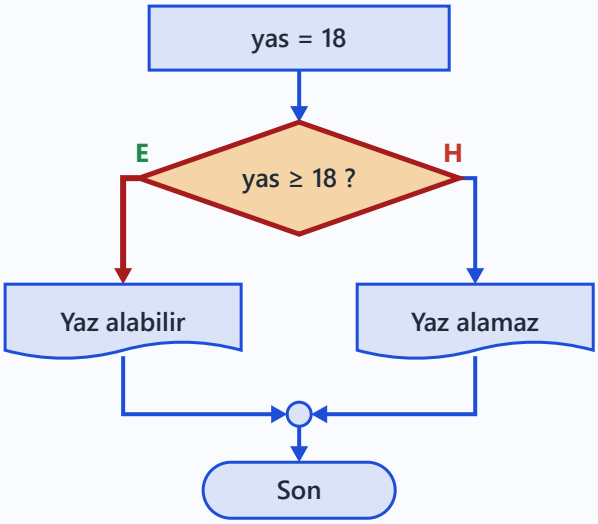
Koşuldan sonra noktalı virgül koymak. `if (x > 5); { … }` derlenir, ama noktalı virgül tek başına **boş deyimdir**. Derleyici bunu "koşul doğruysa hiçbir şey yapma" diye okur. Arkasından gelen blok `if`'e bağlı değildir ve her durumda çalışır. `x = 3` ile denendiğinde `if (x > 5); { System.out.println("x beşten büyük"); }` yine "x beşten büyük" yazar. Derleyici bu satır için uyarı da vermez.

3.1.2.4 Yaş kontrolü

İlk tam karar örneği, ehliyet yaşını denetleyen altı satırlık bir koddur:

```
int yas = 18;
if (yas >= 18) {
    System.out.println("Ehliyet alabilirsiniz.");
} else {
    System.out.println("Ehliyet alamazsınız.");
}
```

Birinci satır `yas` değişkenine 18 atar. İkinci satırda koşul değerlendirilir: `18 >= 18` doğrudur, çünkü `>=` eşitliği de kapsar. Akış diyagramında bu, evet (E) koluna gitmek demektir (Şekil 3.1.4). Üçüncü satır çalışır ve "Ehliyet alabilirsiniz." yazılır. `else` bloğu, yani beşinci satır atlanır.



Şekil 3.1.4: Yaş kontrolünün akış diyagramı. `yas = 18` için koşul doğrudur ve akış kalın çizilen E koluna gider.

`yas`	`yas >= 18`	Çalışan blok	Ekran
16	`false`	`else`	Ehliyet alamazsınız.
18	`true`	`if`	Ehliyet alabilirsiniz.
25	`true`	`if`	Ehliyet alabilirsiniz.

Tablo 3.1.3: Yaş kontrolünün üç değer için sonucu.

Tablodaki 18, bir **sınır değerdir**: koşulun doğru değerden yanlış değere geçtiği nokta. Kararlı kodlar sınırın iki yanında ve tam üstünde denenmelidir. Koşul yanlışlıkla `yas > 18` yazılsaydı 16 ve 25 doğru sonuç verir, yalnız 18 yanlış sonuç verirdi. Bu tür bir hata yalnız sınır değeriyle yapılan denemede ortaya çıkar.

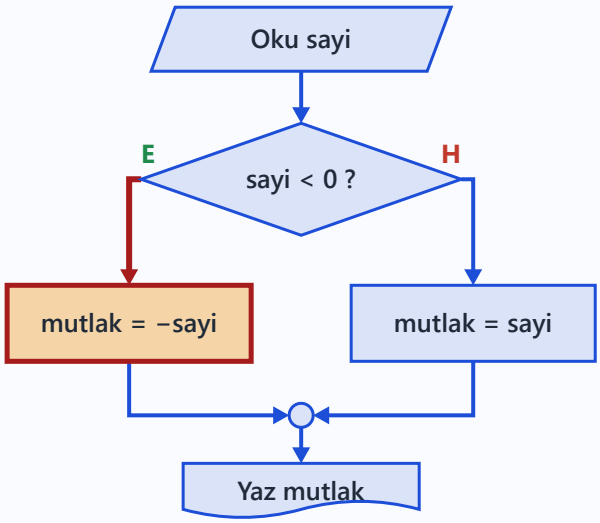
3.1.2.5 Mutlak değer

İkinci örnek kullanıcıdan bir ondalıklı sayı okur ve mutlak değerini yazar. Bir sayının mutlak değeri, sayı negatifse işaret değiştirilmiş hâli, değilse kendisidir: |−7,5| = 7,5 ve |3| = 3.

```
Scanner scanner = new Scanner(System.in);
System.out.print("Bir sayı girin: ");
double sayi = scanner.nextDouble();
double mutlakDeger;
if (sayi < 0) {
    mutlakDeger = -sayi;
} else {
    mutlakDeger = sayi;
}
System.out.println("Mutlak değer: " + mutlakDeger);
```

Kod dosyanın başında `import java.util.Scanner;` satırı olduğunu varsayar. Birinci satır klavyeden okumak için bir `Scanner` nesnesi oluşturur. İkinci satır kullanıcıya ne gireceğini söyler. Burada `println` değil `print` kullanılmıştır: `print` satır atlamaz ve imleç sorunun hemen yanında bekler. Üçüncü satırda `nextDouble()` kullanıcının yazdığı sayıyı okur.

Dördüncü satır `mutlakDeger` değişkenini tanımlar ama değer vermez. Değeri aşağıdaki `if-else` verecektir. Java bu satırı kabul eder, çünkü iki kolun her biri değişkene bir değer yazar. Derleyici bunu denetleyebilir: hangi kol çalışırsa çalışsın, son satıra gelindiğinde değişken mutlaka bir değer almıştır. `else` kolu silinseydi derleyici son satır için `variable mutlakDeger might not have been initialized` hatası verirdi.



Şekil 3.1.5: Mutlak değer akış diyagramı. −7,5 girildiğinde `sayi < 0` doğrudur; işaretlenen kutuda `mutlakDeger = -sayi` çalışır.

Kullanıcı −7,5 girsin. Beşinci satırda `-7.5 < 0` doğrudur. Altıncı satırda `-sayi` ifadesi −(−7,5) = 7,5'i hesaplar ve `mutlakDeger`'e yazar. `else` bloğu atlanır. Son satır `Mutlak değer: 7.5` yazar. Tablo 3.1.4 aynı programın dört girdiyle çalıştırılmasını özetler.

Klavyeden	`sayi`	`sayi < 0`	Ekran
-7,5	−7.5	`true`	Mutlak değer: 7.5
7,5	7.5	`false`	Mutlak değer: 7.5
0	0.0	`false`	Mutlak değer: 0.0
-7.5	—	—	`InputMismatchException`

Tablo 3.1.4: Mutlak değer programı, Türkçe bölge ayarlı bir bilgisayarda.

Son satır Java'ya özgü bir ayrıntıdır. `Scanner`, sayıları bilgisayarın bölge ayarına göre okur. Türkçe ayarda ondalık ayırıcı virgüldür. Bu yüzden `-7,5` kabul edilir, `-7.5` ise sayı sayılmaz ve program şu iletiyle durur:

```
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:947)
        at java.base/java.util.Scanner.next(Scanner.java:1602)
        at java.base/java.util.Scanner.nextDouble(Scanner.java:2573)
```

Okuma bölge ayarına uyar, yazma ise uymaz. `println` bir `double` değeri her zaman noktayla yazar: `7.5`. Aynı programda girişte virgül, çıkışta nokta görülmesinin nedeni budur. Okumanın da noktayla yapılması isteniyorsa `Scanner` oluşturulduktan sonra `scanner.useLocale(java.util.Locale.US)` çağrılır.

3.1.3 Çok yollu kararlar

Bir `if-else` yapısı iki yol arasında seçim yapar. Gerçek problemlerde çoğu zaman daha fazla seçenek vardır. Bir sayı pozitif, negatif ya da sıfır olabilir; bir not A, B, C, D ya da F'ye karşılık gelebilir. Java bu durumlar için iki araç sunar: koşulları art arda dizen `else if` zinciri ve kararları birbirinin içine yerleştiren iç içe yapı. Kesimin sonunda birden çok koşulu tek bir ifadede birleştiren `&&` ve `||` işleçleri ile koşul işleci `?:` işlenir.

3.1.3.1 else-if zinciri

> **Tanım 3.1.4**
>
> **else-if zinciri** (*else-if chain*), koşulları yukarıdan aşağıya sırayla deneyip ilk doğru koşulun bloğunu çalıştıran yapıdır. Hiçbir koşul doğru değilse, varsa, en sondaki `else` bloğu çalışır.

Aşağıdaki tam program bir sayının işaretini bulur. Üç olası sonuç, iki koşul ve bir son `else` ile ayrılır:

```java
import java.util.Scanner;

public class PozitifNegatif {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Bir sayı girin: ");
        double sayi = scanner.nextDouble();
        if (sayi > 0) {
            System.out.println("Sayı pozitif.");
        } else if (sayi < 0) {
            System.out.println("Sayı negatif.");
        } else {
            System.out.println("Sayı sıfır.");
        }
    }
}
```

Program üç ayrı girdiyle çalıştırıldığında şu çıktıları verir:

```
Bir sayı girin: 4
Sayı pozitif.
```

```
Bir sayı girin: -2
Sayı negatif.
```

```
Bir sayı girin: 0
Sayı sıfır.
```

4 girildiğinde ilk koşul `4 > 0` doğrudur. “Sayı pozitif.” yazılır ve zincirin geri kalanına hiç bakılmaz. −2 girildiğinde ilk koşul yanlıştır, akış ikinci koşula geçer: `-2 < 0` doğrudur ve “Sayı negatif.” yazılır. 0 girildiğinde iki koşul da yanlıştır ve son `else` çalışır. Üç kolun hiçbiri aynı anda çalışamaz: bir kol seçildiği anda zincir biter.

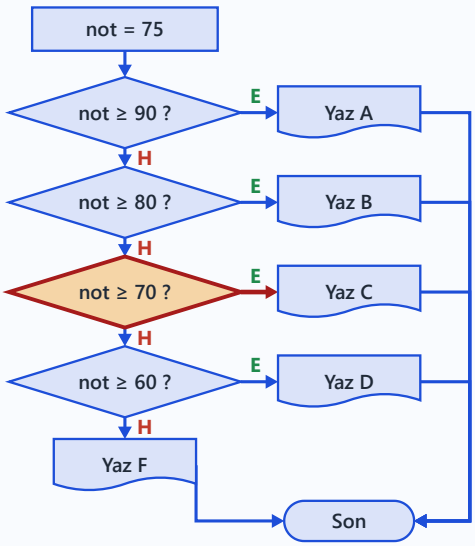
`else if` Java'da ayrı bir anahtar sözcük değildir. Yapı, bir `else` kolunun içine yazılmış yeni bir `if` 'ten ibarettir. Programın derleyicinin gözündeki biçimi şöyledir: `if (sayi > 0) { … } else { if (sayi < 0) { … } else { … } }`. Programcılar bunu daha okunaklı olduğu için tek düzeyde, alt alta yazar. Zincirin her halkası, kendinden önceki bütün koşulların yanlış olduğu bilinerek değerlendirilir. İkinci halkadaki `sayi < 0` koşuluna yalnız pozitif olmayan sayılar ulaşır.

3.1.3.2 Harf notu zincirinin izlenmesi

Zincirin sırayı nasıl kullandığını en iyi harf notu örneği gösterir. Sayısal not 90 ve üstündeyse A, 80–89 arasında B, 70–79 arasında C, 60–69 arasında D, 60'ın altında F verilir.

```
int not = 75;
if (not >= 90) {
    System.out.println("A");
} else if (not >= 80) {
    System.out.println("B");
} else if (not >= 70) {
    System.out.println("C");
} else if (not >= 60) {
    System.out.println("D");
} else {
    System.out.println("F");
}
```

`not = 75` için akış şöyledir. İlk koşul `75 >= 90` yanlıştır ve akış sonraki `else if`'e geçer. `75 >= 80` da yanlıştır. `75 >= 70` doğrudur: ilk doğru koşul bulunmuştur. “C” yazılır. Kalan koşul, yani `not >= 60`, hiç sınanmaz (Şekil 3.1.6).



Şekil 3.1.6: `else-if` zincirinin akış diyagramı, `not = 75`. İlk iki elmasta H koluna gidilir; üçüncü elmasta (`not ≥ 70`) koşul doğrudur ve “Yaz C” kutusuna geçilir.

`not`	`>= 90`	`>= 80`	`>= 70`	`>= 60`	Sınanan koşul	Ekran
95	doğru	—	—	—	1	A
80	yanlış	doğru	—	—	2	B
75	yanlış	yanlış	doğru	—	3	C

`not`	`>= 90`	`>= 80`	`>= 70`	`>= 60`	**Sınanan koşul**	**Ekran**
60	yanlış	yanlış	yanlış	doğru	4	D
45	yanlış	yanlış	yanlış	yanlış	4	F

Tablo 3.1.5: Harf notu zincirinde sınanan koşullar.

Koşullardaki `not < 90` gibi üst sınırların yazılmamış olmasına dikkat edilmelidir. `else if (not >= 80)` halkasına ancak `not >= 90` yanlışsa gelinir; demek ki bu noktada not zaten 90'dan küçüktür. Zincir, üst sınırı sıranın kendisinden alır. Bu yüzden koşulların sırası önemlidir. Aynı dört koşul ters sırayla yazılsın:

```
if (not >= 60) {
    harf = "D";
} else if (not >= 70) {
    harf = "C";
} else if (not >= 90) {
    harf = "A";
} else {
    harf = "F";
}
```

Bu kod 95 için de, 75 için de “D” üretir. 60 ve üstündeki her sayı ilk koşulu sağlar ve zincir ilk halkada biter. `not >= 90` halkasına hiçbir değer ulaşamaz. Derleyici bu hatayı yakalamaz, çünkü kod Java kurallarına uygundur. Yanlış olan mantıktır. Kural basittir: `>=` ile yazılan zincirde eşikler büyükten küçüğe, `<` ile yazılan zincirde küçükten büyüğe dizilir.

Sık yapılan hata

Zincir yerine art arda bağımsız `if` 'ler yazmak. `if (not >= 90) harf = "A"; if (not >= 80) harf = "B"; if (not >= 70) harf = "C";` biçiminde yazılan kodda her `if` ayrı ayrı sınanır. 95 için önce `"A"` , sonra `"B"` , sonra `"C"` atanır ve sonuç C olur. `else` sözcüğü, “önceki koşullar yanlışsa” anlamını taşır; o olmadan koşullar birbirini dışlamaz.

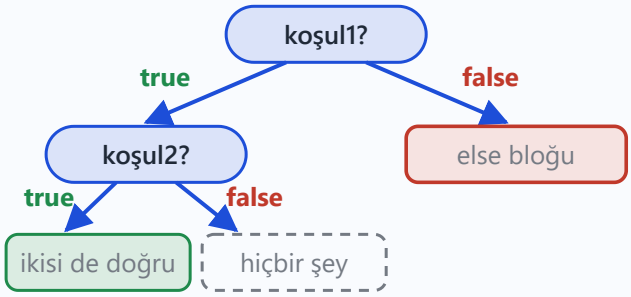
3.1.3.3 İç içe kararlar

Tanım 3.1.5

İç içe karar (*nested decision*), bir karar yapısının bloğu içine yazılmış başka bir karar yapısıdır. İç karar yalnız dış koşul onu çalıştıracak kolu seçtiğinde sınanır.

```
if (koşul1) {
    if (koşul2) {
        // koşul1 VE koşul2 doğru
    }
} else {
    // koşul1 yanlış
}
```

İç içe yapı, ilk sorunun cevabına göre ikinci bir soru sorulacaksa kullanışlıdır (Şekil 3.1.7). İç `if`'in gövdesine ancak `koşul1` ve `koşul2` birlikte doğruysa girilir. `koşul1` yanlışsa iç `if` hiç sınanmaz ve doğrudan dış `else` çalışır.

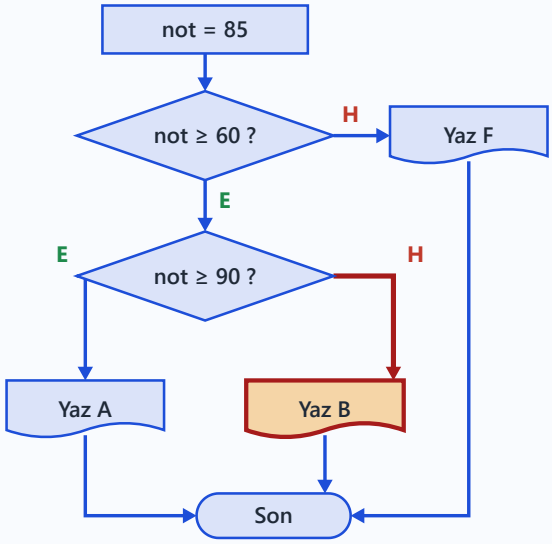


Şekil 3.1.7: İç içe karar ağacı. `koşul1` doğruysa `koşul2` sınanır; yanlışsa dış `else` bloğu çalışır. `koşul1` doğru, `koşul2` yanlışsa hiçbir blok çalışmaz.

Harf notu örneğinin iki kademeli bir sürümü yapıyı somutlaştırır. Önce geçip geçmediğe bakılır, geçen öğrenciler için A ile B ayrılır:

```
int not = 85;
if (not >= 60) {
    if (not >= 90) {
        System.out.println("Harf notu: A");
    } else {
        System.out.println("Harf notu: B");
    }
} else {
    System.out.println("Harf notu: F");
}
```

`not = 85` için dış koşul `85 >= 60` doğrudur ve dış `if` bloğuna girilir. Orada iç koşul sınanır: `85 >= 90` yanlıştır, akış iç `else`'e gider ve “Harf notu: B” yazılır. Dış `else` atlanır (Şekil 3.1.8). `not = 45` olsaydı dış koşul yanlış olur, iç `if` hiç sınanmaz ve doğrudan “Harf notu: F” yazılırdı.



Şekil 3.1.8: İç içe kararın akış diyagramı, `not = 85`. Dış elmasta E koluna, iç elmasta H koluna gidilir ve "Yaz B" kutusu çalışır.

Çok derin iç içe yapılar okunması zor koda yol açar. Dört beş düzey içe girmiş bir `if`'te hangi `else`'in hangi `if`'e ait olduğunu bulmak için parantez saymak gerekir. İki çare vardır. Birincisi düzgün girintidir: her iç düzey dört boşluk içeriden başlar. İkincisi yapıyı düzleştirmektir. Yalnız iki koşulun birlikte doğru olduğu durumla ilgilenilen bir kod `&&` ile tek bir `if`'e indirilebilir: `if (yas >= 18) { if (ehliyetVar) … }` ile `if (yas >= 18 && ehliyetVar) …` aynı işi yapar.

Sarkık else sorunu. Süslü parantezsiz iç içe `if`'lerde bir `else`'in hangi `if`'e bağlandığı belirsiz görünebilir. Java'nın kuralı kesindir: `else`, kendisinden önceki ve henüz `else`'i olmayan **en yakın** `if`'e bağlanır. Aşağıdaki kodda girinti `else`'i iç `if`'e bağlı gösterir ve kural da aynı sonucu verir:

```
int x = 7;
if (x > 5)
    if (x > 10)
        System.out.println("A");
    else
        System.out.println("B");
```

`x = 7` için `7 > 5` doğrudur, iç koşul `7 > 10` yanlıştır ve iç `if`'in `else`'i "B" yazar. Aynı kod `x = 3` ile çalıştırıldığında hiçbir şey yazmaz: dış koşul yanlıştır ve dış `if`'in `else`'i yoktur. `else` sola, dış `if`'in hizasına kaydırılsa da sonuç değişmez; girinti derleyici için anlamsızdır. `else` gerçekten dış `if`'e bağlanmak isteniyorsa iç `if` süslü parantezle kapatılır:

```
int x = 7;
if (x > 5) {
    if (x > 10) {
        System.out.println("A");
    }
} else {
    System.out.println("B");
}
```

Bu sürüm `x = 7` için hiçbir şey, `x = 3` için "B" yazar. İki kod aynı satırlardan oluşur, ama parantezler `else`'in sahibini değiştirmiştir.

3.1.3.4 Birleşik koşullar ve kısa devre

Birden çok koşul mantıksal işleçlerle tek bir koşulda birleştirilir. Java'nın üç mantıksal işleci vardır: `&&` (ve), `||` (veya) ve `!` (değil). `a && b` ancak iki taraf da doğruysa doğrudur. `a || b`, taraflardan en az biri doğruysa doğrudur. `!a`, `a`'nın tersidir.

a	b	a && b	a || b	!a
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

Tablo 3.1.6: Mantıksal işleçlerin doğruluk tablosu.

Mantıksal işleçlerin en sık kullanıldığı yer aralık denetimidir. Notun 60 ile 89 arasında olup olmadığı matematikteki gibi `60 <= not < 90` biçiminde yazılamaz. Java bu ifadeyi soldan sağa okur: önce `60 <= not` hesaplanır ve bir `boolean` çıkar. Ardından bu `boolean` 90 ile karşılaştırılmaya çalışılır, ama bir `boolean`'ın 90'dan küçük olup olmadığı sorulamaz. Derleyici şu iletiyi verir:

```
error: bad operand types for binary operator '<'
  first type:  boolean
  second type: int
```

Doğru yazım iki karşılaştırmayı `&&` ile bağlar: `not >= 60 && not < 90`. Burada `||` kullanmak da yaygın bir hatadır. `not >= 60 || not < 90` her sayı için doğrudur: 60'tan küçük her sayı 90'dan da küçüktür.

Tanım 3.1.6

Kısa devre değerlendirme (*short-circuit evaluation*): `&&` ve `||` önce sol tarafı hesaplar. Sonuç yalnız sol taraftan belli oluyorsa sağ taraf hiç hesaplanmaz. `&&`'de sol taraf `false` ise sonuç `false`, `||`'de sol taraf `true` ise sonuç `true`'dur.

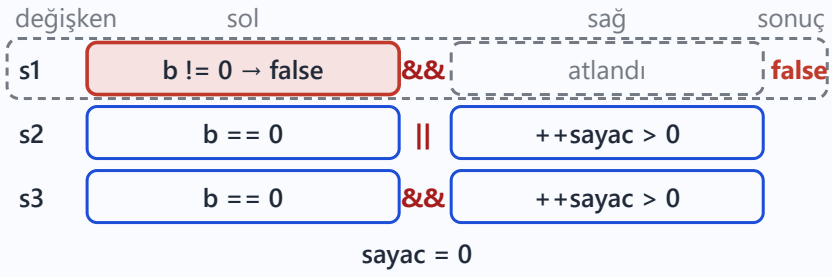
Kısa devre bir hız ayrıntısından fazlasıdır; programın davranışını değiştirir. Aşağıdaki kod bunu üç satırda gösterir:

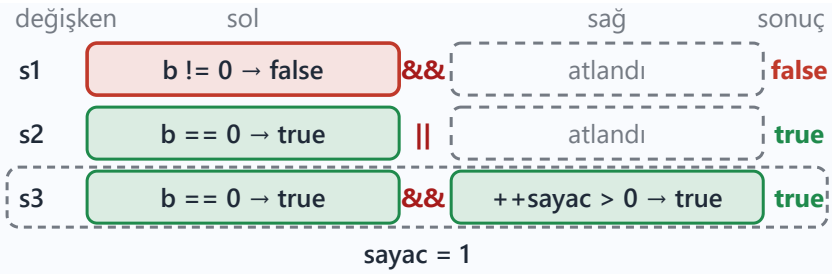
```
int a = 10, b = 0, sayac = 0;
boolean s1 = b != 0 && a / b > 2;
boolean s2 = b == 0 || ++sayac > 0;
boolean s3 = b == 0 && ++sayac > 0;
System.out.println(s1 + " " + s2 + " " + s3);
System.out.println("sayac = " + sayac);
```

```
false true true
sayac = 1
```

Başlangıçta `b = 0`'dır. `a / b` hesaplanırsa sıfıra bölme hatası (`ArithmeticException`) oluşur. `sayac` ise sağ tarafların çalışıp çalışmadığını saymak için konmuştur: `++sayac` her çalıştığında 1 artar.

- `s1`: `&&` önce sol tarafı hesaplar: `b != 0` yanlıştır. Sol taraf yanlışsa `&&`'nin sonucu sağ taraf ne olursa olsun yanlıştır. `a / b > 2` hiç hesaplanmaz ve hata oluşmaz. `s1 = false`.
- `s2`: `||` için sol taraf `b == 0` doğrudur. Sonuç zaten doğrudur; `++sayac` çalışmaz ve `sayac` 0 kalır. `s2 = true`.
- `s3`: yine `&&`; sol taraf `b == 0` doğrudur. Sonuç henüz belli değildir, sağ tarafın da hesaplanması gerekir. `++sayac` çalışır ve `sayac` 1 olur; `1 > 0` doğrudur. `s3 = true`.





Şekil 3.1.9: Kısa devre değerlendirmenin iki anı. Üstte `s1` hesaplanmıştır: sol taraf `false` olduğu için sağ taraf "atlandı" olarak işaretlidir. Altta üç satır da bitmiştir: yalnız `s3`'ün sağ tarafı çalışmış ve `sayac` 1 olmuştur.

Sonuçta üç sağ taraftan yalnız biri çalışmıştır. Bu davranışın iki yüzü vardır. Yararlı yüzü, `s1`'deki gibi tehlikeli bir işlemi korumaktır. `b != 0 && a / b > 2` yazıldığında bölme ancak payda sıfır değilse yapılır. Aynı kalıp nesnelerde de sık görülür: `s != null && s.length() > 0`. Tehlikeli yüzü ise yan etkilidir. Sağ tarafa `++sayac` gibi değişken değiştiren bir işlem yazılırsa bu işlemin çalışıp çalışmayacağı sol tarafa bağlı olur. Kodu okuyan kişi bunu kolayca gözden kaçırır. Yan etkili işlemleri `&&` ve `||`'nin sağına yazmamak en güvenli yoldur.

Java'da kısa devre yapmayan iki işleç daha vardır: `&` ve `|`. `boolean` değerlerle kullanıldıklarında iki tarafı da her zaman hesaplarlar. `b != 0 & a / b > 2` yazılırsa sol taraf yanlış olduğu hâlde bölme yapılır ve program `java.lang.ArithmeticException: / by zero` ile durur. Koşullarda tek `&` yazmak çoğu zaman bir yazım hatasıdır.

3.1.3.5 Koşul işleci

Bir koşula göre iki değerden birini seçmek çok sık yapılan bir iştir. Bunun için `if-else` yazılabilir:

```
int a = 17, b = 42;
int enBuyuk;
if (a > b) {
    enBuyuk = a;
} else {
    enBuyuk = b;
}
System.out.println(enBuyuk);
```

Java aynı işi tek bir ifadeyle yapan **koşul işlecini** (*conditional operator*) sunar. Üç parçası olduğu için üçlü işleç (*ternary operator*) diye de anılır: `koşul ? değer1 : değer2`. Koşul doğruysa ifadenin değeri `değer1`, yanlışsa `değer2`'dir.

```
int a = 17, b = 42;
int enBuyuk = (a > b) ? a : b;
System.out.println(enBuyuk);
```

`17 > 42` yanlış olduğu için ikinci değer seçilir ve 42 yazılır. `if-else` bir **deyimdir**, bir iş yapar. `?:` ise bir **ifadedir**, bir değer üretir. Bu yüzden atamanın sağına, bir metodun argümanına ya da bir dizgi birleştirmesinin içine yazılabilir: `System.out.println(not >= 60 ? "geçti" : "kaldı");`. İç içe koşul işleçleri (`a ? b : c ? d : e`) çabuk okunmaz hâle gelir; ikiden çok seçenekte `else if` zinciri tercih edilir. İki sayıdan büyüğünü bulmak için Java'nın hazır `Math.max(a, b)` metodu da kullanılabilir.

3.1.4 switch-case

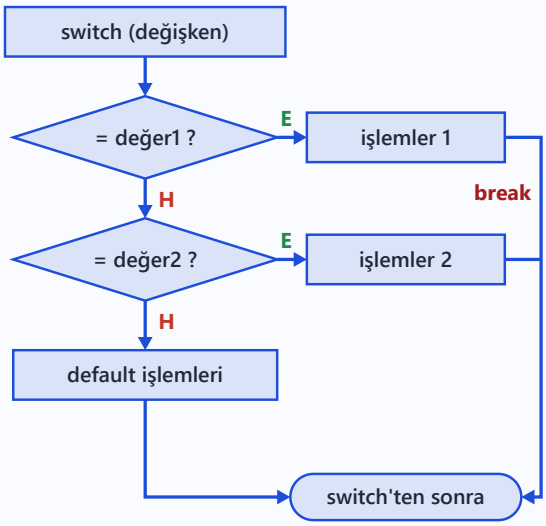
Bazı kararlar hep aynı değişkenin farklı sabit değerlere eşit olup olmadığını sorar: ay 1 mi, 2 mi, 3 mü...; işlem `'+'` mı, `'-'` mi? Bu tür kararlar `else if` zinciriyle de yazılabilir, ama on iki halkalı bir zincirde her halkada `ay ==` tekrarlanır. `switch` yapısı bu durum için tasarlanmıştır.

3.1.4.1 Yapı ve işleyiş

> **Tanım 3.1.7**
>
> **switch deyimi** (*switch statement*), parantez içindeki ifadenin değerini `case` etiketlerindeki sabitlerle karşılaştırır ve eşleşen etiketin altındaki koda geçer. Eşleşen etiket yoksa, varsa, `default` etiketinin altındaki kod çalışır.

```
switch (değişken) {  // kontrol edilen değer
    case değer1:     // her case bir değer
        // İşlemler 1
        break;
    case değer2:
        // İşlemler 2
        break;
    // ...
    default:         // hiçbiri eşleşmezse
        // Varsayılan işlemler (opsiyonel)
}
```

Parantez içindeki ifadeye **seçici** denir. Seçici bir kez hesaplanır. Değeri `case` etiketleriyle yukarıdan aşağıya karşılaştırılır. Eşleşen etiket bulunursa onun altındaki deyimler çalışır. `break`'e gelindiğinde `switch`'ten çıkılır ve kapanış parantezinden sonraki deyime geçilir. Hiçbir etiket eşleşmezse `default` bölümü çalışır. `default` da yoksa hiçbir şey yapılmadan `switch`'ten çıkılır (Şekil 3.1.10).



Şekil 3.1.10: `switch-case` işleyişi. Seçici sırayla `değer1` ve `değer2` ile karşılaştırılır. Eşleşen `case`'in işlemleri çalışır ve `break` ile `switch`'ten çıkılır; hiçbiri eşleşmezse `default` işlemleri çalışır.

Gündelik bir benzetme asansör düğmeleridir. Basılan kat numarası, gidilecek katı doğrudan seçer. Asansör her katta "burası 3. kat mı?" diye durmaz. `switch` da çoğu zaman böyle çalışır: kavramsal olarak etiketlerin sırayla denendiği düşünülebilir, ama derleyici etiketler sık dizilmiş tam sayılarsa bir atlama tablosu kurar ve doğrudan doğru etikete atlar.

3.1.4.2 Mevsim belirleme

Ay numarasından mevsimi bulan kod, `switch`'in bütün parçalarını bir arada gösterir:

```
int ay = 4;
String mevsim;
switch (ay) {
    case 3: case 4: case 5:
        mevsim = "İlkbahar";
        break;
    case 6: case 7: case 8:
        mevsim = "Yaz";
        break;
    case 9: case 10: case 11:
        mevsim = "Sonbahar";
        break;
    case 12: case 1: case 2:
        mevsim = "Kış";
        break;
    default:
        mevsim = "Bilinmiyor";
}
System.out.println(ay + ". ay: " + mevsim);
```

```
4. ay: İlkbahar
```

Birinci satır `ay`'a 4 atar. İkinci satır `mevsim`'i tanımlar ama değer vermez. Üçüncü satırda seçici `ay` bir kez hesaplanır: değeri 4'tür. Ardından eşleşen etiket aranır. Dördüncü satırda aynı satıra yazılmış üç etiket vardır: `case 3: case 4: case 5:`. 4 bunlardan biriyle eşleşir. Üç etiketin altında tek bir kod bulunur; hangisi eşleşirse eşleşsin akış aynı koda iner. `mevsim`'e `"İlkbahar"` atanır. `break`, akışı `switch`'in kapanış parantezinin arkasına taşır. Son satır "4. ay: İlkbahar" yazar.



Şekil 3.1.11: Mevsim örneğinde `case` grupları. `ay = 4` ilk grupla eşleşir ve `mevsim` "İlkbahar" olur.

`default` dalı burada iki iş görür. Birincisi, 0, 13 ya da −5 gibi geçersiz ay numaralarına bir cevap verir. İkincisi, derleyicinin değişken denetimini geçmeyi sağlar. `mevsim` yalnız `case`'lerde atansaydı derleyici şöyle düşünürdü: seçici hiçbir etiketle eşleşmezse `mevsim` hiç değer almaz. Son satırdaki okuma bu yüzden reddedilir:

```
error: variable mevsim might not have been initialized
        System.out.println(mevsim);
                           ^
```

`default` eklenince `switch`'ten çıkan her yol `mevsim`'e bir değer yazmış olur ve hata kaybolur. 0'dan 13'e kadar bütün ay değerleri denendiğinde 0 ve 13 için "Bilinmiyor", diğerleri için doğru mevsim yazılır.

3.1.4.3 Aşağı düşme

Tanım 3.1.8

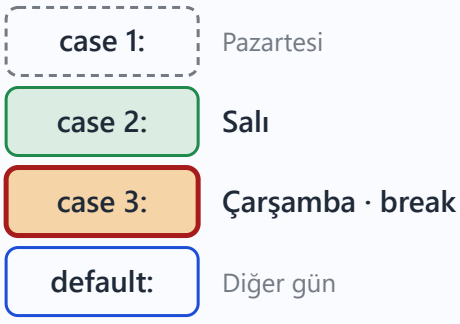
Aşağı düşme (*fall-through*), `break` ile bitmeyen bir `case`'in kodundan sonra alttaki `case`'lerin kodunun da çalışmasıdır.

`case` etiketi bir karşılaştırma deyimi değil, bir **giriş noktasıdır**. `switch` yalnız nereden başlanacağını seçer. Akış bir kez içeri girdikten sonra alttaki etiketlere bakmaz; `break`'e ya da `switch`'in sonuna kadar satır satır ilerler. Aşağıdaki kodda ilk iki `case`'in `break`'i yoktur:

```
int gun = 2;
switch (gun) {
    case 1:
        System.out.println("Pazartesi");
    case 2:
        System.out.println("Salı");
    case 3:
        System.out.println("Çarşamba");
        break;
    default:
        System.out.println("Diğer gün");
}
System.out.println("switch bitti");
```

```
Salı
Çarşamba
switch bitti
```

`gun = 2` ile seçici hesaplanır. `case 1` atlanır, `case 2` eşleşir ve "Salı" yazılır. Burada `break` yoktur. Akış alttaki `case 3:` satırına gelir. Bu etiket "gün 3 mü?" diye sormaz; akış onun altındaki koda düşer ve "Çarşamba" da yazılır, oysa gün 3 değildir. Ardından gelen `break` `switch`'ten çıkarır ve `default` çalışmaz. Son olarak "switch bitti" yazılır (Şekil 3.1.12).



Şekil 3.1.12: Aşağı düşme. `case 2` çalışmış (yeşil), `break` olmadığı için akış `case 3`'e düşer (işaretli). `case 1` atlanmıştır.

`gun`	Giriş noktası	Çalışan kod	Ekran
1	`case 1`	case 1, case 2, case 3, `break`	Pazartesi, Salı, Çarşamba
2	`case 2`	case 2, case 3, `break`	Salı, Çarşamba

`gun`	Giriş noktası	Çalışan kod	Ekran
5	`default`	default	Diğer gün

Tablo 3.1.7: Aynı `switch`'in üç değerle çıktısı.

Aşağı düşme her zaman hata değildir. Mevsim örneğindeki `case 3: case 4: case 5:` satırı aslında bu davranışın bilerek kullanılmasıdır: `case 3` ve `case 4` boş etiketlerdir ve akış alttaki ortak koda düşer. Hata, düşmenin istenmediği yerde `break`'in unutulmasıdır. Derleyici bu durumda uyarı vermez. Bilerek bırakılan düşmeler bu yüzden genellikle bir açıklama satırıyla işaretlenir: `// break yok: aşağı düşme istenir`.

Örnek 3.1.1

Aşağıdaki kod ne yazar?

```
int k = 2;
int puan = 0;
switch (k) {
    case 1:  puan += 10;
    case 2:  puan += 20;
    case 3:  puan += 30;  break;
    case 4:  puan += 40;
}
System.out.println(puan);
```

Seçici 2'dir; akış `case 2`'den girer ve `puan` 20 olur. `break` olmadığı için `case 3`'e düşülür ve `puan` 50 olur. Bu satırdaki `break` `switch`'ten çıkarır; `case 4` çalışmaz. Ekrana 50 yazılır. `k = 1` olsaydı 10 + 20 + 30 = 60, `k = 4` olsaydı 40 yazılırdı.

3.1.4.4 char ve String ile switch

Seçici yalnız `int` olmak zorunda değildir. Bir `char` seçiciyle basit bir hesap makinesi yazılabilir:

```
int a = 12, b = 4;
char islem = '*';
int sonuc;
switch (islem) {
    case '+':  sonuc = a + b;  break;
    case '-':  sonuc = a - b;  break;
    case '*':  sonuc = a * b;  break;
    case '/':  sonuc = a / b;  break;
    default:
        System.out.println("Geçersiz işlem");
        sonuc = 0;
}
System.out.println(a + " " + islem + " " + b
        + " = " + sonuc);
```

```
12 * 4 = 48
```

`char` sabitleri tek tırnakla yazılır: `'+'`, `'*'`. Kısa `case`'ler tek satıra sığdırılabilir: `case '+': sonuc = a + b; break;`. Derleyici için satır sonlarının önemi yoktur. `islem = '*'` için üçüncü etiket eşleşir, `sonuc` 48 olur ve "12 * 4 = 48" yazılır. `islem = '%'` gibi tanımsız bir işlem için `default` çalışır: "Geçersiz işlem" ve ardından "12 % 4 = 0" yazılır. `default`'taki `sonuc = 0;` satırı burada da zorunludur; yoksa son satır `might not have been initialized` hatası verir.

Bu kodda gözden kaçan bir tehlike vardır: `b = 0` ve `islem = '/'` için `a / b` çalışır ve program `ArithmeticException: / by zero` ile durur. Gerçek bir hesap makinesinde `'/'` kolunun önce `b != 0` koşulunu denetlemesi gerekir.

Java 7'den beri seçici bir `String` de olabilir:

```
String komut = "dur";
switch (komut) {
    case "basla":
        System.out.println("Motor çalıştı");
        break;
    case "dur":
        System.out.println("Motor durdu");
        break;
    default:
        System.out.println("Bilinmeyen komut");
}
```

```
Motor durdu
```

Etiketler çift tırnaklı dizgi sabitleridir. Karşılaştırma `equals` metodunun yaptığı gibi içerik üzerinden yapılır; `==` ile yapılan nesne kimliği karşılaştırması değildir. Karşılaştırma büyük-küçük harfe duyarlıdır: `komut = "Dur"` için hiçbir etiket eşleşmez ve "Bilinmeyen komut" yazılır. Kullanıcının yazdığı metin önce küçük harfe çevrilirse bu sorun ortadan kalkar:

```
String komut = "Dur";
switch (komut.toLowerCase()) {
    case "basla":
        System.out.println("Motor çalıştı");
        break;
    case "dur":
        System.out.println("Motor durdu");
        break;
    default:
        System.out.println("Bilinmeyen komut");
}
```

Bu kod "Motor durdu" yazar. Türkçe metinlerde `toLowerCase()` bilgisayarın bölge ayarına bağlıdır: Türkçe ayarda "I" harfi "ı"ya döner. Yalnız ASCII harflerinden oluşan komutlar için `toLowerCase(java.util.Locale.ROOT)` kullanmak bu farkı ortadan kaldırır.

3.1.4.5 switch kuralları

Klasik `switch`'in kuralları, yapının bir atlama tablosuna çevrilebilmesi için konmuştur. Her kural bir derleyici iletisiyle denetlenir (Tablo 3.1.8).

- **Seçicinin tipi.** `byte`, `short`, `char`, `int`, bunların sarmalayıcıları (`Byte`, `Short`, `Character`, `Integer`), `String` ve `enum` olabilir. `long`, `float`, `double` ve `boolean` ile klasik `switch` yazılamaz.
- **Etiketler sabittir.** `case` değerleri derleme sırasında bilinen sabitler olmalıdır: `5`, `'a'`, `"dur"` ya da değeri sabit olan bir `final` değişken. Sıradan bir değişken etiket olamaz.
- **Etiketler tektir.** Aynı değer iki `case`'te kullanılamaz.
- **Yalnız eşitlik sınanır.** Aralık (`ay >= 3 && ay <= 5`) ya da birleşik koşul `case`'e yazılamaz. Bunlar için `if-else` gerekir.

Kod	Derleyici iletisi
`double d = 2.5; switch (d) { … }`	`error: selector type double is not allowed`
`long d = 2; switch (d) { … }`	`error: selector type long is not allowed`
`int b = 2; switch (a) { case b: break; }`	`error: constant expression required`
`switch (a) { case 1: break; case 1: break; }`	`error: duplicate case label`

Tablo 3.1.8: `switch` kurallarının ihlali ve JDK 21 derleyicisinin iletileri.

Üçüncü satırdaki `b`, `final int b = 2;` olarak tanımlansaydı kod derlenirdi. `final` bir değişkenin değeri değişemez ve sabit bir ifadeyle verilmişse derleyici onu bir sabit gibi kullanır. Neden böyle bir kural olduğu, etiketlerin derleme sırasında tabloya yazıldığı düşünülünce anlaşılır: tabloyu kuran derleyici, çalışma sırasında değişebilecek bir değeri bilemez.

3.1.4.6 switch mi, else-if mi?

İki yapı aynı işi farklı güçlü yanlarla yapar. Tablo 3.1.9 farkları bir araya getirir.

	`if-else` / `else if`	`switch-case`
Ne sınanır?	Her türlü `boolean` koşul: aralık, birleşik koşul, farklı değişkenler	Tek bir ifadenin sabit değerlerle eşitliği
Esneklik	Daha esnek	Yalnız belirli tipler ve sabit `case` değerleri
Çok seçenekte	Uzun zincirler karmaşık görünebilir	Düzenli ve okunaklı bir liste
Tuzak	Koşul sırası yanlışsa yanlış dal seçilir	`break` unutulursa alttaki `case`'lere düşülür
Örnek	`if (not >= 90) … else if (not >= 80) …`	`switch (ay) { case 3: … }`

Tablo 3.1.9: `if-else` zinciri ile `switch-case`'in karşılaştırması.

Her `switch` bir `else if` zinciriyle yazılabilir: her `case` bir `==` karşılaştırmasına, `default` son `else`'e dönüşür. Tersi her zaman mümkün değildir. Harf notu örneğindeki `not >= 90` gibi aralıklar `switch`'e ancak 90'dan 100'e kadar on bir ayrı etiket yazılarak aktarılabilir. Seçim kuralı şöyle özetlenebilir: tek bir değişken birkaç sabitle karşılaştırılıyorsa `switch`, aralıklar ya da birden çok değişken varsa `if-else`.

3.1.4.7 switch ifadesi

Java 14 ile `switch`'e yeni bir biçim eklendi. Mevsim örneği bu biçimle şöyle yazılır:

```
int ay = 11;
String mevsim = switch (ay) {
    case 3, 4, 5 -> "İlkbahar";
    case 6, 7, 8 -> "Yaz";
    case 9, 10, 11 -> "Sonbahar";
    case 12, 1, 2 -> "Kış";
    default -> "Bilinmiyor";
};
System.out.println(mevsim);
```

```
Sonbahar
```

Tanım 3.1.9

switch ifadesi (*switch expression*), ok (`->`) etiketleri kullanan ve bir değer üreten `switch` biçimidir. Java 14'ten beri dilin standart parçasıdır.

Yeni biçimin dört farkı vardır. Birincisi, bir `case` virgülle ayrılmış birden çok değer alabilir: `case 3, 4, 5` . İkincisi, ok biçiminde aşağı düşme yoktur. Her kolun sağındaki ifade ya da blok çalışır ve `switch` biter; `break` gerekmez. Üçüncüsü, `switch` bir değer üretir ve bu değer doğrudan bir değişkene atanabilir. Atama bir deyim olduğu için sondaki kapanış parantezinden sonra noktalı virgül konur: `};` . Dördüncüsü, ifade biçimi bütün olasılıkları kapsamalıdır. Seçici bir `int` olduğunda olası değerler sonsuza yakın olduğu için `default` zorunludur. `default` silinirse derleyici şu iletiyi verir:

```
error: the switch expression does not cover all possible input values
```

Bir kolda tek bir değer yerine birkaç deyim gerekiyorsa sağ taraf bir blok olur ve üretilecek değer `yield` ile verilir:

```
int gun = 3;
String tur = switch (gun) {
    case 6, 7 -> "hafta sonu";
    default -> {
        System.out.println("iş günü sayısı hesaplanıyor");
        yield "hafta içi";
    }
};
System.out.println(tur);
```

```
iş günü sayısı hesaplanıyor
hafta içi
```

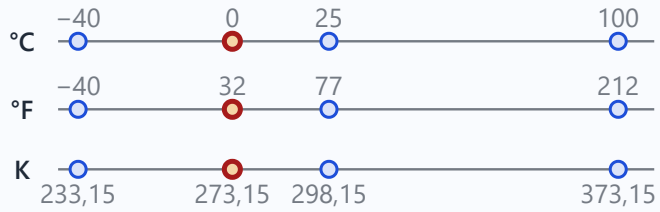
Ok biçimi ile iki noktalı biçim aynı `switch` 'te karıştırılamaz; derleyici `different case kinds used in the switch` iletisini verir. Ok biçimi, değer üretmeyen bir deyim olarak da kullanılabilir: `switch (gun) { case 6, 7 -> System.out.println("tatil"); default -> System.out.println("iş"); }` . Bu derste klasik biçim esas alınır, çünkü eski kodların çoğu onunla yazılmıştır ve aşağı düşme davranışı yalnız onda görülür. Java 14 ve sonrasıyla yazılan yeni kodlarda ok biçimi daha güvenlidir: `break` unutma hatası bu biçimde yapılamaz.

3.1.5 Uygulama: sıcaklık dönüşümü

Kararlardan döngülere geçmeden önce küçük bir sıralı program, formüllerin Java'ya nasıl çevrildiğini gösterir. Problem, bir Celsius değerini Fahrenheit ve Kelvin ölçeklerine dönüştürmektir. Üç ölçeğin birkaç ortak noktası Tablo 3.1.10'da verilmiştir.

Celsius (°C)	Fahrenheit (°F)	Kelvin (K)	Anlamı
−40	−40	233,15	İki ölçeğin kesiştiği nokta
0	32	273,15	Suyun donma noktası
25	77	298,15	Oda sıcaklığı
100	212	373,15	Suyun kaynama noktası (deniz düzeyinde)

Tablo 3.1.10: Sıcaklık ölçekleri arasında karşılık gelen değerler.



Şekil 3.1.13: Üç sıcaklık ölçeği aynı eksende. Kırmızı noktalar 0 °C'nin, yani suyun donma noktasının üç ölçekteki karşılığıdır.

Dönüşüm formülleri şunlardır: °F = (°C × 1,8) + 32, °C = (°F − 32) / 1,8 ve K = °C + 273,15. Java programı ilk ve son formülü tek bir değer için hesaplar:

```
public class Program {
    public static void main(String[] args) {
        double celsius = 25.0;
        double fahrenheit = (celsius * 1.8) + 32;
        double kelvin = celsius + 273.15;
        System.out.println(celsius + " °C = " + fahrenheit + " °F = " + kelvin + " K");
    }
}
```

```
25.0 °C = 77.0 °F = 298.15 K
```

Program sıralıdır: üç satır hesaplama, bir satır yazdırma; karar ya da döngü yoktur. Ondalıklı değerler için `double` kullanılır. Koddaki `1.8` ve `273.15` Java sabitleridir ve kodda ondalık ayırıcı her zaman noktadır. Parantez önceliği açıkça gösterir: önce `celsius * 1.8`, sonra `+ 32`. Çarpma toplamadan

önce yapıldığı için parantez yazılmasa da sonuç aynı olurdu, ama formülün okunuşunu kolaylaştırır. Yazdırma satırındaki `+` işaretleri toplama değil birleştirmedir: bir dizgiyle (`String`) yan yana gelen `+` sayıyı metne çevirip ekler. `double` değerler `25.0` biçiminde yazılır.

Kayan noktalı sayılar her değeri tam tutamaz. 1,8 ikili sistemde sonlu bir kesirle yazılamadığı için bazı sonuçlar küçük bir hata taşır. 37 °C için `celsius * 1.8 + 32` ifadesi `98.60000000000001` yazar; −40 °C için `celsius + 273.15` ise `233.14999999999998` yazar. Bu sayılar yanlış değildir, yalnız son basamakta yuvarlama izi taşır. Ekranda düzgün görünmeleri için `System.out.printf("%.2f%n", kelvin)` ile iki basamağa yuvarlanarak yazdırılırlar. `printf` , `println` 'ın aksine bölge ayarına uyar: Türkçe ayarlı bir bilgisayarda çıktı `233,15` olur.

Sık yapılan hata

Formülü tam sayılarla yazmak. `int c = 37;` için `9 / 5 * c + 32` ifadesi 69 verir: `9 / 5` iki `int` 'in bölümüdür, kesir atılır ve sonuç 1'dir. `c / 5 * 9 + 32` ise 37 / 5 = 7 yüzünden 95 verir. `c * 9 / 5 + 32` soldan sağa hesaplanır: 333 / 5 = 66, sonuç 98; yine de 98,6 yerine 98'dir. Kesirli sonuç için işlemlerden en az biri `double` olmalıdır: `c * 1.8 + 32` ya da `c * 9.0 / 5 + 32` .

3.1.6 Döngüler ve for

Tanım 3.1.10

Döngü (*loop*), bir deyim grubunu bir koşul sağlandıkça tekrarlayan kontrol yapısıdır. Tekrarlanan deyim grubuna **gövde**, gövdenin her bir çalışmasına **tur** (*iteration*) denir.

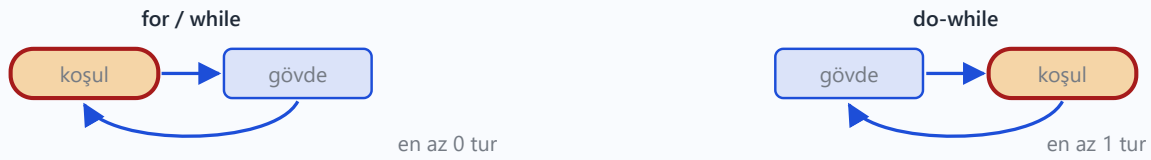
Döngünün gündelik karşılığı pistte on tur koşmaktır. Her turda aynı hareketler yapılır ve tur sayacı bir artar. Sayaç ona ulaşınca koşu biter. Programda da durum aynıdır: gövde aynı kalır, her turda değişen bir ya da birkaç değişken vardır ve bir koşul ne zaman durulacağını söyler. Döngüler bir veri kümesini sırayla işlemek için de kullanılır. Bir dizinin her öğesine bakmak, bir dosyanın her satırını okumak ya da bir ölçüm serisinin ortalamasını almak döngüyle yapılır. Aynı satırı 100 kez yazmak yerine tek bir döngü yazılır; kod kısalır ve tekrar sayısı tek bir sayı değiştirilerek ayarlanır.

3.1.6.1 Döngü türleri

Java'nın üç döngüsü vardır. Üçü de aynı fikre dayanır, fark koşulun nerede sınandığında ve başlangıç ile güncellemenin nereye yazıldığındadır.

- `for` : tekrar sayısı belli olduğunda kullanılır. Başlangıç, koşul ve güncelleme tek satırdadır: `for (int i = 0; i < 5; i++) { … }` .

- `while`: bir koşul sağlandığı sürece çalışır; tekrar sayısı önceden bilinmeyebilir: `while (koşul) { … }`.
- `do-while`: koşul sona yazılır; gövde en az bir kez çalışır: `do { … } while (koşul);`.

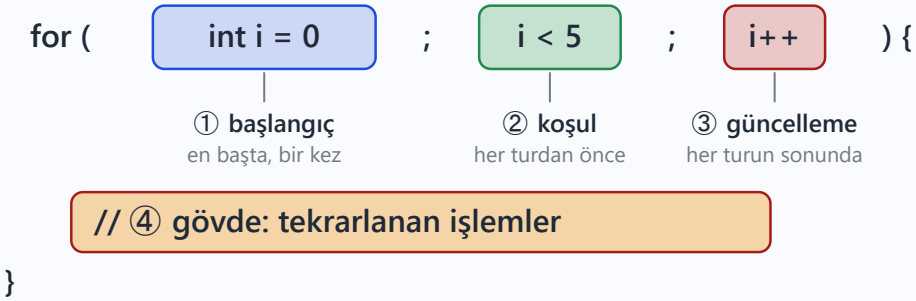


Şekil 3.1.14: Koşulun yeri. `for` ve `while` koşulu gövdeden önce sınar; gövde hiç çalışmayabilir (en az 0 tur). `do-while` koşulu gövdeden sonra sınar; gövde en az bir kez çalışır.

3.1.6.2 for döngüsü

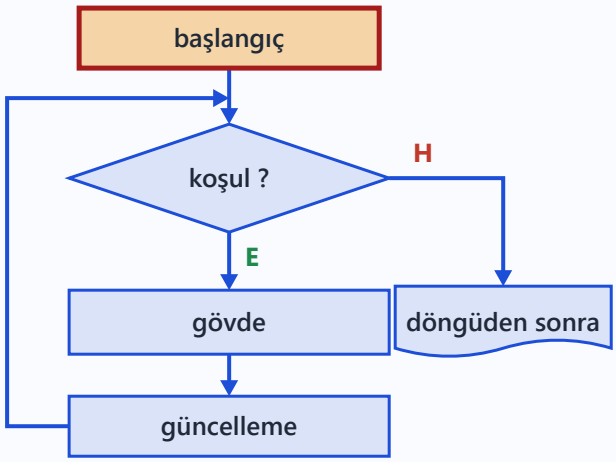
Tanım 3.1.11

for döngüsü (*for loop*), başlangıç, koşul ve güncelleme adımlarını başlığında, aralarında noktalı virgülle, tek satırda toplayan döngüdür.



Şekil 3.1.15: `for` başlığının parçaları. ① Başlangıç en başta bir kez çalışır. ② Koşul her turdan önce sınanır. ③ Güncelleme her turun sonunda çalışır. ④ Gövde tekrarlanan işlemlerdir.

Başlığın üç parçası farklı zamanlarda çalışır (Şekil 3.1.15). Çalışma sırası şöyledir: 1 → 2 → 4 → 3 → 2 → 4 → 3 → … → 2 yanlış → döngüden çık. Yani başlangıç bir kez, ardından her turda koşul, gövde ve güncelleme. Koşul yanlış çıktığında gövde çalışmaz ve döngüden sonraki deyime geçilir. `for (int i = 0; i < 5; i++)` için gövde `i` = 0, 1, 2, 3, 4 değerleriyle 5 kez çalışır. `i` 5 olunca koşul yanlış olur. Koşul toplam 6 kez sınanır: beşi doğru, sonuncusu yanlış.



Şekil 3.1.16: `for` döngüsünün akış diyagramı. Başlangıç bir kez çalışır; koşul doğruysa gövde ve güncelleme yapılıp koşula dönülür, yanlışsa döngüden çıkılır.

Şekil 3.1.16 aynı sırayı akış diyagramıyla gösterir. Döngünün işleyişi altı adımda özetlenebilir:

1. Başlangıç: döngü değişkeni ilk değerini alır. Bu adım bir kez yapılır.
2. Koşul kontrol edilir.
3. Koşul doğruysa gövde çalışır.
4. Güncelleme: döngü değişkeni değişir.
5. Koşula geri dönülür; koşul doğru oldukça 3–4–5 tekrarlanır.
6. Koşul yanlış olunca döngüden çıkılır ve sonraki deyimle devam edilir.

Koşul ilk kontrolde yanlışsa gövde hiç çalışmaz. `for (int i = 10; i <= 1; i++)` başlığı geçerlidir, ama gövdesi sıfır kez çalışır.

Başlıkta tanımlanan `int i` yalnız döngünün içinde yaşar. Döngüden sonra `i`'yi okumaya çalışmak `cannot find symbol` hatası verir. Döngü değişkeninin son değeri döngüden sonra gerekiyorsa değişken döngüden önce tanımlanır ve başlıkta yalnız ilk değeri verilir: `int i; for (i = 1; i < 6; i++) { … }`. Bu durumda döngü bittiğinde `i` 6'dır, çünkü döngü `i < 6` yanlış olunca biter.

Güncelleme yalnız `i++` olmak zorunda değildir. `i--` ile geriye doğru, `i += 2` ile ikişer, `i *= 2` ile katlanarak ilerlenebilir. Tablo 3.1.11 birkaç başlığın davranışını verir.

Başlık	`i` değerleri	Tur
`for (int i = 1; i <= 5; i++)`	1, 2, 3, 4, 5	5
`for (int i = 10; i >= 1; i -= 2)`	10, 8, 6, 4, 2	5
`for (int i = 1; i <= 100; i *= 2)`	1, 2, 4, 8, 16, 32, 64	7
`for (int i = 10; i <= 1; i++)`	—	0
`for (int i = 1; i != 9; i += 2)`	1, 3, 5, 7	4
`for (int i = 1; i != 10; i += 2)`	1, 3, 5, …, 9, 11, …	sonsuz

Başlık	`i` değerleri	Tur
`for (int i = 0; i <= 10; i--)`	0, −1, −2, …	sonsuz*

Tablo 3.1.11: Farklı `for` başlıklarında gövdenin gördüğü `i` değerleri.

Son iki satır dikkat ister. `i != 10` koşulu tek sayılarla hiçbir zaman yanlış olmaz: `i` 9'dan 11'e atlar ve 10'u görmez. Bu yüzden koşullarda `!=` yerine `<` ya da `<=` tercih edilir; `i <= 10` yazılsaydı döngü 1, 3, 5, 7, 9 ile beş turda biterdi. Son satırda güncelleme yanlış yöndedir: `i` azaldıkça `i <= 10` hep doğru kalır. Yıldızlı "sonsuz" sözcüğü pratik anlamdadır. `int`'in en küçük değeri olan −2.147.483.648'den sonra bir eksiltilince en büyük değere, 2.147.483.647'ye sarar ve koşul yanlış olur. Döngü bu yüzden 2.147.483.649 turdan sonra biter; ekrana yazan bir gövdeyle bu saatler sürer.

3.1.6.3 1'den 10'a toplam

Döngülerin en temel kullanımı bir değeri adım adım biriktirmektir. Aşağıdaki kod 1'den 10'a kadar sayıların toplamını bulur:

```
int toplam = 0;
for (int sayac = 1; sayac <= 10; sayac++) {
    toplam += sayac;
}
System.out.println("Toplam: " + toplam);
```

```
Toplam: 55
```

Tanım 3.1.12

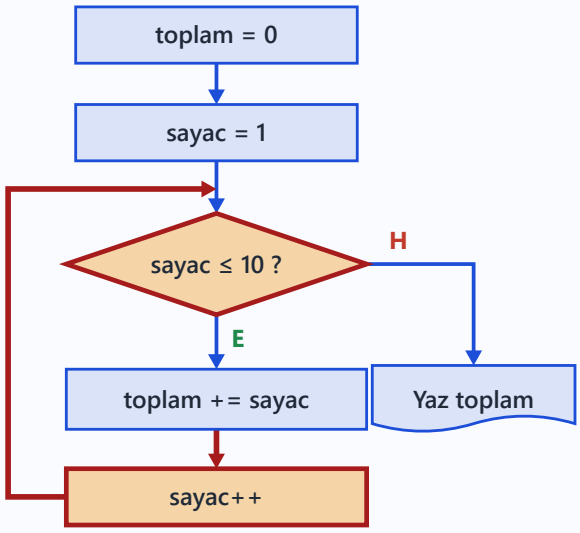
Biriktirici (*accumulator*), döngünün her turunda değer eklenerek sonucu toplayan değişkendir. Toplam için 0 ile, çarpım için 1 ile başlatılır; bu sayılar işlemin etkisiz elemanıdır.

`toplam` 0 ile başlar. Başlangıç adımı `sayac`'ı 1 yapar. Koşul `1 <= 10` doğrudur ve ilk tur çalışır: `toplam += sayac`, yani `toplam = toplam + sayac`, 0 + 1 = 1 verir. Güncelleme `sayac`'ı 2 yapar, koşul yine doğrudur ve ikinci turda `toplam` 1 + 2 = 3 olur. Bu düzen her turda aynı biçimde tekrarlanır (Tablo 3.1.12).

Tur	`sayac`	Koşul `sayac <= 10`	`toplam` (gövdeden sonra)
1	1	doğru	0 + 1 = 1
2	2	doğru	1 + 2 = 3
3	3	doğru	3 + 3 = 6
4	4	doğru	6 + 4 = 10
5	5	doğru	10 + 5 = 15

Tur	`sayac`	Koşul `sayac <= 10`	`toplam` (gövdeden sonra)
6	6	doğru	15 + 6 = 21
7	7	doğru	21 + 7 = 28
8	8	doğru	28 + 8 = 36
9	9	doğru	36 + 9 = 45
10	10	doğru	45 + 10 = 55
—	11	yanlış	55 (döngü biter)

Tablo 3.1.12: 1'den 10'a toplam döngüsünün izlenmesi.



Şekil 3.1.17: Toplam döngüsünün akış diyagramı, ikinci turun koşul sınaması (`sayac = 2`). Kalın çizilen geri dönüş oku güncellemeden koşula giden yoldur.

Gövde tam 10 kez çalışır, koşul 11 kez sınanır. Döngüden sonra `toplam` yazılır. `sayac` ise artık tanımlı değildir: yalnız döngünün içinde yaşar. Sonuç, Gauss'un formülüyle de denetlenebilir: 1'den *n*'ye kadar sayıların toplamı *n(n + 1)/2*'dir ve *n = 10* için 55 verir. Bilinen bir formülle karşılaştırmak, bir döngünün sınırlarını doğrulamanın kolay bir yoludur.

Sık yapılan hata

Biriktiriciyi döngünün içinde başlatmak. `int toplam = 0;` satırı gövdeye taşınırsa her turda `toplam` yeniden 0 olur. Daha da önemlisi, değişken artık gövdenin bloğunda yaşar ve döngüden sonraki `println` satırı `cannot find symbol` hatası verir. Biriktirici her zaman döngüden önce tanımlanır.

3.1.6.4 Aralık toplama

Döngünün sınırları sabit olmak zorunda değildir. Aşağıdaki program kullanıcıdan iki sayı okur ve aradaki tam sayıları toplar:

```
import java.util.Scanner;

public class AralikToplama {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        System.out.print("Başlangıç değeri (a): ");
        int a = klavye.nextInt();
        System.out.print("Bitiş değeri (b): ");
        int b = klavye.nextInt();
        int toplam = 0;
        for (int i = a; i <= b; i++) {
            toplam += i;
        }
        System.out.println("Sayıların toplamı: " + toplam);
    }
}
```

3 ve 7 girildiğinde döngü değişkeni `i`, kullanıcının girdiği `a`'dan başlar ve 3 + 4 + 5 + 6 + 7 = 25 hesaplanır:

```
Başlangıç değeri (a): 3
Bitiş değeri (b): 7
Sayıların toplamı: 25
```

Girdiler ters sırada verilirse sonuç şaşırtıcıdır:

```
Başlangıç değeri (a): 7
Bitiş değeri (b): 3
Sayıların toplamı: 0
```

İlk kontrolde `7 <= 3` yanlıştır. Gövde hiç çalışmaz ve `toplam` 0 kalır. Program çökmez, hata iletisi de vermez; yalnız yanlış bir cevap verir. Çözüm, döngüden önce sırayı düzeltmektir. `a > b` ise iki değişkenin değerleri üçüncü bir değişken yardımıyla yer değiştirilir:

```
if (a > b) {
    int gecici = a;
    a = b;
    b = gecici;
}
```

Bu blok `int toplam = 0;` satırının önüne eklendiğinde 7 ve 3 girdisi de 25 verir. Değişkenlerin yerini değiştirmek için `a = b; b = a;` yazmak yetmez: ilk atamadan sonra `a`'nın eski değeri kaybolur ve iki değişken de 3 olur. `gecici` değişkeni bu değeri saklamak içindir.

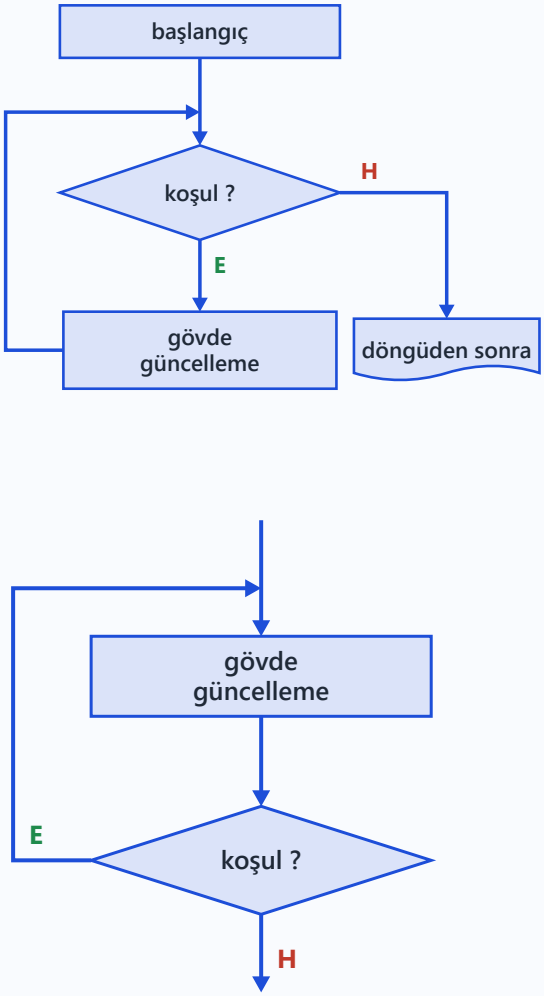
3.1.7 while ve do-while

3.1.7.1 while döngüsü

> **Tanım 3.1.13**
>
> **while döngüsü** (*while loop*), koşulu her turdan önce sınayan ve koşul doğru olduğu sürece gövdeyi tekrarlayan döngüdür. Başlığında yalnız koşul bulunur.

```
int i = 0;
while (i < 5) {
    // Döngü içinde yapılacak işlemler
    i++;
}
```

`while` yalnız koşulu denetler. Başlangıç döngüden önce, güncelleme gövdenin içinde ayrıca yazılır. Koşul gövdeden önce sınandığı için gövde hiç çalışmayabilir. Tekrar sayısı önceden belli değilse `while` doğal seçimdir: “kullanıcı 0 girene kadar oku”, “sayı 0 olana kadar son basamağı at”, “hata kalmayana kadar dene”.



Şekil 3.1.18: Solda `while`, sağda `do-while` akış diyagramı. `while`'da koşul gövdenin önünde, `do-while`'da arkasındadır; iki döngüde de gövde, güncellemeyi kendisi yapar.

Güncelleme programcıya bırakıldığı için `while`'ın en büyük tehlikesi sonsuz döngüdür. Yukarıdaki kodda `i++` unutulursa `i` hep 0 kalır, `0 < 5` hep doğrudur ve program hiç bitmez. Derleyici bunu yakalamaz, çünkü bazı programlar bilerek sonsuz döngüyle yazılır: bir sunucu ya da bir gömülü sistemin ana döngüsü hiç bitmemelidir.

`for` ve `while` birbirine dönüştürülebilir. 1'den 10'a toplam iki biçimde de yazılabilir:

```
int toplam = 0;
int sayac = 1;
while (sayac <= 10) {
    toplam += sayac;
    sayac++;
}
System.out.println("Toplam: " + toplam);
```

```
Toplam: 55
```

İki döngüde de aynı üç parça vardır: başlangıç `sayac = 1`, koşul `sayac <= 10`, güncelleme `sayac++`. `for` bunları tek satırda toplar; `while`'da parçalar dağınıktır ve güncelleme unutulabilir. İki sürüm arasında küçük bir fark daha vardır: `while` sürümünde `sayac` döngüden önce tanımlandığı için döngüden sonra da okunabilir ve değeri 11'dir.

3.1.7.2 Faktöriyel

Bir pozitif tam sayının faktöriyeli, 1'den o sayıya kadar bütün tam sayıların çarpımıdır: 5! = 1 × 2 × 3 × 4 × 5 = 120. Tanım gereği 0! = 1'dir. `while` ile yazılmış hâli şöyledir:

```
int n = 5;
int faktoriyel = 1;
int sayac = 1;
while (sayac <= n) {
    faktoriyel *= sayac;
    sayac++;
}
System.out.println(n + " faktöriyeli: " + faktoriyel);
```

```
5 faktöriyeli: 120
```

Çarpımın başlangıç değeri 1'dir. 0 ile başlansaydı her çarpma 0 verir ve sonuç hep 0 olurdu. Tablo 3.1.13 döngünün her turunu gösterir.

Tur	Koşul	`faktoriyel *= sayac`	`sayac` (turdan sonra)
1	`1 <= 5` doğru	1 × 1 = 1	2
2	`2 <= 5` doğru	1 × 2 = 2	3
3	`3 <= 5` doğru	2 × 3 = 6	4
4	`4 <= 5` doğru	6 × 4 = 24	5
5	`5 <= 5` doğru	24 × 5 = 120	6
—	`6 <= 5` yanlış	120 (döngü biter)	6

Tablo 3.1.13: `n = 5` için faktöriyel döngüsünün izlenmesi.

Faktöriyel çok hızlı büyür ve `int`'in sınırını çabuk aşar. `int` en çok 2.147.483.647'yi tutar; 12! = 479.001.600 sığar, 13! = 6.227.020.800 sığmaz. `n = 13` ile aynı kod 1932053504 yazar. Java tam sayı taşmasında hata vermez, sonuç sessizce yanlıştır. `long` kullanılınca 20!'ye kadar doğru sonuç alınır; 21! `long`'u da taşırır.

3.1.7.3 Üs alma

Bir tabanın tam sayı kuvveti de çarpım biriktirerek hesaplanır: $3^4 = 3 \times 3 \times 3 \times 3 = 81$. Tekrar sayısı (üs) belli olduğu için `for` uygundur:

```
import java.util.Scanner;

public class UsAlma {
    public static void main(String[] args) {
        Scanner tarayici = new Scanner(System.in);
        System.out.print("Taban ve üs girin: ");
        int taban = tarayici.nextInt();
        int us = tarayici.nextInt();
        long sonuc = 1;
        for (int i = 1; i <= us; i++) {
            sonuc = sonuc * taban;
        }
        System.out.println("Sonuç: " + sonuc);
        System.out.println("Math.pow: " + Math.pow(taban, us));
        tarayici.close();
    }
}
```

```
Taban ve üs girin: 3 4
Sonuç: 81
Math.pow: 81.0
```

`sonuc` 1 ile başlar ve döngü `us` kez `taban` ile çarpar. `us = 0` ise döngü hiç dönmez ve sonuç doğru biçimde 1 olur. `sonuc` taşmaya karşı `long` seçilmiştir: 2^{31} `int`'e sığmaz ama `long`'a sığar. `long`'un da sınırı vardır; 2 63 girdisiyle program `Sonuç: -9223372036854775808` yazar.

Java'nın hazır `Math.pow(taban, us)` metodu aynı işi yapar, ama sonucu `double` olarak döndürür. Bu yüzden 81 değil `81.0` yazılır. Kendi döngümüz algoritmanın nasıl çalıştığını gösterir. Gerçek programlarda hazır metotlar tercih edilir: hem denenmiş hem de kesirli üsler gibi döngünün yapamadığı durumları da kapsar.

3.1.7.4 do-while döngüsü

Tanım 3.1.14

do-while döngüsü (*do-while loop*), önce gövdeyi çalıştırıp sonra koşulu sınayan döngüdür. Gövde koşuldan bağımsız olarak en az bir kez çalışır.

```
int i = 0;
do {
    // Döngü içinde yapılacak işlemler
    i++;
} while (i < 5);
```

`do-while`, koşul sağlandığı sürece tekrar eder; tek farkı koşulu turun sonunda sınamasıdır. Yazımda bir ayrıntı vardır: `while (koşul)`'dan sonraki noktalı virgül zorunludur. Unutulursa derleyici `';' expected` hatası verir. `while` döngüsünde ise bu noktalı virgül bir hatadır ve sinsi bir sonsuz döngü doğurabilir.

3.1.7.5 Girdi doğrulama

`do-while`'ın tipik kullanımı kullanıcıdan geçerli bir değer alınana kadar soruyu tekrarlamaktır. Soru en az bir kez sorulmalıdır; bu tam olarak `do-while`'ın yaptığı iştir.

```
import java.util.Scanner;

public class DoWhileGirdi {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        int sayi;
        do {
            System.out.print("1-10 arası bir sayı girin: ");
            sayi = klavye.nextInt();
        } while (sayi < 1 || sayi > 10);
        System.out.println("Kabul edilen sayı: " + sayi);
    }
}
```

Kullanıcı sırayla 15, 0 ve 7 yazsın:

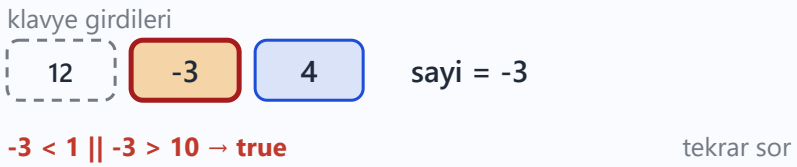
```
1-10 arası bir sayı girin: 15
1-10 arası bir sayı girin: 0
1-10 arası bir sayı girin: 7
Kabul edilen sayı: 7
```

15 ve 0 aralık dışındadır: `sayi < 1 || sayi > 10` koşulu doğrudur ve soru yeniden sorulur. 7 geçerlidir: koşul yanlıştır ve döngüden çıkılır. `sayi` döngüden önce tanımlanmıştır, çünkü hem koşulda hem de döngüden sonra kullanılır. Gövdenin içinde tanımlansaydı `while (sayi < 1 …)` satırı onu göremez ve `cannot find symbol` hatası verirdi: koşul, gövde bloğunun dışındadır.

Tablo 3.1.14 aynı kodu 12, −3 ve 4 girdileriyle izler. Koşuldaki `||`'nin kısa devresi de burada görülür.

Tur	Okunan `sayi`	`sayi < 1`	`sayi > 10`	Koşul	Sonuç
1	12	yanlış	doğru	doğru	soru tekrarlanır
2	−3	doğru	sınanmaz	doğru	soru tekrarlanır
3	4	yanlış	yanlış	yanlış	döngüden çıkılır

Tablo 3.1.14: Girdi doğrulama döngüsünün 12, −3, 4 girdileriyle izlenmesi.



Şekil 3.1.19: Girdi kuyruğu, ikinci tur. 12 okunmuş ve reddedilmiştir; şimdi −3 okunmuştur. `-3 < 1` doğru olduğu için `||` sağ tarafa bakmaz ve soru yeniden sorulur.

Soru toplam üç kez sorulur ve “Kabul edilen sayı: 4” yazılır. Geçerli bir sayı gelene kadar döngü sürer. Kullanıcı sayı yerine “on” gibi bir metin yazarsa `nextInt()` `InputMismatchException` fırlatır. Bu durumu da karşılamak için `hasNextInt()` ile önce girdinin sayı olup olmadığına bakılabilir.

`while` ile `do-while` arasındaki fark, koşul ilk denemede yanlış olduğunda ortaya çıkar:

```
int i = 10;
while (i < 5) {
    System.out.println("while: " + i);
    i++;
}
int j = 10;
do {
    System.out.println("do-while: " + j);
    j++;
} while (j < 5);
System.out.println("i = " + i + ", j = " + j);
```

```
do-while: 10
i = 10, j = 11
```

`i = 10` ile `while` koşulu önce sınar: `10 < 5` yanlıştır ve gövde hiç çalışmaz; `i` 10 kalır. `j = 10` ile `do` bloğuna koşula bakılmadan girilir. “do-while: 10” yazılır ve `j` 11 olur. Koşul ancak şimdi sınanır: `11 < 5` yanlıştır ve döngüden çıkılır. Aynı koşul, farklı sonuç: `while` 0 tur, `do-while` 1 tur çalışmıştır.

Bu farkın gerçek bir sonucu basamak saymada görülür. Bir sayının basamaklarını sayan `while (sayi > 0) { basamak++; sayi /= 10; }` döngüsü 2024 için 4 verir (2024 → 202 → 20 → 2 → 0), ama 0 için 0 verir: koşul ilk denemede yanlıştır. Oysa 0 tek basamaklı bir sayıdır. Döngü `do-while`'a çevrilince gövde en az bir kez çalışır ve 0 için 1 bulunur:

```
int sayi = 0, basamak = 0;
do {
    basamak++;
    sayi /= 10;
} while (sayi != 0);
System.out.println(basamak);
```

Koşul burada `sayi != 0` olarak yazılmıştır. Böylece negatif sayılar da doğru sayılır: −45 için 2 bulunur. `while (sayi > 0)` ile negatif sayılarda döngü hiç dönmez.

3.1.7.6 Üç döngünün karşılaştırması

	`for`	`while`	`do-while`
Koşul ne zaman sınanır?	Her turdan önce	Her turdan önce	Her turdan sonra
En az kaç tur?	0	0	1
Başlangıç / güncelleme	Başlıkta, tek satırda	Döngü dışında / gövdede	Döngü dışında / gövdede
Ne zaman?	Tekrar sayısı belli: “1'den *n*'ye”, “dizinin her öğesi”	Bir koşul değişene kadar: “sayı 0 olana kadar”	İşlem en az bir kez yapılmalı: girdi doğrulama, menü
Söz dizimi	`for (bas; koşul; günc) { }`	`while (koşul) { }`	`do { } while (koşul);`

Tablo 3.1.15: `for`, `while` ve `do-while` döngülerinin karşılaştırması.

Üç döngü birbirine dönüştürülebilir; seçim okunabilirliğe göre yapılır. Bir döngü sayaçla ilerliyorsa `for` yazılır: okuyan kişi başlığa bakarak döngünün kaç kez döneceğini anlar. Döngünün bitişi bir olaya bağlıysa `while` daha açıktır. Gövdenin en az bir kez çalışması gerekiyorsa `do-while` seçilir.

3.1.7.7 Sonsuz döngü ve sınır hataları

Döngü hatalarının çoğu iki gruba girer: hiç bitmeyen döngüler ve bir tur fazla ya da eksik dönen döngüler. İki grubu da derleyici yakalamaz.

Sonsuz döngü, koşulu hiçbir zaman yanlış olmayan döngüdür. En yaygın nedenleri şunlardır: `while`'da güncellemeyi unutmak, güncellemeyi yanlış yönde yapmak (`i--` ile `i <= 10`) ve `!=` koşulunun hedef değeri atlaması. Daha sinsi bir neden, döngü başlığından sonra konan noktalı virgüldür:

```
int i = 0;
while (i < 5); {
    i++;
}
```

Noktalı virgül, gövdesi boş bir döngü kurar. `i` hiç artmaz, `0 < 5` hep doğrudur ve program sessizce donar. Alttaki blok döngüye ait değildir; akış oraya hiç ulaşmaz. Aynı hata `for` başlığında yapılırsa sonsuz döngü oluşmaz ama mantık yine bozulur: `for (int i = 0; i < 3; i++); { System.out.println("merhaba"); }` boş gövdeyi üç kez çalıştırır ve "merhaba"yı yalnız bir kez yazar.

Tanım 3.1.15

Sınır hatası (*off-by-one error*), bir döngünün bir tur fazla ya da bir tur eksik dönmesine yol açan, genellikle başlangıç değerinde ya da koşuldaki karşılaştırma işlecinde yapılan hatadır.

"1'den 10'a kadar" için `i < 10` yazmak 10'u atlar ve 9 tur döner. `i = 0` ile `i <= 10` ise 11 tur döner. Bu hatalar elle izlemeyle kolay bulunur: ilk turda döngü değişkeninin değeri ve son turda değeri ayrı ayrı denetlenir. Bir döngünün kaç tur döndüğü *bitiş − başlangıç + 1* formülüyle de hesaplanabilir (`<=` ile, birer artışta). Tipik bir örnek, her satırı 10 kez "+-" olan bir desendir. Satır 20 karakterdir; iç döngü `j = 0` 'dan başlayıp `j < 21` ile dönerse 21 karakter yazılır ve satır fazladan bir "+" ile biter.

Kayan noktalı sayılarla sayaç tutmak da bir sınır hatası kaynağıdır. `for (double x = 0; x != 1.0; x += 0.1)` döngüsü hiç bitmez: on kez 0,1 eklemek tam olarak 1,0 değil 0,9999999999999999 verir ve `x` 1,0'ı atlar. Döngü sayaçları her zaman tam sayı olmalıdır; ondalık değer gerekiyorsa gövdede sayaçtan hesaplanır: `double x = i * 0.1;` .

3.1.8 İç içe döngüler

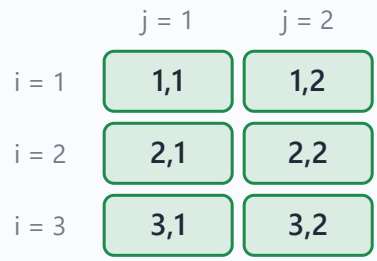
Tanım 3.1.16

İç içe döngü (*nested loop*), bir döngünün gövdesinde bulunan başka bir döngüdür. Dıştaki döngüye **dış döngü**, gövdedekine **iç döngü** denir.

İç içe döngüler iki boyutlu yapılarla çalışırken kullanılır: tablolar, matrisler, ekrana çizilen desenler, bir resmin pikselleri. Benzetmesi saattir. Akrep (dış döngü) bir ilerlediğinde yelkovan (iç döngü) tam bir tur atar. Yelkovan turunu bitirmeden akrep yerinden oynamaz.

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 2; j++) {
        // İç içe döngü gövdesi
    }
}
```

Dış döngünün her turunda iç döngü baştan başlar ve sonuna kadar döner. Bu yüzden gövdenin toplam çalışma sayısı iki döngünün tur sayılarının çarpımıdır: dış 3 tur × iç 2 tur = 6 kez (Şekil 3.1.20).



Şekil 3.1.20: 3 satır, 2 sütunluk ızgara. Her hücre gövdenin bir çalışmasıdır: toplam 6.

Çarpım kuralı, iç içe döngülerin maliyetini de gösterir. Her biri 1.000 tur dönen iki döngü 1.000.000, üç döngü 1.000.000.000 kez gövde çalıştırır. Bir milyar işlem bugünün bilgisayarında birkaç saniye sürebilir. Döngülerin iç içe sayısı arttıkça kod hem karmaşıklaşır hem de yavaşlar. Bir problemin kaç iç içe döngü gerektirdiği, algoritmanın hızını belirleyen başlıca etkendir.

3.1.8.1 İç içe döngünün işleyişi

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 2; j++) {
        System.out.println("i = " + i + ", j = " + j);
    }
}
```

Akış adım adım şöyledir. Dış döngü başlar: `i = 1`, `1 <= 3` doğrudur. İç döngü başlar: `j = 1` ve gövde “i = 1, j = 1” yazar. `j++` ile `j` 2 olur, “i = 1, j = 2” yazılır. `j` 3 olunca `3 <= 2` yanlıştır ve iç döngü tamamlanır. Akış dış döngüye döner: `i++` ile `i` 2 olur, koşul doğrudur ve iç döngü `j = 1`'den **yeniden** başlar. Bu düzen `i = 3` turu da bitene kadar sürer. Son olarak `i` 4 olur, `4 <= 3` yanlıştır ve dış döngü de biter.

Adım	`i`	`j`	Olay	Ekran
1	1	1	iç döngü başlar	i = 1, j = 1
2	1	2		i = 1, j = 2

Adım	`i`	`j`	Olay	Ekran
3	1 → 2	3	`3 <= 2` yanlış: iç döngü biter, `i++`	
4	2	1	iç döngü yeniden başlar	i = 2, j = 1
5	2	2		i = 2, j = 2
6	2 → 3	3	iç döngü biter, `i++`	
7	3	1	iç döngü yeniden başlar	i = 3, j = 1
8	3	2		i = 3, j = 2
9	3 → 4	3	iç döngü biter; `4 <= 3` yanlış: dış döngü biter	

Tablo 3.1.16: İç içe döngünün izlenmesi.

Gövde 3 × 2 = 6 kez çalışır. Dış koşul 4 kez, iç koşul 3 × 3 = 9 kez sınanır: iç döngü her başlayışında iki doğru ve bir yanlış sınama yapar. Çıktıda `i` 'nin yavaş, `j` 'nin hızlı değiştiği görülür: `i` 1'deyken `j` 1 ve 2 değerleri üzerinde gezinir.

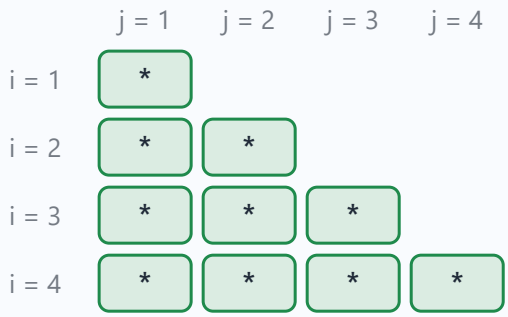
3.1.8.2 Yıldız deseni

İç döngünün sınırı her zaman sabit olmak zorunda değildir. Sınır dış döngünün değişkenine bağlanırsa her satırda farklı sayıda tur dönülür:

```
for (int i = 1; i <= 4; i++) {
    for (int j = 1; j <= i; j++) {
        System.out.print("* ");
    }
    System.out.println();
}
```

```
*
* *
* * *
* * * *
```

İç döngünün koşulu `j <= i` 'dir. Birinci satırda `i = 1` olduğu için iç döngü bir kez döner; ikinci satırda iki, üçüncüde üç, dördüncüde dört kez döner. `print("* ")` satır atlamaz; yıldızlar yan yana dizilir. İç döngü bitince `println()` alt satıra geçer. Bu satır iç döngünün dışında, dış döngünün içindedir: her satırın sonunda bir kez çalışır.



Şekil 3.1.21: Yıldız deseninin son hâli. `i`. satırda `j` = 1 … `i` hücreleri dolmuştur.

Toplam yıldız sayısı 1 + 2 + 3 + 4 = 10'dur. Genel olarak *n* satırlık üçgende *n(n + 1)/2* yıldız yazılır. Bu sayı, iç sınırın dışa bağlı olduğu döngülerin tipik maliyetidir: dikdörtgen ızgaranın n^2'sinin yaklaşık yarısı. İç döngü `j = i`'den başlayıp 4'e kadar dönseydi tur sayıları 4, 3, 2, 1 olur ve toplam yine 10 çıkardı; bu kez desen ters çevrilmiş bir üçgen olurdu.

Sık yapılan hata

`println()`'ı yanlış düzeye koymak. İç döngünün içine yazılırsa her yıldızdan sonra satır atlanır ve on satırlık tek bir sütun çıkar. Dış döngünün dışına yazılırsa bütün yıldızlar tek satıra dizilir. Hangi satırın hangi döngüye ait olduğu yalnız süslü parantezlerle belirlenir; girintinin düzgün tutulması bu ilişkiyi göze görünür kılar.

3.1.8.3 Basamak değeri

```
for (int onlar = 0; onlar <= 4; onlar++) {
    for (int birler = 1; birler <= 9; birler++) {
        int sayi = onlar * 10 + birler;
        System.out.println("Onlar basamağı: " + onlar
                + ", Birler basamağı: " + birler + " --> " + sayi);
    }
}
```

Bu kod iki basamaklı sayıları onlar ve birler basamağından kurar. Kaç satır yazdığını bulmak için iki döngünün tur sayıları çarpılır. Dış döngü `onlar` = 0 … 4 için 5 tur, iç döngü `birler` = 1 … 9 için 9 tur döner: 5 × 9 = 45 satır. İlk ve son satırlar şunlardır:

```
Onlar basamağı: 0, Birler basamağı: 1 --> 1
Onlar basamağı: 0, Birler basamağı: 2 --> 2
Onlar basamağı: 0, Birler basamağı: 3 --> 3
…
Onlar basamağı: 4, Birler basamağı: 8 --> 48
Onlar basamağı: 4, Birler basamağı: 9 --> 49
```

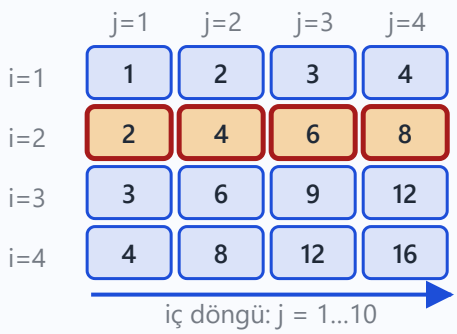
`sayi = onlar * 10 + birler` 1'den 49'a kadar değer alır. `birler` 0'dan başlamadığı için 10, 20, 30 ve 40 hiç yazılmaz; 0 da yazılmaz. 1'den 49'a 49 sayı vardır, bunların 4'ü atlandığı için 45 satır kalır. İki hesap birbirini doğrular.

Kodun yazdırma satırı iki satıra bölünmüştür. Java'da uzun bir dizgi `+` ile parçalara ayrılıp sonraki satırda sürdürülebilir. Tırnak içindeki bir metin ise satır sonunda kesilemez. `"Onlar basamağı:` yazıp satırı bitirmek `unclosed string literal` hatası verir.

3.1.8.4 Çarpım tablosu

```java
for (int i = 1; i <= 10; i++) {
    for (int j = 1; j <= 10; j++) {
        int carpim = i * j;
        System.out.print(i + "x" + j + "=" + carpim + "\t");
    }
    System.out.println(); // Yeni satır
}
```

Dış döngü satırları (`i`), iç döngü sütunları (`j`) gezer: 10 × 10 = 100 çarpım. `print` satır atlamaz; her çarpımın arkasına eklenen `"\t"` bir sekme (*tab*) karakteridir ve sütunları bir sonraki sekme durağına hizalar. Satır bitince `println()` ile alt satıra geçilir.



Şekil 3.1.22: Çarpım tablosunun 4 × 4 köşesi. İşaretli satır `i = 2` turudur: iç döngü `j` = 1, 2, 3, 4 … ile sütunları gezer.

```
1x1=1   1x2=2   1x3=3   1x4=4   1x5=5   1x6=6   …
2x1=2   2x2=4   2x3=6   2x4=8   2x5=10   2x6=12   …
3x1=3   3x2=6   3x3=9   3x4=12   3x5=15   3x6=18   …
…
10x1=10   10x2=20   10x3=30   10x4=40   10x5=50   …
```

Sekme durakları genellikle 8 karakter aralıklıdır. “10x10=100” gibi 9 karakterlik bir hücre bir durağı aşar ve satırın geri kalanını kaydırabilir. Daha düzenli bir tablo için `printf` ile sabit genişlikli alanlar kullanılır: `System.out.printf("%4d", i * j);` her sayıyı 4 karakterlik bir alana sağa yaslayarak yazar. `%-4d` sola yaslar, `%04d` eksik basamakları 0 ile doldurur.

3.1.9 break ve continue

Döngüler normalde koşul yanlış olunca biter. Bazen döngünün ortasında karar vermek gerekir: aranan şey bulunduysa kalan turlar boşunadır; bir öğe işe yaramıyorsa o tur atlanmalıdır. Java bu iki iş için iki deyim sunar.

Tanım 3.1.17

`break`, içinde bulunduğu en yakın döngüyü (ya da `switch`'i) hemen sonlandırır; akış döngüden sonraki deyime geçer. `continue`, o anki turun kalanını atlar ve bir sonraki tura geçer; `for`'da önce güncelleme, sonra koşul çalışır.

3.1.9.1 Döngülerde break

```java
for (int i = 1; i <= 10; i++) {
    if (i == 5) {
        break;
    }
    System.out.println("Döngü adımı: " + i);
}
System.out.println("Döngüden çıkıldı.");
```

```
Döngü adımı: 1
Döngü adımı: 2
Döngü adımı: 3
Döngü adımı: 4
Döngüden çıkıldı.
```

`i` = 1, 2, 3 ve 4 için `i == 5` yanlıştır; `break` atlanır ve adım yazılır. `i = 5`'te koşul doğrudur ve `break` çalışır. Döngü o anda sona erer: 5 yazılmaz, 6–10 hiç denenmez. Akış döngüden sonraki satıra geçer ve "Döngüden çıkıldı." yazılır.



Şekil 3.1.23: Üstte `break`: 1–4 yazılmış, 5'te döngü durmuş, 6–10 hiç işlenmemiştir. Altta `continue`: tek sayılar yazılmış, çift sayılar atlanmış, ama döngü 10'a kadar sürmüştür.

`break`'ten hemen sonra, aynı bloğa yazılan bir deyime hiçbir zaman ulaşılamaz. Java bunu derleme hatası sayar: `for (…) { break; System.out.println(i); }` kodu `error: unreachable statement` verir. `break` bu yüzden hemen her zaman bir `if`'in içinde kullanılır.

3.1.9.2 Asal sayı: bayrak ve break

Bir sayının asal olup olmadığını bulmak `break`'in en bilinen kullanımıdır. Asal sayı, 1'den büyük olan ve yalnız 1'e ve kendisine bölünen tam sayıdır. Program 2'den başlayarak bölen arar; ilk böleni bulduğu anda aramayı keser.

```java
import java.util.Scanner;

public class AsalSayi {
    public static void main(String[] args) {
        Scanner tarayici = new Scanner(System.in);
        System.out.print("Bir sayı girin: ");
        int sayi = tarayici.nextInt();
        boolean asalMi = true;            // bayrak (flag)
        if (sayi < 2) asalMi = false;     // 0 ve 1 asal değildir
        for (int bolen = 2; bolen <= sayi / 2; bolen++) {
            if (sayi % bolen == 0) {      // tam bölünüyor
                asalMi = false;
                break;                    // döngüyü kır
            }
        }
        if (asalMi) {
            System.out.println(sayi + " bir asal sayıdır.");
        } else {
            System.out.println(sayi + " bir asal sayı değildir.");
        }
        tarayici.close();
    }
}
```

```
Bir sayı girin: 21
21 bir asal sayı değildir.

Bir sayı girin: 97
97 bir asal sayıdır.
```

`asalMi` bir **bayraktır** (*flag*): döngü boyunca bir durumun gerçekleşip gerçekleşmediğini taşıyan `boolean` değişken. Önce sayının asal olduğu varsayılır. 0 ve 1 asal olmadığı için `sayi < 2` durumunda bayrak hemen indirilir. Döngü 2'den `sayi / 2`'ye kadar bölenleri dener. Tam bölen bulunursa bayrak indirilir ve `break` ile döngüden çıkılır. Döngüden sonra bayrağa bakılarak sonuç yazılır.

`sayi`	**Denenen bölenler**	**Deneme**	`asalMi`
21	2 (kalan 1), 3 (kalan 0) → `break`	2	`false`
97	2, 3, ..., 48; hiçbiri bölmez	47	`true`
91	2, 3, 4, 5, 6, 7 (kalan 0) → `break`	6	`false`
2	yok (`2 <= 1` yanlış)	0	`true`
1	yok; bayrak `sayi < 2` ile indirilir	0	`false`

Tablo 3.1.17: Asal sayı programının birkaç girdiyle izlenmesi.

97 asaldır ve program bunu ancak 47 denemeden sonra anlar. Daha hızlı bir yol vardır. Bir sayının 1'den ve kendisinden başka böleni varsa bu bölenlerden biri sayının karekökünden küçük ya da eşittir. Çünkü *sayi = a × b* ise *a* ile *b*'nin ikisi birden $\sqrt{sayi}$'den büyük olamaz; çarpımları *sayi*'yi aşardı. Döngü koşulu buna göre değiştirilir:

```
for (int bolen = 2; bolen * bolen <= sayi; bolen++) {
    if (sayi % bolen == 0) {
        asalMi = false;
        break;
    }
}
```

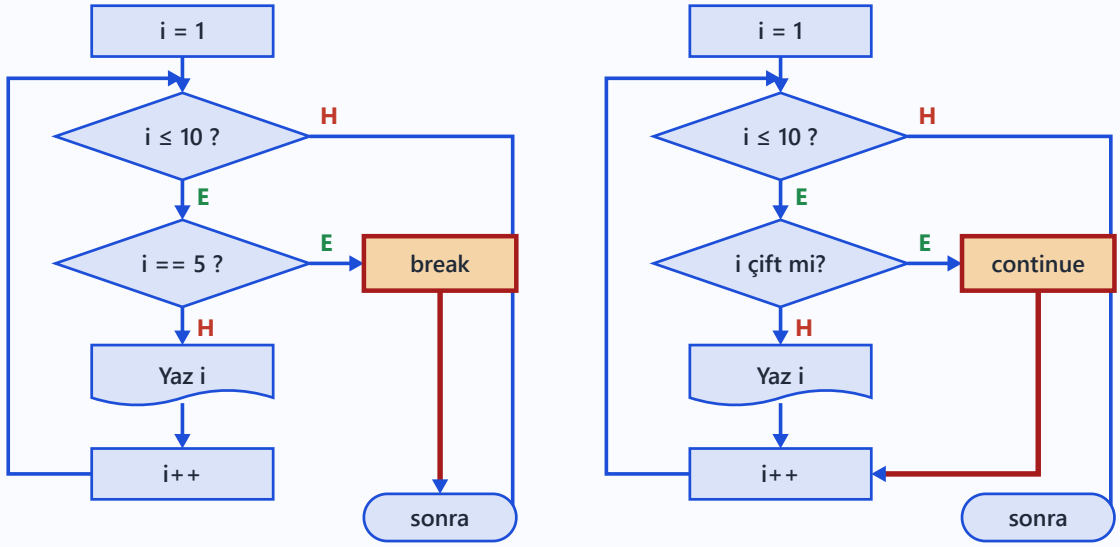
`bolen * bolen <= sayi` koşulu `Math.sqrt` çağırmadan ve kayan nokta hatası taşımadan karekök sınırını uygular. 97 için $\sqrt{97} \approx 9{,}85$'tir; bölenler 2 … 9 denenir ve 47 yerine yalnız 8 deneme yapılır. 1.000.003 gibi yedi basamaklı bir asal sayıda fark çok daha büyüktür: `sayi / 2` sınırıyla 500.000, karekök sınırıyla 999 deneme.

3.1.9.3 Döngülerde continue

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) {
        continue;
    }
    System.out.println("Tek sayı: " + i);
}
```

```
Tek sayı: 1
Tek sayı: 3
Tek sayı: 5
Tek sayı: 7
Tek sayı: 9
```

`i = 1` tektir, koşul yanlıştır ve sayı yazılır. `i = 2` çifttir: `continue` çalışır, gövdenin geri kalanı, yani yazdırma satırı atlanır ve doğrudan `i++` 'ya geçilir. Bu düzen 10'a kadar sürer. `i = 11` olunca `11 <= 10` yanlıştır ve döngü normal biçimde biter. `continue` döngüyü bitirmez, yalnız o turu kısa keser. Aynı iş `continue` olmadan, koşulu tersine çevirerek de yazılabilir: `if (i % 2 != 0) System.out.println(…);` . `continue`, atlanacak durumlar gövdenin başında ayıklanmak istendiğinde, yani gövde uzun olduğunda kodu düzleştirir.



Şekil 3.1.24: Solda `break` : `i == 5` olduğunda akış döngüden tamamen çıkar. Sağda `continue` : `i` çift olduğunda yazdırma atlanır ve güncellemeye (`i++`) geçilir; döngü sürer.

`continue`'nun `while` döngüsünde gizli bir tehlikesi vardır. `for`'da güncelleme başlıktadır ve `continue`'dan sonra da çalışır. `while`'da ise güncelleme gövdenin içindedir ve `continue` onu da atlayabilir:

```
int i = 0;
while (i < 5) {
    if (i == 2) {
        continue;
    }
    System.out.print(i + " ");
    i++;
}
System.out.println("bitti");
```

Bu kod “0 1 ” yazar ve sonra sonsuza kadar döner. `i` 2 olunca `continue` gövdenin kalanını, yani `i++`'yı da atlar. `i` hep 2 kalır, koşul hep doğrudur ve “bitti” hiç yazılmaz. Döngü `for (int i = 0; i < 5; i++)` biçimine çevrilirse güncelleme başlığa taşınır ve çıktı “0 1 3 4 bitti” olur.

3.1.9.4 İç içe döngülerde break ve etiketler

`break` yalnız içinde bulunduğu **en yakın** döngüyü bitirir. İç döngüdeki bir `break` dış döngüye dokunmaz:

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) {
            break;
        }
        System.out.println(i + " " + j);
    }
}
```

```
1 1
2 1
3 1
```

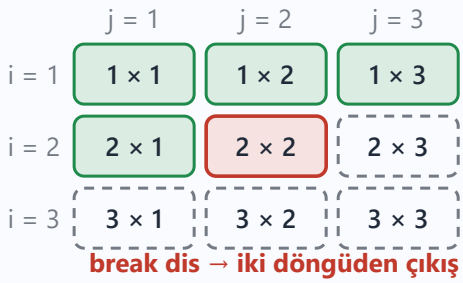
Her `i` için iç döngü `j = 1`'i yazar, `j = 2`'de `break` ile biter. Dış döngü ise sürer ve bir sonraki `i` ile iç döngüyü yeniden başlatır. Sonuçta her `i` için yalnız `j = 1` yazılır.

İki döngüden birden çıkmak gerekiyorsa Java **etiketli** `break` sunar. Dış döngünün önüne bir ad ve iki nokta yazılır; `break` bu adla kullanılır:

```
dis:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (i * j == 4) {
            break dis;
        }
        System.out.println(i + " x " + j);
    }
}
System.out.println("bitti");
```

```
1 x 1
1 x 2
1 x 3
2 x 1
bitti
```

`dis:` etiketi dış döngüyü adlandırır. `i = 1` turunda üç çarpım (1, 2, 3) 4'e eşit değildir ve hepsi yazılır. İç döngü biter, dış döngü `i = 2`'ye geçer. `2 × 1 = 2` yazılır. `i = 2`, `j = 2`'de `2 × 2 = 4` olur ve koşul doğrudur. `break dis;` yalnız iç döngüden değil, etiketli dış döngüden de çıkar. Kalan hücreler (`2 × 3` ve bütün `i = 3` turu) hiç ziyaret edilmez. Akış iki döngünün de sonrasına atlar ve “bitti” yazılır.



Şekil 3.1.25: Etiketli `break`. Yeşil hücreler yazdırılmıştır; `2 × 2` hücresinde koşul doğru olmuş ve `break dis` iki döngüden birden çıkmıştır. Kesikli hücreler hiç ziyaret edilmemiştir.

Etiketsiz `break` olsaydı yalnız iç döngüden çıkılır ve `i = 3` turu da çalışırdı. Bu turdaki çarpımlar 3, 6 ve 9'dur; hiçbiri 4'e eşit olmadığı için üçü de yazılır. Çıktı yedi satır olurdu: 1 x 1, 1 x 2, 1 x 3, 2 x 1, 3 x 1, 3 x 2, 3 x 3 ve ardından "bitti". `continue` da etiketle kullanılabilir. `continue dis;` iç döngünün kalanını bırakır ve dış döngünün bir sonraki turuna geçer:

```
dis:
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (j == 2) {
            continue dis;
        }
        System.out.println(i + " " + j);
    }
}
```

```
1 1
2 1
3 1
```

Etiketler seyrek kullanılır. Akışı birden çok düzey öteye taşıdıkları için kodun izlenmesini zorlaştırırlar. İki döngüden çıkmanın daha yaygın yolu, iç içe döngüyü ayrı bir metoda taşıyıp aranan bulunduğunda `return` ile metottan dönmektir. Java'da `goto` sözcüğü ayrılmış bir sözcüktür ama kullanılamaz; etiketli `break` ve `continue`, dilin izin verdiği tek "atlama" biçimidir.

3.1.10 Sık yapılan hatalar

Kontrol yapılarındaki hataların bir kısmını derleyici yakalar ve iletisiyle yerini gösterir. Bir kısmı ise derlenir, çalışır ve yalnız yanlış sonuç verir. İkinci grup daha tehlikelidir, çünkü programcıyı uyaran hiçbir şey yoktur. Tablo 3.1.18 karar yapılarındaki, Tablo 3.1.19 döngülerdeki tipik hataları toplar. Derleyici iletileri JDK 21 ile alınmıştır.

Hata	Örnek	Sonuç	Doğrusu
`=` ile karşılaştırma	`if (x = 5)`	`incompatible types: int cannot be converted to boolean`	`if (x == 5)`
`if`'ten sonra noktalı virgül	`if (x > 5); { … }`	Derlenir; blok her zaman çalışır	Noktalı virgülü silmek
Parantezsiz blok	`if (k) a(); b(); else c();`	`'else' without 'if'`	Birden çok deyim için `{ }`
Parantezsiz blok, `else`'siz	`if (k) a(); b();`	Derlenir; `b()` her zaman çalışır	`if (k) { a(); b(); }`
Aralığı matematikteki gibi yazmak	`60 <= not < 90`	`bad operand types for binary operator '<'`	`not >= 60 && not < 90`
Zincirde yanlış sıra	`if (not >= 60) … else if (not >= 90) …`	Derlenir; 90 dalına hiç ulaşılmaz	Eşikleri büyükten küçüğe dizmek
`break`'i unutmak	`case 2: … case 3: …`	Derlenir; alttaki `case`'ler de çalışır	Her `case`'i `break` ile bitirmek
`default` olmadan atama	Değişken yalnız `case`'lerde atanıyor	`variable mevsim might not have been initialized`	`default` eklemek
Değişken `case` etiketi	`int b = 2; … case b:`	`constant expression required`	Sabit ya da `final` değişken
Ondalık girişte nokta	Türkçe ayarda `nextDouble()` için `-7.5`	`InputMismatchException`	`-7,5` yazmak ya da `useLocale`

Tablo 3.1.18: Karar yapılarında sık yapılan hatalar.

Hata	Örnek	Sonuç	Doğrusu
Güncellemeyi unutmak	`while (i < 5) { println(i); }`	Sonsuz döngü; derleyici yakalamaz	Gövdede `i++`
Yanlış yönde güncelleme	`for (int i = 0; i <= 10; i--)`	Pratikte sonsuz döngü	`i++`
Döngü başlığında noktalı virgül	`while (i < 5); { i++; }`	Gövdesi boş sonsuz döngü	Noktalı virgülü silmek
Sınır hatası	"1'den 10'a" için `i < 10`	Derlenir; 9 tur döner	`i <= 10`; ilk ve son turu denetlemek
Döngü değişkenini dışarıda kullanmak	`for (int i …) { }` sonra `println(i)`	`cannot find symbol`	`i`'yi döngüden önce tanımlamak
`break`'ten sonra kod	`break; System.out.println(i);`	`unreachable statement`	Satırı `break`'in önüne almak
Yanlış başlangıç değeri	`int faktoriyel = 0;`	Derlenir; sonuç hep 0	Toplam 0, çarpım 1 ile başlar
Biriktiriciyi döngüde sıfırlamak	Gövdede `int toplam = 0;`	Her turda sıfırlanır; döngü dışından erişilemez	Döngüden önce tanımlamak
`while`'da `continue`	`if (i == 2) continue;` ardından `i++`	Sonsuz döngü	`for` kullanmak ya da `i++`'yı önce yapmak
`do-while` sonunda noktalı virgül yok	`} while (i < 3)`	`';' expected`	`} while (i < 3);`

Hata	Örnek	Sonuç	Doğrusu
`int` taşması	`int` ile 13!	Derlenir; 1932053504 yazar	`long`

Tablo 3.1.19: Döngülerde sık yapılan hatalar.

Derlenen hataları bulmanın en güvenilir yolu elle izlemedir. Kod kâğıt üzerinde birkaç küçük girdiyle, her turdan sonra değişkenler bir tabloya yazılarak çalıştırılır. Özellikle üç girdi denenmelidir: döngünün hiç dönmemesi gereken durum (ör. `n = 0`), tek tur dönmesi gereken durum ve bir sınır değeri. Her turdan sonra değişkenleri gösteren bir izleme tablosu, sınır hatalarını kod çalıştırılmadan ortaya çıkarır.

3.1.11 Özet

- **Üç temel yapı:** sıralı işlem, karar ve döngü. Her algoritma bunların birleşimiyle yazılabilir. Sıralı kodda her satır önceki satırların bıraktığı duruma dayanır.
- **Koşul:** sonucu `true` ya da `false` olan `boolean` ifadedir. `=` atama, `==` karşılaştırmadır; Java `if (x = 7)` 'yi derlemez.
- `if` / `else`**:** koşul doğruysa bir blok, değilse diğeri çalışır; iki bloktan en fazla biri çalışır. Parantez yoksa `if` 'e yalnız ilk deyim bağlıdır; girinti anlamsızdır.
- `else if` **zinciri:** koşullar yukarıdan aşağıya denenir, ilk doğru koşulun bloğu çalışır ve zincir biter. Koşulların sırası üst sınırların yerini tutar.
- **İç içe kararlar:** iç koşul yalnız dış koşul doğruysa sınanır. `else`, henüz `else` 'i olmayan en yakın `if` 'e bağlanır. Derin yapılar `&&` ile düzleştirilir.
- **Kısa devre:** `&&` sol taraf `false`, `||` sol taraf `true` ise sağ tarafı hesaplamaz. `b != 0 && a / b > 2` güvenlidir; sağ tarafa yan etkili işlem yazılmaz. `?:` iki değerden birini seçen ifadedir.
- `switch`**:** bir değeri sabit `case` etiketleriyle karşılaştırır. `case` bir giriş noktasıdır; `break` yoksa akış alttaki `case` 'lere düşer. Seçici `byte`, `short`, `char`, `int`, sarmalayıcıları, `String` ya da `enum` olabilir; etiketler sabit ve tektir.
- `switch` **ifadesi (Java 14+):** `case 3, 4, 5 -> …` biçimi; aşağı düşme yoktur, değer üretir, bütün olasılıkları kapsamalıdır (`int` için `default` zorunlu).
- `for`**:** başlangıç, koşul ve güncelleme tek satırda; çalışma sırası başlangıç, sonra her turda koşul → gövde → güncelleme. Başlıkta tanımlanan değişken yalnız döngüde yaşar.
- `while` **ve** `do-while`**:** `while` koşulu önce sınar, 0 tur dönebilir; `do-while` sonra sınar, en az 1 tur döner ve sonunda `;` ister. Girdi doğrulama `do-while` 'ın tipik kullanımıdır.
- **Biriktirici:** toplam 0 ile, çarpım 1 ile başlar ve döngüden önce tanımlanır. `int` 12!'ye, `long` 20!'ye kadar yeter; taşma sessizdir.
- **İç içe döngüler:** iç döngü dış döngünün her turunda baştan başlar; gövde tur sayılarının çarpımı kadar çalışır. İç sınır dışa bağlıysa (`j <= i`) toplam *n(n + 1)/2*'dir.

- `break` **ve** `continue`**:** `break` en yakın döngüyü bitirir, `continue` o turun kalanını atlar. Etiketli `break dis;` dıştaki adlandırılmış döngüden de çıkar. Asal sayı aramasında bayrak ve `break` birlikte kullanılır; sınır `bolen * bolen <= sayi` ile kısaltılır.
- **Hatalar:** sonsuz döngü, sınır hatası, yanlış koşul sırası ve unutulan `break` derlenir ama yanlış çalışır; bunlar elle izlemeyle bulunur.

3.1.12 Kavram listesi

kontrol yapısı *control structure* · koşul *condition* · blok *block* · if deyimi *if statement* · else-if zinciri *else-if chain* · iç içe karar *nested decision* · kısa devre değerlendirme *short-circuit evaluation* · koşul işleci *conditional operator* · switch deyimi *switch statement* · aşağı düşme *fall-through* · switch ifadesi *switch expression* · döngü *loop* · for döngüsü *for loop* · while döngüsü *while loop* · do-while döngüsü *do-while loop* · biriktirici *accumulator* · iç içe döngü *nested loop* · break / continue · tur *iteration* · sonsuz döngü *infinite loop* · sınır hatası *off-by-one error* · bayrak *flag* · etiketli break *labeled break*

3.1.13 Kendini Dene

1. Aşağıdaki zincir `sicaklik` = 18, 15, 0 ve −3 için ne yazar?

```
if (sicaklik < 0) System.out.println("donuyor");
else if (sicaklik < 15) System.out.println("soğuk");
else if (sicaklik < 25) System.out.println("ılık");
else System.out.println("sıcak");
```

Cevap

18: ılık; 15: ılık; 0: soğuk; −3: donuyor. `<` ile yazılan zincirde eşikler küçükten büyüğe dizilir. 15, `15 < 15` yanlış olduğu için “soğuk” halkasını geçer ve “ılık”a düşer. 0 da `0 < 0` yanlış olduğu için “donuyor” olmaz.

2. `int a = 0, b = 5;` için `boolean s = (a != 0) && (b / a > 1) || b > 3;` satırı çalıştırılıyor.

Program sıfıra bölme hatası verir mi? `s` nedir?

Cevap

Hata vermez; `s = true`. `&&`, `||`'den önce gelir; ifade `((a != 0) && (b / a > 1)) || (b > 3)` diye okunur. `a != 0` yanlış olduğu için `b / a` kısa devreyle hiç hesaplanmaz ve parantez içi `false` olur. Ardından `b > 3` doğru çıkar ve sonuç `true`'dur.

3. Aşağıdaki `switch` `k` = 3, 1, 5 ve 7 için `t`'ye hangi değeri verir?

```
int t = 0;
switch (k) {
    case 1: t += 1;
    case 3: t += 3;
    case 5: t += 5; break;
    default: t += 100;
}
```

Cevap

`k = 3`: 3 + 5 = 8. `k = 1`: 1 + 3 + 5 = 9. `k = 5`: 5. `k = 7`: 100. İlk iki `case`'te `break` olmadığı için akış `case 5`'e kadar düşer; `case 5`'teki `break` `default`'a geçişi engeller.

4. `int i; for (i = 3; i <= 30; i += 4) { … }` döngüsü kaç tur döner? Gövde `i`'nin hangi değerlerini görür? Döngüden sonra `i` kaçtır?

Cevap

7 tur: 3, 7, 11, 15, 19, 23, 27. Sonraki değer 31'dir ve `31 <= 30` yanlıştır; döngüden sonra `i = 31`. Tur sayısı $\lfloor (30 - 3) / 4 \rfloor + 1 = 6 + 1 = 7$ diye de hesaplanabilir.

5. `int n = 100, adim = 0; while (n > 1) { n /= 2; adim++; }` kodundan sonra `adim` ve `n` kaçtır?

Cevap

`adim = 6`, `n = 1`. `n`: 100 → 50 → 25 → 12 → 6 → 3 → 1. Tam sayı bölmesi kesri atar: 25 / 2 = 12, 3 / 2 = 1. Döngü, sayıyı 1'e indirmek için gereken yarılama sayısını, yani $\lfloor \log_2 100 \rfloor = 6$'yı bulur.

6. `int x = 0; do { x += 3; } while (x < 0);` ve `int y = 0; while (y < 0) { y += 3; }` kod-

larından sonra `x` ve `y` kaçtır?

Cevap

`x = 3`, `y = 0`. Koşul iki döngüde de ilk sınamada yanlıştır. `do-while` gövdeyi koşula bakmadan bir kez çalıştırır; `while` ise koşulu önce sınar ve gövdeye hiç girmez.

7. Aşağıdaki kodda gövde kaç kez çalışır ve `say` kaç olur?

```
int say = 0;
for (int i = 1; i <= 5; i++)
    for (int j = 1; j <= i; j++)
        if ((i + j) % 2 == 0) say++;
```

Cevap

Gövde 1 + 2 + 3 + 4 + 5 = 15 kez çalışır. Toplamı çift olan çiftler `i` = 1 … 5 için sırasıyla 1, 1, 2, 2, 3 tanedir: `say = 9`. Örneğin `i = 5`'te `j` = 1, 3, 5 için toplam 6, 8, 10'dur.

8. Aşağıdaki kod ne yazar?

```
int s = 0;
for (int i = 1; i <= 10; i++) {
    if (i % 4 == 0) continue;
    if (i > 7) break;
    s += i;
}
System.out.println(s);
```

Cevap

24. 1, 2, 3 eklenir (6). 4, `continue` ile atlanır. 5, 6, 7 eklenir (24). 8, 4'e bölündüğü için yine `continue` ile atlanır; `break` satırına ulaşmaz. 9'da `9 > 7` doğrudur ve `break` döngüyü bitirir; 9 ve 10 eklenmez.

9. Hangisi derlenir? (a) `switch (3.0) { default: }` (b) `final int K = 2; switch (x) { case K:`

`break; }` (c) `if (x = 0) { }` (d) `while (true) { } System.out.println();`

Cevap

Yalnız (b). (a) `selector type double is not allowed`. (b) `K` sabit değerli bir `final` değiş-kendir ve etiket olabilir. (c) `incompatible types: int cannot be converted to boolean`. (d) `while (true)` döngüsünün içinde `break` olmadığı için hiçbir zaman bitmez ve sonraki satır için `unreachable statement` hatası verilir.

3.1.14 İleri okuma

- Oracle, *The Java Tutorials*, "Control Flow Statements": `if`, `switch`, döngüler ve `break` / `continue` için kısa, örnekli resmî anlatım; switch ifadeleri için ayrıca "Switch Expressions" sayfası bulunur.
- A. B. Downey ve C. Mayfield, *Think Java* (2. baskı, 2019), Bölüm 5–6: koşullar, mantıksal işleçler ve döngüler sade bir dille, çok sayıda kısa alıştırmayla işlenir. Kitap çevrim içi ücretsiz okunabilir.
- J. Gosling ve diğ., *The Java Language Specification, Java SE 21 Edition*, Bölüm 14: her deyimin kesin kuralı, aşağı düşme, etiketler ve "ulaşılamaz deyim" denetiminin tam tanımı buradadır.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals* (12. baskı), Kesim 3.8: kontrol akışını C/C++ ile karşılaştırarak anlatır; deneyimli okur için kısa bir başvuru kaynağıdır.

3.1.15 Kaynaklar

1. C. Böhm, G. Jacopini, "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules", *Communications of the ACM*, 9(5), 366–371, 1966.
2. P. Deitel, H. Deitel, *Java: How to Program (Early Objects)*, 11. baskı, Pearson, 2017 (Bölüm 4–5: Control Statements).
3. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019 (Bölüm 5: Conditionals and Logic; Bölüm 6: Loops).
4. J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, G. Bierman, *The Java Language Specification, Java SE 21 Edition*, Oracle, 2023 (Bölüm 14: Blocks, Statements, and Patterns).
5. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson, 2022 (3.8: Control Flow).
6. Oracle, *The Java Tutorials: Control Flow Statements*, docs.oracle.com/javase/tutorial/java/nutsandbolts/flow.html.
7. K. Sierra, B. Bates, T. Gee, *Head First Java*, 3. baskı, O'Reilly, 2022.

KISIM I · BÖLÜM 3.2

Fonksiyonlar

Program büyüdükçe tek bir `main` metodunun içine sığmaz. Aynı hesap farklı yerlerde yeniden gerekir, hatalar uzun kod yığınının içinde kaybolur. Fonksiyon bu iki soruna verilen cevaptır: bir işi bir kez yazar, ona bir ad verir ve istendiği kadar çağırır. Java'da fonksiyonlar her zaman bir sınıfın içinde yazılır ve metot adını alır. Not, bir metodun nasıl tanımlandığını ve çağrıldığını, çağrı sırasında bellekte neler olduğunu ve argümanların neden kopyalandığını adım adım gösterir. Aynı adlı metotlar arasında derleyicinin nasıl seçim yaptığı, hazır Java metotları, değişkenlerin kapsamı ve yaşam süresi ayrıca işlenir. Son kesimler iyi bir metodun nasıl tasarlandığını tartışır. Örneklerdeki bütün programlar JDK 21 ile derlenip çalıştırılmıştır; izleme tablolarındaki değerler ve derleyici iletileri bu çalıştırmalardan alınmıştır.

ÖĞRENME ÇIKTILARI

Tanımlamak ve çağırmak
Erişim belirleyici, dönüş tipi, ad ve parametre listesiyle bir metot yazmak; metodu argümanlarla çağırıp dönüş değerini kullanmak.

Çalışmayı izlemek
Çağrı yığınında çerçevelerin nasıl açılıp kapandığını ve argümanların değer olarak kopyalandığını adım adım göstermek.

Aşırı yükleme
Aynı adlı metotları parametre listeleriyle ayırmak; derleyicinin hangi sürümü seçeceğini argüman tiplerinden önceden söylemek.

Hazır metotlar
`System.out` ve `Math` metotlarını doğru kullanmak; Java kütüphanesinin sık kullanılan sınıflarını ve paketlerini tanımak.

Kapsam
Yerel değişkenlerin, sınıf alanlarının ve iç içe blokların kapsamını ve yaşam süresini açıklamak; gölgelemeyi fark etmek.

Tasarım
Soyutlama, genelleme, DRY ve tek sorumluluk ilkeleriyle iyi metot tasarlamak; bir problemi metotlara bölmek.

3.2.1 Fonksiyon kavramı

Şimdiye kadar yazılan programların hepsi tek bir metottan oluşuyordu: `main`. Kısa programlar için bu yeterlidir. Ama bir öğrenci not sistemi düşünelim. Notlar okunur, ortalama hesaplanır, harf notu bulunur, sonuçlar yazdırılır. Bütün bu adımlar `main`'in içine art arda yazılırsa yüzlerce satırlık tek bir blok ortaya çıkar. Ortalama formülü üç ayrı yerde gerekiyorsa üç kez kopyalanır. Formülde bir hata çıkınca üç yerin de ayrı ayrı düzeltilmesi gerekir. Fonksiyonlar bu düzensizliği ortadan kaldırır.

3.2.1.1 Fonksiyon nedir?

> **Tanım 3.2.1**
>
> **Fonksiyon** (*function*), bir işi yapan, adı olan ve tekrar tekrar kullanılabilen kod parçasıdır. Aldığı parametrelerle ve döndürdüğü değerin tipiyle tanımlanır. Java'da fonksiyonlar her zaman bir sınıfın içinde yazılır ve **metot** (*method*) adını alır.

Fonksiyonu yemek tarifine benzetmek mümkündür. "Çay demle" bir tariftir. Tarif bir kez yazılır ve istendiği kadar uygulanır. Malzemeler tarifin girdileridir; programlamadaki karşılıkları **parametrelerdir**. Ortaya çıkan çay tarifin sonucudur; programlamadaki karşılığı **dönüş değeridir**. Tarifi uygulayan kişi her adımı yeniden düşünmez; yalnızca "çay demle" der ve sonucu alır. Program da bir metodu adıyla çağırır ve sonucunu kullanır.

Fonksiyon aslında yeni bir kavram değildir. Her Java programında bulunan `main` bir metottur. Ekrana yazı yazdırmak için kullanılan `System.out.println(...)` ise Java kütüphanesinde hazır bulunan bir metodun çağrısıdır. Programcı `println`'ın karakterleri ekrana nasıl gönderdiğini bilmez; yalnızca ne yaptığını bilir. Bu not, aynı yapıyı programcının kendi metotlarıyla kurmayı anlatır.

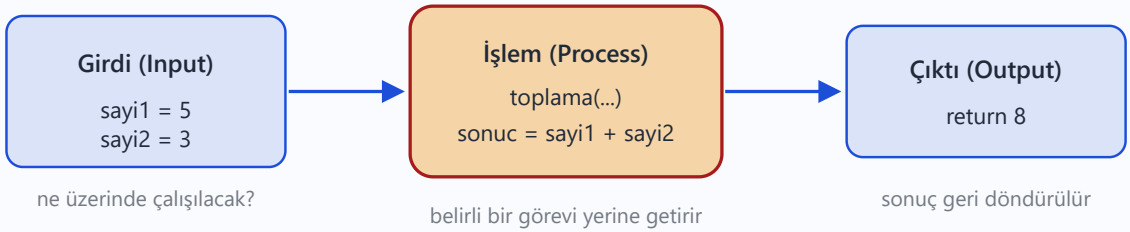
Fonksiyon kullanmanın kazançları şöyle sıralanabilir:

- **Okunabilirlik.** İyi adlandırılmış bir metot çağrısı, ne yapıldığını tek satırda söyler. `ortalamaHesapla(notlar)` satırını okuyan kişi, döngünün ayrıntısına bakmadan ne olduğunu anlar.
- **Kod tekrarının azalması.** Aynı iş bir kez yazılır, her gerektiğinde çağrılır. Bir hata çıktığında tek bir yer düzeltilir.
- **Modülerlik.** Program küçük, yönetilebilir parçalara ayrılır. Her parça ayrı ayrı yazılabilir ve sınanabilir.
- **Düzen.** Parçalar arasındaki bağlantı parametreler ve dönüş değerleriyle sınırlanır. Bir parçanın içi değiştiğinde ötekiler etkilenmez.

"Fonksiyon" ve "metot" sözcükleri bu notta çoğu yerde aynı anlamda kullanılır. Ayrım şudur: fonksiyon genel programlama kavramıdır; metot, bu kavramın Java'daki biçimidir. C gibi dillerde fonksiyonlar herhangi bir sınıfa bağlı olmadan yazılabilir. Java'da ise sınıf dışında metot yazılamaz.

3.2.1.2 Girdi, işlem, çıktı

Her fonksiyon üç aşamada düşünülebilir. **Girdi** aşamasında fonksiyona veri sağlanır; bu veriler parametrelerdir ve fonksiyonun ne üzerinde çalışacağını belirler. **İşlem** aşamasında fonksiyonun gövdesindeki deyimler girdiyi işler. **Çıktı** aşamasında bir sonuç üretilir ve `return` deyimiyle çağırana geri verilir. Şekil 3.2.1 bu üç aşamayı `toplama` fonksiyonu üzerinde gösterir.



Şekil 3.2.1: Girdi, işlem ve çıktı. Girdiler (`sayi1 = 5` , `sayi2 = 3`) `toplama` fonksiyonuna girer. Gövde `sonuc = sayi1 + sayi2` işlemini yapar. `return 8` sonucu çağırana geri verir.

Bu şema matematikteki fonksiyon kavramıyla aynıdır. $f(x) = x^2$ fonksiyonunda x girdidir, kare alma işlemdir, x^2 çıktıdır. f(3) yazıldığında x yerine 3 konur ve sonuç 9 bulunur. Programlamadaki fonksiyon da aynı şekilde işler: girdiyi alır, hesaplar ve sonucu verir.

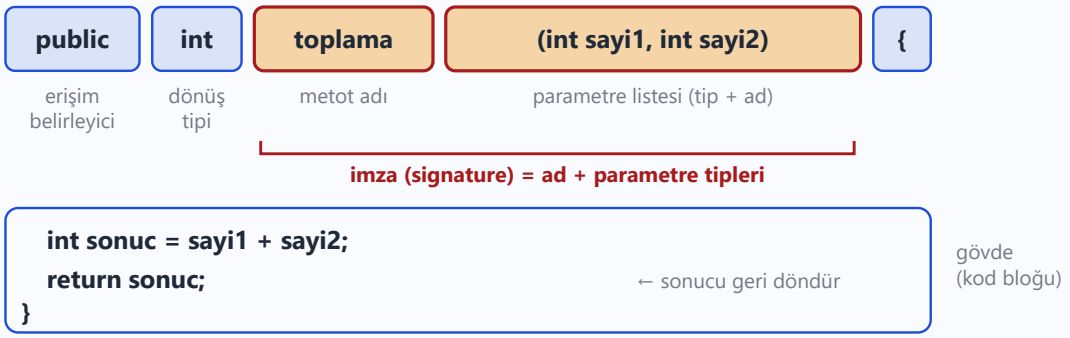
Programlamadaki fonksiyonlar matematiktekinden iki noktada ayrılır. Birincisi, bazı fonksiyonların girdisi yoktur. Ekrana sabit bir selam yazan fonksiyon hiçbir veriye ihtiyaç duymaz. İkincisi, bazı fonksiyonlar değer döndürmez; işleri bir etki bırakmaktır, örneğin ekrana yazmak. Java bu durumu `void` anahtar sözcüğüyle belirtir. “void” İngilizcede “boşluk” demektir; metot geriye hiçbir değer vermez.

3.2.2 Metot tanımı ve çağrısı

Bir metodu kullanmanın iki ayrı adımı vardır. Önce metot **tanımlanır**: adı, girdileri, sonucunun tipi ve gövdesi yazılır. Tanım tek başına hiçbir şey çalıştırmaz; yalnızca tarifi kayda geçirir. Sonra metot **çağrılır**: program akışı metoda gider, gövde çalışır ve akış çağrının yapıldığı yere döner. Bir metot bir kez tanımlanır, ama istendiği kadar çağrılabilir.

3.2.2.1 Metot başlığının parçaları

Bir metot tanımı iki kısımdan oluşur: **başlık** ve **gövde**. Şekil 3.2.2 iki sayıyı toplayan `toplama` metodunun parçalarını ayrı ayrı gösterir.



Şekil 3.2.2: Metot tanımının parçaları. Başlıkta sırasıyla erişim belirleyici (`public`), dönüş tipi (`int`), metot adı (`toplama`) ve parametre listesi bulunur. Ad ile parametre tipleri metodun imzasını oluşturur. Süslü parantezler arasındaki kısım gövdedir.

Parça	Örnekte	Görevi
Erişim belirleyici	`public`	Metodu kimlerin çağırabileceğini söyler. `public` her yerden çağrılabilir demektir.
Dönüş tipi	`int`	Metodun geri vereceği değerin tipi. Değer döndürmeyen metotta `void` yazılır.
Metot adı	`toplama`	Metodun çağrılırken kullanılan adı. Fiil ya da fiil öbeğidir, küçük harfle başlar.
Parametre listesi	`(int sayi1, int sayi2)`	Metodun girdileri. Her parametrenin tipi ve adı ayrı ayrı yazılır; parametreler virgülle ayrılır.
Gövde	`{ ... }`	Metot çağrıldığında çalışacak deyimler.

Tablo 3.2.1: `public int toplama(int sayi1, int sayi2) { ... }` başlığının parçaları.

Erişim belirleyici (*access modifier*) ile dönüş tipi arasına başka **düzenleyiciler** de girebilir. Bu notta en sık görülen düzenleyici `static`'tir; metodun nesne gerektirmeden çağrılabileceğini belirtir. Başlıktaki sıra bellidir: önce düzenleyiciler, sonra dönüş tipi, sonra ad, en son parametre listesi. Erişim belirleyici yazılmayabilir; o zaman metot yalnızca aynı paketteki sınıflardan çağrılabilir. Dönüş tipi ise hiçbir zaman atlanamaz.

Adlandırmada Java geleneği **camelCase** yazımıdır: ilk sözcük küçük harfle başlar, sonraki sözcüklerin ilk harfi büyük yazılır. `ortalamaHesapla`, `asalMi`, `notlariOku` bu kurala uyar. Metot bir iş yaptığı için adı da bir eylem bildirir. `boolean` döndüren metotlarda ad çoğu zaman bir soru gibi okunur: `asalMi(17)`, "17 asal mı?" sorusunun cevabını verir.

Tanım 3.2.2

Metot imzası (*method signature*), metodun adı ile parametre tiplerinin sırasıdır. `toplama` metodunun imzası `toplama(int, int)`'tir. Dönüş tipi, parametre adları ve erişim belirleyici imzaya dahil değildir.

İmza kavramı iki yerde önem kazanır. Birincisi, derleyici bir çağrıyı hangi metotla eşleştireceğine imzaya bakarak karar verir. İkincisi, aynı sınıfta aynı imzalı iki metot bulunamaz. Bu kural, aynı adlı metotların hangi koşulda bir arada yazılabileceğini belirler.

Başlık yazarken yapılan hataların çoğunu derleyici yakalar. Üç tipik örnek ve JDK 21'in verdiği iletiler aşağıdadır. İlkinde ikinci parametrenin tipi unutulmuştur. Matematikte “a, b tam sayı olsun” demek olağandır, ama Java'da her parametre kendi tipini taşır:

```
public class VirgulParam {
    static int topla(int a, b) { return a + b; }
}
```

```
VirgulParam.java:2: error: <identifier> expected
    static int topla(int a, b) { return a + b; }
                            ^
1 error
```

İkincisinde dönüş tipi yazılmamıştır. Dönüş tipi olmayan başlığı derleyici yalnızca sınıfla aynı adlı **yapıcı** (*constructor*) için kabul eder. Yapıcı, nesne oluşturulurken çalışan özel bir metottur. `topla` sınıfın adı olmadığı için hata verilir:

```
public class TipsizMetot {
    static topla(int a, int b) { return a + b; }
}
```

```
TipsizMetot.java:2: error: invalid method declaration; return type required
    static topla(int a, int b) { return a + b; }
           ^
1 error
```

Üçüncüsünde dönüş tipi düzenleyiciden önce yazılmıştır. Derleyici `int`'ten sonra bir ad bekler, `static` ile karşılaşınca iki ayrı hata bildirir:

```
public class StaticSira {
    int static topla(int a, int b) { return a + b; }
}
```

```
StaticSira.java:2: error: <identifier> expected
    int static topla(int a, int b) { return a + b; }
       ^
StaticSira.java:2: error: invalid method declaration; return type required
    int static topla(int a, int b) { return a + b; }
               ^
2 errors
```

Bir başka kural da metotların iç içe tanımlanamamasıdır. Her metot sınıfın gövdesinde, öteki metotların dışında yazılır. `main`'in içine bir metot tanımı konursa derleyici bunu anlamsız bir ifade olarak görür:

```
public class IcIceMetot {
    public static void main(String[] args) {
        static int kare(int n) { return n * n; }
    }
}
```

```
IcIceMetot.java:3: error: illegal start of expression
        static int kare(int n) { return n * n; }
        ^
IcIceMetot.java:5: error: class, interface, enum, or record expected
}
^
2 errors
```

Metotların sınıf içindeki sırası ise önemli değildir. `main` en üstte, yardımcı metotlar altta yazılabilir ya da tersi yapılabilir. Derleyici sınıfın tamamını okuduktan sonra çağrıları eşleştirir.

3.2.2.2 Çağrı, parametre ve argüman

Tanımlanan bir metodu çalıştırmak için metodun adı ve parantez içinde girdileri yazılır. Aşağıdaki program `toplama` metodunu tanımlar ve `main`'den çağırır.

```
public class ToplamaOrnegi {
    public static int toplama(int sayi1, int sayi2) {
        int sonuc = sayi1 + sayi2;
        return sonuc;
    }

    public static void main(String[] args) {
        int sonuc = toplama(5, 3);
        System.out.println("Toplam: " + sonuc);
    }
}
```

```
Toplam: 8
```

Tanım 3.2.3

Metot çağrısı (*method call*), bir metodu argümanlarla çalıştırmaktır. **Parametre** (*parameter*), metot tanımında girdiyi temsil eden değişkendir. **Argüman** (*argument*), çağrı sırasında parametreye verilen gerçek değerdir.

Programda `toplama(5, 3)` ifadesi bir çağrıdır. Çağrıdaki 5 ve 3 argümandır; tanımdaki `sayi1` ve `sayi2` ise parametredir. Ayrımı akılda tutmanın kolay yolu şudur: parametre tanımda yazılır ve bir ad taşır, argüman çağrıda yazılır ve bir değer taşır. Çağrı anında argümanların değerleri sırasıyla parametrelere kopyalanır: `sayi1 = 5`, `sayi2 = 3`.

Çağrı yapılınca program akışı metoda gider. Gövde çalışır, `sonuc` 8 olur ve `return sonuc;` bu değeri geri verir. Akış çağrının yapıldığı yere döner. `toplama(5, 3)` ifadesinin değeri artık 8'dir ve bu değer `main`'deki `sonuc` değişkenine atanır. Son satırdaki `println` da bir metot çağrısıdır; argümanı `"Toplam: " + sonuc` ifadesidir ve ekrana `Toplam: 8` yazılır.

Argümanlar parametrelerle üç bakımdan uyuşmalıdır: **sayı**, **sıra** ve **tip**. İki parametreli bir metoda tek argüman verilirse derleyici çağrıyı reddeder ve beklenen tiplerle verilen tipleri yan yana yazar:

```
public class EksikArguman {
    static int toplama(int sayi1, int sayi2) { return sayi1 + sayi2; }
    public static void main(String[] args) {
        int s = toplama(5);
    }
}
```

```
EksikArguman.java:4: error: method toplama in class EksikArguman cannot be applied to
given types;
        int s = toplama(5);
                ^
  required: int,int
  found:    int
  reason: actual and formal argument lists differ in length
1 error
```

İletideki `required: int,int` satırı metodun beklediği tipleri, `found: int` satırı çağrıda verilenleri gösterir. `reason` satırı ise sorunun argüman sayısında olduğunu söyler. Tip uyuşmazlığında da hata verilir. `toplama(5, 2.5)` çağrısında 2.5 bir `double`'dır ve `int` parametreye bilgi kaybı olmadan sığmaz:

```
TipArguman.java:4: error: incompatible types: possible lossy conversion from double to int
        int s = toplama(5, 2.5);
                        ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
1 error
```

Ters yöndeki dönüşüm ise serbesttir. `double` parametreli bir metoda `int` argüman verilebilir; değer otomatik olarak `double`'a genişletilir. Bu kural, aşırı yüklemede derleyicinin seçimini de belirler.

Argüman yalnızca sabit bir sayı olmak zorunda değildir; değişken ya da herhangi bir ifade olabilir. `kare(a + 1)` çağrısında Java önce `a + 1` ifadesini hesaplar. `a` 3 ise argümanın değeri 4'tür ve bu 4 parametreye kopyalanır. Metot ne `a` değişkenini ne de `a + 1` ifadesini görür; yalnızca 4 değerini alır. Bu kural, argümanların kopyalanması konusunun temelidir.

3.2.2.3 Çağrıyı adım adım izlemek

Bir metot çağrıldığında bellekte neler olduğunu anlamak için iki kavram gerekir: çerçeve ve çağrı yığını.

Tanım 3.2.4

Çerçeve (*stack frame*), bir metot çağrısının parametrelerini ve yerel değişkenlerini tutan bellek kaydıdır. **Çağrı yığını** (*call stack*), çalışmakta olan metotların çerçevelerini üst üste tutan yapıdır. Her çağrı yığının üstüne yeni bir çerçeve ekler; metot bitince çerçevesi yığından kaldırılır.

Yığın, üst üste konan tabaklara benzer. En son konan tabak en üsttedir ve ilk o alınır. Çağrı yığınında da en son çağrılan metodun çerçevesi en üstte durur. Çalışan metot her zaman en üstteki çerçevenin sahibidir. O metot bitince çerçevesi kalkar ve altındaki metot kaldığı yerden devam eder.

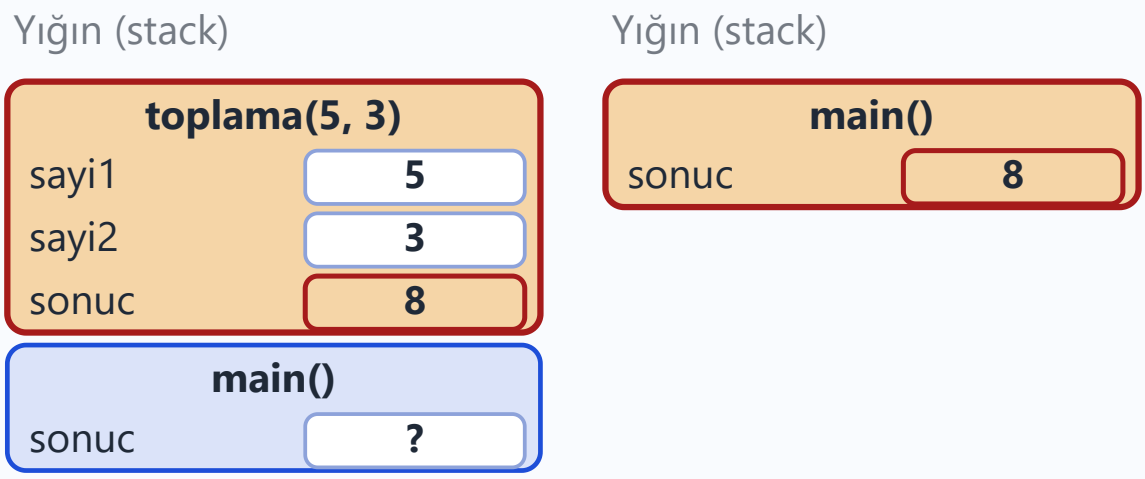
Önceki programın çalışması Tablo 3.2.2'de adım adım verilmiştir. Satır numaraları, sunumdaki izleme slaytının kod düzenine göredir: 1–4. satırlar `toplama`, 6–9. satırlar `main` metodudur.

Adım	Satır	Çağrı yığını	Değişkenler	Çıktı
1	6: `public static void main(String[] args) {`	main()	args=[]	—
2	7: `int sonuc = toplama(5, 3);`	main() → toplama(5, 3)	sayi1=5, sayi2=3	—
3	2: `int sonuc = sayi1 + sayi2;`	main() → toplama(5, 3)	sayi1=5, sayi2=3, sonuc=8	—
4	3: `return sonuc;`	main() → toplama(5, 3)	sayi1=5, sayi2=3, sonuc=8, dönüş=8	—
5	7: `int sonuc = toplama(5, 3);`	main()	args=[], sonuc=8	—
6	8: `System.out.println("Toplam: " + sonuc);`	main()	args=[], sonuc=8	`Toplam: 8`

Tablo 3.2.2: `toplama(5, 3)` çağrısının izlenmesi. Çağrı yığını sütununda en sağdaki çerçeve en üsttedir; değişkenler sütunu o çerçevenin değişkenlerini gösterir.

İlk adımda yığında yalnızca `main`'in çerçevesi vardır. Program her zaman `main` metodundan başlar; `args` komut satırından gelen argümanları tutan dizidir ve burada boştur. İkinci adımda `toplama(5, 3)` çağrılır. Yığının üstüne yeni bir çerçeve açılır ve argümanlar bu çerçevedeki parametrelere kopyalanır.

Üçüncü adımda gövde çalışır: `sonuc = 5 + 3 = 8`. Bu `sonuc`, `toplama`'nın yerel değişkenidir ve `toplama`'nın çerçevesinde durur. Şekil 3.2.3 bu anı gösterir. `main`'in çerçevesinde de `sonuc` adlı bir değişken vardır, ama değeri henüz belli değildir. İki değişken aynı adı taşır, fakat farklı çerçevelerde yaşayan ayrı değişkenlerdir.



Şekil 3.2.3: Çağrı yığınının iki anı. Solda `toplama(5, 3)` çalışıyor: üstteki çerçevede `sayi1 = 5`, `sayi2 = 3`, `sonuc = 8`; alttaki `main` çerçevesinde `sonuc` henüz belirsiz. Sağda metot dönmüş, çerçevesi silinmiş; `main`'deki `sonuc` 8 olmuş.

Dördüncü adımda `return sonuc;` çalışır. 8 değeri çağırana gönderilir, metot biter ve çerçevesi yığından silinir. Silinen çerçeveyle birlikte `sayi1`, `sayi2` ve `toplama`'nın `sonuc`'u da yok olur. Beşinci adımda akış çağrının yapıldığı 7. satıra döner; dönen 8, `main`'in `sonuc` değişkenine atanır. Altıncı adımda `println` çağrılır ve ekrana `Toplam: 8` yazılır. `println` da bir metottur; çalıştığı kısa süre boyunca onun da bir çerçevesi yığına eklenir ve kaldırılır.

Bu izlemeden çıkan sonuç şudur: metot çağrısı, akışın geçici olarak başka bir yere gidip geri gelmesidir. Gidilen yerde yeni bir çalışma alanı açılır, iş bitince bu alan temizlenir. Geriye yalnızca `return` ile gönderilen değer kalır.

3.2.2.4 Değer döndüren ve döndürmeyen metotlar

Aşağıdaki program girdi ve çıktı bakımından farklı üç metot içerir.

```java
public class OrnekFonksiyonlar {
    // Toplama fonksiyonu
    public static int toplama(int sayi1, int sayi2) {
        return sayi1 + sayi2;
    }
    // Kare hesaplama fonksiyonu
    public static double kareHesapla(double sayi) {
        return sayi * sayi;
    }
    // Merhaba dünya fonksiyonu
    public static void merhabaDunya() {
        System.out.println("Merhaba, Dünya!");
    }

    public static void main(String[] args) {
        System.out.println(toplama(2, 7));
        System.out.println(kareHesapla(1.5));
        merhabaDunya();
    }
}
```

```
9
2.25
Merhaba, Dünya!
```

Metot	Girdi	Çıktı	Çağrının kullanımı
`toplama`	2 × `int`	`int`	`println(toplama(2, 7))` → 9
`kareHesapla`	1 × `double`	`double`	`println(kareHesapla(1.5))` → 2.25
`merhabaDunya`	yok	yok (`void`)	`merhabaDunya();` tek başına bir deyim

Tablo 3.2.3: Üç metodun girdileri ve çıktıları.

`toplama` bu kez ara değişken kullanmadan sonucu doğrudan döndürür: `return sayi1 + sayi2;`. Java önce ifadeyi hesaplar, sonra değeri döndürür. Ara değişkenli ve ara değişkensiz iki yazım aynı sonucu verir.

`kareHesapla(1.5)` çağrısı 1,5 × 1,5 = 2,25 değerini döndürür ve ekrana `2.25` yazılır. Java ondalık sayıları kod içinde ve `println` çıktısında nokta ile yazar. `merhabaDunya` metodunun ne parametresi ne de dönüş değeri vardır. Parantezler yine de yazılır: hem tanımda `merhabaDunya()` hem çağrıda `merhabaDunya();`. Parantezsiz `merhabaDunya;` yazmak derleme hatasıdır, çünkü Java parantez görmeden bir adı metot çağrısı olarak tanımaz.

İki tür metot arasındaki asıl fark, çağrının nasıl kullanıldığındadır. Değer döndüren bir metodun çağrısı bir **ifadedir**; değeri vardır. Bu değer bir değişkene atanabilir, başka bir hesaba katılabilir ya da başka bir metoda argüman olarak verilebilir: `println(toplama(2, 7))` ifadesinde `toplama`'nın so-

nucu `println`'ın argümanı olur. `void` bir metodun çağrısının ise değeri yoktur. Bu çağrı yalnızca tek başına bir **deyim** olarak yazılabilir.

Değer döndüren bir metot tek başına da çağrılabilir: `toplama(2, 7);` geçerli bir deyimdir. Ancak dönen değer hiçbir yere atanmadığı için kaybolur. Derleyici bu durumda uyarı vermez. Bu yüzden "metot çağrıldı ama sonuç değişmedi" şikâyetlerinin çoğunda dönen değer hiçbir değişkene atanmamıştır.

3.2.3 return ve void

`return` deyimi metodun sonucunu çağırana verir. Kuralları basittir, ama yeni başlayanların derleyiciyle en çok tartıştığı yerlerden biridir. Bu kesim `return`'ün nasıl çalıştığını, dönen değerin tipinin hangi koşulda kabul edildiğini ve derleyicinin "her yol bir `return` ile bitmeli" kuralını nasıl denetlediğini anlatır.

3.2.3.1 return deyimi

> **Tanım 3.2.5**
>
> **Dönüş değeri** (*return value*), metodun `return` ile çağırana verdiği değerdir. **Dönüş tipi** (*return type*), bu değerin tipidir ve başlıkta metot adından önce yazılır; metot değer döndürmüyorsa dönüş tipi `void`'dir.

`return ifade;` deyiminin iki etkisi vardır. Birincisi, ifadenin değeri hesaplanır ve çağırana verilir. İkincisi, metot **hemen** biter. `return`'den sonra gelen satırlar çalışmaz. Bu özellik, metodun içinde erken çıkış yapmak için kullanılır.

`void` bir metot değer döndürmez, ama yine de erken bitebilir. Bunun için tek başına `return;` yazılır. Aşağıdaki program bölme yapmadan önce böleni denetler. Bölen sıfırsa uyarı yazar ve metottan hemen çıkar.

```
public class Erken {
    static void bolumYazdir(int bolunen, int bolen) {
        if (bolen == 0) {
            System.out.println("Sıfıra bölünemez!");
            return;                      // metottan hemen çık
        }
        System.out.println(bolunen / bolen);
    }
    public static void main(String[] args) {
        bolumYazdir(10, 2);
        bolumYazdir(7, 0);
    }
}
```

```
5
Sıfıra bölünemez!
```

İlk çağrıda `bolen` 2'dir. `if` koşulu yanlıştır, gövdesi atlanır ve `10 / 2` sonucu olan 5 yazılır. İkinci çağrıda `bolen` 0'dır. Koşul doğrudur; uyarı yazılır ve `return;` metodu bitirir. `println(bolunen / bolen)` satırına hiç gelinmez. Bu satıra gelinseydi tam sayı bölmesi sıfıra bölmeye çalışacak ve program `ArithmeticException: / by zero` hatasıyla duracaktı.

Aynı metot `else` ile de yazılabilirdi. Erken `return` kalıbının yararı, hatalı durumları metodun başında ayıklayıp asıl işi girintisiz bırakmasıdır. Denetlenecek koşul sayısı arttıkça bu yazım iç içe `if`'lerden daha okunaklı olur.

`return`'den hemen sonra, aynı blokta bir deyim yazmak derleme hatasıdır. Derleyici bu deyime hiçbir yoldan ulaşılamayacağını görür ve `unreachable statement` iletisini verir:

```
public class Ulasilamaz {
    static int f(int n) {
        return n * 2;
        System.out.println("bitti");
    }
}
```

```
Ulasilamaz.java:4: error: unreachable statement
        System.out.println("bitti");
        ^
1 error
```

3.2.3.2 Dönüş tipiyle uyum

`return`'ün döndürdüğü değerin tipi, başlıkta bildirilen dönüş tipiyle uyumlu olmalıdır. Uyum, atamadaki kuralın aynısıdır: değer, dönüş tipinden bir değişkene atanabiliyorsa döndürülebilir de. Bu yüzden `int` bir değer `double` dönüş tipli bir metottan döndürülebilir; değer otomatik olarak genişletilir. Tersi ise bilgi kaybı demektir ve derleyici kabul etmez:

```
public class Daraltma {
    static int yarim(double x) { return x / 2; }
}
```

```
Daraltma.java:2: error: incompatible types: possible lossy conversion from double to int
    static int yarim(double x) { return x / 2; }
                                        ^
1 error
```

Genişletme serbest olduğu için derlenen ama yanlış sonuç veren bir tuzak doğar. Aşağıdaki iki metodun ikisi de `double` döndürür, ama sonuçları farklıdır.

```
public class Genisletme {
    // int sonuç double dönüş tipine genişler
    static double yarim(int n) {
        return n / 2;
    }
    static double yarimDogru(int n) {
        return n / 2.0;
    }
    public static void main(String[] args) {
        System.out.println(yarim(7));
        System.out.println(yarimDogru(7));
    }
}
```

```
3.0
3.5
```

`yarim(7)` metodunda `n / 2` iki `int` arasında bir bölmedir. Tam sayı bölmesi kesri atar ve sonuç 3 olur. Dönüş tipinin `double` olması bu hesabı değiştirmez; 3 değeri dönerken 3.0'a genişletilir. Yani dönüş tipi, ifadenin nasıl hesaplanacağını değil, hesaplanmış sonucun hangi tipe çevrileceğini belirler. Doğru sonuç için bölmenin kendisi ondalıklı yapılmalıdır: `n / 2.0` ifadesi 3.5 verir.

`void` ile değer döndürme birbirini dışlar. `void` bir metotta `return` ile değer döndürülemez. Değer döndüren bir metotta da değersiz `return;` yazılamaz. Derleyici iki durum için ayrı iletiler verir:

```
public class VoidDeger {
    static void selam() { return "Merhaba"; }
}
```

```
VoidDeger.java:2: error: incompatible types: unexpected return value
    static void selam() { return "Merhaba"; }
                                 ^
1 error
```

```
public class IntBosReturn {
    static int ikiKat(int n) { int s = n * 2; return; }
}
```

```
IntBosReturn.java:2: error: incompatible types: missing return value
    static int ikiKat(int n) { int s = n * 2; return; }
                                                ^
1 error
```

`void` metodun sonucunu kullanmaya çalışmak da hatadır. `void` bir çağrının değeri olmadığı için ne bir değişkene atanabilir ne de `println`'a argüman olarak verilebilir:

```
public class VoidKullan {
    static void selamla() { System.out.println("Selam"); }
    public static void main(String[] args) {
        int x = selamla();
    }
}
```

```
VoidKullan.java:4: error: incompatible types: void cannot be converted to int
        int x = selamla();
                ^
1 error
```

```
VoidPrint.java:4: error: 'void' type not allowed here
        System.out.println(selamla());
                           ^
1 error
```

3.2.3.3 Her yol bir return ile bitmeli

Değer döndüren bir metodun içinde akışın izleyebileceği her yol bir `return` ile bitmelidir. Akış hiçbir `return`'e uğramadan metodun sonundaki süslü paranteze ulaşabiliyorsa derleyici `missing return statement` der. Hatanın en şaşırtıcı biçimi, insana tam görünen koşul zincirlerinde ortaya çıkar:

```
public class EksikReturn {
    static int isaret(int n) {
        if (n > 0) return 1;
        if (n <= 0) return -1;
    }
}
```

```
EksikReturn.java:5: error: missing return statement
    }
    ^
1 error
```

Bir insan için `n > 0` ile `n <= 0` koşulları bütün tam sayıları kapsar; ikisinden biri mutlaka doğrudur. Derleyici ise koşulların değerlerini düşünmez. Yalnızca yapıya bakar: iki `if` vardır ve ikisi de yanlış olabilirmiş gibi değerlendirilir. O zaman akış metodun sonuna ulaşır ve orada `return` yoktur. Aynı hata `if` / `else if` zincirinde de görülür; son dal `else if` olduğu sürece derleyici zincirin bir yerden çıkılabileceğini varsayar.

Çözüm, son koşulu düz bir `else`'e çevirmek ya da metodun sonuna varsayılan bir `return` koymaktır:

```
static int isaret(int n) {
    if (n > 0) return 1;
    return -1;          // n <= 0 olan her durum buraya gelir
}
```

`void` metotlarda böyle bir zorunluluk yoktur. Akış metodun sonuna ulaşınca metot kendiliğinden biter; sonda `return;` yazmak gereksizdir.

3.2.4 static ve nesne üzerinden çağrı

Bu ana kadarki bütün metotların başlığında `static` sözcüğü vardı. Bu sözcük, metodun nasıl çağrılacağını belirler. Nesnelerin ayrıntısına girmeden iki çağrı biçimini ayırt etmek şimdilik yeterlidir.

Tanım 3.2.6

static metot (*static method*), nesneye değil sınıfa ait olan metottur. Çağrılmak için nesne gerekmez. Aynı sınıfın içinden adıyla, başka bir sınıftan `SinifAdi.metot()` biçiminde çağrılır. `static` olmayan metot ise bir **nesne** üzerinden çağrılır: `nesne.metot()`.

`static` metodun en tanıdık örneği `Math` sınıfıdır. Karekök almak için bir `Math` nesnesi oluşturulmaz; doğrudan `Math.sqrt(16)` yazılır. Aynı sınıf içinde sınıf adını yazmaya gerek yoktur:

```
static int toplama(int sayi1, int sayi2) {
    return sayi1 + sayi2;
}
// main içinde, doğrudan:
int sonuc = toplama(5, 3);
```

`static` olmayan bir metodu çağırmak için önce `new` ile o sınıftan bir nesne oluşturulur, sonra metot nesnenin üzerinden çağrılır:

```
public class Hesap {
    int toplama(int sayi1, int sayi2) {
        return sayi1 + sayi2;
    }
    public static void main(String[] args) {
        Hesap hesap = new Hesap();
        int sonuc = hesap.toplama(5, 3);
        System.out.println(sonuc);  // 8
    }
}
```

```
8
```

`Hesap hesap = new Hesap();` satırı `Hesap` sınıfından bir nesne oluşturur ve ona `hesap` adlı bir değişkenle ulaşır. `hesap.toplama(5, 3)` çağrısı metodu bu nesne üzerinden çalıştırır. Nokta işareti "bu nesnenin metodu" anlamına gelir. Sonuç yine 8'dir.

İki biçim karıştırılırsa derleyici hata verir. `main` bir `static` metottur; çalışırken ortada henüz hiçbir nesne yoktur. `static` olmayan bir metodu `main`'den doğrudan çağırmak, "hangi nesnenin metodu?" sorusunu cevapsız bırakır:

```
public class StaticBaglam {
    int toplama(int sayi1, int sayi2) { return sayi1 + sayi2; }
    public static void main(String[] args) {
        int sonuc = toplama(5, 3);
    }
}
```

```
StaticBaglam.java:4: error: non-static method toplama(int,int) cannot be referenced from a
static context
        int sonuc = toplama(5, 3);
                    ^
1 error
```

İletideki "static context" `static` metodun içi demektir. Düzeltmenin iki yolu vardır: ya metodun başlığına `static` eklenir ya da bir nesne oluşturulup metot onun üzerinden çağrılır. Bu bölümdeki metotlar nesnenin durumuna ihtiyaç duymadığı için hepsi `static` yazılır.

`main`'in kendisinin neden `static` olduğu da buradan anlaşılır. Program başladığında JVM (Java Sanal Makinesi) henüz hiçbir nesne oluşturmamıştır. `main`'i bir nesneye ihtiyaç duymadan çağırabilmesi için metodun sınıfa ait olması gerekir.

3.2.5 Değer ile aktarım

Bir metot, kendisine argüman olarak verilen değişkeni değiştirebilir mi? Bu sorunun cevabı Java'nın en çok yanlış anlaşılan konularından biridir. Kısa cevap şudur: hayır. Metot, argümanın kendisini değil kopyasını alır. Ama dizilerde ve nesnelerde kopyalanan şey bir referans olduğu için durum ilk bakışta farklı görünür. Bu kesim iki durumu ayrı ayrı izler.

Tanım 3.2.7

Değer ile aktarım (*pass by value*), çağrı sırasında argümanın değerinin kopyalanıp parametreye verilmesidir. Java bütün argümanları değer ile aktarır. Argüman bir dizi ya da nesne ise kopyalanan değer, o dizinin ya da nesnenin yerini gösteren referanstır.

3.2.5.1 İlkel tipte argüman

Aşağıdaki programda hem `main`'in hem `arttir`'ın `sayi` adlı bir değişkeni vardır. `arttir`, kendi `sayi`'sına 10 ekler.

```
static void arttir(int sayi) {
  sayi = sayi + 10;
  System.out.println("arttir: " + sayi);
}

public static void main(String[] args) {
  int sayi = 5;
  arttir(sayi);
  System.out.println("main: " + sayi);
}
```

```
arttir: 15
main: 5
```

Program `arttir: 15` ve `main: 5` yazar. `main`'deki `sayi` değişmemiştir. Nedenini Tablo 3.2.4 adım adım gösterir.

Adım	Satır	Çağrı yığını	Değişkenler	Çıktı
1	7: `int sayi = 5;`	main()	sayi=5	—
2	8: `arttir(sayi);`	main() → arttir(5)	sayi=5	—
3	2: `sayi = sayi + 10;`	main() → arttir(5)	sayi=15	—
4	3: `System.out.println("arttir: " + sayi);`	main() → arttir(5)	sayi=15	`arttir: 15`
5	9: `System.out.println("main: " + sayi);`	main()	sayi=5	`main: 5`

Tablo 3.2.4: `arttir(sayi)` çağrısının izlenmesi. Değişkenler sütunu en üstteki çerçeveyi gösterir.

İkinci adımda `arttir(sayi)` çağrılır. Java argüman olan `sayi` değişkeninin değerini okur: 5. Yeni açılan çerçevede `arttir`'ın parametresi için ayrı bir kutu ayrılır ve 5 bu kutuya kopyalanır. Artık bellekte iki ayrı `sayi` vardır: biri `main`'in çerçevesinde, biri `arttir`'ın çerçevesinde. Adlarının aynı olması önemsizdir; Java her çerçevenin değişkenlerini ayrı tutar.



Şekil 3.2.4: Değer ile aktarım. Solda `arttir` çalışıyor: üstteki çerçevede kopya 15 olmuş, alttaki `main` çerçeve-

sinde `sayi` hâlâ 5. Sağda metot bitmiş; kopya çerçeveyle birlikte silinmiş, `main`'deki 5 olduğu gibi kalmış.

Üçüncü adımda `sayi = sayi + 10;` çalışır. Bu satır en üstteki çerçevede, yani `arttir`'ın içinde çalıştığı için yalnızca kopyayı değiştirir: kopya 15 olur. Dördüncü adımda `arttir: 15` yazılır. Beşinci adımda metot biter; çerçevesi ve onunla birlikte kopya silinir. `main`'e dönüldüğünde `main`'in `sayi`'sı hiç dokunulmamış hâliyle 5'tir.

Metodun yaptığı değişikliği `main`'e taşımanın tek yolu `return`'dür. `arttir` yeni değeri döndürecek biçimde yazılır ve çağıran bu değeri kendi değişkenine atar:

```
static int arttir(int sayi) {
    return sayi + 10;
}
// main içinde:
sayi = arttir(sayi);   // main'deki sayi artık 15
```

Burada da metot `main`'in değişkenine dokunmaz. Değişikliği yapan, atamayı yazan `main`'in kendisidir. Bu düzen programı güvenli kılar: bir metot çağrıldığında, çağıranın yerel değişkenlerinin değişmeyeceği kesindir.

3.2.5.2 Dizi argümanı: referansın kopyası

Dizi parametreli metotlarda durum farklı görünür. Önce dizinin bellekte nerede durduğunu bilmek gerekir. Java'da bellek iki bölgeye ayrılır. Çerçeveler ve yerel değişkenler **yığında** (*stack*) durur. Diziler ve nesneler ise **heap** adı verilen ayrı bir bölgede tutulur. Bir dizi değişkeni dizinin kendisini değil, dizinin heap'teki yerini gösteren bir **referansı** (*reference*) tutar.

Aşağıdaki metot bir dizi ve bir `int` alır. Diziyi ve tam sayıyı değiştirmeye çalışır, sonra parametresine yepyeni bir dizi atar.

```
static void degistir(int[] d, int x) {
    d[0] = 50;
    x = 50;
    d = new int[2];
    d[1] = 50;
}

public static void main(String[] args) {
    int[] a = {1, 2};
    int x = 1;
    degistir(a, x);
    System.out.println(a[0] + " " + a[1] + " " + x);
}
```

```
50 2 1
```

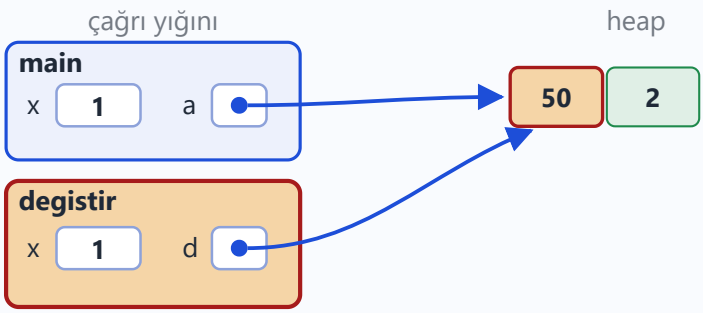
Program `50 2 1` yazar: dizinin ilk öğesi değişmiş, ikincisi değişmemiş, `x` ise hiç değişmemiştir. Tablo 3.2.5 her adımda iki dizinin ve iki `x`'in durumunu gösterir. Tabloda `main`'de oluşturulan dizi A, metodun içinde oluşturulan yeni dizi B diye adlandırılmıştır.

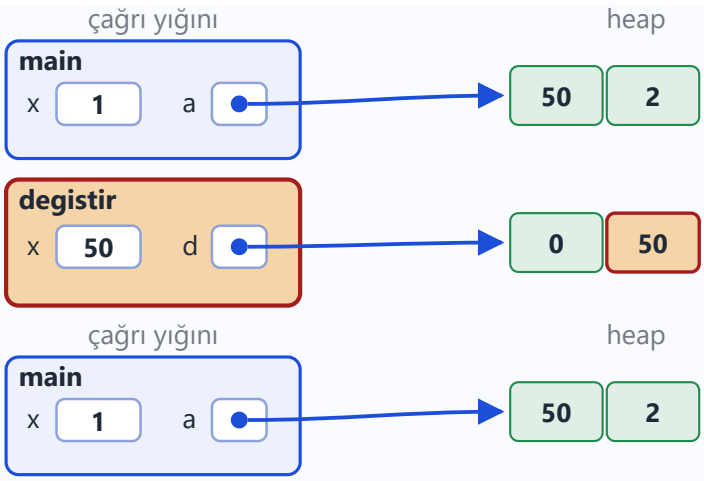
Adım	Satır	`a` → dizi	`d` → dizi	`x` (main)	`x` (degistir)
1	9: `int[] a = {1, 2};`	A {1, 2}	—	—	—
2	10: `int x = 1;`	A {1, 2}	—	1	—
3	1: `static void degistir(int[] d, int x) {`	A {1, 2}	A {1, 2}	1	1
4	2: `d[0] = 50;`	A {50, 2}	A {50, 2}	1	1
5	3: `x = 50;`	A {50, 2}	A {50, 2}	1	50
6	4: `d = new int[2];`	A {50, 2}	B {0, 0}	1	50
7	5: `d[1] = 50;`	A {50, 2}	B {0, 50}	1	50
8	11: `degistir(a, x);`	A {50, 2}	—	1	—
9	12: `System.out.println(a[0] + " " + a[1] + " " + x);`	A {50, 2}	—	1	—

Tablo 3.2.5: `degistir(a, x)` çağrısının izlenmesi. "A {50, 2}" yazımı, değişkenin A dizisini gösterdiğini ve dizinin o anki içeriğini belirtir.

Birinci adımda `int[] a = {1, 2};` heap'te iki öğeli bir dizi oluşturur; `a` bu dizinin referansını tutar. İkinci adımda `x = 1` olur. İlkel bir değer olduğu için 1 doğrudan `main`'in çerçevesinde saklanır.

Üçüncü adımda `degistir(a, x)` çağrılır ve iki argüman da değer ile aktarılır. `d`'ye `a`'daki referansın kopyası verilir; `degistir`'in `x`'ine 1'in kopyası verilir. Referans kopyalanmıştır, ama dizi kopyalanmamıştır. `a` ve `d` artık heap'teki aynı A dizisini gösterir. Şekil 3.2.5'in ilk parçası bu anı çizer: iki ayrı çerçevedeki iki ok aynı diziye uzanır.





Şekil 3.2.5: Referansın kopyası. Üstte `d[0] = 50` sonrası: `a` ve `d` aynı diziyi gösterir, değişiklik ortak diziye yazılır. Ortada `d = new int[2]` ve `d[1] = 50` sonrası: `d` artık yeni diziyi gösterir, bağ kopmuştur. Altta metot bitmiştir; yeni diziyi gösteren referans kalmamıştır.

Dördüncü adımda `d[0] = 50;` çalışır. Bu atama `d` değişkenini değil, `d`'nin gösterdiği dizinin ilk hücresini değiştirir. `d` ve `a` aynı diziyi gösterdiği için değişiklik `main`'den de görülür. Beşinci adımda `x = 50;` yalnızca `degistir`'in kopyasını değiştirir; `main`'in `x`'i 1 kalır.

Altıncı adımda `d = new int[2];` heap'te yeni bir B dizisi oluşturur ve `d`'ye bu dizinin referansını atar. Bu atama `d`'nin kendisini değiştirir. `d` bir kopya olduğu için `a` etkilenmez; `a` hâlâ A'yı gösterir. `d` ile `a` arasındaki bağ bu satırda kopar. Yedinci adımdaki `d[1] = 50;` yeni diziye yazar; A dizisi değişmez.

Sekizinci adımda metot biter ve çerçevesi silinir. B dizisini gösteren tek referans `d` idi; artık hiçbir değişken B'yi göstermez. Böyle bir dizi programdan bir daha erişilemez. JVM'in **çöp toplayıcısı** (*garbage collector*) bu tür nesnelerin belleğini zamanı gelince geri alır. Dokuzuncu adımda ekrana `a[0]`, `a[1]` ve `x` yazılır: `50 2 1`.

Bu örnekten çıkan kural şudur. Java her şeyi değer ile aktarır; referans da kopyalanan bir değerdir. Metot, aldığı referansın gösterdiği diziyi değiştirebilir, çünkü dizi ortaktır. Ama parametreye yeni bir dizi atamak çağıranın değişkenini değiştirmez, çünkü parametre yalnızca bir kopyadır. Bazı kaynaklar Java'nın nesneleri “referans ile aktardığını” yazar. Bu ifade yanlıştır: gerçek referans ile aktarımda `d = new int[2]` satırı `main`'deki `a`'yı da değiştirirdi.

3.2.5.3 Takas tuzağı

Değer ile aktarımın en bilinen sonucu, iki değişkenin değerini değiştiren bir metodun Java'da yazılamamasıdır. Aşağıdaki programda `yerDegistir` metodunun iki sürümü vardır. İlki iki `int` alır, ikincisi bir dizi ve iki indis alır.

```
import java.util.Arrays;

public class Takas {
    static void yerDegistir(int x, int y) {
        int gecici = x;
        x = y;
        y = gecici;
    }
    static void yerDegistir(int[] d, int i, int j) {
        int gecici = d[i];
        d[i] = d[j];
        d[j] = gecici;
    }
    public static void main(String[] args) {
        int[] dizi = {4, 7, 9};
        yerDegistir(dizi[0], dizi[2]);
        System.out.println(Arrays.toString(dizi));
        yerDegistir(dizi, 0, 2);
        System.out.println(Arrays.toString(dizi));
    }
}
```

```
[4, 7, 9]
[9, 7, 4]
```

İlk çağrı `yerDegistir(dizi[0], dizi[2])` 'dir. Argümanlar `dizi[0]` ve `dizi[2]` ifadelerinin değerleridir, yani 4 ve 9. Metot bu iki kopyayı kendi içinde takas eder: `x` 9, `y` 4 olur. Metot bitince kopyalar silinir ve dizi hiç değişmeden kalır: `[4, 7, 9]`.

İkinci çağrı diziyi ve indisleri alır. Bu kez metot, referansın gösterdiği dizinin hücrelerine yazar: `d[0]` 9, `d[2]` 4 olur. Dizi ortak olduğu için değişiklik `main` 'den görülür: `[9, 7, 4]`. `Arrays.toString`, bir diziyi köşeli parantez içinde yazdıran hazır bir metottur.

Değer ile aktarımın davranışı Değer ile Aktarım Laboratuvarı simülatöründe de adım adım izlenebilir. Simülatör ilkel değer, dizi ve nesne geçirmeyi çağrı yığını ve heap üzerinde ayrı ayrı çizer.

3.2.6 Metot aşırı yükleme

Bazen aynı iş farklı tipte ya da farklı sayıda girdiyle yapılmak istenir. İki tam sayıyı, iki ondalıklı sayıyı ve üç tam sayıyı toplayan metotlar düşünelim. Her birine ayrı bir ad vermek mümkündür: `toplaIkiInt`, `toplaIkiDouble`, `toplaUcInt`. Ama bu adlar kullanımı zorlaştırır; programcı her seferinde doğru adı hatırlamak zorunda kalır. Java aynı adın birden çok metot için kullanılmasına izin verir.

3.2.6.1 Aynı ad, farklı parametre listesi

Tanım 3.2.8

Aşırı yükleme (*method overloading*), aynı sınıfta aynı adlı birden çok metot tanımlamaktır. Metotların parametre listeleri farklı olmalıdır: parametre sayısı, tipi ya da sırası. Başka bir deyişle, aşırı yüklenen metotların imzaları farklıdır.

```
public class AsiriYukleme {
    static int topla(int a, int b) {
        return a + b;
    }
    static double topla(double a, double b) {
        return a + b;
    }
    static int topla(int a, int b, int c) {
        return a + b + c;
    }

    public static void main(String[] args) {
        System.out.println(topla(2, 3));
        System.out.println(topla(2.5, 1.2));
        System.out.println(topla(1, 2, 3));
    }
}
```

```
5
3.7
6
```

Sınıfta üç `topla` metodu vardır. İmzaları `topla(int, int)`, `topla(double, double)` ve `topla(int, int, int)`'tir; üçü de birbirinden farklıdır. `topla(2, 3)` çağrısı iki `int` argüman taşır ve ilk metodu çalıştırır: 5. `topla(2.5, 1.2)` iki `double` taşır ve ikinci metodu çalıştırır: 3.7. `topla(1, 2, 3)` üç argüman taşır ve üçüncü metodu çalıştırır: 6. Çıktıdaki `5` ile `3.7`'nin farklı biçimi de seçilen metodu ele verir; `int` döndüren metodun sonucu ondalık nokta olmadan yazılır.

Hangi metodun çalışacağına **derleyici** karar verir. Karar, program çalışmadan önce, derleme zamanında ve yalnızca argümanların tiplerine bakılarak verilir. Derlenmiş programda her çağrı belirli bir metoda bağlanmış durumdadır.

Java kütüphanesi aşırı yüklemeyi yoğun biçimde kullanır. `println` tek bir metot değildir: `println(int)`, `println(double)`, `println(String)`, `println(char)`, `println(boolean)` ve parametresiz `println()` ayrı ayrı tanımlanmıştır. Programcı hepsini tek bir ad olarak görür ve hangi tipi

yazdırırsa yazdırsın aynı adı kullanır. `Math.abs` ve `Math.max` da `int`, `long`, `float` ve `double` için ayrı ayrı aşırı yüklenmiştir.

Aşırı yüklemenin ölçütü imzadır ve dönüş tipi imzaya dahil değildir. Bu yüzden yalnızca dönüş tipi farklı olan iki metot aynı sınıfta bulunamaz:

```
public class DonusTipi {
    static int f(int a) { return a; }
    static double f(int a) { return a; }
}
```

```
DonusTipi.java:3: error: method f(int) is already defined in class DonusTipi
    static double f(int a) { return a; }
                  ^
1 error
```

Kuralın gerekçesi açıktır. `f(5);` gibi dönen değeri kullanmayan bir çağrıda derleyici iki metottan hangisini seçeceğini bilemezdi. Parametre adlarının farklı olması da yetmez: `topla(int a, int b)` ile `topla(int x, int y)` aynı imzayı taşır.

3.2.6.2 Derleyici hangi sürümü seçer?

Argümanların tipleri parametrelerle birebir aynıysa seçim kolaydır. Zor olan, tam eşleşme bulunmadığında ne olduğudur. Derleyici bu durumda argümanları genişleterek uyan metotları arar. Java'daki genişletme dönüşümleri şu sırayı izler:

> `byte → short → int → long → float → double` ve `char → int`

Bir tip, bu zincirde kendisinden sonra gelen her tipe otomatik olarak genişleyebilir. Ters yöndeki daraltma dönüşümleri ise hiçbir zaman otomatik yapılmaz. Seçim üç adımda düşünülebilir:

1. Parametre sayısı argüman sayısına eşit olmayan metotlar elenir.
2. Kalan metotlardan, her argümanın ya aynı tipte olduğu ya da genişletmeyle uyduğu metotlar **uygun** sayılır.
3. Birden çok uygun metot varsa derleyici **en özel** olanı seçer. Bir metodun parametre tiplerinin hepsi ötekinin parametre tiplerine genişleyebiliyorsa o metot daha özeldir. `(int, int)` sürümü `(double, double)`'dan daha özeldir, çünkü `int` `double`'a genişler.

Aşağıdaki program aynı üç metodu dört farklı çağrıyla dener. Çıktısı ve izlemesi aşağıdadır.

```
static int topla(int a, int b) {
    return a + b;
}
static double topla(double a, double b) {
    return a + b;
}
static int topla(int a, int b, int c) {
    return a + b + c;
}
public static void main(String[] args) {
    System.out.println(topla(4, 6));
    System.out.println(topla(4, 2.5));
    System.out.println(topla(7, 1, 2));
    System.out.println(topla('A', 1));
}
```

```
10
6.5
10
66
```

Adım	Satır	Çağrı yığını	Değişkenler	Çıktı
1	11: `System.out.println(topla(4, 6));`	main()	—	—
2	2: `return a + b;`	main() → topla(int, int)	a=4, b=6	—
3	11: `System.out.println(topla(4, 6));`	main()	—	`10`
4	12: `System.out.println(topla(4, 2.5));`	main()	—	—
5	5: `return a + b;`	main() → topla(double, double)	a=4.0, b=2.5	—
6	12: `System.out.println(topla(4, 2.5));`	main()	—	`6.5`
7	13: `System.out.println(topla(7, 1, 2));`	main()	—	—
8	8: `return a + b + c;`	main() → topla(int, int, int)	a=7, b=1, c=2	—
9	13: `System.out.println(topla(7, 1, 2));`	main()	—	`10`
10	14: `System.out.println(topla('A', 1));`	main()	—	—
11	2: `return a + b;`	main() → topla(int, int)	a=65, b=1	—
12	14: `System.out.println(topla('A', 1));`	main()	—	`66`

Tablo 3.2.6: Dört çağrıda seçilen `topla` sürümü. Çağrı yığını sütunu, seçilen sürümün imzasını gösterir.

Birinci çağrı `topla(4, 6)`'dır; argümanlar `(int, int)`'tir. Üç parametreli sürüm sayı yüzünden elenir. `(int, int)` tam eşleşmedir; `(double, double)` da genişletmeyle uyar. En özel olan `(int, int)` seçilir ve 10 yazılır. Şekil 3.2.6'nın ilk parçası bu seçimi gösterir.



Şekil 3.2.6: Derleyicinin aday metotları değerlendirmesi. Üstte `topla(4, 6)` : `(int, int)` tam eşleşir. Ortada `topla(4, 2.5)` : 2.5 `int` 'e daralamaz, `(int, int)` elenir; 4 genişletilir ve `(double, double)` seçilir. Altta `topla('A', 1)` : `char` iki sürüme de genişleyebilir; en özel olan `(int, int)` seçilir.

İkinci çağrı `topla(4, 2.5)` 'tir; argümanlar `(int, double)` 'dır. `(int, int)` sürümü uymaz, çünkü 2.5 `int` 'e örtük olarak dönüşemez. `(double, double)` uyar: 4 değeri 4.0'a genişletilir. Metot `a = 4.0` , `b = 2.5` ile çalışır ve 6.5 döner.

Üçüncü çağrı `topla(7, 1, 2)` üç argüman taşır. Yalnızca üç parametreli sürüm uyar ve 10 döner. Çıktıda ilk ve üçüncü satırın ikisi de 10'dur, ama farklı metotlardan gelmiştir.

Dördüncü çağrı `topla('A', 1)` 'dir; argümanlar `(char, int)` 'tir. `char` tipi Java'da karakterin sayısal kodunu tutar ve `int` 'e de `double` 'a da genişleyebilir. İki sürüm de uygundur; derleyici en özel olanı, `(int, int)` 'i seçer. `'A'` karakterinin kodu 65 olduğu için metot `a = 65` , `b = 1` ile çalışır ve 66 döner. Ekranda bir harf değil, 66 sayısı görülür.

Genişletme zincirinin sonuçları bazen şaşırtıcıdır. Aşağıdaki programda `yaz` metodunun `int` , `long` ve `double` sürümleri vardır; her sürüm farklı bir harf yazar.

```
public class Yaz {
    static void yaz(int a) {
        System.out.print("I");
    }
    static void yaz(long a) {
        System.out.print("L");
    }
    static void yaz(double a) {
        System.out.print("D");
    }
    public static void main(String[] args) {
        yaz(5); yaz(5L);
        yaz(5.0f); yaz('A');
        System.out.println();
        byte b = 3; short s = 4;
        yaz(b); yaz(s); yaz(5_000_000_000L);
        System.out.println();
    }
}
```

```
ILDI
IIL
```

`yaz(5)` tam eşleşmedir: I. `yaz(5L)` 'deki L soneki sayıyı `long` yapar: L. `yaz(5.0f)` 'deki f soneki sayıyı `float` yapar. `float` sürümü yoktur; `float` zincirde yalnızca `double` 'a genişleyebilir: D. `yaz('A')` 'da `char` hem `int` 'e hem `long` 'a hem `double` 'a genişleyebilir; en özel olan `int` 'tir: I. İkinci satırda `byte` ve `short` argümanlar en yakın sürüme, `int` 'e gider: I I. `5_000_000_000L` `int` aralığını aşan bir `long` 'dur: L. Sayıdaki alt çizgiler yalnızca okunurluk içindir; değeri değiştirmez.

En özel metodun bulunamadığı durumlar da vardır. İki uygun metottan hiçbiri ötekinden daha özel değilse derleyici seçim yapmaz ve çağrıyı reddeder:

```
public class Belirsiz {
    static void g(int a, double b) { }
    static void g(double a, int b) { }
    public static void main(String[] args) {
        g(1, 1);
    }
}
```

```
Belirsiz.java:5: error: reference to g is ambiguous
        g(1, 1);
        ^
  both method g(int,double) in Belirsiz and method g(double,int) in Belirsiz match
1 error
```

`g(1, 1)` çağrısında iki sürüm de uygundur: birincisinde ikinci argüman, ikincisinde birinci argüman `double`'a genişletilir. `(int, double)` ile `(double, int)` arasında hiçbiri ötekine genişleyemez; ikisi de "daha özel" değildir. İletideki `ambiguous` sözcüğü "belirsiz" demektir. Çağrı `g(1, 1.0)` biçiminde yazılırsa belirsizlik kalkar. Bu hata, aşırı yüklemede parametre tiplerinin yalnızca sırasını değiştirmenin neden iyi bir fikir olmadığını gösterir.

3.2.7 Fonksiyon içinde fonksiyon

Metotlar yalnızca `main`'den çağrılmaz; bir metot başka bir metodu da çağırabilir. Büyük bir görev bu sayede küçük parçalara bölünür. Her parça kendi işini yapar ve sonucunu bir üst düzeye verir.

3.2.7.1 Metotların birbirini çağırması

Aşağıdaki programda iki sayının karelerinin toplamını bulan `ikiKareToplami` metodu işi kendisi yapmaz. Kare alma işini `kareAl`'a, toplama işini `topla`'ya yaptırır.

```java
public class IcIce {
    public static int kareAl(int sayi) {
        return sayi * sayi;
    }
    public static int topla(int sayi1, int sayi2) {
        return sayi1 + sayi2;
    }
    public static int ikiKareToplami(int sayi1, int sayi2) {
        return topla(kareAl(sayi1), kareAl(sayi2));
    }
    public static void main(String[] args) {
        int sonuc = ikiKareToplami(3, 4);
        System.out.println(sonuc);
    }
}
```

```
25
```

`ikiKareToplami(3, 4)` çağrısının sonucu $3^2 + 4^2 = 9 + 16 = 25$'tir. Kodun önemli satırı `return topla(kareAl(sayi1), kareAl(sayi2));`'dir. Bu satırda bir çağrının argümanları başka çağrılardır. Java bu tür ifadeyi içten dışa hesaplar. `topla` çağrılmadan önce iki argümanının değeri bilinmelidir; bu yüzden önce `kareAl(sayi1)`, sonra `kareAl(sayi2)` çalışır. İki sonuç hazır olunca `topla(9, 16)` çağrılır.

Argümanların hesaplanma sırası Java'da kesin olarak tanımlıdır: **soldan sağa**. Bu kural, argümanlardan biri ekrana yazdığında ya da bir değişkeni değiştirdiğinde fark eder. Aşağıdaki programda `deger` metodu hem değer döndürür hem de adını ekrana yazar; böylece çağrıların sırası çıktıda görülür.

```
public class Sira {
    static int deger(String ad, int v) {
        System.out.print(ad + " ");
        return v;
    }
    static int fark(int a, int b) {
        return a - b;
    }
    public static void main(String[] args) {
        int s = fark(deger("sol", 10), deger("sağ", 3));
        System.out.println("→ " + s);
    }
}
```

```
sol sağ → 7
```

Önce `sol`, sonra `sağ` yazılır. Ardından `fark(10, 3)` çalışır ve 7 döner. C gibi bazı diller bu sırayı belirlemez ve derleyiciye bırakır. Java ise sırayı garanti eder.

Metotların birbirini çağırması programı okunur ve yönetilebilir kılar. `ikiKareToplami`'nın gövdesi ne yaptığını tek satırda söyler: iki karenin toplamı. `kareAl` ve `topla` ayrı ayrı sınanabilir. `kareAl` yanlış çalışıyorsa hata tek bir yerde düzeltilir ve onu kullanan bütün metotlar düzelir.

3.2.7.2 Çağrı yığınını izlemek

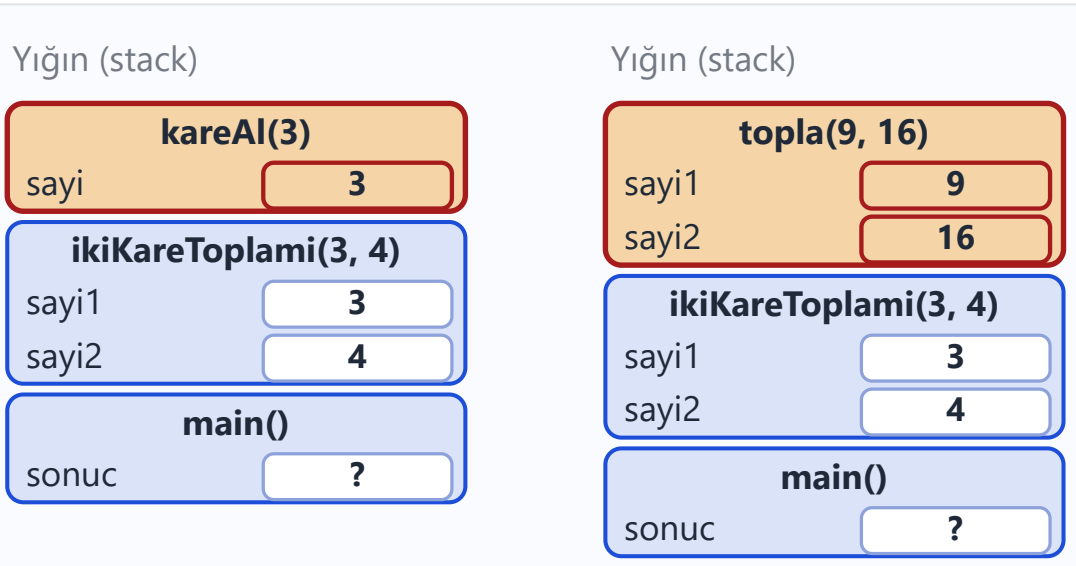
İç içe çağrılarda çağrı yığını birden çok katlı olur. Tablo 3.2.7 programın çalışmasını adım adım verir.

Adım	Satır	Çağrı yığını	Değişkenler	Çıktı
1	12: `int sonuc = ikiKareToplami(3, 4);`	main()	sonuc=?	—
2	9: `return topla(kareAl(sayi1), kareAl(sayi2));`	main() → ikiKareToplami(3, 4)	sayi1=3, sayi2=4	—
3	2: `return sayi * sayi;`	main() → ikiKareToplami(3, 4) → kareAl(3)	sayi=3, dönüş=9	—

Adım	Satır	Çağrı yığını	Değişkenler	Çıktı
4	9: `return topla(kareAl(sayi1), kareAl(sayi2));`	main() → ikiKareToplami(3, 4)	sayi1=3, sayi2=4, arg1=9	—
5	2: `return sayi * sayi;`	main() → ikiKareToplami(3, 4) → kareAl(4)	sayi=4, dönüş=16	—
6	5: `return sayi1 + sayi2;`	main() → ikiKareToplami(3, 4) → topla(9, 16)	sayi1=9, sayi2=16, dönüş=25	—
7	9: `return topla(kareAl(sayi1), kareAl(sayi2));`	main() → ikiKareToplami(3, 4)	sayi1=3, sayi2=4, dönüş=25	—
8	12: `int sonuc = ikiKareToplami(3, 4);`	main()	sonuc=25	—
9	13: `System.out.println(sonuc);`	main()	sonuc=25	`25`

Tablo 3.2.7: `ikiKareToplami(3, 4)` çağrısında çağrı yığını. Değişkenler sütunu en üstteki çerçeveyi gösterir; `arg1`, `topla`'nın hesaplanmış ilk argümanıdır.

Birinci adımda `main`, `ikiKareToplami(3, 4)`'ü çağırır; yığında iki çerçeve olur. İkinci adımda `ikiKareToplami`, `topla`'yı çağırmadan önce argümanlarını hesaplamaya başlar. Üçüncü adımda `kareAl(3)` için üçüncü bir çerçeve açılır; 3 × 3 = 9 döner. Şekil 3.2.7'nin ilk parçası bu anı gösterir.



Şekil 3.2.7: Üç katlı çağrı yığını. Solda `kareAl(3)` çalışıyor; altında `ikiKareToplami(3, 4)` ve `main` bekliyor. Sağda `topla(9, 16)` çalışıyor; kendi `sayi1`'i 9'dur, alttaki `ikiKareToplami`'nın `sayi1`'i ise hâlâ 3'tür.

Dördüncü adımda `kareAl`'ın çerçevesi kapanır. İlk argüman artık bilinir: 9. Beşinci adımda `kareAl(4)` için yeni bir çerçeve açılır ve 16 döner. Bu çerçeve, önceki `kareAl` çerçevesinin yeniden kullanılması değildir; aynı metodun yeni bir çağrısı için açılmış ayrı bir çerçevedir. Her çağrı kendi parametrelerini ve yerel değişkenlerini taşır.

Altıncı adımda argümanlar hazırdır ve `topla(9, 16)` çağrılır. `topla`'nın parametreleri de `sayi1` ve `sayi2` adını taşır. `topla`'nın çerçevesinde `sayi1 = 9`'dur; aynı anda yığının bir alt katındaki `ikiKareToplami` çerçevesinde `sayi1 = 3`'tür. İki değişken birbirini etkilemez. Yedinci adımda `topla`'nın döndürdüğü 25, `ikiKareToplami`'nın da dönüş değeri olur. Sekizinci adımda `main`'e dönülür ve `sonuc = 25` olur. Dokuzuncu adımda 25 yazılır.

İzleme boyunca yığında aynı anda en fazla üç çerçeve bulunmuştur: `main`, `ikiKareToplami` ve o an çalışan yardımcı metot. Yığının derinliği, iç içe çağrıların en uzun zincirine eşittir. `kareAl` ile `topla` aynı anda yığında hiç bulunmamıştır, çünkü biri bitmeden öteki başlamamıştır.

Yığının sınırsız olmadığını da bilmek gerekir. Her çerçeve bellekte yer kaplar. Bir metot kendini durmadan çağırırsa çerçeveler birikir ve yığın dolar; program `StackOverflowError` hatasıyla durur. Kendini çağıran metotlara özyinelemeli metot denir; bu tür metotlarda yığının derinliği ayrıca denetlenmelidir.

3.2.8 Hazır metotlar

Her programcı her işi baştan yazmaz. Java, binlerce sınıf ve metottan oluşan bir standart kütüphaneyle gelir. Karekök almak, ekrana biçimli yazmak, diziyi sıralamak, dosya okumak için hazır metotlar vardır. Bu metotları kullanmak, kendi yazdığımız metotları kullanmaktan farklı değildir: adları, parametreleri ve dönüş tipleri bilinir, içleri bilinmez.

3.2.8.1 Java kütüphanesi ve paketler

Java'nın standart kütüphanesine Java API (*Application Programming Interface*, uygulama programlama arayüzü) denir. Kütüphanedeki sınıflar **paketlere** (*package*) ayrılmıştır. Paket, ilgili sınıfları bir araya toplayan bir ad alanıdır. Tablo 3.2.8 en sık kullanılan sınıfları ve paketlerini listeler.

Alan	Sınıflar	Örnek metotlar	Paket
Ekrana yazma	`System.out`	`print`, `println`, `printf`	`java.lang`
Matematik	`Math`	`sqrt`, `abs`, `max`, `pow`	`java.lang`
Metin	`String`	`length`, `charAt`, `substring`, `equals`	`java.lang`
Dizi	`Arrays`	`sort`, `copyOf`, `binarySearch`, `equals`	`java.util`
Kullanıcı girişi	`Scanner`	`nextInt`, `nextLine`, `close`	`java.util`
Dosya	`File`, `FileWriter`	`createNewFile`, `write`, `close`	`java.io`
Veri tabanı	`DriverManager`, `Statement`	`getConnection`, `executeQuery`, `executeUpdate`	`java.sql`
Veri yapıları	`ArrayList`	`add`, `remove`, `get`, `size`	`java.util`
Zaman	`LocalDateTime`, `DateTimeFormatter`	`now`, `ofPattern`, `format`	`java.time`

Alan	Sınıflar	Örnek metotlar	Paket
Web / grafik	`URI`, `URL`, `JFrame`	`URI.create`, `openConnection`, `setVisible`	`java.net`, `javax.swing`

Tablo 3.2.8: Java API'de en sık kullanılan sınıflar ve metotlar.

`java.lang` paketi özeldir: her Java programına otomatik olarak dahil edilir. `System`, `Math` ve `String` bu pakette bulunduğu için hiçbir hazırlık yapmadan kullanılır. Öteki paketlerdeki sınıflar için dosyanın başına bir **içe aktarma** (*import*) bildirimi yazılır: `import java.util.Scanner;`. Bu bildirim yazılmazsa derleyici `Scanner` adını tanımaz ve `cannot find symbol` der.

Tablodaki metotların bir kısmı `static`'tir ve sınıf adıyla çağrılır: `Math.sqrt(16)`, `Arrays.sort(dizi)`. Bir kısmı ise bir nesne üzerinden çağrılır: `giris.nextInt()`, `metin.length()`. Bir metodun hangi biçimde çağrılacağı, Java'nın resmî belgelerinde metodun başlığında yazar.

3.2.8.2 Ekrana yazdırma

`System.out`, standart çıktıyı, yani konsolu temsil eden hazır bir nesnedir. Üç metodu çok sık kullanılır.

```
public class Yazdirma {
    public static void main(String[] args) {
        System.out.println("Merhaba, Dünya!");
        System.out.print("Bir ");
        System.out.print("satır");
        System.out.println();
        int yas = 20;
        System.out.println("Yaş: " + yas);
        System.out.printf("Pi yaklaşık %.2f%n", Math.PI);
    }
}
```

```
Merhaba, Dünya!
Bir satır
Yaş: 20
Pi yaklaşık 3,14
```

- `System.out.println(x)`, `x`'i yazar ve alt satıra geçer. Argümansız `println()` yalnızca satır sonu yazar.
- `System.out.print(x)`, `x`'i yazar ama aynı satırda kalır. Programda `print("Bir ")` ile `print("satır")` art arda çalışır, ardından `println()` satırı bitirir: `Bir satır`.
- `System.out.printf(biçim, değerler...)`, biçimli yazar. Biçim metnindeki yer tutucuların yerine değerler konur.

Metin ile değer `+` işleciyle birleştirilir: `"Yaş: " + yas` ifadesinin değeri `"Yaş: 20"` metnidir. `+` işleci soldan sağa çalıştığı için birleştirmenin sırası sonucu değiştirir. Bu ayrıntı yeni başlayanların sık takıldığı bir noktadır:

```
import java.util.Locale;

public class Bicim {
    public static void main(String[] args) {
        double ort = 81.66666666666667;
        System.out.printf("%.2f%n", ort);
        System.out.printf(Locale.US, "%.2f%n", ort);
        System.out.printf("%d öğrenci, ortalama %.1f, not %s%n", 3, ort, "BB");
        System.out.printf("[%5d] [%-5d] [%05d]%n", 42, 42, 42);
        System.out.println(1 + 2 + " toplam");
        System.out.println("toplam " + 1 + 2);
        System.out.println("toplam " + (1 + 2));
        System.out.println(String.format("%.3f", Math.PI));
    }
}
```

```
81,67
81.67
3 öğrenci, ortalama 81,7, not BB
[   42] [42   ] [00042]
3 toplam
toplam 12
toplam 3
3,142
```

`1 + 2 + " toplam"` ifadesinde önce `1 + 2` iki sayının toplamıdır: 3. Sonra 3 metne eklenir: `3 toplam`. `"toplam " + 1 + 2` ifadesinde ise ilk işlem bir metinle bir sayının birleştirilmesidir: `"toplam 1"`. Sonuç artık metin olduğu için 2 de metne eklenir: `toplam 12`. Toplamın sayı olarak hesaplanması için parantez gerekir: `"toplam " + (1 + 2)`.

`printf`'teki en sık kullanılan yer tutucular Tablo 3.2.9'da verilmiştir.

Yer tutucu	Anlamı	Örnek	Çıktı
`%d`	tam sayı	`printf("%d", 42)`	`42`
`%.2f`	ondalık sayı, 2 basamak	`printf("%.2f", Math.PI)`	`3,14`
`%s`	metin	`printf("%s", "BB")`	`BB`
`%n`	satır sonu	`printf("a%nb")`	iki satır
`%5d` / `%-5d`	5 karakterlik alana sağa / sola yaslı	`printf("[%5d]", 42)`	`[ 42]`
`%05d`	boşluk yerine sıfırla doldur	`printf("[%05d]", 42)`	`[00042]`

Tablo 3.2.9: `printf` yer tutucuları (Türkçe yerel ayarla çalıştırıldı).

`printf`, ondalık sayıyı yazarken bilgisayarın **yerel ayarını** (*locale*) kullanır. Türkçe ayarlı bir sistemde ondalık ayırıcı virgüldür; bu yüzden `Math.PI` `3,14` olarak yazılır. `println` ise yerel ayara bakmaz ve her zaman nokta kullanır: `3.141592653589793`. Bir program aynı sayıyı iki yöntemle yazarsa iki farklı biçim görülür. Ondalık ayırıcının her sistemde nokta olması isteniyorsa yerel ayar açıkça verilir: `printf(Locale.US, "%.2f%n", ort)` çıktısı `81.67`'dir. `Locale` sınıfı `java.util` paketindedir ve içe aktarılması gerekir.

`%.2f` yer tutucusu sayıyı keserek değil yuvarlayarak yazar: 81,666... değeri `81,67` olur. `String.format` da `printf` ile aynı biçim kurallarını kullanır, ama sonucu ekrana yazmak yerine metin olarak döndürür. Son satırdaki `3,142` bu metottan gelir.

3.2.8.3 Math sınıfı

`Math` sınıfı matematiksel işlemler için hazır metotlar içerir. Metotlarının hepsi `static`'tir; bu yüzden bir `Math` nesnesi oluşturulmaz, metotlar sınıf adıyla çağrılır.

```
double kareKok = Math.sqrt(16);
double mutlakDeger = Math.abs(-17.2);
double maksimum = Math.max(3, 5);
System.out.println(kareKok);
System.out.println(mutlakDeger);
System.out.println(maksimum);
System.out.println(Math.pow(2, 10));
System.out.println(Math.round(2.6));
System.out.println(Math.min(-4, 7));
System.out.println(Math.max(3, 5));
```

```
4.0
17.2
5.0
1024.0
3
-4
5
```

Çıktıdaki her satırın biçimi, metodun dönüş tipini ele verir. Tablo 3.2.10 bu metotları dönüş tipleriyle birlikte listeler.

Metot	Görevi	Dönüş tipi	Örnek → sonuç
`sqrt(x)`	karekök	her zaman `double`	`Math.sqrt(16)` → `4.0`
`pow(a, b)`	a üzeri b	her zaman `double`	`Math.pow(2, 10)` → `1024.0`
`abs(x)`	mutlak değer	argümanın tipi	`Math.abs(-17.2)` → `17.2`
`max(a, b)`, `min(a, b)`	büyük / küçük olan	argümanların ortak tipi	`Math.max(3, 5)` → `5`

Metot	Görevi	Dönüş tipi	Örnek → sonuç
`round(x)`	en yakın tam sayıya yuvarlama	`double` için `long`	`Math.round(2.6)` → `3`
`floor(x)` , `ceil(x)`	aşağı / yukarı yuvarlama	`double`	`Math.floor(-2.5)` → `-3.0`
`PI`	π sabiti (metot değil)	`double`	`Math.PI` → `3.141592653589793`

Tablo 3.2.10: Sık kullanılan `Math` metotları ve dönüş tipleri.

`abs` , `max` ve `min` aşırı yüklenmiştir; argümanın tipine göre farklı sürümleri çalışır. `Math.max(3, 5)` iki `int` alır ve `int` döndürür: `5` . Aynı çağrının sonucu programda bir `double` değişkene atanınca 5.0'a genişletilir ve `5.0` yazılır. `sqrt` ve `pow` 'un ise yalnızca `double` sürümü vardır; tam sayı argüman verilse de sonuç `double` 'dır. Bu yüzden sonucu doğrudan bir `int` değişkene atamak derleme hatasıdır:

```
public class SqrtInt {
    public static void main(String[] args) {
        int k = Math.sqrt(16);
    }
}
```

```
SqrtInt.java:3: error: incompatible types: possible lossy conversion from double to int
        int k = Math.sqrt(16);
                          ^
1 error
```

Sonuç tam sayı olarak gerekiyorsa açık dönüşüm yazılır: `int k = (int) Math.sqrt(16);` . `(int)` dönüşümü ondalık kısmı keser, yuvarlamaz. Dönüş tiplerinin birlikte kullanıldığı ifadelerde sonuç dikkatle izlenmelidir. Aşağıdaki program birkaç ek örnek verir.

```
public class MathEk {
    public static void main(String[] args) {
        System.out.println(Math.round(2.5) + " " + Math.round(-2.5) + " " +
Math.round(-2.6));
        System.out.println(Math.floor(-2.5) + " " + Math.ceil(-2.5));
        System.out.println(Math.sqrt(-4));
        System.out.println((int) Math.pow(2, 10));
        System.out.println(Math.max(Math.abs(-8), Math.sqrt(49)));
        System.out.println(Math.sqrt(16) + Math.max(3, 7) / 2);
        System.out.println(Math.PI);
        System.out.println(Math.hypot(3, 4));
        long r = Math.round(2.6);
        System.out.println(r);
    }
}
```

```
3 -2 -3
-3.0 -2.0
NaN
1024
8.0
7.0
3.141592653589793
5.0
3
```

İlk satır `Math.round`'un davranışını gösterir. Metot sayıya 0,5 ekleyip aşağı yuvarlar. 2,5 için sonuç 3'tür; −2,5 için −2,5 + 0,5 = −2 bulunur. Yani yarımlar her zaman artı sonsuza doğru yuvarlanır; −2,6 ise −3 olur. İkinci satırda `floor` −2,5'i aşağıya, −3,0'a; `ceil` yukarıya, −2,0'a yuvarlar. Negatif bir sayının karekökü gerçel sayılarda tanımsızdır. `Math.sqrt(-4)` hata vermez; özel `NaN` (*Not a Number*, sayı değil) değerini döndürür.

Beşinci satırdaki `Math.max(Math.abs(-8), Math.sqrt(49))` ifadesinde iç çağrılar önce çalışır. `Math.abs(-8)` bir `int` verir: 8. `Math.sqrt(49)` bir `double` verir: 7.0. `max`'ın argümanları `(int, double)` olduğu için aşırı yükleme kuralı devreye girer ve `max(double, double)` sürümü seçilir. Sonuç `8.0`'dır, `8` değil.

Altıncı satırdaki `Math.sqrt(16) + Math.max(3, 7) / 2` ifadesinde işleç önceliği önemlidir. Bölme toplamadan önce yapılır. `Math.max(3, 7)` `int` döndürür: 7. `7 / 2` tam sayı bölmesidir: 3. Sonra `4.0 + 3` hesaplanır ve sonuç `7.0` olur. `Math.hypot(3, 4)` dik kenarları 3 ve 4 olan üçgenin hipotenüsünü verir: 5.0.

3.2.9 Kapsam

Bir değişken programın her yerinden kullanılamaz. Nerede tanımlandığı, nerede görülebileceğini ve ne kadar süre yaşayacağını belirler. Bu kurallar metotların birbirinden bağımsız çalışmasını sağlar: bir metodun içindeki değişken adları, öteki metotlardaki adlarla karışmaz.

3.2.9.1 Yerel değişken

Tanım 3.2.9

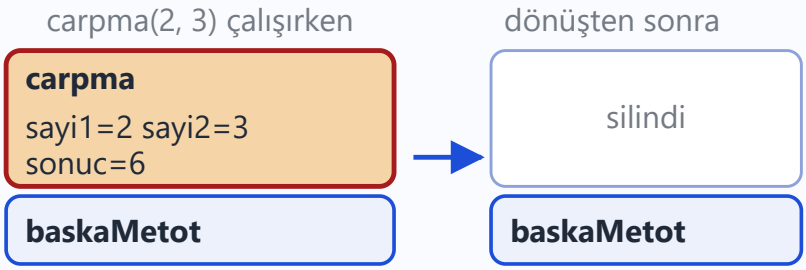
Yerel değişken (*local variable*), bir metodun ya da bir bloğun içinde tanımlanan, yalnızca orada geçerli olan değişkendir. Metodun parametreleri de yerel değişkendir. Yerel değişken yalnızca metot çalışırken var olur; metot dönünce silinir.

Aşağıdaki kodda `carpma` metodu `sonuc` adlı bir yerel değişken kullanır. `baskaMetot`, `carpma`'yı çağırdıktan sonra bu değişkene ulaşmaya çalışır.

```
public int carpma(int sayi1, int sayi2) {
    int sonuc = sayi1 * sayi2; // 'sonuc' bir yerel değişkendir
    return sonuc;
}
public void baskaMetot() {
    carpma(2, 3);
    System.out.println(sonuc);  // HATA
}
```

```
YerelDisari.java:8: error: cannot find symbol
        System.out.println(sonuc);  // HATA
                           ^
  symbol:   variable sonuc
  location: class YerelDisari
1 error
```

Derleyici `sonuc` adını `baskaMetot`'un içinde bulamaz. `sonuc`, `carpma`'ya aittir ve yalnızca `carpma` çalışırken vardır. Şekil 3.2.8 bunu bellek üzerinde gösterir: `carpma(2, 3)` çalışırken çerçevesinde `sayi1`, `sayi2` ve `sonuc` bulunur. Metot dönünce çerçeve ve içindeki değişkenler silinir.



Şekil 3.2.8: Yerel değişkenin ömrü. Solda `carpma(2, 3)` çalışırken yığının üstünde onun çerçevesi vardır: `sayi1 = 2`, `sayi2 = 3`, `sonuc = 6`. Sağda metot dönmüştür; çerçeve silinmiş, altta yalnızca `baskaMetot` kalmıştır.

Değeri metodun dışına taşımanın yolu yine `return`'dür. `baskaMetot` içinde `int s = carpma(2, 3);` yazılırsa dönen 6, `baskaMetot`'un kendi yerel değişkeni olan `s`'ye atanır. Derleme hatası veren kodda `carpma(2, 3);` satırı ise sonucu hiçbir yere atamadığı için 6 kaybolur.

Yerel değişkenlerin bir kuralı daha vardır: kullanılmadan önce değer almaları gerekir. Sınıf alanlarının ve dizi öğelerinin varsayılan değeri (0, `false`, `null`) vardır; yerel değişkenlerin yoktur. Değer verilmemiş bir yerel değişkeni okumak derleme hatasıdır:

```
public class DegersizYerel {
    public static void main(String[] args) {
        int t;
        System.out.print(t);
    }
}
```

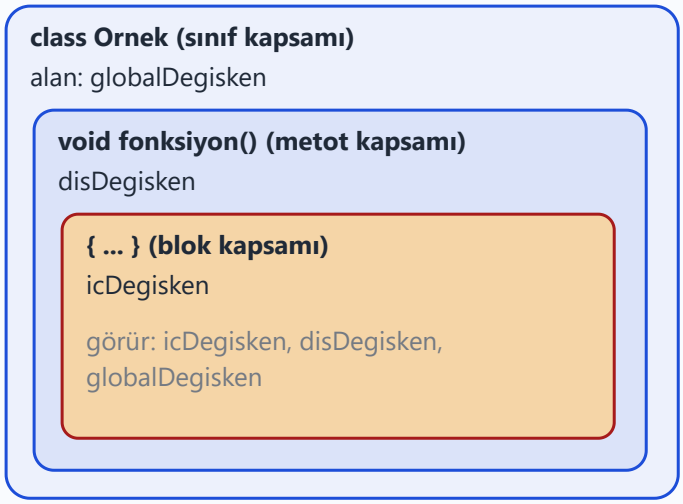
```
DegersizYerel.java:4: error: variable t might not have been initialized
        System.out.print(t);
                         ^
1 error
```

3.2.9.2 Kapsam kuralları ve iç içe bloklar

Tanım 3.2.10

Kapsam (*scope*), bir değişkenin adının programda geçerli olduğu bölgedir. Java'da yerel bir değişkenin kapsamı, tanımlandığı satırdan başlar ve tanımlandığı süslü parantez bloğunun sonunda biter.

Kapsam iki şeyi belirler: değişkene nereden **erişilebileceğini** ve değişkenin ne kadar **yaşayacağını**. Java'da bloklar iç içe olabilir. Sınıfın gövdesi en dıştaki bloktur. Metotlar onun içindedir. Metotların içinde de `if`, döngü ya da tek başına yazılmış süslü parantez blokları bulunabilir. Şekil 3.2.9 bu katmanları gösterir.



Şekil 3.2.9: İç içe kapsamlar. En dışta sınıf kapsamı ve onun alanı `globalDegisken`; içinde metot kapsamı ve `disDegisken`; en içte blok kapsamı ve `icDegisken`. En içteki blok, dıştaki bütün değişkenleri görür.

Kural tek yönlüdür: iç kapsam, dış kapsamdaki değişkenleri görür; dış kapsam, içteki değişkenleri göremez. Aşağıdaki programda iç blok hem metodun hem sınıfın değişkenlerine erişir.

```
public class IcKapsam {
    static int globalDegisken = 10; // global (sınıf alanı)
    static void fonksiyon() {
        int disDegisken = 10;       // yerel kapsam
        {
            int icDegisken = 5;
            // disDegisken ve icDegisken burada erişilebilir
            System.out.println(disDegisken + icDegisken
                    + globalDegisken);
        }
        // icDegisken burada YOK
    }
    public static void main(String[] args) {
        fonksiyon();
    }
}
```

```
25
```

İç bloktaki `println` üç değişkeni de görür: `icDegisken` kendi bloğunda, `disDegisken` bir dış blokta, `globalDegisken` sınıf kapsamındadır. Toplam 10 + 5 + 10 = 25'tir. Bloğun kapanan süslü parantezinden sonra `icDegisken` artık yoktur. Orada `icDegisken` kullanılırsa derleyici `cannot find symbol` der.

Sınıf gövdesinde, metotların dışında tanımlanan `globalDegisken` bir **alandır** (*field*). Sınıfın bütün metotları bu alanı görür. Programlama dillerinde "global değişken" diye anılan, her yerden erişilen ve program boyunca yaşayan değişkenlerin Java'daki karşılığı `static` alanlardır. Java'da sınıf dışında değişken tanımlanamaz; gerçek anlamda global bir değişken yoktur.

Kapsam	Nerede tanımlanır?	Nereden erişilir?	Ne kadar yaşar?
Yerel (blok)	Bir bloğun içinde	Tanımlandığı satırdan bloğun sonuna kadar	Blok çalıştığı sürece
Yerel (metot)	Metot gövdesinde ya da parametre listesinde	Metodun içinden	Metot çalıştığı sürece
`static` alan	Sınıf gövdesinde, metotların dışında	Sınıfın bütün metotlarından	Program boyunca

Tablo 3.2.11: Kapsam türleri.

Döngü değişkenleri de bu kurala uyar. `for (int i = 0; ...)` başlığında tanımlanan `i`'nin kapsamı döngünün kendisidir. Döngü bittikten sonra `i`'yi kullanmak derleme hatasıdır:

```
public class DonguDisi {
    public static void main(String[] args) {
        for (int i = 0; i < 3; i++) { }
        System.out.print(i);
    }
}
```

```
DonguDisi.java:4: error: cannot find symbol
        System.out.print(i);
                         ^
  symbol:   variable i
  location: class DonguDisi
1 error
```

Kapsamın sınırlı olmasının bir yararı da aynı adın birbirini görmeyen bloklarda yeniden kullanılabilmesidir. İki kardeş blok, yani biri bitince öteki başlayan bloklar, aynı adlı değişkenleri birbirine karışmadan tanımlayabilir. Art arda yazılan iki `for` döngüsünün ikisinde de `i` kullanmak bu yüzden geçerlidir:

```
public class KardesBlok {
    public static void main(String[] args) {
        { int t = 1; System.out.print(t + " "); }
        { int t = 2; System.out.println(t); }
        for (int i = 0; i < 2; i++) { int kare = i * i; System.out.print(kare + " "); }
        for (int i = 5; i < 7; i++) { System.out.print(i + " "); }
        System.out.println();
    }
}
```

```
1 2
0 1 5 6
```

3.2.9.3 İsim çakışması ve gölgeleme

Aynı adın farklı kapsamlarda kullanılması iki ayrı sonuç doğurur. Java bunlardan birini yasaklar, ötekine izin verir.

Yasak olan durum: bir iç blok, dıştaki *yerel* bir değişkenle aynı adı kullanamaz. İki değişken aynı anda yaşayacağı ve aynı adı taşıyacağı için hangisinin kastedildiği belirsiz olurdu. Java bu belirsizliğe izin vermez:

```
public void fonksiyon() {
    int disDegisken = 10;
    {
        int disDegisken = 5;  // HATA
    }
}
```

```
AyniAd.java:5: error: variable disDegisken is already defined in method fonksiyon()
            int disDegisken = 5;  // HATA
                ^
1 error
```

Aynı kural parametreler için de geçerlidir, çünkü parametreler de yerel değişkendir. Metodun gövdesinde parametreyle aynı adlı bir değişken tanımlanamaz:

```
public class ParametreAyniAd {
    static int f(int x) {
        int x = 3;
        return x;
    }
}
```

```
ParametreAyniAd.java:3: error: variable x is already defined in method f(int)
        int x = 3;
            ^
1 error
```

İzin verilen durum: bir yerel değişken, sınıfın bir *alanıyla* aynı adı taşıyabilir. Bu durumda yerel değişken alanı **gölgeler**.

Tanım 3.2.11

Gölgeleme (*shadowing*), iç kapsamdaki bir adın dış kapsamdaki aynı adı gizlemesidir. Yerel değişken bir alanı gölgelediğinde, o kapsam içinde ad yerel değişkeni gösterir. Gölgelenen `static` alana sınıf adıyla ulaşılır: `SinifAdi.alan`.

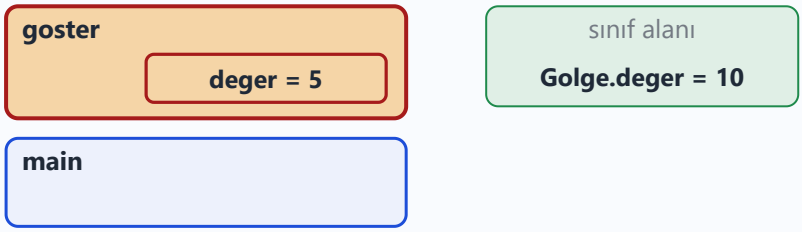
```
public class Golge {
    static int deger = 10;  // sınıf alanı
    static void goster() {
        int deger = 5;      // alanı gölgeler
        System.out.println(deger);
        System.out.println(Golge.deger);
    }
    public static void main(String[] args) {
        goster();
        System.out.println(deger);
    }
}
```

```
5
10
10
```

Java bir adı görünce önce en içteki kapsama bakar, bulamazsa bir dış kapsama çıkar. `goster`'in içinde `deger` adı yerel değişkende bulunur ve 5 yazılır. `Golge.deger` yazımı ise aramayı doğrudan sınıfın alanına yönlendirir ve 10 yazılır. `main`'de `deger` adlı bir yerel değişken yoktur; arama sınıf kapsamına çıkar ve alan bulunur: 10. Tablo 3.2.12 bu adımları izler.

Adım	Satır	Çağrı yığını	Değişkenler	Çıktı
1	9: `goster();`	main()	Golge.deger=10	—
2	3: `static void goster() {`	main() → goster()	Golge.deger=10	—
3	4: `int deger = 5; // alanı gölgeler`	main() → goster()	Golge.deger=10, deger (yerel)=5	—
4	5: `System.out.println(deger);`	main() → goster()	Golge.deger=10, deger (yerel)=5	`5`
5	6: `System.out.println(Golge.deger);`	main() → goster()	Golge.deger=10, deger (yerel)=5	`10`
6	10: `System.out.println(deger);`	main()	Golge.deger=10	`10`

Tablo 3.2.12: Gölgelemenin izlenmesi. `Golge.deger` sınıf alanını, "deger (yerel)" `goster`'in yerel değişkenini gösterir.



Şekil 3.2.10: Gölgeleme anı. Sınıf alanı `Golge.deger = 10` yığının dışında, program boyunca yaşar. `goster`'in çerçevesinde aynı adlı yerel `deger = 5` vardır; metodun içinde `deger` adı bu yerel değişkeni gösterir.

Gölgeleme derleme hatası vermez, ama sinsi hatalara yol açabilir. Programcı alanı değiştirdiğini sanırken yerel bir kopyayı değiştiriyor olabilir. Gölgelemenin bilinçli kullanıldığı yaygın bir yer de vardır: nesnelerle çalışırken yapıcı ve `set` metotlarında parametre, alanla aynı adı taşır ve alana `this.alan` ile ulaşılır.

3.2.9.4 Yaşam süresi

Kapsam bir değişkenin *nereden* görüldüğünü, yaşam süresi *ne zaman* var olduğunu anlatır. Yerel değişkenler bloklarına girildiğinde doğar, bloktan çıkıldığında ölür. `static` alanlar program başlarken bir kez oluşur ve program bitene kadar yaşar. Aşağıdaki program iki tür değişkeni bir arada kullanır.

```
static int cagriSayisi = 0;  // sınıf alanı
static int carpma(int sayi1, int sayi2) {
  cagriSayisi++;
  int sonuc = sayi1 * sayi2;
  return sonuc;
}
public static void main(String[] args) {
  int ilk = carpma(2, 3);
  {
    int ikinci = carpma(ilk, 4);
    System.out.println(ikinci);
  }
  System.out.println(ilk + " " + cagriSayisi);
}
```

```
24
6 2
```

Adım	Satır	Çağrı yığını	Değişkenler	`cagriSayisi`	Çıktı
1	8: `int ilk = carpma(2, 3);`	main()	ilk=?	0	—
2	3: `cagriSayisi++;`	main() → carpma(2, 3)	sayi1=2, sayi2=3	1	—
3	4: `int sonuc = sayi1 * sayi2;`	main() → carpma(2, 3)	sayi1=2, sayi2=3, sonuc=6	1	—
4	8: `int ilk = carpma(2, 3);`	main()	ilk=6	1	—
5	4: `int sonuc = sayi1 * sayi2;`	main() → carpma(6, 4)	sayi1=6, sayi2=4, sonuc=24	2	—
6	10: `int ikinci = carpma(ilk, 4);`	main()	ilk=6, ikinci=24	2	—
7	11: `System.out.println(ikinci);`	main()	ilk=6, ikinci=24	2	`24`
8	13: `System.out.println(ilk + " " + cagriSayisi);`	main()	ilk=6	2	`6 2`

Tablo 3.2.13: Değişkenlerin doğuşu ve ölümü. `cagriSayisi` sütunu sınıf alanının o anki değerini gösterir.

Program başlarken `cagriSayisi` alanı oluşur ve 0 değerini alır. Birinci adımda `carpma(2, 3)` çağrılır. İkinci adımda metot sınıf alanını 1 artırır. Metotlar sınıf alanına erişebilir, çünkü alan sınıf kapsamındadır. Üçüncü adımda yerel `sonuc` doğar ve 6 olur. Dördüncü adımda metot döner; `sayi1`, `sayi2` ve `sonuc` yok olur. Dönen 6, `main`'in yerel değişkeni `ilk`'e atanır.

Beşinci adımda iç bloğun içinden `carpma(ilk, 4)` çağrılır. Yeni çerçevede `sonuc = 24` olur ve alan 2'ye çıkar. Bu `sonuc`, ilk çağrıdaki `sonuc` değildir; ilk çağrının `sonuc`'u çoktan silinmiştir. Altıncı adımda `ikinci` doğar. `ikinci` iç blokta tanımlandığı için yalnızca o blokta yaşar. Şekil 3.2.11 bu anı gösterir.



Şekil 3.2.11: Yaşam süresi. Solda iç blok çalışıyor: `main`'in çerçevesinde `ilk = 6` ve `ikinci = 24`; sınıf alanı `cagriSayisi = 2` ayrı bir bölgede. Sağda blok bitmiş: `ikinci` yok olmuş, `ilk` ve alan yaşamaya devam ediyor.

Yedinci adımda blok içinde `ikinci` yazılır: 24. Sekizinci adımda blok bitmiştir ve `ikinci` artık yoktur. `ilk` ise `main` bitene kadar yaşar. Son satır `6 2` yazar: `ilk` 6'dır ve `carpma` iki kez çağrılmıştır.

Bu örnekte sınıf alanının bir "sayaç" olarak kullanılması dikkat çekicidir. Yerel bir değişken bu işi yapamazdı: `carpma`'nın içinde tanımlanan bir sayaç her çağrıda yeniden doğar ve 0'dan başlar. Çağrılar arasında saklanması gereken bir bilgi, yerel değişkende değil alanda tutulur. Yine de alanlar ölçülü kullanılmalıdır. Her metodun değiştirebildiği bir alan, programın davranışını izlemeyi zorlaştırır.

3.2.10 Soyutlama ve tasarım

Bir metodu derlenir ve doğru çalışır hâle getirmek işin yarısıdır. Öteki yarısı, metodun iyi tasarlanmasıdır: ne kadar iş yapacağı, hangi girdileri alacağı, adının ne olacağı. İyi tasarlanmış metotlar başka programlarda yeniden kullanılır, kolay sınanır ve yıllar sonra bile anlaşılır. Bu kesim tasarımın dayandığı fikirleri, yani soyutlamayı, genellemeyi ve ayrıştırmayı, ardından pratik ilkeleri anlatır.

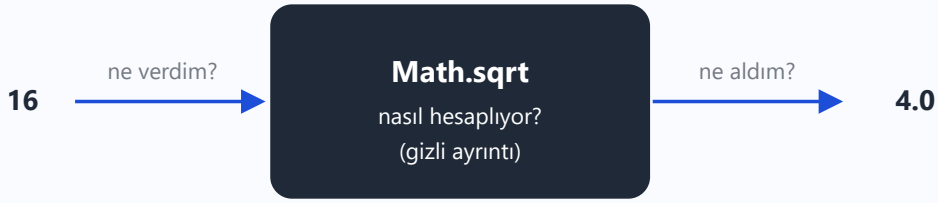
3.2.10.1 Soyutlama ve kara kutu

Tanım 3.2.12

Soyutlama (*abstraction*), karmaşıklığı azaltmak için ayrıntıları gizleyip yalnızca önemli özellikleri göstermektir.

Günlük hayatta soyutlama her yerdedir. Araba sürmek için motorun içini bilmek gerekmez. Direksiyon, pedallar ve vites kolu, sürücünün arabayla konuştuğu bir arayüzdür. Motor değişse bile, örneğin benzinli motor yerine elektrik motoru konsa bile, sürücü aynı direksiyonu ve aynı pedalları kullanır.

Metotlar da bir tür soyutlamadır. Şekil 3.2.12'de `Math.sqrt` bir kara kutu olarak çizilmiştir. Kullanıcı kutuya 16 verir ve 4.0 alır. Kutunun içinde karekökün hangi yöntemle hesaplandığı gizlidir ve kullanıcıyı ilgilendirmez.



Şekil 3.2.12: Kara kutu. Kullanıcı yalnızca girdiyi ("ne verdim?" 16) ve çıktıyı ("ne aldım?" 4.0) bilir. `Math.sqrt`'ın sonucu nasıl hesapladığı gizli bir ayrıntıdır.

Soyutlamanın iki kazancı vardır. Birincisi, programı anlaşılır kılar. Büyük bir program, her biri ayrı ayrı düşünülebilen parçalara ayrılır; bir parçayı kullanan, onun içini bilmek zorunda kalmaz. İkincisi, değişikliği kolaylaştırır. Bir metodun içi değiştiğinde, metodun ne yaptığı aynı kaldığı sürece onu kullanan kod etkilenmez.

Hesaplamalar üzerindeki bu tür soyutlamaya **işlevsel soyutlama** ya da **yordamsal soyutlama** (*procedural abstraction*) denir. Temel fikir, bir fonksiyonun *nasıl* hesapladığından çok *ne* hesapladığının önemli olmasıdır. Kullanıcı yalnızca metodun imzasını ve görevini bilir:

```
// Kullanan yalnızca imzayı bilir:
static boolean asalMi(int sayi)
// "sayi asal ise true döner"
if (asalMi(17)) { ... }
```



Şekil 3.2.13: `asalMi` bir kara kutu olarak. 17 girer, `true` çıkar. Gövde gizlidir; kullanıcı yalnızca girdiyi ve çıktıyı bilir.

`asalMi(17)` çağrısını yazan programcının bilmesi gereken tek şey, metodun bir sayının asal olup olmadığını `boolean` olarak döndürdüğüdür. Metot sayıyı 2'den başlayarak bütün sayılara mı bölüyor, yoksa yalnızca kareköke kadar mı deniyor? Bu soru metodu kullanan için önemsizdir. Yarın daha hızlı bir algoritma yazılırsa yalnızca `asalMi`'nin gövdesi değişir; `if (asalMi(17))` satırı aynen kalır.

En basit örnek yine `System.out.println("Merhaba")`'dır. Bu metot verilen metni konsola yazar. Karakterlerin hangi kodlamayla gönderildiği, işletim sisteminin hangi çağrısının kullanıldığı gibi ayrıntılar programcıdan gizlidir. Soyutlama olmasaydı, ekrana tek bir satır yazmak için bile bu ayrıntıların hepsini bilmek gerekirdi.

Soyutlamanın işlemesi için metodun görevi açık olmalıdır. Kullanıcı gövdeye bakmayacağına göre metodun adı, parametreleri ve dönüş tipi onun ne yaptığını eksiksiz anlatmalıdır. `asalMi(int sayi)` başlığı bunu yapar. `islem2(int a)` başlığı ise hiçbir şey anlatmaz; kullanıcı gövdeyi okumak zorunda kalır ve soyutlama çöker.

3.2.10.2 Genelleme

Bir kod parçasının anlamı çevresine bağlıdır. Bir sayının değeri denklemdeki öteki terimlerle, bir kelimenin anlamı cümledeki yeriyle ilgilidir. Kodda da bir deyimin ne yaptığı, içinde bulunduğu blokta hangi değişkenlerin ne değer taşıdığına bağlıdır. Bu çevreye **bağlam** (*context*) denir.

> **Tanım 3.2.13**
>
> **Genelleme** (*generalization*), bir bağlama ait çözümü başka bağlamlarda da kullanılabilir hâle getirmektir. Kodda genellemenin aracı parametredir: çözümün içine gömülü sabit değerler parametreye dönüştürülür.

```
public class Genelleme {
    // Özel: yalnızca 10 tane '-' yazar
    static void cizgiCiz() {
        System.out.println("----------");
    }
    // Genel: uzunluğu ve karakteri parametre
    static void cizgiCiz(int uzunluk, char karakter) {
        for (int i = 0; i < uzunluk; i++) {
            System.out.print(karakter);
        }
        System.out.println();
    }
    public static void main(String[] args) {
        cizgiCiz();
        cizgiCiz(5, '*');
        cizgiCiz(12, '=');
    }
}
```

```
----------
*****
============
```

Parametresiz `cizgiCiz` yalnızca tek bir iş yapar: on tane tire yazar. Uzunluk ve karakter metodun içine gömülüdür. Beş yıldızlı bir çizgi gerektiğinde bu metot işe yaramaz; yeni bir metot yazmak gerekir. İki parametreli sürüm ise aynı işi genelleştirir. Gömülü 10 sayısı `uzunluk` parametresine, gömülü `'-'` karakteri `karakter` parametresine dönüşmüştür. Artık tek bir metot her uzunlukta ve her karakterle çizgi çizer: `cizgiCiz(5, '*')` beş yıldız, `cizgiCiz(12, '=')` on iki eşittir işareti yazar.

Örnekteki iki metodun adı aynıdır; bu bir aşırı yükleme örneğidir. Özel sürüm genel sürümle de yazılabilirdi: gövdesi tek satırlık `cizgiCiz(10, '-');` çağrısı olurdu. Böylece çizgi çizme mantığı tek bir yerde kalır.

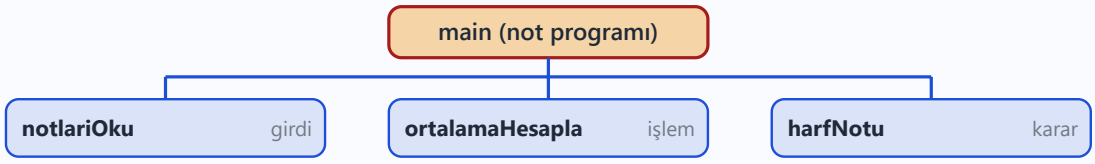
Genellemenin bir sınırı vardır. Her sabiti parametreye dönüştürmek metodu kullanışsız yapar. On parametreli bir metodu çağırmak, işi elle yapmaktan daha zor olabilir. Genelleme, gerçekten değişmesi beklenen değerlerle sınırlı tutulur.

3.2.10.3 Ayrıştırma

Tanım 3.2.14

Ayrıştırma (*decomposition*), büyük bir problemi küçük ve birbirinden bağımsız alt problemlere bölmektir. Programlamada her alt problem bir metot olur.

Bir öğrencinin üç notunu okuyup ortalamasını ve harf notunu yazan bir program düşünelim. Problem üç alt göreve ayrılır: notları okumak, ortalamayı hesaplamak ve ortalamaya göre harf notunu bulmak. Şekil 3.2.14 bu yapıyı gösterir.



Şekil 3.2.14: Not programının parçaları. `main` üç alt metodu çağırır: `notlariOku` (girdi), `ortalamaHesapla` (işlem) ve `harfNotu` (karar).

Aşağıdaki programda dört metodun hepsi tam olarak yazılmıştır. `harfNotu`'ndaki eşikler yaygın bir harf notu tablosuna göre seçilmiştir; farklı bir tablo yalnızca bu metodun değişmesini gerektirir.

```java
import java.util.Scanner;

public class NotProgrami {
    // notlariOku ve harfNotu da ayrı metotlardır
    static int[] notlariOku(Scanner giris, int adet) {
        int[] notlar = new int[adet];
        for (int i = 0; i < adet; i++) {
            notlar[i] = giris.nextInt();
        }
        return notlar;
    }
    static double ortalamaHesapla(int[] notlar) {
        int toplam = 0;
        for (int n : notlar) {
            toplam += n;
        }
        return (double) toplam / notlar.length;
    }
    static String harfNotu(double ort) {
        if (ort >= 90) return "AA";
        if (ort >= 85) return "BA";
        if (ort >= 80) return "BB";
        if (ort >= 75) return "CB";
        if (ort >= 70) return "CC";
        if (ort >= 60) return "DC";
        if (ort >= 50) return "DD";
        return "FF";
    }
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int[] notlar = notlariOku(giris, 3);
        double ort = ortalamaHesapla(notlar);
        System.out.println("Ortalama: " + ort + " -> " + harfNotu(ort));
    }
}
```

Program iki farklı girdiyle çalıştırıldı:

```
Ortalama: 81.66666666666667 -> BB
```

```
Ortalama: 52.333333333333336 -> DD
```

Her metot tek bir iş yapar. `notlariOku`, `Scanner` nesnesini ve okunacak not sayısını parametre olarak alır, okuduğu notları bir dizi olarak döndürür. `Scanner` nesnesi `main`'de bir kez oluşturulur ve metoda verilir; böylece okuma metodu girdinin nereden geldiğini bilmek zorunda kalmaz. `ortalamaHesapla` diziyi alır ve ortalamayı döndürür. `(double) toplam / notlar.length` ifadesin-

deki açık dönüşüm, tam sayı bölmesini önler. 80 + 90 + 75 = 245'tir; 245 / 3 tam sayı bölmesiyle 81 olurdu, ondalıklı bölmeyle 81,666... olur. `harfNotu` ortalamaya bakar ve bir `String` döndürür. Erken `return` kalıbı burada bir `else if` zincirinin yerini tutar: koşullar yukarıdan aşağıya denenir ve ilk sağlanan koşulda metot biter.

`main` artık bir "içindekiler" sayfası gibi okunur: oku, hesapla, yazdır. Ne yapıldığı `main`'de görünür; nasıl yapıldığı alt metotlardadır. Bir hata çıktığında hangi metoda bakılacağı da bellidir. Ortalama yanlışsa `ortalamaHesapla`'ya, harf notu yanlışsa `harfNotu`'na bakılır.

Parçalamanın bir başka kazancı da metotların ayrı ayrı sınanabilmesidir. `harfNotu` klavyeye hiç dokunmadan sınanabilir: `harfNotu(85)` "BA", `harfNotu(84.99)` "BB" döndürmelidir. Okuma ile hesaplama iç içe yazılmış olsaydı her deneme için klavyeden not girmek gerekirdi.

3.2.10.4 Ne zaman fonksiyon? DRY ilkesi

Bir kod parçasının metot olması gerektiğini anlamanın en açık işareti tekrardır. Bu ilkenin adı **DRY**'dır (*Don't Repeat Yourself*, kendini tekrar etme). Kodu kopyalayıp yapıştırma ihtiyacı doğduğu an, o kod bir metot olmaya adaydır.

Aşağıdaki ilk parça tekrarlı, ikincisi metotlu sürümdür. İkisi de iki öğrencinin ortalamasını yazar.

```
double ort1 = (70 + 80 + 90) / 3.0;
System.out.println(ort1);
double ort2 = (55 + 60 + 100) / 3.0;
System.out.println(ort2);
// formül değişirse her yerde
// ayrı ayrı düzeltmek gerekir
```

```
static double ortalama(int a, int b, int c) {
    return (a + b + c) / 3.0;
}
System.out.println(ortalama(70, 80, 90));
System.out.println(ortalama(55, 60, 100));
```

```
80.0
71.66666666666667
80.0
71.66666666666667
```

Tekrarlı sürümde ortalama formülü iki kez yazılmıştır. Formül değişirse, örneğin notlar farklı ağırlık alacak olursa, iki yerin de ayrı ayrı düzeltilmesi gerekir. Gerçek programlarda aynı formül onlarca yerde bulunabilir ve birinin unutulması kolaydır. Metotlu sürümde formül tek bir yerdedir: `ortalama` metodunun gövdesi. Değişiklik bir kez yapılır ve bütün çağrılara yansır.

Çıktıdaki `71.66666666666667` değeri `double`'ın sınırlı duyarlılığından kaynaklanır. 215 / 3 = 71,666... sonsuz bir ondalık açılımdır. `double` bunu yaklaşık 16 anlamlı basamakla saklar ve son basamak yuvarlanır. Kullanıcıya gösterilecek bir sonuçta `printf("%.2f", ...)` ile basamak sayısı sınırlanır.

Tekrar tek ölçüt değildir. Bir kez kullanılan bir kod parçası da metot olmayı hak edebilir. Bunun iki nedeni vardır: parça kendi başına anlamlı bir iş yapıyordur ya da onu bir ad altında toplamak çağıran kodu okunur yapıyordur. Not programındaki `harfNotu` yalnızca bir kez çağrılır, ama ayrı bir metot olması `main`'i sadeleştirir ve sınamayı kolaylaştırır.

3.2.10.5 İyi metot tasarımı

İyi bir metodun özellikleri altı ilkede toplanabilir.

Tek sorumluluk
: Her metot yalnızca bir iş yapar. Metodun görevi "ve" bağlacıyla anlatılıyorsa (oku *ve* hesapla *ve* yazdır) metot bölünmelidir.

Açıklayıcı ad
: Metodun adı ne yaptığını anlatan bir fiildir: `ortalamaHesapla`, `asalMi`, `notlariOku`. `islem2` gibi adlar bir şey anlatmaz.

Parametreler
: Metot ihtiyaç duyduğu verileri parametrelerle alır; gizlice dış değişkenlere bağlı olmaz. Az sayıda parametre okunurluğu artırır.

Dönüş değeri
: Sonuç açıkça `return` ile döndürülür. Hesaplayan metot yazdırmaz; yazdırma işini çağıran yapar.

Kısa ve odaklı
: Bir ekrana sığan, birkaç satırlık metotlar kolay anlaşılır ve kolay sınanır.

Yerinde çağrı
: Metot yalnızca gerektiği yerde çağrılır. Aynı sonucu bir döngüde tekrar tekrar hesaplamak gereksiz iştir; sonuç bir kez hesaplanıp bir değişkende tutulur.

"Hesaplayan metot yazdırmaz" ilkesi en sık çiğnenen ilkedir. Bir öğrencinin ilk aklına gelen tasarım genellikle şudur:

```
static void ortalamaHesaplaVeYaz(int a, int b, int c) {
    System.out.println((a + b + c) / 3.0);
}
```

Bu metot ortalamayı yalnızca ekrana yazabilir. Ortalamanın 50'den büyük olup olmadığını sınamak, ortalamayı bir dosyaya kaydetmek ya da sınıf ortalamasına katmak isteyen kod onu kullanamaz; sonuç ekranda kalmıştır. Daha iyi tasarım, ortalamayı döndürmek ve ne yapılacağına çağıranın karar vermesidir:

```
static double ortalama(int a, int b, int c) {
    return (a + b + c) / 3.0;
}
// çağıran karar verir:
System.out.println(ortalama(70, 80, 90));
if (ortalama(40, 50, 45) < 50) System.out.println("kaldı");
```

"Gizlice dış değişkenlere bağlı olmamak" ilkesi de aynı gerekçeye dayanır. Değerlerini sınıf alanlarından okuyan bir metot, alanların o anki değerine bağlıdır. Metodun ne döndüreceğini anlamak için programın geri kalanında alanların nerede değiştirildiğini bilmek gerekir. Parametre alan bir metodun sonucu ise yalnızca argümanlarına bağlıdır; aynı argümanlarla her seferinde aynı sonucu verir.

3.2.10.6 Adlandırma

İyi ad seçmek yalnızca metotlar için değil, değişkenler için de geçerlidir. Her değişken tek bir şeyi temsil etmelidir. Aynı değişkenin önce yaş, sonra toplam için kullanılması okuyanın kafasını karıştırır ve mantık hatalarına yol açar. Bir değişkenin adı, tuttuğu verinin ne olduğunu anlatmalıdır.

```
int ogrenciSayisi;      // iyi bir adlandırma
int a;                  // anlamsız bir isim
int yas = 25;
String isim = "Ahmet";
double ortalama = 85.5;
String kitapAdi;
```

Anlamsız ya da tek harfli adlar okunurluğu düşürür. İki istisna vardır: döngü sayacı olarak `i`, `j` ve matematiksel formüllerdeki `x`, `y` gibi geleneksel adlar. Tablo 3.2.14 birkaç kötü adı daha iyileriyle karşılaştırır.

Kötü	İyi	Neden?
`int x1, x2;`	`int vizeNotu, finalNotu;`	Hangi notun hangisi olduğu addan anlaşılır.
`double d;`	`double daireAlani;`	Değişkenin neyin değerini tuttuğu bellidir.
`boolean f;`	`boolean asalMi;`	`if (asalMi)` bir soru gibi okunur.

Tablo 3.2.14: Kötü ve iyi değişken adları.

Java'nın yazım gelenekleri de adın türünü ele verir. Değişken ve metot adları camelCase yazılır ve küçük harfle başlar: `ogrenciSayisi`, `notlariOku`. Sınıf adları büyük harfle başlar: `Kitap`, `Scanner`. Değişmeyen sabitler büyük harflerle ve alt çizgiyle yazılır: `MAKS_DEGER`. Bu gelenekler derleyici için zorunlu değildir, ama Java programcıları arasında evrenseldir. Bir adı gören okuyucu, onun değişken mi, sınıf mı, sabit mi olduğunu hemen anlar.

Değişken ve metot adlarında Türkçe karakter kullanmak Java'da teknik olarak serbesttir: `int öğrenciSayısı;` derlenir. Ancak dosyanın kodlaması ile derleyicinin beklediği kodlama uyuşmadığında sorun çıkar. Bu yüzden derslerde ve çoğu projede adlar İngiliz alfabesindeki harflerle yazılır: `ogrenciSayisi`.

3.2.11 Örnek problemler

Aşağıdaki problemler metot çağrısını, değer ile aktarımı ve aşırı yüklemeyi birlikte kullanır. Bütün çözümler Java ile çalıştırılmıştır.

Örnek 3.2.1

Aşağıdaki program ne yazdırır? `a` neden değişmez? `kup(2)` çalışırken yığında en çok kaç çerçeve bulunur?

```
static int f(int x) {
    x = x * 2;
    return x + 1;
}
public static void main(String[] args) {
    int a = 3;
    int b = f(a);
    System.out.println(a + " " + b);
    System.out.println(f(f(1)));
}
```

```
static int kare(int n) { return n * n; }
static int kup(int n) { return n * kare(n); }
// main içinde:
System.out.println(kup(2) + kare(3));
```

```
3 7
7
17
```

`f(a)` çağrısında `a`'nın değeri, yani 3, `f`'nin parametresi `x`'e kopyalanır. `x = x * 2` kopyayı 6 yapar ve `x + 1`, yani 7 döner. `b` 7 olur. `a` değişmez, çünkü `f` yalnızca kopyasını değiştirmiştir: ilk satır `3 7`'dir.

`f(f(1))` ifadesinde önce içteki çağrı çalışır: `f(1)` = 1 × 2 + 1 = 3. Dıştaki çağrı bu değerle çalışır: `f(3)` = 3 × 2 + 1 = 7. İkinci satır `7`'dir.

`kup(2) + kare(3)` ifadesinde `kup(2)`, `2 * kare(2)`'yi hesaplar. `kare(2)` 4 döndürür ve `kup(2)` 8 olur. Sonra `kare(3)` = 9 hesaplanır. Toplam 8 + 9 = 17'dir. `kup(2)` çalışırken en derin an `kare(2)`'nin çalıştığı andır: yığında `main`, `kup(2)` ve `kare(2)` olmak üzere 3 çerçeve vardır. `kare(3)` çağrıldığında `kup`'un çerçevesi çoktan kapanmıştır; o an yığında yalnızca 2 çerçeve bulunur.

Örnek 3.2.2

Aşağıdaki ilk iki `println` hangi `topla` metodunu çağırır? Son satır neden `8` değil `8.0` yazdırır?

```
static int topla(int a, int b) { return a + b; }
static double topla(double a, double b) { return a + b; }
// main içinde:
System.out.println(topla(2, 3.0));
System.out.println(topla(2, 3));
System.out.println(Math.max(Math.abs(-8), Math.sqrt(49)));
```

```
5.0
5
8.0
```

`topla(2, 3.0)` çağrısının argümanları `(int, double)`'dır. `(int, int)` sürümü uymaz, çünkü 3.0 `int`'e daralmaz. `(double, double)` sürümü uyar: 2 değeri 2.0'a genişletilir. Sonuç `5.0`'dır. `topla(2, 3)` çağrısında iki sürüm de uygundur; en özel olan `(int, int)` seçilir ve `5` yazılır. Son satırda `Math.sqrt` `double` döndürdüğü için `Math.max`'ın `(double, double)` sürümü seçilir. 8 ile 7.0'ın büyüğü 8.0'dır.

Örnek 3.2.3

Aşağıdaki metotları yazalım: bir sayının çift olup olmadığını döndüren `ciftMi`; üç sayının en büyüğünü `Math.max` ile bulan `enBuyuk`; `Math.pow` kullanmadan üs alan `usAl`; rakamlar toplamını bulan `basamakToplami`; dairenin alanını döndüren `daireAlani`; daire ve dikdörtgen için aşırı yüklenmiş `alan`.

```java
public class Alistirmalar {
    static boolean ciftMi(int sayi) {
        return sayi % 2 == 0;
    }
    static int enBuyuk(int a, int b, int c) {
        return Math.max(a, Math.max(b, c));
    }
    static int usAl(int taban, int us) {
        int sonuc = 1;
        for (int i = 0; i < us; i++) sonuc *= taban;
        return sonuc;
    }
    static int basamakToplami(int n) {
        int t = 0;
        while (n != 0) { t += n % 10; n /= 10; }
        return t;
    }
    static double daireAlani(double r) {
        return Math.PI * r * r;
    }
    static double alan(double r) {
        return Math.PI * r * r;
    }
    static double alan(double en, double boy) {
        return en * boy;
    }
    public static void main(String[] args) {
        System.out.println(ciftMi(7) + " " + ciftMi(10) + " " + ciftMi(-4));
        System.out.println(enBuyuk(4, 9, 2));
        System.out.println(usAl(2, 10) + " " + usAl(5, 0));
        System.out.println(basamakToplami(2024) + " " + basamakToplami(-2024));
        System.out.printf("%.2f%n", daireAlani(2));
        System.out.println(alan(1) + " " + alan(3, 4));
    }
}
```

```
false true true
9
1024 1
8 -8
12,57
3.141592653589793 12.0
```

`ciftMi`'nin gövdesi tek satırdır: `return sayi % 2 == 0;`. Karşılaştırmanın sonucu zaten bir `boolean`'dır; `if (sayi % 2 == 0) return true; else return false;` yazmaya gerek yoktur. Negatif sayılarda da doğru çalışır: `-4 % 2` 0'dır.

`enBuyuk`, `Math.max`'ı iç içe çağırır: önce `b` ile `c`'nin büyüğü bulunur, sonra bu değer `a` ile karşılaştırılır. `enBuyuk(4, 9, 2)` 9 döndürür. `usAl` bir biriktirici kullanır. `sonuc` 1'den başlar ve `us` kez `taban` ile çarpılır. `usAl(2, 10)` 1024 döndürür. `usAl(5, 0)`'da döngü hiç dönmez ve 1 döner; bu, matematikteki $5^0 = 1$ kuralıyla uyumludur.

`basamakToplami`, sayının son rakamını `n % 10` ile alır ve `n /= 10` ile atar. Tablo 3.2.15 2024 için döngüyü izler.

Tur	`n % 10`	`t` (sonra)	`n / 10`
1	4	4	202
2	2	6	20
3	0	6	2
4	2	8	0
`n = 0`: döngü biter, `return t;` → 8			

Tablo 3.2.15: `basamakToplami(2024)` döngüsünün izlenmesi (`t` 0'dan başlar).

Negatif sayılarda Java'nın kalan işlemi bölünenin işaretini aldığı için `basamakToplami(-2024)` −8 döndürür. Rakamlar toplamının her zaman pozitif olması isteniyorsa metodun başında `n = Math.abs(n);` yazılır.

`daireAlani(2)` = π × 2 × 2 ≈ 12,566 değerini döndürür. `printf("%.2f%n", ...)` bunu iki basamağa yuvarlar; Türkçe yerel ayarda çıktı `12,57`'dir. Aşırı yüklenmiş `alan` metotlarında hangi sürümün çağrılacağını argüman sayısı belirler. `alan(1)` daire sürümünü çalıştırır ve π'yi verir: `3.141592653589793`. `alan(3, 4)` dikdörtgen sürümünü çalıştırır: `12.0`. Tam sayı argümanlar `double` parametrelere genişletildiği için sonuç da `double`'dır.

3.2.12 Sık yapılan hatalar

Metotlarla ilgili hataların bir kısmını derleyici yakalar ve iletisiyle yerini gösterir. Bir kısmı ise derlenen ama beklenmedik sonuç veren programlara yol açar. İkinci gruptakiler daha tehlikelidir, çünkü program hiçbir uyarı vermeden yanlış çalışır.

Sık yapılan hata

`static` olmayan metodu `main`'den çağırmak. İleti: `non-static method toplama(int,int) cannot be referenced from a static context`. Ya metodun başlığına `static` eklenir ya da bir nesne oluşturulup metot onun üzerinden çağrılır.

Sık yapılan hata

Bazı yollarda `return` unutmak. İleti: `missing return statement`. Derleyici koşulların değerini düşünmez; `if (n > 0) ... if (n <= 0) ...` zinciri insana tam görünse de hata verir. Son dal düz bir `else` olur ya da metodun sonuna varsayılan bir `return` konur.

Sık yapılan hata

`void` metodun sonucunu kullanmak. `int x = selamla();` satırı `incompatible types: void cannot be converted to int`, `println(selamla())` ise `'void' type not allowed here` iletisini verir. `void` metot değer üretmez; çağrısı yalnızca tek başına bir deyim olabilir.

Sık yapılan hata

Eksik ya da fazla argüman. İleti: `method toplama in class ... cannot be applied to given types`; altında `required: int,int` ve `found: int` satırları beklenen ve verilen tipleri gösterir. Argümanların sayısı, sırası ve tipleri imzayla uyuşmalıdır.

Sık yapılan hata

Yerel değişkeni metodun dışında kullanmak. İleti: `cannot find symbol ... variable sonuc`. Yerel değişken metot bitince yok olur; değeri `return` ile döndürülür ve çağıran onu kendi değişkenine atar.

Sık yapılan hata

İç blokta dıştaki yerel değişkenle aynı ad. İleti: `variable disDegisken is already defined in method fonksiyon()`. Aynı kural parametreler için de geçerlidir. Bir yerel değişkenin bir *alanla* aynı adı taşıması ise serbesttir; o zaman gölgeleme olur.

Sık yapılan hata

Takas metodunun çalışacağını sanmak. `yerDegistir(int x, int y)` yalnızca kopyaları takas eder; `yerDegistir(dizi[0], dizi[2])` çağrısından sonra dizi aynı kalır. Dizi hücrelerini takas etmek için metot diziyi ve indisleri alır: `yerDegistir(int[] d, int i, int j)`.

Sık yapılan hata

Argümanın değişeceğini sanmak. `arttir(sayi)` çağrısından sonra `sayi` aynı kalır; değişen, metodun kopyasıdır. Yeni değer `return` ile döndürülür ve atanır: `sayi = arttir(sayi);` . Derleyici, dönen değerin atanmamasına uyarı vermez.

Sık yapılan hata

Dönüş tipiyle aşırı yüklemek. `int f(int a)` ve `double f(int a)` birlikte olamaz: `method f(int) is already defined` . İmzayı yalnızca ad ve parametre tipleri belirler; dönüş tipi ve parametre adları imzaya girmez.

Sık yapılan hata

`Math` **sonucunun tipini unutmak.** `int k = Math.sqrt(16);` satırı `possible lossy conversion from double to int` iletisini verir. `sqrt` ve `pow` her zaman `double` döndürür; tam sayı gerekiyorsa `(int)` dönüşümü yazılır.

Sık yapılan hata

Dönüş tipinin hesabı değiştireceğini sanmak. `static double yarim(int n) { return n / 2; }` metodu 7 için 3.0 döndürür. Bölme iki `int` arasında yapılır ve kesir atılır; `double` dönüş tipi yalnızca sonucu genişletir. Doğru yazım `n / 2.0` 'dır.

Sık yapılan hata

Hesaplayan metotta yazdırmak. `ortalamaHesaplaVeYaz` gibi bir metodun sonucu ekranda kalır; başka bir hesapta kullanılamaz. Metot sonucu döndürür, yazdırma işini çağıran yapar.

3.2.13 Özet

- **Fonksiyon** bir işi yapan, adı olan ve tekrar kullanılabilen kod parçasıdır. Java'da her zaman bir sınıfın içinde yazılır ve **metot** adını alır. Girdi parametrelerle gelir, çıktı `return` ile döner.
- Metot başlığı: `erişim [static] dönüşTipi ad(parametreler)` . Her parametre kendi tipini taşır. **İmza** = ad + parametre tiplerinin sırası; dönüş tipi imzaya girmez. Metotlar iç içe tanımlanamaz.
- **Parametre** tanımdaki değişken, **argüman** çağrıdaki değerdir. Argümanlar önce hesaplanır (soldan sağa), sonra parametrelere kopyalanır. Sayı, sıra ve tip uyuşmalıdır.

- Her çağrı **çağrı yığınında** yeni bir **çerçeve** açar; çerçeve parametreleri ve yerel değişkenleri tutar. Metot dönünce çerçeve silinir. Yığının derinliği, iç içe çağrıların en uzun zinciri kadardır.
- `return` değeri verir ve metodu hemen bitirir; arkasındaki deyime ulaşılamaz. Değer döndüren metotta her yol bir `return` ile bitmelidir; derleyici koşulların değerini düşünmez. `void` metot değer döndürmez, tek başına `return;` ile erken çıkabilir.
- Dönen değer dönüş tipine genişleyebilir (`int` → `double`), daralamaz. Dönüş tipi ifadenin nasıl hesaplanacağını değiştirmez.
- `static` metot sınıfa aittir; doğrudan ya da `Sinif.metot()` ile çağrılır. `static` olmayan metot bir nesne üzerinden çağrılır: `nesne.metot()` .
- Java argümanları **değer ile aktarır**. İlkel argümanın kopyası değişse de çağıranın değişkeni değişmez. Dizi argümanında referans kopyalanır: metot ortak diziyi değiştirebilir, ama parametreye yeni dizi atamak çağıranı etkilemez.
- **Aşırı yükleme**: aynı ad, farklı parametre listesi. Seçimi derleyici, derleme zamanında ve argüman tiplerine bakarak yapar. Önce tam eşleşme aranır, yoksa genişletme denenir (`byte → short → int → long → float → double` , `char → int`). Birden çok uygun sürüm varsa en özeli seçilir; en özel sürüm yoksa çağrı belirsizdir.
- `System.out.println` satır sonu ekler, `print` eklemez, `printf` biçimli yazar ve ondalık ayırıcıda yerel ayarı kullanır. `+` birleştirmesi soldan sağa çalışır.
- `Math` metotları `static` 'tir. `sqrt` ve `pow` her zaman `double` , `abs` / `max` / `min` argümanın tipini, `round(double)` `long` döndürür. `java.lang` dışındaki paketler `import` ister.
- Yerel değişken tanımlandığı blokta yaşar ve kullanılmadan önce değer almalıdır. İç kapsam dışı görür, dış kapsam içi göremez. İç blok dıştaki yerelle aynı adı kullanamaz; yerel değişken ise aynı adlı alanı **gölgeler**. `static` alanlar program boyunca yaşar.
- Tasarım: **soyutlama** (ne yaptığı önemli, nasıl yaptığı gizli), **genelleme** (sabitler parametreye), **ayrıştırma** (her alt görev bir metot), **DRY** (tekrar eden kod metoda), tek sorumluluk, açıklayıcı ad, sonucu döndürmek ve yazdırmayı çağırana bırakmak.

3.2.14 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

fonksiyon *function* · metot *method* · metot çağrısı *method call* · parametre *parameter* · argüman *argument* · dönüş tipi *return type* · dönüş değeri *return value* · metot imzası *method signature* · erişim belirleyici *access modifier* · static üye *static member* · çağrı yığını *call stack* · çerçeve *stack frame* · değer ile aktarım *pass by value* · referans *reference* · heap *heap* · aşırı yükleme *method overloading* · genişletme / daraltma dönüşümü *widening / narrowing conversion* · içe aktarma *import* · paket *package* · yerel ayar *locale* · yerel değişken *local variable* · alan *field* · kapsam *scope* · gölgeleme *shadowing*

çöp toplayıcı *garbage collector* | soyutlama *abstraction* | ayrıştırma *decomposition*

3.2.15 Kendini Dene

1. `static int a(int n) { return b(n + 1) * 2; }`, `static int b(int n) { return c(n) + n; }` ve `static int c(int n) { return n * n; }` metotları veriliyor. `main`'den `a(2)` çağrılırsa ne döner? Yığında aynı anda en çok kaç çerçeve bulunur?

   Cevap

   `a(2)`, `b(3)`'ü çağırır. `b(3)`, `c(3)`'ü çağırır ve `c(3)` 9 döndürür. `b(3)` = 9 + 3 = 12 döner. `a(2)` = 12 × 2 = 24 döner. `c(3)` çalışırken yığında `main`, `a`, `b` ve `c` olmak üzere 4 çerçeve vardır; en derin an budur.

2. `static int degis(int[] d, int k) { d[k] = d[k] + k; k = 0; return d[k]; }` metodu, `main`'de `int[] d = {3, 4, 5}; int k = 2; int r = degis(d, k);` ile çağrılıyor. Çağrıdan sonra `d`, `k` ve `r` nedir?

   Cevap

   Metodun `d`'si `main`'deki diziyi gösteren referansın kopyasıdır; `k`'si 2'nin kopyasıdır. `d[2] = 5 + 2 = 7` ortak diziyi değiştirir. `k = 0` yalnızca kopyayı değiştirir. `return d[0]` 3 döndürür. Sonuç: dizi `{3, 4, 7}`, `main`'deki `k` hâlâ 2, `r = 3`.

3. Bir sınıfta `m(long x)`, `m(double x)` ve `m(int x, int y)` metotları var; her biri kendi parametre tiplerini metin olarak döndürüyor. `short s = 7; char ch = 'x';` tanımlarından sonra `m(s)`, `m(ch)`, `m(3.0f)` ve `m(2, 'b')` hangi sürümleri çağırır?

   Cevap

   Tek argümanlı çağrılarda iki parametreli sürüm sayı yüzünden elenir. `short` hem `long`'a hem `double`'a genişleyebilir; `long` daha özeldir: `m(s)` → long. `char` da önce `int`'e, oradan `long`'a genişler; yine `long` seçilir: `m(ch)` → long. `float` yalnızca `double`'a genişler: double. `m(2, 'b')`'de `'b'` `int`'e genişler: int,int. JDK 21 çıktısı: `long long double int,int`.

4. Sınıfta `static int sayac = 0;` alanı ve `static int artir(int sayac) { sayac++; KendiniDene.sayac += 2; return sayac; }` metodu var. `int t = artir(5);` satırından sonra

`t` ve sınıf alanı `sayac` kaçtır?

Cevap

Parametre `sayac` alanı gölgeler. `sayac++` parametreyi 6 yapar; alan etkilenmez. `KendiniDene.sayac += 2` sınıf adıyla alana ulaşır ve onu 2 yapar. Metot parametreyi döndürür: `t = 6`. Alan 2'dir.

5. `static double oran(int a, int b) { return a / b; }` metodu `oran(7, 2)` ve `oran(1, 3)` için ne döndürür? Metot nasıl düzeltilir?

Cevap

`a / b` iki `int` arasında tam sayı bölmesidir. 7 / 2 = 3 ve 1 / 3 = 0 bulunur; dönerken `double`'a genişletilir: `3.0` ve `0.0`. Dönüş tipi hesabı değiştirmez. Düzeltme: `return (double) a / b;`. Bu durumda sonuçlar 3.5 ve 0.333... olur.

6. Aşağıdaki metot çiftlerinden hangileri aynı sınıfta birlikte bulunabilir? (a) `void g(int x)` ve `int g(int y)`; (b) `int h(int a, double b)` ve `int h(double a, int b)`; (c) `int k(int a)` ve `int k(int a, int b)`.

Cevap

(a) olamaz: iki imza da `g(int)`'tir; dönüş tipi ve parametre adı imzaya girmez. (b) birlikte bulunabilir, çünkü parametre tiplerinin sırası farklıdır. Ama `h(1, 1)` gibi bir çağrı `reference to h is ambiguous` hatası verir; derleyici iki sürümden birini seçemez. (c) birlikte bulunabilir: parametre sayıları farklıdır.

7. Bir metot şöyle yazılmış: `static String durum(int puan) { if (puan >= 50) return "geçti"; else if (puan < 50) return "kaldı"; }`. Derlenir mi? Derlenmiyorsa en küçük düzeltme nedir?

Cevap

Derlenmez: `missing return statement`. İki koşul her tam sayıyı kapsasa da derleyici koşulların değerini değerlendirmez; zincirin son dalı `else if` olduğu için akışın metodun sonuna ulaşabileceğini varsayar. En küçük düzeltme `else if (puan < 50)` yerine düz `else` yazmaktır.

8. `System.out.println("Sonuç: " + 2 + 3);` ve `System.out.println(2 + 3 + " sonuç");` satır-

ları ne yazar? Türkçe ayarlı bir bilgisayarda `System.out.printf("%.1f%n", 2.25);` ne yazar?

Cevap

`+` soldan sağa çalışır. İlk satırda önce `"Sonuç: " + 2` bir metin olur, sonra 3 eklenir: `Sonuç: 23`. İkinci satırda önce `2 + 3` sayı olarak toplanır: `5 sonuç`. `printf` yerel ayarı kullandığı için ondalık ayırıcı virgüldür ve sayı bir basamağa yuvarlanır: `2,3`.

9. "Bir öğrencinin notlarını klavyeden oku, ortalamasını hesapla ve 50'den büyükse 'geçti' yaz" işini tek bir `main` yerine metotlara bölmek isteniyor. Hangi metotlar, hangi parametre ve dönüş tipleriyle yazılabilir?

Cevap

Bir olası tasarım: `int[] notlariOku(Scanner giris, int adet)` okur ve diziyi döndürür; `double ortalama(int[] notlar)` hesaplar ve döndürür; `boolean gectiMi(double ort)` karar verir. Yazdırma işi `main`'de kalır. Her metot tek bir iş yapar, verisini parametrelerle alır ve sonucunu döndürür. `ortalama` ve `gectiMi` klavyeye dokunmadan sınanabilir.

3.2.16 İleri okuma

- A. B. Downey ve C. Mayfield, *Think Java* (2. baskı, O'Reilly, 2019), 4. bölüm: metot yazmayı, parametreleri ve yığın diyagramlarını kısa örneklerle anlatır; çevrim içi olarak ücretsiz okunabilir.
- J. Bloch, *Effective Java* (3. baskı, Addison-Wesley, 2018), "Methods" bölümü: parametrelerin denetlenmesi, metot imzalarının tasarımı ve aşırı yüklemenin ne zaman sorun çıkardığı üzerine deneyimli bir programcının önerileri.
- R. C. Martin, *Clean Code* (Prentice Hall, 2008), 3. bölüm "Functions": kısa metot, tek iş, az parametre ve yan etkisiz tasarım ilkelerini örnek kodlar üzerinde tartışır.
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21, §15.12: derleyicinin bir çağrı için uygun metotları nasıl bulduğunu ve en özel olanı nasıl seçtiğini kesin kurallarla tanımlar.

3.2.17 Kaynaklar

- Oracle, *The Java Tutorials*: "Defining Methods", "Passing Information to a Method or a Constructor".
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21 Edition (§8.4 Method Declarations, §6.3 Scope of a Declaration, §15.12 Method Invocation Expressions).
- P. Deitel ve H. Deitel, *Java: How to Program, Early Objects*, 11. baskı, Pearson, 2017 (Bölüm 6: Methods).
- K. Sierra ve B. Bates, *Head First Java*, 3. baskı, O'Reilly, 2022.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, 2022.
- A. B. Downey ve C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019 (Bölüm 4: Methods and Testing).

- J. Bloch, *Effective Java*, 3. baskı, Addison-Wesley, 2018.
- R. C. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*, Prentice Hall, 2008.

KISIM I · BÖLÜM 3.3

Diziler

Bir sınıfın sınav notları, bir sensörün saniyede verdiği ölçümler ya da bir resmin pikselleri aynı türden pek çok değerden oluşur. Bu değerlerin her biri için ayrı bir değişken açmak hem yorucudur hem de değer sayısı değişince kodu bozar. Dizi bu sorunu çözer: aynı tipten çok sayıda değeri tek bir ad altında, numaralı hücrelerde tutar. Not önce dizi kavramını, indisi ve dizinin bellekteki yerleşimini anlatır. Ardından Java'da dizi tanımlamayı, ilk değer vermeyi, `length`'i, `for-each` döngüsünü ve `java.util.Arrays` sınıfını işler. Dizi değişkeninin bir referans olması, kopyalama ve metoda geçirme bu kısmın en önemli konusudur. Sonraki kesimler sınır dışı hatasını, istatistik hesaplarını (ortalama, ortanca, mod), arama ve sıralama algoritmalarını, maliyetlerini ve çok boyutlu dizileri ele alır. Not, iki boyutlu bir karakter dizisi üzerine kurulan Tic Tac Toe oyunuyla biter.

ÖĞRENME ÇIKTILARI

Kavram
Dizinin aynı tipten, sabit sayıda öğeyi ardışık hücrelerde tuttuğunu ve öğelere 0'dan başlayan indisle erişildiğini açıklamak.

Söz dizimi
Tek ve çok boyutlu dizileri tanımlamak; `new` ve ilklendirici listesiyle oluşturmak; `length` ile gezinmek.

Bellek
Dizi değişkeninin bir referans olduğunu, dizinin heap'te durduğunu göstermek; atama, kopyalama ve metoda geçirmenin sonucunu öngörmek.

Hatalar
`ArrayIndexOutOfBoundsException`'ı ve sık derleme hatalarını tanımak; sınır denetimiyle önlemek.

Algoritmalar
Ortalama, ortanca, mod, doğrusal ve ikili arama, eşleştirme, tersine çevirme, seçimli ve kabarcık sıralamasını yazmak; $O(1)$, $O(\log n)$, $O(n)$ ve $O(n^2)$ maliyetlerini karşılaştırmak.

Uygulama
Matrisleri iç içe döngülerle doldurmak, yazdırmak, transpozunu almak ve `char[][]` ile Tic Tac Toe oyunu kurmak.

3.3.1 Dizi kavramı

Şimdiye kadarki programlarda her değişken tek bir değer tuttu: bir sayı, bir karakter ya da bir dizgi. Gerçek problemlerde ise değerler çoğu zaman topluluk hâlinde gelir. Elli öğrencinin notu, bir gün boyunca dakikada bir ölçülen sıcaklık ya da bir devredeki düğüm gerilimleri böyle topluluklardır. Bu kesim diziyi tanımlar, neden gerektiğini gösterir ve dizinin bellekte nasıl durduğunu anlatır.

3.3.1.1 Dizi nedir?

Tanım 3.3.1

Dizi (*array*), aynı tipten, sabit sayıda değeri tek bir ad altında, ardışık hücrelerde tutan yapıdır. Dizideki her değere **öğe** (*element*), öğenin dizideki konum numarasına **indis** (*index*) denir. İlk öğenin indisi 0'dır; n öğeli bir dizinin son indisi $n - 1$'dir.

Dizi, aynı adla tanımlanan ve birden çok değer tutabilen bir değişken gibi düşünülebilir. Tek bir değişkenin bir adı ve bir değeri vardır. Dizinin ise tek bir adı, ama çok sayıda değeri vardır. Değerler birbirinden indisle ayrılır. Bir öğeye erişmek için dizinin adı yazılır, ardından köşeli parantez içinde indis gelir: `dizi[0]`, `dizi[1]`, `dizi[2]`.

İki gündelik benzetme kavramı somutlaştırır. Birincisi bir apartmandır. Apartmanın tek bir adresi vardır; bu adres dizinin adına karşılık gelir. Daireler ise numaralıdır ve “3 numara” demek doğru daireyi bulmaya yeter. Kimse “sağdan ikinci, üst kattaki kapı” gibi bir tarif vermez. İkincisi bir yumurta kolisidir. Kolide kaç hücre olduğu baştan bellidir ve koli sonradan büyümez. Her hücreye yalnız aynı türden bir şey konur. Dizinin iki temel özelliği, sabit boyut ve tek tip, bu benzetmede görünür.

İndisin 0'dan başlaması yeni başlayanlara garip gelir. Gündelik hayatta sayma 1'den başlar: birinci öğrenci, ikinci öğrenci. Java, C ve onlardan türeyen diller ise indisi bir *uzaklık* olarak görür. İndis, öğenin dizinin başından kaç hücre ötede olduğunu söyler. İlk öğe başlangıçtan 0 hücre uzaktadır, dolayısıyla indisi 0'dır. Bu bakış, öğenin bellekteki adresinin hesabını da kolaylaştırır: `i`. öğe, başlangıç adresinden `i` hücre ötededir.

Örnek 3.3.1

`dizi = [10, 20, 30, 40, 50]` olsun. Dizide 5 öğe vardır, dolayısıyla geçerli indisler 0, 1, 2, 3 ve 4'tür.

`dizi[0]` ilk öğedir ve değeri 10'dur. `dizi[4]` son öğedir ve değeri 50'dir. Üçüncü öğe `dizi[2]` 'dir; indis, sıra numarasının bir eksiğidir. `dizi[5]` diye bir öğe yoktur. Böyle bir erişim Java'da çalışma zamanında hata verir.

3.3.1.2 Neden dizi?

Elli öğrencinin notlarının ortalaması hesaplanacak olsun. Dizi kullanmadan her not için ayrı bir değişken gerekir:

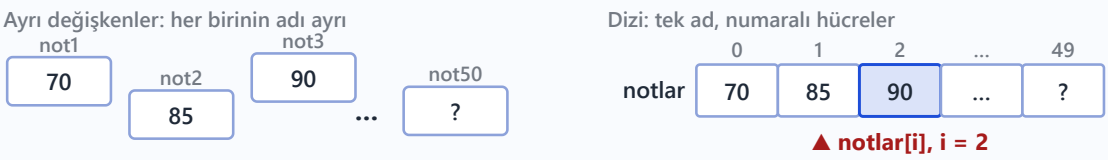
```
int not1 = 70;
int not2 = 85;
int not3 = 90;
// ... not50'ye kadar
int toplam = not1 + not2 + not3 /* + ... */;
```

Bu yaklaşımın üç sorunu vardır. Birincisi, toplam ifadesine elli adın tek tek yazılması gerekir. İkincisi, sınıfa bir öğrenci katılırsa programın değişmesi gerekir: yeni bir değişken açılır ve toplama eklenir. Üçüncüsü, değişkenlerin adları yalnız programcı için anlamlıdır. `not1` ile `not2` arasındaki "sıradalık" ilişkisini Java bilmez. Bir döngü `not` ve `i` 'yi birleştirerek `not3` adını üretemez.

Dizi bu üç sorunu birlikte çözer:

```
int[] notlar = new int[50];
// ... notlar doldurulur
int toplam = 0;
for (int i = 0; i < notlar.length; i++) {
    toplam += notlar[i];
}
```

Burada `notlar` tek bir addır ve elli hücreyi birden temsil eder. Döngü `i` 'yi 0'dan 49'a kadar sayar ve her turda `notlar[i]` ile bir hücreye bakar. Aynı kod 5 not için de 5000 not için de çalışır; yalnız dizinin boyutu değişir. Döngü koşulundaki `notlar.length` dizinin kaç hücresi olduğunu söylediği için koşulun elle değiştirilmesi de gerekmez.

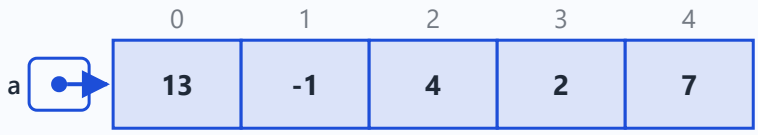


Şekil 3.3.1: Solda adları birbirinden bağımsız ayrı değişkenler, sağda tek adlı ve numaralı hücreli bir dizi. `notlar[i]` yazıldığında hangi hücreye bakılacağını `i` 'nin o anki değeri belirler; şekilde `i` = 2 iken 90 değeri okunur.

Dizinin asıl gücü Şekil 3.3.1'de görülür. İndis sabit bir sayı olmak zorunda değildir; bir değişken ya da bir ifade olabilir. `notlar[i]` yazıldığında hangi hücreye bakılacağı derleme sırasında değil, çalışma zamanında `i`'nin değerine göre belirlenir. Bu yüzden tek bir satır kod, döngü içinde bütün hücrelere sırayla uygulanabilir.

3.3.1.3 Bellekte bir dizi

Dizinin öğeleri bellekte rastgele yerlere dağılmaz. Hücreler yan yana, **ardışık** olarak yerleşir. Şekil 3.3.2 beş öğeli bir `int` dizisini gösterir. Her hücrenin üstünde indisi, içinde değeri vardır.



Şekil 3.3.2: Beş öğeli `a` dizisi. Hücreler bellekte yan yanadır; indisler 0'dan 4'e gider. `a` değişkeni hücrelerin kendisini değil, dizinin başlangıcını gösteren bir referansı tutar.

indis	0	1	2	3	4
öğe	13	−1	4	2	7

Tablo 3.3.1: `a` dizisinin indisleri ve öğeleri.

Tablo 3.3.1'e göre `a[1]` = −1, `a[4]` = 7 ve `a.length` = 5'tir. `a[5]` diye bir hücre yoktur; geçerli indisler 0…4'tür. Bu ayrım, dizilerle ilgili hataların çoğunun kaynağıdır. Dizinin *uzunluğu* hücre sayısıdır; *son indis* ise uzunluğun bir eksiğidir.

Şekildeki ok da önemlidir. `a` adlı değişkenin içinde beş sayı yoktur. Değişken, beş sayının durduğu bellek bölgesini gösteren bir **referans** tutar. Bu ayrıntı ilk bakışta önemsiz görünür, ama dizi ataması-nın, kopyalamanın ve metoda geçirmenin nasıl çalıştığını belirler.

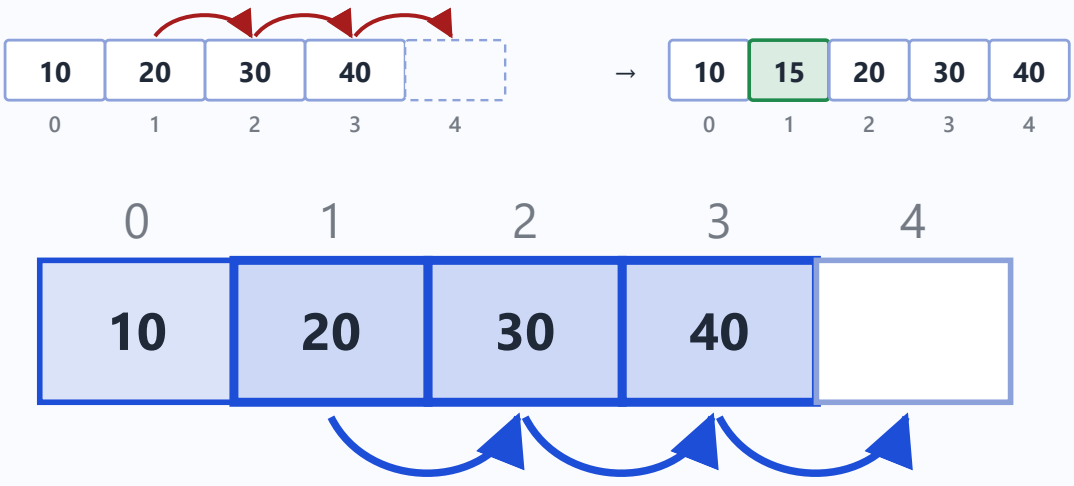
3.3.1.4 Avantajlar ve dezavantajlar

Ardışık yerleşim, dizinin hem en büyük üstünlüğünün hem de en büyük kısıtının kaynağıdır.

Üstünlükler şunlardır. Dizi verileri düzenli ve ardışık biçimde saklar. İndisle erişim hızlıdır: `dizi[i]` her `i` için aynı sürede okunur. İlk öğeye erişmek ile milyonuncu öğeye erişmek arasında fark yoktur. Veriler gruplandığı için döngülerle kolayca işlenir. Bellek kullanımı da yalındır: dizi yalnız öğeleri ve uzunluğunu tutar. Bağlı liste gibi yapılarda her öğenin yanında bir de sonraki öğenin adresi saklanır; dizide bu ek yük yoktur.

Kısıtlar da aynı yerleşimden doğar. Birincisi, dizi **sabit boyutludur**: oluşturulduktan sonra boyutu değiştirilemez. Yer kalmadığında yeni bir öğe eklemek için daha büyük yeni bir dizi oluşturulur ve eski öğeler oraya kopyalanır. İkincisi, ortaya ekleme ya da ortadan silme öğelerin kaydırılmasını ge-

rektirir. Üçüncüsü, dizi yalnız tek tipten öğe tutar.



Şekil 3.3.3: Dizinin ortasına ekleme. (a) Son hücresi boş olan diziye 15, indis 1'e eklenir; 20, 30 ve 40 birer hücre sağa kayar ve dizi `[10, 15, 20, 30, 40]` olur. (b) Aynı kaydırma oklarla: her öğe sağındaki hücreye taşınır.

Şekil 3.3.3 ekleme işleminin maliyetini gösterir. 15 değeri indis 1'e konacaksa indis 1'den sonraki bütün öğeler bir hücre sağa taşınmalıdır. Kaydırmanın yönü önemlidir. Soldan sağa kaydırılırsa önce 20, 30'un üzerine yazılır ve 30 kaybolur. Bu yüzden sondaki öğeden başlanır:

Örnek 3.3.2

`int[] d = {10, 20, 30, 40, 0};` dizisinde son hücre boştur. 15 değeri indis 1'e eklensin.

`for (int i = 3; i >= 1; i--) d[i + 1] = d[i];` döngüsü önce `d[4] = d[3]` (40), sonra `d[3] = d[2]` (30), en son `d[2] = d[1]` (20) atamalarını yapar. Ardından `d[1] = 15;` yazılır.

Sonuç `[10, 15, 20, 30, 40]` olur. Eklenen konumdan sonra *k* öğe varsa *k* kaydırma gerekir. En kötü durumda, başa eklemede, bütün öğeler kayar; bu yüzden ekleme O(n) iş ister.

Boyutu çalışma sırasında değişen listeler için Java'da `ArrayList` sınıfı vardır. `ArrayList` içeride yine bir dizi kullanır. Dizi dolunca daha büyük bir dizi oluşturur ve öğeleri oraya kopyalar. Programcı bu işi görmez, ama maliyet ortadan kalkmaz; yalnız gizlenir. `ArrayList` ve öteki koleksiyonlar `java.util` paketindedir.

3.3.1.5 Dizi türleri ve kullanım alanları

Diziler boyut sayısına göre üç gruba ayrılır. **Tek boyutlu** dizide öğeler tek bir satırda durur; `int[] notlar` böyledir. **İki boyutlu** dizide öğeler satır ve sütunlara yerleşir; bu bir matristir ve `int[][] tablo` biçiminde yazılır. **Çok boyutlu** dizi daha fazla köşeli parantezle yazılır; `int[][][] kup` üç boyutludur.

Dizi, programlama dillerinde en yaygın veri yapısıdır. Elektrik-elektronik mühendisliğinde sık karşılaşılan üç kullanım şunlardır:

- **Ölçümler.** Bir sensörden saniyede alınan örnekler `double[]` dizisinde tutulur. Sayısallaştırılmış bir ses sinyali de böyledir. 44,1 kHz ile örneklenen bir saniyelik ses, 44.100 öğeli bir dizidir.
- **Matrisler.** Devre analizindeki düğüm denklemleri, doğrusal denklem sistemleri ve dönüşüm matrisleri iki boyutlu dizilerle ifade edilir.
- **Resimler.** Gri tonlu bir resim, `int[yukseklik][genislik]` biçiminde bir piksel ızgarasıdır. Renkli resim bir boyut daha ister: her piksel için kırmızı, yeşil ve mavi kanalları.

Örnek 3.3.3

Bir sıcaklık sensörü 1000 ölçüm veriyor ve ölçümler `double[] olcum = new double[1000];` dizisinde tutuluyor. Öğeler kaç bayt yer kaplar?

Java'da her `double` 8 bayttır. $1000 \cdot 8 = 8000$ bayt, yani yaklaşık 8 KB gerekir. Aynı ölçümler `int` olsaydı her biri 4 bayt tutar, toplam 4000 bayt olurdu. JVM dizinin uzunluğu ve tipi için birkaç bayt daha ayırır; bu ek, öğelerin yanında küçüktür.

Bu hesap tasarımda işe yarar. Bir milyon öğeli `double` dizisi 8 MB, 1000 × 1000'lik bir `double` matrisi de yine 8 MB tutar. Boyut sayısı arttıkça bellek hızla büyür.

3.3.2 Java'da diziler

Dizi kavramı bütün dillerde benzerdir, ama ayrıntılar dilden dile değişir. Java'da dizinin nasıl tanımlandığı, hücrelerin hangi değerle başladığı, uzunluğun nereden okunduğu ve dizi değişkeninin ne tuttuğu bu kesimin konusudur. Kesim, dizilerle sık kullanılan `java.util.Arrays` sınıfıyla biter.

3.3.2.1 Dizi tanımlama ve varsayılan değerler

Java'da bir dizi şu genel kalıpla tanımlanır ve oluşturulur:

```
VeriTipi[] diziAdi = new VeriTipi[diziBoyutu];
```

Kalıbın üç parçası vardır. `VeriTipi`, dizide saklanacak değerlerin tipidir: `int`, `double`, `char`, `String` ya da herhangi bir sınıf. `diziAdi`, dizinin adıdır ve değişken adlandırma kurallarına uyar. `diziBoyutu` dizinin kaç öğe içereceğini söyler. Boyut bir sabit olabileceği gibi, çalışma zamanında hesaplanan bir `int` de olabilir.

Satırda aslında iki ayrı iş yapılır. Sol taraf, `VeriTipi[] diziAdi`, bir dizi *değişkeni* tanımlar. Sağ taraf, `new VeriTipi[diziBoyutu]`, belleğin heap adı verilen bölgesinde diziye yer ayırır. Eşittir işareti de bu yeni dizinin referansını değişkene yazar. Değişken ve dizi ayrı şeylerdir. Bu ayrım ilerideki bütün referans konularının temelidir.

```
int[] sayilar = new int[5];      // 5 tamsayı
double[] olcum = new double[100]; // 100 ondalık
String[] isimler = new String[30];
int n = 12;
char[] harfler = new char[n];     // boyut değişkenden
int dizi2[] = new int[3];         // C tarzı, geçerli ama önerilmez
```

Bu satırların hepsi derlenir. `sayilar` beş, `olcum` yüz, `isimler` otuz öğelidir. `harfler` dizisinin boyutu bir değişkenden gelir: `n` 12 olduğu için dizi 12 öğelidir. Son satırdaki `int dizi2[]` yazımı C dilinden kalmadır. Java bunu kabul eder, ama köşeli parantezin tipe bitişik yazılması önerilir: `int[]` "int dizisi" diye okunur ve tipin tamamı tek yerde görünür.

Boyut, oluşturma anında belli olmalıdır. Sonradan değişmez. `new int[n]` yazıldıktan sonra `n`'nin değeri değişse de dizinin boyutu aynı kalır. Boyut negatif olursa dizi oluşturulamaz; program `NegativeArraySizeException` fırlatır:

```
Exception in thread "main" java.lang.NegativeArraySizeException: -2
        at NegBoyut.main(NegBoyut.java:4)
```

Boyutu 0 olan bir dizi ise geçerlidir. `new int[0]` hiç hücresi olmayan bir dizidir ve `length`'i 0'dır. Böyle bir dizi, "hiç sonuç yok" durumunu `null` kullanmadan anlatmak için işe yarar.

Varsayılan değerler

`new` ile oluşturulan dizinin hücreleri boş kalmaz. Java her hücreye tipine göre bir **varsayılan değer** (*default value*) yazar.

Tip	Varsayılan değer
`int`, `long`, `short`, `byte`	`0`
`double`, `float`	`0.0`
`boolean`	`false`
`char`	`'\u0000'` (kodu 0 olan karakter)

Tip	Varsayılan değer
`String` ve bütün nesne tipleri	`null`

Tablo 3.3.2: Dizi öğelerinin varsayılan değerleri.

Varsayılan değerler yerel değişkenlerle karşılaştırıldığında ilginçtir. `int x;` diye tanımlanan bir yerel değişken değer almadan okunursa derleyici `variable x might not have been initialized` hatası verir. Dizi hücreleri ise hiçbir zaman "değersiz" değildir. `new int[5]` beş tane 0 içerir ve hemen okunabilir.

`char` dizisinin varsayılan değeri ayrıca dikkat ister. `'\u0000'` boşluk karakteri değildir; kodu 0 olan, ekranda görünmeyen bir karakterdir. Boşluk karakterinin kodu 32'dir. Boş hücreleri boşlukla göstermek isteyen bir program, `char` dizisini önce `' '` ile doldurmalıdır. `String[]` dizisinin hücreleri de boş dizgi (`""`) değil, `null` 'dır. `null` hiçbir nesneyi göstermeyen referanstır ve üzerinde bir metot çağrılırsa `NullPointerException` oluşur.

Sık yapılan hata

`String[] isimler = new String[30];` satırından sonra `isimler[0].length()` çağrılır. Hücre `null` olduğu için program `NullPointerException` ile durur. `new String[30]` otuz dizgi oluşturmaz; otuz dizgiyi gösterebilecek otuz boş referans oluşturur. Her hücreye ayrıca bir dizgi atanmalıdır.

3.3.2.2 Erişim ve `length`

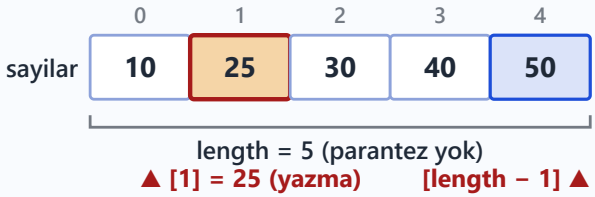
Dizi öğelerine indisle erişilir ve indis 0'dan başlar. `dizi[i]` ifadesi hem okumak hem yazmak için kullanılır. Eşittir işaretinin sağında okur, solunda yazar. Dizinin boyutu `length` özelliğinden alınır.

```
int[] sayilar = {10, 20, 30, 40, 50};
int ilkEleman = sayilar[0];    // İlk elemanı alır
int ikinciEleman = sayilar[1]; // İkinci elemanı alır
int diziBoyutu = sayilar.length; // 5
System.out.println(ilkEleman + " " + ikinciEleman
        + " " + diziBoyutu);
sayilar[1] = 25;               // yazma
int son = sayilar[sayilar.length - 1]; // 50
```

Program şu satırı yazar:

```
10 20 5
```

`sayilar[0]` ilk öğeyi (10), `sayilar[1]` ikinci öğeyi (20) okur. `sayilar.length` 5'tir. `println`'daki `" "` dizgileri sayıları birbirinden ayırır; aralarında dizgi olmasaydı `ilkEleman + ikinciEleman` toplama yapar ve 30 yazardı. `sayilar[1] = 25;` satırı ikinci hücreye yazar. Son satır ise son öğeyi okumanın genel kalıbıdır: `sayilar[sayilar.length - 1]`. Dizi kaç öğeli olursa olsun, bu ifade son öğeyi verir.



Şekil 3.3.4: `sayilar` dizisi yazma işleminden sonra. `sayilar[1] = 25` ikinci hücreyi değiştirmiştir. `length` = 5 hücre sayısıdır; son hücre `[length - 1]`, yani `[4]`'tür.

`length` bir metot değil, dizinin bir özelliğidir (alanıdır). Bu yüzden parantezsiz yazılır: `sayilar.length`. `String` sınıfındaki `length()` ise bir metottur ve parantezle yazılır. Bu iki yazım kolayca karışır:

```
E_LengthParen.java:4: error: cannot find symbol
        System.out.println(a.length());
                           ^
  symbol:   method length()
  location: variable a of type int[]
1 error
```

İleti, `int[]` tipindeki `a` değişkeninde `length()` adlı bir metot bulunamadığını söyler. Tersi de hatalıdır: bir `String` değişkeni için `s.length` yazılırsa derleyici `variable length` bulamaz. Kısa bir kural akılda tutulabilir: dizide parantez yok, dizgide parantez var.

Dizinin `length` alanı `final`'dır, yani değiştirilemez. `sayilar.length = 10;` satırı derlenmez. Dizinin boyutunun sabit olması bu kuralla da güvenceye alınmıştır.

3.3.2.3 İlk değer verme

Dizinin öğeleri baştan biliniyorsa her hücreyi tek tek atamaya gerek yoktur. Değerler süslü parantez içinde, virgülle ayrılarak verilir. Bu yazıma **ilklendirici listesi** (*array initializer*) denir.

```
int[] sayilar = {10, 20, 30, 40, 50};      // 5 elemanlı
int[] n = {1, 2, 3, 4, 5};   // 5 değer olduğundan n 5 elemanlı bir dizi olur
int[] m = new int[]{1, 2, 3}; // uzun biçim: her yerde kullanılabilir
String[] gunler = {"Pzt", "Sal", "Çar"};
char[] sesli = {'a', 'e', 'ı', 'i', 'o', 'ö', 'u', 'ü'};
```

İlklendirici listesi kullanıldığında boyut yazılmaz. Boyutu listedeki değer sayısı belirler: `n` dizisinde beş değer vardır, bu yüzden `n.length` 5'tir. `gunler` üç, `sesli` sekiz öğelidir. Java karakterleri Unicode olarak tuttuğu için `'ı'`, `'ö'` ve `'ü'` gibi Türkçe harfler de geçerli `char` değerleridir.

İlklendirici listesinin iki biçimi vardır. Kısa biçim, `{1, 2, 3}`, **yalnız tanımlama satırında** kullanılabilir. Uzun biçim, `new int[]{1, 2, 3}`, her yerde geçerlidir: atamada, metot çağrısında ve `return`'de. Dizi önce tanımlanıp sonra kısa biçimle doldurulmak istenirse derleyici hata verir:

```
E_KisaSonra.java:4: error: illegal start of expression
        a = {1, 2, 3};
            ^
E_KisaSonra.java:4: error: not a statement
        a = {1, 2, 3};
             ^
E_KisaSonra.java:4: error: ';' expected
        a = {1, 2, 3};
              ^
3 errors
```

İletideki üç hatadan asıl olanı ilkidir: `illegal start of expression`. Derleyici `=` işaretinden sonra bir ifade bekler ve süslü parantezi bir ifadenin başı olarak tanımaz. Sonraki iki ileti bu ilk hatanın yol açtığı karışıklıktır. Derleyici bir hatadan sonra çoğu zaman bir dizi ek ileti üretir; ilk ileti düzeltilince ötekiler de kaybolur. Doğru yazım `a = new int[]{1, 2, 3};`'tür.

Boyut ile listeyi birlikte vermek de yasaktır. Liste zaten boyutu belirlediği için Java iki bilginin çelişme olasılığına izin vermez:

```
E_IkisiBirden.java:3: error: array creation with both dimension expression and
initialization is illegal
        int[] a = new int[3]{1, 2, 3};
                            ^
1 error
```

Hiçbirini vermemek de hatadır. `new int[]` yazılıp ardından liste gelmezse derleyici dizinin kaç hücreli olacağını bilemez:

```
E_BoyutYok.java:3: error: array dimension missing
        int[] a = new int[];
                          ^
1 error
```

Listedeki değerler dizinin tipine uymalıdır. `int` dizisine `2.5` konamaz, çünkü `double`'dan `int`'e dönüşüm kesirli kısmı atar ve Java bunu kendiliğinden yapmaz:

```
E_Lossy.java:3: error: incompatible types: possible lossy conversion from double to int
        int[] a = {1, 2.5, 3};
                      ^
1 error
```

Ters yön sorun çıkarmaz. `double[] d = {1, 2, 3};` geçerlidir; `int` değerler kayıpsız biçimde `double`'a genişler ve dizi `1.0, 2.0, 3.0` tutar. `double[] c = {-40, 0, 25, 100};` listesi de bu yüzden derlenir.

Sık yapılan hata

C dilinden gelen alışkanlıkla `int a[3] = {1, 2, 3};` yazılır. Java'da boyut değişkenin tipinde yer almaz; javac `']' expected` iletisini verir. Java'da tip yalnız "int dizisi"dir; kaç hücre olduğu nesnenin özelliğidir, değişkenin değil. Doğrusu `int[] a = {1, 2, 3};` 'tür.

3.3.2.4 Kod izleme: DiziOrnegi

Aşağıdaki program bir diziyi tanımlar, hücrelerine tek tek değer atar, iki öğeyi okur ve son olarak bütün öğeleri bir döngüyle yazdırır. Kısa olmasına karşın dizinin bütün temel işlemlerini içerir.

```
public class DiziOrnegi {
    public static void main(String[] args) {
        // Bir diziyi tanımlama: 5 elemanlı bir tamsayı dizisi
        int[] sayilar = new int[5];
        // Dizinin elemanlarına değer atama
        sayilar[0] = 10;
        sayilar[1] = 20;
        sayilar[2] = 30;
        sayilar[3] = 40;
        sayilar[4] = 50;
        // Dizinin elemanlarına erişme
        System.out.println("birinci eleman: " + sayilar[0]);
        System.out.println("ucuncu eleman: " + sayilar[2]);
        // Dizinin tüm elemanlarını döngü içerisinde yazdırma
        System.out.print("Elemanlar: ");
        for (int i = 0; i < sayilar.length; i++) {
            System.out.print(sayilar[i] + " ");
        }
    }
}
```

Programın çıktısı:

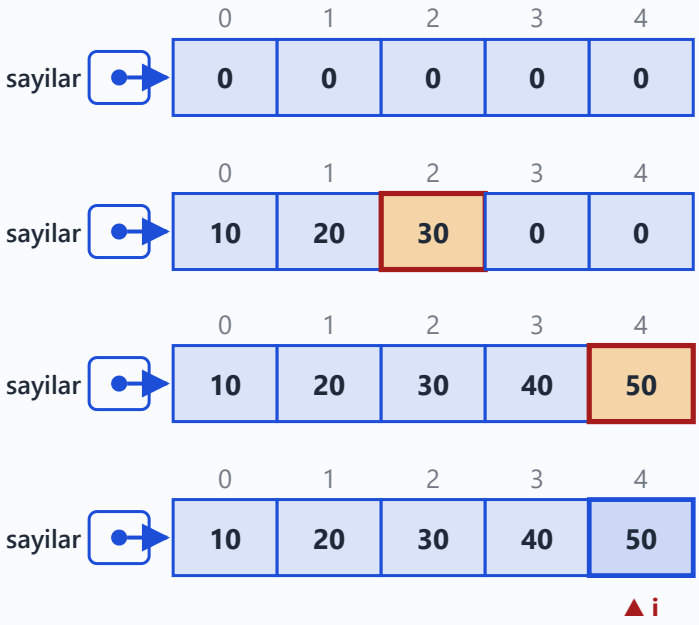
```
birinci eleman: 10
ucuncu eleman: 30
Elemanlar: 10 20 30 40 50
```

`new int[5]` heap'te beş hücrelik bir dizi oluşturur. Her hücre varsayılan değer 0 ile başlar. `sayilar` değişkeni bu dizinin referansını tutar. Ardından beş atama satırı hücreleri sırayla doldurur. Birinci öğenin indisi 0'dır, bu yüzden `sayilar[0]` 10'u okur. Üçüncü öğe `sayilar[2]`'dir; indis, sıra numarasının bir eksiğidir.

Son kısımda `print` kullanılır. `print`, `println`'dan farklı olarak alt satıra geçmez; bu yüzden bütün öğeler aynı satırda, aralarında birer boşlukla yazılır. Satırın sonunda da bir boşluk kalır. Döngü `i`'yi 0'dan başlatır ve `i < sayilar.length` koşulu doğru oldukça devam eder.

`i`	`i < sayilar.length`	Okunan hücre	Ekrana eklenen
0	0 < 5 doğru	`sayilar[0]` = 10	`10`
1	1 < 5 doğru	`sayilar[1]` = 20	`20`
2	2 < 5 doğru	`sayilar[2]` = 30	`30`
3	3 < 5 doğru	`sayilar[3]` = 40	`40`
4	4 < 5 doğru	`sayilar[4]` = 50	`50`
5	5 < 5 yanlış	—	döngü biter

Tablo 3.3.3: DiziOrnegi programındaki yazdırma döngüsünün izlemesi.



Şekil 3.3.5: DiziOrnegi izlemesinden dört an. (a) `new int[5]` sonrası: beş hücre de 0'dır. (b) Üç atamadan sonra: 10, 20, 30. (c) Beş atama bitti. (d) Yazdırma döngüsünde `i` = 4: son geçerli hücre okunur. Bir sonraki

turda `i` = 5 olur ve koşul yanlış çıkar.

Tablo 3.3.3 döngünün altı kez koşul denetlediğini, gövdenin ise beş kez çalıştığını gösterir. Son denetimde `i` 5'tir ve 5 < 5 yanlıştır. Son geçerli indis `length - 1` = 4'tür.

`for (int i = 0; i < dizi.length; i++)` kalıbı bir dizinin bütün indisleri üzerinde 0'dan `length - 1`'e kadar gezinir. Bu kalıp o kadar sık kullanılır ki ezbere yazılır. Kalıbın iki ayrıntısı önemlidir: başlangıç 0'dır ve karşılaştırma `<`'dir. `<=` yazmak sık yapılan bir hatadır ve son turda dizinin dışına çıkar.

Döngünün maliyeti de açıktır. Gövde her öğe için bir kez çalışır; *n* öğeli dizide *n* tur yapılır. İş miktarı dizinin boyutuyla doğru orantılı büyür. Bu maliyete O(n) denir.

3.3.2.5 `for-each` döngüsü

Bir dizinin bütün öğelerini sırayla işlemek o kadar yaygındır ki Java bunun için ayrı bir döngü sunar. Bu döngüye **gelişmiş** `for` (*enhanced for*) ya da kısaca `for-each` **döngüsü** denir.

Tanım 3.3.2

`for (tür eleman : dizi) { … }` biçimindeki `for-each` **döngüsü**, her turda dizinin sıradaki öğesinin bir kopyasını `eleman` değişkenine koyar ve gövdeyi çalıştırır. Döngü dizideki öğe sayısı kadar tur yapar. İndis yazılmaz.

İki nokta üst üste işareti okunuşu kolaylaştırır. İki nokta üst üste "içindeki her … için" diye okunur: `for (int not : notlar)`, "`notlar` içindeki her `not` için" demektir. Döngü değişkeninin tipi dizinin öğe tipiyle uyumlu olmalıdır. `double[]` dizisinde `int` değişkenle gezinilemez, çünkü her turda bir daraltma dönüşümü gerekir:

```
E_ForEachTip.java:4: error: incompatible types: possible lossy conversion from double to
int
        for (int x : d) {
                     ^
1 error
```

Aşağıdaki program aynı dizide önce klasik `for`, sonra `for-each` ile gezinir:

```
int[] notlar = {70, 85, 90};
// Klasik for: indis elimizde
for (int i = 0; i < notlar.length; i++) {
    System.out.println(i + ". not: " + notlar[i]);
}
// for-each: yalnızca değerler
int toplam = 0;
for (int not : notlar) {
    toplam += not;
}
System.out.println("Toplam: " + toplam);
```

```
0. not: 70
1. not: 85
2. not: 90
Toplam: 245
```

Klasik `for` döngüsünde indis elimizdedir; çıktıdaki `0.`, `1.`, `2.` sıra bilgisi bu sayede yazılır. `for-each` döngüsünde indis yoktur; yalnız değerler gelir. Toplam hesabı için indis gerekmediği için ikinci döngü daha kısadır ve daha az hataya açıktır.

`for-each` döngüsünün üç özelliği vardır. Birincisi, indis yazılmadığı için sınır dışı hatası yapılamaz. Döngü dizinin nerede bittiğini kendisi bilir. İkincisi, döngü yalnız okumak içindir. `not = 0;` yazmak dizideki öğeyi değiştirmez, yalnız kopyayı değiştirir. Üçüncüsü, indise ihtiyaç olan işler için uygun değildir. Diziye yazmak, öğenin konumunu bildirmek ya da iki komşu öğeyi karşılaştırmak indis ister. Bu durumlarda klasik `for` kullanılır.

3.3.2.6 Kod izleme: değer gelir, hücre gelmez

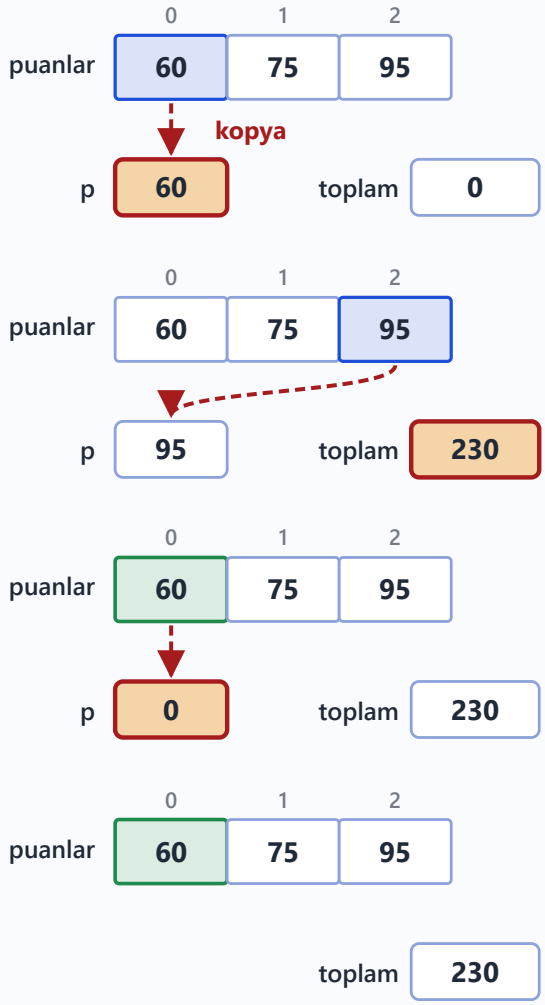
`for-each` döngüsünün okuma amaçlı olduğunu en açık biçimde şu program gösterir:

```
int[] puanlar = {60, 75, 95};
int toplam = 0;
for (int p : puanlar) {
    toplam += p;
}
System.out.println("Toplam: " + toplam);
for (int p : puanlar) {
    p = 0;
}
System.out.println(puanlar[0] + " " + puanlar[2]);
```

Program `Toplam: 230` ve ardından `60 95` yazar. İlk döngü üç puanı toplar. İkinci döngü her öğeyi sıfırlamaya çalışır, ama dizi değişmez.

Döngü	Tur	`p`	Gövde	`toplam`	`puanlar`
—	başlangıç	—	—	0	[60, 75, 95]
1	1	60	`toplam += p`	60	[60, 75, 95]
1	2	75	`toplam += p`	135	[60, 75, 95]
1	3	95	`toplam += p`	230	[60, 75, 95]
2	1	60 → 0	`p = 0`	230	[60, 75, 95]
2	2	75 → 0	`p = 0`	230	[60, 75, 95]
2	3	95 → 0	`p = 0`	230	[60, 75, 95]

Tablo 3.3.4: İki `for-each` döngüsünün izlemesi. `p` her turda sıradaki öğenin kopyasıdır.



Şekil 3.3.6: `for-each` izlemesinden dört an. (a) İlk tur: `puanlar[0]`'ın kopyası `p`'ye gelir, `p` = 60. (b) Son tur sonunda `toplam` = 230. (c) İkinci döngüde `p = 0` yalnız kopyayı değiştirir; dizideki 60 yerinde durur. (d) Döngüler bitince dizi hiç değişmemiştir.

Şekil 3.3.6 nedeni görsel olarak verir. `p`, dizinin hücresi değil, ayrı bir yerel değişkendir. Her turun başında döngü sıradaki hücrenin değerini `p`'ye kopyalar. `p = 0` bu yerel değişkene yazar; hücre ile `p` arasında bir bağ kalmamıştır. Diziye yazmak için klasik `for` ve `puanlar[i] = 0` gerekir.

Örnek 3.3.4

`int[] x = {5, 6, 7};` dizisinin her öğesi ikiyle çarpılsın.

Yanlış yol: `for (int v : x) v = v * 2;`. Turlarda `v` sırasıyla 10, 12 ve 14 olur, ama bu değerler kopyalara yazılır. `Arrays.toString(x)` hâlâ `[5, 6, 7]` verir.

Doğru yol: `for (int i = 0; i < x.length; i++) x[i] = x[i] * 2;`. Bu kez hücrelerin kendisine yazılır ve dizi `[10, 12, 14]` olur.

3.3.2.7 Sıcaklık dönüşümü

Diziler ile `for-each` döngüsü birlikte tablo üretmenin kısa bir yolunu verir. Aşağıdaki program dört Celsius değerini Fahrenhayt ve Kelvin'e çevirir:

```
double[] celsiuslar = {-40, 0, 25, 100};
for (double celsius : celsiuslar) {
    double fahrenheit = (celsius * 1.8) + 32;
    double kelvin = celsius + 273.15;
    System.out.println(celsius + " °C = " + fahrenheit
            + " °F = " + kelvin + " K");
}
```

```
-40.0 °C = -40.0 °F = 233.14999999999998 K
0.0 °C = 32.0 °F = 273.15 K
25.0 °C = 77.0 °F = 298.15 K
100.0 °C = 212.0 °F = 373.15 K
```



Şekil 3.3.7: `celsiuslar` dizisi. `int` olarak yazılan değerler `double`'a genişler ve -40.0, 0.0, 25.0, 100.0 olarak saklanır.

`double[] celsiuslar = {…};` dört değeri bir dizide toplar. Liste `int` sabitlerle yazılmıştır, ama dizi `double` olduğu için değerler `-40.0`, `0.0`, `25.0` ve `100.0` olarak saklanır. Çıktıdaki `.0`'lar bundandır. `for (double celsius : celsiuslar)` "dizideki her öğe için" demektir. Gövde dört kez çalışır

ve tablo tek seferde üretilir.

Tur	`celsius`	`celsius * 1.8 + 32`	`celsius + 273.15`
1	-40.0	-72.0 + 32 = -40.0	233.14999999999998
2	0.0	0.0 + 32 = 32.0	273.15
3	25.0	45.0 + 32 = 77.0	298.15
4	100.0	180.0 + 32 = 212.0	373.15

Tablo 3.3.5: Sıcaklık dönüşümünün izlemesi.

İlk satırdaki `233.14999999999998` bir hata gibi görünür, ama Java doğru çalışır. Ondalıklı sayılar bellekte ikili tabanda saklanır. 273.15 gibi pek çok ondalık sayının ikili tabanda sonlu bir karşılığı yoktur. Sayı en yakın temsil edilebilir değerle saklanır ve işlem sonucunda küçük bir yuvarlama farkı görünür. Aynı nedenle `0.1 + 0.2` de `0.30000000000000004` verir.

Bu fark yazdırırken basamak sayısı sınırlanarak gizlenebilir: `System.out.printf("%.2f K%n", kelvin)`. Burada bir Türkiye ayrıntısı vardır. `printf` sayıyı bilgisayarın dil ayarına göre biçimlendirir. Türkçe ayarlı bir bilgisayarda çıktı `233,15 K` olur; ondalık ayırıcı virgüldür. `println` ise ayardan bağımsız olarak her zaman nokta kullanır.

Sık yapılan hata

Ondalıklı sayıları `==` ile karşılaştırmak. `-40 + 273.15 == 233.15` ifadesi `false` verir. Ondalıklı iki sayının eşitliği, aralarındaki farkın küçük bir eşikten az olmasıyla denetlenir: `Math.abs(a - b) < 1e-9`.

3.3.2.8 Dizi değişkeni bir referanstır

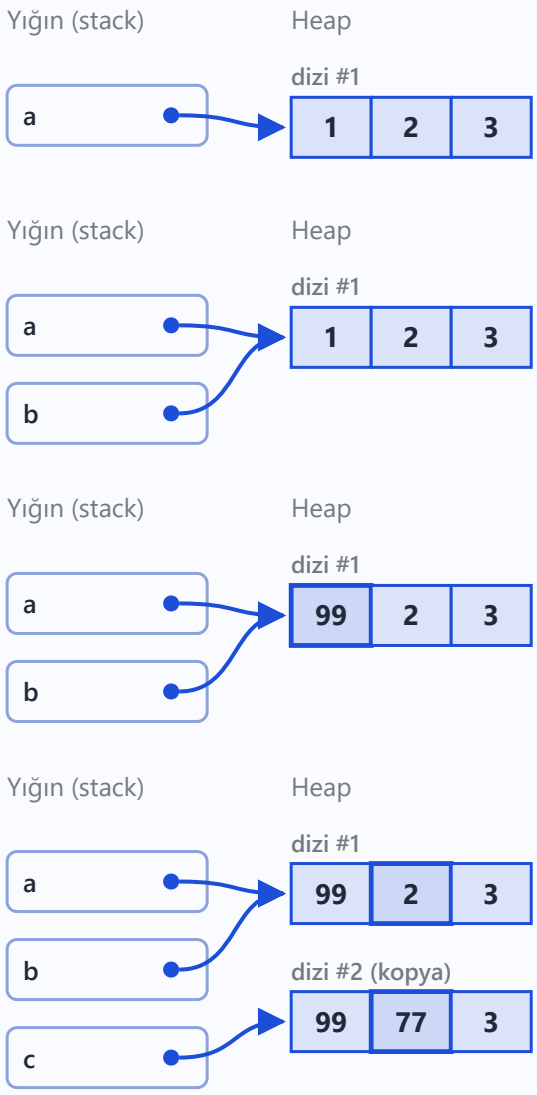
Dizilerin en önemli ve en çok yanlış anlaşılan özelliği budur. `int x = 5;` yazıldığında `x` değişkeninin içinde 5 sayısı durur. `int[] a = {1, 2, 3};` yazıldığında ise `a`'nın içinde üç sayı durmaz. Üç sayı heap'teki bir dizidedir; `a` o dizinin adresini, yani referansını tutar.

Tanım 3.3.3

Referans (*reference*), bir nesnenin bellekteki yerini gösteren değerdir. Diziler, dizgiler ve bütün sınıf nesneleri **referans tipidir**: bu tiplerdeki değişkenler nesnenin kendisini değil, nesneye referansı tutar. İki değişken aynı nesneyi gösteriyorsa biri ötekinin **takma adıdır** (*alias*).

```
int[] a = {1, 2, 3};
int[] b = a;         // kopya değil!
b[0] = 99;
System.out.println(a[0]);
int[] c = a.clone(); // gerçek kopya
c[1] = 77;
System.out.println(a[1] + " " + c[1]);
System.out.println(a == b);
System.out.println(a == c);
```

Program dört satır yazar: `99`, `2 77`, `true` ve `false`. Her satırın nedeni Şekil 3.3.8'de görülür.



Şekil 3.3.8: Dizi referanslarının izlemesi. (a) Heap'te bir dizi oluşur; `a` onun referansını tutar. (b) `b = a` yalnız referansı kopyalar; iki değişken aynı diziyi gösterir. (c) `b[0] = 99` ortak diziyi değiştirdiği için `a[0]` da 99 olur. (d) `clone()` heap'te yeni bir dizi oluşturur; `c[1] = 77` yalnız bu kopyayı değiştirir.

Satırları sırayla izleyelim. `int[] b = a;` yeni bir dizi oluşturmaz. Sağ taraftaki `a`'nın değeri bir referanstır ve bu referans `b`'ye kopyalanır. Artık iki değişken aynı diziyi gösterir. `b[0] = 99` ortak dizinin ilk hücresine yazar. `a[0]` okunduğunda aynı hücreye bakıldığı için 99 gelir.

`a.clone()` ise gerçek bir kopya üretir. Heap'te yeni bir dizi oluşturulur ve öğeler oraya kopyalanır. Kopyalama anında `a[0]` 99 olduğu için kopya `[99, 2, 3]`'tür. `c[1] = 77` yalnız kopyayı değiştirir; asıl dizide `a[1]` hâlâ 2'dir.

Son iki satır `==` işlecinin diziler üzerindeki anlamını gösterir. `==` referansları, yani adresleri karşılaştırır. `a == b` doğrudur, çünkü ikisi aynı diziyi gösterir. `a == c` yanlıştır. `c`'deki dizi o anda `a`'nınkiyle aynı içeriği taşısa bile farklı bir nesnedir. İçerik karşılaştırması için `Arrays.equals` kullanılır.

Diziyi metoda geçirmek

Java'da bütün argümanlar **değer ile aktarılır** (*pass by value*): metoda argümanın bir kopyası gider. Bir `int` geçirildiğinde sayının kopyası gider. Bir dizi geçirildiğinde ise *referansın* kopyası gider. Kopya referans da aynı diziyi gösterir. Bu yüzden metot dizinin hücrelerini değiştirirse değişiklik çağıranın dizisinde görünür.

Örnek 3.3.5

Şu metot ve çağrı verilsin:

```
static void degistir(int[] d, int x) { d[0] = 100; x = 100; }
int[] a = {1, 2, 3}; int x = 1; degistir(a, x);
```

Çağrıdan sonra `a[0]` 100, `x` ise 1'dir. `d`, `a`'daki referansın kopyasıdır ve aynı diziyi gösterir; `d[0] = 100` o dizinin hücresine yazar. Metottaki `x` ise `main`'deki `x`'in kopyasıdır. Kopyaya 100 yazmak asıl değişkeni etkilemez.

Bu kuralın bir sonucu daha vardır. Metot içinde parametreye yeni bir dizi atanırsa, örneğin `d = new int[]{0, 0};`, yalnız yerel kopya yeni diziye bağlanır. Çağıranın değişkeni eski diziyi göstermeye devam eder. Metot bundan sonra `d`'ye ne yazarsa yeni diziye yazar ve çağıran bunu görmez.

Sık yapılan hata

Atamayı kopya sanmak. `int[] yedek = notlar;` yazılıp sonra `notlar` sıralanırsa `yedek` de sıralanmış olur, çünkü ortada tek bir dizi vardır. Gerçek kopya için `notlar.clone()` ya da `Arrays.copyOf(notlar, notlar.length)` kullanılır.

3.3.2.9 `java.util.Arrays` sınıfı

Dizilerle sık yapılan işler için Java hazır metotlar sunar. Bu metotlar `java.util.Arrays` sınıfındadır ve kullanmak için dosyanın başına `import java.util.Arrays;` yazılır. Metotların hepsi `static`'tir; `Arrays.sort(d)` biçiminde sınıf adıyla çağrılır.

```
int[] dizi = {5, 3, 8, 1};
System.out.println(Arrays.toString(dizi));
Arrays.sort(dizi);
System.out.println(Arrays.toString(dizi));
int[] kopya = Arrays.copyOf(dizi, 6);
System.out.println(Arrays.toString(kopya));
int[] dolu = new int[4];
Arrays.fill(dolu, 7);
System.out.println(Arrays.toString(dolu));
int[] x = {1, 2};
int[] y = {1, 2};
System.out.println((x == y) + " " + Arrays.equals(x, y));
System.out.println(Arrays.binarySearch(dizi, 5));
```

```
[5, 3, 8, 1]
[1, 3, 5, 8]
[1, 3, 5, 8, 0, 0]
[7, 7, 7, 7]
false true
2
```

Satır satır: `Arrays.toString` diziyi okunabilir bir metne çevirir. `Arrays.sort` diziyi küçükten büyüğe sıralar. Sıralama **yerinde** yapılır; yeni dizi dönmez, verilen dizi değişir. `Arrays.copyOf(dizi, 6)` 6 hücreli yeni bir dizi oluşturur ve ilk dört hücreye eski öğeleri kopyalar; fazla hücreler varsayılan değer 0'ı alır. Yeni uzunluk eskisinden küçük verilirse dizi kesilir: `copyOf(dizi, 2)` `[1, 3]` verir.

`Arrays.fill(dolu, 7)` bütün hücrelere aynı değeri yazar. `x` ile `y` aynı öğeleri taşıyan iki ayrı dizidir: `x == y` yanlış, `Arrays.equals(x, y)` doğrudur. Son satırdaki `Arrays.binarySearch(dizi, 5)` sıralı dizide 5'i arar ve indisini verir: 2.

Metot	Görevi
`Arrays.toString(d)`	Diziyi `[5, 3, 8, 1]` biçiminde metne çevirir.
`Arrays.deepToString(m)`	İç içe dizileri (matrisleri) metne çevirir.
`Arrays.sort(d)`	Diziyi yerinde, küçükten büyüğe sıralar.
`Arrays.binarySearch(d, x)`	Sıralı dizide `x`'in indisini verir; yoksa negatif bir sayı döner.
`Arrays.copyOf(d, n)`	`n` hücreli yeni dizi oluşturup kopyalar; fazla hücreler varsayılan değeri alır.
`Arrays.fill(d, v)`	Bütün hücrelere `v` yazar.

Metot	Görevi
`Arrays.equals(a, b)`	İki dizinin uzunluğunu ve öğelerini karşılaştırır.

Tablo 3.3.6: `java.util.Arrays` sınıfının sık kullanılan metotları.

Diziyi doğrudan `println` ile yazdırmak beklenen sonucu vermez. `System.out.println(dizi)` öğeleri değil, `[I@1c20c684` gibi bir kimlik yazar. Baştaki `[I` "int dizisi" demektir; `@`'ten sonraki onaltılık sayı nesnenin kimlik kodudur ve her çalıştırmada değişebilir. Öğeleri görmek için `Arrays.toString(dizi)` kullanılır.

İki ayrıntı daha bilinmelidir. Birincisi, dizilerde `indexOf` gibi bir arama metodu yoktur. Sırasız bir dizide arama için döngü yazılır; bu yönteme doğrusal arama denir. İkincisi, `binarySearch` yalnız sıralı dizide doğru çalışır. Aranan değer yoksa sonuç, değerin ekleneceği konumu da kodlayan negatif bir sayıdır. Sıralı `[1, 3, 5, 8]` dizisinde 4 aranırsa sonuç −3'tür. Sıralanmamış dizide sonuç anlamsızdır ve bir uyarı da verilmez.

Metotların maliyeti farklıdır. `toString`, `fill`, `copyOf` ve `equals` her öğeye bir kez dokunduğu için O(n)'dir. `binarySearch` O(log n), `sort` ise ortalamada O(n log n)'dir. `sort`'un bir satırlık yazımı, arkasındaki işin küçük olduğu anlamına gelmez.

3.3.3 Bellek ve hatalar

Dizi kullanan programlarda en sık görülen çalışma zamanı hatası, dizinin sınırları dışına çıkmaktır. Bu hatayı anlamak için önce dizinin bellekte nerede durduğu bilinmelidir. Kesim, Java'nın belleği nasıl böldüğünü, kullanılmayan dizilerin nasıl temizlendiğini ve sınır hatalarının nasıl yakalandığını anlatır.

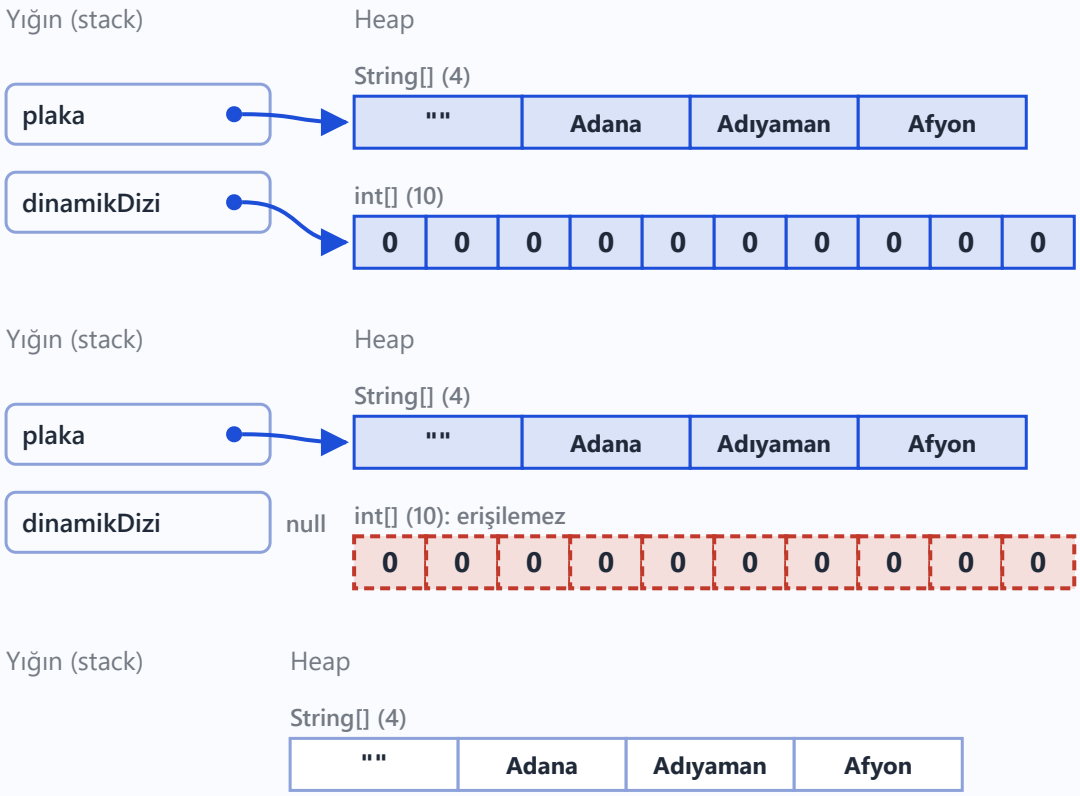
3.3.3.1 Diziler bellekte nerede?

Çalışan bir Java programının belleği iki ana bölgeye ayrılır. **Yığın** (*stack*), metot çağrılarının yerel değişkenlerini tutar. Her metot çağrısı yığında bir **çerçeve** açar; metot bitince çerçeve kalkar ve içindeki değişkenler yok olur. **Heap** ise `new` ile oluşturulan nesnelerin durduğu bölgedir. Heap'teki bir nesne, onu oluşturan metot bittikten sonra da yaşayabilir.

Java'da bütün diziler heap'tedir. Yığında yalnız dizileri gösteren referans değişkenleri durur. C ve C++ dillerinde durum farklıdır. `int a[10];` biçiminde tanımlanan dizi yığında durur. `malloc` ya da `new` ile ayrılan dizi heap'tedir ve işi bitince programcı onu elle serbest bırakır. Java'da bu ayrım yoktur; serbest bırakma da otomatiktir.

```
String[] plaka = {"", "Adana", "Adıyaman", "Afyon"};
System.out.println(plaka[1]);
int[] dinamikDizi = new int[10]; // heap'te yer ayrılır
System.out.println(dinamikDizi.length);
dinamikDizi = null; // referans koparıldı
System.out.println(dinamikDizi == null);
```

Program sırasıyla `Adana`, `10` ve `true` yazar. `plaka` dizisinin ilk hücresi boş dizgidir; böylece plaka kodu 1 olan il `plaka[1]`'de durur ve indis ile plaka kodu örtüşür. Bu, indisin 0'dan başlamasıyla uzlaşmanın pratik bir yoludur. `dinamikDizi`'nin boyutu çalışma zamanında da verilebilirdi; ama dizi oluştuktan sonra boyutu sabittir.



Şekil 3.3.9: Diziler ve çöp toplayıcı. (a) İki dizi de heap'tedir; yığında yalnız referans değişkenleri durur. `String[]` hücreleri de aslında ayrı `String` nesnelerine referanstır. (b) `dinamikDizi = null` bağı koparır; `int` dizisi erişilemez olur. (c) Metot bitince yığın çerçevesi kalkar; başka referansı kalmayan diziler de zamanla toplanır.

`dinamikDizi = null;` satırı diziyi silmez. Yalnız değişken ile dizi arasındaki bağı koparır. Şekil 3.3.9'da görüldüğü gibi dizi heap'te durmaya devam eder, ama artık ona ulaşan bir referans yoktur. Böyle erişilemez nesneleri JVM'in **çöp toplayıcısı** (*garbage collector*) bulur ve belleklerini geri alır.

Tanım 3.3.4

Çöp toplayıcı, hiçbir referansın göstermediği nesneleri bulup kapladıkları belleği geri veren JVM bileşenidir. Java'da `free` ya da `delete` gibi bir serbest bırakma komutu yoktur; bellek yönetimi otomatiktir.

Çöp toplayıcının ne zaman çalışacağı belirli değildir. Bir dizinin erişilemez olması, belleğin hemen boşalacağı anlamına gelmez. JVM uygun bir anda toplama yapar. Programcı açısından önemli olan, gereksiz referansları tutmamaktır. Büyük bir diziyi gösteren bir değişken programın sonuna kadar yaşarsa dizi de yaşar.

Örnek 3.3.6

`int[] p = new int[1000]; int[] q = p; p = new int[5]; q = null;` satırlarından sonra ilk dizi ne olur?

İlk satırda 1000 hücreli dizi oluşur ve `p` onu gösterir. İkinci satırda `q` da aynı diziyi gösterir. Üçüncü satır `p`'yi yeni, 5 hücreli bir diziye bağlar; ilk diziyi hâlâ `q` gösterir. Dördüncü satır bu son bağı da koparır.

Artık 1000 hücreli diziye ulaşan referans kalmamıştır ve çöp toplayıcı onu toplayabilir. Dizi küçülmemiştir: `p = new int[5]` yeni bir dizi oluşturur, eskisini değiştirmez.

3.3.3.2 İndis sınır dışı hatası

Dizinin geçerli indisleri 0 ile `length - 1` arasındadır. Bu aralığın dışındaki bir indisle erişim yapılırsa Java çalışma zamanında `ArrayIndexOutOfBoundsException` fırlatır. Bu istisna, `IndexOutOfBoundsException` sınıfının bir alt sınıfıdır. İstisna yakalanmazsa program o satırda durur.

Tanım 3.3.5

`ArrayIndexOutOfBoundsException`, bir diziye negatif ya da `length`'e eşit veya ondan büyük bir indisle erişildiğinde JVM'in fırlattığı istisnadır. Derleyici indis değerlerini denetlemez; denetim her erişimde çalışma zamanında yapılır.

Bu davranış Java'yı C'den ayırır. C'de `a[10]` gibi bir erişim denetlenmez. Program dizinin hemen arkasındaki belleği okur ya da oraya yazar. Sonuç rastgele bir değer, başka bir değişkenin bozulması ya da çok sonra ortaya çıkan anlaşılmaz bir hatadır. Java ise hatayı tam oluştuğu satırda yakalar ve programı durdurur. Program çöker, ama sessizce yanlış sonuç üretmez.

```java
int[] dizi = {10, 20, 30, 40, 50};
int index = 5;
// Dizi uzunluğunu kontrol etme
if (index >= 0 && index < dizi.length) {
    int deger = dizi[index];
    System.out.println("Değer: " + deger);
} else {
    System.out.println("Geçersiz indis: " + index);
}
// Kontrolsüz erişim:
System.out.println(dizi[index]);
```

```
Geçersiz indis: 5
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds
for length 5
        at SinirDisi.main(SinirDisi.java:13)
```

Programın ilk kısmı güvenli erişimin kalıbını gösterir: indis kullanılmadan önce `index >= 0 && index < dizi.length` koşuluyla denetlenir. 5 geçerli bir indis olmadığı için `else` dalı çalışır ve `Geçersiz indis: 5` yazılır. Son satır ise aynı indisi denetimsiz kullanır ve istisna oluşur.

İstisna iletisi hatayı bulmak için gereken her şeyi söyler. `Index 5` hatalı indistir. `out of bounds for length 5` dizinin uzunluğudur. `at SinirDisi.main(SinirDisi.java:13)` satırı hatanın hangi sınıfta, hangi metotta ve dosyanın kaçıncı satırında oluştuğunu verir. Bir sınır hatasıyla karşılaşıldığında ilk bakılacak yer bu satırdır.

Negatif indis de aynı istisnayı doğurur. Bazı dillerde `d[-1]` son öğe anlamına gelir; Java'da gelmez:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index -1 out of
bounds for length 3
        at Negatif.main(Negatif.java:4)
```

Sık yapılan hata

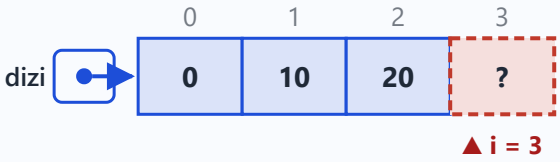
Son indisi `length` sanmak. `a[a.length]` her zaman sınır dışıdır; son öğe `a[a.length - 1]`'dir. Aynı hata döngü koşulunda `i <= a.length` olarak görünür. Üç öğeli bir dizide ileti `Index 3 out of bounds for length 3` olur: hatalı indis ile uzunluk eşittir.

3.3.3.3 Klasik hata: `<=` ile döngü

Sınır dışı hatasının en yaygın kaynağı döngü koşuludur:

```
int[] dizi = new int[3];
for (int i = 0; i <= dizi.length; i++) {
    dizi[i] = i * 10;
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds
for length 3
        at SinirDongu.main(SinirDongu.java:5)
```



Şekil 3.3.10: `i` = 3 olduğunda koşul 3 <= 3 doğru kalır ve var olmayan dördüncü hücreye, `dizi[3]` 'e yazılmaya çalışılır.

Döngü `i` = 0, 1, 2 için hücrelere 0, 10 ve 20 yazar. `i` 3 olduğunda `3 <= 3` doğrudur ve gövde bir kez daha çalışır. `dizi[3]` yoktur ve istisna fırlatılır. Doğru koşul `i < dizi.length`'tir. Sabit bir sayı yazmak, örneğin `i < 3`, bu dizi için doğru sonuç verir; ama dizinin boyutu değişirse kod bozulur. `length` kullanmak koşulu diziye bağlar.

Hatanın hangi turda oluştuğunu görmek için benzer bir programı adım adım izleyelim:

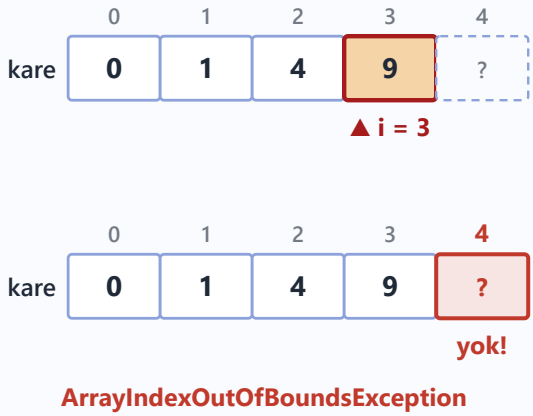
```
int[] kare = new int[4];
for (int i = 0; i <= kare.length; i++) {
    kare[i] = i * i;
}
System.out.println("bitti");
```

`i`	`i <= kare.length`	**Gövde**	`kare`
—	—	başlangıç	[0, 0, 0, 0]
0	0 <= 4 doğru	`kare[0] = 0`	[0, 0, 0, 0]
1	1 <= 4 doğru	`kare[1] = 1`	[0, 1, 0, 0]
2	2 <= 4 doğru	`kare[2] = 4`	[0, 1, 4, 0]
3	3 <= 4 doğru	`kare[3] = 9`	[0, 1, 4, 9]
4	4 <= 4 doğru	`kare[4]` yok: istisna	program durur

Tablo 3.3.7: `<=` ile yazılmış döngünün izlemesi; `kare.length` = 4.

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds
for length 4
        at Kare.main(Kare.java:5)
```

Tablo 3.3.7 hatanın tam yerini gösterir. Son geçerli hücre dördüncü turda dolar. Beşinci turda `4 <= 4` de doğrudur; `<` yazılsaydı döngü burada biterdi. JVM `kare[4]` erişiminde istisna fırlatır ve program bu satırda durur. `"bitti"` hiç yazılmaz, çünkü istisna yakalanmadığı için sonraki satırlara geçilmez.



Şekil 3.3.11: (a) `i` = 3: `kare[3] = 9` ile son geçerli hücre dolar. (b) `i` = 4: `kare[4]` yoktur; JVM `ArrayIndexOutOfBoundsException` fırlatır.

Hatanın kökü sayma alışkanlığıdır. 1'den 4'e kadar sayan biri `<=` yazar; 0'dan başlayıp dört öğe sayan biri ise `< 4` yazmalıdır. İki kalıp birbirine karıştırıldığında ya ilk öğe atlanır ya da son turda dizinin dışına çıkılır. `for (int i = 1; i <= d.length; i++)` ikisini birden yapar: `d[0]` hiç yazılmaz, son turda da istisna oluşur.

3.3.3.4 Oluşturulmamış dizi: `null`

Bir dizi değişkeni tanımlanıp ona dizi bağlanmazsa ya da açıkça `null` atanırsa, değişken hiçbir diziyi göstermez. Böyle bir değişkenle hücreye erişmek `NullPointerException` doğurur:

```
Exception in thread "main" java.lang.NullPointerException: Cannot load from int array
because "<local1>" is null
        at Npe.main(Npe.java:4)
```

Java 15'ten bu yana bu iletiler varsayılan olarak "yardımcı" biçimdedir: neyin `null` olduğunu da söyler. İletideki `"<local1>"`, metodun 1 numaralı yerel değişkenidir, yani `a`'dır. Derleyici varsayılan olarak yerel değişken adlarını `.class` dosyasına yazmaz. Kod `javac -g` ile derlenirse ya da çoğu IDE'de olduğu gibi hata ayıklama bilgisi eklenirse ileti doğrudan `because "a" is null` olur.

Dizi ile `null` arasındaki fark, boş dizi ile yokluk arasındaki farktır. `int[] a = new int[0];` bir dizidir ve `a.length` 0 verir. `int[] a = null;` ise dizi değildir; `a.length` bile `NullPointerException` fırlatır.

3.3.4 İstatistik örnekleri

Bir veri kümesini özetlemenin en bilinen üç ölçüsü ortalama, ortanca ve moddur. Üçü de dizilerle doğal biçimde hesaplanır ve her biri farklı bir teknik gösterir. Ortalama tek bir geçişle toplar, ortanca sıralamaya dayanır, mod sıralı dizide komşuları karşılaştırır. Üç program da verileri kullanıcıdan okuyan aynı bölümle başlar.

3.3.4.1 Kullanıcıdan dizi okuma

```
import java.util.Scanner;
public class OrtalamaHesaplama {
    public static void main(String[] args) {
        // Kullanıcıdan dizi boyutunu al
        Scanner scanner = new Scanner(System.in);
        System.out.print("boyutu girin: ");
        int boyut = scanner.nextInt();
        // Kullanıcıdan elemanları al ve dizi oluştur
        double[] dizi = new double[boyut];
        for (int i = 0; i < boyut; i++) {
            System.out.print((i + 1) + ". elemani girin: ");
            dizi[i] = scanner.nextDouble();
        }
        // ... diziyle hesaplamalar
    }
}
```

Örnek bir çalışmada kullanıcı önce boyutu, ardından dört değeri girer:

```
boyutu girin: 4
1. elemani girin: 7
2. elemani girin: 3
3. elemani girin: 9
4. elemani girin: 5
```

Bu programda dizinin boyutu derleme sırasında bilinmez. `scanner.nextInt()` kullanıcının girdiği sayıyı okur ve `new double[boyut]` diziyi o boyutta oluşturur. Boyut çalışma zamanında belirlenir, ama dizi oluştuktan sonra değişmez. Kullanıcı daha fazla değer girmek isterse yeni bir dizi gerekir.

İstem satırındaki `(i + 1)` küçük ama önemli bir ayrıntıdır. Kullanıcı 1'den sayar ve "1. eleman" görmeyi bekler. Kod ise 0'dan sayar ve değeri `dizi[i]`'ye yazar. Ekranda `i + 1`, kodda `i` kullanılır. Parantezler de gereklidir: `(i + 1) + ". elemani girin: "` önce toplamayı yapar. Parantez olmasaydı `i + 1 + "."` soldan sağa değerlendirilir, yine doğru sonuç verirdi; ama `"" + i + 1` gibi bir yazım `"01"` üretirdi. Parantez niyeti açıkça gösterir.

`Scanner`'ın `nextDouble()` metodu ondalık ayırıcıyı bilgisayarın dil ayarına göre okur. Türkçe ayarlı bir bilgisayarda `7,5` kabul edilir, `7.5` ise `InputMismatchException` doğurur. Tam sayı girişlerinde bu sorun çıkmaz.

3.3.4.2 Ortalama

Ortalama (aritmetik ortalama), değerlerin toplamının değer sayısına bölümüdür. Program okuma bölümünden sonra iki döngü daha çalıştırır: biri toplar, öteki yazdırır.

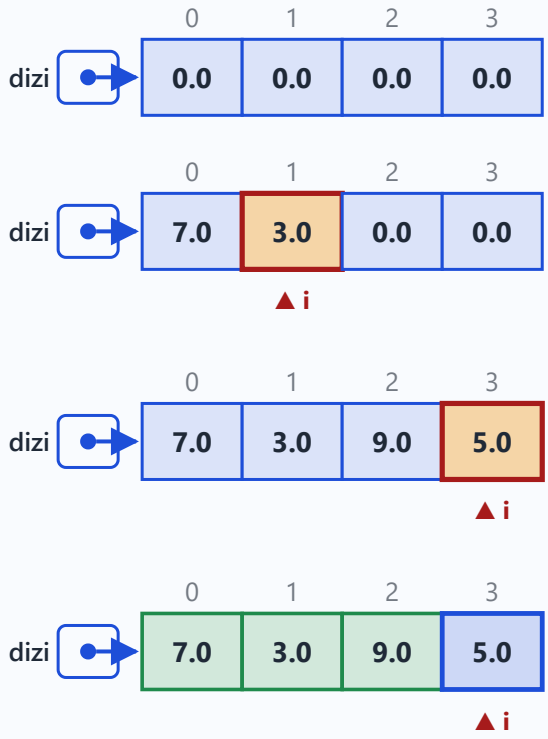
```
Scanner scanner = new Scanner(System.in);
System.out.print("boyutu girin: ");
int boyut = scanner.nextInt();
double[] dizi = new double[boyut];
for (int i = 0; i < boyut; i++) {
    System.out.print((i + 1) + ". elemani girin: ");
    dizi[i] = scanner.nextDouble();
}
double toplam = 0;
for (int i = 0; i < boyut; i++) {
    toplam += dizi[i];
}
double ortalama = toplam / boyut;
System.out.print("Elemanlar: ");
for (int i = 0; i < boyut; i++) {
    System.out.print(dizi[i] + " ");
}
System.out.println();
System.out.println("Ortalama: " + ortalama);
```

```
boyutu girin: 4
1. elemani girin: 7
2. elemani girin: 3
3. elemani girin: 9
4. elemani girin: 5
Elemanlar: 7.0 3.0 9.0 5.0
Ortalama: 6.0
```

Adım	`i`	İşlem	`dizi`	`toplam`
okuma	—	`boyut = 4`, `new double[4]`	[0.0, 0.0, 0.0, 0.0]	—

Adım	`i`	İşlem	`dizi`	`toplam`
okuma	0	`dizi[0] = 7.0`	[7.0, 0.0, 0.0, 0.0]	—
okuma	1	`dizi[1] = 3.0`	[7.0, 3.0, 0.0, 0.0]	—
okuma	2	`dizi[2] = 9.0`	[7.0, 3.0, 9.0, 0.0]	—
okuma	3	`dizi[3] = 5.0`	[7.0, 3.0, 9.0, 5.0]	—
toplama	—	`toplam = 0`	aynı	0.0
toplama	0	`toplam += dizi[0]`	aynı	7.0
toplama	1	`toplam += dizi[1]`	aynı	10.0
toplama	2	`toplam += dizi[2]`	aynı	19.0
toplama	3	`toplam += dizi[3]`	aynı	24.0
bölme	—	`ortalama = 24.0 / 4`	aynı	6.0

Tablo 3.3.8: OrtalamaHesaplama'nın izlemesi; kullanıcı 4, sonra 7, 3, 9, 5 girer.



Şekil 3.3.12: Ortalama izlemesinden dört an. (a) `new double[boyut]` : dört hücre de 0.0. (b) İki değer okundu. (c) Dört değer okundu: 7.0, 3.0, 9.0, 5.0. (d) Toplama döngüsünün son turu: `toplam` = 24.0.

Bölmenin ondalıklı olması bir tasarım kararının sonucudur. `toplam` `double` tanımlandığı için `toplam / boyut` bir `double` ile bir `int`'in bölümüdür. `int` önce `double`'a genişler ve gerçek bölme yapılır: 24.0 / 4 = 6.0. Toplam `int` olsaydı fark görünürdü. 7, 3, 9, 5, 1, 4 kümesi için 29 / 6 tam sayı bölmesiyle 4 verir; doğrusu 4,833…'tür.

Sık yapılan hata

Ortalamayı iki `int` ile hesaplayıp sonucu `double`'a atamak: `double ortalama = toplam / dizi.length;`. Bölme, atamadan *önce* iki `int` arasında yapılır ve kesir zaten atılmıştır; 29 / 6 önce 4 olur, sonra 4.0'a dönüşür. Doğrusu bölmeden önce dönüşümdür: `toplam / (double) dizi.length`.

Programın maliyeti açıktır. Okuma, toplama ve yazdırma döngülerinin her biri n tur yapar; toplam iş $3n$ düzeyindedir ve bu O(n)'dir. Toplam okuma döngüsünde de biriktirilebilirdi; iki döngüyü birleştirmek turu azaltır ama maliyetin büyüme biçimini değiştirmez.

3.3.4.3 Ortanca

Tanım 3.3.6

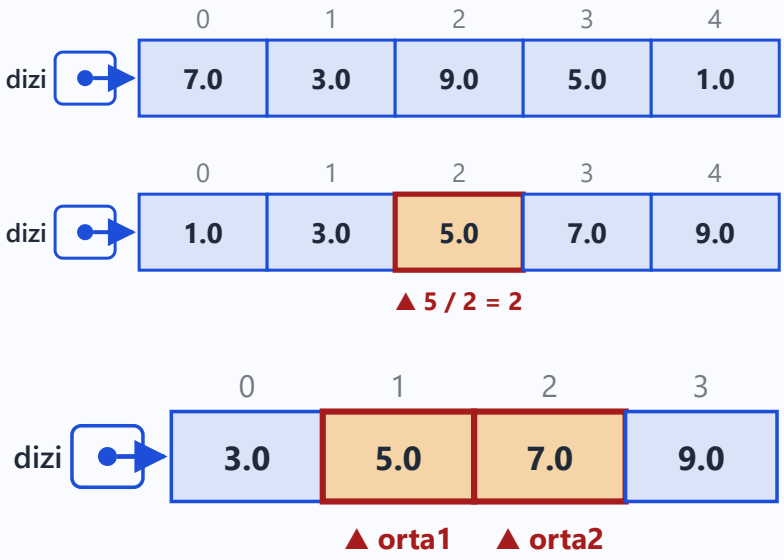
Ortanca (medyan), sıralanmış bir veri kümesinin ortasındaki değerdir. Öğe sayısı tekse tam ortadaki öğe, çiftse ortadaki iki öğenin ortalamasıdır.

Ortanca, uç değerlerden ortalama kadar etkilenmez. 3, 4, 5, 6 ve 100 değerlerinin ortalaması 23,6'dır ve verilerin hiçbirine yakın değildir. Ortancası ise 5'tir. Gelir, ev fiyatı ya da ağ gecikmesi gibi birkaç aşırı değer içeren ölçümlerde ortanca bu yüzden tercih edilir.

```
// boyut ve dizi, OrtalamaHesaplama'daki gibi okunur
Arrays.sort(dizi); // Diziyi sırala
double ortanca;
if (boyut % 2 == 0) { // Çift sayıda eleman
    int orta1 = boyut / 2 - 1;
    int orta2 = boyut / 2;
    ortanca = (dizi[orta1] + dizi[orta2]) / 2;
} else { // Tek sayıda eleman
    ortanca = dizi[boyut / 2];
}
System.out.println("Ortanca Değer: " + ortanca);
```

```
girilenler: 7 3 9 5 1
Elemanlar (sirali): 1.0 3.0 5.0 7.0 9.0
Ortanca Değer: 5.0
```

Hesabın ilk adımı sıralamadır. Ortanca sıralamaya göre tanımlıdır; sırasız bir dizinin ortasındaki öğe rastgele bir değerdir. Şekil 3.3.13 bunu gösterir: girilen sırada ortadaki hücre 9.0'dır, ama ortanca 5.0'tır.



Şekil 3.3.13: Ortanca. (a) Girilen değerler sırasızdır; ortadaki hücre (9.0) ortanca değildir. (b) `Arrays.sort` sonrası, boyut 5 (tek): `dizi[5 / 2]` = `dizi[2]` = 5.0. (c) Boyut 4 (çift; girdiler 7 3 9 5): `orta1` = 1, `orta2` = 2 ve ortanca (5.0 + 7.0) / 2 = 6.0.

Boyut tekse ortadaki indis `boyut / 2`'dir. Boyut 5 için 5 / 2 tam sayı bölmesiyle 2 verir; 2,5 değil. Tam sayı bölmesinin kesir atması burada işe yarar. 0'dan başlayan indislerle 5 öğenin ortası gerçekten indis 2'dedir: solunda iki, sağında iki öğe vardır.

Boyut çiftse tek bir orta öğe yoktur. Boyut 4 için ortadaki iki indis `4 / 2 - 1` = 1 ve `4 / 2` = 2'dir. Girdiler 7, 3, 9, 5 ise sıralı dizi `[3.0, 5.0, 7.0, 9.0]`'dır ve ortanca (5.0 + 7.0) / 2 = 6.0 olur. Dizi `double` olduğu için bölme ondalıklıdır. Aynı kod `int` diziyle yazılsaydı (4 + 5) / 2 gibi bir ifade 4 verirdi; doğrusu 4,5'tir ve `/ 2.0` yazmak gerekir.

Koddaki `double ortanca;` satırı dalların dışındadır. Bu yerleşim zorunludur. Değişken `if` ve `else` dallarının içinde tanımlansaydı her biri yalnız kendi bloğunda yaşardı. Dalların dışındaki `println` onu göremezdi:

```
E_Ortanca.java:10: error: cannot find symbol
        System.out.println("Ortanca Değer: " + ortanca);
                                               ^
  symbol:   variable ortanca
  location: class E_Ortanca
1 error
```

Değişkeni dışarıda tanımlayıp yalnız bir dalda değer vermek de yetmez. `else` dalı olmasaydı derleyici, `if` koşulu yanlışken değişkenin değersiz kalacağını görür ve `variable ortanca might not have been initialized` hatasını verirdi. Koddaki biçimde her iki dal da değer atadığı için sorun yoktur.

`Arrays.sort` bir yan etki taşır: diziyi yerinde değiştirir. Ortanca hesaplandıktan sonra kullanıcının girdiği sıra kaybolmuştur. Sıra daha sonra gerekiyorsa sıralama bir kopya üzerinde yapılır: `double[] sirali = dizi.clone(); Arrays.sort(sirali);`.

Maliyeti sıralama belirler. Ortadaki öğeyi okumak O(1)'dir, ama `Arrays.sort` O(n log n) iş yapar. Ortanca, ortalamadan daha pahalı bir ölçüdür.

3.3.4.4 Mod

Tanım 3.3.7

Mod (tepe değer), bir veri kümesinde en sık tekrar eden değerdir. 4 2 7 2 4 2 9 kümesinin modu 2'dir; 2 üç kez geçer.

Modu bulmanın yalın bir yolu, her değer için kaç kez geçtiğini bütün diziyi tarayarak saymaktır. Bu yol n değerin her biri için n karşılaştırma yapar ve $O(n^2)$ maliyetlidir. Sunumdaki program daha akıllı bir yol izler. Önce diziyi sıralar; böylece eşit değerler yan yana gelir. Sonra tek bir geçişte en uzun "seriyi" bulur.

```
int enCokTekrarEden = dizi[0];
int enCokTekrarSayisi = 1;
int mevcutTekrarEden = dizi[0];
int mevcutTekrarSayisi = 1;
for (int i = 1; i < boyut; i++) {
    if (dizi[i] == dizi[i - 1]) {
        mevcutTekrarSayisi++;
    } else {
        mevcutTekrarSayisi = 1;
        mevcutTekrarEden = dizi[i];
    }
    if (mevcutTekrarSayisi > enCokTekrarSayisi) {
        enCokTekrarSayisi = mevcutTekrarSayisi;
        enCokTekrarEden = mevcutTekrarEden;
    }
}
```

Dört değişken iki seriyi izler. `mevcutTekrarEden` ve `mevcutTekrarSayisi` o anda sayılan serinin değeri ve uzunluğudur. `enCokTekrarEden` ve `enCokTekrarSayisi` şimdiye kadar görülen en uzun serinin değeri ve uzunluğudur. Döngü `i` = 1'den başlar, çünkü her öğe kendisinden önceki öğeyle karşılaştırılır ve ilk öğenin öncesi yoktur. İlk öğe başlangıçta hem mevcut hem en uzun seri sayılır.

Her turda iki karar verilir. Birincisi: öğe bir öncekine eşitse seri uzar; değilse yeni bir seri başlar ve uzunluk 1 olur. İkincisi: mevcut seri en uzun seriden uzunsa en uzun seri güncellenir. Sonuç şöyle yazdırılır:

```
// Sonucu yazdır
System.out.println("Elemanlar sirali: " + Arrays.toString(dizi));
System.out.print("Mod: " + enCokTekrarEden);
System.out.println(" (Tekrar Sayisi: " + enCokTekrarSayisi + ")");
```

```
Elemanlar sirali: [2, 2, 2, 4, 4, 7, 9]
Mod: 2 (Tekrar Sayisi: 3)
```



Şekil 3.3.14: Sıralı dizide mod. (a) Seriler: 2 üç kez, 4 iki kez, 7 ve 9 birer kez. (b) `i` = 3: `dizi[3]` = 4, öncekinden (2) farklıdır ve yeni bir seri başlar.

`i`	Karşılaştırma	Mevcut seri	En uzun seri
—	başlangıç	2 × 1	2 × 1
1	2 == 2	2 × 2	2 × 2 (güncellendi)
2	2 == 2	2 × 3	2 × 3 (güncellendi)
3	4 ≠ 2	4 × 1	2 × 3
4	4 == 4	4 × 2	2 × 3
5	7 ≠ 4	7 × 1	2 × 3
6	9 ≠ 7	9 × 1	2 × 3
7	7 < 7 yanlış: döngü biter. Mod 2, tekrar sayısı 3.		

Tablo 3.3.9: Mod döngüsünün izlemesi; sıralı dizi `[2, 2, 2, 4, 4, 7, 9]`.

Güncelleme koşulundaki `>` işareti eşitlik durumunu belirler. İki seri aynı uzunluktaysa sonradan gelen seri öncekini geçemez; ilk bulunan korunur. Dizi sıralı olduğu için bu, eşit sıklıktaki değerlerden en küçüğünün seçilmesi demektir. `>=` yazılsaydı son bulunan, yani en büyüğü seçilirdi. Birden çok mod olabileceğini bildirmek gerekiyorsa ayrıca saymak gerekir.

Örnek 3.3.7

Sıralı `{2, 2, 6, 6, 6, 9}` dizisinde mod nedir?

İlk seri 2 × 2'dir; en uzun seri 2 × 2 olur. `i` = 2'de 6 başlar (6 × 1). `i` = 3'te seri 6 × 2 olur, ama 2 > 2 yanlıştır ve en uzun seri 2 olarak kalır. `i` = 4'te seri 6 × 3 olur. 3 > 2 doğrudur ve en uzun seri güncellenir. 9 yeni bir seri başlatır.

Program `Mod: 6 (3)` yazar.

Bu yöntemin maliyeti iki parçadır: O(n log n) sıralama ve O(n) tarama. Toplam O(n log n)'dir ve yalın sayma yönteminin $O(n^2)$ maliyetinden çok daha iyidir. Sıralama burada bir hazırlık adımıdır; ardından gelen hesap, eşit değerlerin yan yana durmasına dayanır.

3.3.5 Dizi algoritmaları

Dizi üzerinde en sık yapılan işler aramak, karşılaştırmak, sıralamak ve sırayı değiştirmektir. Bu kesim bu işlerin temel algoritmalarını verir ve her birinin ne kadar iş yaptığını sayar. Sayma işi, algoritmaları karşılaştırmanın ortak dili olan O gösterimine götürür.

3.3.5.1 Doğrusal arama

Tanım 3.3.8

Doğrusal arama (*linear search*), aranan değeri dizinin başından sonuna kadar her öğeyle sırayla karşılaştıran arama yöntemidir. Değer bulununca arama biter; dizinin sonuna gelinirse değer yoktur.

```
static boolean lineerArama(int[] dizi, int hedef) {
    for (int eleman : dizi) {
        if (eleman == hedef) {
            return true;
        }
    }
    return false;
}

// main içinde:
int[] dizi = {10, 20, 30, 40, 50};
boolean sonuc = lineerArama(dizi, 30);
```

`lineerArama` metodu iki parametre alır: aranacak dizi ve hedef değer. `for-each` döngüsü öğeleri sırayla `eleman`'a koyar. Bir öğe hedefe eşitse `return true` hem döngüyü hem metodu hemen bitirir. Döngü hiçbir eşitlik bulmadan biterse son satıra gelinir ve `false` döner. Arama indis gerektirmediği için `for-each` burada uygundur.

Tur	`eleman`	`eleman == hedef`	Sonuç
1	10	10 == 30 yanlış	sıradaki öğe
2	20	20 == 30 yanlış	sıradaki öğe
3	30	30 == 30 doğru	`return true`

Tablo 3.3.10: `lineerArama(dizi, 30)` çağrısının izlemesi.



Şekil 3.3.15: Doğrusal arama. (a) İlk tur: `eleman` = 10, hedef değil. (b) Üçüncü tur: 30 bulunur ve metot `true` döndürür. 40 ve 50'ye hiç bakılmaz.

`main`, `lineerArama(dizi, 30)`'u çağırdığında metoda dizinin referansı ve 30 değeri gider. Dizi kopyalanmaz. Üç karşılaştırma sonucu bulmaya yeter; metot çerçevesi yığından kalkar ve `sonuc` `true` olur. Hedef 35 olsaydı beş karşılaştırmanın hepsi yanlış çıkar ve metot `false` döndürürdü.

Aynı arama metot yazmadan da yapılabilir. Bu kez döngüden çıkmak için `break` kullanılır:

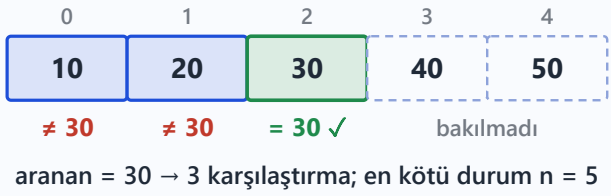
```
int[] dizi = {10, 20, 30, 40, 50};
// Bir dizide belirli bir elemanı arama
int aranan = 30;
boolean bulundu = false;
for (int eleman : dizi) {
    if (eleman == aranan) {
        bulundu = true;
        break;
    }
} // Bu işlem O(n) karmaşıklığına sahiptir.
```

`break`, hedef bulunduğu anda döngüden çıkar. Bu örnekte yalnız üç öğeye bakılır ve `bulundu` `true` olur. `break` olmasaydı sonuç yine doğru olurdu, ama döngü 40 ve 50'ye de gereksiz yere bakardı. `bulundu` değişkeninin `false` ile başlaması önemlidir: hiçbir eşitlik bulunmazsa doğru cevap "yok"tur.

Arama sonucunda yalnız "var mı?" sorusunun cevabı yetmeyebilir. Öğenin *nerede* olduğu gerekiyorsa indis döndürülür ve bu kez klasik `for` kullanılır: `for (int i = 0; i < d.length; i++) if (d[i] == hedef) return i; return -1;` . −1 geçerli bir indis olamayacağı için "yok" anlamında kullanılır. `{12, 7, 30, 5, 18}` dizisinde 5 aranırsa sonuç 3, 99 aranırsa −1'dir.

3.3.5.2 Doğrusal aramanın maliyeti

Bir algoritmanın hızını saniyeyle ölçmek yanıltıcıdır; aynı kod hızlı bir bilgisayarda daha kısa sürer. Bunun yerine algoritmanın yaptığı temel işlem sayılır. Doğrusal aramada temel işlem bir karşılaştırmadır.



Şekil 3.3.16: 30 için doğrusal arama üç karşılaştırma yapar; 40 ve 50'ye bakılmaz. En kötü durumda, hedef sondaysa ya da hiç yoksa, $n = 5$ karşılaştırma gerekir.

Karşılaştırma sayısı hedefin nerede olduğuna bağlıdır. **İyi durumda** hedef ilk öğedir ve tek karşılaştırma yeter. **Kötü durumda** hedef dizinin sonundadır ya da hiç yoktur; n öğeli dizide n karşılaştırma yapılır. Hedef dizide rastgele bir yerdeyse ortalama karşılaştırma sayısı yaklaşık $n/2$'dir.

Tanım 3.3.9

Zaman karmaşıklığı (*time complexity*), bir algoritmanın yaptığı iş miktarının girdi boyutu n ile nasıl büyüdüğünü anlatır. **O gösterimi** bu büyümenin biçimini verir; sabit çarpanları ve küçük terimleri atar. İş miktarı n ile doğru orantılı büyüyorsa algoritma O(n)'dir.

Doğrusal aramanın zaman karmaşıklığı O(n)'dir. $n/2$ ile n arasındaki fark O gösteriminde önemsizdir: ikisi de dizi iki katına çıkınca iki katına çıkar. Sırasız bir dizide daha iyisi yapılamaz. Her öğeye bakmadan "yok" denemez, çünkü bakılmayan öğe hedef olabilir.

3.3.5.3 İkili arama

Dizi sıralıysa durum değişir. Telefon rehberinde bir ad ararken kimse ilk sayfadan başlamaz. Rehber ortadan açılır; aranan ad o sayfadakinden önce geliyorsa sol yarıya, sonra geliyorsa sağ yarıya geçilir. Her bakışta aranacak alan yarıya iner.

Tanım 3.3.10

İkili arama (*binary search*), sıralı bir dizide ortadaki öğeye bakıp hedef küçükse sol yarıda, büyükse sağ yarıda aramayı süren yöntemdir. Arama alanı her adımda yarıya iner. Yalnız sıralı dizide çalışır.

```
static int ikiliAra(int[] dizi, int hedef) {
    int alt = 0, ust = dizi.length - 1;
    while (alt <= ust) {
        int orta = (alt + ust) / 2;
        if (dizi[orta] == hedef) {
            return orta;          // bulundu
        } else if (hedef < dizi[orta]) {
            ust = orta - 1;       // sol yarı
        } else {
            alt = orta + 1;       // sağ yarı
        }
    }
    return -1;                    // alan tükendi
}
// main içinde:
int[] dizi = {3, 9, 14, 27, 42, 48, 55, 61, 70};
System.out.println(ikiliAra(dizi, 48));
System.out.println(ikiliAra(dizi, 50));
```

```
5
-1
```

Metot arama alanını iki değişkenle tutar: `alt` alanın ilk, `ust` son indisidir. Başlangıçta alan bütün dizidir. Her turda ortadaki indis `(alt + ust) / 2` ile bulunur; tam sayı bölmesi kesri atar. Orta öğe hedefse indisi döner. Hedef daha küçükse sağ yarı elenir ve `ust = orta - 1` olur. Hedef daha büyükse sol yarı elenir ve `alt = orta + 1` olur. `alt`, `ust`'ü geçince alan boşalmıştır ve −1 döner.

Hedef	Tur	`alt`	`ust`	`orta`	`dizi[orta]`	Karar
48	1	0	8	4	42	48 > 42: `alt = 5`
48	2	5	8	6	55	48 < 55: `ust = 5`
48	3	5	5	5	48	bulundu: 5 döner
50	1	0	8	4	42	50 > 42: `alt = 5`
50	2	5	8	6	55	50 < 55: `ust = 5`
50	3	5	5	5	48	50 > 48: `alt = 6`
50	—	6	5	—	—	6 <= 5 yanlış: −1 döner

Tablo 3.3.11: `{3, 9, 14, 27, 42, 48, 55, 61, 70}` dizisinde ikili arama.

Tablo 3.3.11 iki aramayı yan yana verir. 48 için orta indisler 4 (42), 6 (55) ve 5 (48)'tir. Üç karşılaştırma yeter. 50 için aynı üç adım atılır, sonra `alt` 6, `ust` 5 olur. Alan boşalmıştır ve 50'nin dizide olmadığı kesinleşir. Doğrusal arama aynı sonuca ancak dokuz karşılaştırmayla ulaşırdı.

Kazancın büyüklüğü dizi büyüdükçe görünür. n öğeli alan her adımda yarıya indiği için en çok yaklaşık $\log_2 n$ adımda tek öğeye iner. İkili aramanın zaman karmaşıklığı bu yüzden O(log n)'dir. n = 1000 için en çok 10 karşılaştırma gerekir. n = 1.000.000 için bu sayı en çok 20'dir. Doğrusal arama aynı diziler için 1000 ve 1.000.000 karşılaştırma yapabilir.

Bunun bir bedeli vardır: dizi sıralı olmalıdır. Sıralama O(n log n) iş ister ve tek bir arama için yapılmaya değmez. Aynı dizide çok sayıda arama yapılacaksa bir kez sıralayıp her aramayı O(log n)'de yapmak kazançlıdır. `Arrays.binarySearch` aynı yöntemi kullanır; sıralanmamış dizide sonucu anlamsızdır.

Sık yapılan hata

Döngü koşulunu `alt < ust` yazmak. Bu durumda alanda tek öğe kaldığında (`alt == ust`) döngü o öğeye bakmadan biter. 48 aramasının üçüncü turunda tam bu durum oluşur. Metot, 48 dizide olduğu hâlde −1 döndürür. Doğru koşul `alt <= ust` 'tür. Aynı biçimde `ust = orta` yazmak da yanlıştır: alan küçülmeyebilir ve döngü sonsuza kadar döner.

3.3.5.4 Eşleştirme

İki dizinin ortak öğelerini bulmak için her öğe öteki dizinin her öğesiyle karşılaştırılır. Bu iş iç içe iki döngüyle yapılır:

```
static void eslestir(int[] dizi1, int[] dizi2) {
    for (int eleman1 : dizi1) {
        for (int eleman2 : dizi2) {
            if (eleman1 == eleman2) {
                System.out.println("Eslesen Eleman: " + eleman1);
            }
        }
    }
}
```

`eslestir(new int[]{3, 7, 1}, new int[]{5, 1, 3, 8})` çağrısı `Eslesen Eleman: 3` ve `Eslesen Eleman: 1` yazar. Çağrıda diziler değişkene atanmadan, `new int[]{…}` uzun biçimiyle doğrudan oluşturulmuştur. Kısa biçim `{3, 7, 1}` burada kullanılamaz.



Şekil 3.3.17: Eşleştirmenin karşılaştırma tablosu. Her satır dış döngünün bir öğesi, her sütun iç döngünün bir öğesidir. 12 hücrenin her biri bir karşılaştırmadır. İki hücrede eşitlik vardır.

Dış döngü `eleman1` = 3 iken iç döngü `dizi2` 'nin dört öğesinin hepsiyle karşılaştırma yapar. Eşleşme bulunduğunda yazdırılır, ama iç döngü durmaz; kalan öğelere de bakar. Aynı iş `eleman1` = 7 ve 1 için tekrarlanır. Şekil 3.3.17 bütün karşılaştırmaları bir tablo olarak gösterir. Toplam 3 × 4 = 12 karşılaştırma yapılır.

Diziler n ve m öğeliyse $n \cdot m$ karşılaştırma gerekir. İki dizi de n öğeliyse bu n^2 olur ve eşleştirme $O(n^2)$ maliyetlidir. İç içe döngüler genellikle bu karesel büyümenin işaretidir. Daha hızlı bir yol da vardır: ikinci dizi önce sıralanır ve birinci dizinin her öğesi ikili aramayla aranır. Bu yolun maliyeti O(n log n)'dir.

3.3.5.5 Karmaşıklık: O(1), O(n), O(n²)

Algoritmaların maliyetini karşılaştırmadan önce en temel işlemin, indisle erişimin maliyeti anlaşılmalıdır. `dizi[2]` ile `dizi[1000000]` aynı sürede okunur. Bunun nedeni dizinin bellekteki yerleşimidir.



Şekil 3.3.18: Dizi öğelerinin bellek adresleri. Dizi 1000 adresinden başlar ve her `int` 4 bayt tutar. `dizi[2]` 'nin adresi 1000 + 2 × 4 = 1008 olarak tek bir hesapla bulunur.

Hücreler ardışık ve eşit boyutlu olduğu için `i`. öğenin adresi bir çarpma ve bir toplamayla hesaplanır:

> adres(`dizi[i]`) = başlangıç + `i` × öğe boyutu

`int` için öğe boyutu 4 bayttır. Başlangıç adresi 1000 ise `dizi[2]` 1008'de, `dizi[1000000]` 4.001.000'de durur. İki hesap da aynı sürer; işlemcinin önceki öğelere bakması gerekmez. Erişimin maliyeti dizinin boyutundan bağımsızdır ve bu O(1), yani **sabit zaman** diye yazılır. İndisin 0'dan başlamasının bir nedeni de bu formüldür: indis 1'den başlasaydı her erişimde bir çıkarma daha gerekirdi.

İşlem	Karmaşıklık	$n = 1000$ için
Erişim `dizi[i]`	O(1), sabit	1
İkili arama (sıralı dizi)	O(log n), logaritmik	≤ 10
Doğrusal arama	O(n), doğrusal	≤ 1000
Tersine çevirme	O(n), doğrusal	500 takas
Eşleştirme (iç içe)	$O(n^2)$, karesel	1.000.000
Seçimli sıralama	$O(n^2)$, karesel	≈ 500.000 (tam olarak 499.500)

Tablo 3.3.12: Dizi işlemlerinin karmaşıklığı ve $n = 1000$ için yaklaşık işlem sayısı.

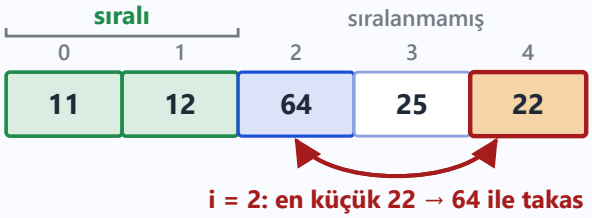
Tablo 3.3.12 farkların büyüklüğünü gösterir. $O(n^2)$ algoritmalar büyük veri kümelerinde verimsizdir. n 10 kat büyürse O(n) bir algoritmanın süresi 10 kat, $O(n^2)$ bir algoritmanınki 100 kat büyür. Bir saniyede 1000 öğeyi sıralayan karesel bir algoritma, 100.000 öğe için yaklaşık üç saat sürer. O(1) erişim ise girdi boyutundan hiç etkilenmez.

3.3.5.6 Seçimli sıralama

Tanım 3.3.11

Seçimli sıralama (*selection sort*), her turda dizinin sıralanmamış kısmındaki en küçük öğeyi bulup bu kısmın başına taşıyan sıralama yöntemidir. Her turdan sonra sıralı kısım bir öğe büyür.

```
static void secimliSirala(int[] dizi) {
    for (int i = 0; i < dizi.length - 1; i++) {
        int minIndeks = i;
        for (int j = i + 1; j < dizi.length; j++) {
            if (dizi[j] < dizi[minIndeks]) {
                minIndeks = j;
            }
        }
        int gecici = dizi[minIndeks];
        dizi[minIndeks] = dizi[i];
        dizi[i] = gecici;
    }
}
```



Şekil 3.3.19: Seçimli sıralamanın bir turu. İlk iki hücre sıralıdır. `i` = 2 turunda sıralanmamış kısmın en küçüğü 22 bulunur ve kısmın başındaki 64 ile yer değiştirir.

Dış döngü `i`'yi 0'dan `length - 2`'ye kadar sayar. `i`, sıralanmamış kısmın başıdır. Her turda `minIndeks` önce `i` olur; yani kısmın ilk öğesi en küçük kabul edilir. İç döngü `j` ile kısmın geri kalanını tarar ve daha küçük bir öğe gördükçe `minIndeks`'i günceller. İç döngü bitince en küçük öğe `dizi[minIndeks]`'tedir ve `dizi[i]` ile yer değiştirir.

Takas üç atamayla, geçici bir değişken yardımıyla yapılır. `dizi[i] = dizi[minIndeks];` `dizi[minIndeks] = dizi[i];` yazılsaydı ilk atama `dizi[i]`'nin eski değerini silerdi ve iki hücre de aynı değeri taşırdı. `gecici` eski değeri saklar. Dış döngünün `length - 1` tur yapması yeterlidir: n − 1 öğe yerine oturunca son öğe kendiliğinden doğru yerdedir.

Sıralama dizinin kendisi üzerinde yapılır ve metot `void`'dir. Metoda dizinin referansı geldiği için takaslar çağıranın dizisini değiştirir. Bu nokta, takas işini ayrı bir metoda taşımak isteyenler için bir tuzak hazırlar.

Sık yapılan hata

Takası `static void yerDegistir(int x, int y) { int g = x; x = y; y = g; }` biçiminde yazıp `yerDegistir(dizi[i], dizi[minIndeks])` çağırmak. Metoda öğelerin yalnız kopyaları gider; metot kopyaları değiştirir ve dizi aynı kalır. Çalışan sürüm dizinin kendisini ve iki indisi alır: `yerDegistir(int[] d, int i, int j)`. Bu metot `d[i]` ile `d[j]`'yi değiştirir ve değişiklik çağıranın dizisinde görünür.

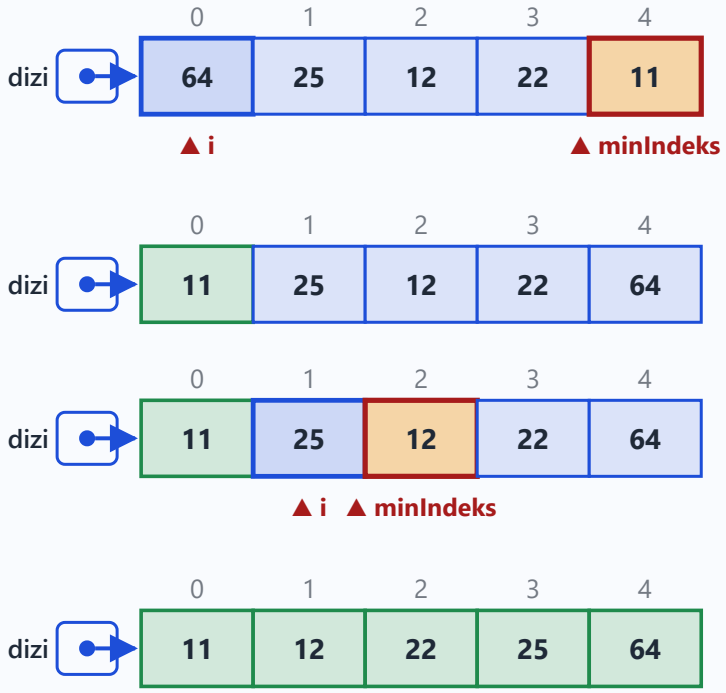
Sunumdaki örnekte `{64, 25, 12, 22, 11}` dizisi sıralanır. `secimliSirala`'ya her turun sonunda `Arrays.toString(dizi)` yazdıran bir satır eklenirse turlar görünür:

```
int[] dizi = {64, 25, 12, 22, 11};
secimliSirala(dizi);
// secimliSirala her turun sonunda yazdırır:
// System.out.println("i=" + i + ": "
//         + Arrays.toString(dizi));
```

Tur `i`	Taranan `j`	En küçük (indis)	Takas	Tur sonunda dizi
başlangıç	—	—	—	[64, 25, 12, 22, 11]

Tur `i`	Taranan `j`	En küçük (indis)	Takas	Tur sonunda dizi
0	1…4	11 (4)	`dizi[0]` ↔ `dizi[4]`	[**11**, 25, 12, 22, 64]
1	2…4	12 (2)	`dizi[1]` ↔ `dizi[2]`	[**11, 12**, 25, 22, 64]
2	3…4	22 (3)	`dizi[2]` ↔ `dizi[3]`	[**11, 12, 22**, 25, 64]
3	4…4	25 (3)	kendisiyle	[**11, 12, 22, 25**, 64]

Tablo 3.3.13: `{64, 25, 12, 22, 11}` dizisinin seçimli sıralamayla sıralanması.



Şekil 3.3.20: Seçimli sıralama izlemesi. (a) `i` = 0: iç döngü en küçük değeri arar ve 11'i (indis 4) bulur. (b) Takas sonrası ilk hücre kesin yerindedir. (c) `i` = 1: en küçük 12 (indis 2). (d) Dört tur sonunda dizi sıralıdır.

Son turda en küçük öğe zaten yerindedir ve kendisiyle takas edilir; dizi değişmez. Kod bu durumu ayrıca denetlemez. Bir `if (minIndeks != i)` koşulu gereksiz takası önleyebilir, ama algoritmanın maliyetini değiştirmez.

Karşılaştırma sayısı girdinin sırasından bağımsızdır. İlk turda iç döngü $n - 1$, ikincide $n - 2$ öğeye bakar; son turda tek öğeye bakar. Toplam:

> $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1) / 2$

5 öğeli dizi için 4 + 3 + 2 + 1 = 10 karşılaştırma yapılır. n = 1000 için 499.500'dür. Dizi baştan sıralı olsa bile iç döngü her turda sonuna kadar döner; seçimli sıralamanın her durumda $O(n^2)$ olmasının nedeni budur. Takas sayısı ise en çok n − 1'dir. Takasın pahalı olduğu durumlarda bu bir üstünlüktür.

3.3.5.7 Kabarcık sıralaması

Tanım 3.3.12

Kabarcık sıralaması (*bubble sort*), dizinin komşu iki öğesini karşılaştırıp ters sıradaysa yer değiştiren sıralama yöntemidir. Her turda kalan kısmın en büyük öğesi sona doğru "kabarcık gibi" yükselir.

```
static void kabarcikSirala(int[] dizi) {
    int n = dizi.length;
    for (int tur = 0; tur < n - 1; tur++) {
        boolean degisti = false;
        for (int j = 0; j < n - 1 - tur; j++) {
            if (dizi[j] > dizi[j + 1]) {  // ters sıra
                int gecici = dizi[j];
                dizi[j] = dizi[j + 1];
                dizi[j + 1] = gecici;
                degisti = true;
            }
        }
        System.out.println(Arrays.toString(dizi));
        if (!degisti) break;  // takas yok: sıralı
    }
}
```

`{64, 25, 12, 22, 11, 36, 48}` dizisi için her turun sonunda dizi şöyle olur:

```
[25, 12, 22, 11, 36, 48, 64]
[12, 22, 11, 25, 36, 48, 64]
[12, 11, 22, 25, 36, 48, 64]
[11, 12, 22, 25, 36, 48, 64]
[11, 12, 22, 25, 36, 48, 64]
```

İç döngü `j`'yi 0'dan başlatır ve `dizi[j]` ile `dizi[j + 1]`'i karşılaştırır. Soldaki büyükse takas yapılır. İlk tur bittiğinde en büyük öğe, 64, en sağdadır; çıktının ilk satırı bunu gösterir. İkinci turda kalan altı öğenin en büyüğü, 48, sağdan ikinci hücreye yerleşir; ama 36 ile 48 zaten yerinde olduğu için bu tur onları değiştirmez.

Her turdan sonra sondaki bir öğe kesin yerine oturur. Bu yüzden iç döngünün sınırı `n - 1 - tur`'dur: yerine oturmuş öğelerle yeniden karşılaştırma yapılmaz. `j + 1`'in de dizinin içinde kalması gerekir. Sınır `n - tur` yazılsaydı ilk turda `j` = 6 olur ve `dizi[7]`'ye erişilmeye çalışılırdı.

`degisti` bayrağı algoritmanın akıllı yanıdır. Bir turda hiç takas yapılmadıysa bütün komşu çiftler doğru sıradadır, yani dizi sıralıdır. Örnekte dördüncü turdan sonra dizi sıralanmıştır. Beşinci tur hiçbir takas yapmaz ve `break` ile döngüden çıkılır; altıncı tura gerek kalmaz. Çıktıdaki son iki satırın aynı olmasının nedeni budur.

Tur	Karşılaştırma	Yerine oturan	Tur sonunda dizi
0	6	64	[25, 12, 22, 11, 36, 48, 64]
1	5	48	[12, 22, 11, 25, 36, 48, 64]
2	4	36	[12, 11, 22, 25, 36, 48, 64]
3	3	25	[11, 12, 22, 25, 36, 48, 64]
4	2	takas yok: `break`	[11, 12, 22, 25, 36, 48, 64]

Tablo 3.3.14: Kabarcık sıralamasının tur özeti; $n = 7$.

Bu çalışmada toplam 6 + 5 + 4 + 3 + 2 = 20 karşılaştırma ve 11 takas yapılır. Dizi baştan sıralı olsaydı ilk turda hiç takas olmaz ve döngü tek turda biterdi: $n - 1$ karşılaştırma, yani O(n). En kötü durumda, dizi ters sıralıyken, bütün turlar çalışır ve karşılaştırma sayısı yine $n(n - 1)/2$ olur; $O(n^2)$. Seçimli sıralamadan farkı, takas sayısının da $O(n^2)$ olabilmesidir. Ters sıralı dizide her karşılaştırma bir takasla sonuçlanır.

Örnek 3.3.8

`{2, 1, 3, 4, 5}` dizisi kabarcık sıralamasıyla kaç turda ve kaç karşılaştırmayla sıralanır?

İlk turda 2 ile 1 takas edilir, sonraki üç karşılaştırmada takas olmaz: 4 karşılaştırma, dizi `[1, 2, 3, 4, 5]`. `degisti` doğru olduğu için ikinci tura geçilir. İkinci turda 3 karşılaştırma yapılır, hiç takas olmaz ve `break` çalışır.

Sonuç: 2 tur, 4 + 3 = 7 karşılaştırma. Seçimli sıralama aynı dizi için 10 karşılaştırma yapardı.

3.3.5.8 Diziyi tersine çevirme

Bir diziyi tersine çevirmenin kolay ama savurgan yolu, aynı boyutta ikinci bir dizi açıp öğeleri sondan başa doğru oraya kopyalamaktır. Daha iyi yol diziyi yerinde çevirmektir:

```java
static void tersineCevir(int[] dizi) {
    int baslangic = 0;
    int bitis = dizi.length - 1;
    while (baslangic < bitis) {
        // Başlangıç ve bitiş elemanlarını değiştir
        int gecici = dizi[baslangic];
        dizi[baslangic] = dizi[bitis];
        dizi[bitis] = gecici;
        // Başlangıç indeksini artır, bitiş indeksini azalt
        baslangic++;
        bitis--;
    }
}
```

```
[1, 2, 3, 4, 5]
→ [5, 4, 3, 2, 1]
[1, 2, 3, 4]
→ [4, 3, 2, 1]
```

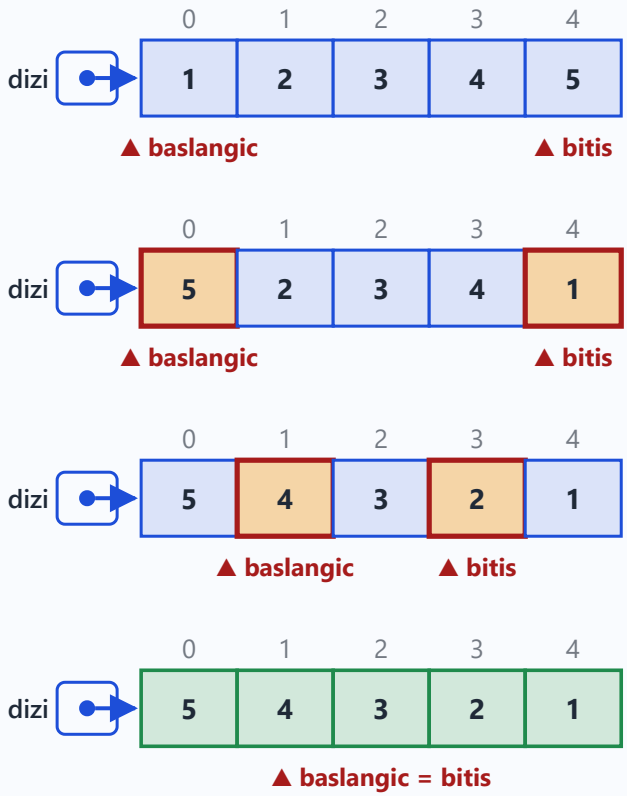


Şekil 3.3.21: İki işaretçi tekniği. `baslangic` soldan, `bitis` sağdan gelir; her turda uçtaki iki öğe takas edilir. Tek uzunlukta dizide ortadaki öğe yerinde kalır.

Bu yöntemin adı **iki işaretçi** (*two pointers*) tekniğidir. `baslangic` 0'dan, `bitis` son indisten başlar. Her turda iki uçtaki öğe takas edilir, ardından işaretçiler birbirine bir adım yaklaşır. `baslangic < bitis` koşulu yanlış olunca, yani işaretçiler buluşunca ya da birbirini geçince, iş biter.

`baslangic`	`bitis`	**Koşul**	`gecici`	**Dizi**
0	4	0 < 4 doğru	1	[5, 2, 3, 4, 1]
1	3	1 < 3 doğru	2	[5, 4, 3, 2, 1]
2	2	2 < 2 yanlış	—	döngü biter

Tablo 3.3.15: `tersineCevir` izlemesi; dizi `[1, 2, 3, 4, 5]`.



Şekil 3.3.22: `tersineCevir` izlemesi. (a) İşaretçiler uçlarda. (b) `dizi[0]` ile `dizi[4]` takas edildi. (c) `dizi[1]` ile `dizi[3]` takas edildi. (d) `baslangic` = `bitis` = 2: döngü biter, ortadaki 3 yerinde kalır.

Beş öğeli dizide iki takas yeter. Dört öğeli dizide de iki takas yapılır: önce 0 ile 3, sonra 1 ile 2; ardından `baslangic` 2, `bitis` 1 olur ve koşul yanlış çıkar. Genel olarak $n/2$ takas (tam sayı bölmesiyle) gerekir. Her takas üç atamadır ve iş miktarı n ile doğru orantılıdır: O(n). Ek dizi gerekmediği için ek bellek O(1)'dir. Böyle çalışan algoritmalara **yerinde** (*in-place*) algoritma denir.

Metot `void` 'dir ve hiçbir şey döndürmez. Yine de çağıranın dizisi değişir, çünkü metoda dizinin referansı gelmiştir. Çıktıdaki `→ [5, 4, 3, 2, 1]` satırı, `main` 'in metot çağrısından sonra kendi dizisini yazdırmasıyla üretilir.

Sık yapılan hata

Tersine çevirme döngüsünü dizinin tamamı üzerinde çalıştırmak: `for (int i = 0; i < d.length; i++)` ile `d[i]` ve `d[d.length - 1 - i]` 'yi takas etmek. İlk yarıda dizi tersine döner, ikinci yarıda aynı çiftler yeniden takas edilir ve dizi eski hâline geri gelir. `{1, 2, 3, 4, 5}` için sonuç yine `[1, 2, 3, 4, 5]` 'tir. Döngü yalnız dizinin yarısında gezinmelidir: `i < d.length / 2` .

3.3.6 Çok boyutlu diziler

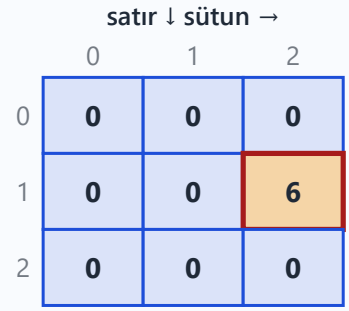
Bazı veriler doğal olarak tablo biçimindedir: bir sınıfın derslere göre notları, bir devrenin admitans matrisi, bir resmin piksel ızgarası. Bu verileri tek boyutlu bir dizide tutmak mümkündür, ama satır ve sütun bilgisini indis hesabıyla elle yönetmek gerekir. Çok boyutlu diziler bu işi dile bırakır. Java iki ve daha fazla boyutlu dizileri destekler.

3.3.6.1 İki boyutlu diziler

Tanım 3.3.13

İki boyutlu dizi (*two-dimensional array*), öğeleri satır ve sütunlara yerleşen, tablo benzeri bir dizidir. Bir öğeye `matris[satir][sutun]` ile erişilir: ilk indis satırı, ikinci indis sütunu verir.

```
int[][] matris = new int[3][3]; // 3x3 boyutunda bir matris
matris[1][2] = 6;  // 2. satır, 3. sütun
```



Şekil 3.3.23: 3×3 matris. Bütün hücreler 0 ile başlar; `matris[1][2] = 6` ikinci satırın üçüncü sütununa yazar.

`new int[3][3]` 3 satır ve 3 sütunluk bir matris oluşturur. Tek boyutlu dizilerde olduğu gibi bütün hücreler varsayılan değerle, burada 0 ile başlar. `matris[1][2] = 6` satırı ikinci satırın üçüncü sütununa yazar. İndisler yine 0'dan başladığı için “2. satır” indis 1'e, “3. sütun” indis 2'ye karşılık gelir.

Satır ve sütunun sırası bir uzlaşmadır, ama tutarlı olmalıdır. Java'da ilk indis dış diziyi, ikinci indis iç diziyi seçer. Matrislerde bu, ilk indisin satır olması demektir; matematikteki a_{ij} yazımıyla da örtüşür. Resimlerde aynı nedenle `[y][x]` sırası kullanılır: önce satır (yükseklik), sonra sütun (genişlik).

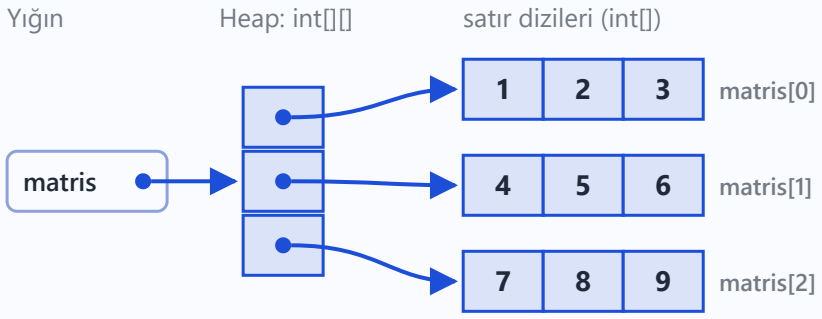
3.3.6.2 Diziler dizisi

İki boyutlu dizilere de ilklendirici listesiyle başlangıç değeri verilebilir. Bu kez her satır kendi süslü parantezi içinde yazılır:

```
int[][] matris = { {1, 2, 3}, {4, 5, 6}, {7, 8, 9} };
System.out.println(matris.length + " "
        + matris[0].length + " " + matris[1][2]);
```

```
3 3 6
```

Çıktıyı anlamak için Java'nın iki boyutlu diziyi nasıl kurduğunu bilmek gerekir. Java'da gerçek bir "tablo" yoktur. 2B dizi aslında **dizilerin dizisidir**. `matris` üç öğeli bir dizidir ve her öğesi bir `int[]`'e, yani bir satıra referanstır.



Şekil 3.3.24: `matris` bir satır dizileri dizisidir. Dış dizinin üç hücresi, heap'teki üç ayrı satır dizisini gösterir. `matris[1]` ikinci satırın kendisidir.

Şekil 3.3.24 bu yapıyı gösterir. `matris.length` dış dizinin uzunluğudur, yani satır sayısıdır: 3. `matris[0]` ilk satırın kendisidir ve bir `int[]`'tir; `matris[0].length` ilk satırın sütun sayısıdır: 3. `matris[1][2]` önce ikinci satırı seçer, sonra o satırın üçüncü öğesini okur: 6.

Bu yapının pratik sonuçları vardır. `matris[1]` tek başına anlamlı bir ifadedir ve bir satırı tek boyutlu dizi olarak verir. `Arrays.toString(matris[1])` `[4, 5, 6]` yazar. Bir satır, tek boyutlu dizi bekleyen bir metoda doğrudan geçirilebilir. Buna karşılık `Arrays.toString(matris)` satırların içeriğini değil, kimliklerini yazar: `[[I@…, [I@…, [I@…]`. İç içe dizileri yazdırmak için `Arrays.deepToString` kullanılır.

Örnek 3.3.9

`int[][] m = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};` matrisinin izi, yani köşegen öğelerinin toplamı bulunsun.

Köşegende satır ve sütun indisi eşittir: `m[0][0]`, `m[1][1]`, `m[2][2]`. Bu yüzden tek bir döngü yeter: `for (int i = 0; i < m.length; i++) iz += m[i][i];`.

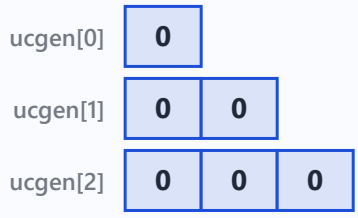
Turlar `iz` 'i 1, 6 ve 15 yapar. Matrisin izi 1 + 5 + 9 = 15'tir. İz yalnız kare matris için tanımlıdır; kare olmayan bir matriste `m[i][i]` bir satırın dışına çıkabilir.

3.3.6.3 Düzensiz diziler ve Pascal üçgeni

Satırlar ayrı diziler olduğu için uzunluklarının eşit olması gerekmez. Satır uzunlukları farklı olan çok boyutlu diziye **düzensiz dizi** (*ragged array*) denir.

```
int[][] ucgen = new int[3][];
for (int i = 0; i < ucgen.length; i++) {
    ucgen[i] = new int[i + 1];
}
System.out.println(ucgen[0].length + " " + ucgen[1].length
        + " " + ucgen[2].length);
int[][] tablo = {{1, 2}, {3, 4, 5}};
for (int[] satir : tablo) {
    for (int x : satir) {
        System.out.print(x + " ");
    }
    System.out.println();
}
```

```
1 2 3
1 2
3 4 5
```



Şekil 3.3.25: Düzensiz dizi: `ucgen` 'in satır uzunlukları 1, 2 ve 3'tür.

`new int[3][]` yalnız ilk boyutu verir. Üç satır referansı oluşturulur, ama satırların kendisi oluşturulmaz; üç referans da `null` 'dır. Satırlar döngüde tek tek ayrılır: `ucgen[i] = new int[i + 1]`. Sonuçta satır uzunlukları 1, 2 ve 3 olur. Satır ayrılmadan ona erişilirse `NullPointerException` oluşur. Yalnız ikinci boyut verilemez: `new int[][3]` derlenmez.

Programın ikinci kısmı ilklendirici listesiyle düzensiz bir dizi kurar: `tablo` 'nun ilk satırı iki, ikinci satırı üç öğelidir. `for-each` ile gezinirken dış döngünün öğesi bir `int[]` satırdır; iç döngü o satırın öğeleri üzerinde gezinir. Düzensiz dizilerde iç döngünün sınırı her satır için ayrı olmalıdır. Klasik `for`

ile yazılırken bu yüzden `matris[i].length` kullanılır. Sabit bir sütun sayısı yazmak kısa satırlarda sınır dışı hatasına yol açar.

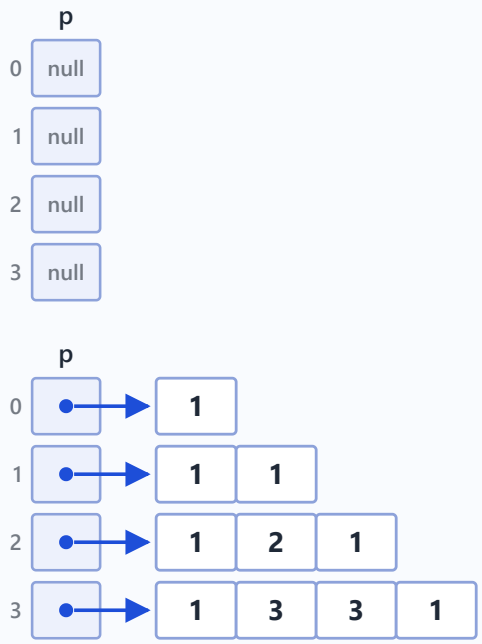
Düzensiz dizinin güzel bir uygulaması Pascal üçgenidir. Üçgenin her satırı bir öncekinden bir öğe uzundur. Satırın iki ucu 1'dir; iç öğeler, üstteki satırda hemen solda ve hemen üstte duran iki öğenin toplamıdır.

```
int[][] p = new int[4][];
for (int i = 0; i < p.length; i++) {
    p[i] = new int[i + 1];
    p[i][0] = 1;
    p[i][i] = 1;
    for (int j = 1; j < i; j++) {
        p[i][j] = p[i - 1][j - 1] + p[i - 1][j];
    }
}
System.out.println(Arrays.toString(p[3]));
```

Program `[1, 3, 3, 1]` yazar. Bu satır, $(a + b)^3$ açılımının katsayılarıdır.

`i`	Satır oluşturma	Uçlar	İç hücreler (`j` = 1 … `i` − 1)	`p[i]`
0	`new int[1]`	`p[0][0] = 1` (iki uç aynı hücre)	yok	[1]
1	`new int[2]`	`p[1][0] = p[1][1] = 1`	yok (`j < 1` hemen yanlış)	[1, 1]
2	`new int[3]`	1, 1	`p[2][1]` = 1 + 1 = 2	[1, 2, 1]
3	`new int[4]`	1, 1	`p[3][1]` = 1 + 2 = 3, `p[3][2]` = 2 + 1 = 3	[1, 3, 3, 1]

Tablo 3.3.16: Pascal üçgeninin satır satır kurulması.



Şekil 3.3.26: Pascal üçgeni. (a) `new int[4][]` yalnız dört satır referansı ayırır; hepsi `null` 'dır. (b) Döngü bittiğinde satır uzunlukları 1, 2, 3 ve 4'tür.

Kodun incelikli yeri iç döngünün sınırlarıdır. `j` 1'den başlar ve `i` 'den küçük kaldıkça sürer; böylece uç hücrelere dokunulmaz. `i` = 0 ve `i` = 1 satırlarında iç hücre yoktur ve iç döngü hiç dönmez. `p[i - 1][j]` ifadesi de güvenlidir: `j` en çok `i - 1` olduğu için bir önceki satırın son indisini geçmez. Satır `i` 'nin `i + 1` öğesi vardır; toplam öğe sayısı 1 + 2 + … + n = $n(n + 1)/2$'dir. Kare bir matris yerine düzensiz dizi kullanmak, aşağı yukarı yarı yarıya bellek kazandırır.

3.3.6.4 Matrisi doldurma ve yazdırma

İki boyutlu bir dizide gezinmenin genel yolu iç içe iki döngüdür. Dış döngü satırlar, iç döngü bir satırın sütunları üzerinde gezinir.

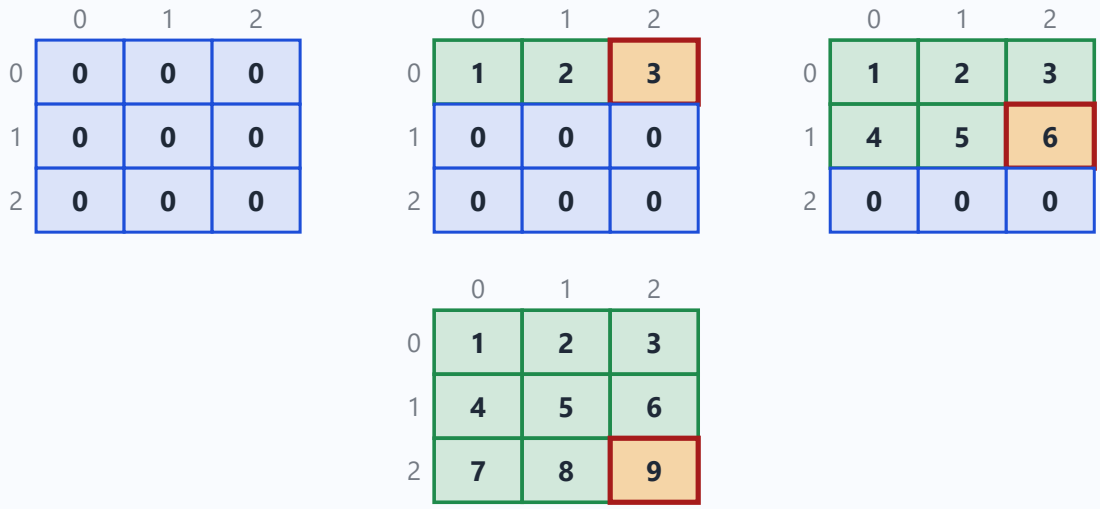
```java
public class CokBoyutluDiziOrnegi {
    public static void main(String[] args) {
        // 3x3 boyutunda iki boyutlu bir dizi (matris)
        int[][] matris = new int[3][3];
        // Matrisin elemanlarına ilk değerlerini atama
        for (int satir = 0; satir < 3; satir++) {
            for (int sutun = 0; sutun < 3; sutun++) {
                matris[satir][sutun] = satir * 3 + sutun + 1;
            }
        }
        // Matrisi ekrana yazdırma
        System.out.println("Matrisin Icerigi:");
        for (int satir = 0; satir < 3; satir++) {
            for (int sutun = 0; sutun < 3; sutun++) {
                System.out.print(matris[satir][sutun] + " ");
            }
            System.out.println(); // Yeni satıra geçme
        }
    }
}
```

```
Matrisin Icerigi:
1 2 3
4 5 6
7 8 9
```

Doldurma döngüsünde her hücreye `satir * 3 + sutun + 1` yazılır. Bu ifade hücreleri satır satır, 1'den 9'a numaralar. `satir * 3`, önceki satırlardaki hücre sayısıdır; her satırda üç hücre vardır. `sutun` satır içindeki konumu ekler. `+ 1` ise numaralamanın 0'dan değil 1'den başlamasını sağlar.

`satir`	`sutun` = 0	`sutun` = 1	`sutun` = 2
0	0 + 0 + 1 = 1	0 + 1 + 1 = 2	0 + 2 + 1 = 3
1	3 + 0 + 1 = 4	3 + 1 + 1 = 5	3 + 2 + 1 = 6
2	6 + 0 + 1 = 7	6 + 1 + 1 = 8	6 + 2 + 1 = 9

Tablo 3.3.17: Doldurma döngüsünün izlemesi; `satir * 3 + sutun + 1`.



Şekil 3.3.27: Matris doldurma. (a) `new int[3][3]` : bütün hücreler 0. (b) İlk satır doldu: 1, 2, 3. (c) İkinci satır doldu. (d) Son hücre `matris[2][2]` = 9.

Hücreler satır satır dolar: önce `[0][0]`, `[0][1]`, `[0][2]`; sonra iç döngü biter, dış döngü bir sonraki satıra geçer. Bu sıraya satır öncelikli gezinme denir. Yazdırma döngüsü aynı sırayla gezinir. Her satırın öğeleri `print` ile yan yana yazılır; iç döngü bitince dış döngüdeki `println()` alt satıra geçer. `println()` iç döngünün içine taşınsaydı her sayı ayrı bir satıra yazılırdı.

İç içe döngülerin maliyeti satır sayısı ile sütun sayısının çarpımıdır. $r \times c$'lik bir matriste gezinmek $r \cdot c$ adım ister. Kare $n \times n$ matris için bu $O(n^2)$'dir. Matrisin kendisi de n^2 öğe taşır; her öğeye bir kez dokunmanın daha ucuz bir yolu yoktur.

Örnek 3.3.10

`int[][] m = {{1, 2, 3}, {4, 5, 6}};` matrisinin her satırının toplamı bir diziye yazılsın.

Sonuç dizisi satır sayısı kadar hücre ister: `int[] t = new int[m.length];`. İç içe döngüde `t[i] += m[i][j];` yazılır. İç döngünün sınırı `m[i].length` 'tir.

İlk satırın toplamı 1 + 2 + 3 = 6, ikinci satırınki 4 + 5 + 6 = 15'tir. Sonuç `t = [6, 15]` olur. Aynı yöntemle iki matrisin toplamı da hesaplanır: `c[i][j] = a[i][j] + b[i][j]`. `{{1, 2}, {3, 4}}` ile `{{5, 6}, {7, 8}}` 'in toplamı `[[6, 8], [10, 12]]` 'dir.

3.3.6.5 Üç ve daha fazla boyut

Java'da üç ve daha fazla boyutlu diziler de kullanılabilir. Her ek boyut bir köşeli parantez daha demektir. Üç boyutlu dizi, iki boyutlu dizilerin dizisidir.

```
// 3x3x3 boyutunda üç boyutlu dizi
int[][][] ucBoyutluDizi = new int[3][3][3];
ucBoyutluDizi[1][2][0] = 16; // katman 1, satır 2, sütun 0
```



Şekil 3.3.28: 3×3×3 dizi: her biri 3×3 bir matris olan üç katman. `ucBoyutluDizi[1][2][0]` ikinci katmanın üçüncü satırının ilk hücresidir.

Daha fazla boyut daha karmaşık verilere izin verir. Bir binanın odaları `[kat][satir][sutun]`, renkli bir resim `[y][x][kanal]`, zamanla değişen bir ölçüm tablosu `[zaman][satir][sutun]` biçiminde tutulabilir. Öğe sayısı boyutların çarpımıdır: `new int[3][3][3]` 3 × 3 × 3 = 27 hücre içerir. Boyut sayısı arttıkça bellek hızla büyür. 1920 × 1080'lik renkli bir resim `int[1080][1920][3]` olarak tutulursa 6.220.800 hücre, yaklaşık 25 MB gerekir.

```
int[][][] ucBoyutluDizi = new int[3][3][3];
for (int x = 0; x < 3; x++) {
    for (int y = 0; y < 3; y++) {
        for (int z = 0; z < 3; z++) {
            ucBoyutluDizi[x][y][z] = x * 9 + y * 3 + z + 1;
        }
    }
}
System.out.println("Uc Boyutlu Dizinin Icerigi:");
for (int x = 0; x < 3; x++) {
    for (int y = 0; y < 3; y++) {
        for (int z = 0; z < 3; z++) {
            System.out.print(ucBoyutluDizi[x][y][z] + " ");
        }
        System.out.println();
    }
    System.out.println();
}
```

```
Uc Boyutlu Dizinin Icerigi:
1 2 3
4 5 6
7 8 9

10 11 12
13 14 15
16 17 18

19 20 21
22 23 24
25 26 27
```

Üç iç içe döngü dizide gezinir. En içteki `z` bir satırın sütunları, ortadaki `y` bir katmanın satırları, en dıştaki `x` katmanlar üzerinde gezinir. Her hücreye `x * 9 + y * 3 + z + 1` yazılır. 9, bir katmandaki hücre sayısıdır; formül, matris doldurmadaki `satir * 3 + sutun + 1` kalıbının bir boyut genişletilmiş hâlidir. Sonuç olarak hücreler 1'den 27'ye numaralanır. `ucBoyutluDizi[1][2][0]` = 1 · 9 + 2 · 3 + 0 + 1 = 16'dır. Her katmandan sonra dış döngüdeki `println()` boş bir satır basar ve katmanlar çıktıda birbirinden ayrılır.

3.3.6.6 Matris transpozu

Tanım 3.3.14

Bir *M* matrisinin **transpozu** *T*, satırları sütuna, sütunları satıra dönüşmüş matristir: *T[j][i] = M[i][j]*. *r* × *c*'lik bir matrisin transpozu *c* × *r*'liktir.

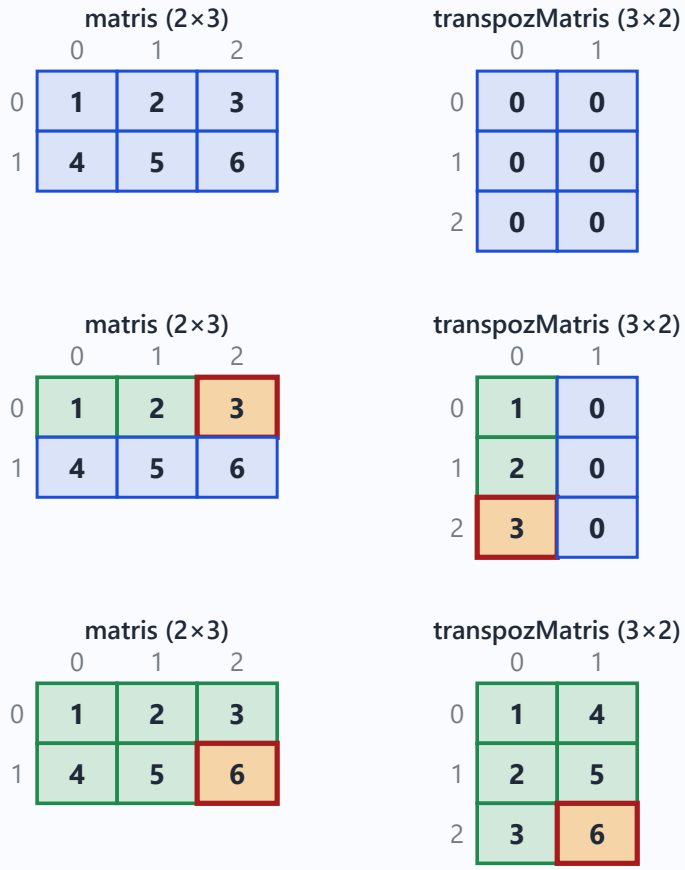
```
public static void main(String[] args) {
    int[][] matris = {
        {1, 2, 3},
        {4, 5, 6}
    };
    System.out.println("Orijinal Matris:");
    yazdirMatris(matris);
    int satirSayisi = matris.length;
    int sutunSayisi = matris[0].length;
    int[][] transpozMatris = new int[sutunSayisi][satirSayisi];
    for (int i = 0; i < satirSayisi; i++) {
        for (int j = 0; j < sutunSayisi; j++) {
            transpozMatris[j][i] = matris[i][j];
        }
    }
    System.out.println("Transpoz Matris:");
    yazdirMatris(transpozMatris);
}
```

```
Orijinal Matris:
1 2 3
4 5 6
Transpoz Matris:
1 4
2 5
3 6
```

Program önce boyutları diziden okur: `satirSayisi = matris.length` = 2, `sutunSayisi = matris[0].length` = 3. Transpoz için yeni bir dizi gerekir ve bu dizinin boyutları ters sıradadır: `new int[sutunSayisi][satirSayisi]`, yani 3 × 2. Kare olmayan bir matris yerinde transpoz edilemez, çünkü sonuç farklı biçimdedir.

i	j	Okunan	Yazılan
0	0	`matris[0][0]` = 1	`transpozMatris[0][0]`
0	1	`matris[0][1]` = 2	`transpozMatris[1][0]`
0	2	`matris[0][2]` = 3	`transpozMatris[2][0]`
1	0	`matris[1][0]` = 4	`transpozMatris[0][1]`
1	1	`matris[1][1]` = 5	`transpozMatris[1][1]`
1	2	`matris[1][2]` = 6	`transpozMatris[2][1]`

Tablo 3.3.18: Transpozun hücre hücre kopyalanması.



Şekil 3.3.29: Transpoz. (a) 2×3 `matris` ve 0'larla dolu 3×2 `transpozMatris`. (b) `matris`'in ilk satırı, transpozun ilk sütunu oldu. (c) Son kopya: `matris[1][2]` = 6, `transpozMatris[2][1]`'e yazılır.

Döngüler asıl matriste satır satır gezinir. Her öğe, transpozda indisleri yer değiştirmiş hücreye yazılır. Tablo 3.3.18 bu yer değiştirmeyi gösterir: `i` = 0 iken `matris`'in ilk satırı okunur ve `transpozMatris`'in ilk sütununa yazılır. Köşegendeki `[1][1]` hücresi yerinde kalır. İş miktarı öğe sayısıyla orantılıdır: $r \cdot c$ kopya.

Sık yapılan hata

Transpoz dizisini asıl matrisle aynı boyutta oluşturmak: `new int[satirSayisi][sutunSayisi]`. 2×3'lük bir matris için bu, 2×3'lük bir dizi demektir. `transpozMatris[2][0]`'a yazılmak istendiğinde satır indisi 2 sınır dışıdır ve `ArrayIndexOutOfBoundsException` oluşur. Kare matrislerde bu hata görünmez; kod ancak kare olmayan bir matrisle denendiğinde ortaya çıkar.

3.3.6.7 Yardımcı metot: `yazdirMatris`

Transpoz programı matrisi iki kez yazdırır: önce asıl matrisi, sonra transpozu. Yazdırma kodunu iki kez yazmak yerine iş bir yardımcı metoda taşınmıştır:

```
public static void yazdirMatris(int[][] matris) {
    for (int i = 0; i < matris.length; i++) {
        for (int j = 0; j < matris[i].length; j++) {
            System.out.print(matris[i][j] + " ");
        }
        System.out.println();
    }
}
```

Metodun parametresi `int[][] matris`'tir ve her boyuttaki iki boyutlu diziyle çalışır. Bunun nedeni, satır sayısının `matris.length`'ten, sütun sayısının `matris[i].length`'ten, yani diziden okunmasıdır. Aynı metot 2×3'lük asıl matrisi de 3×2'lik transpozu da doğru yazdırır. Düzensiz bir dizi verilse onu da doğru yazdırır.

Metoda dizinin kopyası değil, referansı gider. Bu yüzden büyük bir matris için bile çağrı ucuzdur: yalnız bir adres kopyalanır. Metot diziyi değiştirmediği için referansla geçirmenin bir yan etkisi de yoktur.

Sınırlar sabit yazılsaydı ne olurdu? Döngüler `i < 3` ve `j < 3` ile yazılmış bir yazdırma metodu 3×2'lik transpoz matrisine uygulandığında ilk satırın üçüncü öğesine erişmeye çalışır:

```
0 0 Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 2 out of
bounds for length 2
        at TranspozSabit.yazdir(TranspozSabit.java:9)
        at TranspozSabit.main(TranspozSabit.java:4)
```

Çıktıda iki 0 yazıldıktan sonra istisna oluşur. Yığın izi iki satırlıdır: hata `yazdir` metodunun 9. satırındadır ve bu metot `main`'in 4. satırından çağrılmıştır. Yığın izi en içteki çağrıdan dışarı doğru okunur. Sabit sayı yerine `length` kullanmak, metodu her boyutta doğru çalışır hâle getirir.

3.3.7 Uygulama: Tic Tac Toe

Bu kesim, diziler hakkında öğrenilenleri tek bir programda birleştirir: iki boyutlu dizi, iç içe döngüler, metoda dizi geçirme, kullanıcıdan girdi ve sınır denetimi. Program, iki kişinin aynı bilgisayarda oynadığı Tic Tac Toe (X-O) oyunudur.

3.3.7.1 Oyun ve veri yapısı

Oyun 3×3'lük bir tahtada oynanır. İki oyuncu, X ve O, sırayla boş bir hücreye kendi işaretini koyar. Kendi işaretinden üç tanesini aynı satıra, aynı sütuna ya da aynı çapraza yerleştiren oyunu kazanır. Boş hücre kalmazsa ve kimse kazanmamışsa oyun berabere biter.

Tahta için en doğal veri yapısı `char[][] tahta = new char[3][3];`'tür. Her hücre bir karakter tutar: `'X'`, `'O'` ya da boş hücre için `' '`. Bir hamle tek bir atamadır: `tahta[satir][sutun] = oyuncu`. Oyunun bütün durumu, tahta ve sıranın kimde olduğu bilgisinden ibarettir.

3.3.7.2 `main` ve metotlara bölme

```
import java.util.Scanner;

public class TicTacToe {
    static Scanner scanner = new Scanner(System.in);

    public static void main(String[] args) {
        char[][] tahta = new char[3][3]; // 3x3'lük tahta
        char oyuncu = 'X'; // İlk oyuncu X
        boolean oyunDevamEdiyor = true;
        tahtaDoldur(tahta); // Tüm hücreler boş
        while (oyunDevamEdiyor) {
            tahtayiGoster(tahta);
            hamleYap(tahta, oyuncu);
            oyunDevamEdiyor = oyunDevamEdiyorMu(tahta, oyuncu);
            // Oyuncu değişimi
            oyuncu = (oyuncu == 'X') ? 'O' : 'X';
        }
        tahtayiGoster(tahta);
    }
    // ... metotlar
}
```

`main` kısa ve okunaklıdır, çünkü iş dört metoda bölünmüştür: tahtayı doldurmak, göstermek, hamle almak ve oyunun sürüp sürmediğini denetlemek. Döngü, `oyunDevamEdiyor` doğru oldukça bu adımları tekrarlar. Her turun sonunda sıra öteki oyuncuya geçer. `(oyuncu == 'X') ? 'O' : 'X'` koşul işlecidir: oyuncu X ise O, değilse X değerini verir.

Tahta metotlara referansla gider. `hamleYap` tahtanın bir hücresine yazdığında değişiklik `main`'deki tahtada görünür, çünkü ortada tek bir dizi vardır. Bu, dizilerin referans tipi olmasının işe yarayan yüzüdür: metotlar ortak bir durumu değiştirebilir.

`Scanner` nesnesi `static` bir alan olarak sınıfın en üstünde tanımlanmıştır. Böylece bütün metotlar aynı `Scanner`'ı kullanır. Metotların kendisi de `static`'tir. `main` static olduğu için, bir nesne oluşturmadan çağırdığı metotların da static olması gerekir. `static` sözcüğü unutulursa derleyici şu iletiyi verir:

```
E_Static.java:4: error: non-static method tahtaDoldur(char[][]) cannot be referenced from
a static context
        tahtaDoldur(tahta);
        ^
1 error
```

3.3.7.3 Tahtayı doldurma ve gösterme

```
public static void tahtaDoldur(char[][] tahta) {
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            tahta[i][j] = ' ';
        }
    }
}

public static void tahtayiGoster(char[][] tahta) {
    System.out.println("-------------");
    for (int i = 0; i < 3; i++) {
        System.out.print("| ");
        for (int j = 0; j < 3; j++) {
            System.out.print(tahta[i][j] + " | ");
        }
        System.out.println("\n-------------");
    }
}
```

Boş tahta şöyle görünür:

```
-------------
|   |   |   |
-------------
|   |   |   |
-------------
|   |   |   |
-------------
```

`new char[3][3]` ile oluşturulan tahtanın hücreleri `'\u0000'` ile başlar. Bu karakter ekranda görünmez ya da bazı uçbirimlerde garip bir simge olarak çıkar. Ayrıca boş hücre denetiminde `' '` ile karşılaştırma yapılacaktır. `tahtaDoldur` bu yüzden bütün hücrelere önce boşluk karakteri yazar.

`tahtayiGoster` her satırı `| X | O | |` biçiminde yazar. Satırın başında `"| "`, her hücreden sonra `" | "` basılır. Satır sonundaki `"\n-------------"` dizgisi iki işi tek `println` ile yapar: `\n` satırı bitirir, ardından ayırıcı çizgi gelir ve `println` bir alt satıra geçer.

3.3.7.4 Hamle alma

```
static Scanner scanner = new Scanner(System.in);

public static void hamleYap(char[][] tahta, char oyuncu) {
    int satir, sutun;
    do {
        System.out.print("Sıra " + oyuncu
                + " oyuncusunda. Satır ve sütun seçin (1-3): ");
        satir = scanner.nextInt() - 1;
        sutun = scanner.nextInt() - 1;
    } while (satir < 0 || satir > 2 || sutun < 0 || sutun > 2
            || tahta[satir][sutun] != ' ');
    tahta[satir][sutun] = oyuncu;
}
```

Kullanıcı satır ve sütunu 1–3 arasında girer, çünkü insanlar 1'den sayar. Kod `- 1` ile bu değerleri 0–2 indislerine çevirir. Girdi `do-while` döngüsüyle alınır. `do-while`, gövdeyi en az bir kez çalıştırır ve koşulu sonda denetler. Soru önce sorulur; geçersiz ya da dolu bir hücre seçildiyse koşul doğru çıkar ve soru yeniden sorulur.

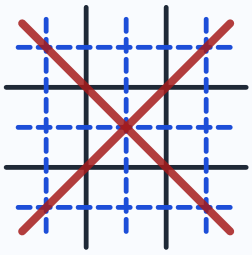
Koşuldaki sıralama bilinçlidir. Önce dört sınır denetimi, en son hücre denetimi yazılmıştır. `||` işleci **kısa devre** (*short-circuit*) değerlendirme yapar: soldan sağa ilerler ve ilk doğru koşulda durur. Kullanıcı 4 5 girerse `satir` 3, `sutun` 4 olur. `satir > 2` doğrudur ve değerlendirme orada durur. `tahta[3][4]` hiç okunmaz, istisna oluşmaz ve soru yeniden sorulur. Hücre denetimi sınır denetimlerinden önce yazılsaydı program `ArrayIndexOutOfBoundsException` ile çökerdi.

> **Sık yapılan hata**
>
> Koşulu `tahta[satir][sutun] != ' ' || satir < 0 || …` sırasıyla yazmak. Kısa devre yalnız soldan sağa işler; hücreye erişim en solda olduğu için sınır denetimlerinden önce yapılır. Geçerli girdilerle program sorunsuz çalışır, hata ancak kullanıcı aralık dışı bir sayı girdiğinde ortaya çıkar. Testte sınır değerleri (0, 4, −1) bu yüzden ayrıca denenmelidir.

3.3.7.5 Kazanan ve beraberlik

Oyun her hamleden sonra iki soruyu cevaplar: kazanan var mı, boş hücre kaldı mı? `oyunDevamEdiyorMu` metodu önce kazanmayı denetler:

```
public static boolean oyunDevamEdiyorMu(char[][] tahta, char oyuncu) {
    // Kazanan durumlarını kontrol et
    if ((tahta[0][0] == oyuncu && tahta[0][1] == oyuncu && tahta[0][2] == oyuncu) ||
        (tahta[1][0] == oyuncu && tahta[1][1] == oyuncu && tahta[1][2] == oyuncu) ||
        (tahta[2][0] == oyuncu && tahta[2][1] == oyuncu && tahta[2][2] == oyuncu) ||
        (tahta[0][0] == oyuncu && tahta[1][0] == oyuncu && tahta[2][0] == oyuncu) ||
        (tahta[0][1] == oyuncu && tahta[1][1] == oyuncu && tahta[2][1] == oyuncu) ||
        (tahta[0][2] == oyuncu && tahta[1][2] == oyuncu && tahta[2][2] == oyuncu) ||
        (tahta[0][0] == oyuncu && tahta[1][1] == oyuncu && tahta[2][2] == oyuncu) ||
        (tahta[0][2] == oyuncu && tahta[1][1] == oyuncu && tahta[2][0] == oyuncu)) {
        System.out.println("Tebrikler, " + oyuncu + " oyuncusu kazandı!");
        return false;
    }
    // ... beraberе kontrolü
```



3 satır + 3 sütun + 2 çapraz

Şekil 3.3.30: Sekiz kazanma çizgisi: 3 satır, 3 sütun ve 2 çapraz.

Sekiz kazanma çizgisi tek tek denetlenir. Her çizgi üç hücrenin de oyuncunun işaretini taşımasını ister; bu koşullar `&&` ile bağlanır. Çizgilerden biri yeterlidir; çizgiler `||` ile bağlanır. Yalnız son hamleyi yapan oyuncu denetlenir. Yeni bir çizgiyi ancak o tamamlamış olabilir; öteki oyuncu kazanmış olsaydı oyun bir önceki hamlede biterdi.

Kazanan varsa metot `false` döndürür. Bu ilk bakışta ters görünür, ama metodun adı “oyun devam ediyor mu?” sorusudur ve cevap “hayır”dır. Uzun koşul ifadesi döngülerle de yazılabilirdi: üç satır ve üç sütun için bir döngü, iki çapraz için ayrı denetim. Açık yazım uzun, ama hangi çizgilerin denetlendiğini doğrudan gösterir.

Kazanan yoksa beraberlik denetlenir:

```
// oyunDevamEdiyorMu metodunun devamı
// Berabere durumunu kontrol et
boolean berabere = true;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (tahta[i][j] == ' ') {
            berabere = false;
            break;
        }
    }
    if (!berabere) {
        break;
    }
}
if (berabere) {
    System.out.println("Oyun berabere bitti.");
    return false;
}
return true;
```

Bu kod bir varsayımla başlar: oyun berabere bitmiştir (`berabere = true`). Tek bir boş hücre bu varsayımı çürütür. İç döngü boş hücre bulduğunda `berabere` 'yi `false` yapar ve `break` ile çıkar. `break` yalnız içinde bulunduğu döngüden çıkar. Dış döngü çalışmayı sürdürür. Bu yüzden dış döngünün gövdesinde ikinci bir denetim gerekir: `if (!berabere) break;` .

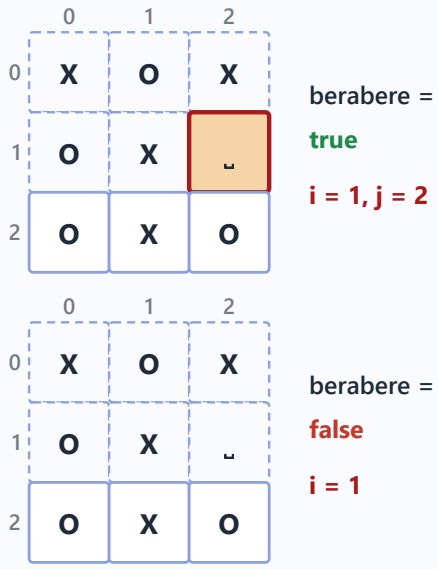
Denetimlerin sırası da önemlidir. Önce kazanma, sonra beraberlik denetlenir. Son hamle tahtayı doldurup aynı anda bir çizgiyi de tamamlayabilir. Sıra ters olsaydı böyle bir oyun yanlışlıkla berabere ilan edilirdi.

İki `break` 'in nasıl çalıştığını görmek için yalnız beraberlik kodunu, tek boş hücresi olan bir tahtayla izleyelim:

```
boolean berabere = true;
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 3; j++) {
        if (tahta[i][j] == ' ') {
            berabere = false;
            break;
        }
    }
    if (!berabere) {
        break;
    }
}
System.out.println(berabere ? "berabere" : "devam");
```

`i`	`j`	`tahta[i][j]`	**İşlem**	`berabere`
0	0	`'X'`	boş değil	true
0	1	`'O'`	boş değil	true
0	2	`'X'`	iç döngü kendiliğinden biter; `!berabere` yanlış	true
1	0	`'O'`	boş değil	true
1	1	`'X'`	boş değil	true
1	2	`' '`	boş hücre: `berabere = false`, iç `break`	false
1	—	—	`!berabere` doğru: dış `break`	false

Tablo 3.3.19: Beraberlik denetiminin izlemesi; tahta `X O X` · `O X ␣` · `O X O` (␣: boşluk).



Şekil 3.3.31: Beraberlik denetimi. (a) `(1, 2)` hücresinde boşluk bulundu. (b) Dış `break`: satır 2 hiç denetlenmez.

Program `devam` yazar. Dokuz hücreden yalnız altısına bakılmıştır; satır 2 hiç denetlenmez. Boş hücre bulunduktan sonra aramayı sürdürmek gereksizdir, çünkü sonuç artık değişmez. Bu tür bir "ilk karşı örnekte dur" kalıbı, doğrusal aramadaki `break`'in iki boyutlu karşılığıdır.

3.3.7.6 Örnek oyun

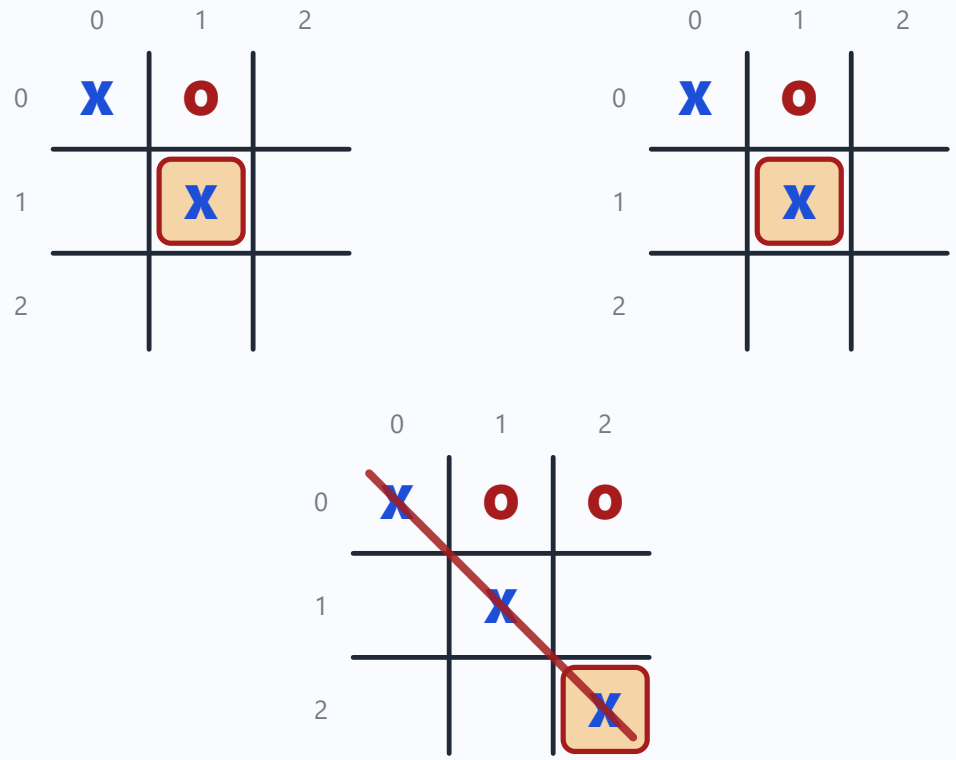
Aşağıdaki girdilerle bir oyun oynansın. Her çift, sırası gelen oyuncunun satır ve sütun seçimidir:

```
1 1   1 2   2 2   2 2   1 3   3 3
```

Oyuncu	Girdi	Atama	Sonuç
X	1 1	`tahta[0][0] = 'X'`	kazanan yok, boş hücre var; sıra O'ya geçer
O	1 2	`tahta[0][1] = 'O'`	sıra X'e geçer

Oyuncu	Girdi	Atama	Sonuç
X	2 2	`tahta[1][1] = 'X'`	sıra O'ya geçer
O	2 2	—	`tahta[1][1]` dolu: soru yeniden sorulur
O	1 3	`tahta[0][2] = 'O'`	sıra X'e geçer
X	3 3	`tahta[2][2] = 'X'`	ana çapraz X: kazanma iletisi, döngü biter

Tablo 3.3.20: Örnek oyunun hamleleri.



Şekil 3.3.32: Örnek oyun. (a) Üç hamleden sonra tahta. (b) O, dolu hücre (2, 2)'yi seçer; `do-while` koşulu doğru çıkar ve soru yeniden sorulur. (c) X ana çaprazı tamamlar.

Programın son çıktısı:

```
Tebrikler, X oyuncusu kazandı!
-------------
| X | O | O |
-------------
|   | X |   |
-------------
|   |   | X |
-------------
```

Dördüncü hamlede O'nun seçtiği hücre doludur. `tahta[1][1] != ' '` doğru çıkar ve `do-while` aynı soruyu bir kez daha sorar. Bu sırada sıra değişmez, çünkü `hamleYap` geçerli bir hamle yapılmadan dönmez. Son hamlede X ana çaprazı (`[0][0]`, `[1][1]`, `[2][2]`) tamamlar. `oyunDevamEdiyorMu` kazanma iletisini yazar ve `false` döndürür. `main`'deki döngü biter ve tahta son kez gösterilir.

Programın geliştirilebilecek yönleri vardır. Kullanıcı sayı yerine harf girerse `nextInt()` `InputMismatchException` fırlatır ve program durur. Beraberlik denetimi ayrı bir `tahtaDoluMu` metoduna taşınabilir; böylece `oyunDevamEdiyorMu` kısalır ve her metot tek bir soruyu cevaplar. Kazanma denetimi de son hamlenin satırına, sütununa ve gerekiyorsa çaprazlarına bakacak biçimde daraltılabilir.

3.3.8 Sık yapılan hatalar

Dizilerle ilgili hatalar iki gruba ayrılır. Derleme hataları program çalışmadan önce javac'ın verdiği iletilerle görünür ve düzeltilmesi görece kolaydır. Çalışma zamanı hataları ise program çalışırken istisnayla ortaya çıkar ya da hiç ileti vermeden yanlış sonuç üretir. Aşağıdaki iki tablo, notun gövdesinde ayrı ayrı işlenen hataları bir arada verir. İletiler JDK 21'in javac ve java araçlarıyla alınmıştır.

Hata	Kod	javac iletisi	Düzeltme
Sonradan kısa ilk değer verme	`a = {1, 2, 3};`	`illegal start of expression`	`a = new int[]{1, 2, 3};`
Hem boyut hem liste	`new int[3]{1, 2, 3}`	`array creation with both dimension expression and initialization is illegal`	Yalnız biri verilir.
Boyutu unutmak	`new int[]`	`array dimension missing`	`new int[n]`
Yanlış tipte öğe	`int[] a = {1, 2.5, 3};`	`incompatible types: possible lossy conversion from double to int`	`double[]` ya da tam sayı değerler
`length()` yazmak	`a.length()`	`cannot find symbol … method length()`	`a.length`
C tarzı boyut	`int a[3] = {…};`	`']' expected`	`int[] a = {…};`
Dalın içinde tanımlanan değişken	`if (…) { double ortanca = …; }`	`cannot find symbol … variable ortanca`	Değişken dalların dışında tanımlanır.
`static` olmayan metot	`public void tahtaDoldur(…)`	`non-static method … cannot be referenced from a static context`	`public static void …`

Tablo 3.3.21: Dizilerle ilgili sık derleme hataları.

Hata	Sonuç	Düzeltme
Son indisi `length` sanmak: `a[a.length]`, `i <= a.length`	`ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3`	`a[a.length - 1]`, `i < a.length`
Oluşturulmamış dizi: `int[] a = null; a[0]`	`NullPointerException: Cannot load from int array …`	Önce `new` ile dizi oluşturulur.
Diziyi doğrudan yazdırmak: `println(a)`	`[I@1c20c684` gibi bir kimlik	`Arrays.toString(a)`
Atamayı kopya sanmak: `b = a`	`b[0]`'ı değiştirmek `a[0]`'ı da değiştirir.	`a.clone()`, `Arrays.copyOf`

Hata	Sonuç	Düzeltme
`==` ile içerik karşılaştırmak	Aynı öğeli iki dizi için `false`	`Arrays.equals(a, b)`
İlk indisi 1 sanmak	İlk öğe atlanır ya da sınır aşılır.	Kodda `dizi[0]`; ekranda `(i + 1) + ". eleman"`
`for-each` ile diziye yazmak	Dizi değişmez; yalnız kopya değişir.	Klasik `for` ve `dizi[i] = …`
Sıralanmamış dizide ikili arama	Anlamsız sonuç, uyarı yok	Önce `Arrays.sort`
Tam sayı bölmesiyle ortalama	29 / 6 = 4, sonra 4.0	`toplam / (double) n`

Tablo 3.3.22: Dizilerle ilgili sık çalışma zamanı ve mantık hataları.

Bu hataların çoğu iki kaynağa iner. Birincisi indisin 0'dan başlaması ve son indisin `length - 1` olmasıdır. İkincisi dizi değişkeninin değer değil referans tutmasıdır. Bir dizi kodu yazılırken bu iki nokta denetlenirse hataların büyük kısmı daha yazılırken önlenir. Kod denenirken de sınır durumları ayrıca sınanmalıdır: boş dizi, tek öğeli dizi, hedefin ilk ve son öğe olduğu durumlar, tek ve çift uzunluk.

3.3.9 Özet

- Dizi, aynı tipten, sabit sayıda öğeyi ardışık hücrelerde tutan yapıdır. Öğelere 0'dan başlayan indisle erişilir; *n* öğeli dizinin son indisi *n − 1*'dir.
- Dizi `new int[n]` ile oluşturulur ve hücreler varsayılan değer alır. Bu değer sayılar için 0 ya da 0.0, `boolean` için `false`, `char` için `'\u0000'`, nesneler için `null`'dır.
- İlklendirici listesi `{1, 2, 3}` yalnız tanımlama satırında kullanılır; başka yerde `new int[]{1, 2, 3}` yazılır. Boyut ile liste birlikte verilemez.
- `length` bir özelliktir ve parantezsiz yazılır. Dizinin bütün indislerinde `for (int i = 0; i < dizi.length; i++)` kalıbıyla gezinilir.
- `for-each` döngüsü her turda sıradaki öğenin kopyasını verir. Okumak için uygundur; diziye yazmak ve indis gerektiren işler için klasik `for` kullanılır.
- Dizi değişkeni bir referanstır. `b = a` aynı diziyi gösteren ikinci bir ad oluşturur; `clone()` ve `Arrays.copyOf` yeni dizi oluşturur. `==` adresleri, `Arrays.equals` içerikleri karşılaştırır.
- Metoda dizi geçirildiğinde referansın kopyası gider. Metot hücreleri değiştirirse değişiklik çağıranda görünür; parametreye yeni dizi atamak ise çağıranı etkilemez.
- Bütün diziler heap'tedir; yığında yalnız referanslar durur. Erişilemez dizileri çöp toplayıcı temizler.
- Sınır dışı bir indis `ArrayIndexOutOfBoundsException` doğurur. İleti hatalı indisi, uzunluğu ve satır numarasını verir.
- Ortalama tek geçişle O(n)'de, ortanca sıralama sonrası ortadaki öğeyle, mod sıralı dizide en uzun seriyle bulunur. Ortanca ve mod sıralama yüzünden O(n log n)'dir.
- Erişim O(1), ikili arama O(log n), doğrusal arama ve tersine çevirme O(n), eşleştirme, seçimli sıralama ve kabarcık sıralaması $O(n^2)$'dir. İkili arama yalnız sıralı dizide çalışır.

- Çok boyutlu dizi dizilerin dizisidir. `m.length` satır, `m[i].length` o satırın sütun sayısıdır. Satırlar farklı uzunlukta olabilir; döngülerde bu yüzden `m[i].length` kullanılır.
- Tic Tac Toe birçok tekniği birleştirir: `char[][]` tahta, referansla paylaşılan durum, `do-while` ile girdi doğrulama, kısa devre ile sınır denetimi ve iki `break` ile iç içe döngüden çıkma.

3.3.10 Kavram listesi

dizi *array* · öğe *element* · indis *index* · ilklendirici listesi *array initializer* · varsayılan değer *default value* · for-each döngüsü *for-each loop* · referans tipi *reference type* · null *null reference* · değer ile aktarım *pass by value* · yığın *stack (memory)* · heap *heap (memory)* · çöp toplayıcı *garbage collector* · dinamik dizi *dynamic array* · doğrusal arama *linear search* · ikili arama *binary search* · zaman karmaşıklığı (O gösterimi) *time complexity (Big-O)* · seçimli sıralama *selection sort* · kabarcık sıralaması *bubble sort* · iki boyutlu dizi *two-dimensional array* · düzensiz dizi *ragged array* · kısa devre değerlendirme *short-circuit evaluation*

3.3.11 Kendini Dene

1. `int[] d = {3, 8, 1, 9, 4};` iken `for (int i = d.length - 1; i >= 0; i -= 2) s += d[i];` döngüsünden sonra `s` kaçtır? (`s` 0 ile başlar.)

   Cevap

   Döngü son indisten, 4'ten başlar ve ikişer azalır: `i` = 4, 2, 0. Okunan öğeler `d[4]` = 4, `d[2]` = 1 ve `d[0]` = 3'tür. `i` −2 olunca koşul yanlış çıkar. `s` = 4 + 1 + 3 = **8**. Başlangıçta `d.length` yazılsaydı ilk turda `d[5]` 'e erişilir ve istisna oluşurdu.

2. Şu satırlardan sonra ne yazdırılır? `int[] x = {2, 4, 6}; int[] y = x; int[] z = x.clone(); y[1] = 0; z[2] = 9;` ardından `Arrays.toString(x)`, `x == y` ve `Arrays.equals(x, z)`.

   Cevap

   `y`, `x` ile aynı diziyi gösterir; `y[1] = 0` bu diziyi değiştirir ve `x` = `[2, 0, 6]` olur. `z` ayrı bir kopyadır; `z[2] = 9` yalnız kopyayı değiştirir ve `z` = `[2, 4, 9]` olur. `x == y` **true**'dur. `Arrays.equals(x, z)` **false**'tur, çünkü içerikler artık farklıdır.

3. Sıralı `{2, 5, 8, 12, 16, 23, 38, 56, 72, 91}` dizisinde ikili aramayla 23 aranıyor. Hangi orta

indislere bakılır? Kaç karşılaştırma yapılır?

Cevap

`alt` = 0, `ust` = 9 → `orta` = 4 (16); 23 > 16, `alt` = 5. `orta` = (5 + 9) / 2 = 7 (56); 23 < 56, `ust` = 6. `orta` = (5 + 6) / 2 = 5 (23): bulundu. Orta indisler 4, 7 ve 5'tir. **3 karşılaştırma** yapılır ve metot 5 döndürür. 40 aransaydı dördüncü adımda `orta` = 6 (38) olur, ardından alan boşalır ve −1 döner.

4. `{5, 1, 4, 2, 8}` dizisi seçimli sıralamayla sıralanıyor. Her turun sonunda dizi nasıldır? Kaç karşılaştırma yapılır?

Cevap

`i` = 0: en küçük 1 (indis 1), takas: `[1, 5, 4, 2, 8]`. `i` = 1: en küçük 2 (indis 3), takas: `[1, 2, 4, 5, 8]`. `i` = 2: en küçük 4 zaten yerinde. `i` = 3: en küçük 5 zaten yerinde. Dizi ikinci turda sıralanmıştır, ama seçimli sıralama bunu fark etmez. Karşılaştırma sayısı her durumda 4 + 3 + 2 + 1 = **10**'dur.

5. `int[][] m = {{1, 2}, {3, 4}, {5, 6}};` için `m.length`, `m[0].length` ve ikinci sütunun toplamı nedir?

Cevap

`m` üç satırlı bir dizidir: `m.length` = **3**. İlk satır iki öğelidir: `m[0].length` = **2**. İkinci sütunun indisi 1'dir: `m[0][1] + m[1][1] + m[2][1]` = 2 + 4 + 6 = **12**. Döngü `for (int i = 0; i < m.length; i++) t += m[i][1];` biçiminde yazılır; satır değişir, sütun sabittir.

6. `int[][] r = new int[3][]; r[0] = new int[]{1}; r[2] = new int[2];` satırlarından sonra `r[2].length + r[1].length` ne verir?

Cevap

Bir sayı vermez; program `NullPointerException` ile durur. `new int[3][]` yalnız üç satır referansı oluşturur. `r[0]` ve `r[2]`'ye satır atanmıştır, ama `r[1]` hâlâ `null`'dır. `r[1].length` bir `null` referansın alanını okumaya çalışır. İleti `Cannot read the array length because "…[1]" is null` biçimindedir.

7. `static void f(int[] d) { d = new int[]{9, 9}; }` ve `static void g(int[] d) { d[0] = 9; }` verilsin. `int[] a = {1, 2};` iken önce `f(a)`, sonra `g(a)` çağrılıyor. Her çağrıdan sonra `a`

nedir?

Cevap

`f(a)` sonrası `a` hâlâ `[1, 2]`'dir. `f`'deki `d`, referansın kopyasıdır; yeni dizi atamak yalnız bu kopyayı yeni diziye bağlar. `g(a)` sonrası `a` = `[9, 2]` olur. `g`'deki `d` aynı diziyi gösterir ve `d[0] = 9` o dizinin hücresine yazar.

8. Sıralı `{1, 1, 3, 3, 3, 5, 5, 5}` dizisine notta verilen mod algoritması uygulanıyor. Sonuç nedir? `>` yerine `>=` yazılsaydı ne değişirdi?

Cevap

3'ün serisi uzunluk 3'e ulaşınca 3 > 2 doğru çıkar; en uzun seri 3 × 3 olur. 5'in serisi de 3'e ulaşır, ama 3 > 3 yanlıştır. Sonuç **mod 3, tekrar sayısı 3**'tür; eşitlikte ilk bulunan korunur. `>=` ile 5'in serisi eşit uzunlukta en uzun seriyi geçer ve sonuç 5 olurdu. Bu kümenin aslında iki modu vardır.

9. `{2, 1, 3, 4, 5}` dizisi `degisti` bayraklı kabarcık sıralamasıyla kaç turda biter? Bayrak olmasaydı kaç karşılaştırma yapılırdı?

Cevap

İlk turda 2 ile 1 takas edilir (4 karşılaştırma). İkinci turda takas yapılmaz (3 karşılaştırma) ve `break` çalışır: **2 tur, 7 karşılaştırma**. Bayrak olmasaydı bütün turlar çalışırdı: 4 + 3 + 2 + 1 = **10** karşılaştırma.

3.3.12 İleri okuma

- Oracle, *The Java Tutorials*, "Arrays" ve "The for Statement" sayfaları: dizi tanımlamayı, ilklendirici listesini ve çok boyutlu dizileri anlatır. `java.util.Arrays` ve gelişmiş `for` da kısa örneklerle verilir. Java'nın resmî ve ücretsiz başlangıç kaynağıdır.
- A. B. Downey ve C. Mayfield, *Think Java* (2. baskı), Bölüm 7 "Arrays and References": dizileri referans kavramı üzerinden anlatır. Takma ad, kopyalama ve diziyi metoda geçirme konularında bu notu tamamlar.
- P. Deitel ve H. Deitel, *Java: How to Program* (11. baskı), Bölüm 7 "Arrays and ArrayLists": bol örnekli, uzun bir anlatımdır ve diziden `ArrayList`'e geçişi de içerir.
- J. Gosling ve diğerleri, *The Java Language Specification, Java SE 21 Edition*, Bölüm 10 "Arrays": dizi tiplerinin, oluşturmanın, varsayılan değerlerin ve sınır denetiminin kesin tanımıdır. Bir ayrıntıda son söz bu belgededir.

3.3.13 Kaynaklar

1. Oracle, *The Java Tutorials*: “Arrays”; “The for Statement” (enhanced for).
2. Oracle, *Java SE 21 API Specification*: `java.util.Arrays`.
3. J. Gosling ve diğerleri, *The Java Language Specification, Java SE 21 Edition*, Oracle, 2023 (Bölüm 10: Arrays).
4. P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (Bölüm 7: Arrays and ArrayLists).
5. K. Sierra ve B. Bates, *Head First Java*, 3. baskı, O'Reilly, 2022.
6. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson, 2022 (3.10 Arrays).
7. A. B. Downey ve C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019 (Bölüm 7: Arrays and References).

KISIM I · BÖLÜM 3.4

Dizgiler

Programların büyük kısmı metinle çalışır. Bir kullanıcı adı, bir e-posta adresi, bir dosya yolu ya da ekrana yazılan bir ileti hep karakterlerden oluşan bir metindir. Java'da bu tür metinler `String` sınıfının nesneleriyle saklanır. Bu not dizgilerin nasıl oluşturulduğunu, `+` ile birleştirmenin hangi sırayla yürüdüğünü ve karakterlere indisle nasıl erişildiğini anlatır. Ardından en sık kullanılan dizgi metotları gelir. Dizgilerin değiştirilemez olması, Türkçe harflerde yerel ayarın etkisi ve döngüde birleştirme için `StringBuilder` ayrı kesimlerde işlenir. `Scanner` ile klavyeden metin okumanın inceliklerinden sonra dizgilerin karşılaştırılması ele alınır: `==` ile `equals()` arasındaki fark, dizgi havuzu ve `compareTo`. Son kesimler `printf` ile biçimli çıktıyı, palindrom ve kelime frekansı uygulamalarını ve sık yapılan hataları kapsar. Sunumdaki bütün kodlar JDK 21 ile çalıştırılmış, çıktıları ve hata iletileri gerçek çalıştırmadan alınmıştır.

ÖĞRENME ÇIKTILARI

Dizgi oluşturmak
`String` nesnesini dizgi sabitiyle ve `new` ile oluşturmak; `+` birleştirmesinin soldan sağa nasıl işlendiğini açıklamak.

İndis ve metotlar
`charAt`, `length`, `indexOf` ve `substring` ile dizginin parçalarına erişmek; dizgide döngüyle karakter karakter gezinmek.

Değiştirilemezlik
Dizgi metotlarının yeni nesne döndürdüğünü bellek çizimiyle göstermek; yerel ayarın büyük-küçük harf dönüşümüne etkisini bilmek; döngüde `StringBuilder` kullanmak.

Kullanıcı girişi
`next()`, `nextLine()` ve `nextInt()` arasındaki farkı açıklamak; satır sonu tuzağından kaçınmak.

Karşılaştırma
`==` ile `equals()` farkını dizgi havuzuyla açıklamak; `compareTo` ve öteki karşılaştırma metotlarını doğru kullanmak.

Biçimli çıktı
`printf` ve `String.format` ile genişlik, hizalama ve hassasiyeti ayarlamak.

3.4.1 Dizgi kavramı

Bilgisayar her veriyi sayı olarak saklar. Bir harf de bellekte bir sayıdır; `char` tipi tek bir karakterin kodunu tutar. Gerçek programlar ise tek harfle değil, kelimelerle ve cümlelerle çalışır. Bir öğrencinin adı, bir ürünün açıklaması ya da bir sunucudan gelen yanıt, ardışık karakterlerden oluşan birer metindir. Java bu tür metinler için hazır bir sınıf sunar. Bu kesim o sınıfı tanıtır, dizginin nasıl oluşturulduğunu ve karakterlerine nasıl erişildiğini anlatır.

3.4.1.1 Dizgi nedir?

Tanım 3.4.1

Dizgi (*string*), karakterlerden oluşan bir metni saklayan nesnedir. Java'da dizgilerin tipi `java.lang.String` sınıfıdır. Kaynak kodda çift tırnak arasına yazılan `"Tarkan"` gibi bir metin **dizgi sabiti** (*string literal*) adını alır.

Tanımdaki birkaç ayrıntı ilk bakışta gözden kaçar. Birincisi, `String` bir **sınıftır**, ilkel tip değildir. Bu yüzden adı büyük harfle başlar: `int`, `double`, `char` küçük harfle yazılır, `String` ise büyük S ile. Bir `String` değişkeni metnin kendisini değil, metni tutan nesnenin bellekteki yerini, yani bir **referansı** saklar. İkincisi, `String` sınıfı `java.lang` paketindedir. Bu paket her programa kendiliğinden eklenir, dolayısıyla `import` satırı gerekmez. Üçüncüsü, dizgi sabiti yazmak için `new` kullanılmaz. `String ad = "Tarkan";` satırı nesneyi kendisi oluşturur.

Java karakterleri **Unicode** ile saklar. Unicode, dünyadaki yazı sistemlerinin karakterlerine birer numara veren standarttır. Bu sayede `ç`, `ğ`, `ı`, `İ`, `ö`, `ş` ve `ü` gibi Türkçe harfler dizgide sorunsuz durur. Dizgi aslında sıralı karakter kutularından oluşur. Her kutunun bir sıra numarası vardır ve numaralandırma 0'dan başlar (Şekil 3.4.1).



Şekil 3.4.1: `"Tarkan"` dizgisi altı karakter kutusundan oluşur. Kutuların altındaki 0–5 sayıları indislerdir.

Tek karakter ile tek karakterlik dizgi farklı şeylerdir. `'T'` tek tırnakla yazılır ve tipi `char`'dır; bellekte 84 sayısını tutan iki baytlık bir değerdir. `"T"` ise çift tırnakla yazılır ve uzunluğu 1 olan bir `String` nesnesidir. İkisi birbirinin yerine kullanılamaz. `char c = "T";` satırı derlenmez, `String s = 'T';` satırı da derlenmez.

Sunumdaki ilk örnek üç dizgi oluşturur ve yazdırır:

```
String ad = "Tarkan";
String adSoyad = "Mustafa Sandal";
String nokta = "(" + 3 + ", " + 5 + ")";
System.out.println(ad);
System.out.println(adSoyad);
System.out.println(nokta);
```

```
Tarkan
Mustafa Sandal
(3, 5)
```

İlk iki satırda değişkenlere dizgi sabitleri atanır. Boşluk da bir karakterdir: `"Mustafa Sandal"` 14 karakterlik bir dizgidir. Üçüncü satır sayılardan bir dizgi kurar. `"("` bir dizgi olduğu için ardından gelen `3` metne çevrilir ve eklenir; aynı kural `", "`, `5` ve `")"` için de işler. Sonuçta `nokta` değişkeni `"(3, 5)"` metnini gösterir. Bu birleştirme kuralı biraz aşağıda ayrıntılı ele alınıyor, çünkü yeni başlayanların en sık şaşırdığı yer burasıdır.

3.4.1.2 Dizgi oluşturmak

Bir dizgi birkaç yolla oluşturulur. En yaygını dizgi sabitidir. Bunun yanında `String` sınıfının yapıcıları, sayıyı dizgiye çeviren `String.valueOf` metodu ve iki dizgiyi uç uca ekleyen `concat` metodu da kullanılır. Sunumdaki kodun ilk yarısı bu yolları bir arada gösterir:

```
String s0 = "Giresun";            // dizgi sabiti
char[] harfler = {'J', 'a', 'v', 'a'};
String s1 = new String(harfler);  // char[]
String s2 = String.valueOf(42);   // sayıdan
String s3 = "Giresun".concat(" Üni.");
System.out.println(s1 + " " + s2 + " " + s3);

System.out.println("Toplam: " + 3 + 5);
System.out.println("Toplam: " + (3 + 5));
System.out.println(3 + 5 + " elma");
System.out.println('a' + 'b' + "c");
System.out.println("" + 'a' + 'b');
```

```
Java 42 Giresun Üni.
Toplam: 35
Toplam: 8
8 elma
195c
ab
```

Kodun ilk altı satırı şöyle çalışır:

- `String s0 = "Giresun";` en sık görülen biçimdir: dizgi sabiti doğrudan değişkene atanır.
- `new String(harfler)` bir `char` dizisindeki karakterlerden yeni bir dizgi kurar. Dizi `{'J', 'a', 'v', 'a'}` olduğu için `s1` `"Java"` olur. Bu yapıcı, karakterleri tek tek hesaplanan bir metni dizgiye dönüştürmek için kullanışlıdır.
- `String.valueOf(42)` 42 sayısının metin karşılığını, yani `"42"` dizgisini döndürür. `valueOf` bir **sınıf metodudur**: bir nesne üzerinden değil, sınıf adıyla çağrılır.
- `"Giresun".concat(" Üni.")` iki dizgiyi birleştirir ve `"Giresun Üni."` döndürür. Burada metot doğrudan bir dizgi sabiti üzerinden çağrılır; sabit de bir nesne olduğu için bu geçerlidir.

Yazdırma satırı bu üç dizgiyi aralarına boşluk koyarak birleştirir: `Java 42 Giresun Üni.`. Kodun ikinci yarısı ise `+` işlecinin davranışını gösterir. Çıktının ikinci satırındaki `Toplam: 35` çoğu öğrenciyi şaşırtır; nedeni bir sonraki alt kesimin konusudur.

`new String("Giresun")` biçimi de derlenir ve çalışır, ama gereksiz bir nesne üretir. Bu biçim karşılaştırma kesiminde dizgi havuzunu anlatmak için kullanılacak. Gündelik kodda bir sabitten dizgi oluşturmak için `new` yazmanın bir yararı yoktur.

3.4.1.3 Birleştirme ve + işlecinin sırası

> **Tanım 3.4.2**
>
> **Dizgi birleştirme** (*string concatenation*), iki değeri uç uca ekleyerek yeni bir dizgi üretmektir. Java'da `+` işlecinin işlenenlerinden en az biri `String` ise öteki işlenen de dizgiye çevrilir ve iki metin birleştirilir. İki işlenen de sayıysa `+` sayısal toplama yapar.

Aynı `+` işareti iki ayrı iş yapar: toplama ve birleştirme. Hangisinin yapılacağına her işlemde ayrı ayrı karar verilir. Karar için yalnız o anki iki işlenenin tipine bakılır. Ayrıca `+` soldan sağa işlenir. Yani `a + b + c` ifadesi `(a + b) + c` diye hesaplanır. Bu iki kural birlikte düşünülünce sunumdaki çıktıların hepsi açıklanır:

- `"Toplam: " + 3 + 5`: önce `"Toplam: " + 3` hesaplanır. Sol yan dizgi olduğu için 3 metne çevrilir: `"Toplam: 3"`. Ardından `"Toplam: 3" + 5` gelir ve yine birleştirme yapılır. Sonuç `Toplam: 35`'tir.
- `"Toplam: " + (3 + 5)`: parantez önce hesaplanır. İçerideki iki işlenen de `int` olduğu için toplama yapılır, 8 çıkar. Sonra 8 dizgiye eklenir: `Toplam: 8`.
- `3 + 5 + " elma"`: soldaki iki işlenen sayıdır, önce toplanır ve 8 olur. Ancak ondan sonra dizgiyle karşılaşılır: `8 elma`.
- `'a' + 'b' + "c"`: `char` bir sayı tipidir. İki `char` toplanınca kodları toplanır: 97 + 98 = 195. Sonra `"c"` eklenir: `195c`.

- `"" + 'a' + 'b'`: baştaki boş dizgi her şeyi değiştirir. `"" + 'a'` zaten bir dizgidir, `"a"`. Ardından `'b'` metin olarak eklenir: `ab`.

Kural kısaca şudur: ifade soldan sağa okunur. Bir dizgiyle karşılaşılana kadar sayılar toplanır. Bir dizgi ortaya çıktığı andan sonraki her `+` birleştirmedir. Sayıların toplanması isteniyorsa parantez kullanılır.

Sunumdaki izleme slaytı aynı kuralı değişkenlerle gösterir:

```
int a = 4, b = 2;
String s1 = "Sonuç: " + a + b;
String s2 = "Sonuç: " + (a + b);
String s3 = a + b + " kalem";
String s4 = 'x' + 'y' + "z";
String s5 = "" + 'x' + 'y';
System.out.println(s1 + " | " + s2);
System.out.println(s3 + " | " + s4 + " | " + s5);
```

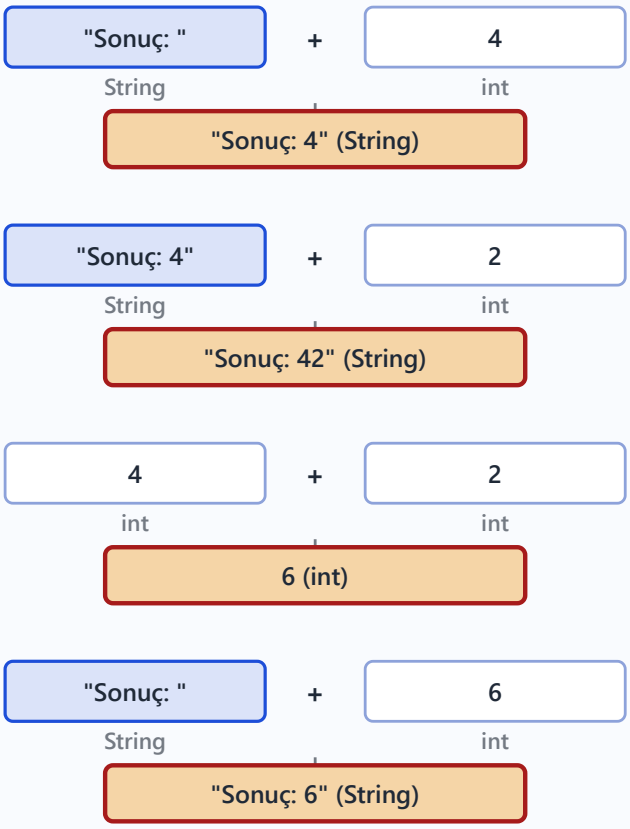
```
Sonuç: 42 | Sonuç: 6
6 kalem | 241z | xy
```

Tablo 3.4.1'deki her satır bir `+` işlemini gösterir. Sütunlar, işlenenlerin tipini ve o adımın sonucunu verir.

Değişken	İşlem	Sol işlenen	Sağ işlenen	Sonuç
`s1`	`"Sonuç: " + a`	String	int	birleştirme: `"Sonuç: 4"`
	`… + b`	String	int	birleştirme: `"Sonuç: 42"`
`s2`	`(a + b)`	int	int	toplama: 6
	`"Sonuç: " + 6`	String	int	birleştirme: `"Sonuç: 6"`
`s3`	`a + b`	int	int	toplama: 6
	`6 + " kalem"`	int	String	birleştirme: `"6 kalem"`
`s4`	`'x' + 'y'`	char	char	toplama: 120 + 121 = 241
	`241 + "z"`	int	String	birleştirme: `"241z"`
`s5`	`"" + 'x'`	String	char	birleştirme: `"x"`
	`"x" + 'y'`	String	char	birleştirme: `"xy"`

Tablo 3.4.1: `a = 4`, `b = 2` iken her `+` işleminin türü ve sonucu. İşlemler soldan sağa yapılır.

`s1` ile `s2` arasındaki fark yalnız bir çift paranteze dayanır, ama sonuç 42 ile 6 kadar farklıdır. Şekil 3.4.2 bu iki satırın ilk adımlarını sunumdaki çizimlerle gösterir. Üstteki kutuların altında işlenenlerin tipi, alttaki kutuda ise o adımın sonucu ve tipi yazılıdır.



Şekil 3.4.2: `s1` ve `s2` satırlarının adımları. (a) Sol yan `String` olduğu için 4 metne eklenir. (b) Sonuç hâlâ `String`'dir; 2 de metin olarak eklenir ve `"Sonuç: 42"` çıkar. (c) Parantez içinde iki `int` toplanır, 6 olur. (d) 6 dizgiye eklenir.

`s4` satırı ayrı bir dikkat ister. `char` değerleri metin gibi görünse de Java onları sayı olarak işler. `'x'` karakterinin Unicode kodu 120, `'y'`'nin kodu 121'dir. İki `char` yan yana toplandığında sonuç bir `int` olur, 241. Bu sayı ancak `"z"` ile karşılaşınca metne çevrilir. `s5`'te ise baştaki `""` ilk işlemi birleştirmeye çevirir ve iki harf metin olarak yan yana gelir. Karakterleri birleştirmek için başa boş dizgi koymak yaygın bir yoldur. Aynı iş `String.valueOf('x') + 'y'` ile de yapılır.

Sık yapılan hata

Ekrana bir toplam yazdırırken parantezi unutmak: `System.out.println("Toplam: " + x + y);` satırı `x` ile `y`'yi toplamaz, yan yana yazar. `x = 10`, `y = 5` iken çıktı `Toplam: 105` olur. Doğrusu `"Toplam: " + (x + y)`'dir. Hata derleyicinin gözünden kaçar, çünkü iki yazım da geçerli Java'dır.

Bir ayrıntı daha var: birleştirme her seferinde **yeni** bir `String` nesnesi üretir. `"Sonuç: " + a + b` ifadesinde mantıksal olarak iki ara dizgi oluşur. Az sayıda parça için bunun bir önemi yoktur. Bir döngüde binlerce kez birleştirme yapıldığında ise maliyet hızla büyür. Değiştirilemezlik kesiminde bu maliyet hesaplanıyor.

3.4.1.4 İndisleme

Tanım 3.4.3

İndis (*index*), bir karakterin dizgideki sıra numarasıdır. İlk karakterin indisi 0, son karakterin indisi `length() - 1`'dir. Bir karaktere indisiyle erişmeye **indisleme** (*indexing*) denir.

İndislemenin iki temel metodu vardır. `length()` dizgideki karakter sayısını döndürür. `charAt(i)` ise `i` indisindeki karakteri bir `char` olarak verir. `"Tarkan"` altı karakterlidir; indisleri 0'dan 5'e kadar gider (Şekil 3.4.3). Uzunluk 6 olduğu hâlde 6 numaralı bir karakter yoktur. Numaralandırma 0'dan başladığı için son indis her zaman uzunluğun bir eksiğidir.



Şekil 3.4.3: İlk karakter `charAt(0)`, son karakter `charAt(length() - 1)`, yani `charAt(5)` ile alınır.

Sunumdaki örnek bu metotları `"Tarkan"` üzerinde dener:

```
String isim = "Tarkan";
System.out.println(isim.length());
System.out.println(isim.charAt(0));
System.out.println(isim.charAt(isim.length() - 1));
char c = isim.charAt(2);
System.out.println(c + 1);
System.out.println((char) (c + 1));
```

```
6
T
n
115
s
```

İlk satır uzunluğu, 6'yı yazar. `charAt(0)` ilk harfi, T'yi verir. Son harf için indis elle yazılmaz, uzunluktan hesaplanır: `isim.length() - 1` = 5 ve `charAt(5)` = n. Bu yazım her uzunluktaki dizgide çalışır. Sonraki satırda `c`, indis 2'deki `'r'` karakterini alır.

Son iki satır `char`'ın bir sayı olduğunu gösterir. `'r'` karakterinin Unicode kodu 114'tür. `c + 1` ifadesinde `char` ile `int` toplanır ve sonuç `int` olur: 115. Ekrana harf değil sayı çıkar. Bir sonraki harfi görmek için sonuç `(char)` ile geri çevrilir: 115 kodlu karakter `s`'dir. Harfleri kaydırmaya dayanan basit şifreleme örnekleri bu dönüşümü kullanır.

`length()` bir **metottur**, yani sonunda parantez vardır. Dizilerde ise uzunluk bir **alandır** ve parantezsiz yazılır: `dizi.length`. İki yazım sık karıştırılır ve ikisi de derleme hatası verir:

```
Uzun.java:4: error: cannot find symbol
        int n = s.length;
                  ^
  symbol:   variable length
  location: variable s of type String
```

```
DiziUzun.java:4: error: cannot find symbol
        int n = d.length();
                  ^
  symbol:   method length()
  location: variable d of type int[]
```

İletilerdeki `symbol` satırı farkı açıkça söyler. Birincide derleyici `String` içinde `length` adlı bir *değişken* arar ve bulamaz. İkincide `int[]` içinde `length()` adlı bir *metot* arar ve yine bulamaz. Bir başka karışıklık da dizgiye dizi gibi köşeli parantezle erişmeye çalışmaktır. `s[0]` yazımı C ve Python'da çalışır ama Java'da derlenmez:

```
Arr.java:4: error: array required, but String found
        char c = s[0];
                  ^
```

Geçersiz bir indis derleme sırasında yakalanmaz, çünkü indisin değeri çoğu zaman program çalışırken belli olur. Dizginin dışına taşan bir indis çalışma zamanında bir istisnaya yol açar:

```
String s = "Java";
System.out.println(s.charAt(4));
```

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
    Index 4 out of bounds for length 4
```

`"Java"` dört karakterlidir; geçerli indisler 0, 1, 2 ve 3'tür. `charAt(4)` çağrısı `StringIndexOutOfBoundsException` fırlatır ve program durur. İletideki "Index 4 out of bounds for length 4" sözü sorunun yerini kesin olarak verir: 4 numaralı indis istendi, uzunluk ise 4. Negatif indis de aynı istisnayı doğurur; `"Java".charAt(-1)` için ileti "Index -1 out of bounds for length 4" olur.

> **Sık yapılan hata**
>
> Son karakteri `s.charAt(s.length())` ile almaya çalışmak. Uzunluk, son indisten bir fazladır ve bu çağrı her dizgide `StringIndexOutOfBoundsException` verir. Aynı hata döngüde `i <= s.length()` koşuluyla da yapılır. Doğru koşul `i < s.length()` 'tir.

3.4.1.5 Karakterlerde döngüyle gezinmek

Bir dizginin bütün karakterlerine tek tek bakmak için `for` döngüsü kullanılır. Döngü değişkeni 0'dan başlar ve uzunluğa ulaşmadan durur. Sunumdaki program kullanıcıdan bir kelime alır ve bu kelimenin tek sıralı harflerini yazar; döngü değişkeni her turda 2 artar:

```
Scanner scanner = new Scanner(System.in);
System.out.print("Bir kelime giriniz: ");
String kelime = scanner.nextLine();
for (int i = 0; i < kelime.length(); i += 2) {
  char harf = kelime.charAt(i);
  System.out.println((i + 1) + ". harf: " + harf);
}
scanner.close();
```

Program `Scanner` ile klavyeden bir satır okur. `Scanner` kullanmak için dosyanın başına `import java.util.Scanner;` satırı eklenir; okuma metotlarının ayrıntısı kullanıcı girişi kesimindedir. `print`, `println` 'den farklı olarak alt satıra geçmez. Bu yüzden kullanıcının yazdığı kelime istemle aynı satırda görünür. Kullanıcı `Giresun` yazıp `Enter`'a bastığında çıktı şöyle olur:

```
Bir kelime giriniz: Giresun
1. harf: G
3. harf: r
5. harf: s
7. harf: n
```

Tablo 3.4.2 döngünün her turunu verir. `"Giresun"` yedi karakterlidir; indisleri 0'dan 6'ya kadar gider. Döngü değişkeni 0, 2, 4, 6 değerlerini alır. Ekrana yazılan sıra numarası insanların alıştığı biçimde 1'den başlasın diye `i + 1` yazdırılır. Buradaki parantez gereklidir: `(i + 1) + ". harf: "` önce

toplamayı yapar. Bu satırda parantez olmasa da sonuç aynı çıkardı, çünkü ifade bir sayıyla başlar. Yine de parantez, toplamanın birleştirmeden önce yapıldığını açıkça gösterir. `"Harf " + i + 1` gibi dizgiyle başlayan bir ifadede ise parantezsiz yazım `i` = 0 iken `Harf 1` yerine `Harf 01` üretir.

Tur	`i`	`i < 7`	`charAt(i)`	Yazılan satır
1	0	doğru	`'G'`	`1. harf: G`
2	2	doğru	`'r'`	`3. harf: r`
3	4	doğru	`'s'`	`5. harf: s`
4	6	doğru	`'n'`	`7. harf: n`
—	8	yanlış	—	döngü biter

Tablo 3.4.2: `kelime = "Giresun"` (uzunluk 7) için döngünün izlenmesi.

Son satır önemlidir. `i` 8 olduğunda koşul yanlış çıkar ve `charAt(8)` hiç çağrılmaz. Koşul `i < kelime.length()` biçiminde yazıldığı için döngü her uzunlukta güvenle durur. Kelime çift uzunlukta olsaydı, örneğin `"Ordu"`, döngü 0 ve 2 indislerini okur ve `i` = 4'te biterdi. Boş bir satır girilirse uzunluk 0'dır ve gövde hiç çalışmaz.

Döngü sonunda `scanner.close()` çağrılır. Bu çağrı `Scanner`'ın kullandığı kaynağı serbest bırakır. Klavyeden okuyan bir `Scanner` kapatıldığında `System.in` da kapanır ve programın geri kalanında klavyeden yeniden okunamaz. Bu yüzden `close()` ancak bütün okumalar bittikten sonra çağrılır.

3.4.2 Dizgi metotları

`String` sınıfı metinle ilgili gündelik işlerin çoğu için hazır metot sunar: bir kelimeyi aramak, metnin bir parçasını almak, harfleri büyütmek, boşlukları temizlemek, metni parçalara bölmek. Bu metotların hepsi bir dizgi nesnesi üzerinden, nokta ile çağrılır: `s.length()`, `s.indexOf("a")`. Ortak bir özellikleri daha var: hiçbiri üzerinde çağrıldığı dizgiyi değiştirmez. Sonuç her zaman yeni bir değer olarak döner.

3.4.2.1 Temel metotlar

Tablo 3.4.3 en sık kullanılan metotları `s = "Java Programlama"` dizgisi üzerinde gösterir. Bu dizgi 16 karakterlidir; dördüncü indisteki boşluk da bir karakter sayılır. Tablodaki bütün sonuçlar JDK 21 ile çalıştırılarak alınmıştır.

Metot	Ne yapar?	Örnek ve sonuç
`length()`	Karakter sayısını döndürür.	`s.length()` → 16
`charAt(i)`	`i` indisindeki karakteri (`char`) döndürür.	`s.charAt(0)` → `'J'`
`indexOf(str)`	Alt dizginin ilk başladığı indis; bulunamazsa −1.	`s.indexOf("Programlama")` → 5 `s.indexOf("C++")` → −1

Metot	Ne yapar?	Örnek ve sonuç
`indexOf(str, bas)`	Aramaya `bas` indisinden başlar.	`s.indexOf("a", 4)` → 10
`lastIndexOf(str)`	Son geçtiği yerin indisi.	`s.lastIndexOf('a')` → 15
`substring(i1, i2)`	`i1` dahil, `i2` hariç aradaki alt dizgi.	`s.substring(5, 12)` → `"Program"`
`substring(i1)`	`i1`'den sona kadar.	`s.substring(5)` → `"Programlama"`
`toLowerCase()`	Küçük harfli yeni dizgi.	`"java programlama"`
`toUpperCase()`	Büyük harfli yeni dizgi.	`"JAVA PROGRAMLAMA"`

Tablo 3.4.3: Temel dizgi metotları, `s = "Java Programlama"` için.

Tanım 3.4.4

Alt dizgi (*substring*), bir dizginin ardışık karakterlerinden oluşan parçasıdır. `s.substring(a, b)`, `s`'nin `a` indisinden başlayıp `b - 1` indisinde biten parçasını döndürür. Parçanın uzunluğu `b - a`'dır.

`indexOf` bir arama metodudur. Dizgiyi baştan sona tarar ve aranan metnin ilk geçtiği yerin *başlangıç* indisini verir. Aranan metin hiç geçmiyorsa −1 döndürür. −1 geçerli bir indis olamayacağı için "bulunamadı" anlamını güvenle taşır. Bu yüzden arama sonucu kullanılmadan önce çoğu zaman `if (s.indexOf(x) != -1)` biçiminde denetlenir. Metodun iki parametreli sürümü aramaya belli bir indisten başlar. `s.indexOf("a", 4)` çağrısı 1 ve 3'teki `a` harflerini atlar ve ilk olarak indis 10'daki `a`'yı bulur. `lastIndexOf` ise dizgiyi sondan başa tarar ve son eşleşmeyi verir. `indexOf` ve `lastIndexOf` hem dizgi hem tek karakter (`'a'`) kabul eder.

`substring` metodunun en çok karıştırılan yanı bitiş indisinin *hariç* olmasıdır. Bu kuralın iki pratik yararı vardır. Birincisi, parçanın uzunluğu iki indisin farkıdır: `substring(5, 12)` 12 − 5 = 7 karakter verir. İkincisi, ardışık parçalar kolayca yan yana gelir: `s.substring(0, k) + s.substring(k)` her `k` için `s`'nin kendisini verir. Tek parametreli `substring(5)` ise `substring(5, s.length())` ile aynıdır.

Geçersiz aralık verilirse `substring` de `StringIndexOutOfBoundsException` fırlatır. Bitiş, uzunluktan büyük olamaz; başlangıç da bitişten büyük olamaz. `"Java".substring(2, 6)` çağrısının iletisi aralığı matematikteki yarı açık aralık gösterimiyle verir:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException:
    Range [2, 6) out of bounds for length 4
```

İletideki `[2, 6)` yazımı "2 dahil, 6 hariç" demektir. Bu gösterim `substring`'in kuralını da özetler. Buna karşılık `s.substring(s.length())` geçerlidir ve boş dizgi döndürür. Başlangıç ile bitişin eşit olması boş bir aralık anlamına gelir.

3.4.2.2 indexOf ve substring adım adım

Sunumdaki izleme slaytı tablodaki metotları sırayla çalıştırır ve her değişkenin aldığı değeri karakter kutuları üzerinde gösterir:

```
String str = "Java Programlama";
int indeks = str.indexOf("Programlama");
int uzunluk = str.length();
String alt = str.substring(5, 12);
String son = str.substring(5);
String kucuk = str.toLowerCase();
String buyuk = str.toUpperCase();
```

Kodda hiçbir şey yazdırılmaz; önemli olan her satırdan sonra değişkenlerin aldığı değerlerdir. Tablo 3.4.4 bu değerleri sırayla verir.

Satır	Değişken	Değer	Açıklama
2	`indeks`	5	`"Programlama"` 5. indiste başlar
3	`uzunluk`	16	indisler 0 … 15
4	`alt`	`"Program"`	indis 5 … 11; 12 − 5 = 7 karakter
5	`son`	`"Programlama"`	indis 5'ten sona kadar
6	`kucuk`	`"java programlama"`	yeni dizgi; `str` değişmedi
7	`buyuk`	`"JAVA PROGRAMLAMA"`	yeni dizgi; `str` değişmedi

Tablo 3.4.4: `str = "Java Programlama"` üzerinde her satırdan sonra oluşan değer.

Şekil 3.4.4 iki önemli adımı çizimle gösterir. Vurgulu kutular, metodun eriştiği karakterlerdir. `substring(5, 12)` adımında 12 numaralı kutu (`l`) vurgulanmaz; bitiş indisinin hariç olduğu çizimde de görülür.



Şekil 3.4.4: (a) `indexOf("Programlama")` eşleşmenin başladığı indisi, 5'i verir. (b) `substring(5, 12)` 5–11 indislerini alır: `"Program"` .

Son iki satırda `toLowerCase()` ve `toUpperCase()` çağrılır. İkisi de yeni bir dizgi döndürür ve bu dizgi başka bir değişkene atanır. `str` değişkeni program boyunca `"Java Programlama"` olarak kalır. Bir metodun "dizgiyi değiştirdiği" izlenimi yanıltıcıdır; bunun nedeni değiştirilemezlik kesiminde açıklanıyor.

Örnek 3.4.1

Bir e-posta adresini kullanıcı adı ve alan adı diye ikiye ayıralım. Adres `e = "ogrenci@giresun.edu.tr"` olsun. `@` işaretinin yeri `int at = e.indexOf('@');` ile bulunur. `"ogrenci"` yedi harflidir; harfler 0–6 indislerini kaplar ve `@` indis 7'dedir. Kullanıcı adı `@`'dan önceki kısımdır: `e.substring(0, at) = e.substring(0, 7) =` `ogrenci`. Bitiş indisi hariç olduğu için `@` parçaya girmez. Alan adı `@`'dan sonraki kısımdır: `e.substring(at + 1) = e.substring(8) =` `giresun.edu.tr`. Burada `+ 1` unutulsaydı sonuç `@giresun.edu.tr` olurdu. Yöntem adres uzunluğundan bağımsızdır; indisler elle sayılmaz, `indexOf` ile bulunur.

3.4.2.3 Diğer yararlı metotlar

Temel metotların yanında birkaç metot daha gündelik kodda sık geçer. Sunumdaki örnek bunları bir arada dener:

```
String s = "  Merhaba Dünya  ";
System.out.println("[" + s.trim() + "]");
System.out.println("[" + s.replace('a', 'e') + "]");
String[] meyveler = "elma,armut,kiraz".split(",");
System.out.println(meyveler.length);
System.out.println(meyveler[1]);
System.out.println(String.join(" - ", meyveler));
System.out.println("ab".repeat(3));       // Java 11+
System.out.println("".isEmpty());
System.out.println("   ".isBlank());      // Java 11+
char[] dizi = "Java".toCharArray();
System.out.println(dizi.length + " " + dizi[0]);
System.out.println("Giresun".replace("sun", "SUN"));
System.out.println(String.valueOf(3.5) + 1);
System.out.println(Integer.parseInt("42") + 1);
```

```
[Merhaba Dünya]
[  Merhebe Dünye  ]
3
armut
elma - armut - kiraz
abababab
true
true
4 J
GireSUN
3.51
43
```

Satırları sırayla açıklayalım:

- `trim()` dizginin başındaki ve sonundaki boşlukları atar. Aradaki boşluklara dokunmaz: `" Merhaba Dünya "` → `[Merhaba Dünya]`. Köşeli parantezler boşlukların görünmesi için eklendi. Kullanıcı girdisini işlemeden önce `trim()` çağırmak yaygın bir alışkanlıktır. Java 11 ile gelen `strip()` aynı işi Unicode'daki bütün boşluk karakterleri için yapar.
- `replace('a', 'e')` bütün `a` harflerini `e` yapar: `[ Merhebe Dünye ]`. Yalnız ilk eşleşmeyi değil, hepsini değiştirir. Boşluklar korunmuştur, çünkü `replace` yeni dizgiyi `s`'nin kendisinden üretir. Önceki satırdaki `trim()` sonucu hiçbir yere atanmamıştı.
- `split(",")` dizgiyi verilen ayraçtan böler ve parçaları bir `String[]` dizisi olarak döndürür. `"elma,armut,kiraz"` üç parçaya ayrılır; `meyveler.length` 3, `meyveler[1]` ise `armut`'tur.
- `String.join(" - ", meyveler)` bölmenin tersini yapar: parçaları aralarına ayraç koyarak birleştirir. Sonuç `elma - armut - kiraz`'dır.
- `"ab".repeat(3)` dizgiyi üç kez yan yana yazar: `ababab`. Bu metot Java 11'de eklendi.
- `isEmpty()` uzunluk 0 ise `true` döndürür. `isBlank()` (Java 11) dizgi boşsa ya da yalnız boşluk karakterlerinden oluşuyorsa `true` döndürür. `" "` boş değildir ama boşluktan ibarettir: `isEmpty()` `false`, `isBlank()` `true` verir.
- `toCharArray()` dizginin karakterlerini yeni bir `char` dizisine kopyalar. Dizinin uzunluğu 4, ilk öğesi `'J'`'dir. Dizi değiştirilebilir; bu yüzden karakterler üzerinde yer değiştirme gibi işler dizi üzerinde yapılıp sonuç `new String(dizi)` ile geri çevrilebilir.
- `replace` tek karakterle sınırlı değildir; dizgi de alır: `"Giresun".replace("sun", "SUN")` → `GireSUN`.
- Son iki satır sayı ile dizgi arasındaki dönüşümü gösterir. `String.valueOf(3.5)` bir dizgidir, `"3.5"`; ona 1 eklemek birleştirmedir ve sonuç `3.51` olur. `Integer.parseInt("42")` ise dizgiyi gerçek bir `int`'e çevirir; 1 eklenince 43 çıkar. Ondalıklı sayılar için `Double.parseDouble` kullanılır.

`parseInt` yalnız rakamlardan (ve başta isteğe bağlı bir işaretten) oluşan bir dizgiyi kabul eder. Boşluk ya da harf içeren bir metin verilirse `NumberFormatException` fırlatır:

```
Exception in thread "main" java.lang.NumberFormatException:
    For input string: " 12 "
```

Bu yüzden kullanıcıdan ya da bir dosyadan gelen sayı metni önce `trim()` ile temizlenir: `Integer.parseInt(metin.trim())`. Bölme ile birlikte kullanıldığında bu adım daha da önem kazanır. `" 3, 12 "` gibi bir satır virgülden bölününce parçaların başında ve sonunda boşluk kalır.

`split`'in iki inceliği daha vardır. Ardışık iki ayraç arasında boş bir parça oluşur. Dizginin sonundaki boş parçalar ise atılır. Ayrıca `split`'in parametresi bir **düzenli ifadedir** (*regular expression*): metin aramada kullanılan bir kalıp dilidir. `" +"` kalıbı "bir ya da daha çok boşluk" anlamına gelir:

```
System.out.println(Arrays.toString("a b  c".split(" ")));
System.out.println(Arrays.toString("a b  c".split(" +")));
System.out.println(Arrays.toString("a,,b,".split(",")));
```

```
[a, b, , c]
[a, b, c]
[a, , b]
```

İlk satırda iki ardışık boşluk boş bir parça üretir ve dizi dört öğeli olur. İkinci satır birden çok boşluğu tek ayraç sayar. Üçüncü satırda ortadaki boş parça kalır, sondaki atılır; dizi üç öğelidir. Nokta (`.`) ve dikey çizgi (`|`) düzenli ifadelerde özel anlam taşır. Bu yüzden `"a.b".split(".")` beklenen sonucu vermez; noktadan bölmek için `split("\\.")` yazılır.

3.4.3 Değiştirilemezlik

Önceki kesimlerde her metodun yeni bir değer döndürdüğü birkaç kez vurgulandı. Bu bir rastlantı değil, `String` sınıfının temel tasarım kararıdır. Bu kesim kararın ne anlama geldiğini, bellekte nasıl göründüğünü ve doğurduğu maliyeti anlatır. Türkçe metinlerde büyük-küçük harf dönüşümünün tuzakları ve çok sayıda birleştirme için tasarlanan `StringBuilder` sınıfı da bu kesimdedir.

3.4.3.1 Değiştirilemez nesne

Tanım 3.4.5

Değiştirilemez (*immutable*) nesne, oluşturulduktan sonra durumu hiç değişmeyen nesnedir. `String` nesneleri değiştirilemezdir: bir dizginin karakterleri ve uzunluğu nesne oluştuğu anda belirlenir ve sonra değişmez. `toUpperCase`, `replace`, `substring`, `concat` ve `+` orijinal nesneye dokunmaz; sonucu yeni bir nesne olarak döndürür.

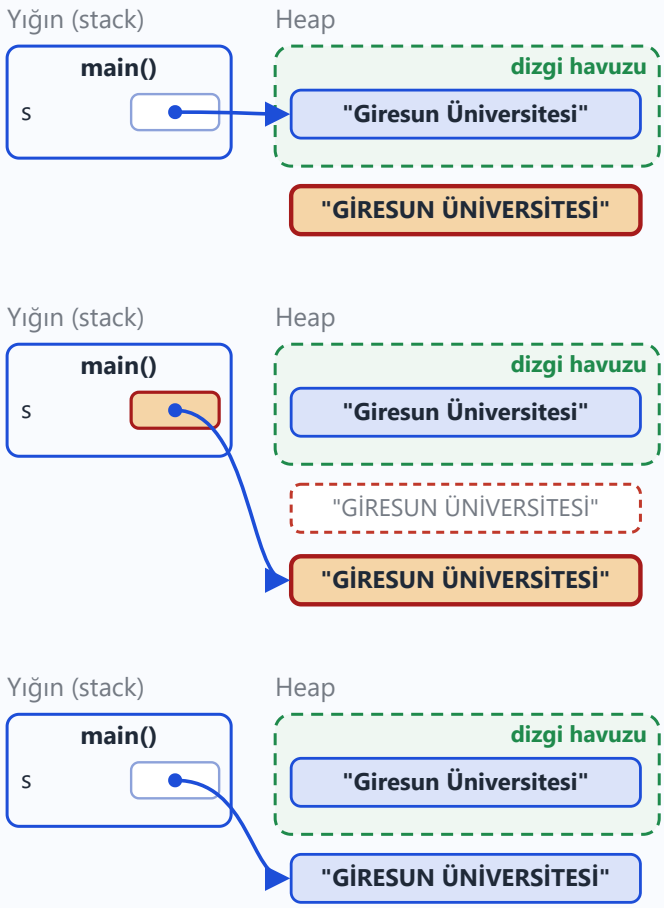
Bir benzetme yardımcı olur. Dizgi bir taş tablete benzer. Tabletteki yazı kazınıp düzeltilemez. Değişiklik gerekirse yeni bir tablet yazılır ve eskisinin yerine konur. Sunumdaki örnek bu davranışı gösterir:

```
String s = "Giresun Üniversitesi";
s.toUpperCase();      // sonuç kaybolur
System.out.println(s);
s = s.toUpperCase();  // sonuç s'ye atanır
System.out.println(s);
```

```
Giresun Üniversitesi
GİRESUN ÜNİVERSİTESİ
```

İkinci satırdaki `s.toUpperCase();` çağrısı çalışır ve büyük harfli yeni bir dizgi üretir. Ancak sonuç hiçbir değişkene atanmaz ve kaybolur. Bu yüzden ilk `println` orijinal metni yazar. Dördüncü satırda sonuç `s`'ye atanır. Burada da eski nesne değişmez; `s` değişkeni artık yeni nesneyi gösterir. Değişen nesne değil, değişkenin gösterdiği yerdir.

Şekil 3.4.5 bu iki durumu bellek çizimiyle verir. Sol bölümdeki yığın (*stack*), metodun yerel değişkenlerini tutar; `s` burada durur. Sağdaki **heap** ise nesnelerin yaşadığı bölgedir. `s` değişkeninden çıkan ok, gösterdiği nesneyi işaret eder.



Şekil 3.4.5: Değiştirilemezlik bellekte. (a) `toUpperCase()` yeni bir nesne üretir, ama sonuç atanmadığı için `s` eski nesneyi göstermeyi sürdürür. (b) İkinci çağrı yine yeni bir nesne üretir ve bu kez referansı `s`'ye atanır. (c) `s` artık büyük harfli nesneyi gösterir; eski nesne değişmeden durur.

İlk çağrının ürettiği nesneyi hiçbir değişken göstermez. Kimsenin göstermediği bu tür nesneler **çöp toplayıcıya** (*garbage collector*) kalır. Çöp toplayıcı, hiçbir referansın göstermediği nesnelerin belleğini zamanı geldiğinde geri alan JVM bileşenidir. Programcının bu belleği elle serbest bırakması gerekmez.

Çıktının ikinci satırındaki `GİRESUN ÜNİVERSİTESİ`, noktalı büyük `İ` harfleriyle yazılmıştır. Bu sonuç Türkçe yerel ayarlı bir bilgisayarda alınır. İngilizce ayarlı bir bilgisayarda aynı program `GIRESUN ÜNIVERSITESI` yazar. Nedeni bir sonraki alt kesimde açıklanıyor.

Değiştirilemezliğin bir bedeli vardır: her değişiklik yeni bir nesne demektir. Java'yı tasarlayanlar bu bedeli şu yararlar için göze almıştır:

- **Güvenli paylaşım.** Aynı dizgiyi gösteren iki değişken birbirini bozamaz. Bir metoda verilen dizginin o metot içinde değiştirilmesinden korkmak gerekmez.
- **Dizgi havuzu.** İçeriği aynı dizgi sabitleri tek bir nesneyi paylaşabilir. Paylaşım ancak nesne değişmiyorsa güvenlidir. Havuz karşılaştırma kesiminde ayrıntılı işleniyor.

- **İş parçacıkları arasında güvenlik.** Aynı anda çalışan program parçaları değişmeyen bir nesneyi ek önlem almadan ortak kullanabilir.

Sık yapılan hata

Dizgi metodunun sonucunu atamayı unutmak: `ad.trim();`, `ad.toUpperCase();` ya da `ad.replace(" ", "");` satırları `ad`'ı değiştirmez. Derleyici uyarı vermez, program çalışır ama hiçbir şey olmaz. Doğru yazım `ad = ad.trim();` biçimindedir. Aynı nedenle `s.charAt(0) = 'j';` gibi bir karakteri yerinde değiştirme girişimi derlenmez.

3.4.3.2 Türkçe harfler ve yerel ayar

Tanım 3.4.6

Yerel ayar (*locale*), bir dil ve bölgeye özgü biçim kurallarının bütünüdür: büyük-küçük harf dönüşümü, ondalık ayırıcı, binlik ayırıcı, tarih biçimi. Java'da `java.util.Locale` sınıfıyla temsil edilir. Türkiye Türkçesi için yerel ayar `tr-TR`'dir.

Türkçe alfabede `i` harfinin noktalı ve noktasız iki biçimi vardır ve ikisinin de büyük harfi ayrıdır: `i` ↔ `İ`, `ı` ↔ `I`. İngilizcede ise yalnız `i` ↔ `I` çifti bulunur. Parametresiz `toUpperCase()` ve `toLowerCase()` bilgisayarın varsayılan yerel ayarını kullanır. Aynı program bu yüzden farklı makinelerde farklı sonuç verebilir:

```
import java.util.Locale;

String s = "Giresun Üniversitesi";
System.out.println(s.toUpperCase());
System.out.println(s.toUpperCase(Locale.ROOT));
System.out.println("TITLE".toLowerCase());
System.out.println("TITLE".toLowerCase().equals("title"));
System.out.println("TITLE".toLowerCase(Locale.ROOT));
```

```
GİRESUN ÜNİVERSİTESİ
GIRESUN ÜNIVERSITESI
tıtle
false
title
```

Çıktı Türkçe ayarlı bir bilgisayarda alınmıştır. İlk satırda varsayılan ayar Türkçe olduğu için `i` harfleri `İ` olur. İkinci satırda `Locale.ROOT` verilir. Bu, dilden bağımsız, temel Unicode kurallarını kullanan yerel ayardır; `i` harfleri `I` olur. Üçüncü satır tuzağı gösterir: `"TITLE"` Türkçe kurallarla küçültülün-

ce `I` harfleri `ı` olur ve sonuç `tıtle` çıkar. Dördüncü satırdaki karşılaştırma bu yüzden `false` verir. Son satırda `Locale.ROOT` ile beklenen `title` elde edilir. Tablo 3.4.5 dört harfin iki ayardaki dönüşümünü özetler.

Dönüşüm	Türkçe (`tr`)	`ROOT` / İngilizce
`i` → büyük	`İ`	`I`
`I` → küçük	`ı`	`i`
`ı` → büyük	`I`	`I`
`İ` → küçük	`i`	`i` (`i` + birleşen nokta; iki karakter)

Tablo 3.4.5: Türkçe (`tr`) ve dilden bağımsız (`Locale.ROOT`) ayarlarda büyük-küçük harf dönüşümü.

Tablonun son satırı daha az bilinen bir durumdur. `İ` harfi Türkçe dışındaki kurallarla küçültülünce tek bir `i` değil, `i` ile üstüne konan bir "birleşen nokta" karakteri çıkar. Görünüşte fark yoktur ama dizginin uzunluğu artar. `"İSTANBUL".toLowerCase(Locale.ROOT)` 9 karakterlik bir dizgi verir; `"İki".toLowerCase(Locale.ROOT).equals("iki")` ise `false`'tur.

Pratik kural şöyledir. Program içi metinlerde, yani komut adı, dosya uzantısı, protokol anahtarı gibi makinenin okuyacağı metinlerde `Locale.ROOT` kullanılır. Kullanıcıya gösterilecek Türkçe metinlerde ise Türkçe ayar açıkça verilir: `metin.toUpperCase(Locale.forLanguageTag("tr-TR"))`. Böylece program hangi bilgisayarda çalışırsa çalışsın aynı sonucu üretir. `Locale` sınıfı `java.util` paketindedir ve `import java.util.Locale;` satırını gerektirir.

Sık yapılan hata

Kullanıcı girdisini `girdi.toLowerCase().equals("exit")` ile denetlemek. Türkçe ayarlı bir bilgisayarda kullanıcı `EXIT` yazarsa sonuç `"exıt"` olur ve koşul tutmaz. Hata yalnız bazı makinelerde ortaya çıktığı için bulunması zordur. Büyük-küçük harf farkını yok saymak için `equalsIgnoreCase` ya da `toLowerCase(Locale.ROOT)` kullanılır.

3.4.3.3 Döngüde birleştirmenin maliyeti

Değiştirilemezliğin bedeli en çok döngüde birleştirme yapılırken hissedilir. Sunumdaki kodun ilk bölümü 1'den 5'e kadar sayıları bir dizgide toplar:

```
String sonuc = "";
for (int i = 1; i <= 5; i++) {
  sonuc += i + " ";   // her turda yeni String
}
System.out.println(sonuc);

StringBuilder sb = new StringBuilder();
for (int i = 1; i <= 5; i++) {
  sb.append(i).append(' ');   // aynı nesne büyür
}
String metin = sb.toString();
System.out.println(metin);

StringBuilder ters = new StringBuilder("Giresun");
ters.reverse();
System.out.println(ters);
StringBuilder k = new StringBuilder("Java");
k.insert(0, ">> ").append('!');
k.setCharAt(3, 'j');
System.out.println(k + " " + k.length());
```

```
1 2 3 4 5
1 2 3 4 5
nuseriG
>> java! 8
```

İlk döngüde `sonuc += i + " ";` satırı her turda yeni bir `String` nesnesi üretir. Eski içerik ve yeni parça bu nesneye kopyalanır, eski nesne de çöpe kalır. Bir turda kopyalanan karakter sayısı, o ana kadar biriken metnin uzunluğu kadardır. Her tur bir karakter ekliyorsa kopyalanan toplam karakter sayısı $1 + 2 + \ldots + n = n(n+1)/2$ olur, yani yaklaşık $n^2/2$. Örneğin bin turluk bir döngüde 500.500 karakter kopyalanır. Bu büyüme $O(n^2)$ ile gösterilir: tur sayısı iki katına çıkınca iş dört katına çıkar.

Beş turluk bu örnekte fark hissedilmez. Her tur iki karakter (rakam ve boşluk) ekler; toplam 2 + 4 + 6 + 8 + 10 = 30 karakter kopyalanır. Ama on binlerce satırlık bir rapor ya da büyük bir dosyanın içeriği bu yolla kurulursa program belirgin biçimde yavaşlar.

3.4.3.4 StringBuilder

Tanım 3.4.7

`StringBuilder`, **değiştirilebilir** (*mutable*) bir karakter **tamponudur** (*buffer*). İçinde karakterleri tutan bir dizi ve bu dizinin kaç hücresinin dolu olduğunu gösteren bir uzunluk bulunur. `append`, `insert`, `reverse`, `setCharAt` ve `deleteCharAt` gibi metotlar yeni nesne üretmez; aynı nesnenin içeriğini değiştirir. İş bitince `toString()` ile bir `String` elde edilir.

Kodun ikinci bölümü aynı metni `StringBuilder` ile kurar. `sb.append(i).append(' ')` her turda tamponun sonuna bir sayı ve bir boşluk ekler. Tampon tek bir nesnedir ve gerektikçe büyür. Döngü bitince `sb.toString()` içerikten bir `String` üretir. İki yöntemin çıktısı aynıdır: `1 2 3 4 5` (sonda bir boşlukla).

`append` metodu tamponun kendisini döndürür. Bu yüzden çağrılar zincir gibi art arda yazılabilir: `sb.append(i).append(' ')`. İlk `append` `sb`'yi döndürür, ikinci `append` yine `sb` üzerinde çalışır. `append`'in her ilkel tip ve `String` için ayrı sürümü vardır; sayı, karakter, mantıksal değer hepsi metne çevrilerek eklenir.

Kodun son bölümü diğer metotları dener. `new StringBuilder("Giresun")` içeriği `"Giresun"` olan bir tampon oluşturur. `reverse()` tamponu yerinde ters çevirir ve `nuseriG` yazılır. `println`'e bir `StringBuilder` verildiğinde `toString()` kendiliğinden çağrılır. Son örnekte `"Java"` tamponunun başına `insert(0, ">> ")` ile üç karakter eklenir, sonuna `'!'` konur ve indis 3'teki `J` `setCharAt` ile `j` yapılır. Sonuç `>> java!`, uzunluk 8'dir. `String` ile bu işlem mümkün değildir; karakteri değiştirmek için her seferinde yeni bir dizgi kurmak gerekir.

Sunumdaki izleme slaytı tamponun içini hücre hücre gösterir:

```
StringBuilder sb = new StringBuilder("Java");
sb.append(' ').append(8);
sb.insert(0, ">> ");
sb.setCharAt(3, 'j');
sb.deleteCharAt(sb.length() - 1);
sb.append("programlama dili");
String s = sb.toString();
System.out.println(s + " " + sb.length() + " " + sb.capacity());
```

```
>> java programlama dili 24 42
```

Bir `StringBuilder`'ın iki ölçüsü vardır. **Uzunluk** (`length()`) dolu hücre sayısıdır. **Kapasite** (`capacity()`) ise içerideki dizinin toplam hücre sayısıdır. Bir dizgiden oluşturulan tamponun kapasitesi, dizginin uzunluğuna 16 eklenerek bulunur. `"Java"` için bu 4 + 16 = 20'dir. Boş bir `new StringBuilder()`'ın kapasitesi ise 16'dır. Tablo 3.4.6 her satırdan sonraki durumu verir; değerler `length()` ve `capacity()` yazdırılarak doğrulanmıştır.

Satır	İçerik	Uzunluk	Kapasite	Açıklama
`new StringBuilder("Java")`	`[Java]`	4	20	16 + 4
`append(' ').append(8)`	`[Java 8]`	6	20	sona eklenir, nesne aynı
`insert(0, ">> ")`	`[>> Java 8]`	9	20	eski karakterler 3 hücre sağa kayar
`setCharAt(3, 'j')`	`[>> java 8]`	9	20	indis 3 yerinde değişir

Satır	İçerik	Uzunluk	Kapasite	Açıklama
`deleteCharAt(length() - 1)`	`[>> java ]`	8	20	sondaki `8` silinir
`append("programlama dili")`	`[>> java programlama dili]`	24	42	24 > 20: tampon büyür
`toString()`	—	24	42	içerikten yeni bir `String`

Tablo 3.4.6: `StringBuilder` izlemesi. Köşeli parantez içindeki metin tamponun dolu kısmıdır.

`insert` ve `deleteCharAt` ucuz işlemler değildir. Başa bir şey eklemek için sonraki bütün karakterlerin sağa kaydırılması gerekir; ortadan silmek de sola kaydırma ister. Bu kaydırmalar tampon uzunluğuyla orantılı iş yapar. `append` ise yer olduğu sürece yalnız sona yazar ve sabit zamanda biter.

Son `append` 16 karakter ekler ve uzunluk 24'e çıkar. 24 karakter 20 hücreye sığmaz. Bu durumda `StringBuilder` daha büyük yeni bir dizi ayırır, eski içeriği oraya kopyalar ve eski diziyi bırakır. JDK'daki kural yeni kapasiteyi eskisinin iki katının iki fazlası olarak seçer: 2 × 20 + 2 = 42. Sunumdaki *(20 + 1) × 2* yazımı da aynı sayıyı verir. Gereken uzunluk bundan da büyükse doğrudan o uzunluk alınır. Şekil 3.4.6 büyüme anını gösterir.



Şekil 3.4.6: `StringBuilder` tamponu. (a) `insert(0, ">> ")` sonrası: eski karakterler üç hücre sağa kaymıştır. (b) 24 karakter 20 hücreye sığmaz; kapasite 42'ye çıkar ve içerik yeni diziye kopyalanır.

Büyüme sırasında kopyalama yapıldığı hâlde `StringBuilder` neden hızlıdır? Kapasite her büyümede yaklaşık iki katına çıkar. Boş bir tampona karakterler tek tek eklendiğinde kapasite 16, 34, 70, 142, 286, 574, 1150 diye ilerler. Bin karakter eklemek için yalnız altı büyüme gerekir. Bu büyümelerde kopyalanan toplam karakter sayısı, eklenen karakter sayısının birkaç katını geçmez. Bu yüzden n karakter eklemenin toplam maliyeti $O(n)$'dir; `+=` ile yapılan birleştirmenin $O(n^2)$ maliyetiyle karşılaştırıldığında fark çok büyüktür.

Bu, `+` işlecinin hiç kullanılmaması gerektiği anlamına gelmez. Birkaç parçalık birleştirmelerde `+` okunaklıdır ve yeterince hızlıdır; Java derleyicisi tek bir ifadedeki birleştirmeleri zaten verimli biçimde çevirir. `StringBuilder`, bir döngü içinde çok sayıda parça birleştirilirken tercih edilir. Yaklaşık uzunluk baştan biliniyorsa `new StringBuilder(1000)` gibi bir başlangıç kapasitesi vermek büyüme sayısını azaltır.

3.4.4 Kullanıcı girişi: Scanner

Bir program çoğu zaman kullanıcıdan bilgi ister: bir ad, bir yaş, bir cümle. Klavyeden gelen her şey önce karakter olarak gelir. Bu karakterlerin kelimeye, satıra ya da sayıya dönüştürülmesi `Scanner` sınıfının işidir. Okuma metotları birbirine benzese de davranışları ince noktalarda ayrılır. Bu ayrımlar bilinmezse programlar kullanıcının sorusunu atlamak gibi tuhaf hatalar verir.

3.4.4.1 Scanner ve okuma metotları

> **Tanım 3.4.8**
>
> `java.util.Scanner`, bir kaynaktaki metni parçalara ayırarak okuyan sınıftır. `new Scanner(System.in)` klavyeden okuyan bir nesne oluşturur. Klavyede yazılan satır `Enter`'a basılınca bir **girdi tamponuna** girer; satır sonu karakteri (⏎) de tampona eklenir. `Scanner` tamponda bir **imleç** tutar ve her okuma imleci okunan kısmın sonuna taşır.

Dört okuma metodu sık kullanılır:

- `next()` önce baştaki boşlukları (boşluk, sekme, satır sonu) atlar, sonra bir sonraki boşluğa kadar okur. Sonuç tek bir kelimedir.
- `nextLine()` imlecin bulunduğu yerden satır sonuna kadar her şeyi okur. Boşlukları atlamaz. Satır sonu karakterini tüketir ama sonuca katmaz.
- `nextInt()` ve `nextDouble()` baştaki boşlukları atlar ve bir sayı okur. Sayının arkasındaki satır sonunu tamponda **bırakır**. Sayı yerine harf girilirse `InputMismatchException` fırlatır.

Klavyeden okuyan bir program dosyanın başında `import java.util.Scanner;` satırını içerir. Okumalar bitince `klavye.close()` çağrılır. Sunumdaki örnek bir satırı önce iki kelime, sonra satırın geri kalanı olarak okur:

```
Scanner klavye = new Scanner(System.in);
System.out.println("bir metin giriniz");
String girdi = klavye.next();
System.out.println(girdi);    // bilgisayar
girdi = klavye.next();
System.out.println(girdi);    // ve
girdi = klavye.nextLine();
System.out.println(girdi);    //  programlamaya giriş
```

Kullanıcı şu satırı yazar ve `Enter`'a basar:

```
bilgisayar ve programlamaya giriş
```

Program şunu yazar:

```
bir metin giriniz
bilgisayar
ve
 programlamaya giriş
```

Çıktının son satırındaki baştaki boşluk dikkat çeker. “ve” kelimesinden sonra gelen boşluk satırın bir parçasıdır ve `nextLine()` boşlukları atlamaz. Bu yüzden okunan metin `" programlamaya giriş"` olur.

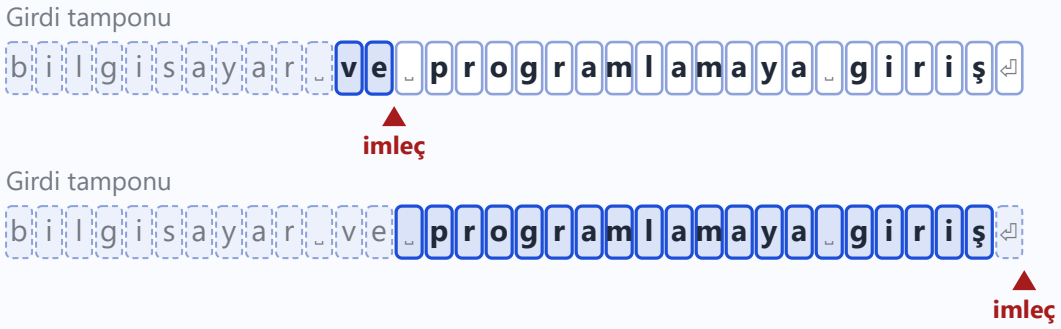
3.4.4.2 next() ve nextLine() adım adım

Tablo 3.4.7 her okumada imlecin nereden nereye ilerlediğini verir. Tampondaki karakterlerin indisleri 0'dan başlar: `"bilgisayar"` 0–9, ilk boşluk 10, `"ve"` 11–12, ikinci boşluk 13, kalan metin 14–32 indislerini kaplar ve satır sonu 33'tedir.

Çağrı	İmleç önce	Okunan	İmleç sonra	Açıklama
`next()`	0	`"bilgisayar"`	10	ilk boşluğa kadar okur; boşluk tamponda kalır
`next()`	10	`"ve"`	13	önce 10'daki boşluğu atlar
`nextLine()`	13	`" programlamaya giriş"`	34	boşluğu atlamaz; ⏎'yi tüketir, sonuca katmaz

Tablo 3.4.7: `bilgisayar ve programlamaya giriş⏎` satırı okunurken imlecin konumu.

Şekil 3.4.7 ikinci ve üçüncü okumanın sonunda tamponun durumunu gösterir. Okunmuş karakterler soluk, imlecin yeri bir üçgenle işaretlidir.



Şekil 3.4.7: Girdi tamponu. (a) İkinci `next()` boşluğu atlayıp `"ve"` 'yi okur; imleç yine bir boşluğun önündedir. (b) `nextLine()` boşluğu atlamadan satır sonuna kadar okur ve ⏎'yi tüketir.

Programcı açısından sonuç şudur: tek kelimelik bir yanıt (ad, şehir, komut) için `next()`, boşluk içerebilen bir yanıt (ad soyad, adres, cümle) için `nextLine()` kullanılır. `next()` ile ad soyad okunursa yalnız ad alınır, soyad tamponda kalır ve bir sonraki okumaya karışır.

3.4.4.3 nextInt() sonrası nextLine() tuzağı

Sayı okuyan metotlarla `nextLine()` art arda kullanıldığında yeni başlayanların neredeyse hepsinin karşılaştığı bir hata ortaya çıkar. Sunumdaki program önce yaşı, sonra adı sorar:

```
Scanner klavye = new Scanner(System.in);
System.out.print("Yaş: ");
int yas = klavye.nextInt();
System.out.print("Ad: ");
String ad = klavye.nextLine();
System.out.println("[" + ad + "] " + yas);
```

Kullanıcı `19` yazıp `Enter`'a basar. Program adı sormadan biter:

```
Yaş: 19
Ad: [] 19
```

Neden böyle olduğu tampona bakınca anlaşılır (Şekil 3.4.8). Tamponda `19⏎` vardır. `nextInt()` yalnız `19`'u okur ve imleci ⏎'nin önünde bırakır. Ardından gelen `nextLine()` görevini yerine getirir: imleçten satır sonuna kadar okur. İmleç ile satır sonu arasında hiçbir karakter olmadığı için sonuç boş dizgidir. `nextLine()` ⏎'yi tüketir ve hemen döner. Kullanıcı daha adını yazmaya fırsat bulamadan `ad` değişkeni `""` olmuştur.



Şekil 3.4.8: `nextInt()` 19'u okuduktan sonra imleç satır sonu karakterinin önünde bekler.

Çözüm, sayıyı okuduktan sonra artakalan satır sonunu ayrı bir `nextLine()` çağrısıyla tüketmektir:

```
int yas = klavye.nextInt();
klavye.nextLine();          // ⏎'yi at
System.out.print("Ad: ");
String ad = klavye.nextLine();
```

```
Yaş: 19
Ad: Ayşe
[Ayşe] 19
```

Araya eklenen `klavye.nextLine();` çağrısının sonucu kullanılmaz; tek görevi ⏎'yi tamponda temizlemektir. Bundan sonraki `nextLine()` kullanıcının yeni satırını bekler. İkinci bir çözüm, bütün girdiyi `nextLine()` ile okuyup sayıları kendisi çevirmektir: `int yas = Integer.parseInt(klavye.nextLine().trim());`. Bu yöntemde tamponda hiçbir zaman artık satır sonu kalmaz.

Örnek 3.4.2

Kullanıcıdan önce yaş, sonra ad-soyad isteyen ve sonucu `printf` ile yazan program, tuzağa düşmeden şöyle yazılır:

```
Scanner k = new Scanner(System.in);
System.out.print("Yaş: ");
int yas = k.nextInt();
k.nextLine();   // satır sonunu at
System.out.print("Ad: ");
String ad = k.nextLine();
System.out.printf("Ad: %s, Yaş: %d%n", ad, yas);
```

Kullanıcı `20` ve `Ali Veli` girdiğinde konsol şöyle görünür:

```
Yaş: 20
Ad: Ali Veli
Ad: Ali Veli, Yaş: 20
```

`nextInt()` 20'yi okur, ⏎ tamponda kalır. `k.nextLine();` bu ⏎'yi tüketir. Son `nextLine()` kullanıcının yeni satırını bekler ve `"Ali Veli"`'yi boşluğuyla birlikte okur. Ad soyad için `next()` kullanılsaydı yalnız `"Ali"` okunurdu. `printf` satırı biçimli çıktı kesiminde açıklanıyor.

3.4.5 Dizgileri karşılaştırmak

İki dizginin aynı olup olmadığını sormak basit görünür. Ama Java'da bu sorunun iki ayrı anlamı vardır: "iki değişken aynı nesneyi mi gösteriyor?" ve "iki nesnenin içeriği aynı mı?". İlk soru `==` ile, ikincisi `equals()` ile sorulur. Bu kesim iki sorunun farkını, farkı bazen gizleyen dizgi havuzunu ve sıralama için kullanılan `compareTo` metodunu anlatır.

3.4.5.1 == ile karşılaştırma

`int` gibi ilkel tiplerde `==` iki değeri karşılaştırır. `String` değişkenleri ise değer değil referans tutar. Bu yüzden `==` iki dizginin **referanslarını**, yani bellekteki adreslerini karşılaştırır, içeriklerini değil. İki değişken aynı nesneyi gösteriyorsa sonuç `true`'dur. İçerik aynı olsa bile nesneler farklıysa sonuç `false`'tur. Sunumdaki örnek:

```
String str1 = "Giresun";
String str2 = "Giresun";
String str3 = new String("Giresun");
System.out.println(str1 == str2);        // true
System.out.println(str1 == "Giresun");   // true
System.out.println(str1 == str3);        // false
System.out.println(str3 == "Giresun");   // false
```

```
true
true
false
false
```

Dört karşılaştırmada içerik hep `"Giresun"`'dur, ama sonuçların ikisi `false`'tur. `str1 == str2`'nin `true` çıkması bir şans değildir; dizgi havuzunun sonucudur. `str3` ise `new` ile oluşturulmuş ayrı bir nesnedir.

3.4.5.2 Dizgi havuzu

Tanım 3.4.9

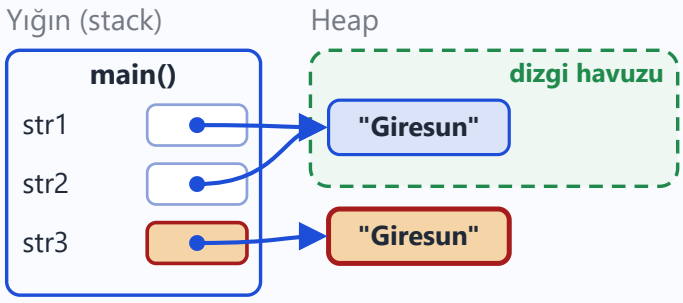
Dizgi havuzu (*string pool*), JVM'in dizgi sabitlerini tuttuğu özel bölgedir. Kaynak kodda içeriği aynı olan bütün dizgi sabitleri havuzdaki tek bir nesneyi paylaşır. `new String(...)` ise havuza bakmaz; her çağrısında heap'te yeni bir nesne oluşturur.

Havuzun amacı bellek tasarrufudur. Bir programda `"Giresun"` sabiti yüz farklı yerde geçebilir. Her geçişte yeni bir nesne oluşturmak yerine hepsi aynı nesneyi gösterir. Bu paylaşım ancak dizgiler değiştirilemez olduğu için güvenlidir. Bir yerdeki kod `"Giresun"` nesnesini değiştirebilseydi öteki doksan dokuz yer de etkilenirdi.

Tablo 3.4.8 sunumdaki havuz izlemesini satır satır verir. Nesneler #1 ve #2 diye numaralandırılmıştır.

Satır	`str1`	`str2`	`str3`	Açıklama
`String str1 = "Giresun";`	#1			sabit havuzdaki #1'i verir
`String str2 = "Giresun";`	#1	#1		aynı sabit: yeni nesne oluşmaz
`String str3 = new String("Giresun");`	#1	#1	#2	`new` havuz dışında yeni nesne üretir
`str1 == str2`	#1 ile #1			`true`
`str1 == "Giresun"`	#1 ile #1			`true`
`str1 == str3`	#1 ile #2			`false`
`str3 == "Giresun"`	#2 ile #1			`false`

Tablo 3.4.8: Dizgi havuzu izlemesi. #1 havuzdaki nesne, #2 havuz dışındaki nesnedir.



Şekil 3.4.9: Üç bildirimden sonra bellek. `str1` ve `str2` havuzdaki aynı nesneyi, `str3` havuz dışındaki ayrı bir nesneyi gösterir.

Havuza giren dizgiler yalnız kaynak koddaki sabitlerle sınırlı değildir. Derleme zamanında hesaplanabilen ifadeler de sabit sayılır. `"Gire" + "sun"` ifadesinin iki yanı da sabittir; derleyici onu doğrudan `"Giresun"` olarak yazar ve sonuç havuzdaki nesneyi gösterir. Bir yanında değişken bulunan birleştir-

me ise çalışma zamanında yapılır ve yeni bir nesne üretir. `final` ile tanımlanıp sabitle başlatılan bir değişken ise derleyici için bir sabittir:

```
String str1 = "Giresun";
String a = "Gire";
final String fa = "Gire";
System.out.println((a + "sun") == str1);   // false
System.out.println((fa + "sun") == str1);  // true
```

`intern()` metodu bir dizginin havuzdaki eşini döndürür. Havuzda aynı içerikli bir nesne varsa o nesnenin referansı, yoksa dizgi havuza eklenir ve kendisi döner. `str3.intern() == str1` bu yüzden `true`'dur. `intern()` gündelik kodda nadiren gerekir; varlığı havuzun nasıl çalıştığını anlamaya yardım eder.

Havuz ve değiştirilemezlik adım adım izlenmek istenirse ders sitesindeki "Dizgi Havuzu" simülatörü kullanılabilir. Simülatör, sabitlerin havuza girişini, `new String` ile oluşan ayrı nesneleri ve `StringBuilder` tamponunun yerinde değişmesini görsel olarak gösterir.

3.4.5.3 equals() ile içerik karşılaştırma

Tanım 3.4.10

`a.equals(b)`, `a` ve `b` dizgilerinin içeriğini karakter karakter karşılaştırır. Uzunluklar ve her konumdaki karakter aynıysa `true` döndürür. Nesnelerin aynı olup olmadığına bakmaz. `b` `null` ise `false` döndürür.

Sunumdaki örnek aynı üç değişkeni bu kez `equals` ile karşılaştırır ve birkaç durum ekler:

```
String str1 = "Giresun";
String str2 = "Giresun";
String str3 = new String("Giresun");
System.out.println(str1.equals(str2));
System.out.println(str1.equals("Giresun"));
System.out.println(str1.equals(str3));
System.out.println(str3.equals("Giresun"));
System.out.println(str3.intern() == str1);
String a = "Gire";
String b = a + "sun";  // yeni nesne
System.out.println(b == str1);
System.out.println(b.equals(str1));
```

```
true
true
true
true
true
false
true
```

İlk dört satır, nesne hangi yolla oluşmuş olursa olsun `true` verir; içerik hep `"Giresun"`'dur. Beşinci satır `intern()`'ün havuzdaki eşini döndürdüğünü gösterir. Son iki satırda `b`, `a` değişkeniyle `"sun"` sabitinin çalışma zamanında birleştirilmesiyle oluşur. Ortaya çıkan nesne yenidir: `b == str1` `false`, `b.equals(str1)` `true`'dur.

Pratikte en tehlikeli durum kullanıcı girdisidir. `Scanner`'dan okunan bir dizgi çalışma zamanında oluşur ve her zaman yeni bir nesnedir. Bu yüzden `girdi == "evet"` koşulu, kullanıcı tam olarak `evet` yazsa bile neredeyse her zaman `false` verir. Hata sinsidir: dizgi sabitleriyle yapılan denemelerde havuz yüzünden `==` doğru çalışıyor gibi görünür.

`equals` çağrısının bir de `null` yönü vardır. `null`, hiçbir nesneyi göstermeyen referans değeridir. `null` bir referans üzerinden metot çağrılamaz:

```
String cevap = null;
System.out.println("evet".equals(cevap)); // false
System.out.println(cevap + "!");          // null!
System.out.println(cevap.equals("evet")); // istisna
```

İlk satırda metot `"evet"` sabiti üzerinden çağrılır; sabit hiçbir zaman `null` olmadığı için çağrı güvenlidir ve `null` argümanla `false` döner. İkinci satırda birleştirme `null`'ı `"null"` metnine çevirir; istisna oluşmaz ve `null!` yazılır. Üçüncü satır ise `NullPointerException` fırlatır. Java 15'ten beri bu istisnanın iletisi varsayılan olarak hangi çağrının neden başarısız olduğunu açıkça söyler:

```
Exception in thread "main" java.lang.NullPointerException:
    Cannot invoke "String.equals(Object)" because "cevap" is null
```

İletideki değişken adı (`"cevap"`), sınıf dosyası hata ayıklama bilgisiyle derlendiğinde görünür. Geliştirme ortamları bu bilgiyi varsayılan olarak ekler. Komut satırında yalnız `javac` ile derlenen bir programda ise ad yerine yerel değişkenin sıra numarası yazılır:

```
Exception in thread "main" java.lang.NullPointerException:
    Cannot invoke "String.equals(Object)" because "<local1>" is null
```

`javac -g` ile derlenince ileti yine `"cevap"` adını gösterir. İki ileti de aynı hatayı anlatır: `null` olan bir değişken üzerinden `equals` çağrılmıştır. Güvenli yazım, sabiti öne koymaktır: `"evet".equals(cevap)`.

3.4.5.4 Karşılaştırma metotları

`equals` dışında dizgiler için birkaç karşılaştırma metodu daha vardır. Tablo 3.4.9 bunları döndürdükleri tiple birlikte listeler.

Metot	Döndürdüğü	Açıklama
`equals(str)`	`boolean`	İçerik birebir aynı mı?
`equalsIgnoreCase(str)`	`boolean`	Büyük-küçük harf farkı yok sayılınca aynı mı?
`startsWith(str)`	`boolean`	Dizgi verilen metinle başlıyor mu?
`endsWith(str)`	`boolean`	Dizgi verilen metinle bitiyor mu?
`contains(str)`	`boolean`	Verilen metin dizginin içinde geçiyor mu?
`isEmpty()` / `isBlank()`	`boolean`	Uzunluk 0 mı? / Yalnız boşluktan mı oluşuyor? (`isBlank`: Java 11+)
`compareTo(str)`	`int`	Sözlük sırası: negatif (önce), 0 (eşit), pozitif (sonra).
`compareToIgnoreCase(str)`	`int`	Harf büyüklüğünü yok sayan `compareTo`.

Tablo 3.4.9: Dizgi karşılaştırma metotları.

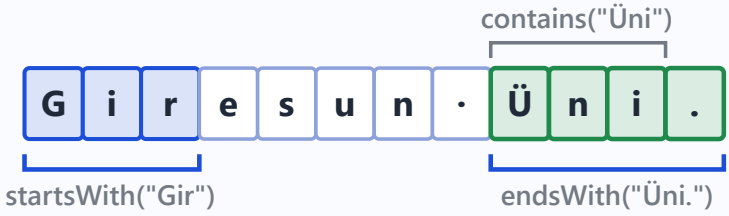
Adında `IgnoreCase` geçmeyen metotlar büyük-küçük harfe duyarlıdır. `equals` ve `equalsIgnoreCase`, `null` argüman alınca istisna fırlatmaz, `false` döndürür. Sunumdaki örnek bu metotları `"Giresun Üni."` dizgisi üzerinde dener; yorumlardaki sonuçlar programın gerçek çıktısıdır:

```
String s = "Giresun Üni.";
System.out.println(s.startsWith("Gir"));                // true
System.out.println(s.startsWith("Gır"));                // false
System.out.println(s.endsWith("Üni."));                 // true
System.out.println(s.endsWith("Üni"));                  // false
System.out.println(s.equalsIgnoreCase("Giresun Üni.")); // true
System.out.println(s.equalsIgnoreCase("Gıresun üni.")); // true
System.out.println(s.equalsIgnoreCase("GIresun Uni.")); // false
System.out.println(s.contains("Üni"));                  // true
```

Sonuçların bazıları ilk bakışta şaşırtıcıdır:

- `startsWith("Gır")` `false`'tur. `ı` ile `i` ayrı karakterlerdir; Unicode kodları 305 ve 105'tir.
- `endsWith("Üni")` `false`'tur, çünkü dizgi noktayla biter. Son karakter `'.'`'dır.
- `equalsIgnoreCase("Gıresun üni.")` `true`'dur. Metot karakterleri karşılaştırırken her iki yanı da büyük harfe çevirir. `ı` ile `i`'nin büyük biçimleri aynıdır, `I`; bu yüzden iki dizgi eşit sayılır. `equalsIgnoreCase` yerel ayara bakmaz; her makinede aynı sonucu verir.

- `equalsIgnoreCase("GIresun Uni.")` `false`'tur. `U` ile `Ü` ayrı harflerdir ve hiçbir büyük-küçük dönüşümü onları eşitlemez.



Şekil 3.4.10: `startsWith`, `contains` ve `endsWith` metotlarının eşleştiği bölgeler.

Boş dizgiyle yapılan karşılaştırmalar da bilinmeye değer. Her dizgi boş dizgiyle başlar ve onu içerir: `s.startsWith("")` ve `s.contains("")` her zaman `true`'dur. `s.indexOf("")` ise 0 döndürür.

3.4.5.5 compareTo ve sözlük sırası

Tanım 3.4.11

`a.compareTo(b)`, iki dizgiyi **sözlük sırasına** (*lexicographic order*) göre karşılaştırır. `a` önce geliyorsa negatif, eşitse 0, sonra geliyorsa pozitif bir `int` döndürür. Karşılaştırma baştan başlar ve ilk farklı karakterde biter; sonuç bu iki karakterin Unicode kodlarının farkıdır. Farklı karakter yoksa ve biri ötekinin önekiyse sonuç uzunlukların farkıdır.

Sunumdaki örnek kuralın bütün durumlarını içerir:

```
System.out.println("elma".compareTo("armut"));
System.out.println("armut".compareTo("elma"));
System.out.println("elma".compareTo("elma"));
System.out.println("Elma".compareTo("elma"));
System.out.println("Elma".compareToIgnoreCase("elma"));
System.out.println("ev".compareTo("evet"));
System.out.println("çay".compareTo("dut"));
```

```
4
-4
0
-32
0
-2
131
```

Satırları kurala göre okuyalım:

- `"elma".compareTo("armut")`: ilk karakterler farklıdır. `'e'` − `'a'` = 101 − 97 = **4**. Pozitif: "elma" "armut"tan sonra gelir.
- `"armut".compareTo("elma")`: aynı karakterler ters sırada, **−4**. Yer değiştirince işaret değişir.
- Eşit dizgiler **0** verir.
- `"Elma".compareTo("elma")`: `'E'` − `'e'` = 69 − 101 = **−32**. Unicode'da büyük harfler küçük harflerden önce gelir.
- `compareToIgnoreCase` harf büyüklüğünü yok sayar ve **0** verir.
- `"ev".compareTo("evet")`: ilk iki karakter aynıdır ve `"ev"` biter. Biri ötekinin önekidir; sonuç uzunluk farkıdır: 2 − 4 = **−2**.
- `"çay".compareTo("dut")`: `'ç'`'nin kodu 231, `'d'`'nin kodu 100'dür. 231 − 100 = **131**. Pozitif bir sonuç, "çay"ın "dut"tan sonra geldiği anlamına gelir.

Son satır önemli bir sınırı gösterir: `compareTo` Türkçe alfabe sırasını bilmez. Türkçede `ç`, `c` ile `d` arasındadır. Ama Unicode'da Türkçeye özgü harflerin kodları Latin harflerinden büyüktür: `ç` 231, `ı` 305, `ş` 351, `ü` 252, `z` ise 122. Unicode sırasına göre sıralanan bir Türkçe kelime listesinde `ç`, `ı`, `ş` ile başlayan kelimeler `z`'den sonraya düşer. Doğru Türkçe sıralama için `java.text.Collator` kullanılır:

```
import java.text.Collator;
import java.util.Arrays;
import java.util.Locale;

String[] m = {"zeytin", "çilek", "Elma", "armut", "ıhlamur", "incir"};
String[] m2 = m.clone();
Arrays.sort(m);                       // Unicode sırası
System.out.println(Arrays.toString(m));
Collator tr = Collator.getInstance(Locale.forLanguageTag("tr-TR"));
Arrays.sort(m2, tr);                  // Türkçe alfabe sırası
System.out.println(Arrays.toString(m2));
```

```
[Elma, armut, incir, zeytin, çilek, ıhlamur]
[armut, çilek, Elma, ıhlamur, incir, zeytin]
```

`Arrays.sort` dizgileri `compareTo` ile sıralar. Sonuçta büyük harfle başlayan "Elma" en başa, "çilek" ve "ıhlamur" en sona düşer. Türkçe `Collator` ise alfabe sırasını uygular ve büyük-küçük harf farkını sıralamada ikinci plana atar. Kullanıcıya gösterilecek listelerde ikinci yol seçilir.

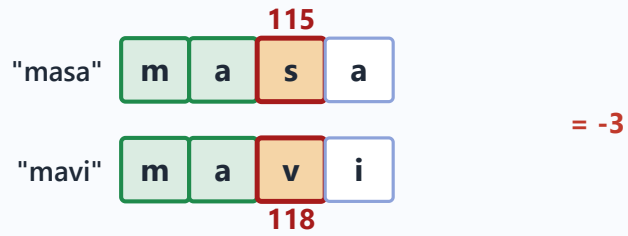
Sunumdaki izleme slaytı `compareTo`'nun adımlarını üç dizgi çifti üzerinde gösterir:

```
String a = "masa", b = "mavi";
int r1 = a.compareTo(b);
int r2 = "kar".compareTo("karne");
int r3 = "Zeki".compareTo("ali");
System.out.println(r1 + " " + r2 + " " + r3);
```

```
-3 -2 -7
```

Çift	`k`	Karakterler (kodlar)	Durum
`"masa"`, `"mavi"`	0	`'m'` (109), `'m'` (109)	eşit, devam
	1	`'a'` (97), `'a'` (97)	eşit, devam
	2	`'s'` (115), `'v'` (118)	farklı: 115 − 118 = **−3**
`"kar"`, `"karne"`	0–2	hepsi eşit; `"kar"` biter	önek: 3 − 5 = **−2**
`"Zeki"`, `"ali"`	0	`'Z'` (90), `'a'` (97)	farklı: 90 − 97 = **−7**

Tablo 3.4.10: `compareTo` izlemesi. `k` karşılaştırılan indistir.



Şekil 3.4.11: `"masa"` ile `"mavi"` indis 2'de ayrılır; sonuç kodların farkıdır, 115 − 118 = −3.

Üçüncü çift, büyük harf kuralının ne kadar ileri gittiğini gösterir: `Z` bile `a`'dan önce gelir. Sonucun büyüklüğü genellikle bir anlam taşımaz; önemli olan işaretidir. Bu yüzden karşılaştırma `if (a.compareTo(b) < 0)` biçiminde yazılır, `== -1` biçiminde yazılmaz. `<` ve `>` işleçleri dizgilerle çalışmaz; derleyici “bad operand types” hatası verir:

```
Kiyas.java:4: error: bad operand types for binary operator '<'
        if (a < b) System.out.println(a);
              ^
  first type:  String
  second type: String
```

3.4.6 Biçimli çıktı

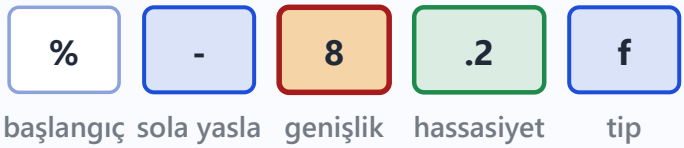
`println` değeri olduğu gibi yazar. Bir not çizelgesi, bir fatura ya da bir ölçüm tablosu yazdırılırken bu yetmez. Sütunların hizalı olması, ondalıklı sayıların belli sayıda basamakla yazılması gerekir. Java bu iş için `printf` metodunu ve onunla aynı kuralları kullanan `String.format`'ı sunar.

3.4.6.1 printf ve biçim belirteçleri

Tanım 3.4.12

`System.out.printf(biçim, parametreler…)` bir **biçim dizgisi** ve ardından istenen sayıda parametre alır. Biçim dizgisi, düz metnin arasına yerleştirilmiş **biçim belirteçleri** (*format specifier*) içerir. Her belirteç `%` ile başlar ve bir tip harfiyle biter; aradaki isteğe bağlı kısım bayrak, genişlik ve hassasiyeti verir. Parametreler belirteçlerin yerine sırayla yerleşir.

Bir belirtecin genel biçimi `%[bayrak][genişlik][.hassasiyet]tip`'tir (Şekil 3.4.12). `%-8.2f` belirtecinde `-` bayrağı sola yaslar, 8 genişliği, `.2` hassasiyeti, `f` ise tipi verir.



Şekil 3.4.12: Bir biçim belirtecinin parçaları: başlangıç (`%`), bayrak, genişlik, hassasiyet ve tip.

Tablo 3.4.11 en sık kullanılan tip harflerini verir. Ondalık ayırıcı yerel ayara bağlıdır. Türkçe ayarda virgül, `Locale.US` ile nokta kullanılır.

Belirteç	Tip	Örnek → çıktı
`%d`	Tam sayı (`byte`, `short`, `int`, `long`)	42 → `42`
`%f`	Ondalıklı (`float`, `double`)	`%.1f` : 3.14159 → `3,1`
`%s`	Dizgi (her değer metne çevrilir)	`%5s` : `"ab"` → `␣␣␣ab`
`%c`	Karakter	`'Z'` → `Z`
`%b`	Mantıksal	`true` → `true`
`%n`	Satır sonu	—

Tablo 3.4.11: Temel biçim belirteçleri (çıktılar `tr-TR` ayarında).

`printf` kendiliğinden alt satıra geçmez. Satır sonu için biçim dizgisinin sonuna `\n` ya da `%n` eklenir. `%n` işletim sisteminin satır sonu karakterini kullanır; Windows'ta iki karakterlik `\r\n`, Linux ve macOS'ta `\n` yazar. Bu yüzden taşınabilir kodda `%n` tercih edilir.

Genişlik, değerin kaplayacağı en az karakter sayısıdır. Değer daha kısaysa boşlukla doldurulur: `%Wd` sağa yaslar, `%-Wd` sola yaslar. Değer genişlikten uzunsa kesilmez, alan büyür. Hassasiyet ondalıklı sayılarda virgülden sonraki basamak sayısıdır ve son basamak yuvarlanır: `%.2f` iki basamak yazar. `%8.2f` ise iki ondalık basamakla birlikte toplam 8 karakterlik bir alan ayırır.

3.4.6.2 Genişlik ve hassasiyet

Sunumdaki örnek aynı üç değeri dört farklı biçimle yazar:

```
int i = 3;
float f = -17;
String s = "java";

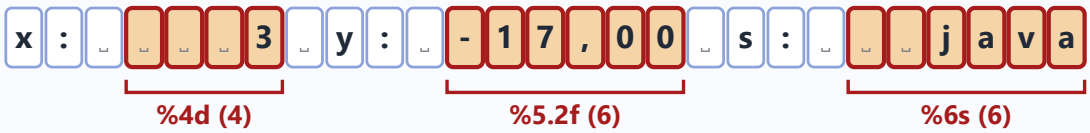
System.out.printf("x: %d y: %f s: %s\n", i, f, s);
System.out.printf("x: %4d y: %5.2f s: %6s\n", i, f, s);
System.out.printf("x: %-4d y: %-7.2f s: %-6s\n", i, f, s);
System.out.printf("x: %-4d y: %-6.3f s: %-6s\n", i, f, s);
```

```
x: 3 y: -17,000000 s: java
x:    3 y: -17,00 s:   java
x: 3    y: -17,00  s: java··
x: 3    y: -17,000 s: java··
```

Çıktı Türkçe ayarda alınmıştır. Satır sonundaki boşluklar görünsün diye `·` ile gösterildi. Değişkenlerin ikisi açıklama ister: `f` bir `float`'tır ve `-17` tam sayısı ona atanınca -17,0 olur. `%f`'ye hassasiyet verilmezse 6 ondalık basamak yazılır. Satırları sırayla inceleyelim:

1. Genişlik verilmemiştir. Her değer kendi uzunluğu kadar yer kaplar: `3`, `-17,000000`, `java`.
2. `%4d` 3'ü dört karakterlik alana sağa yaslar: üç boşluk ve 3. `%5.2f` iki ondalıkla `-17,00` yazar; bu 6 karakterdir ve 5 genişliğe sığmaz, alan büyür. `%6s` dört harfli `java`'nın soluna iki boşluk koyar.
3. Eksi bayrağı sola yaslar; boşluklar değerin sağına eklenir. `%-4d` → `3···`, `%-7.2f` → `-17,00·`, `%-6s` → `java··`.
4. `%-6.3f` üç ondalıkla `-17,000` yazar. Değer 7 karakterdir ve 6 genişlikten uzundur; dolgu yapılmaz.

Şekil 3.4.13 ikinci satırı karakter karakter gösterir. Her belirtecin kapladığı alan alt kısımda işaretlidir.



Şekil 3.4.13: `"x: %4d y: %5.2f s: %6s"` çıktısı karakter karakter. `%4d` 4, `%5.2f` 6 (sığmadığı için büyüyen alan), `%6s` 6 karakter kaplar.

Genişlik en çok tablo yazdırırken işe yarar. Her satırda aynı belirteçler kullanılınca sütunlar alt alta düzgün hizalanır. Metin sütunları genellikle sola (`%-10s`), sayı sütunları sağa (`%6d` , `%8.2f`) yaslanır. Böylece sayıların birler basamağı alt alta gelir.

Örnek 3.4.3

Aşağıdaki satırın Türkçe ayardaki çıktısını karakter karakter bulalım:

```
String ad = "Ali";
double not1 = 72.456;
System.out.printf("[%-5s][%3d][%7.2f]%n", ad, 7, not1);
```

`%-5s` : `"Ali"` üç harflidir, beş karakterlik alana sola yaslanır ve sağına iki boşluk eklenir: `Ali··`. `%3d` : 7 tek basamaklıdır, soluna iki boşluk gelir: `··7`. `%7.2f` : 72,456 iki ondalığa yuvarlanır, 72,46 olur. Bu metin 5 karakterdir; yedi karakterlik alanı doldurmak için başa iki boşluk eklenir: `··72,46`. Sonuç:

```
[Ali  ][  7][  72,46]
```

Yuvarlama kuralı da bilinmeye değer. `printf` , sayının ondalık gösterimine bakar ve yarımı sıfırdan uzağa yuvarlar (*HALF_UP*). `Math.round` 'dan farklı olarak negatif yarımlarda da sıfırdan uzaklaşır:

```
System.out.printf("[%.2f][%.2f][%.0f][%.0f][%.0f]%n", 2.345, 2.355, 0.5, 1.5, -2.5);
System.out.printf(Locale.US, "%,.2f %08.3f %+d %X%n", 1234567.891, 3.14159, 5, 255);
```

```
[2,35][2,36][1][2][-3]
1,234,567.89 0003.142 +5 FF
```

2,345 ve 2,355 beklendiği gibi yukarı yuvarlanır. 0,5 → 1, 1,5 → 2 ve −2,5 → −3 olur. İkinci satırdaki bayraklar sonraki alt kesimde açıklanıyor.

3.4.6.3 Diğer biçimler ve String.format

```
System.out.printf("|%5s|%-5s|%05d|%.1f|%n", "ab", "ab", 42, 3.14159);
System.out.printf("%c %b %e %x %%%n", 'Z', true, 1234.5, 255);
String mesaj = String.format("%s: %d puan", "Ayşe", 85);
System.out.println(mesaj);
System.out.printf("%,d%n", 1234567);
System.out.printf(Locale.US, "%.2f%n", 2.5);       // import java.util.Locale;
System.out.println("%s = %d".formatted("x", 7));   // Java 15+
```

```
|   ab|ab   |00042|3,1|
Z true 1,234500e+03 ff %
Ayşe: 85 puan
1.234.567
2.50
x = 7
```

Bu örnekteki yeni öğeler şunlardır:

- `%05d` : `0` bayrağı alanı boşluk yerine sıfırla doldurur: `00042`. Fatura ve öğrenci numarası gibi sabit uzunluklu numaralarda kullanılır.
- `%e` bilimsel gösterimdir: 1234,5 = 1,2345 × 10^3, çıktıda `1,234500e+03`. `%x` onaltılık tabanda yazar: 255 → `ff`; büyük harf için `%X` . `%%` yüzde işaretinin kendisini yazar.
- `String.format` aynı kuralları kullanır ama ekrana yazmaz; biçimlenmiş metni bir `String` olarak döndürür. Sonuç bir değişkende saklanabilir, bir dosyaya yazılabilir ya da başka bir metne eklenebilir.
- `%,d` binlik ayırıcı koyar. Türkçe ayarda ayırıcı noktadır: `1.234.567`. ABD ayarında virgüldür.
- `printf` 'in ilk parametresi bir `Locale` olabilir. `Locale.US` verilince ondalık ayırıcı nokta olur: `2.50`. Başka bir programın okuyacağı veriler (CSV dosyası gibi) bu yolla yazılır.
- Java 15 ile gelen `formatted` metodu biçim dizgisi üzerinden çağrılır: `"%s = %d".formatted("x", 7)` ile `String.format("%s = %d", "x", 7)` aynı sonucu verir.

Biçim dizgisi derleyici için sıradan bir metindir. Belirteç ile parametrenin tipi uyuşmazsa hata derleme sırasında değil, çalışırken ortaya çıkar. `printf("%d", 3.5)` çağrısı `%d` 'nin beklediği tam sayı yerine bir `double` alır ve istisna fırlatır:

```
Exception in thread "main" java.util.IllegalFormatConversionException:
    d != java.lang.Double
```

İleti “ `d` belirteci `Double` ile kullanılamaz” demektir. `printf` dönüşüm yapmaz; sayıyı kesip 3 yazmaz. Doğrusu `%f` ya da `%.0f` kullanmaktır. Belirteç sayısı parametre sayısından fazlaysa `MissingFormatArgumentException` fırlatılır. `%s` ise her tiple çalışır, çünkü her değer metne çevrilebi-

lir.

3.4.7 Uygulamalar

Önceki kesimlerin araçları, yani indisleme, döngü, karşılaştırma ve birleştirme, birlikte kullanılınca gerçek problemler çözülür. Bu kesim sunumdaki iki uygulamayı ve alıştırma slaytlarındaki küçük metotları ayrıntılı işler.

3.4.7.1 Palindrom denetimi

Tanım 3.4.13

Palindrom, tersten okunuşu düzden okunuşuyla aynı olan metindir: “kayak”, “ana”, “kek”. Cümlelerde boşluk ve noktalama işaretleri yok sayılır.

Bir metnin palindrom olup olmadığını anlamanın en doğrudan yolu metni ters çevirip orijinaliyle karşılaştırmaktır:

```
String metin = "Kayak";
metin = metin.toLowerCase();
String ters = "";
for (int i = metin.length() - 1; i >= 0; i--) {
  ters += metin.charAt(i);
}
System.out.println(ters);
System.out.println(metin.equals(ters));
```

```
kayak
true
```

Kod üç adımda çalışır. Önce metin küçük harfe çevrilir; çevrilmeseydi `K` ile `k` farklı sayılırdı. `toLowerCase()` yeni bir dizgi döndürdüğü için sonuç yeniden `metin`'e atanır. İkinci adımda boş bir `ters` dizgisi hazırlanır ve döngü son indisten 0'a doğru iner. Her turda o indisteki karakter `ters`'in sonuna eklenir. Üçüncü adımda iki dizgi `equals` ile karşılaştırılır. Tablo 3.4.12 döngünün turlarını verir.

`i`	`i >= 0`	`charAt(i)`	`ters` **(tur sonunda)**
4	doğru	`'k'`	`"k"`
3	doğru	`'a'`	`"ka"`
2	doğru	`'y'`	`"kay"`

`i`	`i >= 0`	`charAt(i)`	`ters` **(tur sonunda)**
1	doğru	`'a'`	`"kaya"`
0	doğru	`'k'`	`"kayak"`
−1	yanlış	—	döngü biter

Tablo 3.4.12: `metin = "kayak"` için ters çevirme döngüsü.



Şekil 3.4.14: Döngü bittiğinde `metin` ve `ters`. İçerikler aynıdır; `equals` `true` döndürür.

Karşılaştırmanın `equals` ile yapılması zorunludur. `ters` çalışma zamanında birleştirmeyle kurulmuş yeni bir nesnedir; `metin == ters` her zaman `false` verirdi. Döngüdeki `ters += …` satırı her turda yeni bir dizgi üretir. Kısa metinlerde bunun önemi yoktur, ama aynı iş `new StringBuilder(metin).reverse().toString()` ile hem kısa hem verimli yapılır.

Cümlelerde önce harf olmayan karakterler silinir. `metin.replaceAll("[^\\p{L}]", "")` çağrısı düzenli ifade kullanır: `\p{L}` Unicode'daki her harfi, `[^…]` ise “bunlar dışındaki her karakteri” anlatır. Böylece yalnız harfler kalır. “Ey Edip Adana'da pide ye” cümlesi temizlenip küçültülünce `"eyedipadanadapideye"` olur ve bu bir palindromdur.

Ters çevirme yöntemi metnin bir kopyasını oluşturur. Bellek harcamadan aynı denetim iki indisle de yapılır: biri baştan, biri sondan başlar ve ortada buluşana kadar karakterleri karşılaştırır. Bir eşitsizlik bulunduğu anda sonuç `false`'tur; geri kalan karakterlere bakmaya gerek kalmaz. İki yöntemin de maliyeti metnin uzunluğuyla doğru orantılıdır.

3.4.7.2 Kelime frekansı

Bir metinde her kelimenin kaç kez geçtiğini saymak, metin işlemenin en temel problemlerinden biridir. Sunumdaki çözüm yalnız dizilerle ve iç içe iki döngüyle çalışır:

```
String metin = "Bir iki bir üç İki bir";
String[] kelimeler = metin.toLowerCase().split(" ");
boolean[] sayildi = new boolean[kelimeler.length];
for (int i = 0; i < kelimeler.length; i++) {
  if (sayildi[i]) {
    continue; // bu kelime daha önce sayıldı
  }
  int frekans = 1;
  for (int j = i + 1; j < kelimeler.length; j++) {
    if (kelimeler[i].equals(kelimeler[j])) {
      frekans++;
      sayildi[j] = true;
    }
  }
  System.out.println(kelimeler[i] + ": " + frekans);
}
```

```
bir: 3
iki: 2
üç: 1
```

Kod şu adımları izler:

- `metin.toLowerCase().split(" ")` önce metni küçültür, sonra boşluklardan böler. Küçültme sayesinde "Bir/bir" ve "İki/iki" aynı kelime sayılır. Türkçe ayarda `İ` küçültülünce `i` olur. Dilden bağımsız kurallarla ise `İ`, `i` ile birleşen noktaya dönüşür ve "İki" ayrı bir kelime gibi sayılır; çıktı Türkçe ayara göredir.
- `sayildi` dizisi her kelime için bir bayrak tutar. `boolean` dizisi oluşturulduğunda bütün öğeler `false`'tur.
- Dış döngü her kelimeye bir kez bakar. Kelimenin bayrağı `true` ise o kelime daha önce sayılmıştır; `continue` turun geri kalanını atlar.
- İç döngü yalnız sağdaki kelimeleri tarar. Eşleşen her kelime için sayaç artar ve eşleşenin bayrağı `true` yapılır. Böylece aynı kelime ikinci kez raporlanmaz.

Kelimeler `equals` ile karşılaştırılır. `split`'in ürettiği parçalar çalışma zamanında oluşan ayrı nesnelerdir. `==` kullanılsaydı hiçbir eşleşme bulunmaz ve her kelimenin frekansı 1 çıkardı. Bu durum çalıştırılarak doğrulandı: `==` ile "bir" kelimesi yalnız bir kez sayılır.

Sunumdaki izleme slaytı aynı kodu `"elma Armut elma kiraz ARMUT"` metni üzerinde çalıştırır:

```
String metin = "elma Armut elma kiraz ARMUT";
String[] k = metin.toLowerCase().split(" ");
boolean[] sayildi = new boolean[k.length];
for (int i = 0; i < k.length; i++) {
  if (sayildi[i]) continue;
  int frekans = 1;
  for (int j = i + 1; j < k.length; j++) {
    if (k[i].equals(k[j])) {
      frekans++;
      sayildi[j] = true;
    }
  }
  System.out.println(k[i] + ": " + frekans);
}
```

```
elma: 2
armut: 2
kiraz: 1
```

`i`	`k[i]`	**Eşleşen** `j`	`frekans`	`sayildi`	**Çıktı**
0	elma	2	2	F F **T** F F	`elma: 2`
1	armut	4	2	F F T F **T**	`armut: 2`
2	elma	—	—	F F T F T	bayrak açık, `continue`
3	kiraz	yok	1	F F T F T	`kiraz: 1`
4	armut	—	—	F F T F T	bayrak açık, `continue`

Tablo 3.4.13: Kelime frekansı izlemesi. `k = ["elma", "armut", "elma", "kiraz", "armut"]`; bayraklar her dış turun sonundadır.



Şekil 3.4.15: `i` = 0 turunun sonunda kelime dizisi ve bayraklar. İndis 2'deki "elma" sayıldı olarak işaretlenmiştir.

Algoritmanın maliyeti kelime sayısının karesiyle büyür. *n* kelime için iç döngü en kötü durumda *(n − 1) + (n − 2) + … + 1 = n(n − 1)/2* karşılaştırma yapar. Birkaç cümlelik bir metin için bu çok azdır. Bir kitabın bütün kelimeleri sayılacaksa anahtar-değer çiftleri tutan bir koleksiyon (`HashMap`) çok daha hızlı sonuç verir.

3.4.7.3 Küçük dizgi metotları

Sunumun alıştırma ve örnek problem slaytları, dizgi metotlarını birleştiren birkaç kısa metot içerir. Bu metotlar indisleme ve döngünün ortak kalıplarını gösterir.

Ters çevirme. `tersCevir` metodu palindrom örneğindeki döngüyü bir metoda taşır:

```
static String tersCevir(String s) {
  String ters = "";
  for (int i = s.length() - 1; i >= 0; i--) {
    ters += s.charAt(i);
  }
  return ters;
}
```

`tersCevir("Giresun")` çağrısı `nuseriG` döndürür. Büyük `G` sona geçmiştir; metot harf büyüklüğüne dokunmaz.

Karakter sayma. `say` metodu bir karakterin kaç kez geçtiğini bulur:

```
static int say(String s, char aranan) {
  int adet = 0;
  for (int i = 0; i < s.length(); i++) {
    if (s.charAt(i) == aranan) {
      adet++;
    }
  }
  return adet;
}
```

`s.charAt(i) == aranan` karşılaştırması doğrudur, çünkü iki yan da `char`'dır ve `char` ilkel bir tiptir. `==` burada referansı değil değeri karşılaştırır. `say("Karadeniz", 'a')` 2 döndürür: `K` büyük harftir ve `'a'` ile eşleşmez; eşleşen yalnız indis 1 ve 3'teki `a`'lardır.

Baş harfi büyütme. `basHarfBuyuk` ilk harfi büyük, kalanını küçük yapar:

```
static String basHarfBuyuk(String kelime) {
  return kelime.substring(0, 1).toUpperCase()
       + kelime.substring(1).toLowerCase();
}
```

Metot iki parçayı ayrı ayrı işleyip birleştirir: `substring(0, 1)` ilk harf, `substring(1)` kalanıdır. `basHarfBuyuk("jAVA")` → `Java`. Türkçe ayarda `basHarfBuyuk("istanbul")` → `İstanbul`, `basHarfBuyuk("ılgaz")` → `Ilgaz` verir. Boş dizgiyle çağrılırsa `substring(0, 1)` geçersiz bir aralık

ister ve `StringIndexOutOfBoundsException` fırlatılır. Sağlam bir sürüm başta `if (kelime.isEmpty()) return kelime;` denetimi yapar.

Sesli harf sayma. Bir karakterin bir kümede olup olmadığını anlamak için `indexOf` kullanılabilir:

```
String kelime = "Ordu";
String sesliler = "aeıioöuüAEIİOÖUÜ";
int sayac = 0;
for (int i = 0; i < kelime.length(); i++) {
  char c = kelime.charAt(i);
  if (sesliler.indexOf(c) != -1) {
    sayac++;
  }
}
System.out.println("Sesli: " + sayac);
```

```
Sesli: 2
```

`sesliler` dizgisi Türkçenin sekiz sesli harfini büyük ve küçük biçimleriyle içerir. Her karakter için `sesliler.indexOf(c)` çağrılır; sonuç −1 değilse karakter seslidir. `"Ordu"`'da `O` ve `u` sesli harftir: `Sesli: 2`. Bu kalıp sekiz dallı bir `if` zincirinden hem kısa hem okunaklıdır.

Kelime sayısı. Kelimeler arasında tek boşluk varsa kelime sayısı tek satırda bulunur: `cumle.split(" ").length`. `"Java ile nesne yönelimli programlama"` için sonuç 5'tir. Kelimeler arasında birden çok boşluk olabiliyorsa `cumle.trim().split(" +").length` yazılır.

Örnek 3.4.4

Sunumdaki ilk örnek problem değiştirilemezlik, `substring`, `indexOf` ve birleştirme sırasını bir arada sınar:

```
String s = "Karadeniz";
s.toUpperCase();
String t = s.substring(2, 5);
System.out.println(t + " " + t.length());
System.out.println(s.indexOf("de"));
System.out.println(s.indexOf("x"));
System.out.println(1 + 2 + "3" + 4 + 5);
```

```
rad 3
4
-1
3345
```

`s.toUpperCase()` sonucu atanmadığı için `s` değişmez. `"Karadeniz"`'in indisleri K0 a1 r2 a3 d4 e5 n6 i7 z8'dir. `substring(2, 5)` 2, 3, 4 indislerini alır: `"rad"`, uzunluk 3. `"de"` indis 4'te başlar. `"x"` dizgide yoktur, −1. Son satır soldan sağa hesaplanır: 1 + 2 = 3 (iki sayı), 3 + `"3"` = `"33"`, ardından `"334"` ve `"3345"`.

İkinci problem havuzu sınar:

```
String a = "ders";
String b = "de" + "rs";
String c = new String("ders");
String d = c;
System.out.println(a == b);
System.out.println(a == c);
System.out.println(c == d);
System.out.println(a.equals(c));
```

```
true
false
true
true
```

`"de" + "rs"` derleme zamanında hesaplanan bir sabittir ve havuzdaki `a` ile aynı nesneyi gösterir: `a == b` `true`. `c` `new` ile oluşmuş ayrı bir nesnedir: `a == c` `false`. `d = c` referansı kopyalar; ikisi aynı nesnedir: `true`. İçerikler aynı olduğu için `a.equals(c)` `true`'dur.

Üçüncü problem baş harfleri ve e-posta parçalarını çıkarır:

```
public static String bas(String ad, String soyad) {
  return ad.charAt(0) + "." + soyad.charAt(0) + ".";
}
System.out.println(bas("Sercan", "Külcü"));
String e = "ogrenci@giresun.edu.tr";
int at = e.indexOf('@');
System.out.println(e.substring(0, at));
System.out.println(e.substring(at + 1));
```

```
S.K.
ogrenci
giresun.edu.tr
```

`ad.charAt(0) + "."` ifadesinde sağ yan bir dizgi olduğu için birleştirme yapılır: `"S."`. İki `char` doğrudan toplansaydı sonuç bir sayı olurdu; `'S' + '.'` 83 + 46 = 129 verir. E-posta kısmı daha önce çözülen örneğin aynısıdır.

3.4.8 Sık yapılan hatalar

Dizgilerle ilgili hataların bir kısmını derleyici yakalar; bunların iletisi okunursa düzeltmesi kolaydır. Bir kısmı ise derlenir ve çalışırken yanlış sonuç verir ya da istisna fırlatır. Tablo 3.4.14 ilk grubu, Tablo 3.4.15 ikinci grubu özetler. İletiler JDK 21'in `javac`'ından alınmıştır.

Hata	Derleyici iletisi	Düzeltme
`String s = 'Merhaba';`	`unclosed character literal`	Tek tırnak yalnız tek karakter içindir; dizgi çift tırnakla yazılır.
`char c = "a";`	`incompatible types: String cannot be converted to char`	`char c = 'a';` ya da `"a".charAt(0)`.
`s.length`	`cannot find symbol` / `symbol: variable length`	Dizgide `length()` metottur. Dizide tersi: `d.length`.
`string s = …;`	`cannot find symbol` / `symbol: class string`	Sınıf adı `String`'dir; Java büyük-küçük harfe duyarlıdır.
`int n = "42";`	`incompatible types: String cannot be converted to int`	`Integer.parseInt("42")`.
`s.charAt(0) = 'j';`	`unexpected type` / `required: variable`	Dizgi değiştirilemez; yeni dizgi ya da `StringBuilder.setCharAt`.
`s[0]`	`array required, but String found`	`s.charAt(0)`.
`if (a < b)` (dizgiler)	`bad operand types for binary operator '<'`	`a.compareTo(b) < 0`.

Tablo 3.4.14: Derleyicinin yakaladığı dizgi hataları.

İlk satırdaki hata için `javac` aslında üç ileti verir: iki kez `unclosed character literal` ve bir kez `not a statement`. Derleyici tek tırnağı bir karakter sabitinin başlangıcı sanar, tırnağın kapanmadığını görür ve ardından gelenleri anlamlandıramaz. Bu tür zincirleme iletilerde ilk ileti asıl nedeni gösterir:

```
Tirnak.java:3: error: unclosed character literal
        String s = 'Merhaba';
                   ^
```

Hata	Belirti	Düzeltme
İçeriği `==` ile karşılaştırmak	Kullanıcı girdisi doğru olsa da koşul `false`.	`"evet".equals(girdi)`
Sonucu atamayı unutmak	`s.toUpperCase();` sonrası `s` değişmez.	`s = s.toUpperCase();`

Hata	Belirti	Düzeltme
Son indisi `length()` sanmak	`StringIndexOutOfBoundsException: Index 4 out of bounds for length 4`	`s.charAt(s.length() - 1)`; döngüde `i < s.length()`
`nextInt` sonrası `nextLine`	Satır boş okunur, soru atlanır.	Araya bir `klavye.nextLine();`
`null` dizgide metot çağırmak	`NullPointerException: Cannot invoke "String.equals(Object)"`	Sabiti öne yazmak: `"evet".equals(cevap)`
`substring`'in bitişini dahil sanmak	`s.substring(5, 12)` 8 değil 7 karakter verir.	Uzunluk = bitiş − başlangıç.
Parantezsiz toplama	`"Toplam: " + x + y` → `Toplam: 105`	`"Toplam: " + (x + y)`
Varsayılan yerel ayarla harf dönüşümü	`"TITLE".toLowerCase()` Türkçe ayarda `tıtle`.	`toLowerCase(Locale.ROOT)` ya da `equalsIgnoreCase`
`printf`'te yanlış belirteç	`IllegalFormatConversionException`	Tam sayıya `%d`, ondalıklıya `%f`.

Tablo 3.4.15: Derlenen ama yanlış çalışan dizgi kodları.

İkinci tablodaki hataların ortak noktası derleyicinin onları görememesidir. `==` ve `equals` ikisi de geçerli Java'dır. Sonucu atanmayan bir metot çağrısı da geçerli bir deyimdir. Bu hatalara karşı en iyi koruma, programı farklı girdilerle denemek ve çıktıyı beklenen değerle karşılaştırmaktır. Özellikle boş dizgi, tek karakterli dizgi, Türkçe harfler ve baştaki-sondaki boşluklar iyi birer deneme girdisidir.

3.4.9 Özet

- **String** karakter dizisini saklayan, değiştirilemez bir nesnedir. Dizgi sabiti (`"..."`) ya da `new` ile oluşur; `java.lang` paketinde olduğu için `import` gerektirmez.
- **Birleştirme:** `+` soldan sağa işlenir. İşlenenlerden biri dizgiyse birleştirme, ikisi de sayıysa toplama yapılır. `char` bir sayıdır: `'a' + 'b'` = 195. Sayıları toplamak için parantez kullanılır.
- **İndis** 0'dan `length() - 1`'e gider. `charAt(i)` bir `char` döndürür. Dışarı taşan indis `StringIndexOutOfBoundsException` fırlatır. Dizgide `length()` metottur, dizide `length` alandır.
- **Metotlar:** `indexOf` (bulunamazsa −1), `substring(a, b)` (`b` hariç, uzunluk `b - a`), `toUpperCase`, `replace`, `trim`, `split`, `join`; hepsi yeni değer döndürür ve sonucun atanması gerekir.
- **Yerel ayar:** parametresiz büyük-küçük harf dönüşümü makinenin ayarına bağlıdır. Türkçe ayarda `i ↔ İ`, `ı ↔ I`. Program içi metinlerde `Locale.ROOT` kullanılır.
- **StringBuilder:** değiştirilebilir tampondur. Döngüde `+=` ile birleştirme $O(n^2)$, `append` ile $O(n)$ iş yapar. Kapasite dolunca yaklaşık iki katına çıkar (20 → 42).
- **Scanner:** `next()` bir kelime, `nextLine()` satırın kalanını okur. `nextInt()` sonrası ⏎ tamponda kalır; ardından gelen `nextLine()` boş dizgi döndürür.
- `==` / `equals`: `==` aynı nesne mi diye, `equals` aynı içerik mi diye sorar. Dizgilerin içeriği her zaman `equals` ile karşılaştırılır; `null`'a karşı sabit öne yazılır.

- **Dizgi havuzu:** içeriği aynı dizgi sabitleri tek nesneyi paylaşır. `new` ve çalışma zamanındaki birleştirme yeni nesne üretir; `intern()` havuzdaki eşini verir.
- **compareTo:** ilk farklı karakterin kod farkını, önek durumunda uzunluk farkını döndürür. Unicode sırasını kullanır; Türkçe alfabe sırası için `Collator` gerekir.
- **printf:** `%d`, `%f`, `%s`, `%c`, `%b`, `%n`. `%Wd` sağa, `%-Wd` sola yaslar; `%.2f` iki ondalık basamak yazar. Sığmayan değer alanı büyütür. Ayırıcılar yerel ayara bağlıdır.

3.4.10 Kavram listesi

dizgi *string* · dizgi sabiti *string literal* · dizgi birleştirme *string concatenation* · karakter (char) *character* · Unicode *Unicode* · indis *index* · indisleme *indexing* · değiştirilemez *immutable* · referans *reference* · heap *heap* · çöp toplayıcı *garbage collector* · yerel ayar *locale* · tampon *buffer* · dizgi havuzu *string pool* · null *null reference* · biçim belirteci *format specifier* · derleme zamanı / çalışma zamanı *compile time / run time* · çalışma zamanı hatası *runtime error* · istisna *exception* · sabit değer *literal* · referans tipi *reference type*

3.4.11 Kendini Dene

1. `System.out.println(2 + 3 + "x" + 2 + 3);` ne yazdırır? Sondaki iki sayının toplamının yazılması için ifade nasıl değiştirilir?

   Cevap

   `5x23` yazar. Soldaki iki sayı dizgiyle karşılaşılmadan önce toplanır (5). `"x"` 'ten sonraki her `+` birleştirmedir. Sondaki toplam için parantez gerekir: `2 + 3 + "x" + (2 + 3)` → `5x5`.

2. `String s = "Karadeniz Teknik";` için `s.indexOf(' ')`, `s.substring(s.indexOf(' ') + 1)` ve `s.lastIndexOf('e')` kaçtır?

   Cevap

   “Karadeniz” dokuz harflidir; boşluk indis 9'dadır. `substring(10)` boşluktan sonrasını verir: `Teknik`. `T` indis 10'da, `e` indis 11'dedir ve sonrasında başka `e` yoktur: `lastIndexOf('e')` = 11.

3. `StringBuilder sb = new StringBuilder("abc"); sb.append("de").insert(2, "X").reverse();` satırlarından sonra `sb` 'nin içeriği ve uzunluğu nedir? Aynı işlemler `String` ile

yapılsaydı kaç nesne oluşurdu?

Cevap

`"abc"` → `append` → `"abcde"` → `insert(2, "X")` → `"abXcde"` → `reverse` → `edcXba`, uzunluk 6. Bütün işlemler aynı nesne üzerinde yapılır. `String` ile her adım yeni bir dizgi üretirdi: başlangıç dizgisinin yanında üç yeni nesne daha.

4. `"Ankara".compareTo("Antalya")` ve `"abc".compareTo("ab")` ne döndürür? Bu iki dizgi alfabetik sıraya konursa hangisi önce gelir?

Cevap

İlk çiftte indis 2'de `'k'` (107) ile `'t'` (116) ayrılır: 107 − 116 = −9. Negatif olduğu için "Ankara" önce gelir. İkinci çiftte `"ab"`, `"abc"`'nin önekidir; sonuç uzunluk farkıdır: 3 − 2 = 1. "ab" önce gelir.

5. `String x = "Ada"; String y = "A" + "da"; String z = new String("Ada"); String w = z.intern();` satırlarından sonra `x == y`, `x == z`, `x == w` ve `z.equals(y)` ne verir?

Cevap

`true false true true`. `"A" + "da"` derleme zamanında `"Ada"` sabitine dönüşür ve havuzdaki nesneyi gösterir. `z` havuz dışında ayrı bir nesnedir. `intern()` havuzdaki eşini, yani `x`'in nesnesini döndürür. İçerikler aynı olduğu için `equals` `true` verir.

6. Türkçe ayarlı bir bilgisayarda `System.out.printf("[%6.1f][%-4s][%3d]%n", 9.96, "ok", 12345);` çıktısı nedir?

Cevap

`[ 10,0][ok  ][12345]`. 9,96 bir ondalığa yuvarlanınca 10,0 olur; bu 4 karakterdir ve başa iki boşluk eklenir. `"ok"` sola yaslanır, sağa iki boşluk gelir. 12345 beş basamaklıdır ve 3 genişliğe sığmaz; alan büyür, sayı kesilmez.

7. Klavyeden önce `␣␣42␣␣Ayşe Yılmaz⏎`, sonra `Son⏎` girilir. `int n = k.nextInt(); String r = k.nextLine(); String t = k.next();` satırlarından sonra `n`, `r` ve `t` nedir?

Cevap

`nextInt()` baştaki boşlukları atlar ve 42'yi okur. `nextLine()` boşlukları atlamaz; satırın kalanını alır: `r = "  Ayşe Yılmaz"` (başta iki boşluk). ⏎ tüketildiği için `next()` ikinci satırdaki kelimeyi okur: `t = "Son"`.

8. `String u = "Merhaba"; u.replace('a', 'e'); u.concat("!"); System.out.println(u);` ne yazdırır? Programcı `Merhebe!` görmek istiyorsa kod nasıl düzeltilir?

   Cevap

   `Merhaba` yazar. İki metot da yeni dizgi döndürür ama sonuçlar atanmadığı için kaybolur. Düzeltilmiş hâli: `u = u.replace('a', 'e').concat("!");` ya da iki ayrı atama.

9. `"a-b--c".split("-")` kaç öğeli bir dizi döndürür? `"KIRMIZI".toLowerCase().equals("kirmizi")` Türkçe ayarlı bir bilgisayarda ne verir?

   Cevap

   Ardışık iki tire arasında boş bir parça oluşur: `["a", "b", "", "c"]`, 4 öğe. İkinci ifade `false` verir: Türkçe kurallarla `I` küçültülünce `ı` olur ve sonuç `"kırmızı"`'dır. `toLowerCase(Locale.ROOT)` ya da `equalsIgnoreCase` `true` verir.

3.4.12 İleri okuma

- Oracle, *The Java Tutorials*, “Numbers and Strings” kesimi: `String`, `StringBuilder` ve biçimli sayı çıktısını kısa örneklerle anlatan resmî öğretici.
- Oracle, *Java SE 21 API Documentation*, `java.util.Formatter` sınıfı: `printf`'te kullanılabilen bütün bayrakların, genişlik ve hassasiyet kurallarının ve tip harflerinin eksiksiz listesi.
- J. Gosling vd., *The Java Language Specification*, Java SE 21, §3.10.5 ve §15.18.1: dizgi sabitlerinin havuzda paylaşılmasını ve `+` ile birleştirmenin kurallarını kesin biçimde tanımlar.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, Bölüm 3.6: dizgileri, `StringBuilder`'ı ve Unicode'un Java'daki karşılığını deneyimli bir programcının gözünden işler.

3.4.13 Kaynaklar

1. Oracle, *The Java Tutorials*: “Strings”, “Comparing Strings and Portions of Strings”, “The StringBuilder Class”, “Formatting Numeric Print Output”.
2. Oracle, *Java SE 21 API Documentation*: `java.lang.String`, `java.lang.StringBuilder`, `java.util.Scanner`, `java.util.Formatter`, `java.util.Locale`.
3. J. Gosling vd., *The Java Language Specification*, Java SE 21 Edition (§3.10.5 String Literals, §15.18.1 String Concatenation Operator +).
4. P. Deitel, H. Deitel, *Java: How to Program*, 11. baskı, Pearson, 2017 (Bölüm 14: Strings, Characters and Regular Expressions).
5. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, 2022 (Bölüm 3.6: Strings).
6. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019 (Bölüm 6: Loops and Strings).

KISIM I · BÖLÜM 3.5

Özyineleme

Bazı problemler kendi içlerinde kendilerinin daha küçük bir kopyasını taşır. 5!, 4!'in 5 katıdır; 48729'un rakamları toplamı, 4872'nin rakamları toplamına 9 eklenerek bulunur. Özyineleme, bu yapıyı doğrudan koda döker: metot, işin küçük kısmını kendisine yeniden sorar ve gelen cevaptan kendi cevabını kurar. Önce taban durum, özyinelemeli adım ve çağrı yığını tanıtılır. Ardından faktöriyel, toplam, dört sayı serisi, rakamlar toplamı, palindrom, Öklid algoritması ve Hanoi kuleleri Java ile yazılır ve çağrı çağrı izlenir. Son kesimler özyinelemenin bedelini ele alır: yığın taşması, aynı değerin defalarca hesaplanması, döngüye çevirme, kuyruk özyinelemesi ve bellekleme. Kodların hepsi JDK 21 ile çalıştırılmış; çıktılar, çağrı sayıları ve süreler bu çalıştırmalardan alınmıştır.

ÖĞRENME ÇIKTILARI

Kavram
Özyinelemeyi, taban durumu ve özyinelemeli adımı tanımlamak; bir problemi kendisinin küçük bir kopyasına indirgemek.

Çalışma biçimi
Çağrı yığınında çerçevelerin nasıl biriktiğini ve geri döndüğünü adım adım izlemek; çağrı ağacı çizmek.

Yazma
Faktöriyel, toplam, seri, rakamlar toplamı, palindrom, EBOB ve Hanoi kuleleri için özyinelemeli Java metotları yazmak.

Hatalar
Sonsuz özyinelemeyi, `StackOverflowError`'ı ve eksik `return` hatalarını tanımak ve önlemek.

Verimlilik
Özyinelemeli ve ilerlemeli çözümleri karşılaştırmak; gerektiğinde özyinelemeli bir metodu döngüye çevirmek.

İleri teknikler
Kuyruk özyinelemesini ve belleklemeyi açıklamak; tekrarlanan hesaplamaları ortadan kaldırmak.

3.5.1 Özyineleme kavramı

Metot, bir işi adlandırıp tek satırla çağrılabilir hâle getirir. `main` bir metodu çağırabilir, o metot da başka bir metodu çağırabilir. Bir metodun çağırdığı metot kendisi de olabilir. Java bunu yasaklamaz ve bu olanak, bazı problemleri döngülerden çok daha kısa ve açık yazmayı sağlar.

3.5.1.1 Özyineleme nedir?

> **Tanım 3.5.1**
>
> **Özyineleme** (*recursion*), bir metodun görevini küçük parçalara bölerek kendini çağırmasıdır. Her çağrı problemin daha küçük ama aynı tipten bir kopyasını çözer; en küçük kopya ise doğrudan çözülür. Kendini çağıran metoda **özyinelemeli metot** denir.

Java'da özyineleme için özel bir anahtar sözcük yoktur. Metot gövdesinde kendi adını, sıradan bir metot çağırır gibi çağırır. Derleyici için `faktoriyel`'in `faktoriyel`'i çağırması, `main`'in `println`'i çağırmasından farklı değildir. Fark, çalışma sırasında ortaya çıkar: aynı metodun birden çok çağrısı aynı anda bellekte bekler.

Özyinelemenin gündelik hayatta üç tanıdık karşılığı vardır. **Matruşka bebeklerinde** her bebeğin içinde daha küçük bir bebek bulunur. En küçük bebek açılmaz; özyinelemedeki karşılığı taban durumdur. **Sinema salonunda** karanlıkta oturan biri kaçıncı sırada olduğunu öğrenmek istesin. Öndekine sorar, o da öndekine sorar. İlk sıradaki kişi “1” der. Cevaplar geri gelirken her kişi duyduğu sayıya 1 ekler ve arkasındakine söyler. Soru öne doğru ilerler, cevap arkaya doğru döner. Özyinelemeli bir metodun çalışması tam olarak böyledir. **Karşılıklı iki aynada** ise görüntünün içinde görüntü, onun içinde yine görüntü belirir ve bu dizi bitmez. Bu da taban durumu olmayan özyinelemenin resmidir.

Özyineleme her problem için doğru seçim değildir. Ama bazı problem türlerinde döngüden çok daha anlaşılır bir çözüm verir. Ağaç biçimli veriler, örneğin klasörler ve alt klasörler, bu türdendir. Tanımı kendine dayanan sayı serileri de öyledir. Problemi ikiye bölüp parçaları ayrı çözen böl ve fethet (*divide and conquer*) algoritmaları da bu gruba girer. Hanoi kuleleri gibi bazı bulmacalar döngüyle zor, özyinelemeyle birkaç satırda yazılır.

3.5.1.2 Taban durum ve özyinelemeli adım

Her özyinelemeli metot iki parçadan oluşur. Biri durmayı, öteki ilerlemeyi sağlar.

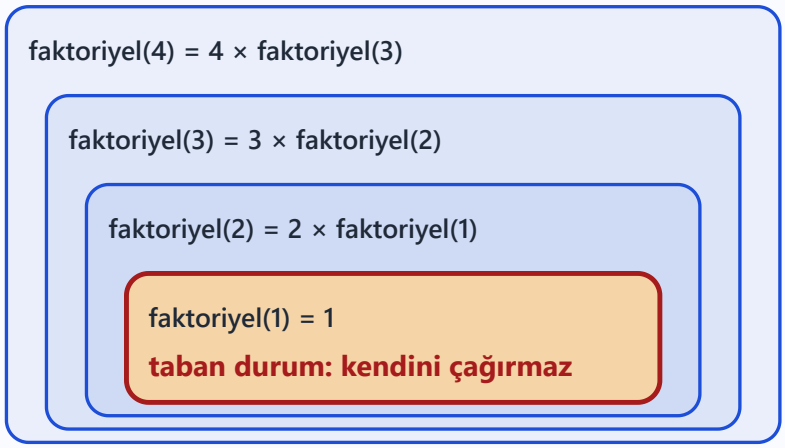
Tanım 3.5.2

Taban durum (*base case*), cevabı doğrudan bilinen en küçük problemdir. Bu durumda metot kendini çağırmaz ve özyineleme durur. **Özyinelemeli adım** (*recursive step*) ise problemi bir adım küçültüp metodun kendisine sorar ve gelen cevabı kullanarak kendi cevabını kurar.

Faktöriyel bu iki parçayı en açık gösteren örnektir. 1! = 1 bilinir; bunun için hesap gerekmez. 4! ise doğrudan bilinmez, ama 3! bilinirse 4! = 4 × 3! diye bulunur. Matematikteki tümevarımsal tanım, özyinelemeli metodun birebir karşılığıdır:

> $n! = 1$ ($n \le 1$ ise), $n! = n \times (n-1)!$ ($n > 1$ ise)

İlk satır taban durumdur, ikinci satır özyinelemeli adımdır. Tanım döngüsel görünse de değildir: 4! tanımlanırken 3!, 3! tanımlanırken 2! kullanılır. Her adımda argüman küçülür ve sonunda 1'e varılır. Şekil 3.5.1 bu iç içe yapıyı kutularla gösterir.



Şekil 3.5.1: `faktoriyel(4)` problemi, her katta bir küçülerek `faktoriyel(1)` taban durumuna iner. En içteki kutu kendini çağırmaz; cevabı doğrudan 1'dir.

İki parçanın ikisi de zorunludur. Taban durum yoksa metot kendini sonsuza kadar çağırır. Özyinelemeli adım problemi küçültmüyorsa taban duruma hiç varılmaz ve sonuç yine aynıdır. Özyinelemeli her metodun uyması gereken iki temel kural bu iki koşuldan doğar.

3.5.1.3 Çağrı yığını

Özyinelemenin nasıl çalıştığını anlamak için Java'nın metot çağrılarını bellekte nasıl tuttuğunu bilmek gerekir.

Tanım 3.5.3

Her metot çağrısı için bellekte bir **yığın çerçevesi** (*stack frame*) açılır. Çerçevede çağrının parametreleri, yerel değişkenleri ve metot bitince dönülecek yer tutulur. Çerçeveler **çağrı yığınına** (*call stack*) üst üste konur; yalnız tepedeki çerçeve çalışır, alttakiler bekler.

Yığın, üst üste konan tabaklara benzer. Yeni tabak en üste konur, alınacak tabak da en üstten alınır. Bu düzene **son giren ilk çıkar** (*last in, first out*, LIFO) denir. Bir metot çağrıldığında çerçevesi yığının tepesine eklenir. Metot `return` ile bittiğinde çerçevesi silinir ve döndürdüğü değer bir alttaki çerçeveye iletilir. Alttaki çerçeve, tam bıraktığı yerden çalışmaya devam eder.

Özyinelemede aynı metodun birden çok çerçevesi aynı anda yaşar. `faktoriyel(4)`, `faktoriyel(3)`'ü çağırdığında ilk çerçeve silinmez; çarpmayı yapabilmek için alttan gelecek değeri bekler. Her çerçevenin kendi `n`'si vardır. Yığında dört ayrı `n` bulunabilir: biri 4, biri 3, biri 2, biri 1. Bunlar aynı adı taşıyan ama birbirinden bağımsız değişkenlerdir. Bir çerçevedeki `n`'nin değişmesi ötekileri etkilemez.



Şekil 3.5.2: `faktoriyel(4)` çağrısının en derin anı. Yığının tepesinde `faktoriyel(1)` çalışır; altındaki üç çerçeve çarpımı tamamlamak için bekler, en altta `main` sonucu bekler. Çağrıda çerçeve eklenir (sol ok), dönüşte silinir (sağ ok).

Şekil 3.5.2, `main`'in `faktoriyel(4)`'ü çağırdığı programın en derin anını gösterir. O anda yığında beş çerçeve vardır: `main` ve dört `faktoriyel` çerçevesi. Yalnız tepedeki `faktoriyel(1)` çalışır. Alttaki her çerçeve bir çarpımın yarısında durmuş, eksik çarpanı bekler: `4 * ?`, `3 * ?`, `2 * ?`.

Yığının boyutu sınırlıdır. Java'da her iş parçacığının yığını, program başlarken belirlenen sabit bir bellek alanıdır. Çok derin bir özyineleme bu alanı doldurur ve program `StackOverflowError` ile durur. Bu sınır, özyinelemenin en önemli pratik kısıtıdır.

Sık yapılan hata

Özyinelemeli çağrıda değişkenin "üzerine yazıldığını" sanmak. `faktoriyel(3)` çağrıldığında `faktoriyel(4)` 'ün `n` 'si 3 olmaz. Yeni çerçevede yeni bir `n` açılır ve eski çerçevedeki `n = 4` olduğu gibi bekler. Dönüşte çarpmanın `4 * 6` diye yapılabilmesinin nedeni budur.

3.5.2 Faktöriyel ve toplam

Faktöriyel ve 1'den n'ye toplam, özyinelemenin en basit iki örneğidir. İkisi de aynı kalıbı izler: argümanı bir azaltıp kendini çağırır, gelen sonuca kendi katkısını ekler. Bu kalıp iyi anlaşıldığında sonraki örneklerin çoğu onun çeşitlemesi olarak okunur.

3.5.2.1 Faktöriyel metodu

n'nin faktöriyeli 1'den n'ye kadar sayıların çarpımıdır: n! = 1 × 2 × … × n. Tanım gereği 0! = 1'dir. Aynı çarpım özyinelemeli olarak n! = n × (n − 1)! diye de yazılır. Java'daki karşılığı şöyledir:

```
int faktoriyel(int n) {
    if (n == 0 || n == 1) {
        return 1;   // taban durum
    } else {
        return n * faktoriyel(n - 1); // özyineleme
    }
}
```

Birinci satır metodun başlığıdır: `int` bir parametre alır, `int` bir değer döndürür. İkinci ve üçüncü satırlar taban durumdur. `n` 0 ya da 1 ise cevap 1'dir ve metot kendini çağırmadan döner. Beşinci satır özyinelemeli adımdır: n!, (n − 1)!'in n ile çarpımıdır. `faktoriyel(n - 1)` ifadesi hesaplanmadan çarpma yapılamaz; bu yüzden metot önce kendini çağırır, cevabı bekler, sonra çarpar ve döner.

Metodun her zaman sona ereceği gösterilebilir. Her çağrıda `n` 1 azalır. Başlangıçta `n ≥ 0` ise bu azalma eninde sonunda 1'e ya da 0'a varır ve taban durum çalışır. Bu ilerleme garantisi her özyinelemeli metotta aranması gereken özelliktir. `faktoriyel(5)` çağrısı 120, `faktoriyel(0)` çağrısı 1 döndürür.

n	`faktoriyel(n)`	Gerçek n!	Doğru mu?
0	1	1	evet
5	120	120	evet
12	479001600	479.001.600	evet
13	1932053504	6.227.020.800	hayır (taşma)

n	`faktoriyel(n)`	Gerçek n!	Doğru mu?
17	−288522240	355.687.428.096.000	hayır (negatif bile)

Tablo 3.5.1: `int` döndüren `faktoriyel`'in bazı değerleri (JDK 21).

Kodun matematiksel olarak doğru olması, her sonucun doğru çıkacağı anlamına gelmez. `int` en çok 2.147.483.647'yi tutabilir. 12! = 479.001.600 bu sınırın altındadır, 13! = 6.227.020.800 ise üstündedir. Java taşmayı haber vermez: çarpımın sığmayan üst bitleri atılır ve geriye anlamsız bir sayı kalır (Tablo 3.5.1). `faktoriyel(13)` 1932053504 döndürür; 17! için sonuç negatiftir. Dönüş tipi `long` yapılırsa 20! = 2.432.902.008.176.640.000'e kadar doğru sonuç alınır; 21! `long`'a da sığmaz. Daha büyük faktöriyeller için `java.math.BigInteger` kullanılır.

Sık yapılan hata

Taşmayı derleyicinin ya da çalışma zamanının yakalayacağını sanmak. `int` ve `long` çarpımlarında Java hiçbir uyarı ya da istisna üretmez. Sonucun aralığı önceden kestirilmeli ve uygun tip seçilmelidir. Taşmanın istisnayla bildirilmesi istenirse `Math.multiplyExact` kullanılır; bu metot taşmada `ArithmeticException` fırlatır.

3.5.2.2 faktoriyel(4) çağrısının açılımı

Özyinelemeli bir çağrının sonucunu elle bulmanın en kısa yolu, çağrıyı yerine koyarak açmaktır. Her `faktoriyel(k)` yazılan yere `k * faktoriyel(k - 1)` konur ve taban duruma varılınca geri hesaplanır:

```
faktoriyel(4)
  = 4 * faktoriyel(3)
  = 4 * (3 * faktoriyel(2))
  = 4 * (3 * (2 * faktoriyel(1)))
  = 4 * (3 * (2 * 1))
  = 4 * (3 * 2)
  = 4 * 6
  = 24
```

Açılım üç evreden oluşur. İlk dört satır **iniştir**. Her çağrı yeni bir çağrı açar ve çarpımlar bekletilir, çünkü sağ taraftaki değer henüz bilinmez. Beşinci satırda **dibe**, yani taban duruma varılır. `faktoriyel(1)` doğrudan 1 döndürür. Kod `n == 1`'de durduğu için `faktoriyel(0)` hiç çağrılmaz. Son üç satır **çıkıştır**. Bekleyen çarpımlar içten dışa doğru tamamlanır: önce 2 × 1 = 2, sonra 3 × 2 = 6, en son 4 × 6 = 24.

Parantezlerin iç içe girmesi, çağrı yığınının yazıyla gösterilmiş hâlidir. En içteki parantez yığının tepesidir ve ilk o tamamlanır. Her kapanan parantez, yığından silinen bir çerçeveye karşılık gelir. Çarpmaların 4 × 3 × 2 × 1 sırasıyla değil, ters sırayla yapıldığına dikkat edilmelidir. Çarpmada sıra so-

nucu değiştirmez; ama ekrana yazdırma gibi sırası önemli işlerde iniş ile çıkış arasındaki fark belirleyici olur.

3.5.2.3 Çağrı yığınında izleme

Aynı hesap, tam bir programın çalışması olarak da izlenebilir. Aşağıdaki programda `main`, `faktoriyel(4)`'ü çağırıp sonucu yazdırır:

```
static int faktoriyel(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * faktoriyel(n - 1);
    }
}

public static void main(String[] args) {
    System.out.println(faktoriyel(4));
}
```

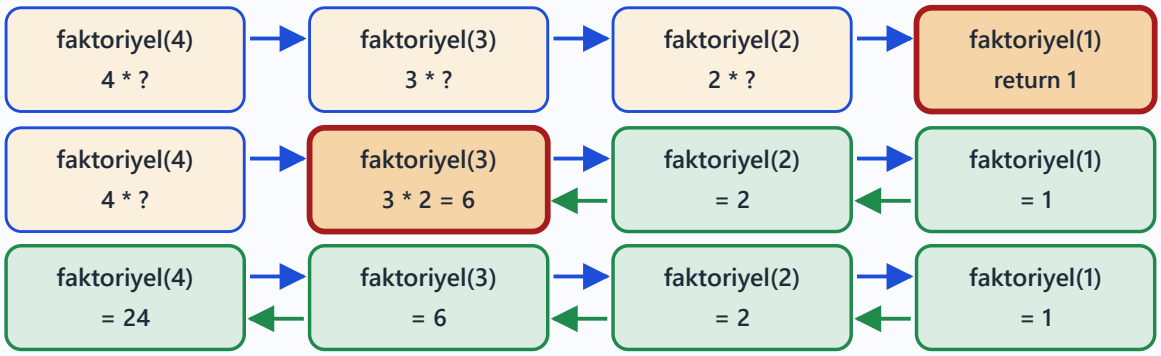
```
24
```

Tablo 3.5.2, programın her adımında çağrı yığınını aşağıdan yukarıya doğru verir. “Bekleyen iş” sütunu, çerçevenin dönüşte yapacağı hesaptır.

Adım	Olay	Yığın (alttan üste)	Bekleyen iş
1	`main`, `faktoriyel(4)`'ü çağırır	main	sonucu yazdırmak
2	`n = 4`: koşul yanlış, `faktoriyel(3)` çağrılır	main, f(4)	`4 * ?`
3	`n = 3`: koşul yanlış, `faktoriyel(2)` çağrılır	main, f(4), f(3)	`3 * ?`
4	`n = 2`: koşul yanlış, `faktoriyel(1)` çağrılır	main, f(4), f(3), f(2)	`2 * ?`
5	`n = 1`: taban durum, 1 döner	main, f(4), f(3), f(2), f(1)	—
6	`f(2)` 1'i alır: `2 * 1 = 2` döner	main, f(4), f(3), f(2)	—
7	`f(3)` 2'yi alır: `3 * 2 = 6` döner	main, f(4), f(3)	—
8	`f(4)` 6'yı alır: `4 * 6 = 24` döner	main, f(4)	—
9	`main` 24'ü yazdırır	main	—

Tablo 3.5.2: `faktoriyel(4)` çağrısının adım adım izlenmesi.

Dört `faktoriyel` çağrısı yapılır. En derin anda, yani beşinci adımda yığında beş çerçeve vardır: `main` ile dört `faktoriyel`. Şekil 3.5.3 aynı izlemenin üç anını çağrı zinciri olarak gösterir. Üstte inişin dibi, altta bütün dönüşler tamamlandıktan sonraki durum vardır.



Şekil 3.5.3: `faktoriyel(4)` çağrı zinciri. Üstte: taban duruma varılmıştır, üç çerçeve `4 * ?`, `3 * ?`, `2 * ?` çarpımlarını bekler. Ortada: `faktoriyel(2)` 2 döndürmüştür, `faktoriyel(3)` şimdi `3 * 2 = 6`'yı hesaplar. Altta: bütün çerçeveler değer döndürmüştür; `faktoriyel(4) = 24`.

Bir özyinelemeli metodun zaman ve bellek maliyeti bu izlemeden okunur. `faktoriyel(n)` tam n çağrı yapar ve her çağrıda bir karşılaştırma ile bir çarpma vardır. Çalışma süresi n ile doğru orantılıdır, yani O(n)'dir. Yığında aynı anda en çok n çerçeve bulunur; bellek de O(n)'dir. Aynı işi yapan döngü de O(n) sürer, ama tek bir çerçeveyle çalışır ve belleği O(1)'dir. Derinlik büyüdükçe bu fark önem kazanır.

3.5.2.4 1'den n'ye toplam

1'den n'ye kadar sayıların toplamı da aynı kalıpla yazılır: toplam(n) = n + toplam(n − 1) ve toplam(1) = 1. Faktöriyelden tek farkı, çarpma yerine toplama yapılması ve taban durumun yalnız `n == 1` olmasıdır.

```
int toplam(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n + toplam(n - 1);
    }
}
```

```
toplam(5) = 5 + toplam(4)
          = 5 + (4 + (3 + (2 + toplam(1))))
          = 5 + (4 + (3 + (2 + 1)))
          = 15
```

Açılımda her çağrı kendi sayısını bekleyen bir toplama olarak dışarıda bırakır. Dibe varıldığında 2 + 1 = 3, sonra 3 + 3 = 6, 4 + 6 = 10 ve 5 + 10 = 15 hesaplanır. `toplam(100)` aynı yolla 5050'yi verir. Bu sonuç, Gauss'un n(n + 1)/2 formülüyle de doğrulanır: 100 · 101 / 2 = 5050.

Bu metodun bir zayıf noktası vardır. Taban durum yalnız `n == 1`'dir. `toplam(0)` ya da negatif bir sayıyla çağrıldığında `n` hiç 1 olmaz: 0, −1, −2, … diye sonsuza kadar azalır. Sonuç sonsuz özyinelemedir ve program `StackOverflowError` ile durur. Güvenli sürümde taban durum `if (n <= 0) return 0;` olur. Bu sürüm geçersiz girdileri de yakalar ve `toplam(1)` için 1 + 0 = 1'i doğru verir.

Örnek 3.5.1

Güvenli taban durumlu `toplam`, `toplam(3)` için şöyle açılır:

1. `toplam(3)` : 3 ≤ 0 yanlış; `3 + toplam(2)` bekler.
2. `toplam(2)` : `2 + toplam(1)` bekler.
3. `toplam(1)` : `1 + toplam(0)` bekler.
4. `toplam(0)` : 0 ≤ 0 doğru; taban durum, 0 döner.
5. Dönüşte 1 + 0 = 1, 2 + 1 = 3, 3 + 3 = 6 hesaplanır. Sonuç 6'dır.

Eski sürüme göre bir çağrı fazladır (dört çağrı), ama hiçbir girdide sonsuz özyinelemeye girmez. `toplam(-5)` doğrudan 0 döndürür.

3.5.3 Seriler

Birçok sayı serisi, her terimi önceki terimlerden hesaplayan bir kuralla tanımlanır. Böyle bir kurala **yineleme bağıntısı** (*recurrence relation*) denir. Yineleme bağıntısı, özyinelemeli bir metoda neredeyse harfi harfine çevrilir: başlangıç değerleri taban durum, bağıntının kendisi özyinelemeli adım olur. Bu kesimde dört seri ele alınır. Fibonacci ve Lucas serilerinde metot kendini iki kez çağırır. Catalan ve Bell serilerinde ise çağrılar bir döngünün içinde yapılır.

3.5.3.1 Fibonacci serisi

Tanım 3.5.4

Fibonacci serisi 0 ile başlar, ardından 1 gelir: F(0) = 0, F(1) = 1. Sonraki her terim önceki iki terimin toplamıdır: F(n) = F(n − 1) + F(n − 2).

Seri, adını İtalyan matematikçi Leonardo Fibonacci'den alır. Fibonacci 1202'de yazdığı *Liber Abaci* kitabında tavşanların çoğalmasıyla ilgili bir problem sorar. Her çift tavşan iki aylık olunca her ay yeni bir çift doğurur; bir yılın sonunda kaç çift tavşan olur? Ay ay çift sayısı bu seriyi izler. Serinin ilk on üç terimi Tablo 3.5.3'te verilmiştir.

n	0	1	2	3	4	5	6	7	8	9	10	11	12

F(n)	0	1	1	2	3	5	8	13	21	34	55	89	144

Tablo 3.5.3: Fibonacci serisinin ilk terimleri.

Ardışık iki terimin oranı, n büyüdükçe altın orana (≈ 1,618) yaklaşır: 144 / 89 ≈ 1,6180. Tanımı Java'ya çevirmek kolaydır:

```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    } else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Taban durum `n <= 1`'dir ve iki başlangıç değerini tek satırda karşılar: `n` 0 ise 0, 1 ise 1 döner. Yani bu durumda cevap `n`'nin kendisidir. Özyinelemeli adımda metot kendini **iki kez** çağırır. Faktöriyeldeki çağrılar düz bir zincir oluştururken burada her çağrı iki dala ayrılır ve çağrılar bir ağaç oluşturur. Bu tür özyinelemeye **ağaç özyinelemesi** (*tree recursion*) denir.

3.5.3.2 fibonacci(4) çağrı ağacı

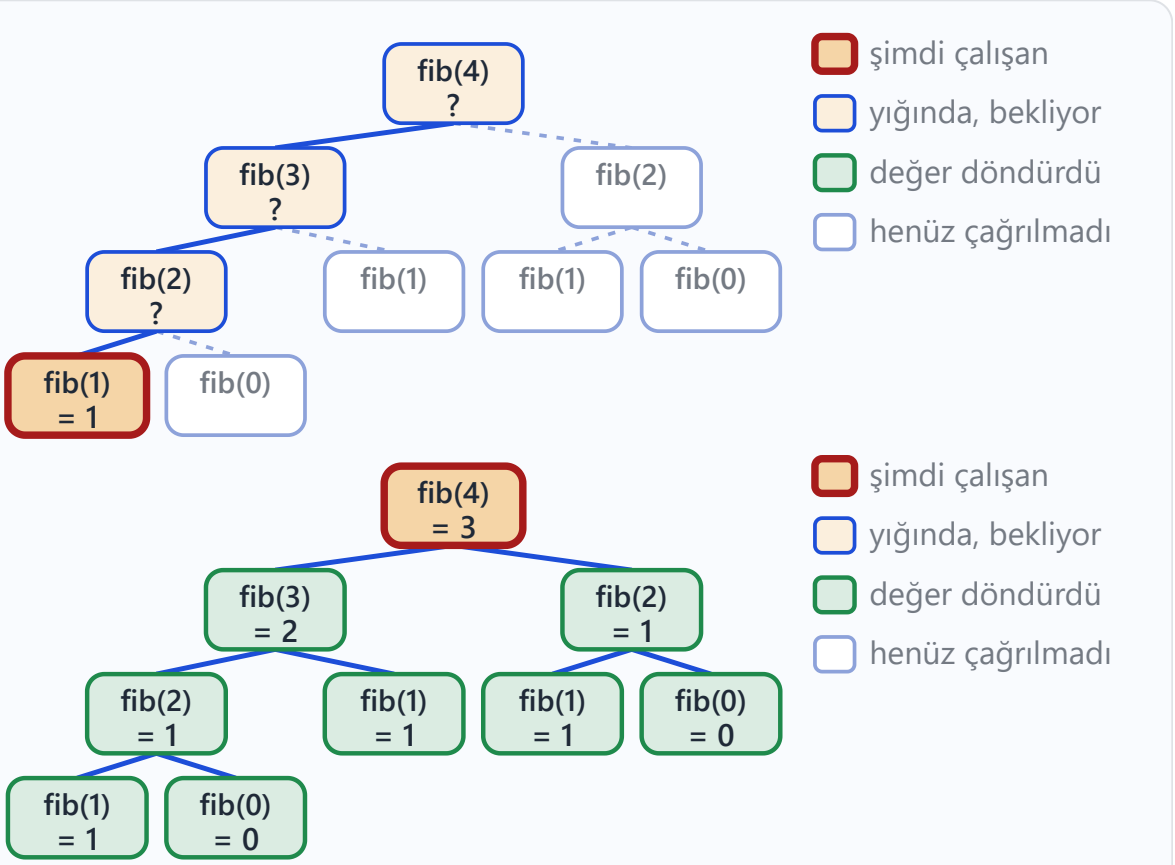
Ağaç özyinelemesinde çağrıların sırası önemlidir. Java bir ifadeyi soldan sağa hesaplar. `fibonacci(n - 1) + fibonacci(n - 2)` ifadesinde önce sol terim tamamen hesaplanır. Sağ terim için çağrı, sol terimin bütün alt çağrıları bittikten sonra yapılır. Bu yüzden ağaç, en soldaki dalın dibine kadar inilerek, ardından sağa doğru ilerlenerek gezilir.

Sıra	Çağrı	Derinlik	Olay	Döner
1	`fib(4)`	1	sol terim için `fib(3)`	3
2	`fib(3)`	2	sol terim için `fib(2)`	2
3	`fib(2)`	3	sol terim için `fib(1)`	1
4	`fib(1)`	4	taban durum	1
5	`fib(0)`	4	taban durum; `fib(2) = 1 + 0`	0
6	`fib(1)`	3	taban durum; `fib(3) = 1 + 1`	1
7	`fib(2)`	2	yeniden hesaplanır; sol terim için `fib(1)`	1
8	`fib(1)`	3	taban durum	1
9	`fib(0)`	3	taban durum; `fib(2) = 1 + 0`, `fib(4) = 2 + 1`	0

Tablo 3.5.4: `fibonacci(4)` çağrılarının sırası. Derinlik, çağrının yığındaki `fibonacci` çerçevesi sayısıdır.

Tablo 3.5.4, dokuz çağrıyı yapılış sırasıyla verir. Yığın hiçbir anda dört `fibonacci` çerçevesinden derine inmez, çünkü en uzun dal 4 → 3 → 2 → 1'dir. Ama toplam çağrı sayısı 9'dur ve `fib(2)` iki kez, `fib(1)` üç kez, `fib(0)` iki kez hesaplanmıştır. Şekil 3.5.4 ağacın iki anını gösterir: üstte inişin ilk kez

dibe vardığı an, altta bütün çağrıların değer döndürdüğü son durum.



Şekil 3.5.4: `fibonacci(4)` çağrı ağacı. Üstte dördüncü çağrı: sol kol `fib(1)`'e kadar inmiştir; üstündeki üç çerçeve yığında bekler, sağdaki düğümler henüz çağrılmamıştır. Altta son durum: dokuz düğümün hepsi değer döndürmüştür ve `fib(4) = 3`'tür.

Tekrarlanan hesaplama küçük n için zararsız görünür. Ama ağacın büyüklüğü n ile hızla artar. `fibonacci(n)`'in yaptığı çağrı sayısı Ç(n) = 1 + Ç(n − 1) + Ç(n − 2) bağıntısını sağlar ve Ç(n) = 2F(n + 1) − 1'dir. `fibonacci(5)` 15, `fibonacci(10)` 177, `fibonacci(40)` 331.160.281 çağrı yapar. Ağaç her düzeyde yaklaşık 1,6 kat genişler ve aynı alt ağaçlar defalarca kurulur.

3.5.3.3 Lucas serisi

Lucas serisi, Fibonacci ile aynı bağıntıyı farklı başlangıç değerleriyle kullanır. Fransız matematikçi Édouard Lucas'nın adını taşır.

Tanım 3.5.5

Lucas serisi 2 ile başlar, ardından 1 gelir: L(0) = 2, L(1) = 1. Sonraki terimler Fibonacci'deki gibi hesaplanır: L(n) = L(n − 1) + L(n − 2).

```
int lucas(int n) {
    if (n == 0) {
        return 2;
    } else if (n == 1) {
        return 1;
    } else {
        return lucas(n - 1) + lucas(n - 2);
    }
}
```

Fibonacci'de iki başlangıç değeri `return n;` ile tek satırda karşılanabiliyordu, çünkü F(0) = 0 ve F(1) = 1'dir. Lucas'ta L(0) = 2 olduğu için bu kısayol yoktur ve iki ayrı taban durum gerekir. `else if` zinciri önce `n == 0`'ı, sonra `n == 1`'i sınar; ikisi de değilse özyinelemeli adım çalışır.

n	0	1	2	3	4	5	6	7	8	9	10
L(n)	2	1	3	4	7	11	18	29	47	76	123

Tablo 3.5.5: Lucas serisinin ilk terimleri.

İki seri birbirine bağlıdır: L(n) = F(n − 1) + F(n + 1). Örneğin L(5) = F(4) + F(6) = 3 + 8 = 11'dir. Lucas'ın çağrı ağacı Fibonacci'ninkiyle aynı biçimdedir, çünkü taban durumlar yine n = 0 ve n = 1'dedir. Bu yüzden `lucas(5)` de 15 çağrı yapar.

3.5.3.4 Catalan serisi

Catalan sayıları birçok sayma probleminin cevabıdır. Seri, Belçikalı matematikçi Eugène Catalan'ın adını taşır.

> **Tanım 3.5.6**
>
> **Catalan serisi** 1 ile başlar: C(0) = 1, C(1) = 1. Kapalı formülü C(n) = (2n)! / ((n + 1)! · n!)'dir. Özyinelemeli formülü ise C(n) = Σ C(i) · C(n − 1 − i)'dir; toplam i = 0'dan n − 1'e kadar alınır.

Özyinelemeli formülde bir terim, kendisinden küçük *bütün* terimlerden kurulur. Java kodu bu toplamı bir `for` döngüsüyle hesaplar ve özyinelemeli çağrıları döngünün içinde yapar:

```
long catalan(int n) {
    if (n <= 1) {
        return 1;
    }
    long sonuc = 0;
    for (int i = 0; i < n; i++) {
        sonuc += catalan(i) * catalan(n - i - 1);
    }
    return sonuc;
}
```

Dönüş tipi `long`'dur, çünkü Catalan sayıları hızla büyür. `sonuc` 0'dan başlar; toplamanın etkisiz elemanı 0'dır. Döngü `i`'yi 0'dan `n - 1`'e kadar ilerletir ve her turda `catalan(i) * catalan(n - i - 1)` çarpımını toplama ekler. Örneğin C(3) şöyle hesaplanır:

> C(3) = C(0)·C(2) + C(1)·C(1) + C(2)·C(0) = 1·2 + 1·1 + 2·1 = 5

Kapalı formül de aynı sonucu verir: 6! / (4! · 3!) = 720 / 144 = 5. Serinin ilk dokuz terimi 1, 1, 2, 5, 14, 42, 132, 429, 1430'dur; kod çalıştırıldığında da bu değerler çıkar.

Catalan sayıları ne sayar? En bilinen cevap, n çift parantezin doğru dizilişleridir. Doğru dizilişte her açılan parantez daha sonra kapanır ve hiçbir noktada kapanan parantez sayısı açılanı geçmez. n = 3 için beş diziliş vardır: `((()))`, `(()())`, `(())()`, `()(())`, `()()()`. Aynı sayı, n + 1 sayının çarpımının kaç farklı sırayla parantezlenebileceğini ve n düğümlü kaç farklı ikili ağaç bulunduğunu da verir.

Bu metodun maliyeti Fibonacci'den de hızlı büyür. Her `catalan(n)`, küçük bütün terimleri iki kez çağırır ve hiçbir sonucu saklamaz. Ölçülen çağrı sayıları n = 2, 3, 4, 5, 6 için sırasıyla 5, 15, 45, 135, 405'tir; n 1 artınca çağrı sayısı üç katına çıkar. Bu tür bir metot küçük n için kısa ve açıktır, ama büyük n için belleklemeye ihtiyaç duyar.

3.5.3.5 Bell serisi

Tanım 3.5.7

Bell serisi 1 ile başlar: B(0) = 1, B(1) = 1. Genel formül B(n) = Σ C(n − 1, k) · B(k)'dir; toplam k = 0'dan n − 1'e kadar alınır. Burada C(n, k), n elemandan k eleman seçmenin yolu sayısı, yani kombinasyondur.

```
int bell(int n) {
    if (n == 0) { return 1; }
    int sayi = 0;
    for (int k = 0; k < n; k++) {
        sayi += katsayi(n - 1, k) * bell(k);
    }
    return sayi;
}

int katsayi(int n, int k) {       // C(n, k)
    return faktoriyel(n) / (faktoriyel(k) * faktoriyel(n - k));
}
```

Kod iki metottan oluşur. `bell`, Catalan'daki gibi bir döngü içinde kendini çağırır ve her terimi kombinasyon katsayısıyla çarpıp toplar. `katsayi` ise C(n, k) = n! / (k! · (n − k)!) formülünü `faktoriyel` metoduyla hesaplar. Burada bir metodun hem kendini hem de başka bir özyinelemeli metodu çağırdığı görülür.

Bell sayıları, n elemanlı bir kümenin kaç farklı biçimde gruplara ayrılabileceğini sayar. {a, b, c} kümesi için beş yol vardır: tek grup `abc`; bir eleman ayrı, ikisi birlikte olan üç yol `a|bc`, `b|ac`, `c|ab`; her eleman ayrı `a|b|c`. Gerçekten de B(3) = 5'tir. Serinin ilk sekiz terimi 1, 1, 2, 5, 15, 52, 203, 877'dir.

Bu kodda taşma iki ayrı yerden gelir. `katsayi`, `int` döndüren `faktoriyel`'i kullanır ve 13! `int`'e sığmaz. `bell(14)`, `katsayi(13, k)`'yi çağırdığı için yanlışlaşır. `bell(13)` = 27.644.437 doğru çıkar, `bell(14)` ise 77.914.768 verir; doğru değer 190.899.322'dir. Bu yüzden kod yalnız n ≤ 13 için doğrudur. `faktoriyel` düzeltilse bile sonuç `int` olduğu için B(16) = 10.480.142.147 yine sığmazdı.

> **Sık yapılan hata**
>
> Kombinasyonu faktöriyellerin bölümüyle hesaplamak. C(n, k) küçük bir sayı olsa bile ara sonuçtaki n! çok büyüktür ve taşar. Daha güvenli yol, C(n, k) = C(n − 1, k − 1) + C(n − 1, k) bağıntısını (Pascal üçgeni) ya da çarpıp hemen bölen bir döngüyü kullanmaktır.

3.5.4 Özyinelemeli metot yazma

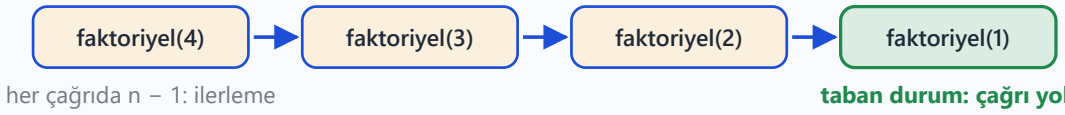
Özyinelemeli bir metodu okumak, yazmaktan kolaydır. Yazarken iki kurala uymak ve probleme belirli bir bakış açısıyla yaklaşmak gerekir. Önce iki kural, ardından bu kurallar çiğnendiğinde ortaya çıkan hata, en son da bir problemi özyinelemeli düşünmenin adımları işlenir.

3.5.4.1 İki temel kural

Teorem 3.5.1 (Sonlanma kuralları)

Kural 1, taban durum: metotta, kendini çağırmadan sonlanan en az bir taban durum bulunur. **Kural 2, ilerleme:** her özyinelemeli çağrı taban duruma doğru ilerler; parametre her çağrıda azalır (ya da artar), dizgi kısalır, aralık daralır. İki kural birlikte sağlanırsa özyineleme her girdi için sonlu sayıda çağrıdan sonra durur.

İlerlemenin ne demek olduğu, "taban duruma uzaklık" diye bir ölçüyle somutlaştırılır. Faktöriyelde bu ölçü `n`'nin kendisidir: her çağrıda 1 azalır ve 1'e varınca durur (Şekil 3.5.5). Rakamlar toplamında ölçü basamak sayısıdır, palindromda dizginin uzunluğu, Öklid algoritmasında ikinci parametre. Ölçü bir doğal sayıysa ve her çağrıda kesin olarak küçülüyorsa sonsuza kadar küçülemez; sonunda taban duruma varılır.



Şekil 3.5.5: `faktoriyel(4)` her çağrıda `n`'yi bir azaltır: 4, 3, 2, 1. `n = 1` taban durumdur ve yeni çağrı yapılmaz.

En sık iki hata bu kuralların çiğnenmesidir. **Sonsuz özyinelemede** taban durum ya hiç yoktur ya da ona hiç ulaşılamaz. **Yetersiz ilerlemede** parametre değişmez; örneğin `return n * faktoriyel(n);` yazılmıştır. İki hatanın sonucu da aynıdır: `StackOverflowError`.

Kurallara üç alışkanlık eklenir. Birincisi, her özyinelemeli çağrı problemin daha küçük bir alt kümesine odaklanır. İkincisi, taban durum her zaman özyinelemeli çağrıdan önce sınanır; çağrı önce yapılırsa sınama hiç sırası gelmeden yığın dolar. Üçüncüsü, olası bütün girdiler hesaba katılır. Negatif sayı, 0, boş dizgi ya da `Integer.MIN_VALUE` taban duruma ulaşıyor mu, sorusu sorulur.

3.5.4.2 Sonsuz özyineleme

Sonsuz özyinelemenin nasıl göründüğünü tam bir programla görmek öğreticidir. Aşağıdaki program, taban durumu yalnız `n == 1` olan `toplam`'ı 0 ile çağırır:

```
public class Toplam {
    static int toplam(int n) {
        if (n == 1) {
            return 1;
        } else {
            return n + toplam(n - 1);   // satır 6
        }
    }
    public static void main(String[] args) {
        System.out.println(toplam(0));
    }
}
```

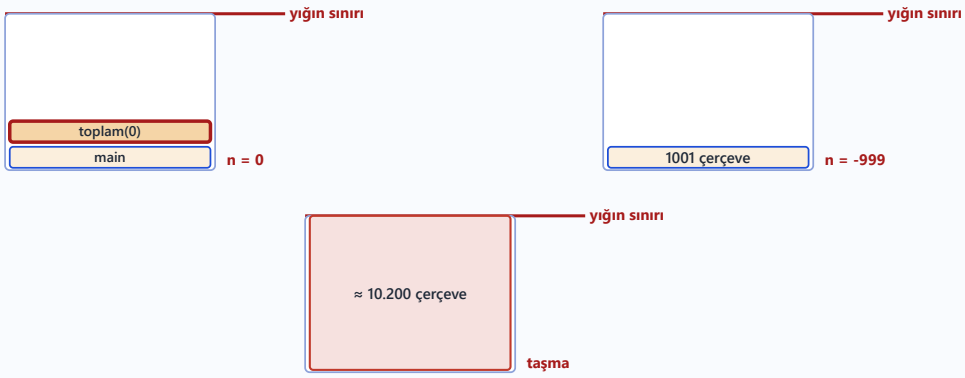
Program derlenir; derleyici hiçbir uyarı vermez. Çalıştırıldığında ise bir süre sonra şu iletiyle durur:

```
Exception in thread "main" java.lang.StackOverflowError
        at Toplam.toplam(Toplam.java:6)
        at Toplam.toplam(Toplam.java:6)
        at Toplam.toplam(Toplam.java:6)
        ...
```

Tanım 3.5.8

Yığın taşması (*stack overflow*), çok derin ya da sonsuz özyinelemede çağrı yığınının dolmasıdır. Java sanal makinesi (*Java Virtual Machine*, JVM) bu durumda `java.lang.StackOverflowError` fırlatır ve yakalanmazsa program durur.

Ne olduğunu adım adım izleyelim. `main`, `toplam(0)`'ı çağırır. `n = 0` için `0 == 1` yanlıştır ve `0 + toplam(-1)` hesaplanmak üzere yeni bir çağrı yapılır. `n = -1` de 1 değildir; `-1 + toplam(-2)` bekler. `n` her çağrıda taban durumdan biraz daha uzaklaşır. Her çağrı yığına yeni bir çerçeve ekler ve hiçbiri dönemez, çünkü hepsi bir sonrakinin sonucunu bekler. Bin çağrıdan sonra yığında 1001 çerçeve vardır ve hâlâ hata yoktur. Sonunda yığın için ayrılan bellek dolar (Şekil 3.5.6).



Şekil 3.5.6: `toplam(0)` çağrısında yığının dolması. Solda iki çerçeve, ortada 1001 çerçeve, sağda yığın sınırına varılmış ve taşma olmuştur. Şekildeki ölçümde sınır ≈ 10.200 çağrıda aşılmıştır; sayı JVM ayarına göre değişir.

Taşmanın kaç çağrıdan sonra olduğu sabit değildir. Yığının boyutuna, çerçevenin büyüklüğüne ve JVM'in o anda kodu yorumlayıp yorumlamadığına bağlıdır. JDK 21 ile Windows'ta varsayılan ayarlarla yapılan beş ölçümde taşma 18.487 ile 18.822 çağrı arasında oldu. Yığın `-Xss512k` seçeneğiyle yarıya indirildiğinde derinlik 6.235'e düştü. Aynı JVM içinde ikinci kez denendiğinde, derlenmiş kodun çerçeveleri küçüldüğü için 41.480 çağrıya kadar inilebildi. Sonuç olarak varsayılan yığınla birkaç on bin çağrı derinliği güvenli sınırın ötesidir.

Bu hatanın iki önemli özelliği vardır. Birincisi, derleyici onu yakalamaz; sorun ancak çalışma zamanında ortaya çıkar. Derleyici her yolun bir `return` ile bittiğini görür ve metodu geçerli sayar; değerlerin taban duruma ulaşıp ulaşmayacağını bilemez. İkincisi, hata iletisinin altındaki **yığın izi** (*stack trace*) tanıdık bir görüntü verir: aynı satır (`Toplam.java:6`) alt alta yüzlerce kez tekrarlanır. JVM yığın izinin en çok 1024 satırını yazdırır. Aynı satırın tekrar tekrar görünmesi sonsuz özyinelemenin işaretidir.

Çözüm üç yoldan biridir. Taban durum genişletilir (`n <= 1` ya da `n <= 0`). Girdi metodun başında denetlenir ve geçersizse bir istisna fırlatılır. Problem doğası gereği çok derinse, örneğin bir milyon öğeli bir listede geziniliyorsa, özyineleme yerine döngü kullanılır.

> **Sık yapılan hata**
>
> `StackOverflowError`'ı `try-catch` ile yakalayıp programı sürdürmek. `StackOverflowError` bir `Exception` değil, `Error`'dır; programın düzeltilmesi gereken bir hatası olduğunu bildirir. Yakalamak taşmayı önlemez, yalnız belirtiyi gizler. Doğru çözüm taban durumu ya da ilerlemeyi düzeltmektir.

3.5.4.3 Özyinelemeli düşünme

Özyinelemeli bir metot yazarken dört adım izlenir:

1. **Küçültme:** problem, kendisinin daha küçük bir kopyası cinsinden ifade edilir.

2. **Taban durum:** cevabı doğrudan bilinen en küçük girdi bulunur.
3. **Güven:** küçük kopyanın doğru çözüldüğü varsayılır; çağrının içi zihinde açılmaz.
4. **Birleştirme:** küçük cevaptan büyük cevap kurulur.

Üçüncü adım en zor olanıdır ve **inanç sıçraması** (*leap of faith*) diye anılır. Yeni başlayanlar bir özyinelemeli çağrı gördüklerinde onun içine girip bütün zinciri kafalarında açmaya çalışır. Küçük örneklerde bu işe yarar, ama Hanoi kuleleri gibi örneklerde hızla içinden çıkılmaz hâle gelir. Güven adımı şunu söyler: metodun n − 1 için doğru çalıştığı varsayılarak n için de doğru çalıştığı gösterilebiliyorsa, metot her n için doğrudur. Bu, matematikteki tümevarımın ta kendisidir. Taban durum tümevarımın başlangıç adımı, özyinelemeli adım ise tümevarım adımıdır.

Örnek 3.5.2

Bir sayının basamak sayısı dört adımla bulunur. *Küçültme:* 48729'un basamak sayısı, 4872'nin basamak sayısından 1 fazladır. 4872, 48729'dan son basamak atılarak, yani `48729 / 10` tam sayı bölmesiyle elde edilir. *Taban durum:* tek basamaklı bir sayının (`n < 10`) basamak sayısı 1'dir. *Güven:* `basamakSayisi(4872)` 'nin 4 döndüreceği kabul edilir. *Birleştirme:* sonuç 1 + 4 = 5'tir.

```
int basamakSayisi(int n) {
    if (n < 10) {                       // taban durum
        return 1;
    }
    return 1 + basamakSayisi(n / 10);   // küçült + birleştir
}
```

`basamakSayisi(48729)` 5, `basamakSayisi(7)` doğrudan 1 döndürür. Çağrılar 48729 → 4872 → 487 → 48 → 4 diye ilerler; beş çağrının sonuncusu taban durumdur. Kodda `else` yoktur. Taban durum `return` ile bittiği için sonraki satıra yalnız `n >= 10` iken gelinir. Bu yazım, özyinelemeli metotlarda sık görülür: önce taban durum elenir, kalan gövde genel durumu işler.

Metodun negatif sayılarda yanlış sonuç verdiğine dikkat edilmelidir. −45 için `-45 < 10` doğrudur ve sonuç 1 olur; bu yanlıştır. Negatif girdiler önemliyse metodun başına bir işaret düzeltmesi eklenir. Bu düzeltmenin nasıl yapıldığı rakamlar toplamı örneğinde görülür.

3.5.4.4 İnişte ve dönüşte yapılan iş

Özyinelemeli bir metotta iş iki yerde yapılabilir: kendini çağırmadan önce ya da çağrı döndükten sonra. Bu fark, özellikle ekrana yazdırma gibi sırası görünen işlerde çok belirgindir. Aşağıdaki iki metot yalnız iki satırın sırasıyla ayrılır:

```
static void yaz(int n) {
    if (n == 0) return;
    System.out.print(n + " ");
    yaz(n - 1);
}
static void yaz2(int n) {
    if (n == 0) return;
    yaz2(n - 1);
    System.out.print(n + " ");
}
static int gizem(int a, int b) {
    if (b == 0) return 0;
    return a + gizem(a, b - 1);
}
```

`yaz(3)`, sayıyı yazdırıp sonra kendini çağırır. Yazdırma inişte yapılır ve çıktı `3 2 1` olur. `yaz2(3)` ise önce kendini çağırır, yazdırmayı dönüşe bırakır. `yaz2(3)`, `yaz2(2)`'yi, o da `yaz2(1)`'i, o da `yaz2(0)`'ı çağırır. `yaz2(0)` hiçbir şey yazmadan döner. Ardından en derindeki çerçeve olan `yaz2(1)` 1'i yazar, sonra `yaz2(2)` 2'yi, en son `yaz2(3)` 3'ü. Çıktı `1 2 3`'tür.

An	Yığının tepesi	`yaz` ekranı	`yaz2` ekranı
iniş	n = 3	`3`	
iniş	n = 2	`3 2`	
iniş	n = 1	`3 2 1`	
dip	n = 0	`3 2 1`	
dönüş	n = 1	`3 2 1`	`1`
dönüş	n = 2	`3 2 1`	`1 2`
dönüş	n = 3	`3 2 1`	`1 2 3`

Tablo 3.5.6: `yaz(3)` ve `yaz2(3)` çağrılarında yazdırma anları.

Aynı kodda üçüncü bir metot, `gizem`, bulunur. `gizem(4, 3)` şöyle açılır: 4 + `gizem(4, 2)` = 4 + 4 + `gizem(4, 1)` = 4 + 4 + 4 + `gizem(4, 0)` = 4 + 4 + 4 + 0 = 12. Metot a × b'yi, a'yı b kez toplayarak hesaplar. Taban durum `b == 0`'dır ve ilerleme `b`'nin 1 azalmasıyla sağlanır. Ama `gizem(4, -1)` çağrısında `b` −1, −2, … diye azalır ve hiçbir zaman 0 olmaz. Sonuç yine `StackOverflowError`'dır. İkinci kural, ilerlemenin taban durumun *yönünde* olmasını ister; yalnız "değişiyor" olması yetmez.

Dönüşte yapılan iş, özyinelemenin en güçlü yanlarından biridir. Bir sayının ikili gösterimini yazdırmak bunun güzel bir örneğidir. Kalanlar (n % 2) en düşük basamaktan başlayarak bulunur, ama en yüksek basamaktan başlayarak yazdırılmalıdır. Dönüşte yazdıran bir metot bu ters çevirmeyi kendiliğinden yapar:

```
static void ikili(int n) {
    if (n > 1) {
        ikili(n / 2);
    }
    System.out.print(n % 2);
}
```

`ikili(13)`, 13 → 6 → 3 → 1 diye iner. Dipte `ikili(1)` 1 % 2 = 1'i yazar. Dönüşte sırasıyla 3 % 2 = 1, 6 % 2 = 0 ve 13 % 2 = 1 yazılır. Çıktı `1101`'dir: 13 = 8 + 4 + 1. Döngüyle aynı işi yapmak için kalanları bir dizide toplayıp ters sırada yazdırmak gerekirdi. Burada o dizinin işini çağrı yığını görür.

3.5.5 Örnekler

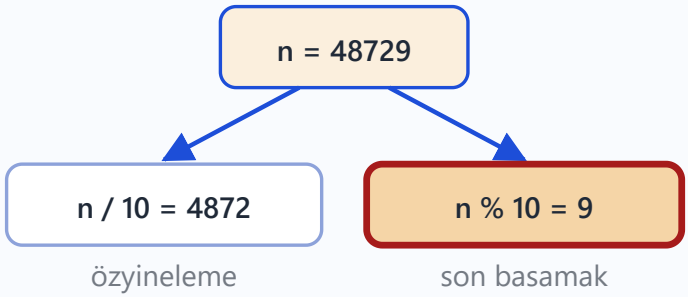
Bu kesimdeki dört örnek, özyinelemenin farklı yüzlerini gösterir. Rakamlar toplamında problem sayının basamaklarına göre küçülür. Palindromda dizgi iki uçtan kısalır ve iki ayrı taban durum vardır. Öklid algoritmasında dönüşte hiçbir iş yapılmaz. Hanoi kulelerinde ise metot kendini iki kez çağırır ve döngüyle yazması zor bir problem birkaç satıra iner.

3.5.5.1 Rakamlar toplamı

Bir tam sayının rakamlarının toplamı, örneğin 48729 için 4 + 8 + 7 + 2 + 9 = 30, özyinelemeyle şöyle bulunur:

```
int rakamlarToplami(int n) {
    if (n < 0) {
        return rakamlarToplami(-n);
    } else if (n < 10) {
        return n;
    } else {
        return rakamlarToplami(n / 10) + (n % 10);
    }
}
```

Metodun üç dalı vardır. İkinci ve üçüncü satırlar negatif sayıyı ele alır: işaret değiştirilir ve aynı metot pozitif sayıyla yeniden çağrılır. Rakamların toplamı işaretten bağımsızdır; −48729'un rakamları da 4, 8, 7, 2, 9'dur. Dördüncü ve beşinci satırlar taban durumdur: tek basamaklı bir sayının rakamları toplamı kendisidir. Son dal genel durumdur: son basamak ile kalan basamakların toplamı toplanır.



Şekil 3.5.7: 48729 sayısı iki parçaya ayrılır: `n / 10 = 4872` özyinelemeyle işlenir, `n % 10 = 9` son basamaktır.

Ayrıştırma iki tam sayı işlemiyle yapılır (Şekil 3.5.7). `n / 10`, iki tam sayı arasında bölme olduğu için kesirli kısmı atar ve son basamağı silinmiş sayıyı verir: 48729 / 10 = 4872. `n % 10`, 10'a bölümden kalanı, yani son basamağı verir: 48729 % 10 = 9. Her çağrıda sayı bir basamak kısalır; ilerleme garantilidir. Açılım şöyledir:

```
rakamlarToplami(-48729)
  = rakamlarToplami(48729)
  = rakamlarToplami(4872) + 9
  = (rakamlarToplami(487) + 2) + 9
  = ((rakamlarToplami(48) + 7) + 2) + 9
  = (((rakamlarToplami(4) + 8) + 7) + 2) + 9
  = (((4 + 8) + 7) + 2) + 9
  = ((12 + 7) + 2) + 9
  = (19 + 2) + 9
  = 21 + 9
  = 30
```

İlk çağrı yalnız işareti düzeltir. Sonraki her adımda son basamak (9, 2, 7, 8) bekleyen bir toplama olarak dışarıda kalır. Taban durum `rakamlarToplami(4)` → 4'tür. Dönüşte toplamlar içten dışa yapılır: 4 + 8 = 12, 12 + 7 = 19, 19 + 2 = 21, 21 + 9 = 30. Toplam altı çağrı yapılır: bir işaret düzeltmesi ve beş basamak. Genel olarak d basamaklı bir sayı için d çağrı (negatifse d + 1) yapılır; süre ve bellek O(d)'dir.

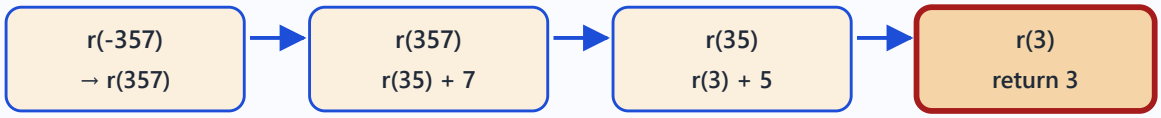
Daha kısa bir örnek, `rakamlarToplami(-357)`, çağrı yığınıyla birlikte Tablo 3.5.7'de izlenmiştir. Kısalık için metot adı `r` diye yazılmıştır.

```
static int rakamlarToplami(int n) {
    if (n < 0) {
        return rakamlarToplami(-n);
    } else if (n < 10) {
        return n;
    } else {
        return rakamlarToplami(n / 10) + (n % 10);
    }
}

public static void main(String[] args) {
    System.out.println(rakamlarToplami(-357));
}
```

Adım	Tepedeki çağrı	Çalışan dal	Bekleyen iş / dönen değer
1	`r(-357)`	`n < 0`	`r(357)` çağrılır; sonucu aynen döndürecek
2	`r(357)`	genel durum	`r(35) + 7` bekler
3	`r(35)`	genel durum	`r(3) + 5` bekler
4	`r(3)`	`n < 10`	taban durum, 3 döner
5	`r(35)`	dönüş	3 + 5 = 8 döner
6	`r(357)`	dönüş	8 + 7 = 15 döner
7	`r(-357)`	dönüş	15'i değiştirmeden döndürür
8	`main`	—	15 yazdırılır

Tablo 3.5.7: `rakamlarToplami(-357)` çağrısının izlenmesi.



Şekil 3.5.8: `rakamlarToplami(-357)` çağrı zincirinin en derin anı. `r(3)` taban durumdur; üstündeki çerçeveler sırasıyla `r(357)`'yi, `r(35) + 7`'yi ve `r(3) + 5`'i bekler.

Dört çağrı yapılır: bir işaret düzeltmesi ve üç basamak. Sonuç 3 + 5 + 7 = 15'tir (Şekil 3.5.8).

Sık yapılan hata

İşaret değiştirmenin her zaman pozitif sayı verdiğini sanmak. `int`'in en küçük değeri `Integer.MIN_VALUE` = −2.147.483.648'dir ve +2.147.483.648 `int`'e sığmaz. Java'da `-Integer.MIN_VALUE` yine −2.147.483.648 verir. Bu yüzden

`rakamlarToplami(Integer.MIN_VALUE)` ilk dala tekrar tekrar girer ve `StackOverflowError` ile biter. `Math.abs(Integer.MIN_VALUE)` da aynı nedenle negatif bir sayı döndürür. Bu tek değeri ayrıca ele almak ya da hesabı `long` ile yapmak gerekir.

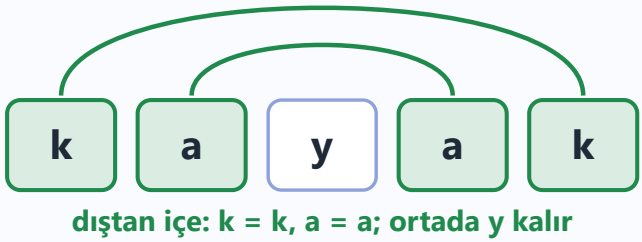
3.5.5.2 Palindrom kontrolü

Tanım 3.5.9

Palindrom, tersten okunuşu düz okunuşuyla aynı olan dizgidir: “kayak”, “abba”, “ey edip adanada pide ye”.

Bir dizginin palindrom olup olmadığı dıştan içe doğru bakılarak anlaşılır. İlk ve son karakter aynıysa, bu iki karakter atıldığında kalan dizginin de palindrom olması gerekir (Şekil 3.5.9). Bu, özyinelemeli bir tanımdır:

```
static boolean palindromMu(String s) {
    if (s.length() <= 1) {
        return true;
    } else if (s.charAt(0) == s.charAt(s.length() - 1)) {
        return palindromMu(s.substring(1, s.length() - 1));
    }
    return false;
}
```



Şekil 3.5.9: “kayak” dıştan içe karşılaştırılır: k = k, a = a. Ortadaki y tek kalır ve kendisiyle eşleşir.

Kod iki `String` metodunu kullanır. `s.charAt(i)`, dizginin i. karakterini verir; indisler 0'dan başlar, son karakterin indisi `s.length() - 1`'dir. `s.substring(1, s.length() - 1)` ise 1. indisten başlayıp `s.length() - 1`. indisin hemen öncesinde biten parçayı verir; yani ilk ve son harfi atılmış dizgiyi. `substring`'in ikinci parametresi parçaya dâhil değildir.

Metotta **iki taban durum** vardır. Uzunluk 1 ya da 0 ise dizgi palindromdur ve `true` döner. Boş dizgi, çift uzunluklu palindromlarda ortaya çıkar: “abba” → “bb” → “” diye kısalır. Uçlar farklıysa dizgi palindrom değildir ve `false` döner; içerisine bakmaya gerek yoktur. Bu ikinci taban durum, `if` ve

`else if`'in ikisi de tutmadığında son satırdaki `return false;` ile karşılanır.

Girdi	Çağrı zinciri	Sonuç
`"kayak"`	"kayak" → "aya" → "y"	`true`
`"abba"`	"abba" → "bb" → ""	`true`
`"java"`	"java": j ≠ a	`false`
`"Kayak"`	"Kayak": K ≠ k	`false`

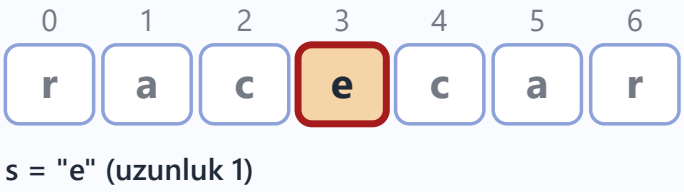
Tablo 3.5.8: `palindromMu` için birkaç girdi.

Son satır önemli bir ayrıntıyı gösterir: karşılaştırma büyük-küçük harfe duyarlıdır. `'K'` ile `'k'` farklı karakterlerdir ve “Kayak” palindrom sayılmaz. “ey edip adanada pide ye” ise boşluklarıyla birlikte simetrik olduğu için `true` verir. Boşluk ve noktalama içeren tümcelerde önce bu karakterlerin atılması ve harflerin küçültülmesi gerekir.

“racecar” sözcüğü için yapılan çağrılar Tablo 3.5.9'da izlenmiştir.

Çağrı	`s`	Uzunluk	Uçlar	Sonraki çağrı / dönüş
1	`"racecar"`	7	'r' = 'r'	`s.substring(1, 6)` = `"aceca"`
2	`"aceca"`	5	'a' = 'a'	`s.substring(1, 4)` = `"cec"`
3	`"cec"`	3	'c' = 'c'	`s.substring(1, 2)` = `"e"`
4	`"e"`	1	—	taban durum: `true` döner

Tablo 3.5.9: `palindromMu("racecar")` çağrısının izlenmesi.



Şekil 3.5.10: Dördüncü çağrıda incelenen parça yalnız ortadaki “e”dir. Uzunluk 1 ≤ 1 olduğu için taban durum çalışır.

Dördüncü çağrı taban durumdur (Şekil 3.5.10). Dönüşte her çerçeve aldığı `true`'yu değiştirmeden yukarı aktarır; dönüşte yapılacak başka bir iş yoktur. Sonuç: “racecar” bir palindromdur.

Maliyet açısından bu metot ilk bakışta göründüğünden pahalıdır. n uzunluklu bir dizgi için yaklaşık n/2 çağrı yapılır. Ama her çağrıdaki `substring`, Java'da yeni bir dizgi nesnesi oluşturur ve karakterleri kopyalar. Kopyalanan karakterlerin toplamı $(n - 2) + (n - 4) + \ldots \approx n^2/4$'tür. Yani süre $O(n^2)$, bellek de bekleyen dizgiler yüzünden $O(n^2)$'dir. Dizgi yerine iki indis taşıyan bir sürüm, örneğin `palindromMu(s, bas, son)`, bu kopyalamayı önler ve O(n) sürede çalışır.

İndisli sürüm şöyledir:

```
static boolean palindromMu(String s, int bas, int son) {
    if (bas >= son) {
        return true;
    } else if (s.charAt(bas) != s.charAt(son)) {
        return false;
    }
    return palindromMu(s, bas + 1, son - 1);
}
```

Dizgi hiç kısalmaz; onun yerine incelenen parçanın sınırları daralır. `bas` parçanın ilk, `son` son karakterinin indisidir. İlk çağrı `palindromMu(s, 0, s.length() - 1)` biçiminde yapılır. Taban durumlar aynı mantığı izler. `bas >= son` ise parçada en çok bir karakter kalmıştır ve sonuç `true`'dur. Uçlar farklıysa sonuç `false`'tur. İlerleme, iki indisin birbirine yaklaşmasıyla sağlanır: aradaki fark her çağrıda 2 azalır. “racecar” için çağrılar (0, 6), (1, 5), (2, 4), (3, 3) olur ve dördüncü çağrı taban durumdur. Boş dizgi için ilk çağrı (0, −1) olur ve hemen `true` döner. Bu teknik, dizgi ve dizi üzerinde çalışan özyinelemeli metotlarda yaygındır: veriyi kopyalamak yerine sınırları parametre olarak taşımak.

3.5.5.3 En büyük ortak bölen

İki pozitif tam sayının **en büyük ortak böleni** (EBOB), ikisini de kalansız bölen en büyük sayıdır. İngilizce adı *greatest common divisor* olduğu için kodda metot `gcd` diye adlandırılmıştır. Yaklaşık 2300 yıllık Öklid algoritması EBOB'u bölme kalanlarıyla bulur:

> EBOB(x, y) = EBOB(y, x mod y), EBOB(x, 0) = x

```
static int gcd(int x, int y) {
    if (y == 0) {
        return x;
    }
    return gcd(y, x % y);
}
```

Taban durum `y == 0`'dır: bir sayının 0 ile EBOB'u kendisidir. Genel durumda `x` yerine `y`, `y` yerine `x % y` geçer. Bağıntının doğruluğu şuna dayanır: x ile y'yi bölen her sayı x − k·y'yi de böler, dolayısıyla x mod y'yi de böler. Ortak bölenler kümesi her adımda aynı kalır, yalnız sayılar küçülür.

İlerleme kalanın küçülmesiyle sağlanır. x mod y her zaman y'den küçüktür; ikinci parametre her çağrıda kesin olarak azalır ve sonunda 0'a ulaşır. Parametrelerin sırası da önemli değildir. `gcd(93, 444)` çağrısında 93 % 444 = 93 olduğu için ilk adım yalnız iki sayının yerini değiştirir ve `gcd(444, 93)` olur.

```
gcd(444, 93)
  = gcd(93, 72)     // 444 % 93 = 72
  = gcd(72, 21)     // 93 % 72 = 21
  = gcd(21, 9)      // 72 % 21 = 9
  = gcd(9, 3)       // 21 % 9 = 3
  = gcd(3, 0)       // 9 % 3 = 0
  = 3
```

Çağrı	x	y	x % y	Olay
`gcd(444, 93)`	444	93	72	444 = 4 · 93 + 72
`gcd(93, 72)`	93	72	21	93 = 1 · 72 + 21
`gcd(72, 21)`	72	21	9	72 = 3 · 21 + 9
`gcd(21, 9)`	21	9	3	21 = 2 · 9 + 3
`gcd(9, 3)`	9	3	0	9 = 3 · 3 + 0
`gcd(3, 0)`	3	0	—	taban durum: 3 döner

Tablo 3.5.10: `gcd(444, 93)` çağrılarının izlenmesi.

Altı çağrı yapılır ve EBOB(444, 93) = 3 bulunur (Tablo 3.5.10). Bu metodun öncekilerden farkı dönüşte görülür. `return gcd(y, x % y);` satırında çağrı döndükten sonra yapılacak hiçbir iş yoktur. Son çağrının bulduğu 3, her çerçeveden değişmeden geçerek `main`'e ulaşır. Böyle çağrılara kuyruk çağrısı denir. Başka bir örnek: `gcd(48, 18)` → `gcd(18, 12)` → `gcd(12, 6)` → `gcd(6, 0)` → 6.

Öklid algoritması çok hızlıdır. Her iki adımda ilk sayı en az yarıya iner; bu yüzden çağrı sayısı sayıların basamak sayısıyla orantılıdır, yani O(log min(x, y))'dir. En kötü durum ardışık iki Fibonacci sayısıyla ortaya çıkar; örneğin `gcd(89, 55)` on çağrı yapar.

3.5.5.4 Hanoi kuleleri

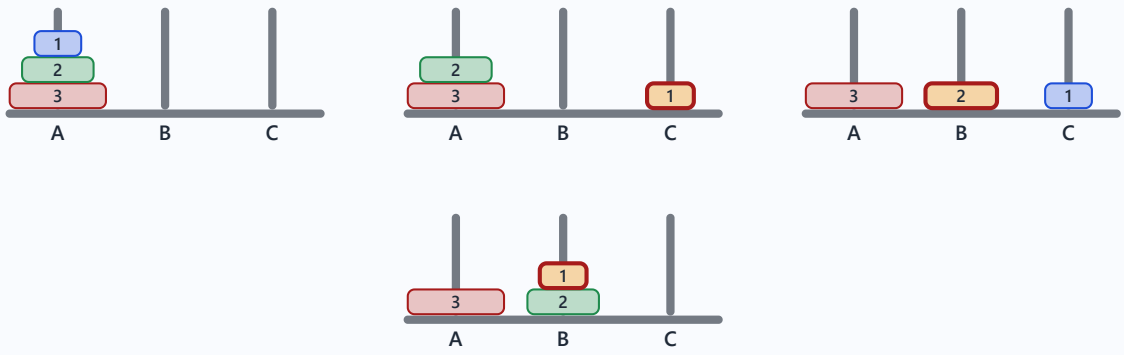
Hanoi kuleleri, 1883'te Édouard Lucas'nın ortaya attığı bir bulmacadır. Üç çubuk (A, B, C) vardır ve A çubuğunda büyükten küçüğe dizilmiş n disk bulunur. Disklerin hepsi C'ye taşınacaktır. İki kural vardır: her hamlede yalnız bir disk, bir çubuğun en üstündeki disk taşınır; büyük bir disk hiçbir zaman küçük bir diskin üstüne konmaz.

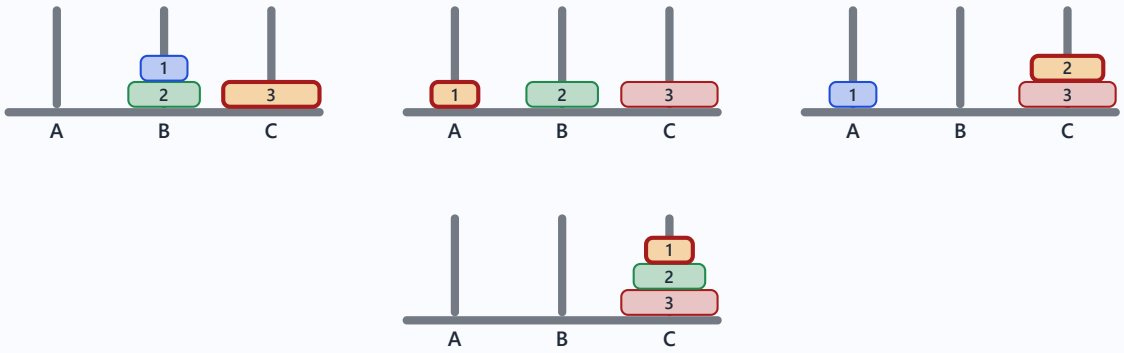
Bulmaca döngüyle düşünüldüğünde zordur, ama özyinelemeli düşünüldüğünde çözüm kendiliğinden ortaya çıkar. En büyük disk C'ye ancak üstündeki n − 1 disk başka bir yere, yani yardımcı çubuk B'ye çekildiğinde taşınabilir. Öyleyse çözüm üç adımdır: üstteki n − 1 disk yardımcıya, en büyük disk hedefe, sonra n − 1 disk yardımcıdan hedefe taşınır. Birinci ve üçüncü adımlar, bir disk eksik aynı problemdir. Taban durum tek disktir: doğrudan taşınır.

```java
public class HanoiKuleleri {
    static void coz(int n, char kaynak,
                    char hedef, char yardimci) {
        if (n == 1) {   // tek disk: doğrudan taşı
            System.out.println(n + ": " + kaynak + "->" + hedef);
        } else {
            coz(n - 1, kaynak, yardimci, hedef);
            System.out.println(n + ": " + kaynak + "->" + hedef);
            coz(n - 1, yardimci, hedef, kaynak);
        }
    }
    public static void main(String[] args) {
        coz(3, 'A', 'C', 'B');
    }
}
```

```
1: A->C
2: A->B
1: C->B
3: A->C
1: B->A
2: B->C
1: A->C
```

`coz` metodu dört parametre alır: disk sayısı ve kaynak, hedef, yardımcı çubukların adları. Çubuk adları `char` tipindedir. Özyinelemeli çağrılarda çubukların rolleri değişir. İlk çağrıda `coz(n - 1, kaynak, yardimci, hedef)` yazılır: n − 1 disk kaynaktan *yardımcıya* taşınır ve bu sırada asıl hedef yardımcı rolünü üstlenir. Üçüncü satırdaki çağrıda `coz(n - 1, yardimci, hedef, kaynak)` yazılır: n − 1 disk yardımcıdan hedefe taşınır, bu kez kaynak boş kaldığı için yardımcı olur. Yazdırılan satırdaki sayı, taşınan diskin numarasıdır; en küçük disk 1'dir.





Şekil 3.5.11: `coz(3, 'A', 'C', 'B')` : başlangıç ve yedi hamlenin her birinden sonraki durum (soldan sağa, yukarıdan aşağıya). Üçüncü hamleden sonra üstteki iki disk B'dedir; dördüncü hamle en büyük diski C'ye taşır.

Hamle	Çıktı	Yazdıran çağrı	Anlamı
1	`1: A->C`	`coz(1, A, C, B)`	taban durum
2	`2: A->B`	`coz(2, A, B, C)`	2. disk yardımcıya
3	`1: C->B`	`coz(1, C, B, A)`	üstteki 2 disk artık B'de
4	`3: A->C`	`coz(3, A, C, B)`	en büyük disk hedefe
5	`1: B->A`	`coz(1, B, A, C)`	2 disk C'ye taşınmaya başlar
6	`2: B->C`	`coz(2, B, C, A)`	2. disk 3. diskin üstüne
7	`1: A->C`	`coz(1, A, C, B)`	bulmaca çözüldü

Tablo 3.5.11: `coz(3, 'A', 'C', 'B')` hamlelerinin hangi çağrıdan geldiği.

Hamleler Tablo 3.5.11'de, yapan çağrıyla birlikte verilmiştir. İlk üç hamle `coz(2, A, B, C)` 'nin işidir; dördüncü hamle `coz(3, …)` 'ün ortadaki satırıdır; son üç hamle `coz(2, B, C, A)` 'nın işidir (Şekil 3.5.11). Çağrı yığını hiçbir anda üç `coz` çerçevesinden derine inmez; yığın derinliği disk sayısına eşittir.

Teorem 3.5.2 (Hanoi hamle sayısı)

n diskli Hanoi kuleleri en az $2^n - 1$ hamlede çözülür ve `coz` metodu tam bu kadar hamle yapar.

Hamle sayısı H(n), metodun yapısından okunur: n − 1 disk için iki kez çözüm ve bir hamle. $H(n) = 2 \cdot H(n-1) + 1$ ve $H(1) = 1$'dir. Buradan $H(2) = 3$, $H(3) = 7$, $H(4) = 15$ bulunur ve genel olarak $H(n) = 2^n - 1$'dir. Bu sayı çok hızlı büyür. 10 disk için 1023 hamle gerekir. Efsanedeki 64 disk için $2^{64} - 1 \approx 1{,}8 \times 10^{19}$ hamle gerekir; saniyede bir hamleyle bu, yaklaşık 585 milyar yıl sürer. Metodun süresi de $O(2^n)$'dir, ama bu kez israf yoktur: her hamle gereklidir ve daha kısa bir çözüm bulunmaz.

Bulmacanın ilginç bir özelliği, ilk hamlenin disk sayısına bağlı olmasıdır. Ölçülen ilk hamleler, n tek ise `1: A->C`, n çift ise `1: A->B`'dir. Örneğin 4 disk için 15 hamle yapılır ve ilk hamle en küçük diski yardımcı çubuğa taşır. Döngüyle yazılmış Hanoi çözümleri bu tür gözlemlere dayanır ve özyinelemeli sürümden çok daha uzundur. Özyinelemeli sürüm ise problemin yapısını olduğu gibi yansıttığı için birkaç satırdır.

3.5.6 Verimlilik

Özyinelemeli bir çözüm kısa ve açık olabilir, ama bu onun hızlı ya da hafif olduğu anlamına gelmez. Bu kesim özyinelemenin bedelini üç başlıkta toplar: her çağrının kendi maliyeti, yığın derinliği ve aynı değerlerin defalarca hesaplanması. Ardından özyinelemeli metotların döngüye nasıl çevrildiği gösterilir.

3.5.6.1 Bir çağrının bedeli

Her metot çağrısında JVM bir iş yapar: yeni çerçeve açılır, parametreler kopyalanır, dönülecek yer kaydedilir. Dönüşte çerçeve silinir ve değer alttaki çerçeveye taşınır. Bir döngü turunda ise yalnız birkaç değişken güncellenir. Bu yüzden aynı işi yapan özyinelemeli metot çoğu zaman döngüden biraz yavaştır.

Daha önemli fark bellektedir. Özyinelemeli çağrılar yapıldıkça bekleyen kısmi sonuçlar, yani her çerçevedeki `4 * ?` gibi yarım işler, çağrı yığınında birikir. Tablo 3.5.12 üç ana bedeli özetler.

Bedel	Açıklama	Örnek
Bellek	Her bekleyen çağrı bir çerçeve tutar.	`faktoriyel(n)` : n çerçeve, O(n) bellek; döngüyle O(1).
Derinlik sınırı	Yığın dolunca `StackOverflowError` alınır.	Varsayılan yığınla birkaç on bin çağrı derinliğinden sonra.
Tekrar	Ağaç özyinelemesi aynı değeri defalarca hesaplayabilir.	`fibonacci(40)` : 331.160.281 çağrı.

Tablo 3.5.12: Özyinelemenin üç bedeli.

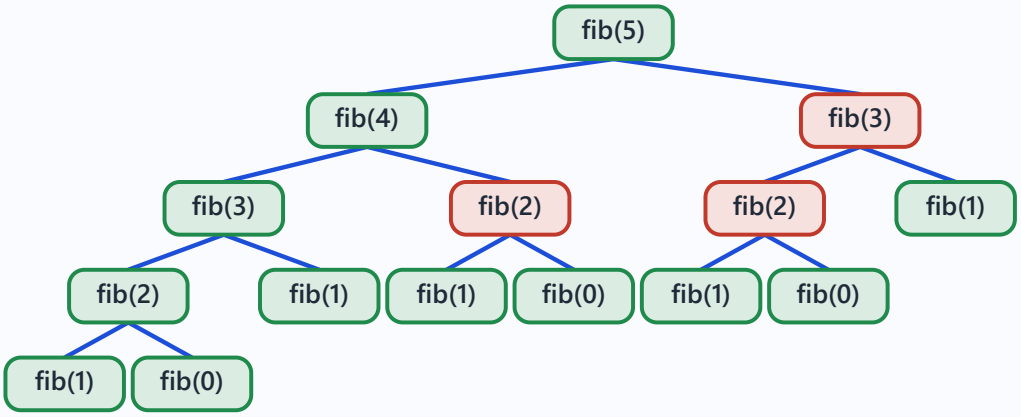
> **Tanım 3.5.10**
>
> **İlerlemeli çözüm** (*iterative solution*), tekrarı özyineleme yerine döngüyle yapan çözümdür. Kısmi sonuçları yığında biriktirmez; bir biriktirici değişkende tutar ve tek bir çerçeveyle çalışır.

Sorunlu durumlarda çözüm ilerlemeli sürüme geçmektir. Faktöriyel bunun en basit örneğidir: `faktoriyel(100000)` özyinelemeli sürümde, sonucun taşması bir yana, büyük olasılıkla `StackOverflowError` ile biter. JDK 21 ile yapılan denemede gerçekten de böyle oldu. Döngülü sürüm

aynı çağrıyı tek çerçevede ve bir saniyeden kısa sürede bitirir; sonucu yanlış olsa da hiçbir zaman yığın taşmasına uğramaz.

3.5.6.2 Tekrarlanan hesaplama

Ağaç özyinelemesinin asıl sorunu bellek değil, zamandır. Şekil 3.5.12, `fibonacci(5)` 'in çağrı ağacını gösterir. Ağaçta 15 düğüm vardır. Kırmızı düğümler, daha önce hesaplanmış değerlerin yeniden hesaplanmasıdır: `fib(3)` iki kez, `fib(2)` üç kez hesaplanır.



Şekil 3.5.12: `fibonacci(5)` çağrı ağacı: 15 çağrı. Kırmızı düğümler daha önce hesaplanmış değerlerin yeniden hesaplanmasıdır.

Tekrar n büyüdükçe katlanarak artar. Çağrı sayısı Ç(n) = 2F(n + 1) − 1 formülüyle bulunur ve n 1 artınca yaklaşık 1,6 katına çıkar; bu oran altın orandır. Tablo 3.5.13, JDK 21 ile ölçülen çağrı sayılarını ve süreleri verir.

n	Çağrı sayısı	F(n)	Ölçülen süre
10	177	55	< 0,1 ms
20	21.891	6.765	≈ 0,3 ms
30	2.692.537	832.040	≈ 5 ms
40	331.160.281	102.334.155	≈ 0,5 s
45	3.672.623.805	1.134.903.170	≈ 11 s

Tablo 3.5.13: Belleklemesiz `fibonacci(n)` için çağrı sayısı ve ölçülen süre.

Tablodaki her on adımda çağrı sayısı yaklaşık 120 katına çıkar. `fibonacci(40)` yarım saniye sürer. Bu hızla `fibonacci(50)` iki dakikayı bulur; yani dakikalar alır. Bu koda Java'ya özgü bir sorun daha eklenir: F(47) = 2.971.215.073, `int` 'in sınırını aşar. `int` döndüren `fibonacci(50)` iki dakika çalıştıktan sonra −298632863 gibi anlamsız bir sayı döndürür. Doğru değer 12.586.269.025'tir.

Sorunun kaynağı, metodun daha önce hesapladığı değerleri unutmasıdır. `fibonacci(40)` 'ın ihtiyaç duyduğu farklı değer sayısı yalnız 41'dir (F(0)'dan F(40)'a kadar). Her biri bir kez hesaplanıp saklansa birkaç düzine işlem yeterdi. Bunu yapmanın iki yolu vardır: döngüye çevirmek ve bellekleme.

3.5.6.3 Döngüye çevirme

Doğrusal özyinelemeli metotlar, yani kendini bir kez çağıran metotlar, genellikle kolayca döngüye çevrilir. Yöntem şudur: dönüşte yapılan iş, döngüde bir biriktirici değişkenle yapılır. Faktöriyelin iki sürümü yan yana şöyledir:

```
int faktoriyel(int n) {
    if (n == 0 || n == 1) {
        return 1;
    } else {
        return n * faktoriyel(n - 1);
    }
}
```

```
int faktoriyel(int n) {
    int carpim = 1;
    for (int i = 1; i <= n; i++) {
        carpim *= i;
    }
    return carpim;
}
```

Döngülü sürümde `carpim` 1'den başlar; çarpmanın etkisiz elemanı 1'dir. Döngü `i` 'yi 1'den `n` 'ye kadar ilerletir ve her turda `carpim` 'ı `i` ile çarpar. `faktoriyel(5)` için `carpim` sırasıyla 1, 1, 2, 6, 24, 120 olur. `n = 0` için döngü hiç dönmez ve 1 döner; 0! = 1 ayrıca yazılmaya gerek kalmadan karşılanır. İki sürüm aynı sonucu verir, ama bellek kullanımları farklıdır (Şekil 3.5.13).



Şekil 3.5.13: Özyinelemeli `faktoriyel(5)` beş çerçeve açar; döngülü sürüm tek çerçevede `carpim` değişkenini günceller.

Rakamlar toplamı ve palindrom da aynı yolla döngüye çevrilir:

```
int rakamlarToplami(int n) {
    int toplam = 0;
    int kalan = Math.abs(n);
    while (kalan > 0) {
        toplam += kalan % 10;
        kalan /= 10;
    }
    return toplam;
}
```

`kalan`, sayının mutlak değeriyle başlar. Her turda son basamak `toplam`'a eklenir ve `kalan`'dan atılır. 48729 için `toplam` 9, 11, 18, 26, 30 olur; `kalan` 4872, 487, 48, 4, 0 olur ve döngü biter. `rakamlarToplami(-48729)` 30 döndürür; özyinelemeli sürümle aynıdır. İşaret `Math.abs` ile atılır, çünkü rakamların toplamı negatif olamaz. Bu sürüm `Integer.MIN_VALUE` için yığın taşmasına uğramaz, ama `Math.abs` negatif bir değer döndürdüğü için döngüye hiç girmez ve 0 verir; doğru cevap 47'dir.

```
boolean palindromMu(String s) {
    s = s.toLowerCase();
    int uzunluk = s.length();
    for (int i = 0; i < uzunluk / 2; i++) {
        if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {
            return false;
        }
    }
    return true;
}
```

Döngülü palindrom, dizgiyi kısaltmak yerine iki uçtan içe doğru ilerleyen indisler kullanır. `i` baştan, `uzunluk - 1 - i` sondan ilerler. Döngü dizginin yarısına kadar döner; tek uzunlukta ortadaki karakter kendisiyle eşleşir ve bakılmaz. İlk farklı çiftte `false` döner. Hiç farklı çift bulunmazsa döngü biter ve `true` döner. Bu sürüm başta `toLowerCase()` çağırdığı için “Kayak”'ı da palindrom sayar.

“Kepek” için izleme Tablo 3.5.14'te verilmiştir.

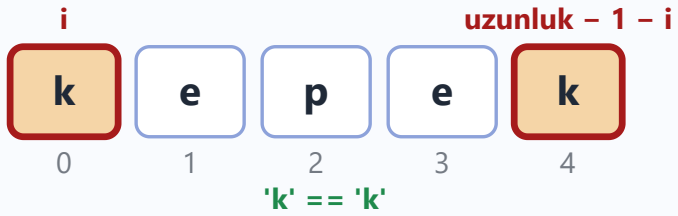
```
static boolean palindromMu(String s) {
    s = s.toLowerCase();
    int uzunluk = s.length();
    for (int i = 0; i < uzunluk / 2; i++) {
        if (s.charAt(i) != s.charAt(uzunluk - 1 - i)) {
            return false;
        }
    }
    return true;
}

public static void main(String[] args) {
    System.out.println(palindromMu("Kepek"));
}
```

Adım	`s`	`uzunluk`	`i`	Karşılaştırma	Sonuç
1	`"Kepek"`	—	—	—	parametre
2	`"kepek"`	—	—	—	`toLowerCase()` yeni dizgi döndürür
3	`"kepek"`	5	—	—	döngü sınırı 5 / 2 = 2
4	`"kepek"`	5	0	`s.charAt(0)` 'k', `s.charAt(4)` 'k'	eşit

Adım	`s`	`uzunluk`	`i`	Karşılaştırma	Sonuç
5	`"kepek"`	5	1	`s.charAt(1)` 'e', `s.charAt(3)` 'e'	eşit
6	`"kepek"`	5	2	`2 < 2` yanlış	döngü biter
7	`"kepek"`	5	—	—	`true` döner ve yazdırılır

Tablo 3.5.14: Döngülü `palindromMu("Kepek")` çağrısının izlenmesi.



Şekil 3.5.14: `i = 0` turunda karşılaştırılan indisler: baştan 0., sondan 5 − 1 − 0 = 4. indis. İki karakter de 'k'dir.

Ortadaki 'p' hiç karşılaştırılmaz; kendisiyle eşleşmesi zaten kesindir (Şekil 3.5.14). Tek çerçeve kullanılır ve özyinelemeli sürümdeki gibi her adımda yeni bir dizgi üretilmez. Süre O(n), ek bellek ise `toLowerCase()`'in ürettiği tek kopya dışında O(1)'dir.

Ağaç özyinelemesi de döngüye çevrilebilir, ama bu kez biriktirici tek değişken değildir. Fibonacci'de her terim önceki *iki* terime bağlıdır; döngü de bu iki terimi taşır:

```
static long fibonacci(int n) {
    long a = 0, b = 1;
    for (int i = 0; i < n; i++) {
        long yeni = a + b;
        a = b;
        b = yeni;
    }
    return a;
}
```

`a` ve `b`, ardışık iki Fibonacci sayısıdır ve F(0) = 0, F(1) = 1 ile başlar. Her turda ikisi birden bir adım ileri kayar: yeni `a` eski `b`, yeni `b` eski toplam olur. Geçici `yeni` değişkeni olmadan yazılırsa eski `a` kaybolur. n turdan sonra `a` = F(n)'dir. `fibonacci(10)` için `(a, b)` çifti (0, 1), (1, 1), (1, 2), (2, 3), (3, 5), (5, 8), (8, 13), (13, 21), (21, 34), (34, 55), (55, 89) diye ilerler ve 55 döner. Bu sürüm n tur döner: süre O(n), bellek O(1)'dir. `fibonacci(50)` = 12586269025'i ve `fibonacci(90)`'ı anında verir. Özyinelemeli sürümün 331 milyon çağrıyla yaptığı işi bu döngü 40 turda yapar.

Sık yapılan hata

Döngü sınırını `i <= uzunluk / 2` ya da `i < uzunluk` yazmak. İlk yazım sonucu değiştirmez ama gereksiz bir tur ekler. İkincisinde her çift iki kez karşılaştırılır. Asıl tehlike, karşı indisi `uzunluk - i` diye yazmaktır: `i = 0` 'da `s.charAt(5)` çağrılır ve `StringIndexOutOfBoundsException` fırlatılır. Son karakterin indisi `uzunluk - 1` 'dir.

3.5.6.4 Özyineleme mi döngü mü?

İki yaklaşım aynı tekrarın iki farklı yazımıdır. Tablo 3.5.15 aralarındaki farkları özetler.

	Özyineleme	**Döngü (ilerlemeli)**
Durma	Taban durum	Döngü koşulu
İlerleme	Daha küçük argümanla çağrı	Sayaç ya da değişken güncelleme
Bellek	Her çağrı için bir çerçeve: O(derinlik)	Genellikle O(1)
Hız	Çağrı maliyeti; ağaç özyinelemesinde tekrar	Genellikle daha hızlı
Hata	Taban durum yoksa `StackOverflowError`	Koşul bozuksa sonsuz döngü (program donar)
Okunabilirlik	Ağaç, böl ve fethet, Hanoi, seriler: çok doğal	Basit sayma ve biriktirme işleri için doğal

Tablo 3.5.15: Özyineleme ile döngünün karşılaştırması.

Hata satırındaki fark ilginçtir. Bozuk bir döngü hiç bitmez ve program donar; dışarıdan durdurulması gerekir. Bozuk bir özyineleme ise kısa sürede yığını doldurur ve bir hata iletisiyle durur. Hatayı bulmak açısından ikincisi daha kolaydır, çünkü yığın izi sorunun yerini gösterir.

Her özyinelemeli çözüm döngüyle yazılabilir; tersi de doğrudur. Doğrusal özyinelemede bu çeviri bir biriktiriciyle kolayca yapılır. Hanoi gibi ağaç özyinelemelerinde ise döngülü sürüm, çağrı yığınının işini kendi tuttuğu bir yığın veri yapısıyla taklit etmek zorundadır ve kod uzar. Seçim üç ölçüte göre yapılır. Kodun açıklığı özyinelemeden yanaysa ve derinlik küçükse (Hanoi'de disk sayısı, dengeli bir ağaçta yükseklik) özyineleme uygundur. Derinlik girdiyle doğrusal büyüyorsa (bir milyon öğeli liste) döngü seçilir. Performans kritikse ve ağaç özyinelemesi tekrar içeriyorsa döngü ya da bellekleme gerekir.

3.5.7 Kuyruk özyinelemesi

Özyinelemeli metotlarda bellek sorunu, dönüşte yapılacak işlerin yığında beklemesinden doğar. Dönüşte yapılacak hiçbir iş kalmazsa çerçevelerin beklemesine de gerek kalmaz. Kuyruk özyinelemesi bu fikre dayanır.

3.5.7.1 Kuyruk çağrısı ve biriktirici

Tanım 3.5.11

Kuyruk özyinelemesi (*tail recursion*), metodun son işleminin özyinelemeli çağrının kendisi olduğu özyinelemedir; çağrı döndükten sonra yapılacak iş kalmaz. Sonuç, ek bir **biriktirici** (*accumulator*) parametrede taşınır.

Ayrım bir satırda görülür. Sıradan faktöriyel ile kuyruk özyinelemeli faktöriyelin özyinelemeli satırları şöyledir:

```
return n * faktoriyel(n - 1);
// çağrı döndükten SONRA çarpma var
```

```
return faktoriyel(n - 1, n * birikim);
// çarpma çağrıdan ÖNCE; son iş çağrı
```

Soldaki satırda önce `faktoriyel(n - 1)` çağrılır, döndükten *sonra* sonuç `n` ile çarpılır. Çarpma bekleyen iştir ve `n`'nin değeri bu yüzden çerçevede saklanmak zorundadır. Sağdaki satırda çarpma çağrıdan *önce*, argüman hesaplanırken yapılır. `n * birikim` değeri yeni çağrıya parametre olarak gönderilir. Çağrı döndüğünde gelen değer olduğu gibi döndürülür; yapılacak başka iş yoktur (Şekil 3.5.15).



Şekil 3.5.15: Solda sıradan özyineleme: her çerçeve dönüşte yapılacak bir çarpım tutar. Sağda kuyruk özyinelemesi: çerçeveler yalnız argümanları taşır, bekleyen iş yoktur.

Bu biçim, derleyicinin **kuyruk çağrısı optimizasyonu** (*tail call optimization*, TCO) yapmasını mümkün kılar. Bekleyen iş olmadığı için yeni çağrı, eski çerçeveyi yeniden kullanabilir; çağrı bir döngü gibi işlenir ve çerçeveler yığılmaz. Bellek O(n)'den O(1)'e iner ve derinlik sınırı ortadan kalkar.

Java ise bu optimizasyonu yapmaz. Ne `javac` derleyicisi ne de JVM, kuyruk çağrılarını döngüye çevirir. Kuyruk özyinelemeli Java kodu yine her çağrı için bir çerçeve açar ve O(n) yığın kullanır; derinlik sınırı da aynen geçerlidir. Kuyruk çağrısı optimizasyonu Scheme dilinde zorunludur. Scala'da derleyici kendini çağıran kuyruk özyinelemelerini döngüye çevirir ve `@tailrec` işareti bunu denetletir.

Kotlin'de `tailrec` anahtar sözcüğüyle işaretlenen metotlar döngüye çevrilir. Java'da kuyruk özyinelemesinin değeri, döngüye çevirmeyi kolaylaştırmasıdır: biriktiricili bir kuyruk özyinelemesi, neredeyse mekanik olarak bir `while` döngüsüne dönüştürülür.

> **Sık yapılan hata**
>
> Çağrının satırın sonunda durmasını kuyruk çağrısı sanmak. `return 1 + basamak(n / 10);` satırında çağrı en sağdadır, ama dönüşte 1 eklenir; kuyruk çağrısı değildir. `return ters(s.substring(1)) + s.charAt(0);` satırında da dönüşte birleştirme yapılır. Ölçüt yazılış yeri değil, çağrıdan sonra iş kalıp kalmadığıdır.

3.5.7.2 Örnekler ve sarmalayıcı metot

Doğrusal özyinelemeli bir metot, bir biriktirici parametre eklenerek kuyruk özyinelemeli yapılır. Rakamlar toplamı ve faktöriyel için sonuç şöyledir:

```
int rakamlarToplami(int sayi, int toplam) {
    if (sayi == 0) {
        return toplam;
    } else {
        return rakamlarToplami(sayi / 10,
                               toplam + sayi % 10);
    }
}
```

Son basamak çağrıdan önce `toplam`'a eklenir. `rakamlarToplami(48729, 0)` şöyle ilerler: (48729, 0) → (4872, 9) → (487, 11) → (48, 18) → (4, 26) → (0, 30). `sayi == 0` olunca biriktiricideki 30 döner. Bu sürüm yalnız `n ≥ 0` için doğrudur; `rakamlarToplami(-357, 0)`, Java'nın negatif sayılarda kalanı negatif vermesi yüzünden −15 döndürür.

```
int faktoriyel(int n, int birikim) {
    if (n == 0) {
        return birikim;
    } else {
        return faktoriyel(n - 1, n * birikim);
    }
}
```

Faktöriyelde çarpım çağrıdan önce `birikim`'e katılır ve `faktoriyel(5, 1)` 120 döndürür. Taban durum bu kez `n == 0`'dır; o anda biriktiricide sonuç hazırdır.

Biriktiricinin başlangıç değeri, işlemin etkisiz elemanıdır: toplama için 0, çarpma için 1. Bu ayrıntı metodu kullanan kişiye yük olmamalıdır. Bu yüzden biriktiriciyi gizleyen tek parametreli bir **sarmalayıcı metot** yazılır:

```
static int faktoriyel(int n) {
    return faktoriyel(n, 1);
}
```

İki metot aynı adı taşır, ama parametre sayıları farklıdır. Java bu durumda çağrıdaki argüman sayısına bakarak hangisinin kastedildiğini anlar; bu özelliğe metot aşırı yüklemesi (*method overloading*) denir. Kullanıcı `faktoriyel(5)` yazar ve biriktiriciyi hiç görmez.

Bir de karşı örnek: `ters` metodu bir dizgiyi ters çevirir.

```
static String ters(String s) {
    if (s.isEmpty()) return "";
    return ters(s.substring(1)) + s.charAt(0);
}
```

`ters("java")`, `ters("ava") + 'j'` olarak hesaplanır; o da `ters("va") + 'a'` ister ve böyle sürer. Sonuç `"avaj"`'dır. Bu metot kuyruk özyinelemeli değildir, çünkü çağrı döndükten sonra birleştirme (`+`) yapılır. `gcd` ise kuyruk özyinelemelidir: son işi çağrının kendisidir.

Fibonacci'nin kuyruk özyinelemeli sürümü, döngülü sürümdeki iki değişkeni parametre olarak taşır:

```
static long fibK(int n, long a, long b) {
    if (n == 0) {
        return a;
    }
    return fibK(n - 1, b, a + b);
}
```

`a` ve `b` yine ardışık iki terimdir ve ilk çağrı `fibK(n, 0, 1)`'dir. Her çağrı `n`'yi 1 azaltır ve çifti bir adım kaydırır. `n` 0'a indiğinde `a`, istenen terimdir. `fibK(10, 0, 1)` 55 döndürür ve yalnız 11 çağrı yapar. Ağaç özyinelemeli sürümle karşılaştırıldığında fark çarpıcıdır: aynı bağıntı, sonucun nasıl taşındığına göre 177 çağrıyla ya da 11 çağrıyla hesaplanır. Döngülü sürüm ile kuyruk özyinelemeli sürüm satır satır eşleşir. Döngü değişkenleri parametrelere, döngü koşulu taban durumun tersine, bir tur da bir çağrıya karşılık gelir.

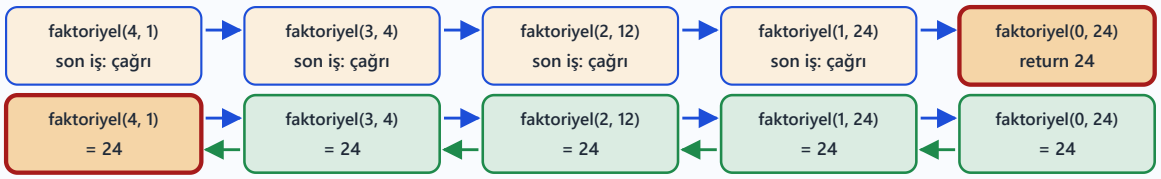
3.5.7.3 faktoriyel(4, 1) çağrısının izlenmesi

```
static int faktoriyel(int n, int birikim) {
    if (n == 0) {
        return birikim;
    } else {
        return faktoriyel(n - 1, n * birikim);
    }
}
```

Çağrı	n	birikim	Çağrıdan önceki iş	Dönüşte yapılan iş
1	4	1	4 * 1 = 4 → `faktoriyel(3, 4)`	24'ü aynen döndürür
2	3	4	3 * 4 = 12 → `faktoriyel(2, 12)`	24'ü aynen döndürür
3	2	12	2 * 12 = 24 → `faktoriyel(1, 24)`	24'ü aynen döndürür
4	1	24	1 * 24 = 24 → `faktoriyel(0, 24)`	24'ü aynen döndürür
5	0	24	taban durum	24 döner

Tablo 3.5.16: `faktoriyel(4, 1)` çağrısının izlenmesi.

Bütün çarpmalar inişte yapılır (Tablo 3.5.16). Taban duruma varıldığında sonuç zaten `birikim`'dedir. Dönüşte hiçbir hesap yapılmaz; 24 aynen yukarı aktarılır. Sıradan faktöriyelle karşılaştırıldığında çarpmaların sırası da tersine dönmüştür: burada 4, 3, 2, 1 sırasıyla çarpılır.



Şekil 3.5.16: `faktoriyel(4, 1)` çağrı zinciri. Üstte taban durum: sonuç `birikim`'de hazırdır. Altta bütün çerçeveler 24'ü değiştirmeden döndürmüştür.

Şekil 3.5.16 önemli bir gerçeği gösterir: Java kuyruk çağrısını optimize etmediği için beş `faktoriyel` çerçevesinin hepsi yine de yığına girer. Kuyruk çağrısı optimizasyonu yapan bir dilde tek çerçeve yeterdi. Aynı metodun Java'daki döngü karşılığı şöyledir: `while (n != 0) { birikim = n * birikim; n = n - 1; } return birikim;`. Parametreler değişken olur, özyinelemeli çağrı ise bu değişkenlerin güncellenmesine dönüşür.

3.5.8 Bellekleme

Ağaç özyinelemesindeki tekrarı gidermenin bir yolu döngüye geçmekti. İkinci yol, özyinelemeli yapıyı koruyup metoda bir bellek vermektir. Metot bir değeri ilk kez hesapladığında onu bir yere yazar; aynı değer yeniden istenirse hesaplamadan okur.

> **Tanım 3.5.12**
>
> **Bellekleme** (*memoization*), bir metodun hesapladığı sonuçları saklayıp aynı argüman yeniden geldiğinde hesaplamadan döndürmesidir. Sonuçların saklandığı yapıya **önbellek** (*cache*) denir. Metot böylece çağrılar arasında durum bilgisi taşır.

3.5.8.1 Belleklemeli Fibonacci

Java'da önbellek için en kolay araç `Map`'tir. `Map`, anahtarları değerlere eşleyen bir veri yapısıdır; burada anahtar `n`, değer F(n)'dir. `Map<Integer, Long>` yazımı, anahtarların `Integer`, değerlerin `Long` olduğunu belirtir. `Map` ilkel tipleri tutamadığı için `int` ve `long` yerine bunların sarmalayıcı sınıfları kullanılır; Java dönüşümü kendiliğinden yapar.

```
import java.util.HashMap;
import java.util.Map;

static Map<Integer, Long> hafiza = new HashMap<>();

static long fibonacci(int n) {
    if (hafiza.containsKey(n)) {
        return hafiza.get(n);    // daha önce hesaplandı
    }
    if (n == 0 || n == 1) {
        return n;
    }
    long sonuc = fibonacci(n - 1) + fibonacci(n - 2);
    hafiza.put(n, sonuc);        // sonucu sakla
    return sonuc;
}
```

Metodun gövdesi dört adımdan oluşur. Önce belleğe bakılır: `hafiza.containsKey(n)`, `n` anahtarı daha önce yazılmışsa `true` döndürür ve değer `hafiza.get(n)` ile okunup döndürülür. Bellekte yoksa taban durum sınanır. İkisi de değilse değer her zamanki gibi iki özyinelemeli çağrıyla hesaplanır. Son olarak `hafiza.put(n, sonuc)` sonucu saklar ve sonuç döndürülür.

`hafiza` bir alan (*field*) olarak, metodun dışında ve `static` tanımlanmıştır. Yerel değişken olsaydı her çağrıda yeniden ve boş olarak oluşturulur, hiçbir şey hatırlanmazdı. Bütün çağrıların aynı belleği paylaşması için onu metodun dışında tutmak gerekir. Dönüş tipi de `long`'a çevrilmiştir. Bellekleme

büyük n'leri hızla hesaplanabilir kıldığı için `int` artık yetmez: `fibonacci(90)` anında 2880067194370816120 döndürür. `long`'a sığan en büyük Fibonacci sayısı F(92) = 7540113804746346429'dur.

Anahtarlar 0'dan başlayan ardışık tam sayılar olduğu için `Map` yerine bir dizi de kullanılabilir:

```
static long[] tablo = new long[93];

static long fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    if (tablo[n] != 0) {
        return tablo[n];
    }
    tablo[n] = fibonacci(n - 1) + fibonacci(n - 2);
    return tablo[n];
}
```

Dizinin boyu 93'tür, çünkü `long`'a sığan son terim F(92)'dir. Java yeni bir `long` dizisinin bütün öğelerini 0 ile doldurur. Bu sürümde 0, "henüz hesaplanmadı" anlamına gelir. Bu kabul güvenlidir, çünkü n ≥ 2 için hiçbir Fibonacci sayısı 0 değildir. Dizi, `Map`'e göre daha az bellek kullanır ve daha hızlıdır. Sarmalayıcı nesneler oluşturulmaz, anahtar araması da tek bir indis erişimine iner. Bedeli esnekliktir: dizinin boyu önceden bilinmelidir ve anahtarlar küçük, negatif olmayan tam sayılar olmalıdır. `fibonacci(90)` bu sürümde de 2880067194370816120 döndürür.

3.5.8.2 Belleklemeli fibonacci(4) izleme

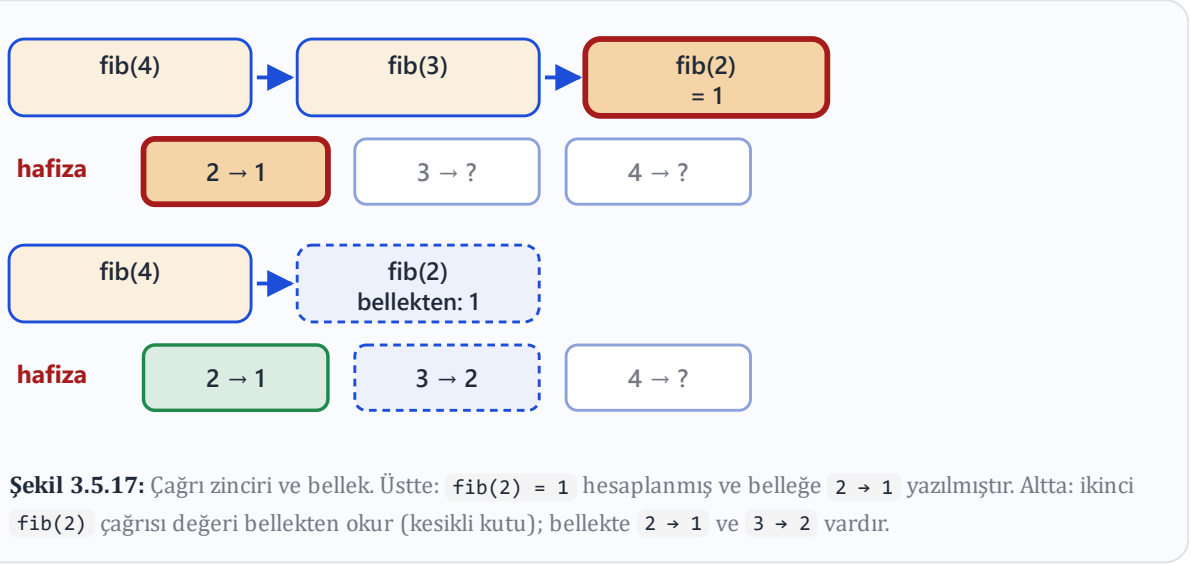
```
static Map<Integer, Long> hafiza = new HashMap<>();
static long fibonacci(int n) {
    if (hafiza.containsKey(n)) {
        return hafiza.get(n);
    }
    if (n == 0 || n == 1) {
        return n;
    }
    long sonuc = fibonacci(n - 1) + fibonacci(n - 2);
    hafiza.put(n, sonuc);
    return sonuc;
}
```

Çağrı	Argüman	Olay	Döner	`hafiza` (sonra)
1	4	bellekte yok, taban durum değil → `fibonacci(3)`	3	{}
2	3	bellekte yok → `fibonacci(2)`	2	{}

Çağrı	Argüman	Olay	Döner	`hafiza` (sonra)
3	2	bellekte yok → `fibonacci(1)`	1	{}
4	1	taban durum	1	{}
5	0	taban durum; `fibonacci(2) = 1 + 0` saklanır	0	{2=1}
6	1	taban durum; `fibonacci(3) = 1 + 1` saklanır	1	{2=1, 3=2}
7	2	**bellekte var**, hesaplanmadan döner; `fibonacci(4) = 2 + 1` saklanır	1	{2=1, 3=2, 4=3}

Tablo 3.5.17: Bellek boşken `fibonacci(4)` çağrısının izlenmesi.

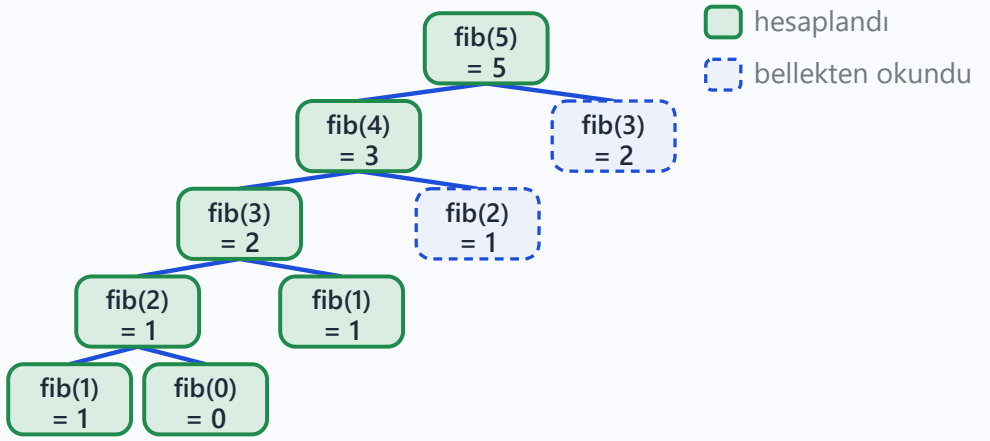
İlk altı çağrı belleklemesiz sürümle aynıdır (Tablo 3.5.17). Fark yedinci çağrıda ortaya çıkar. `fibonacci(4)`'ün sağ terimi `fibonacci(2)`'dir ve bu değer beşinci çağrının sonunda belleğe yazılmıştır. Metot ilk satırında onu bulur ve 1'i hemen döndürür. Belleklemesiz sürüm burada `fib(2)`'nin altında iki çağrı daha yapardı: `fib(1)` ve `fib(0)`. Toplam çağrı sayısı 9'dan 7'ye iner. Taban durum değerleri belleğe yazılmaz; onlar zaten hesap gerektirmez.



Şekil 3.5.17: Çağrı zinciri ve bellek. Üstte: `fib(2) = 1` hesaplanmış ve belleğe `2 → 1` yazılmıştır. Altta: ikinci `fib(2)` çağrısı değeri bellekten okur (kesikli kutu); bellekte `2 → 1` ve `3 → 2` vardır.

3.5.8.3 Çağrı ağacı ve kazanç

Belleklemenin etkisi n büyüdükçe çarpıcı hâle gelir. Şekil 3.5.18, belleklemeli `fibonacci(5)`'in çağrı ağacını gösterir. Sol kol 5'ten 0'a kadar bir kez hesaplanır. Sağdaki `fib(2)` ve `fib(3)` bellekten gelir ve altlarında yeni dal açılmaz. 15 yerine 9 çağrı yapılır.



Şekil 3.5.18: Belleklemeli `fibonacci(5)` çağrı ağacı: 9 çağrı. Kesikli düğümler bellekten okunan değerlerdir; ağaç bir zincire ve kısa yan dallara dönüşür.

Genel olarak bellek boşken `fibonacci(n)`, $n \geq 1$ için tam $2n - 1$ çağrı yapar. Sol kol `n`'den 1'e inerken n çağrı yapılır. 2'den n'ye kadar her değer için sağ terim bir çağrı daha ister; bunlar $n - 1$ çağrıdır. Toplam $n + (n - 1) = 2n - 1$'dir. Üstel büyüme doğrusala inmiştir. Tablo 3.5.18 iki sürümü ölçümlerle karşılaştırır.

n	Belleklemesiz çağrı	Belleklemeli çağrı	Belleklemesiz süre	Belleklemeli süre
5	15	9	< 0,1 ms	< 0,1 ms
20	21.891	39	≈ 0,3 ms	< 0,1 ms
40	331.160.281	79	≈ 0,5 s	< 1 ms
90	$\approx 9 \cdot 10^{18}$	179	yüzyıllar	< 1 ms

Tablo 3.5.18: Belleklemesiz ve belleklemeli Fibonacci (JDK 21 ile ölçülen süreler).

Süreler makineye göre değişir, oran değişmez. Son satırdaki belleklemesiz çağrı sayısı $2F(91) - 1$'dir; saniyede bir milyar çağrıyla bile yaklaşık 300 yıl sürerdi. Belleklemeli sürüm aynı sonucu 179 çağrıda verir.

Bu kazancın bir bedeli vardır: bellek. Saklanan sonuçlar n değeri kadar kayıt tutar ve her kayıt bir `Integer` ile bir `Long` nesnesi içerir. Bellekleme, hız için bellek harcamaktır. Fibonacci'de bu takas açıkça kazançlıdır; bazı problemlerde ise saklanacak sonuç sayısı belleğe sığmayacak kadar çok olabilir. Bir başka sınır yığındır: belleklemeli sürümün sol kolu hâlâ n çerçeve derinliğe iner. `fibonacci(100000)` belleklemeyle de `StackOverflowError` verir, üstelik sonucu `long`'a sığmaz.

3.5.8.4 Belleklemeli faktöriyel

Aynı kalıp her özyinelemeli metoda uygulanabilir: önce belleğe bak, yoksa hesapla, sakla, döndür.

```
static Map<Integer, Long> hafiza = new HashMap<>();

static long faktoriyel(int n) {
    if (hafiza.containsKey(n)) {
        return hafiza.get(n);
    }
    if (n == 0 || n == 1) {
        return 1;
    }
    long sonuc = n * faktoriyel(n - 1);
    hafiza.put(n, sonuc);
    return sonuc;
}
```

`long` sayesinde 20!'e kadar doğru sonuç alınır: `faktoriyel(20)` = 2432902008176640000. `faktoriyel(21)` ise taşar ve −4249290049419214848 döndürür; taşmış değer de belleğe yazılır.

Burada ilginç bir durum vardır. Faktöriyel ağaç özyinelemesi değildir; tek bir `faktoriyel(n)` çağrısında hiçbir değer iki kez hesaplanmaz. Bu yüzden bellek tek bir çağrıyı hızlandırmaz, hatta `Map` işlemleri yüzünden biraz yavaşlatır. Kazanç, program birçok kez faktöriyel istediğinde ortaya çıkar. `hafiza` statik olduğu için çağrılar arasında korunur. `faktoriyel(10)` çağrıldıktan sonra 1!'den 10!'e kadar değerler bellektedir. Ardından `faktoriyel(12)` istenirse yalnız 12 × 11! ve 11 × 10! diye iki yeni çarpma yapılır; 10! bellekten okunur. Kombinasyon gibi faktöriyeli defalarca kullanan hesaplarda bu kazanç anlamlıdır.

Sık yapılan hata

Belleğe bakmayı ya da sonucu yazmayı unutmak. `hafiza.put` satırı yazılmazsa bellek hep boş kalır ve metot belleklemesiz sürüm gibi çalışır. `containsKey` denetimi taban durumdan sonra ve özyinelemeli çağrıdan sonra yazılırsa da bellek hiç işe yaramaz. Doğru sıra: belleğe bak, taban durum, hesapla, sakla, döndür.

3.5.9 Sık yapılan hatalar

Özyinelemeli kodlardaki hataların çoğu birkaç kalıba girer. Bazıları derleme sırasında, bazıları çalışma sırasında ortaya çıkar; en sinsileri ise hiçbir ileti vermez. Tablo 3.5.19 bu kalıpları JDK 21'in gerçek iletileriyle toplar.

Hata	Örnek	Belirti	Çözüm
Taban durum yok ya da ulaşılamıyor	`toplam(0)`, taban durum `n == 1`	`Exception in thread "main" java.lang.StackOverflowError`	Taban durumu genişletmek: `n <= 0`
İlerleme yok	`return n * faktoriyel(n);`	`StackOverflowError`	Argümanı küçültmek: `n - 1`

Hata	Örnek	Belirti	Çözüm
İlerleme taban durumu atlıyor	`if (n == 0) return 0;` `return f(n - 3);`, `f(7)`	`StackOverflowError` (7, 4, 1, −2, …)	`n <= 0` gibi aralık sınayan taban durum
Eksik `return`	`if … else if` ile biten metot	`error: missing return statement`	Son dalı `else` yapmak
Çağrının sonucunu kullanmamak	`faktoriyel(n - 1);` tek başına	İleti yok; değer kaybolur	`return n * faktoriyel(n - 1);`
Taşma	`int` faktöriyel, Bell katsayısı	İleti yok; 13!'te yanlış sonuç	`long` ya da `BigInteger`
Gereksiz ağaç özyinelemesi	Belleklemesiz `fibonacci(50)`	Dakikalar süren çalışma	Döngü ya da bellekleme

Tablo 3.5.19: Özyinelemede sık yapılan hatalar.

Eksik `return` hatası, Java derleyicisinin koşulları değerlendirmemesinden doğar. Aşağıdaki metot bir sayının en büyük rakamını bulmak için yazılmıştır:

```
static int enBuyuk(int n) {
    if (n < 10) {
        return n;
    } else if (n % 10 > enBuyuk(n / 10)) {
        return n % 10;
    } else if (n % 10 <= enBuyuk(n / 10)) {
        return enBuyuk(n / 10);
    }
}
```

```
error: missing return statement
    }
    ^
```

İki `else if` koşulu birbirinin tersidir ve biri mutlaka doğrudur. Ama derleyici bunu anlamaz; koşulların değerini derleme sırasında hesaplamaz. Gördüğü şey, üç koşulun da yanlış olabileceği ve o durumda metodun değer döndürmeden biteceğidir. Son dal `else` yapılınca hata kaybolur. Aynı metotta ikinci bir sorun daha vardır: `enBuyuk(n / 10)` bir çağrıda üç kez hesaplanabilir. Bu da çağrı sayısını her basamakta üçe katlar. Doğru yazım, alt problemin sonucunu bir kez hesaplayıp bir değişkende tutmaktır: `int geri = enBuyuk(n / 10); return Math.max(n % 10, geri);`. Bu sürüm 48729 için 9 döndürür ve beş çağrı yapar.

Çağrının sonucunu kullanmamak, özellikle yeni başlayanlarda görülür. `faktoriyel(n - 1);` satırı geçerli bir Java deyimidir: metot çağrılır, değer hesaplanır ve sonra atılır. Derleyici bunun için uyarı vermez. Özyinelemeli çağrının döndürdüğü değer, ya `return` ifadesinin içinde kullanılmalı ya da bir değişkene yazılmalıdır.

Taşma ve gereksiz ağaç özyinelemesi en sinsi iki hatadır, çünkü hiçbiri ileti vermez. Program çalışır ve bir sonuç üretir. Taşma yanlış bir sayı verir. Ağaç özyinelemesi ise doğru sonucu çok geç verir. İkisini de önlemenin yolu, metodu yazarken girdinin ne kadar büyüyebileceğini düşünmek ve sınır değerlerle denemektir.

3.5.10 Özet

- **Özyineleme:** bir metodun kendini daha küçük bir girdiyle çağırmasıdır. Java'da özel bir anahtar sözcük gerekmez; metot gövdesinde kendi adını çağırır.
- **Taban durum ve ilerleme:** taban durum, doğrudan çözülen en küçük durumdur ve özyinelemeyi durdurur. Her çağrı taban duruma yaklaşmalıdır (`n - 1` , `n / 10` , `substring` , `x % y`). İki kuraldan biri bozulursa `StackOverflowError` alınır.
- **Çağrı yığını:** her çağrıya bir çerçeve açılır; çerçevenin kendi parametreleri ve yerel değişkenleri vardır. Çerçeveler LIFO sırasıyla kapanır. Varsayılan yığınla derinlik birkaç on bin çağrıyı geçemez; sınır JVM ayarına göre değişir.
- **İniş ve dönüş:** çağrıdan önce yapılan iş inişte, sonra yapılan iş dönüşte yürür. Dönüşte yazdıran metot sırayı ters çevirir (`yaz2` , `ikili`).
- **Seriler:** yineleme bağıntısı özyinelemeli metoda doğrudan çevrilir. Fibonacci ve Lucas kendini iki kez çağırır; Catalan ve Bell çağrıları döngü içinde yapar. `int` 12!'ye, `long` 20!'ye kadar yeter; taşma sessizdir.
- **Örnekler:** rakamlar toplamı basamak basamak, palindrom iki uçtan, Öklid algoritması kalanla küçülür. Hanoi kuleleri $2^n - 1$ hamle yapar ve özyinelemeyle birkaç satırdır.
- **Ağaç özyinelemesi:** birden çok çağrı (Fibonacci, Hanoi) ağaç oluşturur; tekrar varsa çağrı sayısı üstel büyür. Belleklemesiz `fibonacci(40)` 331.160.281 çağrı yapar.
- **İlerlemeli çözüm:** döngü ve biriktirici değişkenle tek çerçevede, sabit bellekle çalışır ve genellikle daha hızlıdır. Her özyinelemeli çözüm döngüyle yazılabilir.
- **Kuyruk özyinelemesi:** son iş çağrının kendisidir ve sonuç biriktirici parametrede taşınır. Biriktiricinin başlangıcı etkisiz elemandır ve bir sarmalayıcı metotla gizlenir. Java kuyruk çağrısı optimizasyonu yapmaz.
- **Bellekleme:** hesaplanan sonuçlar bir `Map` 'te saklanıp yeniden kullanılır. Belleklemeli Fibonacci 2n − 1 çağrı yapar; bedeli bellektir.

3.5.11 Kavram listesi

özyineleme *recursion* · taban durum *base case* · özyinelemeli adım *recursive step* · çağrı yığını *call stack* · yığın çerçevesi *stack frame* · yığın taşması *stack overflow* · yığın izi *stack trace* · ağaç özyinelemesi *tree recursion* · yineleme bağıntısı *recurrence relation* · inanç sıçraması *leap of faith*

ilerlemeli çözüm *iterative solution* kuyruk özyinelemesi *tail recursion* biriktirici *accumulator* kuyruk çağrısı optimizasyonu *tail call optimization* bellekleme *memoization* Öklid algoritması *Euclidean algorithm* palindrom *palindrome*

3.5.12 Kendini Dene

1. Aşağıdaki metot için `f(6)` kaçtır? Metot kaç kez çağrılır (ilk çağrı dahil)? Metot ne hesaplar?

```
static int f(int n) {
    if (n <= 0) return 0;
    return (n % 2 == 0 ? n : 0) + f(n - 1);
}
```

Cevap

`f(6)` = 6 + 0 + 4 + 0 + 2 + 0 + 0 = 12. Argümanlar 6, 5, 4, 3, 2, 1, 0'dır; 7 çağrı yapılır. Metot 1'den n'ye kadar çift sayıların toplamını hesaplar. Taban durum `n <= 0` olduğu için negatif girdide de durur.

2. `g(3)` ne yazdırır?

```
static void g(int n) {
    if (n == 0) return;
    System.out.print(n);
    g(n - 1);
    System.out.print(n);
}
```

Cevap

`321123`. İlk `print` inişte çalışır ve 3, 2, 1 yazar. İkinci `print` dönüşte çalışır; en derindeki çerçeve önce döndüğü için 1, 2, 3 yazılır.

3. Aşağıdaki üs metodu `us(2, 10)` için hangi değeri döndürür ve kaç çağrı yapar? Üssü bir azaltan

`taban * us(taban, us - 1)` sürümü kaç çağrı yapardı?

```
static long us(long t, int u) {
    if (u == 0) return 1;
    long y = us(t, u / 2);
    return u % 2 == 0 ? y * y : y * y * t;
}
```

Cevap

1024 döner; üsler 10, 5, 2, 1, 0 olduğu için 5 çağrı yapılır. `us(2, 5)` = 32 (tek üs: 4 · 4 · 2), sonra 32 · 32 = 1024. Üssü bir azaltan sürüm 11 çağrı yapardı. Üssü yarıya indirmek çağrı sayısını O(u)'dan O(log u)'ya düşürür. Alt sonucun `y` 'de tutulması önemlidir; `us(t, u / 2) * us(t, u / 2)` yazılsaydı kazanç kaybolurdu.

4. `basamakSayisi` metodu, biriktirici kullanan kuyruk özyinelemeli bir `bs(int n, int k)` metodu olarak nasıl yazılır? `bs(48729, 0)` kaç döndürmelidir?

Cevap

```
static int bs(int n, int k) {
    if (n < 10) return k + 1;
    return bs(n / 10, k + 1);
}
```

`k`, o ana kadar atılan basamakları sayar ve 0 ile başlar. (48729, 0) → (4872, 1) → (487, 2) → (48, 3) → (4, 4) → 4 + 1 = 5. Son iş çağrı olduğu için kuyruk özyinelemelidir.

5. `static int h(int n) { if (n == 0) return 1; return h(n - 1) + h(n - 1); }` için `h(4)` kaçtır ve kaç çağrı yapılır? Son satır `return 2 * h(n - 1);` yapılırsa çağrı sayısı ne olur?

Cevap

`h(4)` = 2^4 = 16. Her çağrı iki çağrı açtığı için ağaçta 1 + 2 + 4 + 8 + 16 = 31 çağrı vardır. `2 * h(n - 1)` ile aynı sonuç 5 çağrıda bulunur. Aynı alt problemi iki kez çağırmak, sonucu değiştirmeden maliyeti üstele çıkarır.

6. `gcd(89, 55)` kaç çağrı yapar ve sonucu nedir? En derin anda yığında `main` dahil kaç çerçeve var-

dır?

Cevap

EBOB 1'dir. Zincir (89, 55) → (55, 34) → (34, 21) → (21, 13) → (13, 8) → (8, 5) → (5, 3) → (3, 2) → (2, 1) → (1, 0)'dır; 10 çağrı yapılır. Hepsi aynı anda yığındadır, çünkü Java kuyruk çağrılarını optimize etmez: `main` ile 11 çerçeve. Ardışık Fibonacci sayıları, her adımda kalanın en yavaş küçüldüğü durumdur.

7. Aşağıdaki metot `say("kelebek", 'e')` için ne döndürür? Kaç çağrı yapılır? Taban durum hangisidir?

```
static int say(String s, char c) {
    if (s.length() == 0) return 0;
    int k = s.charAt(0) == c ? 1 : 0;
    return k + say(s.substring(1), c);
}
```

Cevap

3 döner: “kelebek”te üç 'e' vardır. Dizgi her çağrıda bir karakter kısalır; yedi harf için 7 çağrı ve boş dizgi için 1 çağrı, toplam 8 çağrı yapılır. Taban durum boş dizgidir ve 0 döndürür.

8. 5 diskli Hanoi kuleleri `coz(5, 'A', 'C', 'B')` ile çözülüyor. Kaç hamle yazdırılır ve ilk hamle nedir? En büyük disk kaçıncı hamlede taşınır?

Cevap

$2^5 - 1 = 31$ hamle. Disk sayısı tek olduğu için ilk hamle `1: A->C` 'dir. En büyük disk, üstteki 4 diskin $2^4 - 1 = 15$ hamlede B'ye çekilmesinden sonra, 16. hamlede A'dan C'ye taşınır.

3.5.13 İleri okuma

- A. B. Downey ve C. Mayfield, *Think Java* (2. baskı, 2019), Bölüm 8: özyinelemeli metotlar yığın diyagramlarıyla ve “inanç sıçraması” kavramıyla anlatılır. Kitap çevrim içi ücretsiz okunabilir.
- R. Sedgewick ve K. Wayne, *Computer Science: An Interdisciplinary Approach*, Kesim 2.3: Öklid, Hanoi, Gray kodları ve özyinelemeli çizimlerle özyinelemenin gücü; üstel Fibonacci tuzağı ve dinamik programlama ile çaresi işlenir.
- P. Deitel ve H. Deitel, *Java: How to Program*, Bölüm 18: özyinelemeyi yineleme ile karşılaştırır ve fraktal çizimleri, sekiz vezir ve labirent gibi ek örnekler verir.

- H. Abelson ve G. J. Sussman, *Structure and Interpretation of Computer Programs* (2. baskı, MIT Press, 1996), Kesim 1.2: doğrusal ve ağaç özyinelemesi, ilerlemeli süreç ve kuyruk çağrısı kavramlarının klasik anlatımı.

3.5.14 Kaynaklar

1. P. Deitel, H. Deitel, *Java: How to Program (Early Objects)*, 11. baskı, Pearson, 2017 (Bölüm 18: Recursion).
2. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019 (Bölüm 8: Recursive Methods).
3. J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, D. Smith, G. Bierman, *The Java Language Specification, Java SE 21 Edition*, Oracle, 2023 (15.12: Method Invocation Expressions).
4. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson, 2022.
5. Oracle, *The Java Tutorials: Defining Methods; Returning a Value from a Method*, docs.oracle.com/javase/tutorial/java/javaOO/.
6. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016 (2.3: Recursion).
7. H. Abelson, G. J. Sussman, J. Sussman, *Structure and Interpretation of Computer Programs*, 2. baskı, MIT Press, 1996.

KISIM II

Nesne Yönelimli Programlama

Bölüm 4–5

KISIM II · BÖLÜM 4

Nesneler ve Sınıflar, Kalıtım ve Çok Biçimlilik

Nesne yönelimli programlama, veriyi ve bu veri üzerinde çalışan davranışı sınıflar içinde bir araya getirir. Bölüm sınıf ve nesne, yapıcılar, static üyeler, this referansı, kayıtlar, enum tipleri ve iç sınıflarla başlar; sarmalama, erişim belirleyiciler, değiştirilemez sınıflar ve paketlerle sürer. Kalıtımda super, geçersiz kılma ve Object sınıfı, çok biçimlilikte ise referans tipleri, tip dönüşümü, dinamik bağlama ve aşırı yükleme ele alınır. Son kısım soyut sınıfları, arayüzleri, lambda ifadelerini, mühürlü tipleri, jenerikleri, etiketleri ve yaygın tasarım desenlerini kapsar.

Bu bölümün kodu: `github.com/sercankulcu/object-oriented-programming-java/Bolum04/src`

BU BÖLÜMÜN NOTLARI

4.1 Nesneler ve Sınıflar

4.2 Sarmalama

4.3 Kalıtım

4.4 Çok Biçimlilik

4.5 Soyut Sınıflar ve Arayüzler

KISIM II · BÖLÜM 4.1

Nesneler ve Sınıflar

Şimdiye kadarki programlar `static` metotlardan ve bu metotlara aktarılan değişkenlerden oluşuyordu. Veri bir yerde, onu işleyen kod başka bir yerde duruyordu. Nesne yönelimli programlama bu ikisini bir araya getirir: bir öğrencinin adı, numarası ve not ortalaması ile bu verilerle çalışan metotlar aynı birimin içinde yaşar. Bu birimin taslağı **sınıftır**; taslaktan üretilen her somut örnek bir **nesnedir**. Not, bir sınıfın nasıl yazıldığını, `new` ile nesne oluşturulurken bellekte neler olduğunu ve değişkenlerin nesneyi değil, nesneye giden referansı tuttuğunu adım adım gösterir. Örneğe ait ve `static` üyeler, ilklendirme sırası, `this` ve yapıcı zinciri, `==` ile `equals` farkı, sarmalayıcı sınıflar, kayıtlar, `enum` ve iç sınıflar da notun konuları arasındadır. Not, bir kesir sınıfını baştan tasarlayarak biter. Bütün programlar JDK 21 ile derlenip çalıştırılmıştır; çıktılar, izleme tablolarındaki son değerler ve derleyici iletileri bu çalıştırmalardan alınmıştır.

ÖĞRENME ÇIKTILARI

Sınıf ve nesne
Sınıfı bir taslak, nesneyi o taslaktan üretilmiş somut bir örnek olarak açıklamak; alanları ve metotlarıyla bir sınıf yazmak.

Nesne oluşturmak
`new` işlecini ve yapıcıları kullanmak; alanların varsayılan değerlerini ve derleyicinin eklediği varsayılan yapıcıyı bilmek.

Örnek ve sınıf üyeleri
Örneğe ait üyelerle `static` üyeleri ayırt etmek; ilklendirici bloklarının ve yapıcıların çalışma sırasını izlemek.

Referanslar
Değişkenin nesneyi değil referansı tuttuğunu bellek çizimiyle göstermek; `this`, `==` ve `equals` farkını açıklamak.

Yapıcı zinciri ve erişim
`this(...)` ile yapıcıları zincirlemek; erişim belirleyicilerin ve veri gizlemenin amacını açıklamak.

Özel sınıf türleri
Sarmalayıcı sınıfları, kayıtları, `enum` türünü ve iç sınıfları tanımak; `RasyonelSayi` örneğiyle bir sınıfı baştan tasarlamak.

4.1.1 Nesne yönelimli programlama

Bir öğrenci bilgi sistemini `static` metotlarla yazdığımızı düşünelim. Her öğrencinin adı bir dizide, numarası ikinci bir dizide, ortalaması üçüncü bir dizide tutulur. `ortalamaHesapla`, `derseKaydet` ve `transkriptYaz` gibi metotlar bu dizileri parametre olarak alır. Program büyüdükçe sorunlar belirir. Diziler birbirinden ayrı olduğu için 5. öğrencinin adı ile 5. öğrencinin ortalaması arasındaki bağı yalnızca indis kurar. Bir dizi sıralanıp öteki unutulursa adlar başka öğrencilerin notlarıyla eşleşir. Bir öğrenciye yeni bir bilgi eklemek, örneğin bölümünü tutmak, dördüncü bir dizi ve onu taşıyan yeni parametreler demektir. Bu dizilere dokunan her metodun değişmesi gerekir.

Nesne yönelimli programlama bu dağınıklığa başka bir düzenle cevap verir. Bir öğrencinin bütün bilgileri tek bir birimde toplanır. Bu bilgilerle çalışan kod da aynı birimin içine yazılır. Program, birbirine iş yaptıran bu birimlerden kurulur.

Tanım 4.1.1

Nesne yönelimli programlama (*object-oriented programming*, OOP), programı birbiriyle etkileşen nesnelerden kuran programlama yaklaşımıdır. Her nesne kendi verisini taşır ve bu veri üzerinde çalışan işlemleri sunar. Nesneler birbirinin metotlarını çağırarak birlikte çalışır.

4.1.1.1 Nesne: durum ve davranış

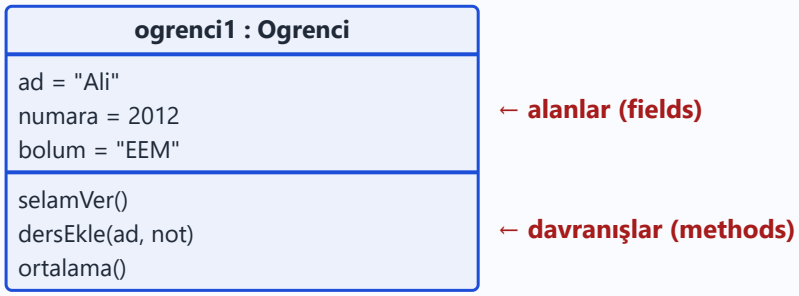
Gerçek dünyada açıkça ayırt edilebilen her varlık bir nesne olarak modellenebilir. Bir öğrenci, bir masa, bir çember, ekrandaki bir düğme, hatta bankadan çekilen bir kredi bile bir nesnedir. Bu nesnelerin her birinde iki şey bulunur. Birincisi nesnenin **durumudur**: o an taşıdığı veriler. Öğrencinin adı, numarası ve bölümü onun durumunu oluşturur. İkincisi nesnenin **davranışıdır**: nesnenin yapabildiği işler. Öğrenci bir derse kaydolabilir, notlarını görebilir. Tablo 4.1.1 beş nesnenin durum ve davranışlarını yan yana verir.

Nesne	Alanlar (durum)	Davranışlar
Öğrenci	ad, numara, bölüm	`derseKaydol()`, `notGor()`
Masa	en, boy, malzeme	`tasi()`
Çember	merkez, yarıçap	`alan()`, `cevre()`
Düğme	yazı, renk, etkin mi	`tikla()`

Nesne	Alanlar (durum)	Davranışlar
Kredi	tutar, faiz, vade	`taksitHesapla()`

Tablo 4.1.1: Gerçek dünyadan nesneler. Durum alanlarla, davranış metotlarla temsil edilir.

Programlama dilinde durum ve davranışın karşılıkları bellidir. Nesnenin sakladığı veri parçalarına **alan** (*field*), nesnenin yapabildiği işlere **metot** (*method*) denir. Nesneler birbirinin metotlarını çağırarak iş görür. Bir okul programında öğrenci, öğretmen ve ders nesneleri birbirine böyle "mesaj gönderir". Öğretmen nesnesi bir ders nesnesinin `notGir` metodunu çağırır; ders nesnesi de öğrencinin kaydını günceller. Mesaj göndermek, Java'da bir nesnenin metodunu çağırmak demektir.



Şekil 4.1.1: `ogrenci1` nesnesi. Üst bölümde alanlar ve o anki değerleri, alt bölümde nesnenin metotları vardır. Veri ve onu kullanan kod aynı kutunun içindedir.

Şekil 4.1.1, `Ogrenci` tipindeki `ogrenci1` nesnesini çizer. Kutunun üst bölümünde `ad`, `numara` ve `bolum` alanları ile değerleri, alt bölümünde ise `selamVer()`, `dersEkle(ad, not)` ve `ortalama()` metotları bulunur. `ortalama()` metodu bir dizi parametresi almaz; ihtiyaç duyduğu notlar zaten nesnenin içindedir. Veriyle kodun bu birlikteliği programı daha düzenli kılar. Kod yeniden kullanılabilir hâle gelir ve gerçek dünyadaki karşılığına benzediği için daha kolay anlaşılır.

4.1.1.2 Prosedürel ve nesne yönelimli yaklaşım

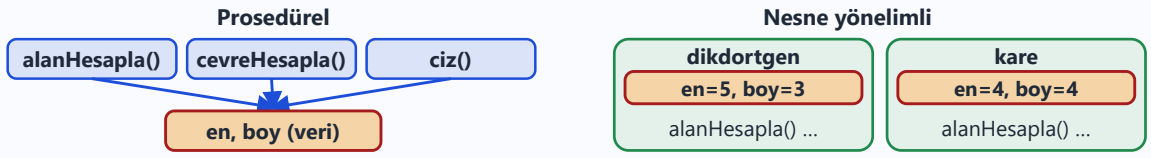
Şimdiye kadarki programlar **prosedürel** (yordamsal, *procedural*) tarzda yazıldı. Bu tarzda temel birim fonksiyondur. Veri ayrı değişkenlerde durur ve onu isteyen her fonksiyona parametre olarak verilir. Nesne yönelimli tarzda temel birim nesnedir; veri nesnenin içinde saklanır ve dışarıya yalnızca nesnenin izin verdiği yollarla açılır. Tablo 4.1.2 iki yaklaşımı beş açıdan karşılaştırır.

	Prosedürel yaklaşım	Nesne yönelimli yaklaşım
Temel birim	Fonksiyon (metot)	Nesne (veri + metotlar)
Veri	Ayrı değişkenlerde; her fonksiyon erişebilir	Nesnenin içinde; dışarıya kontrollü açılır
Örnek	`alanHesapla(en, boy)`	`dikdortgen.alanHesapla()`
Büyük programlar	Değişiklik birçok fonksiyonu etkiler	Değişiklik çoğunlukla bir sınıfın içinde kalır

	Prosedürel yaklaşım	Nesne yönelimli yaklaşım
Yeniden kullanım	Fonksiyonları kopyalayarak	Sınıfları kullanarak ve kalıtımla

Tablo 4.1.2: Prosedürel ve nesne yönelimli yaklaşımın karşılaştırması.

Tablonun "Örnek" satırı farkı en kısa yoldan gösterir. Prosedürel çağrıda `alanHesapla` metodu dikdörtgenin enini ve boyunu parametre olarak bekler. Çağıran, bu iki sayıyı bir yerde saklamak ve doğru sırayla vermek zorundadır. Nesne yönelimli çağrıda en ve boy dikdörtgen nesnesinin içindedir. Çağıran yalnızca hangi dikdörtgenin alanını istediğini söyler: `dikdortgen.alanHesapla()`.



Şekil 4.1.2: Solda prosedürel düzen: üç fonksiyon aynı ortak veriye dışarıdan erişir. Sağda nesne yönelimli düzen: her nesne kendi verisini (`en` , `boy`) ve bu veriyle çalışan metotları birlikte taşır.

"Büyük programlar" satırı ise asıl kazancı söyler. Şekil 4.1.2'nin solunda üç fonksiyon aynı veriye bağlıdır. Verinin biçimi değişirse, örneğin en ve boy yerine köşe koordinatları tutulmaya başlanırsa, üç fonksiyonun da değişmesi gerekir. Sağdaki düzende veri bir sınıfın içine kapanmıştır. Biçim değişikliği o sınıfın metotlarını etkiler; sınıfı kullanan kod `alanHesapla()` çağrısını aynen yazmaya devam eder.

Java'da her kod bir sınıfın içinde yazılır. Bu yüzden şimdiye kadarki programların da bir sınıfı vardı, ama o sınıflar yalnızca `static` metotları bir arada tutan kaplardı; onlardan hiç nesne üretilmedi. Nesne yönelimli programlamada sınıflar asıl görevini yapar: nesne üretir.

4.1.1.3 Üç temel özellik

Nesne yönelimli dilleri prosedürel dillerden ayıran üç özellik vardır. Bu notun konusu sınıf ve nesnenin kendisidir. Yine de üç özelliğin neye hizmet ettiğini baştan bilmek, sınıf yazarken verilen kararları anlamayı kolaylaştırır.

- **Sarmalama** (*encapsulation*). Nesne, alanlarının ve davranışlarının ayrıntısını öteki nesnelerden gizleyebilir. Dışarıya karşı kapalı bir kutu gibidir; yalnızca izin verdiği metotlarla kullanılır. Bir televizyonun içindeki devreler kapalı bir kasadadır; kullanıcı yalnızca kumandanın düğmelerine erişir.
- **Kalıtım** (*inheritance*). Bir sınıf, başka bir sınıfın alanlarını ve metotlarını devralabilir. Böylece ortak kod bir kez yazılır ve sınıflar bir hiyerarşi içinde düzenlenir.
- **Çok biçimlilik** (*polymorphism*). Bir nesne birden çok tip olarak görülebilir. Aynı çağrı farklı nesnelerde farklı davranış üretir. `sekil.alan()` çağrısı bir dairede πr^2, bir karede a^2 hesaplar.

Bu üçlüye çoğu zaman dördüncü olarak **soyutlama** (*abstraction*) eklenir. Soyutlama, gereksiz ayrıntıyı gizleyip yalnızca önemli özellikleri göstermektir. Bir `BankaHesabi` sınıfı gerçek bir hesabın bütün özelliklerini değil, programın işine yarayanları, örneğin sahip adını ve bakiyeyi modeller. Hesabın açıldığı şubenin kapı rengi modele girmez.

4.1.2 Sınıf ve nesne

Bir fabrikada aynı modelden binlerce araba üretilir. Her arabanın bir rengi, bir şasi numarası, bir kilometre sayacı vardır ve bu değerler arabadan arabaya değişir. Ama hepsi aynı çizimden çıkar: hangi parçaların olacağını, motorun nasıl çalışacağını çizim belirler. Java'da bu çizimin karşılığı sınıf, üretilen her arabanın karşılığı nesnedir.

4.1.2.1 Sınıf: nesnenin taslağı

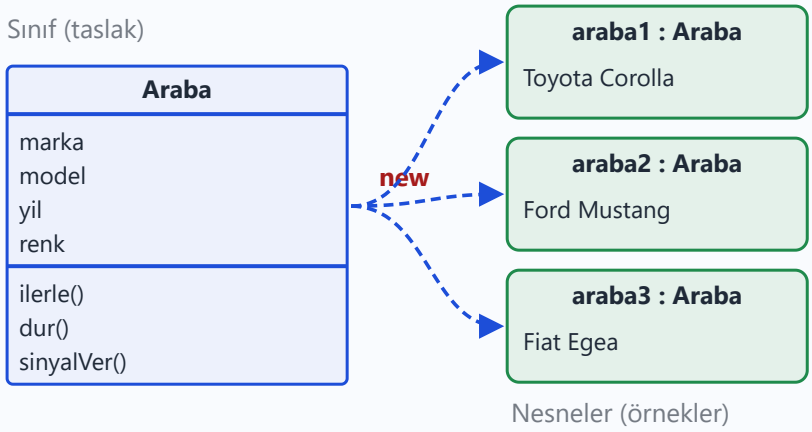
Tanım 4.1.2

Sınıf (*class*), bir nesnenin taslağıdır. Nesnelerin taşıyacağı verileri (alanları) ve yapabileceği işlemleri (metotları) tanımlar. Bir sınıftan farklı verileri taşıyan istenen sayıda nesne üretilebilir.

Tanım 4.1.3

Nesne (*object*), bir sınıfın **örneğidir** (*instance*): sınıfın tarif ettiği yapının bellekte oluşturulmuş somut hâlidir. Her nesne, sınıfın tanımladığı alanların kendi kopyalarını taşır.

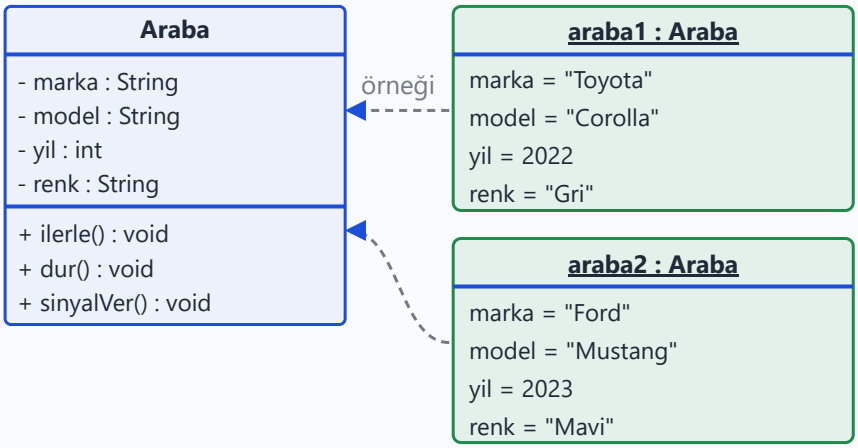
Sınıf ile nesne arasındaki ilişkiyi kurabiye kalıbı iyi anlatır. Kalıp bir kez yapılır ve kurabiyenin biçimini belirler. Kalıpla kesilen her kurabiye ayrı bir kurabiyedir: birine çikolata, ötekine fındık konabilir. Kalıbın kendisi yenmez. Sınıf da tek başına veri taşımaz; veriyi ondan üretilen nesneler taşır.



Şekil 4.1.3: `Araba` sınıfı ve ondan `new` ile üretilen üç nesne. Sınıf alan adlarını ve metotları tanımlar; her nesne bu alanlara kendi değerlerini koyar.

Şekil 4.1.3'ün solunda kesik çizgili kutu `Araba` sınıfıdır. Sınıf `marka`, `model`, `yil` ve `renk` alanlarını, `ilerle()`, `dur()` ve `sinyalVer()` metotlarını tanımlar. Sağdaki üç nesne aynı sınıftan üretilmiştir: Toyota Corolla, Ford Mustang ve Fiat Egea. Üçünde de aynı alan adları vardır, ama değerler farklıdır. Aynı sınıftan üretilen iki nesne aynı alan adlarını taşır; her birinin alan değerleri kendine aittir. Birinin rengini değiştirmek ötekinin rengini değiştirmez.

Sınıflar genellikle **UML sınıf diyagramıyla** (*Unified Modeling Language*, Birleşik Modelleme Dili) çizilir. Diyagramda sınıf üç bölmeli bir kutudur: en üstte sınıfın adı, ortada alanlar, altta metotlar. Her üyenin önündeki işaret erişim düzeyini gösterir: `-` `private`, `+` `public` demektir. Alanların yanında tipleri, metotların yanında dönüş tipleri yazılır. Nesneler de benzer bir kutuyla çizilir; ama başlıkta `araba1 : Araba` biçiminde nesnenin adı ve sınıfı bulunur ve başlığın altı çizilir.



Şekil 4.1.4: UML gösterimi. Solda `Araba` sınıfı (ad / alanlar / metotlar), sağda bu sınıfın iki nesnesi. Nesne kutularında alanların o anki değerleri yazılıdır; kesik oklar "örneği" ilişkisini gösterir.

4.1.2.2 Sınıf tanımlamak

Aşağıdaki kod Şekil 4.1.4'teki `Araba` sınıfının ilk Java sürümüdür. Diyagramda alanlar `private` işaretlidir; bu ilk sürümde ise hiçbir erişim belirleyici yazılmamıştır. Alanlar bu yüzden paket erişimlidir ve aynı klasördeki başka bir sınıf onlara doğrudan değer verebilir.

```
public class Araba {
  // Alanlar (fields)
  String marka;
  String model;
  int yil;
  String renk;

  // Metotlar (methods)
  void ilerle() {
    System.out.println(marka + " ilerliyor.");
  }

  void dur() {
    System.out.println(marka + " durdu.");
  }

  void sinyalVer() {
    System.out.println(marka + " sinyal veriyor.");
  }
}
```

Bir sınıf tanımı `class` anahtar sözcüğü ve sınıfın adıyla başlar. Sınıfın gövdesi süslü parantezler arasındadır. Ad büyük harfle başlar ve birden çok sözcükten oluşuyorsa her sözcüğün ilk harfi büyük yazılır: `Araba`, `BankaHesabi`, `RasyonelSayi`. `public` bir sınıf, kendisiyle aynı adı taşıyan bir dosyada bulunmalıdır. `public class Araba` tanımı `Araba.java` dosyasına yazılır. Bir dosyada birden çok sınıf olabilir, ama bunlardan en çok biri `public` olabilir.

Alanlar sınıfın gövdesinde, metotların dışında tanımlanır. Tanım bir yerel değişken tanımına benzer: tip ve ad. `String marka;` satırı her `Araba` nesnesinde `marka` adlı bir `String` alanı bulunacağını söyler. Alanlar sınıfın her metodundan doğrudan kullanılabilir. `ilerle()` metodu `marka`'yı parametre olarak almaz; onu sınıfın alanı olarak tanır.

Metotların başlığında `static` yoktur. Şimdiye kadarki metotların hepsi `static`'ti ve sınıf adıyla, nesnesiz çağrılıyordu. Buradaki metotlar ise belirli bir arabanın metotlarıdır. `ilerle()` çağrıldığında hangi arabanın markasının yazılacağını, metodun hangi nesne üzerinden çağrıldığı belirler. Bu tür metotlara **örneğe ait metot** (*instance method*) denir.

Sınıfta `main` metodu yoktur. Bu yüzden `Araba` tek başına çalıştırılamaz; bir yapı taşıdır ve başka bir sınıf onu kullanır. Gerçek programlarda sınıfların çoğu böyledir: yalnızca bir sınıf `main`'i taşır, öteki-ler onun kullandığı tipleri tanımlar.

4.1.2.3 Sınıfı kullanmak

`AracTesti` sınıfı iki `Araba` nesnesi oluşturur, alanlarına değer verir ve metotlarını çağırır. İki dosya aynı klasördeyse `javac Araba.java AracTesti.java` ile derlenir, `java AracTesti` ile çalıştırılır.

```
public class AracTesti {
  public static void main(String[] args) {
    Araba araba1 = new Araba();
    araba1.marka = "Toyota";
    araba1.model = "Corolla";
    araba1.yil = 2022;
    araba1.renk = "Gri";

    Araba araba2 = new Araba();
    araba2.marka = "Ford";
    araba2.model = "Mustang";
    araba2.yil = 2023;
    araba2.renk = "Mavi";

    araba1.ilerle();
    araba2.sinyalVer();
    araba1.dur();
  }
}
```

```
Toyota ilerliyor.
Ford sinyal veriyor.
Toyota durdu.
```

İlk satırdaki `new Araba()` bellekte yeni bir `Araba` nesnesi oluşturur. `Araba araba1` tanımı ise `Araba` tipinde bir değişken açar. Atamadan sonra `araba1` yeni oluşturulan nesneyi gösterir. Değişkenin tipi bir sınıf adıdır; Java'da her sınıf aynı zamanda yeni bir tiptir.

Sonraki dört satır nesnenin alanlarına değer verir. `araba1.marka = "Toyota";` satırındaki nokta, "`araba1`'in gösterdiği nesnenin `marka` alanı" anlamına gelir. Bu işarete **nokta işleci** (*dot operator*) denir. İkinci nesne için aynı alanlara başka değerler verilir. Artık bellekte iki ayrı `marka` alanı vardır: biri `"Toyota"`, öteki `"Ford"` değerini taşır.

Son üç satır metot çağırır. `araba1.ilerle()` çağrısında metodun gövdesindeki `marka`, `araba1`'in markasıdır: `Toyota ilerliyor.` yazılır. `araba2.sinyalVer()` çağrısında aynı ad `araba2`'nin markasını gösterir: `Ford sinyal veriyor.` Metot kodu bir kez yazılmıştır; hangi nesnenin verisiyle çalışacağını, noktanın solundaki nesne belirler.

Alanlara dışarıdan böyle doğrudan değer vermek iyi bir alışkanlık değildir. `araba1.yil = -5;` satırı da derlenir ve nesneyi anlamsız bir duruma sokar. Bir nesnenin alanlarına ilk değerlerini vermenin doğru yolu yapıcılardır; değerleri sonradan değiştirmenin doğru yolu ise denetim yapan metotlardır.

Buradaki yazım yalnızca nokta işlecini tanıtmak içindir.

4.1.2.4 Sınıfın üyeleri

Sınıfın gövdesinde tanımlanan her şeye sınıfın **üyesi** (*member*) denir. İki temel üye türü vardır:

- **Alanlar**, nesnelerin taşıyabileceği verileri temsil eder. Bir `Ogrenci` sınıfında `ad` ve `soyad` birer alandır.
- **Metotlar**, nesnelerin yapabileceği işlemleri tanımlar. Bir `Kopek` sınıfında `havla()` ve `kos()` birer metottur.

Her üyenin iki ayrı özelliği daha vardır. Birincisi **erişim düzeyidir**: üye `private`, `protected`, `public` ya da hiçbiri yazılmadığında paket erişimli olabilir. Erişim düzeyi, üyeyi kimin kullanabileceğini belirler. İkincisi **sahipliktir**: üye ya her nesnede ayrı bulunur (örneğe ait, varsayılan durum) ya da `static` ile tanımlanıp sınıfın kendisine ait olur.

Sınıfın gövdesinde iki tür yapı daha bulunabilir. **Yapıcılar** sınıfın içinde yazılır ve metotlara benzer; ama yalnızca nesne oluşturulurken çalışır. Java dil tanımı onları metot değil, ayrı bir üye türü sayar. Bir sınıfın içinde başka sınıflar da tanımlanabilir; bunlara **iç sınıf** denir.

4.1.2.5 Erişim belirleyiciler ve veri gizleme

> **Tanım 4.1.4**
>
> **Erişim belirleyici** (*access modifier*), bir üyeye sınıfın dışından kimin erişebileceğini belirten anahtar sözcüktür. Java'da dört erişim düzeyi vardır: `public` (herkes), `protected` (aynı paket ve alt sınıflar), yazılmadığında **paket erişimi** (aynı paketteki sınıflar) ve `private` (yalnızca sınıfın kendisi).

Genel kural basittir: alanlar `private`, dışarıya sunulan metotlar `public` tanımlanır. Alanın değeri dışarıdan bir **erişimci** (*getter*) ile okunur; adı geleneksel olarak `get` ile başlar. Değer bir **değiştirici** (*setter*) ile değiştirilir; adı `set` ile başlar. Değiştiricinin asıl görevi gelen değeri denetlemek ve geçersiz değeri reddetmektir.

```
public class BankaHesabi {
  private double hesapBakiyesi;   // gizli alan
  public double getHesapBakiyesi() {
    return hesapBakiyesi;
  }
  public void setHesapBakiyesi(double b) {
    if (b >= 0) hesapBakiyesi = b;  // geçersizi reddet
  }
}
// başka bir sınıfta:
hesap.hesapBakiyesi = -500;
```

`hesapBakiyesi` alanı `private`'tır. `BankaHesabi` sınıfının dışındaki bir sınıf bu alanı okuyamaz da değiştiremez de. Kodun son satırı başka bir sınıfa yazıldığında derleyici şu iletiyi verir:

```
class BankaHesabi {
  private double hesapBakiyesi;   // gizli alan
}
public class OzelErisim {
  public static void main(String[] args) {
    BankaHesabi hesap = new BankaHesabi();
    hesap.hesapBakiyesi = -500;
  }
}
```

```
OzelErisim.java:7: error: hesapBakiyesi has private access in BankaHesabi
    hesap.hesapBakiyesi = -500;
         ^
1 error
```

Dışarıdan erişimin tek yolu iki `public` metottur. `getHesapBakiyesi()` değeri döndürür. `setHesapBakiyesi(b)` ise yeni değeri yazmadan önce denetler: `b` negatifse atama yapılmaz. Aşağıdaki deneme programı önce geçerli, sonra geçersiz bir bakiye vermeye çalışır:

```
public class HesapTesti {
  public static void main(String[] args) {
    BankaHesabi hesap = new BankaHesabi();
    hesap.setHesapBakiyesi(1500);
    hesap.setHesapBakiyesi(-500);   // reddedilir
    System.out.println(hesap.getHesapBakiyesi());
  }
}
```

```
1500.0
```

`-500` reddedildiği için bakiye `1500.0` kalır. Program bakiyeyi negatif yapamaz; bunu engelleyen kural tek bir yerde, `setHesapBakiyesi`'nin içinde yazılıdır. Alan `public` olsaydı aynı kuralı alanı değiştiren her satırın ayrıca denetlemesi gerekirdi ve bir tanesinin unutulması nesneyi bozardı.

Tanım 4.1.5

Veri gizleme (*data hiding*), alanlara yalnızca kendi sınıfının metotlarının doğrudan erişebildiği düzendir. Alanlar `private` yapılır, dış dünya nesneyle yalnızca `public` metotlar üzerinden konuşur. Veri gizleme sarmalamanın temelidir.

Veri gizlemenin ikinci kazancı değişiklik özgürlüğüdür. Bakiyeyi `double` yerine kuruş cinsinden bir `long` alanda tutmaya karar verildiğini düşünelim. Alan `private` olduğu için onu dışarıda kullanan hiçbir satır yoktur. Yalnızca iki metodun gövdesi değişir; sınıfı kullanan kod hiç değişmez.

Sık yapılan hata

Her `private` alana düşünmeden bir get ve bir set metodu eklemek. Denetim yapmayan `public void setBakiye(double b) { bakiye = b; }` metodu alanı fiilen `public` yapar. Değiştirici yalnızca alanın dışarıdan değişmesi gerçekten gerekiyorsa yazılır ve geçersiz değerleri reddeder. Hiç değişmemesi gereken alanın değiştiricisi olmaz.

4.1.3 Yapıcılar ve nesne oluşturma

`AracTesti`'nde her nesne beş satırda kuruldu: önce `new Araba()`, sonra dört ayrı atama. Bu düzende atamalardan biri unutulursa nesne yarım kalır ve hata ancak çalışma sırasında, belki çok sonra ortaya çıkar. Yapıcılar nesnenin oluşturulduğu anda geçerli bir duruma getirilmesini sağlar.

4.1.3.1 Yapıcı nedir?

Tanım 4.1.6

Yapıcı (*constructor*), nesne oluşturulurken çalışan özel bir kod bloğudur. `new` ile her nesne oluşturulduğunda kendiliğinden çağrılır. Adı sınıfın adıyla birebir aynıdır (büyük-küçük harf dahil) ve dönüş tipi yoktur; `void` bile yazılmaz.

Yapıcının görevi alanlara ilk değerlerini vermek ve nesnenin kullanılmaya hazır olması için gereken başka işleri yapmaktır. Aşağıdaki `Ogrenci` sınıfının yapıcısı iki parametre alır ve onları nesnenin alanlarına yazar.

```
public class Ogrenci {
  private String ad;
  private int numara;

  // Yapıcı: sınıfla aynı ad, dönüş türü yok
  public Ogrenci(String ad, int numara) {
    this.ad = ad;
    this.numara = numara;
  }
}

// Kullanımı:
Ogrenci ogr = new Ogrenci("Ali", 2012);
```

Yapıcının başlığı bir metot başlığına benzer, ama iki fark taşır. Ad, sınıfın adıdır: `Ogrenci`. Adın önünde dönüş tipi yoktur. Yapıcı bir değer döndürmez; `new` ifadesinin sonucu her zaman yeni nesnenin kendisidir. Gövdedeki `this.ad = ad;` satırı, `ad` parametresinin değerini nesnenin `ad` alanına yazar. Buradaki `this`, "oluşturulmakta olan nesne" anlamına gelir.

Yapıcılar genellikle `public` tanımlanır, çünkü nesneyi sınıfın dışındaki kod oluşturur. Bazen nesne oluşturmayı kısıtlamak istenir. Bu durumda yapıcı `private` yapılır ve sınıfın dışından `new` çağrılamaz. Tek nesnelik (*singleton*) sınıflar bu yolla kurulur. Bir sınıfın farklı parametre listeleriyle birden çok yapıcısı olabilir.

4.1.3.2 new ifadesi

Nesne oluşturmanın genel biçimi şudur:

```
SinifAdi nesneAdi = new SinifAdi(parametreler);

Araba corolla = new Araba("Toyota", "Corolla", 2022);
Scanner klavye = new Scanner(System.in);
Kitap kitap = new Kitap();
```

İlk satır kalıbı, sonraki üç satır üç örneği gösterir. Eşittir işaretinin solunda değişken tanımı, sağında `new` ifadesi vardır. Java'nın kendi sınıfları da aynı yolla kullanılır: klavyeden okumak için `new Scanner(System.in)`, değiştirilebilir bir metin için `new StringBuilder("Java")` yazılır. Aşağıdaki program dört nesne oluşturur ve kullanır.

```
import java.util.Scanner;
public class Olusturma {
  public static void main(String[] args) {
    Scanner klavye = new Scanner(System.in);
    StringBuilder metin = new StringBuilder("Java");
    String baslik = "Nesneler";           // kısayol
    String yazar = new String("Sercan");  // uzun yol
    metin.append(" 21");
    System.out.println(metin + " / " + baslik);
    System.out.println(yazar.length());
    System.out.println(klavye.nextInt() * 2);
  }
}
```

Klavyeden `21` girildiğinde program şunu yazar:

```
Java 21 / Nesneler
6
42
```

`new` ifadesi üç adımda çalışır:

1. `new`, **heap** adı verilen bellek bölgesinde nesne için yer ayırır. Nesnenin bütün alanları önce varsayılan değerlerini alır: sayılar 0, `boolean` alanlar `false`, referans alanlar `null` olur.
2. Argümanlara uyan yapıcı çağrılır ve alanlara ilk değerlerini verir.
3. `new` ifadesinin sonucu yeni nesnenin **referansıdır**. Bu referans eşittir işaretinin solundaki değişkene atanır.

`String`'de bu kuralın tanıdık bir istisnası vardır: metin sabiti yazmak da bir `String` nesnesi verir. `String baslik = "Nesneler";` satırı `new` kullanmadan bir `String` nesnesinin referansını `baslik`'a koyar. `new String("Sercan")` yazımı da geçerlidir, ama her çalıştığında gereksiz yere yeni bir nesne üretir. Yalnızca `String baslik;` yazmak ise hiçbir nesne oluşturmaz; yalnızca referans tutacak bir değişken tanımlar.

Çıktının ikinci satırındaki `6`, `yazar.length()` çağrısının sonucudur: `"Sercan"` altı harflidir. Bu çağrı da nokta işleciyle yapılmış bir örnek metodu çağrısıdır. Programcının kendi yazdığı sınıflarla Java'nın hazır sınıfları arasında kullanım açısından fark yoktur.

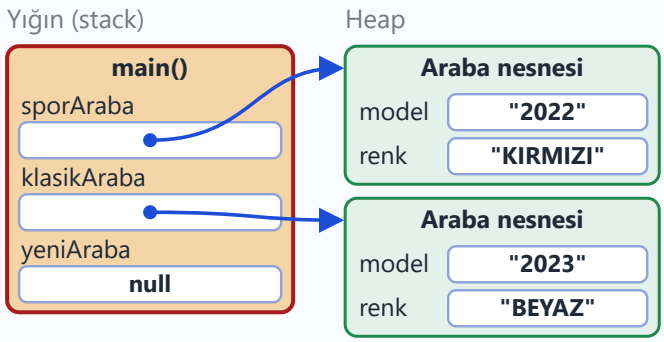
4.1.3.3 Referanslar

Bir nesneye doğrudan dokunulamaz. Program bir nesneyle yalnızca ona giden **referans** aracılığıyla çalışır. Sınıf tipinden bir değişken nesnenin kendisini değil, bu referansı saklar.

Tanım 4.1.7

Referans (*reference*), heap'teki bir nesnenin yerini gösteren değerdir. Nesnenin bellekteki adresi gibi düşünülebilir. Sınıf tipinden değişkenler (`Araba` , `String` , `Scanner` , diziler) referans tutar. Hiçbir nesneyi göstermeyen referansın değeri `null` 'dır.

`int` , `double` , `boolean` gibi ilkel tipli değişkenler değeri kendileri tutar. `int x = 5;` satırından sonra `x` 'in kutusunda 5 sayısı durur. `Araba a = new Araba(...);` satırından sonra ise `a` 'nın kutusunda bir araba değil, arabanın yerini gösteren bir ok durur. Bu farkı akılda tutmanın kolay yolu bir benzetmedir: değişken bir televizyon kumandası, nesne ise televizyondur. Kumandayı kopyalamak ikinci bir televizyon yaratmaz; iki kumanda aynı televizyonu açıp kapatır.



Şekil 4.1.5: Yerel değişkenler yığında, nesneler heap'te durur. `sporAraba` ve `klasikAraba` iki ayrı nesneyi gösterir; `yeniAraba` henüz hiçbir nesneyi göstermez (`null`).

Bellek iki bölgeye ayrılır. Çalışan metotların çerçeveleri ve yerel değişkenleri **yığında** (*stack*) durur. Nesneler ise **heap**'te (*heap*) tutulur. Şekil 4.1.5'te `main` 'in çerçevesi soldadır. Çerçevedeki `sporAraba` ve `klasikAraba` kutularından çıkan oklar heap'teki iki nesneye uzanır. Üçüncü değişken `yeniAraba` tanımlanmış ama ona henüz bir nesne atanmamıştır; değeri `null` 'dır. Bu çizim biçimi notun geri kalanında hep kullanılır: kutu değişkeni, ok referansı, heap'teki çerçeveli kutu nesneyi gösterir.

4.1.3.4 Nesne oluşturmayı adım adım izlemek

Aşağıdaki `Araba` sınıfı iki alan, iki parametreli bir yapıcı ve bir metot içerir. `main` iki nesne oluşturup ikisinin bilgisini yazdırır.

```
public class Araba {
  private String model;
  private String renk;

  public Araba(String model, String renk) {
    this.model = model;
    this.renk = renk;
  }

  public void bilgiYazdir() {
    System.out.println(model + " - " + renk);
  }

  public static void main(String[] args) {
    Araba sporAraba = new Araba("2022", "KIRMIZI");
    Araba klasikAraba = new Araba("2023", "BEYAZ");
    sporAraba.bilgiYazdir();
    klasikAraba.bilgiYazdir();
  }
}
```

```
2022 - KIRMIZI
2023 - BEYAZ
```

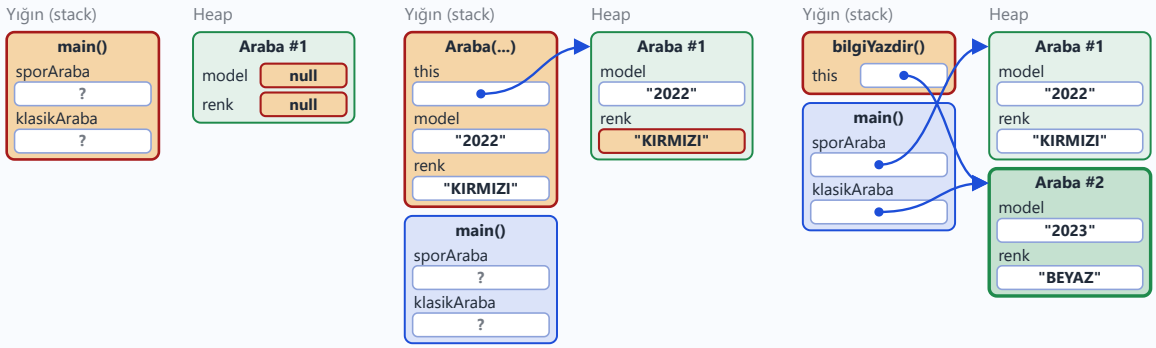
Tablo 4.1.3 programın sekiz adımını verir. Satır numaraları yukarıdaki kodun satırlarıdır. “Değişkenler” sütunu o anda en üstteki çerçevenin değişkenlerini gösterir; `→#1` yazımı değişkenin heap'teki 1 numaralı nesneyi gösterdiğini, `?` ise değişkene henüz değer verilmediğini belirtir.

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	15: `Araba sporAraba = new Araba("2022", "KIRMIZI");`	main()	sporAraba=?, klasikAraba=?	—
2	5: `public Araba(String model, String renk) {`	main() → Araba(model, renk)	model="2022", renk="KIRMIZI"	—
3	6: `this.model = model;`	main() → Araba(model, renk)	model="2022", renk="KIRMIZI"	—
4	7: `this.renk = renk;`	main() → Araba(model, renk)	model="2022", renk="KIRMIZI"	—
5	15: `Araba sporAraba = new Araba("2022", "KIRMIZI");`	main()	sporAraba=→#1, klasikAraba=?	—
6	16: `Araba klasikAraba = new Araba("2023", "BEYAZ");`	main()	sporAraba=→#1, klasikAraba=→#2	—
7	17: `sporAraba.bilgiYazdir();` 11: `System.out.println(model + " - " + renk);`	main() → bilgiYazdir()	sporAraba=→#1, klasikAraba=→#2	`2022 - KIRMIZI`

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
8	18: `klasikAraba.bilgiYazdir();` 11: `System.out.println(model + " - " + renk);`	main() → bilgiYazdir()	sporAraba=→#1, klasikAraba=→#2	`2023 - BEYAZ`

Tablo 4.1.3: `new Araba(...)` çağrılarının ve `bilgiYazdir()` 'ın izlenmesi.

Birinci adımda 15. satır çalışmaya başlar. `new` heap'te bir `Araba` nesnesi için yer ayırır. `model` ve `renk` `String` tipinde, yani referans tipinde alanlardır; varsayılan değerleri `null` 'dır. Nesne bu anda vardır ama içi boştur.



Şekil 4.1.6: Üç an. Solda `new` nesneyi ayırmış, alanlar `null` . Ortada yapıcı çalışıyor: yapıcının çerçevesindeki `this` yeni nesneyi gösteriyor ve iki alan dolmuş. Sağda ikinci `bilgiYazdir()` çağrısı: `this` bu kez 2 numaralı nesneyi gösteriyor.

İkinci adımda yapıcı çağrılır ve yığında yeni bir çerçeve açılır. Parametreler argümanların kopyasını alır: `model = "2022"` , `renk = "KIRMIZI"` . Çerçevede bir değişken daha vardır: `this` . Java, yapıcıyı çağırırken yeni nesnenin referansını `this` adıyla gizlice verir. Üçüncü ve dördüncü adımda `this.model = model;` ve `this.renk = renk;` çalışır. Her satırda eşittir işaretinin solundaki `this.model` nesnenin alanı, sağındaki `model` ise parametredir. Bu iki satırdan sonra nesne hazırdır (Şekil 4.1.6, orta).

Beşinci adımda yapıcı biter ve çerçevesi silinir. Parametreler ve `this` onunla birlikte yok olur; ama nesne heap'te kalır. `new` ifadesinin değeri olan referans `sporAraba` 'ya atanır. Altıncı adım aynı işleri ikinci nesne için yapar. Heap'te kendi `model` ve `renk` kopyalarıyla ayrı bir nesne oluşur ve `klasikAraba` onu gösterir.

Yedinci adımda `sporAraba.bilgiYazdir()` çağrılır. Metodun çerçevesinde de bir `this` vardır ve değeri noktanın solundaki referanstır: 1 numaralı nesne. Metodun gövdesindeki `model` ve `renk` adları `this.model` ve `this.renk` anlamına gelir. Bu yüzden `2022 - KIRMIZI` yazılır. Sekizinci adımda aynı metot `klasikAraba` üzerinden çağrılır; `this` bu kez 2 numaralı nesneyi gösterir ve `2023 - BEYAZ` yazılır. Kod tek, veriler nesne başına ayrıdır.

Bir nesnenin bellekte kapladığı yer, alanlarının toplamı kadardır. Metotlar her nesnede kopyalanmaz; sınıfın kodu bellekte bir kez bulunur. Bu yüzden bin `Araba` nesnesi oluşturmak bin kez iki referanslık yer tutar, ama `bilgiYazdir`'ın kodu yalnızca bir kez yer kaplar.

4.1.3.5 Varsayılan yapıcı ve varsayılan değerler

İlk `Araba` sınıfında hiç yapıcı yoktu, ama `new Araba()` sorunsuz derlendi. Bunun nedeni derleyicinin yardımıdır.

> **Tanım 4.1.8**
>
> **Varsayılan yapıcı** (*default constructor*), sınıfa hiç yapıcı yazılmamışsa derleyicinin eklediği parametresiz ve gövdesi boş yapıcıdır. Sınıfa tek bir yapıcı bile yazılırsa derleyici varsayılan yapıcıyı eklemez.

Nesne oluşturulurken her zaman bir yapıcı çağrılır; varsayılan yapıcı, programcı yazmadığında bu çağrının bir hedefi olmasını sağlar. Gövdesi boş olduğu için alanlara hiçbir değer vermez. Değer verilmeyen alanlar `new`'ün birinci adımında aldıkları varsayılan değerlerde kalır. Tablo 4.1.4 bu değerleri listeler.

Alan tipi	Varsayılan değer
`byte`, `short`, `int`, `long`	`0`
`float`, `double`	`0.0`
`boolean`	`false`
`char`	`'\u0000'` (kod 0)
Referans (`String`, diziler, nesneler)	`null`

Tablo 4.1.4: Alanların varsayılan değerleri. Yerel değişkenler varsayılan değer almaz.

```
class Nokta3 {
  int x;
  double agirlik;
  boolean gorunur;
  char harf;
  String ad;
}
public class Varsayilan {
  public static void main(String[] args) {
    Nokta3 n = new Nokta3();   // varsayılan yapıcı
    System.out.println(n.x + " " + n.agirlik);
    System.out.println(n.gorunur + " " + n.ad);
    System.out.println((int) n.harf);
  }
}
```

```
0 0.0
false null
0
```

`Nokta3` sınıfında hiç yapıcı yoktur; `new Nokta3()` derleyicinin eklediği varsayılan yapıcıyı çağırır. Beş alanın hiçbirine değer verilmemiştir ve her biri tablodaki değeri taşır. `char` alanının değeri 0 kodlu karakterdir. Bu karakter ekranda görünmez, bu yüzden program onu `(int)` ile sayıya çevirerek yazar: `0`.

Varsayılan değer kuralı yalnızca alanlar ve dizi öğeleri içindir. Bir metodun içindeki yerel değişken varsayılan değer almaz. Değer verilmeden okunan yerel değişkeni derleyici kabul etmez:

```
public class IlkDegersiz {
  public static void main(String[] args) {
    StringBuilder sb;
    sb.append("x");
  }
}
```

```
IlkDegersiz.java:4: error: variable sb might not have been initialized
    sb.append("x");
    ^
1 error
```

Bu farkın nedeni güvenliktir. Derleyici bir metodun içindeki bütün yolları inceleyip yerel değişkenin okunmadan önce değer alıp almadığını görebilir. Alanlar için bunu yapamaz, çünkü bir alan herhangi bir metottan, herhangi bir sırayla okunabilir. Java bu yüzden alanları en azından tanımlı bir değerle

başlatır.

4.1.3.6 Parametresiz ve parametreli yapıcı

Bir sınıfa birden çok yapıcı yazılabilir. Aşağıdaki `Arac` sınıfının biri parametresiz, biri üç parametreli iki yapıcısı vardır.

```
public class Arac {
  private String marka;
  private String model;
  private int yil;
  public Arac() {                 // parametresiz yapıcı
    marka = "Bilinmiyor";
    model = "Bilinmiyor";
    yil = 0;
  }
  public Arac(String marka, String model, int yil) {
    this.marka = marka;
    this.model = model;
    this.yil = yil;
  }
  public void bilgiYazdir() {
    System.out.println(marka + " " + model + " " + yil);
  }
  public static void main(String[] args) {
    Arac a1 = new Arac();
    Arac a2 = new Arac("Renault", "Clio", 2021);
    a1.bilgiYazdir();
    a2.bilgiYazdir();
  }
}
```

```
Bilinmiyor Bilinmiyor 0
Renault Clio 2021
```

`Arac()` programcının kendi yazdığı parametresiz yapıcıdır. Varsayılan yapıcıdan farkı, alanlara anlamlı başlangıç değerleri vermesidir: marka ve model `null` yerine `"Bilinmiyor"` olur. İki yapıcı aynı adı taşır ve parametre listeleriyle ayrılır. Bu, metotlardaki aşırı yüklemenin yapıcılardaki karşılığıdır: **yapıcı aşırı yükleme**. Hangi yapıcının çalışacağını `new` ifadesindeki argümanlar belirler. `new Arac()` parametresiz yapıcıyı, `new Arac("Renault", "Clio", 2021)` üç parametreli yapıcıyı çağırır.

Sınıfa herhangi bir yapıcı yazıldığında derleyici artık varsayılan yapıcı eklemez. Yalnızca parametreli yapıcısı olan bir sınıfta `new` ile parametresiz nesne oluşturmak derleme hatasıdır:

```
public class AracVarsayilan {
  private String marka;
  public AracVarsayilan(String marka) {
    this.marka = marka;
  }
  public static void main(String[] args) {
    AracVarsayilan a = new AracVarsayilan();
  }
}
```

```
AracVarsayilan.java:7: error: constructor AracVarsayilan in class AracVarsayilan cannot be
applied to given types;
    AracVarsayilan a = new AracVarsayilan();
                       ^
  required: String
  found:    no arguments
  reason: actual and formal argument lists differ in length
1 error
```

İletinin alt satırları neyin beklendiğini açıkça söyler: `required: String`, yani bir `String` argüman bekleniyor; `found: no arguments`, yani hiç argüman verilmemiş. Parametresiz nesne oluşturmak gerekiyorsa parametresiz yapıcı da elle yazılır.

Sık yapılan hata

Yapıcıya dönüş tipi yazmak. `public void Hata1(String m)` yapıcı değil, sınıfla aynı adı taşıyan sıradan bir metottur. Sınıfta başka yapıcı olmadığı için derleyici parametresiz varsayılan yapıcıyı ekler ve `new Hata1("selam")` şu iletiyle reddedilir:

```
Hata1.java:7: error: constructor Hata1 in class Hata1 cannot be applied to given
types;
    Hata1 h = new Hata1("selam");
              ^
  required: no arguments
  found:    String
  reason: actual and formal argument lists differ in length
1 error
```

İletide `required: no arguments` yazması, derleyicinin yalnızca parametresiz yapıcıyı gördüğünü gösterir. `void` silinince yapıcı yerine gelir.

4.1.4 Alanlar

Alanlar sınıfın içinde, metotların dışında tanımlanan değişkenlerdir. Yerel değişkenlerden üç noktada ayrılırlar. Varsayılan değer alırlar. Bir metot bitince yok olmazlar, nesneyle birlikte yaşarlar. Sınıfın bütün metotlarından görünürler. Bir alanın kime ait olduğu ise `static` kelimesine bağlıdır.

4.1.4.1 Örneğe ait ve sınıfa ait alanlar

Tanım 4.1.9

Örneğe ait alan (*instance field*), `static` olmadan tanımlanan alandır; her nesnede ayrı bir kopyası bulunur. **Sınıfa ait alan** (*static field*), `static` ile tanımlanan alandır; sınıfta tek bir kopyası bulunur ve bütün nesneler bu kopyayı paylaşır.

	Örneğe ait alan	**Sınıfa ait alan (`static`)**
Tanım	`private String ad;`	`private static int toplamHesapSayisi;`
Kopya sayısı	Her nesnede ayrı bir kopya	Sınıfta tek kopya, bütün nesneler paylaşır
Ne zaman oluşur?	Her `new` ile	Sınıf belleğe yüklenirken, bir kez
Erişim	`nesne.ad`	`SinifAdi.toplamHesapSayisi`
Örnek	`Ogrenci` : `ad` , `soyad`	`BankaHesabi` : `toplamHesapSayisi`

Tablo 4.1.5: Örneğe ait ve sınıfa ait alanların karşılaştırması.

Bir alanın örneğe mi sınıfa mı ait olacağını şu soru belirler: değer nesneden nesneye değişir mi? Her öğrencinin adı farklıdır; `ad` örneğe ait bir alandır. Bankada açılmış toplam hesap sayısı ise tek bir sayıdır. Her hesap nesnesinde ayrı bir sayaç tutmak hem gereksiz olurdu hem de sayaçlar birbirinden farklı değerler gösterirdi. Bu yüzden sayaç `static` tanımlanır.

4.1.4.2 static sayacı izlemek

Aşağıdaki sınıfta her hesap kendi sahibinin adını tutar; bütün hesaplar ise tek bir sayacı paylaşır. Yapıcı her çalıştığında sayaç bir artar.

```
public class BankaHesabi {
  private String hesapSahibi;
  private static int toplamHesapSayisi = 0;

  public BankaHesabi(String hesapSahibi) {
    this.hesapSahibi = hesapSahibi;
    toplamHesapSayisi++;
  }

  public static int getToplamHesapSayisi() {
    return toplamHesapSayisi;
  }

  public static void main(String[] args) {
    BankaHesabi h1 = new BankaHesabi("Ali");
    BankaHesabi h2 = new BankaHesabi("Ayşe");
    BankaHesabi h3 = new BankaHesabi("Can");
    System.out.println(h2.hesapSahibi);
    int sayi = BankaHesabi.getToplamHesapSayisi();
    System.out.println(sayi);
  }
}
```

```
Ayşe
3
```

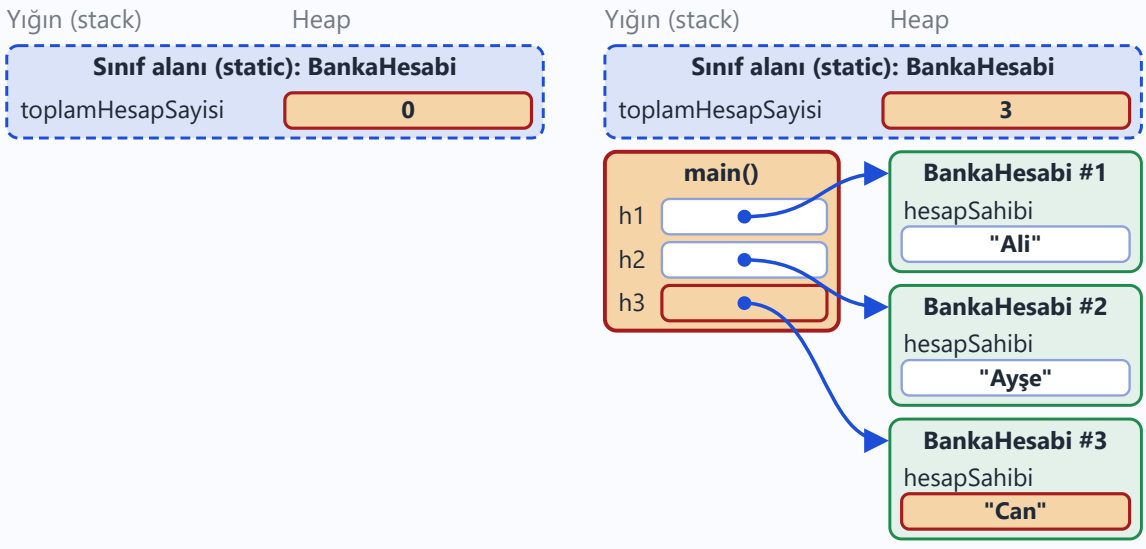
Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	3: `private static int toplamHesapSayisi = 0;`	(sınıf yükleniyor)	toplamHesapSayisi=0	—
2	5: `public BankaHesabi(String hesapSahibi) {` 6: `this.hesapSahibi = hesapSahibi;` 7: `toplamHesapSayisi++;`	main() → BankaHesabi("Ali")	toplamHesapSayisi=1, h1=?	—
3	15: `BankaHesabi h1 = new BankaHesabi("Ali");`	main()	toplamHesapSayisi=1, h1=→#1	—
4	16: `BankaHesabi h2 = new BankaHesabi("Ayşe");`	main()	toplamHesapSayisi=2, h1=→#1, h2=→#2	—
5	17: `BankaHesabi h3 = new BankaHesabi("Can");`	main()	toplamHesapSayisi=3, h1=→#1, h2=→#2, h3=→#3	—
6	18: `System.out.println(h2.hesapSahibi);`	main()	toplamHesapSayisi=3	`Ayşe`

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
7	19: `int sayi = BankaHesabi.getToplamHesapSayisi();` 10: `public static int getToplamHesapSayisi() {` 11: `return toplamHesapSayisi;`	main() → getToplam-HesapSayisi()	toplamHesapSayisi=3	—
8	20: `System.out.println(sayi);`	main()	toplamHesapSayisi=3, sayi=3	`3`

Tablo 4.1.6: `static` sayacın izlenmesi. Değişkenler sütununda `toplamHesapSayisi` sınıf alanının değeridir.

Birinci adım `main`'den önce gerçekleşir. `BankaHesabi` sınıfı belleğe yüklenirken `static` alan bir kez oluşur ve ilk değerini alır: 0. Bu anda heap'te hiçbir `BankaHesabi` nesnesi yoktur; ama sayaç vardır. `static` alan nesnelerin değil, sınıfın parçasıdır ve yaşamı sınıfın yüklenmesiyle başlar.

İkinci adımda `new BankaHesabi("Ali")` ilk nesneyi oluşturur. Yapıcının ilk satırı nesnenin kendi `hesapSahibi` alanına `"Ali"` yazar. İkinci satırdaki `toplamHesapSayisi++` ise ortak sayacı 1'e çıkarır. Yapıcının içinden `static` alana sınıf adı yazmadan erişilebilir; derleyici adın bir sınıf alanı olduğunu bilir. Üçüncü adımda `h1` bu nesneyi göstermeye başlar.



Şekil 4.1.7: Solda sınıf yüklenmiş, sayaç 0, henüz nesne yok. Sağda üç nesne oluşmuş: her birinin kendi `hesapSahibi` alanı var, ama sayaç tek kutudur ve değeri 3'tür.

Dördüncü ve beşinci adımlar aynı işi iki kez daha yapar. Her nesnenin ayrı bir `hesapSahibi` alanı oluşur, ama artırılan sayaç hep aynı kutudur: 2, sonra 3. Şekil 4.1.7'nin sağ parçası bu anı çizer. Heap'te üç nesne, üç ayrı `hesapSahibi` kutusu vardır; `toplamHesapSayisi` ise nesnelerin dışında, sınıfa ait bölgede tek bir kutudur.

Altıncı adımda `h2.hesapSahibi` okunur: örneğe ait alan nesne üzerinden okunur ve `Ayşe` yazılır. Yedinci adımda `static` metot `getToplamHesapSayisi()` sınıf adıyla çağrılır. Bu metodun çerçevesinde `this` yoktur, çünkü metot hiçbir nesne üzerinden çağrılmamıştır. Metot yalnızca `static` alanı okur ve 3 döndürür. Sekizinci adımda dönen değer `sayi`'ya atanır ve yazdırılır.

`main` metodu `BankaHesabi` sınıfının içinde olduğu için `h2.hesapSahibi` satırı derlenir, oysa alan `private`'tır. `private`, "yalnızca bu sınıfın içindeki kod" demektir; erişimi sınıf belirler, nesne belirlemez. Aynı satır başka bir sınıfın `main`'ine yazılsaydı `has private access` hatası alınırdı.

4.1.4.3 Alan tanımları ve final

Bir alan tanımının genel biçimi şudur: `[erişim] [static] [final] tip ad [= ilkDeğer];`. Köşeli parantezdeki parçalar isteğe bağlıdır. Aşağıdaki iki sınıf farklı birleşimleri gösterir.

```
public class Kitap {
  // Örneğe ait alanlar
  private String baslik;
  private String yazar;
  private int sayfaSayisi;
  // Sınıfa ait alan
  private static int toplamKitapSayisi;
  // final tanımlanmış bir alan (sabit)
  public static final double PI = 3.14159265359;
}
```

```
public class Alanlar {
  public int a;                       // genel
  int b = 1, c = 2;                   // varsayılan (paket)
  private double x;                   // özel
  private static int xx;              // statik, özel
  public static int y;                // statik, genel
  public final int CONST1 = 5;        // değiştirilemez
  private static final int CONST2 = 6;  // statik sabit
}
```

`Kitap` sınıfında üç örnek alanı, bir sınıf alanı ve bir sabit vardır. `Alanlar` sınıfı ise erişim belirleyicilerle `static` ve `final`'ın birleşimlerini sıralar. `int b = 1, c = 2;` satırı aynı tipte iki alanı tek satırda tanımlar ve ikisine de ilk değer verir. Erişim belirleyici yazılmadığı için bu iki alan paket erişimlidir.

`final` ile tanımlanan alana değer yalnızca bir kez verilir; sonra değiştirilemez. Değer ya tanım satırında ya da yapıcıda verilir. Sonradan değer atamaya çalışan satır derlenmez:

```
public class FinalAta {
  public final int CONST1 = 5;
  void degistir() {
    CONST1 = 6;
  }
}
```

```
FinalAta.java:4: error: cannot assign a value to final variable CONST1
    CONST1 = 6;
    ^
1 error
```

Sınıf sabitleri genellikle `static final` tanımlanır ve adları alt çizgiyle ayrılmış büyük harflerle yazılır: `MAX_DEGER`, `PI`. `static` olduğu için sabitin tek bir kopyası vardır; `final` olduğu için değeri değişmez. `Kitap` sınıfındaki `PI` yalnızca örnek olarak tanımlanmıştır. Java'nın hazır bir sabiti zaten vardır ve daha duyarlıdır: `Math.PI`.

`Alanlar` sınıfındaki `CONST1` `final`'dır ama `static` değildir. Bu yüzden her nesnede ayrı bir kopyası bulunur ve hepsinin değeri 5'tir. Değeri herkes için aynı olan bir sabiti her nesnede ayrıca saklamak bellek israfıdır; böyle sabitler `static final` yapılır.

4.1.4.4 İlklendirici blokları ve ilklendirme sırası

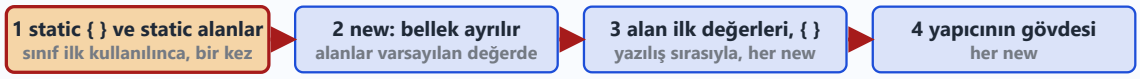
Bir alana ilk değer vermenin üç yolu vardır: tanım satırında, yapıcıda ve ilklendirici bloğunda. Üçüncüsü, tek satıra sığmayan hazırlık işleri için kullanılır.

Tanım 4.1.10

İlklendirici blok (*initializer block*), sınıfın gövdesine doğrudan yazılan bir kod bloğudur. **Sınıfa ait ilklendirici** (`static { ... }`), sınıf belleğe ilk kez yüklenirken yalnızca bir kez çalışır. **Örneğe ait ilklendirici** (`{ ... }`), her nesne oluşturulduğunda yapıcının gövdesinden önce çalışır.

Sınıfa ait ilklendirici, `static` alanlara karmaşık bir ilk değer vermek için kullanılır; örneğin bir tabloyu döngüyle doldurmak için. Örneğe ait ilklendirici ise bütün yapıcılarda ortak olan işleri tek bir yerde toplar. Bir nesne oluşturulurken adımların sırası şudur:

1. Sınıf ilk kez kullanılıyorsa `static` alanların ilk değerleri ve `static` bloklar, yazılış sırasıyla bir kez çalışır.
2. `new` nesne için bellek ayırır; bütün alanlar varsayılan değer alır.
3. Alanların tanım satırındaki ilk değerleri ve örnek ilklendirici blokları, yazılış sırasıyla çalışır.
4. Yapıcının gövdesi çalışır.



Şekil 4.1.8: Nesne oluşturma sırası. Birinci kutu yalnızca sınıf ilk kez kullanılırken, öteki üçü her `new`'de çalışır.

Üçüncü adımdaki "yazılış sırası" önemlidir. Tanım satırındaki ilk değerler ve `{ }` blokları ayrı gruplar olarak değil, sınıfta yukarıdan aşağıya hangi sırayla yazılmışlarsa o sırayla çalışır. Bir ilklendirici bloğu bir alan tanımından önce yazılmışsa önce blok çalışır. Aşağıdaki `Kopek` sınıfında iki tür ilklendirici de vardır.

```
public class Kopek {
  String adi;                 // alan
  Kopek(String adi) {         // yapıcı metot
    this.adi = adi;
    System.out.println(adi);
  }
  String havla() {            // örneğe ait metot
    return adi + " haw haw haw!";
  }
  static {                    // sınıfa ait ilklendirici
    System.out.println("Köpekler!");
  }
  {                           // örneğe ait ilklendirici
    System.out.println("Köpek oluşturuluyor.");
  }
  public static void main(String... strings) {
    var d1 = new Kopek("Kara");
    var d2 = new Kopek("Gece");
    d2.havla();
  }
}
```

```
Köpekler!
Köpek oluşturuluyor.
Kara
Köpek oluşturuluyor.
Gece
```

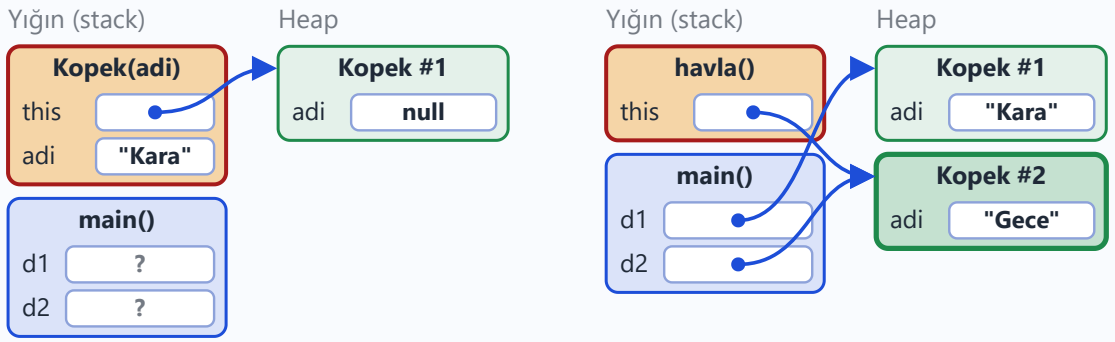
`main`'deki iki yazım yeni olabilir. `var` (Java 10'dan beri) yerel değişkenin tipini sağdaki ifadeden çıkarır: `var d1 = new Kopek("Kara");` satırı `Kopek d1 = ...` ile aynıdır. `String... strings` ise `String[] strings` yerine geçen değişken sayıda parametre yazımıdır; `main` için ikisi eşdeğerdir.

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	11: `System.out.println("Köpekler!");`	static ilklendirici	—	`Köpekler!`

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
2	17: `var d1 = new Kopek("Kara");`	main()	d1=?, d2=?	—
3	14: `System.out.println("Köpek oluşturuluyor.");`	main() → Kopek("Kara")	d1=?, d2=?	`Köpek oluşturuluyor.`
4	4: `this.adi = adi;`	main() → Kopek("Kara")	adi="Kara"	—
5	5: `System.out.println(adi);`	main() → Kopek("Kara")	adi="Kara"	`Kara`
6	18: `var d2 = new Kopek("Gece");`	main()	d1=→#1, d2=→#2	`Köpek oluşturuluyor.` ⏎ `Gece`
7	19: `d2.havla();` 8: `return adi + " haw haw haw!";`	main() → havla()	d1=→#1, d2=→#2	—

Tablo 4.1.7: `Kopek` sınıfında ilklendiricilerin ve yapıcıların çalışma sırası.

Birinci adımda `main` daha başlamadan `Kopek` sınıfı yüklenir, çünkü `main` bu sınıfın metodudur. Sınıf yüklenirken `static` blok çalışır ve `Köpekler!` yazılır. İkinci adımda `new Kopek("Kara")` nesne için yer ayırır; `adi` alanı `null`'dır. Üçüncü adımda yapıcının gövdesinden önce örnek ilklendirici çalışır: `Köpek oluşturuluyor.` Dördüncü ve beşinci adımlarda yapıcının gövdesi alanı doldurur ve adı yazar: `Kara`.



Şekil 4.1.9: Solda örnek ilklendirici çalışmış, yapıcının çerçevesi açık; nesnenin `adi` alanı henüz `null`. Sağda iki nesne de hazır; `havla()` 2 numaralı nesne üzerinde çalışıyor.

Altıncı adım ikinci nesne için ilklendiriciyi ve yapıcıyı yeniden çalıştırır: `Köpek oluşturuluyor.` ve `Gece`. `static` blok bu kez çalışmaz, çünkü sınıf zaten yüklüdür. Yedinci adımda `d2.havla()` çağrılır. Metot `"Gece haw haw haw!"` metnini döndürür, ama `main` bu değeri ne bir değişkene atar ne de yazdırır. Döndürülen değer kullanılmadan kaybolur ve ekrana hiçbir şey çıkmaz. Bu metnin ekranda görünmesi için `System.out.println(d2.havla());` yazmak gerekirdi.

Örnek 4.1.1

Aşağıdaki program ne yazdırır?

```
class A {
  static { System.out.print("S"); }
  { System.out.print("I"); }
  A() { System.out.print("C"); }
}
public class Problem3 {
  public static void main(String[] args) {
    System.out.print("M");
    new A();
    new A();
    System.out.println();
  }
}
```

`main` önce `M` yazar. `A` sınıfı `main`'in sınıfı olmadığı için program başlarken yüklenmez; ilk `new A()` ifadesinde yüklenir. O anda `static` blok bir kez çalışır: `S`. Ardından ilk nesne için örnek ilklendirici (`I`) ve yapıcı (`C`) çalışır. İkinci `new A()`'da sınıf zaten yüklüdür; yalnızca `I` ve `C` yazılır. Sonuç:

```
MSICIC
```

4.1.5 Metotlar

Metotlar sınıfın davranışını tanımlar. Metot tanımlama, parametre, dönüş değeri ve çağrı yığını temel programlamadan bilinir. Burada asıl soru metotların nesnelerle ilişkisidir: bir metot hangi nesnenin verisiyle çalışır ve bunu nasıl bilir?

4.1.5.1 Örneğe ait ve sınıfa ait metotlar

Alanlarda olduğu gibi metotlarda da iki tür vardır. **Örneğe ait metot** bir nesneyle ilişkilidir ve o nesnenin alanlarını kullanır: `telefon.aramaYap(...)`. **Sınıfa ait metot** (`static` metot) sınıfın kendisiyle ilişkilidir ve sınıf adı üzerinden çağrılır: `CepTelefonu.sistemiGuncelle(...)`.

```
public class CepTelefonu {
  private String model;
  private static String isletimSistemi = "Sürüm 1";

  // Örneğe ait metot
  public void aramaYap(String numara) {
    System.out.println(model + " arıyor: " + numara);
  }

  // Örneğe ait metot
  public void mesajGonder(String alici, String mesaj) {
    System.out.println(alici + " <- " + mesaj);
  }

  // Sınıfa ait metot (static)
  public static void sistemiGuncelle(String yeniSurum) {
    isletimSistemi = yeniSurum;
  }

  public static void main(String[] args) {
    CepTelefonu telefon = new CepTelefonu();
    telefon.model = "A54";
    telefon.aramaYap("5551234567");
    telefon.mesajGonder("Ayşe", "Geliyorum");
    CepTelefonu.sistemiGuncelle("Sürüm 2");
    System.out.println(isletimSistemi);
  }
}
```

```
A54 arıyor: 5551234567
Ayşe <- Geliyorum
Sürüm 2
```

`aramaYap` ve `mesajGonder` örneğe ait metotlardır. `aramaYap`'ın gövdesindeki `model`, metodu çağıran telefonun modelidir. `sistemiGuncelle` ise `static`'tir: bütün telefonların ortak işletim sistemi sürümünü değiştirir ve belirli bir telefona bağlı değildir. Bu yüzden `static` bir metot nesneye özgü verilere, örneğin `model`'e, doğrudan erişemez. Hangi telefonun modeli? Sorunun cevabı yoktur, çünkü metot bir telefon üzerinden çağrılmamıştır. `main`'in son satırı `isletimSistemi`'ni sınıf adı yazmadan okur; `main` de aynı sınıfın `static` metodu olduğu için bu yazım geçerlidir.

4.1.5.2 Değer döndüren bir metot: Eposta

Bir metodun başlığı onun ne yaptığını özetler. Aşağıdaki `gonder` metodunun adı `gonder`'dir. Üç `String` parametresi vardır: alıcı, konu ve içerik. Dönüş tipi `boolean`'dır ve gönderimin başarılı olup olmadığını bildirir.

```
public class Eposta {
  public boolean gonder(String alici, String konu,
            String icerik) {
    // Gerçek gönderim yerine bir benzetim
    if (alici == null || !alici.contains("@")) {
      return false;               // başarısız
    }
    System.out.println(alici + " <- " + konu);
    return true;                    // başarılı
  }

  public static void main(String[] args) {
    Eposta e = new Eposta();
    boolean s1 = e.gonder("ali@ornek.com", "Ödev", "Ek");
    boolean s2 = e.gonder("ali.ornek.com", "Ödev", "Ek");
    System.out.println(s1 + " " + s2);
  }
}
```

```
ali@ornek.com <- Ödev
true false
```

Metot gerçek bir e-posta göndermez; gönderimi bir benzetimle temsil eder. Alıcı adresi `null`'sa ya da `@` içermiyorsa `false` döndürür. Koşuldaki `||` kısa devre işlecidir: `alici == null` doğruysa sağ taraf hiç hesaplanmaz. Sıra ters yazılsaydı `null` bir adres için `alici.contains("@")` çağrısı `NullPointerException` fırlatırdı. Adres geçerliyse metot gönderimi yazar ve `true` döndürür. `main` iki gönderim dener; ikinci adreste `@` yerine nokta vardır, bu yüzden ikinci çağrı ekrana bir şey yazmadan `false` döndürür.

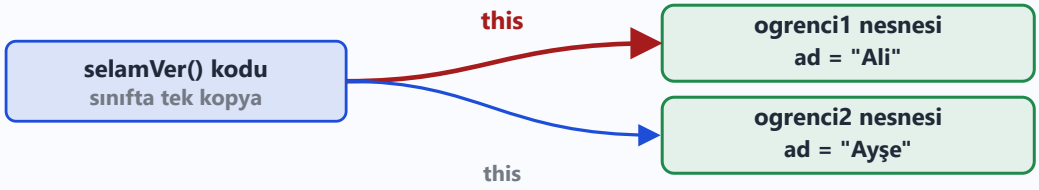
`boolean` döndüren bir metodun her yolu bir `return` ile bitmelidir. Gövdesi henüz yazılmamış, yalnızca yorum içeren bir metot bile derlenmez:

```
public class EksikReturn {
  public boolean gonder(String alici, String konu, String icerik) {
    // daha sonra yazılacak
  }
}
```

```
EksikReturn.java:4: error: missing return statement
  }
  ^
1 error
```

4.1.5.3 Tek kod, birçok nesne

Örneğe ait alanlar ve metotlar sınıf içinde `static` kelimesi olmadan tanımlanır. Her nesne alanların kendi kopyasını taşır; bu yüzden nesneler farklı verilerle çalışabilir. Örneğe ait bir üyeyi kullanmak için önce bir nesne gerekir: önce `new`, sonra `nesne.metot()`. Metodun kodu ise bütün nesneler için tektir. Her çağrıda hangi nesnenin verisiyle çalışılacağını `this` belirler.



Şekil 4.1.10: `selamVer()` 'in kodu sınıfta tek kopyadır. `ogrenci1.selamVer()` çağrısında `this` birinci nesneyi, `ogrenci2.selamVer()` çağrısında ikinci nesneyi gösterir.

	Örneğe ait	`static`
Kime ait?	Nesneye	Sınıfa
Çağrı	`nesne.metot()`	`Sinif.metot()`
Nesne gerekir mi?	Evet	Hayır
`this` var mı?	Evet	Hayır
Örnek	`str.length()`	`Math.sqrt(x)`

Tablo 4.1.8: Örneğe ait ve `static` metotların karşılaştırması.

Tablonun son satırı tanıdık iki çağrıyı karşılaştırır. `str.length()` bir metnin uzunluğunu verir; hangi metnin uzunluğu olduğunu `str` belirler. `Math.sqrt(x)` ise bir sayının karekökünü verir. Bu iş hiçbir `Math` nesnesine bağlı değildir; gerekli bütün bilgi `x` parametresindedir. Bir metodun `static` olup olmayacağına bu ölçüye bakarak karar verilir: metot nesnenin alanlarını kullanıyorsa örneğe ait, yalnızca parametreleriyle çalışıyorsa `static` olabilir.

4.1.5.4 Nokta işleci ve nesne parametresi

Nesne oluşturulduktan sonra alanlarına ve metotlarına nokta işleciyle erişilir: `nesne.metot(...)`, `nesne.alan`. Bir metot parametre olarak başka bir nesne de alabilir. Bu durumda metodun içinde iki nesne vardır: metodun çağrıldığı nesne ve parametre olarak gelen nesne.

```
public class Telefon {
  String numara;
  public Telefon(String numara) {
    this.numara = numara;
  }
  public void aramaYap(Telefon telefon) {
    System.out.println(numara + " arıyor: " + telefon.numara);
  }
  public void mesajGonder(Telefon telefon, String mesaj) {
    System.out.println(telefon.numara + " <- " + mesaj);
  }
  public void kameraAc() {
    System.out.println(numara + " kamera açtı.");
  }
  public static void main(String[] args) {
    Telefon telefon1 = new Telefon("5551234567");
    Telefon telefon2 = new Telefon("5051234567");
    telefon1.aramaYap(telefon2);
    telefon2.mesajGonder(telefon1, "Merhaba!");
    telefon1.kameraAc();
  }
}
```

```
5551234567 arıyor: 5051234567
5551234567 <- Merhaba!
5551234567 kamera açtı.
```

`telefon1.aramaYap(telefon2)` çağrısında metodun gövdesindeki yalın `numara`, metodun çağrıldığı nesnenin, yani `telefon1`'in numarasıdır. `telefon.numara` ise parametre olarak gelen nesnenin, yani `telefon2`'nin numarasıdır. Birinci satır bu yüzden "5551234567 arıyor: 5051234567" biçimindedir.

İkinci çağrı dikkatle okunmalıdır. `telefon2.mesajGonder(telefon1, "Merhaba!")` çağrısında metodu çağıran `telefon2`'dir, ama gövde yalnızca parametredeki telefonun numarasını kullanır: `telefon.numara`, yani `telefon1`'in numarası. Çıktıda 5551234567 görünmesinin nedeni budur. Metodun adı "mesaj gönder" olsa da yazdırılan numara alıcınındır; göndereni gösteren `numara` gövdede hiç kullanılmamıştır. Bir metodun gövdesinde yalın alan adı ile parametre üzerinden erişilen alan adı farklı nesnelere aittir; bu ayrım gözden kaçarsa yanlış nesnenin verisi okunur.

Parametre olarak gelen nesnenin `private` alanlarına da erişilebilir, yeter ki erişen kod aynı sınıfın içinde olsun. `numara` burada paket erişimlidir, ama `private` olsaydı `telefon.numara` yazımı yine derlenirdi. Erişim kuralı nesne başına değil, sınıf başınadır.

4.1.5.5 Dikdörtgen sınıfı

Prosedürel ve nesne yönelimli yaklaşımın karşılaştırmasındaki dikdörtgen örneği şimdi tam bir sınıf olarak yazılabilir.

```
public class Dikdortgen {
  private double uzunluk;     // dikdörtgenin uzunluğu
  private double genislik;    // dikdörtgenin genişliği

  // Yapıcı metot
  public Dikdortgen(double uzunluk, double genislik) {
    this.uzunluk = uzunluk;
    this.genislik = genislik;
  }

  public double alanHesapla() {      // alanı hesaplar
    return uzunluk * genislik;
  }

  public double cevreHesapla() {     // çevreyi hesaplar
    return 2 * (uzunluk + genislik);
  }
}
```

Alanlar `private`'tır; dışarıdan yalnızca metotlarla kullanılır. Yapıcı iki kenarı alır ve nesneye kaydeder. `alanHesapla()` ve `cevreHesapla()` parametre almaz, çünkü ihtiyaç duydukları veriler nesnenin içindedir. Prosedürel karşılığı `alanHesapla(5.0, 3.0)` olurdu; nesneyle yazımı `dikdortgen.alanHesapla()` olur. Aşağıdaki test sınıfı iki dikdörtgen oluşturur.

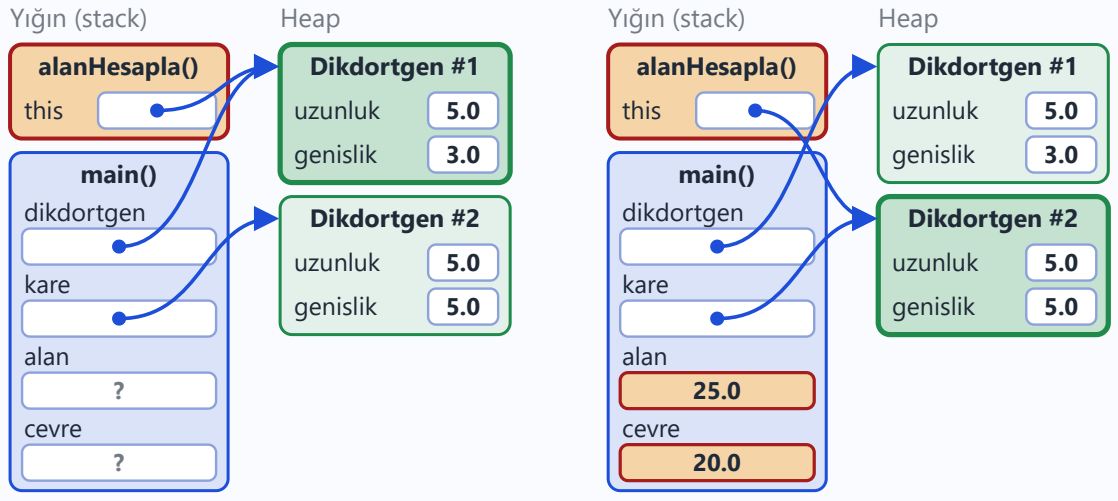
```
public class DikdortgenTesti {
  public static void main(String[] args) {
    Dikdortgen dikdortgen = new Dikdortgen(5.0, 3.0);
    Dikdortgen kare = new Dikdortgen(5.0, 5.0);
    double alan = dikdortgen.alanHesapla();
    double cevre = dikdortgen.cevreHesapla();
    System.out.println("Alanı: " + alan
        + " Çevresi: " + cevre);
    alan = kare.alanHesapla();
    cevre = kare.cevreHesapla();
    System.out.println("Alanı: " + alan
        + " Çevresi: " + cevre);
  }
}
```

```
Alanı: 15.0 Çevresi: 16.0
Alanı: 25.0 Çevresi: 20.0
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	3: `Dikdortgen dikdortgen = new Dikdortgen(5.0, 3.0);` 4: `Dikdortgen kare = new Dikdortgen(5.0, 5.0);`	main()	dikdortgen=→#1, kare=→#2	—
2	5: `double alan = dikdortgen.alanHesapla();`	main() → alanHesapla()	dikdortgen=→#1, kare=→#2	—
3	6: `double cevre = dikdortgen.cevreHesapla();`	main() → cevreHesapla()	alan=15.0	—
4	7: `System.out.println("Alanı: " + alan` 8: `+ " Çevresi: " + cevre);`	main()	alan=15.0, cevre=16.0	`Alanı: 15.0` `Çevresi: 16.0`
5	9: `alan = kare.alanHesapla();` 10: `cevre = kare.cevreHesapla();`	main() → alanHesapla()	alan=25.0, cevre=20.0	—
6	11: `System.out.println("Alanı: " + alan` 12: `+ " Çevresi: " + cevre);`	main()	alan=25.0, cevre=20.0	`Alanı: 25.0` `Çevresi: 20.0`

Tablo 4.1.9: `DikdortgenTesti`'nin izlenmesi.

Birinci adımda iki nesne oluşur; her birinin kendi `uzunluk` ve `genislik` değerleri vardır. İkinci adımda `dikdortgen.alanHesapla()` çağrılır. Metodun çerçevesinde `this` 1 numaralı nesneyi gösterir ve metot 5.0 × 3.0 = 15.0 döndürür. Üçüncü adımda `cevreHesapla()` aynı nesne için 2 × (5.0 + 3.0) = 16.0 bulur. Dördüncü adımda iki `double` değer metne eklenip yazılır; `double` değerler `+` ile birleştirilirken ondalık kısımla birlikte, `15.0` biçiminde yazılır.



Şekil 4.1.11: Solda `dikdortgen.alanHesapla()` çalışıyor: `this` 1 numaralı nesneyi gösteriyor. Sağda aynı metot `kare` üzerinden çağrılmış; `this` 2 numaralı nesneyi gösteriyor, `alan` ve `cevre` yeni değerlerini almış.

Beşinci adımda aynı iki metot `kare` üzerinden çağrılır. `this` bu kez 2 numaralı nesneyi gösterir: 5.0 × 5.0 = 25.0 ve 2 × (5.0 + 5.0) = 20.0. Bu değerler `alan` ve `cevre` değişkenlerinin eski değerlerinin üzerine yazılır. Altıncı adımda ikinci satır yazdırılır. `println` çağrısının iki satıra bölünmüş olması bir sorun değildir: Java'da bir deyim noktalı virgül gelene kadar sürer.

Her iki metodun maliyeti sabittir: birkaç aritmetik işlem yapar ve ek bellek kullanmaz. Nesneye bir alan eklemek, örneğin dikdörtgenin rengini tutmak, bu metotları etkilemez. Prosedürel sürümde ise renk bilgisini taşımak için her çağrıya yeni bir parametre eklemek gerekirdi.

4.1.6 static üyeler

`static` kelimesi alanlarda da metotlarda da karşımıza çıktı. Kuralları bir araya toplamak, en sık karşılaşılan derleme hatasının nedenini de açıklar.

4.1.6.1 Nesnesiz erişim

Tanım 4.1.11

`static` **üye** (*static member*), `static` anahtar sözcüğüyle tanımlanan ve herhangi bir nesneye değil sınıfın kendisine ait olan alan ya da metottur. `static` üyeye erişmek için nesne oluşturmak gerekmez; sınıf adı ve nokta kullanılır: `SinifAdi.alanAdi`, `SinifAdi.metotAdi(argumanlar)`.

```
public class Matematik {
  // Statik toplama metodu
  public static int toplama(int sayi1, int sayi2) {
    return sayi1 + sayi2;
  }

  public static void main(String[] args) {
    int sonuc = Matematik.toplama(5, 3);
    System.out.println("Toplama Sonucu: " + sonuc);
    System.out.println(Math.max(4, 9));
    System.out.println(Math.sqrt(16.0));
  }
}
```

```
Toplama Sonucu: 8
9
4.0
```

`Matematik.toplama(5, 3)` çağrısı hiçbir `Matematik` nesnesi oluşturmadan yapılır. `main` aynı sınıfın içinde olduğu için sınıf adı yazılmadan `toplama(5, 3)` da yazılabilirdi. `Math.max` ve `Math.sqrt` aynı türden çağrılardır: `Math` sınıfının `static` metotları. Java'nın hazır kütüphanesinde bunun çok örneği vardır: `Math.PI`, `Integer.parseInt("12")`, `Integer.MAX_VALUE`. Her programın başladığı `main` de `static`'tir. Program başladığında JVM (Java Sanal Makinesi) henüz hiçbir nesne oluşturmamıştır; `main`'i nesnesiz çağırabilmesi için metodun sınıfa ait olması gerekir.

`static` bir üyeye nesne üzerinden de erişilebilir: `h1.toplamHesapSayisi`. Java bu yazımı kabul eder, ama tercih edilmez. Kodu okuyan kişi üyenin nesneye ait olduğunu sanır ve `h1`'in sayacının `h2`'ninkinden farklı olabileceğini düşünür. Derleyici bu yazımı `BankaHesabi.toplamHesapSayisi` olarak işler; hangi nesnenin kullanıldığı önemsizdir. `-Xlint` seçeneğiyle derlenirse `static variable should be qualified by type name` uyarısı da verir.

4.1.6.2 static bağlamın kuralları

Örneğe ait bir metot hem örnek üyelerini hem `static` üyeleri doğrudan kullanabilir. Böyle bir metot her zaman bir nesne üzerinden çağrılır; o nesne `this`'tir ve örnek alanları onun alanlarıdır. `static` sınıf alanı da zaten tektir. `static` bir metodun ise `this`'i yoktur. Bu metot örnek üyelerine yalnızca açıkça verilmiş bir nesne üzerinden ulaşabilir. Tablo 4.1.10 dört durumu özetler.

Nereden	Örnek alanı / metodu	`static` alan / metot
Örneğe ait metottan	✓ doğrudan (`ad`, `this.ad`)	✓ doğrudan
`static` metottan (`main` dahil)	✗ ancak bir nesne üzerinden (`ogr.ad`)	✓ doğrudan

Tablo 4.1.10: Hangi metot hangi üyeyi doğrudan kullanabilir?

Kuralın ihlali, nesne yönelimli programlamaya yeni başlayanların en sık gördüğü derleme hatasıdır. `main`'den örneğe ait bir metodu doğrudan çağırmak:

```
public class Selam {
  private String ad = "Ali";
  public void selamVer() {
    System.out.println("Merhaba " + ad);
  }
  public static void main(String[] args) {
    selamVer();
  }
}
```

```
Selam.java:7: error: non-static method selamVer() cannot be referenced from a static
context
    selamVer();
    ^
1 error
```

`static` bir metottan örneğe ait bir alanı doğrudan okumak:

```
public class Selam2 {
  private String ad = "Ali";
  public static void yaz() {
    System.out.println(ad);
  }
}
```

```
Selam2.java:4: error: non-static variable ad cannot be referenced from a static context
    System.out.println(ad);
                       ^
1 error
```

İki iletide de aynı ifade geçer: `cannot be referenced from a static context`. "`static` bağlam", `this`'in olmadığı yer demektir. `selamVer()` çağrısı aslında `this.selamVer()`'dir, `ad` okuması da `this.ad`'dır. `main`'de `this` olmadığı için derleyici hangi nesnenin metodunun ya da alanının kastedildiğini bilemez. Açıkça `this` yazmak da bir şey değiştirmez; ileti bu kez `non-static variable this cannot be referenced from a static context` olur.

Çözümün iki yolu vardır ve hangisinin seçileceği tasarıma bağlıdır. Metot gerçekten bir nesnenin verisine ihtiyaç duyuyorsa çağrı bir nesne üzerinden yapılır: `new Selam().selamVer();`. Metot hiçbir nesnenin verisini kullanmıyorsa `static` yapılır. İkinci yol yalnızca metodun gerçekten nesneden bağımsız olduğu durumda doğrudur. Derleme hatasını susturmak için bütün metotları ve alanları `static` yapmak programı yeniden prosedürel tarza döndürür. Her nesnenin ayrı verisi olması gereken yerde tek bir ortak kopya kalır.

4.1.7 Referanslar ve this

Değişkenin nesneyi değil referansı tuttuğu bilgisi, ilk bakışta yalnızca bir ayrıntı gibi görünür. Ama atamanın, karşılaştırmanın ve metotlara nesne aktarmanın nasıl davrandığını bu ayrıntı belirler.

4.1.7.1 Referans atama ve takma ad

Java'da nesneler referanslar aracılığıyla işlenir. Bir referans değişkenini başka bir değişkene atamak nesneyi kopyalamaz; yalnızca referansı kopyalar. Atamadan sonra iki değişken aynı nesneyi gösterir.

Tanım 4.1.12

Takma ad (*alias*), aynı nesneyi gösteren birden çok referans değişkeninin her biridir. Nesne bir tanedir; takma adlardan biri üzerinden yapılan değişiklik ötekilerden de görünür.

Gündelik hayatta da benzer durumlar vardır. Kütüphane rafındaki etiket kitabın kendisi değildir; kitabın yerini gösterir. Bir arabanın iki anahtarı olabilir; iki anahtar da aynı arabayı açar. Anahtarlardan biriyle arabaya binip koltuğu geri itmek, öteki anahtarla binen kişinin de geri itilmiş koltuk bulması demektir. Aşağıdaki program `marka` alanı `private` olan, erişimci ve değiştiricisi bulunan bir `Araba` sınıfıyla bu durumu gösterir.

```
public class Araba {
  private String marka;

  public Araba(String marka) {
    this.marka = marka;
  }

  public String getMarka() {
    return marka;
  }

  public void setMarka(String marka) {
    this.marka = marka;
  }

}
```

```
public static void main(String[] args) {
  Araba araba1 = new Araba("Toyota");
  Araba araba2 = araba1;
  System.out.println("Araba 1: " + araba1.getMarka());
  System.out.println("Araba 2: " + araba2.getMarka());
  araba2.setMarka("Ford");
  System.out.println("Araba 1: " + araba1.getMarka());
  araba1 = null;
  System.out.println("Araba 2: " + araba2.getMarka());
}
```

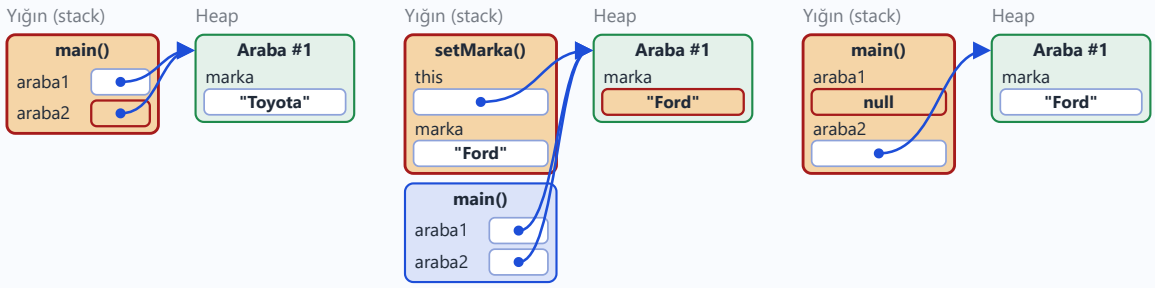
```
Araba 1: Toyota
Araba 2: Toyota
Araba 1: Ford
Araba 2: Ford
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	2: `Araba araba1 = new Araba("Toyota");`	main()	araba1=→#1, araba2=?	—
2	3: `Araba araba2 = araba1;`	main()	araba1=→#1, araba2=→#1	—

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
3	4: `System.out.println("Araba 1: " + araba1.getMarka());` 5: `System.out.println("Araba 2: " + araba2.getMarka());`	main()	araba1=→#1, araba2=→#1	`Araba 1: Toyota ⏎` `Araba 2: Toyota`
4	6: `araba2.setMarka("Ford");`	main() → setMarka("Ford")	araba1=→#1, araba2=→#1	—
5	7: `System.out.println("Araba 1: " + araba1.getMarka());`	main()	araba1=→#1, araba2=→#1	`Araba 1: Ford`
6	8: `araba1 = null;`	main()	araba1=null, araba2=→#1	—
7	9: `System.out.println("Araba 2: " + araba2.getMarka());`	main()	araba1=null, araba2=→#1	`Araba 2: Ford`

Tablo 4.1.11: Referans atamanın izlenmesi. Satır numaraları `main`'in satırlarıdır.

Birinci adımda heap'te tek bir nesne oluşur ve `araba1` onu gösterir. İkinci adımdaki `Araba araba2 = araba1;` satırı yeni bir nesne oluşturmaz. `araba1`'in kutusundaki referans `araba2`'nin kutusuna kopyalanır. Heap'te hâlâ tek nesne vardır; artık ona giden iki ok bulunur. Üçüncü adımda iki değişken üzerinden de marka okunur ve ikisi de `Toyota` verir: okunan alan aynı alandır.



Şekil 4.1.12: Soldan sağa: `araba2 = araba1` sonrası iki ok tek nesneye gider; `araba2.setMarka("Ford")` ortak nesneyi değiştirir; `araba1 = null` yalnızca bir oku keser, nesne `araba2` üzerinden yaşamaya devam eder.

Dördüncü adımda `araba2.setMarka("Ford")` çağrılır. Metodun `this`'i `araba2`'nin gösterdiği nesnedir, yani tek nesne. Beşinci adımda `araba1` üzerinden okunan marka da `Ford`'dur: değişiklik paylaşılmıştır. Bu davranış iki değişkenin aynı nesneyi gösterdiği bilinmezse şaşırtıcıdır. Program `araba2`'yi değiştirdiğini düşünür, ama aslında `araba1`'in gösterdiği nesneyi de değiştirmiştir.

Altıncı adımda `araba1 = null;` çalışır. Bu satır nesneyi silmez; yalnızca `araba1` değişkeninin nesneyle bağını koparır. Nesne yerinde durur ve `araba2` onu göstermeye devam eder. Yedinci adımda `araba2` üzerinden okunan marka `Ford`'dur.

Atamanın maliyeti de bu yüzden sabittir. Nesnenin kaç alanı olursa olsun `b = a` yalnızca bir referans kopyalar. Nesnenin gerçekten kopyalanması gerekiyorsa yeni bir nesne oluşturulup alanlar tek tek aktarılır; Java bunu atamayla kendiliğinden yapmaz.

Aynı kural metotlara nesne aktarırken de geçerlidir. Java bütün argümanları değer ile aktarır. Argüman bir nesne referansıysa kopyalanan değer referanstır. Metot bu kopya üzerinden ortak nesneyi değiştirebilir, ama parametresine yeni bir nesne atamak çağıranın değişkenini etkilemez:

```
class Kutu { int deger; }
public class Problem4 {
  static void degistir(Kutu k) {
    k.deger = 100;
    k = new Kutu();
    k.deger = 7;
  }
  public static void main(String[] args) {
    Kutu x = new Kutu();
    x.deger = 1;
    degistir(x);
    System.out.println(x.deger);
  }
}
```

```
100
```

`degistir(x)` çağrısında `k`, `x`'teki referansın kopyasını alır. `k.deger = 100;` ortak nesneye yazar; `x` de aynı nesneyi gösterdiği için değişiklik `main`'den görünür. `k = new Kutu();` ise yalnızca yerel kopyayı yeni bir nesneye yöneltir. Bundan sonraki `k.deger = 7;` yeni nesneye yazar; `x`'in nesnesi etkilenmez. Program `100` yazar.

4.1.7.2 null ve NullPointerException

`null`, bir referans değişkeninin hiçbir nesneyi göstermediği anlamına gelir. `null` bir referans üzerinden alan okumak ya da metot çağırmak, okunacak bir nesne olmadığı için çalışma zamanında hataya yol açar.

```
public class Bos {
  String ad;

  public static void main(String[] args) {
    Bos b = null;
    System.out.println(b.ad.length());
  }
}
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot read field "ad" because
"<local1>" is null
        at Bos.main(Bos.java:6)
```

Program derlenir, çünkü `b`'ye bir değer verilmiştir: `null`. Çalışırken 6. satırda `b.ad` okunmak istenir ve `NullPointerException` fırlatılır. JDK 15'ten beri ileti, hatanın yerini varsayılan olarak ayrıntılı söyler: `ad` alanı okunamamıştır, çünkü değişken `null`'dır. İletideki `"<local1>"` yazımı, `main`'in 1 numaralı yerel değişkenini, yani `b`'yi gösterir. Program hata ayıklama bilgisiyle (`javac -g`) derlenseydi değişkenin adı görünürdü: `because "b" is null`.

Yerel değişkene hiç değer verilmeden kullanılırsa durum farklıdır: program derlenmez bile (`variable sb might not have been initialized`). `NullPointerException` ise derleyicinin yakalayamadığı, çalışma sırasında ortaya çıkan bir hatadır. Bir alan bunun en sık kaynağıdır: `String` tipindeki bir alana değer verilmemişse varsayılan değeri `null`'dır ve üzerinden çağrılan ilk metot programı durdurur. Önlem, referansı kullanmadan önce denetlemektir:

```
public class Bos2 {
  String ad;

  public static void main(String[] args) {
    Bos2 b = new Bos2();
    if (b.ad != null) {
      System.out.println(b.ad.length());
    } else {
      System.out.println("ad henüz verilmemiş");
    }
    System.out.println(b.ad.length());
  }
}
```

```
ad henüz verilmemiş
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()"
because "<local1>.ad" is null
        at Bos2.main(Bos2.java:11)
```

`if` bloğu `ad`'ın `null` olduğunu görür ve güvenli yolu seçer. Son satır ise aynı alanı denetimsiz kullanır. Bu kez ileti `"<local1>.ad"` der: `b` değil, `b`'nin `ad` alanı `null`'dır.



Şekil 4.1.13: `null` referans hiçbir nesneyi göstermez; `b.ad` okunmak istenince `NullPointerException` fırla-

tılır. Sağda, hiçbir referansın göstermediği nesneyi çöp toplayıcı siler.

Hiçbir referansın göstermediği nesneye programdan bir daha ulaşılamaz. Böyle nesnelerin belleğini JVM'in **çöp toplayıcısı** (*garbage collector*) zamanı gelince geri alır. Java'da nesneyi elle silmek gerekmez ve bunun bir yolu da yoktur. Referans atama izlemesinde `araba2` de `null` yapılsaydı nesneyi gösteren son referans kaybolur ve nesne çöp toplamaya uygun hâle gelirdi. Çöp toplayıcının ne zaman çalışacağı önceden bilinmez; program yalnızca nesnenin artık ulaşılamaz olduğundan emin olabilir.

4.1.7.3 this anahtar sözcüğü

Tanım 4.1.13

`this`, örneğe ait bir metodun ya da yapıcının içinde, metodun çağrıldığı (yapıcıda ise oluşturulmakta olan) nesneyi gösteren referanstır. `static` metotlarda `this` yoktur.

`this` en sık yapıcılarda, alanın adı ile parametrenin adı aynı olduğunda kullanılır: `this.ad = ad;`. Bu durumda yalın `ad` adı parametreyi gösterir, çünkü en yakın tanım parametredir. Parametre aynı adlı alanı **gölgeler**. Alana ulaşmak için `this.ad` yazılır. `this` unutulursa ne olur?

```
public class Golge {
  private String ad;

  public Golge(String ad) {
    ad = ad;            // this unutuldu!
  }

  public static void main(String[] args) {
    Golge g = new Golge("Ali");
    System.out.println(g.ad);
  }
}
```

```
null
```



Şekil 4.1.14: Yapıcıda `ad = ad;` parametreyi kendine atar; nesnenin `ad` alanı `null` kalır. `this.ad = ad;` yazılsaydı `"Ali"` nesnenin alanına giderdi.

`ad = ad;` satırının iki yanındaki `ad` da parametredir. Satır parametreyi kendine atar ve hiçbir şeyi değiştirmez. Nesnenin `ad` alanına hiç dokunulmaz; varsayılan değeri olan `null`'da kalır. Program `null` yazar. En tehlikelisi şudur: derleyici bu satır için hata vermez. Kod derlenir ve çalışır; hata ancak alan kullanıldığında, belki bambaşka bir yerde `NullPointerException` olarak ortaya çıkar.

Alan `final` olsaydı derleyici hatayı yakalardı. `private final String ad;` tanımında alan yapıcıda mutlaka değer almalıdır. `ad = ad;` alana değer vermediği için derleyici `variable ad might not have been initialized` iletisini verir. Hiç değişmeyecek alanları `final` yapmak bu yüzden yalnızca belgeleme değil, bir güvenlik önlemidir.

Gölgeleme yoksa `this` yazmak isteğe bağlıdır. Metot gövdesindeki yalın `model`, derleyici için `this.model` demektir. Aşağıdaki izleme bunu adım adım gösterir.

```
public class Ogrenci {
  private String ad;

  public Ogrenci(String ad) {
    this.ad = ad;
  }

  public void selamVer() {
    System.out.println("Merhaba, ben " + this.ad);
  }

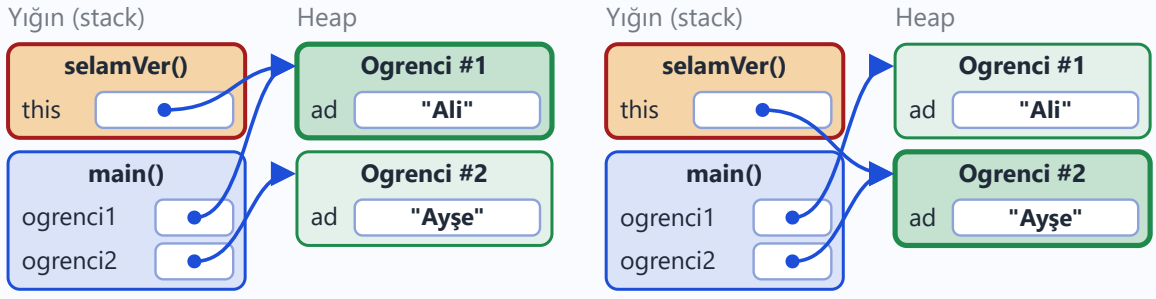
  public static void main(String[] args) {
    Ogrenci ogrenci1 = new Ogrenci("Ali");
    Ogrenci ogrenci2 = new Ogrenci("Ayşe");
    ogrenci1.selamVer();
    ogrenci2.selamVer();
  }
}
```

```
Merhaba, ben Ali
Merhaba, ben Ayşe
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	13: `Ogrenci ogrenci1 = new Ogrenci("Ali");` 4: `public Ogrenci(String ad) {` 5: `this.ad = ad;`	main() → Ogrenci("Ali")	ad="Ali"	—
2	14: `Ogrenci ogrenci2 = new Ogrenci("Ayşe");`	main()	ogrenci1=→#1, ogrenci2=→#2	—
3	15: `ogrenci1.selamVer();` 8: `public void selamVer() {` 9: `System.out.println("Merhaba, ben " + this.ad);`	main() → selamVer()	ogrenci1=→#1, ogrenci2=→#2	`Merhaba, ben Ali`
4	16: `ogrenci2.selamVer();` 8: `public void selamVer() {` 9: `System.out.println("Merhaba, ben " + this.ad);`	main() → selamVer()	ogrenci1=→#1, ogrenci2=→#2	`Merhaba, ben Ayşe`

Tablo 4.1.12: `this`'in her çağrıda hangi nesneyi gösterdiği.

Birinci adımda yapıcı çalışır; `this` yeni oluşturulan nesnedir ve `this.ad = ad;` ile alana `"Ali"` yazılır. İkinci adımda ikinci nesne aynı yapıcıyla oluşur; o sırada `this` 2 numaralı nesneyi gösterir. Üçüncü adımda `ogrenci1.selamVer()` çağrılır. Noktanın solundaki nesne metodun `this`'i olur ve `Merhaba, ben Ali` yazılır. Dördüncü adımda aynı kod, farklı bir `this` ile çalışır ve `Merhaba, ben Ayşe` yazar. Bu metotta `this.ad` yerine yalnızca `ad` yazmak aynı sonucu verirdi, çünkü metodun `ad` adlı bir parametresi yoktur.



Şekil 4.1.15: Solda `ogrenci1.selamVer()`: `this` 1 numaralı nesneyi gösterir. Sağda `ogrenci2.selamVer()`: aynı metot, ama `this` 2 numaralı nesneyi gösterir.

4.1.7.4 this ile nesneyi aktarmak

`this` bir referans olduğu için başka bir metoda argüman olarak da verilebilir. Bir nesne kendini başka bir nesneye tanıtmak istediğinde bu yol kullanılır.

```
public class Takim {
  private String ad;
  public Takim(String ad) {
    this.ad = ad;
  }
  public void oyuncuEkle(Oyuncu oyuncu) {
    oyuncu.takimaKatil(this);   // bu nesne
  }
  public String getAd() {
    return ad;
  }
  public static void main(String[] args) {
    Takim takim = new Takim("Giresunspor");
    Oyuncu oyuncu = new Oyuncu("Mert");
    takim.oyuncuEkle(oyuncu);
  }
}
```

Aynı dosyada, `public` olmayan ikinci bir sınıf:

```
class Oyuncu {
  private String ad;
  Oyuncu(String ad) {
    this.ad = ad;
  }
  void takimaKatil(Takim takim) {
    String t = takim.getAd();
    System.out.println(ad + " -> " + t);
  }
}
```

```
Mert -> Giresunspor
```

`takim.oyuncuEkle(oyuncu)` çağrısında metodun `this`'i `takim` nesnesidir. Metot gövdesindeki `oyuncu.takimaKatil(this)` bu referansı oyuncuya verir. `takimaKatil`'in `takim` parametresi artık aynı takım nesnesini gösterir ve `takim.getAd()` ile adı okur. Bu sırada iki çerçevede iki ayrı `this` vardır: `oyuncuEkle`'nin `this`'i takım, `takimaKatil`'in `this`'i oyuncudur. `takimaKatil`'deki yalın `ad` da oyuncunun adıdır. Çıktı `Mert -> Giresunspor` olur.

`Oyuncu` sınıfı `Takim.java` dosyasının içinde, `public` olmadan tanımlanmıştır. Bir dosyada yalnızca bir `public` sınıf bulunabilir, ama yardımcı sınıflar aynı dosyada yer alabilir. Bu sınıflar paket erişimlidir; aynı klasördeki öteki sınıflar onları kullanabilir.

4.1.7.5 this(...) ile yapıcı zinciri

Bir sınıfın birden çok yapıcısı varsa çoğu aynı işleri yapar. Aynı atamaları her yapıcıya kopyalamak yerine yapıcılar birbirine devredebilir.

Tanım 4.1.14

`this(...)`, bir yapıcının içinden aynı sınıfın başka bir yapıcısını çağıran yazımdır. Bu yolla kurulan yapıcı dizisine **yapıcı zinciri** (*constructor chaining*) denir. `this(...)` yapıcının ilk deyimi olmalıdır.

```
class Calisan {
  private String ad;
  private double maas;
  Calisan(String ad) {
    this(ad, 20000.0);   // aynı sınıfın yapıcısı
  }
  Calisan(String ad, double maas) {
    this.ad = ad;        // ilk değerler tek yerde
    this.maas = maas;
  }
}
```

```
Elif 20000.0
```

Ortak ilk değer atamaları tek bir yapıcıda toplanır; burada bu yapıcı `Calisan(String, double)`'dır. Öteki yapıcı eksik argümanı ekleyerek ona devreder: `this(ad, 20000.0)`. Maaş verilmeden oluşturulan her çalışan 20000.0 ile başlar. Varsayılan maaş değişirse tek bir sayı değişir. Atama kuralları değişirse, örneğin adın boş olmaması denetlenmek istenirse, denetim tek bir yapıcıya yazılır.

Zincirin iki kuralı vardır ve ikisini de derleyici denetler. `this(...)` yapıcının ilk satırında olmalıdır:

```
public class ThisIlk {
  int x;
  ThisIlk() { x = 1; this(2); }
  ThisIlk(int x) { this.x = x; }
}
```

```
ThisIlk.java:3: error: call to this must be first statement in constructor
  ThisIlk() { x = 1; this(2); }
                     ^
1 error
```

Yapıcılar birbirini döngüsel çağıramaz; böyle bir zincir hiç bitmezdi:

```
public class ThisDongu {
  ThisDongu() { this(1); }
  ThisDongu(int x) { this(); }
}
```

```
ThisDongu.java:3: error: recursive constructor invocation
  ThisDongu(int x) { this(); }
                     ^
1 error
```

Zincir kaç yapıcıdan geçerse geçsin tek bir nesne kurulur. Çağrılan yapıcının gövdesi önce biter; sonra çağıranın kalan satırları çalışır. Aşağıdaki izleme bunu gösterir.

```
class Kitap {
  String ad;
  String yazar;
  int sayfaSayisi;
  public Kitap(String ad) {
    this.ad = ad;
  }
  public Kitap(String ad, String yazar) {
    this(ad);  // yapıcı zincirleme
    this.yazar = yazar;
  }
}
// main içinde:
Kitap k2 = new Kitap("Nutuk", "Atatürk");
System.out.println(k2.ad + " / "
    + k2.yazar + " / " + k2.sayfaSayisi);
```

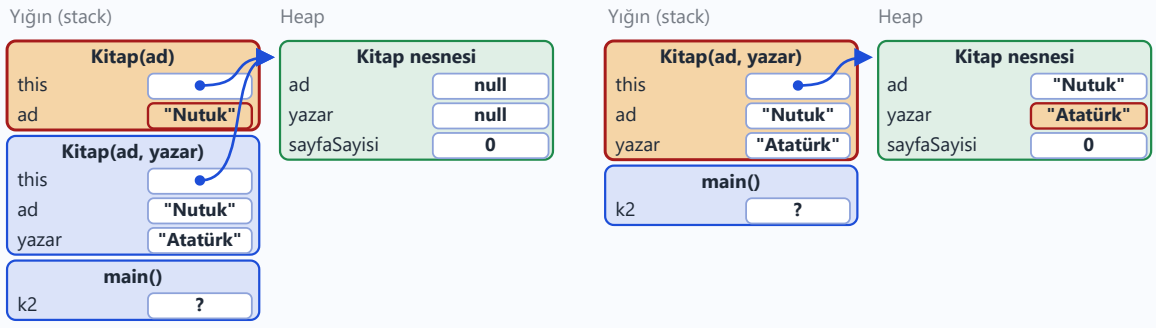
```
Nutuk / Atatürk / 0
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	14: `Kitap k2 = new Kitap("Nutuk", "Atatürk");`	main()	k2=?	—
2	8: `public Kitap(String ad, String yazar) {`	main() → Kitap(ad, yazar)	ad="Nutuk", yazar="Atatürk"	—
3	9: `this(ad); // yapıcı zincirleme`	main() → Kitap(ad, yazar) → Kitap(ad)	ad="Nutuk"	—

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
4	6: `this.ad = ad;`	main() → Kitap(ad, yazar) → Kitap(ad)	ad="Nutuk"	—
5	10: `this.yazar = yazar;`	main() → Kitap(ad, yazar)	ad="Nutuk", yazar="Atatürk"	—
6	14: `Kitap k2 = new Kitap("Nutuk", "Atatürk");`	main()	k2=→Kitap	—
7	15: `System.out.println(k2.ad + " / "` 16: `+ k2.yazar + " / " + k2.sayfaSayisi);`	main()	k2=→Kitap	`Nutuk / Atatürk / 0`

Tablo 4.1.13: `new Kitap("Nutuk", "Atatürk")` çağrısında yapıcı zincirinin izlenmesi.

Birinci adımda `new` heap'te yeni bir `Kitap` nesnesi ayırır; alanlar varsayılan değerlerini alır: `null`, `null`, `0`. İkinci adımda argümanlar iki `String` olduğu için iki parametreli yapıcı seçilir; `this` yeni nesneyi gösterir. Üçüncü adımda `this(ad)` çalışır ve tek parametreli yapıcı aynı nesne için çağrılır. Yığında artık iki yapıcı çerçevesi üst üste durur; ikisinin `this`'i de aynı nesnedir.



Şekil 4.1.16: Solda zincirin en derin anı: iki yapıcı çerçevesi açık, ikisinin `this`'i de aynı nesneyi gösteriyor. Sağda iç yapıcı bitmiş, dış yapıcı `yazar` alanını doldurmuş.

Dördüncü adımda iç yapıcı `this.ad = ad;` ile `ad` alanına `"Nutuk"` yazar ve biter. Beşinci adımda dış yapıcı kaldığı yerden sürer ve kalan satırını çalıştırır: `this.yazar = yazar;`. Altıncı adımda iki yapıcı da bitmiştir; `new` ifadesinin değeri olan referans `k2`'ye atanır. Yedinci adımda alanlar okunur. `sayfaSayisi`'na hiçbir yapıcı değer vermediği için varsayılan değeri yazılır: `Nutuk / Atatürk / 0`.

Bu sırada bir yan sonuç da vardır. Alanların tanım satırındaki ilk değerleri ve örnek ilklendiricileri, zincirin `this(...)` ile başlamayan yapıcısında, o yapıcının gövdesinden hemen önce bir kez çalışır. `this(...)` ile başlayan yapıcı bu işi zincirin sonundaki yapıcıya bırakır. Böylece zincir ne kadar uzun olursa olsun ilklendiriciler her nesne için tam bir kez çalışır.

4.1.8 Nesneleri karşılaştırma: == ve equals

İki `int` değişkeni `==` ile karşılaştırmak sayıları karşılaştırır. İki referans değişkeni `==` ile karşılaştırmak ise nesneleri değil, değişkenlerin kutularındaki referansları karşılaştırır. Soru "bu iki nesne aynı içerikte mi?" değil, "bu iki değişken aynı nesneyi mi gösteriyor?" olur.

Tanım 4.1.15

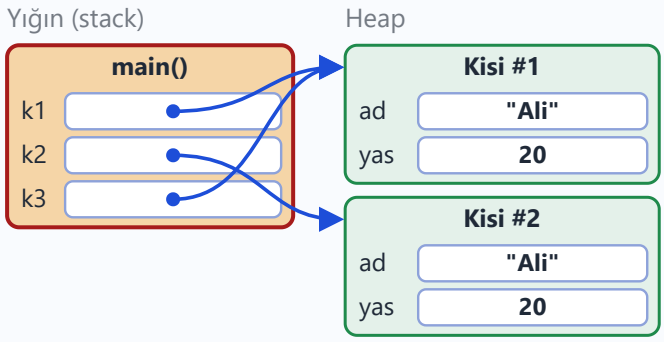
Referans eşitliği, iki referansın aynı nesneyi göstermesidir ve `==` ile sınanır. **İçerik eşitliği**, iki nesnenin anlamca aynı değeri taşımasıdır ve `equals()` metoduyla sınanır. `equals`'ın neyi "aynı" sayacağını sınıfı yazan programcı belirler.

Aşağıdaki programda `Kisi` sınıfı `equals` metodunu kendisi yazar: iki kişinin adı ve yaşı aynıysa onları eşit sayar. `Kisi0` sınıfı ise `equals` yazmaz. Programın son bölümü klavyeden bir sözcük okur ve onu bir metin sabitiyle karşılaştırır.

```
import java.util.Objects;
import java.util.Scanner;
class Kisi {
  private String ad;
  private int yas;
  Kisi(String ad, int yas) {
    this.ad = ad;
    this.yas = yas;
  }
  @Override
  public boolean equals(Object o) {
    if (this == o) return true;
    if (!(o instanceof Kisi)) return false;
    Kisi k = (Kisi) o;
    return yas == k.yas && Objects.equals(ad, k.ad);
  }
  @Override
  public int hashCode() {
    return Objects.hash(ad, yas);
  }
  @Override
  public String toString() {
    return ad + " (" + yas + ")";
  }
}
class Kisi0 {
  String ad;
  Kisi0(String ad) { this.ad = ad; }
}
public class Esitlik {
  public static void main(String[] args) {
    Kisi k1 = new Kisi("Ali", 20);
    Kisi k2 = new Kisi("Ali", 20);
    Kisi k3 = k1;
    System.out.println(k1 == k3);
    System.out.println(k1 == k2);
    System.out.println(k1.equals(k2));
    System.out.println(k1.hashCode() == k2.hashCode());
    System.out.println(k1);
    Kisi0 a = new Kisi0("Ali");
    Kisi0 b = new Kisi0("Ali");
    System.out.println(a.equals(b));
    Scanner klavye = new Scanner(System.in);
    String okunan = klavye.next();
    System.out.println(okunan == "Java");
    System.out.println(okunan.equals("Java"));
  }
}
```

Klavyeden `Java` girildiğinde program şunu yazar:

```
true
false
true
true
Ali (20)
false
false
true
```



Şekil 4.1.17: `k1` ve `k3` aynı `Kisi` nesnesini, `k2` aynı içerikli ayrı bir nesneyi gösterir.

İfade	Sonuç	Neden
`k1 == k3`	`true`	İki değişken aynı nesneyi gösterir.
`k1 == k2`	`false`	Farklı nesneler; içerikleri aynı olsa da referanslar farklı.
`k1.equals(k2)`	`true`	`equals` yeniden yazılmış; ad ve yaş aynı.

Tablo 4.1.14: Üç referans için karşılaştırma sonuçları.

`equals`, `hashCode`, `toString` ve `getClass` metotları her sınıfa Java'nın en üstteki sınıfı `Object`'ten gelir. `Object`'in `equals` sürümü yalnızca referansları karşılaştırır; yani `==` gibi davranır. `Kisi0` sınıfı `equals` yazmadığı için `a.equals(b)` `false` verir: iki ayrı nesne vardır ve `Object`'in sürümü içeriğe bakmaz. Bir sınıfın nesneleri içerikleriyle karşılaştırılacaksa `equals` o sınıfta yeniden yazılmalıdır.

`Kisi`'deki `equals` üç adımda çalışır. Önce `this == o` ile aynı nesne olup olmadığına bakar; öyleyse hemen `true` döner. Sonra `instanceof` ile gelen nesnenin bir `Kisi` olup olmadığını denetler; değilse, ya da `o` `null`'sa, `false` döner. Son adımda alanları karşılaştırır. `yas` bir `int` olduğu için `==` ile, `ad` bir `String` olduğu için `Objects.equals` ile karşılaştırılır. Bu yardımcı metot iki taraftan biri `null` olduğunda da güvenle çalışır. Parametrenin tipi `Kisi` değil `Object`'tir. Tip `Kisi` yazılsaydı metot `Object`'in `equals`'ını yeniden yazmış olmaz, yalnızca aşırı yüklemiş olurdu; `@Override` işareti bu hatayı derleme sırasında yakalar.

`equals` yeniden yazılınca `hashCode` da yazılmalıdır. Kural şudur: `equals`'a göre eşit olan iki nesnenin `hashCode` değerleri de eşit olmalıdır. Programdaki dördüncü satır bunu doğrular. Kural çiğnenirse `HashMap` ve `HashSet` gibi koleksiyonlar eşit nesneleri farklı yerlerde arar ve bulamaz. `Objects.hash(ad, yas)`, `equals`'ta kullanılan alanların aynılarından bir değer üretir. `toString` ise nesnenin metin hâlini verir; `println(k1)` bu metodu kendiliğinden çağırır ve `Ali (20)` yazar. `toString` yazılmasaydı `Kisi@1b6d3586` gibi sınıf adı ve sayıdan oluşan bir metin görülürdü.

`String` ve sarmalayıcı sınıflar da nesnedir ve aynı kurala uyar. Programın son iki satırı bunu gösterir. Klavyeden okunan `"Java"` çalışma sırasında oluşturulmuş yeni bir `String` nesnesidir. Koddaki `"Java"` sabiti ise başka bir nesnedir. İçerikleri aynı olsa da `==` `false`, `equals` `true` verir. İki metin sabiti `==` ile karşılaştırıldığında bazen `true` görülmesi, Java'nın aynı sabiti tek bir nesne olarak saklamasındandır. Bu bir kural değil, bir gerçekleme ayrıntısıdır; metinler her zaman `equals` ile karşılaştırılır.

4.1.9 Sarmalayıcı sınıflar

4.1.9.1 İlkel tipin nesne karşılığı

> **Tanım 4.1.16**
>
> **Sarmalayıcı sınıf** (*wrapper class*), bir ilkel tipin değerini bir nesnenin içinde taşıyan sınıftır. Java'daki sekiz ilkel tipin her birinin bir sarmalayıcı sınıfı vardır.

İlkel tip	Sarmalayıcı sınıf	İlkel tip	Sarmalayıcı sınıf
`byte`	`Byte`	`float`	`Float`
`short`	`Short`	`double`	`Double`
`int`	`Integer`	`char`	`Character`
`long`	`Long`	`boolean`	`Boolean`

Tablo 4.1.15: İlkel tipler ve sarmalayıcı sınıfları.

Adların çoğu ilkel tipin büyük harfle başlayan hâlidir; iki istisna `Integer` ve `Character`'dır. Bu sınıflara üç nedenle ihtiyaç duyulur:

- **Bazı yerlerde nesne gerekir.** `ArrayList<Integer>` gibi jenerik koleksiyonlar ilkel tip alamaz; `ArrayList<int>` yazımı derlenmez.
- **"Değer yok" durumu.** İlkel tipe `null` atanamaz; sarmalayıcı tipteki değişkene atanabilir. Bir sınavın notu henüz girilmemişse `Integer not = null;` bunu açıkça gösterir. `int` ile 0 yazılsaydı "sıfır aldı" ile "henüz not yok" ayırt edilemezdi.

- **Yardımcı metotlar ve sabitler.** `Integer.parseInt("12")` metni sayıya çevirir, `Integer.MAX_VALUE` en büyük `int` değerini verir, `Character.isDigit(c)` bir karakterin rakam olup olmadığını söyler.



Şekil 4.1.18: `int` değeri doğrudan yığında tutulur. `Integer` değişkeni ise heap'teki bir nesneyi gösterir; değer o nesnenin içindedir ve değişkene `null` da atanabilir.

Sarmalayıcıların bir bedeli vardır. Şekil 4.1.18'de görüldüğü gibi `int` değeri değişkenin kutusundadır; `Integer` ise ayrı bir nesne, ona giden bir referans ve okunurken bir dolaylı erişim gerektirir. Milyonlarca sayı işleyen bir döngüde bu fark ölçülebilir. Bu yüzden gerekmedikçe ilkel tip tercih edilir.

4.1.9.2 Otomatik kutulama ve kutudan çıkarma

Tanım 4.1.17

Otomatik kutulama (*autoboxing*), ilkel bir değerin gerektiğinde derleyicinin eklediği bir çağrıyla sarmalayıcı nesneye dönüştürülmesidir. **Kutudan çıkarma** (*unboxing*), sarmalayıcı nesnedeki değerin ilkel tipe geri dönüştürülmesidir.

```java
public class Sarmalayici {
  public static void main(String[] args) {
    // int tipinde bir değişken
    int sayi = 42;
    // Integer wrapper sınıfı kullanımı
    Integer wrapperSayi = Integer.valueOf(sayi);
    // Autoboxing (otomatik kutulama)
    Integer wrapperSayi2 = sayi;
    // Unboxing (açık ve otomatik)
    int sayi2 = wrapperSayi2.intValue();
    int sayi3 = wrapperSayi2;
    System.out.println(sayi2 + sayi3);
    System.out.println(wrapperSayi.equals(wrapperSayi2));
    System.out.println(Integer.parseInt("123") + 1);
    System.out.println(Integer.MAX_VALUE);
    System.out.println(Character.isDigit('7'));
    System.out.println(Double.parseDouble("2.5") * 2);
    Integer bos = null;      // int bos = null; derlenmez
    System.out.println(bos);
  }
}
```

```
84
true
124
2147483647
true
5.0
null
```

`Integer wrapperSayi2 = sayi;` satırı bir `int`'i `Integer` değişkene atar. Derleyici bu satırı sessizce `Integer.valueOf(sayi)` çağrısına çevirir; bu otomatik kutulamadır. `int sayi3 = wrapperSayi2;` satırı ise `wrapperSayi2.intValue()` çağrısına çevrilir; bu da kutudan çıkarmadır. Açık biçimler (`Integer.valueOf`, `intValue()`) ile otomatik biçimler aynı kodu üretir.

Çıktının ilk satırı `42 + 42 = 84`'tür. İki sarmalayıcının içeriği aynıdır: `true`. `"123"` metni sayıya çevrilip 1 eklenir: `124`. Dördüncü satır en büyük `int` değeridir: `2147483647`, yani $2^{31} - 1$. `'7'` bir rakamdır: `true`. `"2.5"` `double`'a çevrilip 2 ile çarpılır: `5.0`. Son satırda `null` değerli bir `Integer` yazdırılır. `println` `null` referansı `null` sözcüğüyle yazar; burada kutudan çıkarma olmadığı için hata da olmaz. Aynı değeri bir `int`'e atamak ise derlenmez:

```java
public class IntNull {
  public static void main(String[] args) {
    int bos = null;
  }
}
```

```
IntNull.java:3: error: incompatible types: <null> cannot be converted to int
    int bos = null;
              ^
1 error
```

4.1.9.3 Integer önbelleği ve null tuzağı

Otomatik dönüşümler kodu kısaltır, ama iki tuzak gizler. İkisi de sarmalayıcıların birer nesne olduğunun unutulmasından doğar.

```
import java.util.ArrayList;
public class Tuzaklar {
  public static void main(String[] args) {
    ArrayList<Integer> liste = new ArrayList<>();
    liste.add(5);                // autoboxing: 5 -> Integer
    int ilk = liste.get(0);      // unboxing: Integer -> int
    System.out.println(ilk + 1);
    Integer a = 127, b = 127;
    Integer c = 128, d = 128;
    System.out.println(a == b);
    System.out.println(c == d);
    System.out.println(c.equals(d));
    Integer bos = null;
    int x = bos;                 // NullPointerException
  }
}
```

```
6
true
false
true
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.lang.Integer.intValue()" because "<local7>" is null
        at Tuzaklar.main(Tuzaklar.java:14)
```

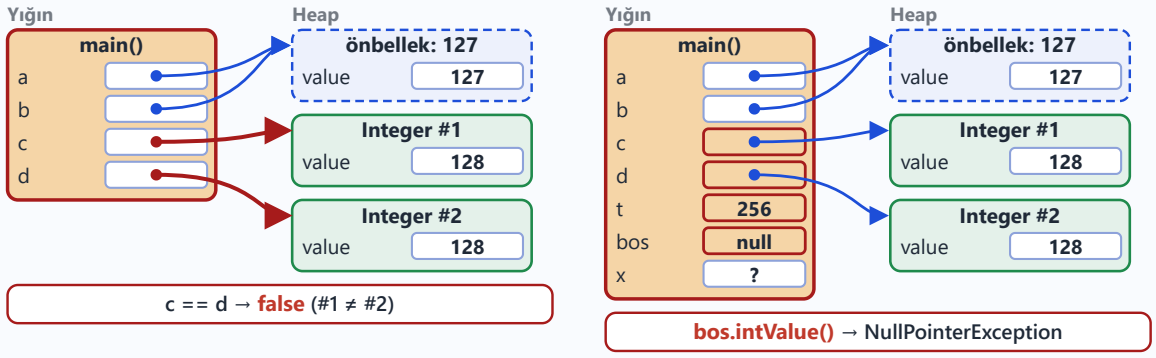
İlk üç satır sorunsuzdur: `liste.add(5)` 5'i kutular, `liste.get(0)` nesneyi verir ve `int ilk`'e atanırken kutudan çıkarılır; `ilk + 1` 6'dır. Sonraki iki karşılaştırma ise şaşırtıcıdır: `a == b` `true`, `c == d` `false` verir. Değerler eşit olduğu hâlde ikinci karşılaştırma neden yanlış çıkar? Aşağıdaki izleme adım adım gösterir.

```
Integer a = 127, b = 127;
Integer c = 128, d = 128;
System.out.println(a == b);
System.out.println(c == d);
System.out.println(c.equals(d));
int t = c + d;
System.out.println(t);
Integer bos = null;
int x = bos;
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	1: `Integer a = 127, b = 127;`	—	a=→önbellek(127), b=→önbellek(127)	—
2	2: `Integer c = 128, d = 128;`	—	a=→önbellek, b=→önbellek, c=→#1, d=→#2	—
3	3: `System.out.println(a == b);`	—	a=→önbellek, b=→önbellek	`true`
4	4: `System.out.println(c == d);`	—	c=→#1, d=→#2	`false`
5	5: `System.out.println(c.equals(d));`	—	c=→#1, d=→#2	`true`
6	6: `int t = c + d;`	—	c=→#1, d=→#2, t=256	—
7	7: `System.out.println(t);`	—	t=256	`256`
8	8: `Integer bos = null;`	—	t=256, bos=null	—
9	9: `int x = bos;`	—	bos=null, x=?	`Exception in thread "main" ⏎ java.lang.NullPointerException`

Tablo 4.1.16: `Integer` önbelleğinin ve `null` kutudan çıkarmanın izlenmesi.

Birinci adımda `Integer a = 127` otomatik kutulamayla `Integer.valueOf(127)` çağrısına dönüşür. `valueOf` metodu −128 ile 127 arasındaki değerler için her seferinde yeni nesne oluşturmaz; bu değerlerin nesneleri bir önbellekte hazırdır. `a` ve `b` önbellekteki aynı nesneyi alır. İkinci adımda 128 bu aralığın dışındadır; `valueOf` her çağrıda yeni bir nesne oluşturur ve `c` ile `d` iki ayrı nesneyi gösterir.



Şekil 4.1.19: Solda `c == d` : değerler aynı (128), ama iki ok iki ayrı nesneye gider; `a` ve `b` ise önbellekteki tek nesneyi paylaşır. Sağda `bos` hiçbir nesneyi göstermez; `int x = bos` kutudan çıkarmaya çalışınca `NullPointerException` fırlatılır.

Üçüncü ve dördüncü adımlarda `==` referansları karşılaştırır. `a` ile `b` aynı nesneyi gösterdiği için sonuç `true`, `c` ile `d` farklı nesneleri gösterdiği için `false`'tur. Değerler aynıdır, nesneler farklıdır. Beşinci adımda `c.equals(d)` nesnelerin içindeki `int` değerleri karşılaştırır ve `true` verir. Tuzak şuradadır: küçük sayılarla yapılan bir deneme `==`'nin doğru çalıştığı izlenimini verir; kod büyük sayılarla karşılaşınca sessizce yanlış sonuç üretir. Sarmalayıcı nesneler her zaman `equals` ile karşılaştırılır.

Altıncı adımda `int t = c + d;` çalışır. `+` işleci nesnelerle çalışmaz; iki nesne de kutudan çıkarılır (`intValue()`) ve 128 + 128 = 256 bulunur. `t` ilkel bir `int`'tir ve değeri yığındaki kutusunda durur. Aynı kural karşılaştırmada da geçerlidir: bir `Integer` bir `int` ile `==` kullanılarak karşılaştırılırsa `Integer` kutudan çıkarılır ve sayılar karşılaştırılır. Yedinci adımda 256 yazılır.

Sekizinci adımda `Integer bos = null;` çalışır. Sarmalayıcı değişkene `null` atanabilir. Dokuzuncu adımda `int x = bos;` gizlice `bos.intValue()` çağrısına çevrilir. Gösterilen nesne yoktur ve `NullPointerException` fırlatılır. İleti bu gizli çağrıyı açıkça adlandırır: `Cannot invoke "java.lang.Integer.intValue()"`. Kodda hiçbir metot çağrısı görünmeyen bir satırın bu hatayı vermesi, otomatik kutudan çıkarmanın en tehlikeli yanıdır.

Sık yapılan hata

Sarmalayıcı nesneleri `==` ile karşılaştırmak. `Integer` değerleri −128…127 aralığında `==` ile doğru sonuç verir gibi görünür; aralığın dışında aynı değerler `false` verir. Önbelleğin üst sınırı JVM ayarıyla büyütülebilir, bu yüzden sınıra güvenmek de doğru değildir. İçerik her zaman `equals` ile karşılaştırılır.

4.1.10 Kayıt ve enum

Bazı sınıflar yalnızca birkaç veriyi bir arada taşımak için yazılır: bir noktanın iki koordinatı, bir öğrencinin adı ve numarası. Bazı tipler ise önceden bilinen, sınırlı bir değer kümesinden birini alır: haftanın günleri, mevsimler. Java her iki durum için kısa ve güvenli birer yazım sunar.

4.1.10.1 Kayıt (record)

Tanım 4.1.18

Kayıt (*record*, Java 16 ve sonrası), yalnızca veri taşıyan sınıfları kısa yazmak için tasarlanmış bir sınıf türüdür. Kaydın bütün alanları `private final`'dır ve değiştirici metodu yoktur; bu yüzden nesneleri **değiştirilemez** (*immutable*).

```
record Nokta(double x, double y) { }
```

Bu tek satır `x` ve `y` adlı iki `double` bileşeni olan bir kayıt tanımlar. Derleyici bu tanımdan iki `private final` alan ve iki parametreli bir yapıcı üretir. Ayrıca `x()` ve `y()` erişimcilerini, içerikle karşılaştıran bir `equals`'ı, ona uygun bir `hashCode`'u ve okunaklı bir `toString`'i ekler. Erişimcilerin adları `get` ile başlamaz; bileşenin adını taşır. Aynı sınıfı elle yazmak yaklaşık 30 satır tutardı.

```
public class KayitTesti {
  public static void main(String[] args) {
    Nokta nokta1 = new Nokta(1.0, 2.5);
    Nokta nokta2 = new Nokta(1.0, 2.5);
    boolean esitMi = nokta1.equals(nokta2);
    System.out.println(esitMi);
    System.out.println(nokta1 == nokta2);
    System.out.println(nokta1);
    System.out.println(nokta1.x() + nokta1.y());
  }
}
```

```
true
false
Nokta[x=1.0, y=2.5]
3.5
```

İki nokta aynı koordinatlarla oluşturulmuştur. `equals` içeriği karşılaştırdığı için `true`, `==` referansları karşılaştırdığı için `false` verir. `println(nokta1)` kaydın ürettiği `toString`'i çağırır: `Nokta[x=1.0, y=2.5]`. Son satır erişimcilerle okunan iki değeri toplar: 3.5.

Kaydın alanı değiştirilmeye çalışılırsa program derlenmez:

```
record Nokta(double x, double y) { }
public class KayitDegistir {
  public static void main(String[] args) {
    Nokta n = new Nokta(1.0, 2.5);
    n.x = 3.0;
  }
}
```

```
KayitDegistir.java:5: error: x has private access in Nokta
    n.x = 3.0;
     ^
1 error
```

Bir kayda yeni bir değer gerekiyorsa yeni bir kayıt oluşturulur. Kayda kendi metotları da eklenebilir. Yapıcıya denetim eklemek için bileşen listesi tekrar yazılmadan kısa bir yapıcı tanımlanır:

```
record Nokta(double x, double y) {
  Nokta {
    if (Double.isNaN(x) || Double.isNaN(y))
      throw new IllegalArgumentException("NaN olamaz");
  }
  double uzaklik() { return Math.sqrt(x * x + y * y); }
  Nokta kaydir(double dx) { return new Nokta(x + dx, y); }
}
public class KayitEk {
  public static void main(String[] args) {
    Nokta n = new Nokta(3.0, 4.0);
    Nokta m = n.kaydir(1.0);
    System.out.println(n + " " + n.uzaklik());
    System.out.println(m);
    System.out.println(n.hashCode() == new Nokta(3.0, 4.0).hashCode());
  }
}
```

```
Nokta[x=3.0, y=4.0] 5.0
Nokta[x=4.0, y=4.0]
true
```

`Nokta { ... }` bloğu kısa yapıcıdır (*compact constructor*). Parametreleri derleyici ekler ve blok yalnızca denetimi yapar; alanlara atama blok bittikten sonra kendiliğinden yapılır. `uzaklik()` noktanın orijine uzaklığını hesaplar: $\sqrt{(3^2 + 4^2)} = 5.0$. `kaydir` nesneyi değiştirmez, yeni bir `Nokta` döndürür; `n` eski değerinde kalır. Son satır, eşit iki kaydın `hashCode` değerlerinin de eşit olduğunu doğrular.

4.1.10.2 enum

Tanım 4.1.19

enum (numaralandırma tipi, *enumeration*), sabit ve sınırlı bir değer kümesini temsil eden özel bir sınıf türüdür. Her değer, o tipin program boyunca tek olan bir nesnesidir.

```
enum HaftaGunleri {
  PAZARTESI, SALI, CARSAMBA, PERSEMBE,
  CUMA, CUMARTESI, PAZAR
}
public class GunTesti {
  public static void main(String[] args) {
    HaftaGunleri[] tumGunler = HaftaGunleri.values();
    HaftaGunleri persembe = HaftaGunleri.valueOf("PERSEMBE");
    System.out.println(tumGunler.length + " gün");
    System.out.println(persembe + " " + persembe.ordinal());
    for (HaftaGunleri gun : tumGunler) {
      boolean haftaSonu = gun == HaftaGunleri.CUMARTESI
          || gun == HaftaGunleri.PAZAR;
      if (haftaSonu) {
        System.out.println(gun + ": hafta sonu");
      }
    }
  }
}
```

```
7 gün
PERSEMBE 3
CUMARTESI: hafta sonu
PAZAR: hafta sonu
```

`HaftaGunleri` tipinin yalnızca yedi değeri vardır ve başka bir değer oluşturulamaz. Haftanın günlerini `int` ile 0–6 arası sayılarla tutmak da mümkündü, ama o zaman `gun = 9` gibi anlamsız bir atamayı derleyici yakalayamazdı. `enum` tipli bir değişkene yalnızca o tipin değerleri atanabilir.

Her `enum` tipiyle birlikte üç yararlı metot gelir. `values()` bütün değerleri tanım sırasıyla bir dizi olarak verir; üzerinde döngü kurulabilir. `valueOf("PERSEMBE")` metni aynı adlı değere çevirir. `ordinal()` değerin 0'dan başlayan sıra numarasını verir: `PERSEMBE` dördüncü değer olduğu için 3.

Her değer tek bir nesne olduğu için `enum` değerleri `==` ile güvenle karşılaştırılır; döngüdeki hafta sonu denetimi böyle yazılmıştır.

`enum` bir sınıf olduğu için alanları, yapıcısı ve metotları olabilir. Aşağıdaki örnekte her gezegen kendi çekim katsayısını taşır. Program bir de `switch` ile `enum` kullanımını ve `valueOf`'un adı birebir aradığını gösterir.

```
enum Gezegen {
  MERKUR(0.38), DUNYA(1.0), MARS(0.38);
  private final double cekim;
  Gezegen(double cekim) { this.cekim = cekim; }
  double agirlik(double kutle) { return kutle * cekim; }
}
public class EnumEk {
  public static void main(String[] args) {
    Gezegen g = Gezegen.MARS;
    switch (g) {
      case DUNYA -> System.out.println("ev");
      case MARS -> System.out.println("kızıl");
      default -> System.out.println("başka");
    }
    System.out.println(g.agirlik(70.0));
    System.out.println(Gezegen.valueOf("Mars"));
  }
}
```

```
kızıl
26.6
Exception in thread "main" java.lang.IllegalArgumentException: No enum constant
Gezegen.Mars
        at java.base/java.lang.Enum.valueOf(Enum.java:293)
        at Gezegen.valueOf(EnumEk.java:1)
        at EnumEk.main(EnumEk.java:16)
```

Değerlerin yanındaki parantezler yapıcıya verilen argümanlardır. `enum`'un yapıcısı her zaman `private`'tır; dışarıdan `new Gezegen(...)` yazılamaz. `switch`'te `case` etiketlerine tip adı yazılmaz: `case MARS`. 70 × 0.38 = 26.6 hesaplanır. Son satırda `valueOf("Mars")` çağrılır; değerin adı `MARS` olduğu için eşleşme bulunmaz ve `IllegalArgumentException` fırlatılır. `valueOf` büyük-küçük harf ayrımı yapar.

4.1.11 İç sınıflar

Tanım 4.1.20

İç sınıf (*inner class*), başka bir sınıfın içinde tanımlanan sınıftır. İç sınıf, dış sınıfın alanlarına ve metotlarına, `private` olanlar dahil, doğrudan erişebilir.

İç sınıflar yalnızca bir sınıfın işine yarayan yardımcı sınıfları o sınıfın içinde toplamak için kullanılır. Grafik arayüzlerde bir düğmeye tıklandığında çalışacak kodu taşıyan olay dinleyici (*listener*) sınıfları bunun en yaygın örneğidir. Nerede tanımlandığına göre dört tür iç sınıf vardır.

Tür	Nerede tanımlanır?	Nasıl oluşturulur?
`static` iç sınıf	Sınıf gövdesinde, `static` ile	Dış sınıf nesnesi olmadan: `new Outer.StaticInner()`
Üye iç sınıf	Sınıf gövdesinde, `static` olmadan	Bir dış sınıf nesnesi üzerinden: `outer.new MemberInner()`
Yerel iç sınıf	Bir metodun içinde	Yalnızca o metodun içinden
Anonim iç sınıf	Bir ifadenin içinde, adsız	Tanımlandığı yerde tek nesne; bir arayüzü ya da soyut sınıfı uygular

Tablo 4.1.17: İç sınıf türleri.

4.1.11.1 static iç sınıf

```
public class Outer {
  // Dış sınıfın içinde statik bir iç sınıf
  static class StaticInner {
    void display() {
      System.out.println("Statik İç Sınıf");
    }
  }

  public static void main(String[] args) {
    // Dış sınıf örneği oluşturmadan erişim
    Outer.StaticInner staticInner = new Outer.StaticInner();
    staticInner.display();
  }
}
```

```
Statik İç Sınıf
```

`static` iç sınıf, dış sınıfın içine yerleştirilmiş sıradan bir sınıf gibidir. Dış sınıfın bir nesnesi olmadan oluşturulabilir: `new Outer.StaticInner()`. Tam adı `Outer.StaticInner`'dır; dış sınıf burada bir ad alanı görevi görür. Bir dış nesneye bağlı olmadığı için dış sınıfın yalnızca `static` üyelerine doğrudan erişebilir. Dış sınıfın bir örnek alanını okumaya çalışırsa tanıdık ileti alınır:

```
public class IcStaticDisAlan {
  private int sayi = 5;
  static class StaticInner {
    void display() { System.out.println(sayi); }
  }
}
```

```
IcStaticDisAlan.java:4: error: non-static variable sayi cannot be referenced from a static
context
    void display() { System.out.println(sayi); }
                                        ^
1 error
```

4.1.11.2 Üye iç sınıf

```
public class Outer {
  private int sayi = 5;          // dış sınıfın özel alanı

  class MemberInner {            // üye iç sınıf
    void display() {
      System.out.println("Üye İç Sınıf, sayi = " + sayi);
    }
  }

  public static void main(String[] args) {
    // Dış sınıf örneği üzerinden üye iç sınıfa erişim
    Outer outer = new Outer();
    Outer.MemberInner memberInner = outer.new MemberInner();
    memberInner.display();
  }
}
```

```
Üye İç Sınıf, sayi = 5
```

Üye iç sınıf, dış sınıfın bir üyesi olarak `static` olmadan tanımlanır. Her üye iç sınıf nesnesi bir dış nesneye bağlıdır. Bu yüzden oluşturulurken bir dış nesne gerekir ve yazım alışılmadıktır: `outer.new MemberInner()`. Bu bağ sayesinde `display()` dış nesnenin `private` `sayi` alanını okuyabilir. `main` içinde doğrudan `new MemberInner()` yazmak derlenmez:

```
public class UyeIcStatic {
  private int sayi = 5;
  class MemberInner {
    void display() { System.out.println(sayi); }
  }
  public static void main(String[] args) {
    MemberInner m = new MemberInner();
  }
}
```

```
UyeIcStatic.java:7: error: non-static variable this cannot be referenced from a static
context
    MemberInner m = new MemberInner();
                    ^
1 error
```

İleti yine `this`'ten söz eder. `new MemberInner()`, sınıfın içinde `this.new MemberInner()` anlamına gelir: iç nesne `this`'e bağlanacaktır. `main` `static` olduğu için `this` yoktur. Bağlantının nasıl kurulduğunu aşağıdaki izleme gösterir.

```
public class Outer {
    private int sayi = 5;
    class MemberInner {
        void display() {
            System.out.println("sayi = " + sayi);
        }
    }
    public static void main(String[] args) {
        Outer o1 = new Outer();
        Outer o2 = new Outer();
        o2.sayi = 8;
        Outer.MemberInner m = o2.new MemberInner();
        m.display();
    }
}
```

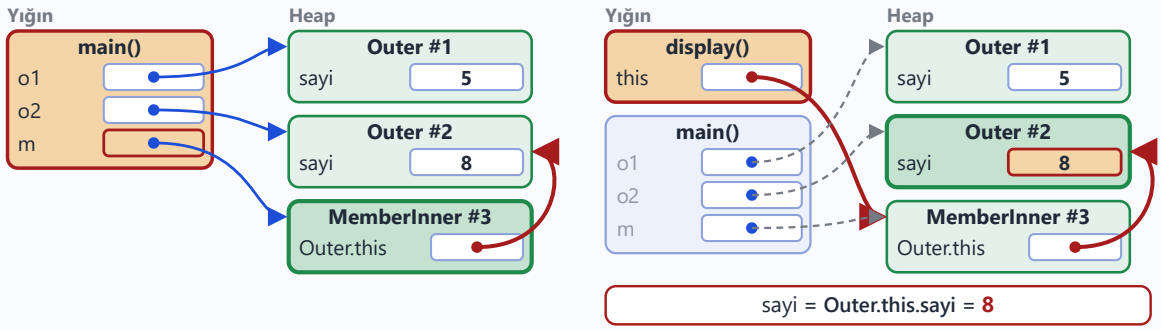
```
sayi = 8
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	9: `Outer o1 = new Outer();`	—	o1=→#1	—
2	10: `Outer o2 = new Outer();`	—	o1=→#1, o2=→#2	—
3	11: `o2.sayi = 8;`	—	o1=→#1, o2=→#2	—

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
4	12: `Outer.MemberInner m = o2.new MemberInner();`	—	o2=→#2, m=→#3	—
5	13: `m.display();` 4: `void display() {`	—	this=→#3	—
6	5: `System.out.println("sayi = " + sayi);`	—	this=→#3, sayi=Outer.this.sayi=8	`sayi = 8`

Tablo 4.1.18: Üye iç sınıf nesnesinin dış nesneye bağlanması.

Birinci ve ikinci adımlarda iki `Outer` nesnesi oluşur; ikisinin de kendi `sayi` alanı vardır ve 5 ile başlar. Üçüncü adımda `o2.sayi = 8;` yalnızca 2 numaralı nesneyi değiştirir. `main` `Outer`'ın içinde olduğu için `private` alana yazabilir. Dördüncü adımda `o2.new MemberInner()` iç nesneyi oluşturur. İç nesne, onu oluşturan dış nesnenin referansını gizli bir alanda saklar. Bu alanın Java'daki adı `Outer.this`'tir.



Şekil 4.1.20: Solda iç nesne (#3) oluşmuş; gizli `Outer.this` alanı 2 numaralı dış nesneyi gösteriyor. Sağda `display()` çalışıyor: `sayi`, `Outer.this.sayi` demektir ve ok 2 numaralı nesneye, 8 değerine gider.

Beşinci adımda `m.display()` çağrılır. Metodun `this`'i iç nesnedir. İç sınıfın kendi `sayi` alanı yoktur. Altıncı adımda derleyici yalın `sayi` adını `Outer.this.sayi` olarak çözer: iç nesnenin gizli alanındaki ok 2 numaralı dış nesneye gider ve `sayi = 8` yazılır. İç nesne `o1.new MemberInner()` ile oluşturulsaydı ok 1 numaralı nesneye gider ve 5 yazılırdı. İç ve dış sınıfta aynı adlı alanlar olduğunda iç sınıfın kendi alanı `this.x`, dış nesnenin alanı `Outer.this.x` yazımıyla ayırt edilir.

4.1.11.3 Yerel ve anonim iç sınıf

```
public class Outer {
  void outerMethod() {
    class LocalInner {         // metodun içinde tanımlanır
      void display() {
        System.out.println("Yerel İç Sınıf");
      }
    }
    // Yerel iç sınıfa yalnızca bu metodun içinden erişilir
    LocalInner localInner = new LocalInner();
    localInner.display();
  }

  public static void main(String[] args) {
    Outer outer = new Outer();
    outer.outerMethod();
  }
}
```

```
Yerel İç Sınıf
```

Yerel iç sınıf bir metodun içinde tanımlanır ve yalnızca o metodun içinden kullanılabilir. Metodun dışında `LocalInner` adı bilinmez:

```
public class YerelDisari {
  void outerMethod() {
    class LocalInner { void display() { } }
    new LocalInner().display();
  }
  void baska() {
    LocalInner li = new LocalInner();
  }
}
```

```
YerelDisari.java:7: error: cannot find symbol
    LocalInner li = new LocalInner();
    ^
  symbol:   class LocalInner
  location: class YerelDisari
YerelDisari.java:7: error: cannot find symbol
    LocalInner li = new LocalInner();
                        ^
  symbol:   class LocalInner
  location: class YerelDisari
2 errors
```

Yerel iç sınıf, bir metodun içinde birkaç kez gereken küçük bir yardımcı tip için kullanılır. Kapsamı bir yerel değişkeninki gibidir: tanımlandığı bloğun sonunda biter.

```
interface MyInterface {
  void myMethod();
}

public class Outer {
  public static void main(String[] args) {
    // Genellikle arayüzleri veya soyut sınıfları uygular
    MyInterface myInterface = new MyInterface() {
      @Override
      public void myMethod() {
        System.out.println("Anonim İç Sınıf");
      }
    };
    myInterface.myMethod();
    MyInterface kisa = () -> System.out.println("Lambda");
    kisa.myMethod();
    System.out.println(myInterface.getClass().getName());
  }
}
```

```
Anonim İç Sınıf
Lambda
Outer$1
```

Anonim iç sınıf (*anonymous class*) adı olmayan bir sınıftır; tanımlandığı yerde tek bir nesnesi oluşturulur. Genellikle bir arayüzü ya da soyut sınıfı uygular. `new MyInterface() { ... }` yazımı "`MyInterface`'i uygulayan adsız bir sınıf tanımla ve ondan bir nesne oluştur" demektir. Süslü parantezlerin içine arayüzün metotlarının gövdeleri yazılır. Sondaki noktalı virgül atama deyimine aittir; unutulursa derleme hatası alınır.

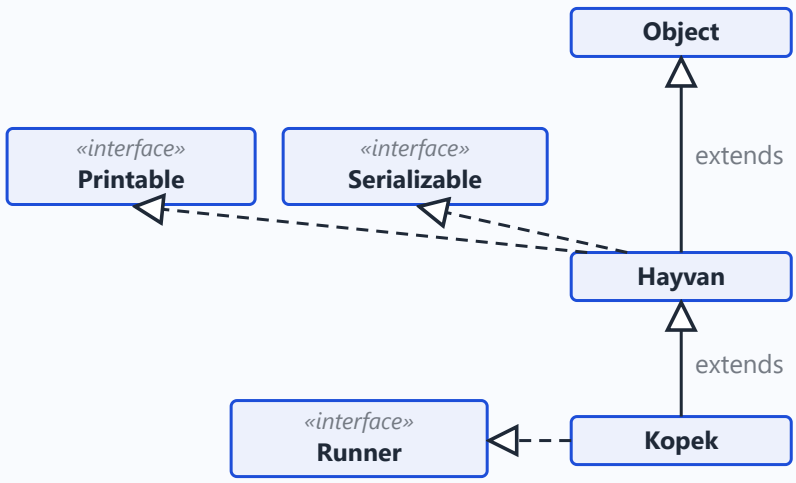
Adsız sınıfın da derleyicinin verdiği bir adı vardır. Programın son satırı bunu yazdırır: `Outer$1`, yani `Outer`'ın içindeki birinci anonim sınıf. Derleme klasöründe `Outer$1.class` adlı ayrı bir dosya da oluşur. Yalnızca tek bir soyut metodu olan arayüzler için Java 8'den beri daha kısa bir yazım vardır: lambda ifadesi. `() -> System.out.println("Lambda")` aynı işi tek satırda görür.

4.1.12 Sınıf türleri ve tasarım

4.1.12.1 Bir sınıf, birçok tip

Bir nesne yalnızca tek bir sınıfa ait olabilir: `new` ile hangi sınıftan oluşturulduysa o sınıfın nesnesidir. Ama birçok tipi olabilir. Aşağıdaki tanımlarda `Kopek`, `Hayvan`'dan türer ve `Runner` arayüzünü uygular; `Hayvan` da `Printable` ve `Serializable` arayüzlerini uygular.

```
interface Printable { ... }
interface Runner { ... }
class Hayvan implements Printable, Serializable { ... }
class Kopek extends Hayvan implements Runner { ... }
var kazanan = new Kopek(...);
```



Şekil 4.1.21: UML tip hiyerarşisi. Düz çizgi ve boş üçgen kalıtımı (`extends`), kesik çizgi bir arayüzün uygulanmasını (`implements`) gösterir. Her sınıf en üstte `Object`'ten türer.

`kazanan` değişkeninin gösterdiği nesne yalnızca `Kopek` sınıfına aittir. Ama bu nesnenin tipleri `Kopek`, `Runner`, `Hayvan`, `Printable`, `Serializable` ve `Object`'tir. `instanceof` işleci altısı için de `true` verir. Tanımları gerçek gövdelerle doldurup sınayan küçük bir program:

```
import java.io.Serializable;
interface Printable { }
interface Runner { }
class Hayvan implements Printable, Serializable { }
class Kopek extends Hayvan implements Runner { }
public class Tipler {
  public static void main(String[] args) {
    var kazanan = new Kopek();
    Object o = kazanan;
    System.out.println((o instanceof Kopek) + " " + (o instanceof Runner) + " " + (o
instanceof Hayvan)
        + " " + (o instanceof Printable) + " " + (o instanceof Serializable) + " " + (o
instanceof Object));
    System.out.println(kazanan.getClass().getName());
  }
}
```

```
true true true true true true
Kopek
```

Bir nesnenin birden çok tip olarak görülebilmesi çok biçimliliğin temelidir. `Hayvan` tipinde bir değişken bir `Kopek` nesnesini gösterebilir; `Runner` bekleyen bir metot bir `Kopek` alabilir. Buradan çıkan sonuç şudur: "sınıf" ile "tip" aynı kavram değildir.

4.1.12.2 Sınıf çeşitleri

Sınıflar amaçlarına göre farklı biçimlerde yazılır. Tablo 4.1.19 sık karşılaşılan çeşitleri listeler. Bir sınıf birden çok çeşide aynı anda girebilir: `Math` hem araç sınıfıdır hem `final`'dır.

Çeşit	Özellik	Örnek
Numaralandırma (`enum`)	Değerleri sabit bir nesne kümesidir	`enum Gun { PAZARTESI, SALI, ... }`
Tek nesnelik (*singleton*)	Yalnızca bir nesnesi oluşturulabilir	`private` yapıcı + `static` tek nesne
Soyut (`abstract`)	Kendi nesnesi oluşturulamaz	`abstract class Sekil`
Mühürlü (`sealed` , Java 17+)	Alt tipleri sabit bir kümedir	`sealed interface Hareket permits Yuru, Kos`
`final`	Alt sınıf türetilemez	`final class String`
Kayıt (`record`)	Nesneleri değiştirilemez	`record Nokta(double x, double y)`
Araç (*utility*)	Yalnızca `static` metotlar içerir	`Math` , `Arrays`
Uygulama	`main` metodunu içerir	Programın başladığı sınıf

Tablo 4.1.19: Sınıf çeşitleri.

```
final class Araclar {             // araç ve final sınıf
  private Araclar() { }          // nesne oluşturulamaz
  static int kare(int x) { return x * x; }
}
class Ayarlar {                   // singleton
  private static final Ayarlar TEK = new Ayarlar();
  private Ayarlar() { }
  static Ayarlar getir() { return TEK; }
}
abstract class Sekil {            // soyut sınıf
  abstract double alan();
}
// mühürlü (sealed, Java 17+)
sealed interface Hareket permits Yuru, Kos { }
final class Yuru implements Hareket { }
final class Kos implements Hareket { }
public class Cesitler {           // uygulama sınıfı
  public static void main(String[] args) {
    System.out.println(Araclar.kare(7));
    System.out.println(Ayarlar.getir() == Ayarlar.getir());
  }
}
```

```
49
true
```

`Araclar` bir araç sınıfıdır. Yapıcısı `private` olduğu için dışarıdan nesnesi oluşturulamaz; zaten nesneye ihtiyaç yoktur, çünkü bütün metotları `static`'tir. `final` olduğu için ondan alt sınıf da türetilemez. İki kısıtlamayı da derleyici uygular:

```
final class Araclar {
  private Araclar() { }
}
class AltAraclar extends Araclar { }
public class FinalTuret { }
```

```
FinalTuret.java:4: error: cannot inherit from final Araclar
class AltAraclar extends Araclar { }
                         ^
FinalTuret.java:4: error: Araclar() has private access in Araclar
class AltAraclar extends Araclar { }
^
2 errors
```

`Ayarlar` tek nesnelik bir sınıftır. Tek nesne, sınıf yüklenirken `private static final` bir alanda oluşturulur. Yapıcı `private` olduğu için başka nesne oluşturulamaz; nesneye yalnızca `getir()` ile ulaşılır. `getir()` her çağrıda aynı referansı döndürdüğü için `Ayarlar.getir() == Ayarlar.getir()` `true` verir. Başka bir sınıfta `new Ayarlar()` yazılırsa:

```
TekilNew.java:8: error: Ayarlar() has private access in Ayarlar
    Ayarlar a = new Ayarlar();
                ^
1 error
```

`Sekil` soyut bir sınıftır: `alan()` metodunun gövdesi yoktur, çünkü bir "şeklin" alanı hesaplanamaz; ancak bir dairenin ya da karenin alanı hesaplanabilir. Soyut sınıftan nesne oluşturulamaz:

```
SoyutNew.java:6: error: Sekil is abstract; cannot be instantiated
    Sekil s = new Sekil();
              ^
1 error
```

`Hareket` mühürlü bir arayüzdür. `permits` listesi onu uygulayabilecek tipleri sayar: yalnızca `Yuru` ve `Kos`. Listede olmayan bir sınıf `Hareket`'i uygulamaya çalışırsa derlenmez:

```
MuhurIhlal.java:4: error: class is not allowed to extend sealed class: Hareket (as it is
not listed in its 'permits' clause)
final class Ucus implements Hareket { }
      ^
1 error
```

Mühürlü tipler, alt tiplerin kümesinin tasarım gereği kapalı olduğu durumlar için kullanılır. Derleyici bütün olası alt tipleri bildiği için, örneğin bir `switch`'in her durumu karşılayıp karşılamadığını denetleyebilir.

4.1.12.3 Tasarım örneği: RasyonelSayi

Şimdiye kadarki parçalar bir araya getirilerek baştan bir sınıf tasarlanabilir. `RasyonelSayi` sınıfı `pay/payda` biçimindeki kesirleri temsil eder. Bir sınıfı tasarlarken üç soru sorulur:

- **Tanımlama**: Sınıfı kullanan kod neyi görecek? Burada `public` bir yapıcı, iki kesri toplayan `topla` ve kesri metin olarak veren `toString`.
- **Temsil**: Veri nasıl saklanacak? İki `private int` alan: `pay` ve `payda`. Yapıcı kesri sadeleştirerek saklar; böylece her nesne her zaman en sade biçimdedir.
- **Gerçekleme**: Metotların gövdeleri ve dışarıya açılmayan yardımcılar. Sadeleştirme için gereken en büyük ortak bölen, gizli bir `ebob` metodunda Öklid algoritmasıyla bulunur.

```
public class RasyonelSayi {
  // Temsil: gizli alanlar
  private int pay, payda;
  // Tanımlama: public yapıcı ve metotlar
  public RasyonelSayi(int pay, int payda) {
    int e = ebob(Math.abs(pay), Math.abs(payda));
    this.pay = pay / e;
    this.payda = payda / e;
  }
  public RasyonelSayi topla(RasyonelSayi d) {
    int yeniPay = pay * d.payda + d.pay * payda;
    return new RasyonelSayi(yeniPay, payda * d.payda);
  }
  @Override
  public String toString() { return pay + "/" + payda; }
  // Gerçekleme ayrıntısı: gizli yardımcı metot
  private static int ebob(int a, int b) {
    while (b != 0) { int k = a % b; a = b; b = k; }
    return a;
  }
}
```

Sınama programı `a = new RasyonelSayi(8, 10)` ve `b = new RasyonelSayi(2, 3)` oluşturur ve önce `println(a)`, sonra `println(a.topla(b))` çağırır. Programın bütün çıktısı şudur:

```
4/5
22/15
11/12 3/4
-2/3 1/-3 0/1
1/1
```

Çıktının ilk iki satırı bu iki çağrıdır. 8/10 sadeleşip 4/5 olarak saklanmıştır. 4/5 + 2/3 = (4 × 3 + 2 × 5) / (5 × 3) = 22/15'tir; 22 ile 15'in ortak böleni olmadığı için kesir sade kalır. Üçüncü satırı `a = (6, 8)` ve `b = (1, 6)` ile yapılan aynı çağrı üretir. Bu çağrı aşağıda adım adım izlenir.

```
public RasyonelSayi(int pay, int payda) {
    int e = ebob(Math.abs(pay), Math.abs(payda));
    this.pay = pay / e;
    this.payda = payda / e;
}
private static int ebob(int a, int b) {
    while (b != 0) { int k = a % b; a = b; b = k; }
    return a;
}
public RasyonelSayi topla(RasyonelSayi d) {
    int yeniPay = pay * d.payda + d.pay * payda;
    return new RasyonelSayi(yeniPay, payda * d.payda);
}
RasyonelSayi a = new RasyonelSayi(6, 8);   // main
RasyonelSayi b = new RasyonelSayi(1, 6);
System.out.println(a.topla(b) + " " + a);
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı

1	14: `RasyonelSayi a = new RasyonelSayi(6, 8); // main`
	1: `public RasyonelSayi(int pay, int payda) {`
		—	pay=6, payda=8	—

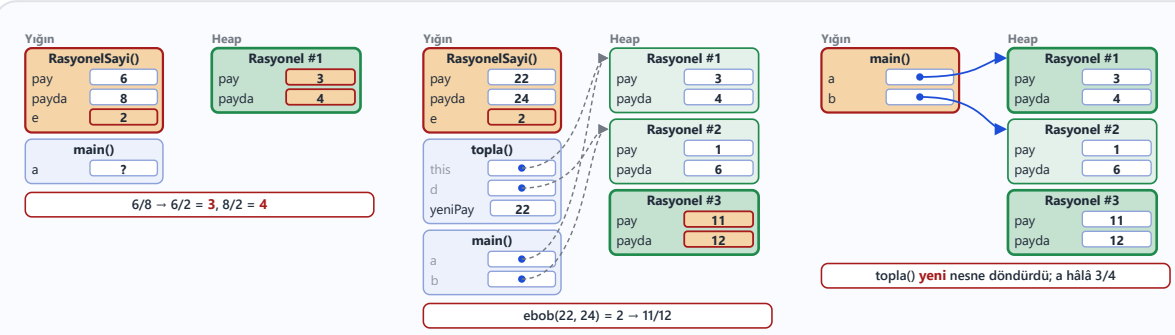
2	2: `int e = ebob(Math.abs(pay), Math.abs(payda));` 6: `private static int ebob(int a, int b) {`	—	a=6, b=8	—
3	7: `while (b != 0) { int k = a % b; a = b; b = k; }`	—	a=8, b=6, k=6	—
4	7: `while (b != 0) { int k = a % b; a = b; b = k; }`	—	a=6, b=2, k=2	—
5	7: `while (b != 0) { int k = a % b; a = b; b = k; }`	—	a=2, b=0, k=0	—
6	8: `return a;` 3: `this.pay = pay / e;` 4: `this.payda = payda / e;`	—	e=2, this.pay=3, this.payda=4	—
7	15: `RasyonelSayi b = new RasyonelSayi(1, 6);`	—	a=→#1 (3/4), b=→#2 (1/6)	—
8	16: `System.out.println(a.topla(b) + " " + a);` 10: `public RasyonelSayi topla(RasyonelSayi d) {` 11: `int yeniPay = pay * d.payda + d.pay * payda;`	—	this=→#1, d=→#2, yeniPay=22	—
9	12: `return new RasyonelSayi(yeniPay, payda * d.payda);` 1: `public RasyonelSayi(int pay, int payda) {` 2: `int e = ebob(Math.abs(pay), Math.abs(payda));`	—	pay=22, payda=24, e=2	—

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
10	16: `System.out.println(a.topla(b) + " " + a);`	—	a=→#1 (3/4), sonuç=→#3 (11/12)	`11/12 3/4`

Tablo 4.1.20: `RasyonelSayi` yapıcısının, `ebob`'un ve `topla`'nın izlenmesi. Satır numaraları yukarıdaki kısaltılmış kodun satırlarıdır.

Birinci adımda `new RasyonelSayi(6, 8)` nesneyi ayırır; alanlar önce 0'dır, yapıcının parametreleri 6 ve 8'dir. İkinci adımda gizli yardımcı `ebob(6, 8)` çağrılır. Bu metot `private` olduğu için sınıfı kullanan kod onu göremez ve çağıramaz. `static`'tir, çünkü yalnızca parametreleriyle çalışır; hiçbir nesnenin alanına dokunmaz.

Üçüncü, dördüncü ve beşinci adımlar Öklid algoritmasının üç turudur. Her turda `k = a % b` hesaplanır, sonra `a ← b` ve `b ← k` olur. İlk turda 6 % 8 = 6'dır; sayılar yalnızca yer değiştirir: a = 8, b = 6. İkinci turda 8 % 6 = 2: a = 6, b = 2. Üçüncü turda 6 % 2 = 0: a = 2, b = 0. `b` sıfır olunca döngü biter ve `a`, yani 2 döndürülür. Altıncı adımda yapıcı payı ve paydayı 2'ye bölerek saklar: nesne 3/4 olur.



Şekil 4.1.22: Soldan sağa: ilk nesne sadeleşmiş, 3/4; `topla` içinden yeni bir `RasyonelSayi(22, 24)` kuruluyor; sonuç 11/12 ayrı bir nesne, `a` hâlâ 3/4.

Yedinci adım aynı işleri `(1, 6)` için yapar. `ebob(1, 6)` 1 döndürür; kesir zaten sadedir ve 1/6 olarak saklanır. Sekizinci adımda `a.topla(b)` çağrılır. Metodun `this`'i 1 numaralı nesne, `d` parametresi 2 numaralı nesnedir. `yeniPay = pay * d.payda + d.pay * payda` = 3 × 6 + 1 × 4 = 22 bulunur. `d.payda` başka bir nesnenin `private` alanıdır, ama kod aynı sınıfın içinde olduğu için okunabilir.

Dokuzuncu adımda `new RasyonelSayi(22, 24)` çağrılır. Yapıcı yine `ebob`'u çalıştırır: `b` sırasıyla 24, 22, 2, 0 değerlerini alır ve sonuç 2'dir. Yeni nesne 11/12 olarak saklanır. Onuncu adımda `println` önce dönen sonucu, sonra `a`'yı yazar: `11/12 3/4`. `topla` mevcut nesneyi değiştirmemiş, yeni bir nesne döndürmüştür. `RasyonelSayi`'de değiştirici metot yoktur ve hiçbir metot alanları değiştirmez; nesneleri bu anlamda değiştirilemezdir.

Maliyet açısından `ebob`'un döngüsü küçük sayının basamak sayısıyla orantılı sayıda tur atar; Öklid algoritmasının tur sayısı O(log min(a, b))'dir. Yapıcı ve `topla` bu yüzden çok hızlıdır. `topla`'nın her çağrısı yeni bir nesne oluşturur; değiştirilemez tasarımın bedeli budur. Karşılığında bir kesri tutan değişken başka bir yerde paylaşılsa bile değeri hiçbir zaman beklenmedik biçimde değişmez.

Sınama programının son iki satırı sınır durumları dener. `new RasyonelSayi(-4, 6)` doğru biçimde `-2/3` olur. `new RasyonelSayi(0, 5)` için `ebob(0, 5) = 5` bulunur ve kesir `0/1` olarak saklanır. Son satırda 1/2 + 1/2 = 4/4 hesaplanır ve sadeleşip `1/1` olur. Bu denemeler tasarımın iki sınırını da ortaya çıkarır. `new RasyonelSayi(3, -9)` ise `1/-3` olarak saklanır: yapıcı işareti paydadan paya taşımaz. Aynı değer iki farklı biçimde saklanabildiği için bu sınıfa yazılacak bir `equals` eşit kesirleri farklı sayabilir. Düzeltme, yapıcıda `payda` negatifse ikisinin de işaretini çevirmektir. İkinci sınır sıfır paydadır. `new RasyonelSayi(0, 0)` çağrısında `ebob(0, 0)` 0 döndürür ve bölme şu hatayı verir:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at RasyonelSayi.<init>(RasyonelSifir.java:10)
        at RasyonelSifir.main(RasyonelSifir.java:3)
```

Paydanın sıfır olmasına izin vermeyen bir yapıcı, hatayı kesrin oluşturulduğu anda ve anlamlı bir iletiyle bildirmelidir. Örneğin yapıcının ilk satırı `if (payda == 0) throw new IllegalArgumentException("payda 0 olamaz");` olabilir. Temsilin gizli olması bu düzeltmeleri kolaylaştırır: değişiklik yalnızca yapıcıyı etkiler, sınıfı kullanan kod aynen kalır.

4.1.13 Örnek problemler ve alıştırmalar

Örnek 4.1.2

Aşağıdaki program ne yazdırır?

```
class Sayac {
  static int toplam = 0;
  int adet = 0;
  void arttir() { toplam++; adet++; }
}
public class Problem1 {
  public static void main(String[] args) {
    Sayac a = new Sayac(), b = new Sayac();
    a.arttir(); a.arttir(); b.arttir();
    System.out.print(a.adet + " " + b.adet);
    System.out.println(" " + Sayac.toplam);
  }
}
```

```
2 1 3
```

`adet` örneğe ait bir alandır; her nesnede ayrı bir kopyası vardır. `a.arttir()` iki kez, `b.arttir()` bir kez çağrıldığı için `a.adet` 2, `b.adet` 1 olur. `toplam` ise `static` 'tir; üç çağrı da aynı kutuyu artırır ve değeri 3 olur. Çıktı `2 1 3` 'tür.

Örnek 4.1.3

Aşağıdaki program ne yazdırır?

```
class Kutu { int deger; }
public class Problem2 {
  public static void main(String[] args) {
    Kutu k1 = new Kutu();
    k1.deger = 5;
    Kutu k2 = k1;
    k2.deger = 9;
    Kutu k3 = new Kutu();
    k3.deger = 9;
    System.out.println(k1.deger);
    System.out.println(k1 == k2);
    System.out.println(k2 == k3);
  }
}
```

```
9
true
false
```

`Kutu k2 = k1;` yeni nesne oluşturmaz; `k2` ile `k1` aynı nesneyi gösterir. `k2.deger = 9` bu ortak nesneye yazar, bu yüzden `k1.deger` de 9'dur. `k1 == k2` `true` verir. `k3` ise `new` ile oluşturulmuş ayrı bir nesnedir; değeri de 9 olsa `k2 == k3` `false` verir.

Örnek 4.1.4

Aşağıdaki iki program ne yazdırır?

```
record Para(int lira) { }
public class Problem5 {
  public static void main(String[] args) {
    Para p1 = new Para(10);
    Para p2 = new Para(10);
    Integer i1 = 1000, i2 = 1000;
    System.out.println(p1.equals(p2));
    System.out.println(p1 == p2);
    System.out.println(i1.equals(i2));
    System.out.println(p1);
  }
}
```

```
true
false
true
Para[lira=10]
```

```
class Nokta {
  int x, y;
  Nokta(int x, int y) {
    this.x = x;
    this.y = y;
  }
}
public class Problem6 {
  public static void main(String[] args) {
    Nokta a = new Nokta(1, 2);
    Nokta b = new Nokta(1, 2);
    System.out.println(a.equals(b));
    System.out.println(a.x == b.x);
  }
}
```

```
false
true
```

Birinci programda `Para` bir kayıttır. Kaydın `equals`'ı içeriği karşılaştırır: `true`. `==` ise iki ayrı nesneyi karşılaştırır: `false`. `i1` ve `i2` 1000 değerli iki ayrı `Integer` nesnesidir; önbellek aralığının dışında oldukları için `==` `false` verirdi, ama `equals` her zaman güvenlidir: `true`. Son satır kaydın ürettiği `toString`'dir. İkinci programda `Nokta` sıradan bir sınıftır ve `equals` yazmamıştır. `Object`'in sürümü referansları karşılaştırır: `false`. `a.x == b.x` ise iki `int` değeri karşılaştırır: `true`.

Aşağıdaki beş alıştırma bu notun konularını bir araya getirir. Ardından bir çözüm programı ve çıktısı gelir.

1. `Kitap` sınıfı: `private` `baslik`, `yazar`, `sayfaSayisi`; üç parametreli yapıcı, get metotları ve `toString()`.
2. `Kitap`'a bir `static` `toplamKitapSayisi` sayacı ve yalnızca pozitif değeri kabul eden `setSayfaSayisi`.
3. `Daire` sınıfı: `yaricap`, `alan()`, `cevre()`; `new Daire(3)` için iki ondalık basamakla yazdırma.
4. `Mevsim` enum'unun değerlerini sıra numaralarıyla yazdırma.
5. `OgrenciKaydi(String ad, int numara)` kaydı ve bir nesnesinin yazdırılması.

```
class Kitap {
  private String baslik, yazar;
  private int sayfaSayisi;
  private static int toplamKitapSayisi = 0;
  Kitap(String baslik, String yazar, int sayfaSayisi) {
    this.baslik = baslik;
    this.yazar = yazar;
    setSayfaSayisi(sayfaSayisi);
    toplamKitapSayisi++;
  }
  String getBaslik() { return baslik; }
  String getYazar() { return yazar; }
  int getSayfaSayisi() { return sayfaSayisi; }
  void setSayfaSayisi(int sayfaSayisi) {
    if (sayfaSayisi > 0) this.sayfaSayisi = sayfaSayisi;
  }
  static int getToplamKitapSayisi() { return toplamKitapSayisi; }
  @Override
  public String toString() { return baslik + " (" + yazar + ", " + sayfaSayisi + " s.)"; }
}
class Daire {
  private double yaricap;
  Daire(double yaricap) { this.yaricap = yaricap; }
  double alan() { return Math.PI * yaricap * yaricap; }
  double cevre() { return 2 * Math.PI * yaricap; }
}
enum Mevsim { ILKBAHAR, YAZ, SONBAHAR, KIS }
record OgrenciKaydi(String ad, int numara) { }
public class Alistirmalar {
  public static void main(String[] args) {
    Kitap k = new Kitap("Nutuk", "Atatürk", 600);
    System.out.println(k);
    Kitap k2 = new Kitap("Deneme", "Yazar", -5);
    System.out.println(k2.getSayfaSayisi() + " " + Kitap.getToplamKitapSayisi());
    Daire d = new Daire(3);
    System.out.printf("%.2f %.2f%n", d.alan(), d.cevre());
    for (Mevsim m : Mevsim.values()) System.out.print(m.ordinal() + ":" + m + " ");
    System.out.println();
    System.out.println(new OgrenciKaydi("Ali", 7));
  }
}
```

```
Nutuk (Atatürk, 600 s.)
0 2
28,27 18,85
0:ILKBAHAR 1:YAZ 2:SONBAHAR 3:KIS
OgrenciKaydi[ad=Ali, numara=7]
```

`Kitap`'ın yapıcısı sayfa sayısını doğrudan alana yazmaz; `setSayfaSayisi`'nı çağırır. Böylece denetim tek bir yerde kalır ve yapıcı da ondan yararlanır. −5 sayfalı ikinci kitapta değiştirici değeri reddeder; alan varsayılan değeri 0'da kalır. Sayaç ise iki kitabı da sayar: `0 2`. `Daire` için `printf("%.2f %.2f%n", ...)` kullanılır; Türkçe yerel ayarda ondalık ayırıcı virgüldür: π × 9 ≈ 28,27 ve 2 × π × 3 ≈ 18,85. `Mevsim`'in değerlerinde `values()` ile gezinilir ve `ordinal()` 0'dan başlar.

4.1.14 Sık yapılan hatalar

Nesne ve sınıflarla ilgili hataların çoğunu derleyici yakalar ve iletisi hatanın yerini doğrudan gösterir. Bir kısmı ise derlenen ama yanlış çalışan programlara yol açar; bunlar daha tehlikelidir.

Sık yapılan hata

Örneğe ait metodu `main`'den nesnesiz çağırmak. İleti: `non-static method selamVer() cannot be referenced from a static context`. Çağrı bir nesne üzerinden yapılır (`new Selam().selamVer()`); metot gerçekten hiçbir nesnenin verisini kullanmıyorsa `static` yapılır.

Sık yapılan hata

Yapıcıya `void` yazmak. `public void Hata1(String m)` yapıcı değil, sıradan bir metottur. `new Hata1("selam")` şu iletiyle reddedilir: `constructor Hata1 in class Hata1 cannot be applied to given types`. Yapıcının dönüş tipi olmaz.

Sık yapılan hata

Yalnızca parametreli yapıcı varken parametresiz nesne oluşturmak. Bir yapıcı yazılınca varsayılan yapıcı eklenmez; `new Urun() cannot be applied to given types` iletisini verir. Gerekiyorsa parametresiz yapıcı da yazılır.

Sık yapılan hata

`new` unutmak. `Urun u = Urun("Kalem");` satırı bir metot çağrısı gibi okunur ve derleyici böyle bir metot arar:

```
NewUnutma.java:7: error: cannot find symbol
    Urun u = Urun("Kalem");
             ^
  symbol:   method Urun(String)
  location: class NewUnutma
1 error
```

Sık yapılan hata

`private` **alana dışarıdan erişmek.** İleti: `hesapBakiyesi has private access in BankaHesabi`. Alana sınıfın get ve set metotlarıyla erişilir.

Sık yapılan hata

`static` **metotta** `this` **kullanmak.** İleti: `non-static variable this cannot be referenced from a static context`. `static` metodun bir nesnesi yoktur.

Sık yapılan hata

Yapıcıda `this` **unutmak (gölgeleme).** `ad = ad;` derlenir, ama parametreyi kendine atar; alan `null` kalır. Doğrusu `this.ad = ad;`. Alan `final` yapılırsa derleyici hatayı yakalar.

Sık yapılan hata

`this(...)` **'yi ilk satırdan sonra yazmak.** İleti: `call to this must be first statement in constructor`. Öteki yapıcıya devretmek yapıcının ilk deyimi olmalıdır.

Sık yapılan hata

Nesneleri `==` **ile karşılaştırmak.** `k1 == k2` içerikleri aynı iki ayrı nesne için `false` verir. İçerik `equals()` ile karşılaştırılır; `String` ve sarmalayıcı sınıflar dahil. Kendi sınıfında `equals` yazılmamışsa `Object` 'in sürümü de `==` gibi davranır.

Sık yapılan hata

`null` **referansı kullanmak.** Nesne oluşturulmadan ya da `null` atandıktan sonra metot çağrılamaz; `NullPointerException` fırlatılır. Değer verilmemiş referans alanları da `null` 'dır.

Kullanmadan önce `null` denetimi yapılır.

Sık yapılan hata

Kayıt alanını değiştirmeye çalışmak. İleti: `x has private access in Nokta`. Kayıt değiştirilemezdir; yeni değer için yeni bir nesne oluşturulur.

Sık yapılan hata

Üye iç sınıfı `static main`'de doğrudan oluşturmak. İleti: `non-static variable this cannot be referenced from a static context`. Doğrusu `outer.new MemberInner()`; ya da iç sınıf `static` yapılır.

4.1.15 Özet

- **Nesne yönelimli programlama** programı birbiriyle etkileşen nesnelerden kurar. Her nesnenin **durumu** (alanlar) ve **davranışı** (metotlar) vardır. Üç temel özellik sarmalama, kalıtım ve çok biçimliliktir; çoğu zaman soyutlama da eklenir.
- **Sınıf** nesnenin taslağıdır; **nesne** `new` ile üretilen örneğidir. Aynı sınıfın nesneleri aynı alan adlarını taşır, ama her birinin değerleri ayrıdır. Metot kodu sınıfta tektir.
- **Erişim belirleyiciler**: `public`, `protected`, paket erişimi, `private`. Alanlar `private`, dışarıya sunulan metotlar `public` olur; değer erişimci ile okunur, denetim yapan değiştirici ile yazılır (**veri gizleme**).
- **Yapıcı** sınıfla aynı adı taşır ve dönüş tipi yoktur. `new` heap'te yer ayırır, alanlara varsayılan değer verir, yapıcıyı çağırır ve referansı döndürür. Hiç yapıcı yazılmazsa derleyici parametresiz **varsayılan yapıcıyı** ekler; bir yapıcı yazılınca eklemez.
- Alanların varsayılan değerleri: sayılar 0, `boolean` `false`, `char` `'\u0000'`, referanslar `null`. Yerel değişkenler varsayılan değer almaz.
- **Örneğe ait alan** her nesnede ayrı, `static` **alan** sınıfta tek ve ortaktır. `static` metodun `this`'i yoktur; örnek üyelerine ancak bir nesne üzerinden ulaşır. `final` alana bir kez değer verilir; sınıf sabitleri `static final` ve büyük harfle yazılır.
- İlklendirme sırası: `static` alanlar ve `static { }` blokları sınıf ilk kullanılırken bir kez; sonra her `new`'de varsayılan değerler, alan ilk değerleri ve `{ }` blokları (yazılış sırasıyla), en son yapıcının gövdesi.
- Değişken nesneyi değil **referansı** tutar. Atama referansı kopyalar ve **takma ad** oluşturur. Metotlara nesne aktarmak da referansın kopyasını verir. `null` referans üzerinden erişim `NullPointerException` fırlatır; referansı kalmayan nesneyi **çöp toplayıcı** siler.

- `this` metodun çağrıldığı nesnedir; gölgelenen alana `this.ad` ile ulaşılır ve nesne `this` ile başka metotlara verilebilir. `this(...)` aynı sınıfın başka yapıcısını çağırır, ilk satırda olmalıdır ve tek bir nesne kurar.
- `==` referans eşitliğini, `equals` içerik eşitliğini sınar. `equals` yeniden yazılınca `hashCode` da yazılır. `String` ve sarmalayıcılar `equals` ile karşılaştırılır.
- **Sarmalayıcı sınıflar** ilkel değerleri nesne olarak taşır. Otomatik kutulama `valueOf`, kutudan çıkarma `xxxValue()` çağrısıdır. −128…127 arası `Integer`'lar önbellekten gelir; `null` bir sarmalayıcıyı kutudan çıkarmak `NullPointerException` fırlatır.
- **Kayıt** değiştirilemez veri sınıfıdır; yapıcı, erişimciler, `equals`, `hashCode` ve `toString` kendiliğinden üretilir. **enum** sabit bir değer kümesidir; `values()`, `valueOf()`, `ordinal()` ve `==` ile kullanılır.
- **İç sınıflar**: `static` (dış nesne gerekmez), üye (bir dış nesneye bağlı, `outer.new`), yerel (metodun içinde) ve anonim (adsız, tek nesne).
- Bir nesne tek bir sınıfa aittir, ama birçok tipi olabilir. Sınıf çeşitleri: `enum`, tek nesnelik, soyut, mühürlü, `final`, kayıt, araç ve uygulama sınıfı. Sınıf tasarımında tanımlama, temsil ve gerçekleme ayrılır; temsil gizli tutulur.

4.1.16 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

nesne yönelimli programlama *object-oriented programming* · prosedürel programlama *procedural programming* · sınıf *class* · nesne *object* · örnek *instance* · üye *member* · alan *field* · metot *method* · nokta işleci *dot operator* · erişim belirleyici *access modifier* · erişimci *getter* · değiştirici *setter* · veri gizleme *data hiding* · sarmalama *encapsulation* · yapıcı *constructor* · varsayılan yapıcı *default constructor* · yapıcı zinciri *constructor chaining* · referans *reference* · heap *heap* · yığın *stack* · null *null reference* · çöp toplayıcı *garbage collector* · static üye *static member* · ilklendirici blok *initializer block* · this *this keyword* · gölgeleme *shadowing* · sarmalayıcı sınıf *wrapper class* · otomatik kutulama / kutudan çıkarma *autoboxing / unboxing* · kayıt *record* · enum *enum* · değiştirilemez *immutable* · iç sınıf *inner class* · anonim sınıf *anonymous class* · mühürlü sınıf / arayüz *sealed class / interface* · soyutlama *abstraction*

4.1.17 Kendini Dene

1. `class Sayac2 { static int s = 0; int n; Sayac2() { s += 2; n = s; } }` sınıfından art

arda üç nesne (`p` , `q` , `r`) oluşturuluyor. `p.n` , `q.n` , `r.n` ve `Sayac2.s` kaçtır?

Cevap

`s` `static` 'tir ve üç yapıcı da aynı kutuyu artırır: 2, 4, 6. Her yapıcı o anki değeri nesnenin kendi `n` alanına kopyalar. Sonradan `s` 'nin değişmesi önceki nesnelerin `n` 'ini etkilemez. Sonuç: `2 4 6 6` .

2. `Hucre` sınıfının tek bir `int v` alanı ve `v` 'yi atayan bir yapıcısı var. `static void degis(Hucre h, Hucre g) { h.v = 10; g = h; g.v = 20; }` metodu `a = new Hucre(1)` ve `b = new Hucre(2)` ile `degis(a, b)` olarak çağrılıyor. Sonra `a.v` ve `b.v` kaçtır?

Cevap

`h` , `a` 'nın nesnesini gösterir; `h.v = 10` onu 10 yapar. `g = h` yalnızca yerel kopya `g` 'yi `a` 'nın nesnesine yöneltir; `b` değişmez. `g.v = 20` artık `a` 'nın nesnesine yazar. Sonuç: `a.v = 20` , `b.v = 2` .

3. `class C { static int t = 0; int a = ++t; { t *= 10; } C() { t++; } }` sınıfından iki nesne oluşturuluyor. Birinci nesnenin `a` 'sı, ikinci nesnenin `a` 'sı ve sonunda `C.t` kaçtır?

Cevap

Her nesnede sıra şudur: alan ilk değeri, sonra örnek ilklendirici, sonra yapıcı gövdesi. Birinci nesne: `a = ++t` → t = 1, a = 1; `t *= 10` → 10; `t++` → 11. İkinci nesne: a = 12, t = 120, t = 121. Sonuç: `1 12 121` .

4. `Zincir` sınıfında `int x; String s = "";` alanları ve şu yapıcılar var: `Zincir() { this(1); s += "a"; }` , `Zincir(int x) { this(x, x + 1); s += "b"; }` , `Zincir(int x, int y) { this.x = x * y; s += "c"; }` . `new Zincir()` sonrasında `x` ve `s` nedir?

Cevap

Zincir en içteki yapıcıya kadar iner: `Zincir(1, 2)` . `s = ""` ilk değeri bu yapıcının gövdesinden hemen önce bir kez verilir. En içteki gövde önce çalışır: x = 2, s = "c". Zincir geri sararken her yapıcı kendi kalan satırını çalıştırır: "cb", sonra "cba". Sonuç: `2 cba` .

5. `Integer i1 = 127, i2 = 127, i3 = 1000, i4 = 1000; Long l = 127L;` tanımlarından sonra

`i1 == i2`, `i3 == i4`, `i3.equals(i4)`, `l.equals(127)` ve `l == 127` ne verir?

Cevap

`true false true false true`. 127 önbellekten gelir, 1000 için iki ayrı nesne oluşur. `equals` değerleri karşılaştırır. `l.equals(127)`'de 127 bir `Integer`'a kutulanır; `Long`'un `equals`'ı yalnızca başka bir `Long`'u eşit sayar, bu yüzden `false`. `l == 127`'de ise `l` kutudan çıkarılır ve iki sayı karşılaştırılır: `true`.

6. Bu bölümdeki `RasyonelSayi` sınıfıyla `new RasyonelSayi(10, -4)` nasıl yazdırılır? `new RasyonelSayi(2, 6).topla(new RasyonelSayi(1, 6))` ne verir?

Cevap

`ebob(10, 4) = 2`; pay ve payda 2'ye bölünür ve işaret paydada kalır: `5/-2`. İkinci ifadede 2/6 önce 1/3 olarak saklanır. 1/3 + 1/6 = (1 × 6 + 1 × 3) / 18 = 9/18 bulunur ve yapıcı bunu sadeleştirir: `1/2`.

7. Bir öğrenci `main`'deki `non-static method ... cannot be referenced from a static context` hatasından kurtulmak için sınıftaki bütün alanları ve metotları `static` yapıyor. Program derleniyor. Bu çözümün sorunu nedir?

Cevap

Alanlar `static` olunca her alanın sınıfta tek bir kopyası kalır. İki nesne oluşturulsa bile ikisi aynı alanları paylaşır; birinin adını değiştirmek ötekinin adını da değiştirir. Program fiilen prosedürel hâle gelir. Doğru çözüm, nesneye bağlı metotları bir nesne üzerinden çağırmak ve yalnızca gerçekten nesneden bağımsız olanları `static` yapmaktır.

8. `RasyonelSayi` sınıfında `new RasyonelSayi(0, 0)` ne olur? Bu hatayı ve işaret sorununu gidermek için hangi metot, nasıl değiştirilir? Sınıfı kullanan kodun değişmesi gerekir mi?

Cevap

`ebob(0, 0)` 0 döndürür ve `pay / e` satırı `ArithmeticException: / by zero` fırlatır. Yalnızca yapıcı değişir: başta `payda == 0` ise anlamlı iletili bir `IllegalArgumentException` fırlatılır; sadeleştirmeden sonra `payda` negatifse pay ve paydanın işareti çevrilir. Alanlar `private` ve temsil gizli olduğu için sınıfı kullanan kod hiç değişmez.

4.1.18 İleri okuma

- Oracle, *The Java Tutorials*, "Classes and Objects" dersi: sınıf tanımlama, yapıcılar, `this`, erişim denetimi, sınıf üyeleri ve iç içe sınıflar konularını kısa örneklerle işler; çevrim içi ve ücretsizdir.

- A. B. Downey ve C. Mayfield, *Think Java* (2. baskı, O'Reilly, 2019), 10. ve 11. bölümler: nesneleri, referansları ve kendi sınıfını yazmayı yığın diyagramlarıyla adım adım anlatır.
- J. Bloch, *Effective Java* (3. baskı, Addison-Wesley, 2018): `equals`, `hashCode` ve `toString`'in doğru yazılması, değiştirilemez sınıflar ve `static` fabrika metotları üzerine deneyimli bir programcının önerileri.
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21, 8. bölüm: alanların, ilklendiricilerin, yapıcıların, `enum`'ların ve kayıtların kesin kurallarını tanımlar; ilklendirme sırasıyla ilgili sorularda başvurulacak kaynaktır.

4.1.19 Kaynaklar

- Oracle, *The Java Tutorials*: “Classes and Objects”, “Nested Classes”, “Enum Types”, “Record Classes”, “Autoboxing and Unboxing”.
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21 Edition (Bölüm 8 Classes: 8.3 Fields, 8.6–8.7 Initializers, 8.8 Constructors, 8.9 Enum Classes, 8.10 Record Classes).
- P. Deitel ve H. Deitel, *Java: How to Program, Early Objects*, 11. baskı, Pearson, 2017 (Bölüm 3 ve 8: Classes and Objects).
- K. Sierra ve B. Bates, *Head First Java*, 3. baskı, O'Reilly, 2022.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, 2022 (Bölüm 4: Objects and Classes).
- A. B. Downey ve C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019 (Bölüm 10–11).
- J. Bloch, *Effective Java*, 3. baskı, Addison-Wesley, 2018 (`equals`, `hashCode`, `toString` maddeleri).

KISIM II · BÖLÜM 4.2

Sarmalama

Bir sınıf yazmak, veriyi ve o veriyle çalışan metotları bir araya getirmekle başlar. Ama bu birliktelik tek başına yetmez. Alanlar herkese açıksa her kullanıcı nesneyi istediği gibi değiştirir; saniyesi 75 olan bir saat ya da bakiyesi eksiye düşmüş bir hesap ortaya çıkar. Sarmalama, nesnenin verisini dışarıya kapatıp ona yalnızca sınıfın kendi metotlarıyla ulaşılmasını sağlar. Böylece kurallar tek bir yerde, sınıfın içinde uygulanır. Not bu fikri adım adım kurar. İlk kesimler bilgi saklamayı, "ne / nasıl" ayrımını ve metodun ön ve son koşullarından oluşan sözleşmeyi anlatır. Ardından erişim belirleyiciler, erişimci ve değiştirici metotlar ve bir banka hesabı sınıfının ayrıntılı izlemesi gelir. Not daha sonra arayüz ile gerçeklemeyi ayırır, dışarı sızan referansları ve değiştirilemez sınıfları inceler. Son kesim sınıfları paketlerde gruplamayı ve `import` bildirimini anlatır. Bütün programlar JDK 21 ile derlenip çalıştırılmıştır; çıktılar, izleme tablolarındaki son değerler ve derleyici iletileri bu çalıştırmalardan alınmıştır.

ÖĞRENME ÇIKTILARI

Sarmalamayı açıklamak
Veriyi ve onu işleyen metotları tek sınıfta toplamayı, iç ayrıntıları gizlemeyi (bilgi saklama) ve "ne / nasıl" ayrımını örnekle anlatmak.

Metot sözleşmesi yazmak
Bir metodun ön koşulunu ve son koşulunu Javadoc yorumuyla belirtmek; ön koşul bozulduğunda ne yapılacağını seçmek ve kodda uygulamak.

Erişimi doğru seçmek
`public`, `protected`, paket erişimi ve `private` düzeylerinin kimlere açık olduğunu söylemek; derleyicinin verdiği erişim hatalarını okumak.

Erişimci ve değiştirici yazmak
Alanları `private` yapıp doğrulama yapan erişim metotları yazmak; `this.alan = alan` kalıbını ve gölgeleme tuzağını bilmek.

Arayüz ile gerçeklemeyi ayırmak
Sınıfın dış yüzünü değiştirmeden iç yapısını değiştirmek; `interface` ve `implements` ile tanışmak; sızan referansı savunmacı kopya ile kapatmak ve değiştirilemez sınıf tasarlamak.

Paket kullanmak
`package` ve `import` ile sınıfları gruplamak, paketli bir programı komut satırında derleyip çalıştırmak, sınıf adı çakışmalarını çözmek.

4.2.1 Sarmalama ve bilgi saklama

Bir sınıfın alanları ve metotları tanımlandıktan sonra ilk tasarım sorusu şudur: bu alanlara kim dokunabilir? Sınıfın her alanı her yerden okunup yazılabiliyorsa, sınıfın kendi kuralları kâğıt üzerinde kalır. Bir saatin saniyesi 0 ile 59 arasında olmalıdır. Kuralı bilen sınıftır, ama alanı değiştiren her satır onu ayrı ayrı hatırlamak zorunda kalır. Programda yüzlerce satır varsa birinin unutması kaçınılmazdır. Sarmalama bu sorunu kökünden çözer: kuralı uygulama işini sınıfın kendisine bırakır.

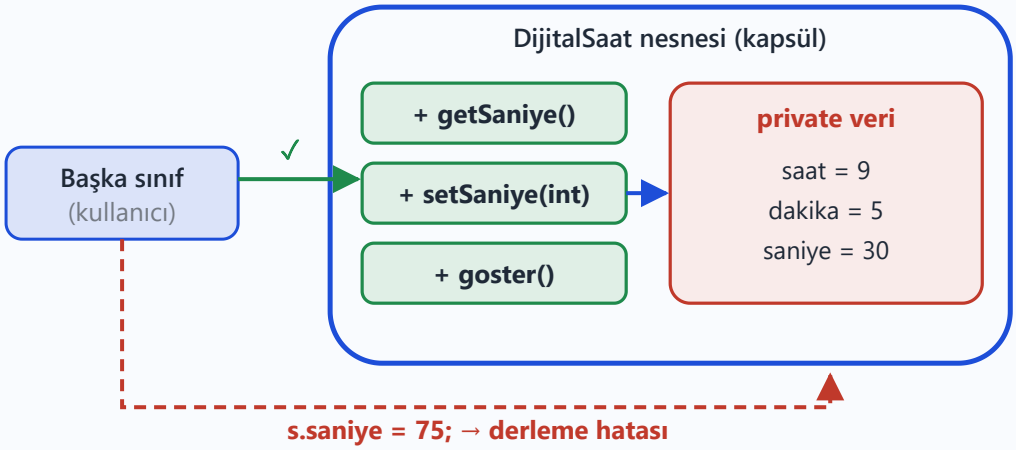
4.2.1.1 Sarmalama nedir?

Tanım 4.2.1

Sarmalama (*encapsulation*), bir nesnenin verisini (alanlarını) ve bu veriyi işleyen metotları tek bir birimde, sınıfta toplamaktır. Veri dışarıya kapatılır (`private`); dışarıdan yalnızca sınıfın sunduğu metotlar çağrılabilir.

Tanımda iki ayrı fikir vardır. Birincisi **bir araya getirmek**tir: veri ile onu işleyen kod aynı sınıfta durur. İkincisi **kapatmak**tır: veri dışarıdan doğrudan görülmez, değiştirilmez. İkinci fikir olmadan birincisi yarım kalır. Alanları `public` olan bir sınıf da veriyle metotları bir arada tutar, ama veriyi korumaz.

Sarmalamayı ilaç kapsülüne benzetmek mümkündür. Kapsülün içindeki toz görünmez; hasta yalnızca kapsülü yutar ve tozun nasıl hazırlandığıyla ilgilenmez. Sınıf da kendi verisinin bekçisidir. Saniyeyi 75 yapmak isteyen bir çağrı, sınıfın kapısından geçemez. Şekil 4.2.1 bu düzeni bir `DijitalSaat` nesnesi üzerinde gösterir.



Şekil 4.2.1: Sarmalanmış bir `DijitalSaat` nesnesi. Özel veri (`saat` , `dakika` , `saniye`) nesnenin içindedir ve dışarıya kapalıdır. Başka bir sınıf yalnızca `getSaniye()` , `setSaniye(int)` ve `goster()` gibi açık metotları çağırabilir. Alana doğrudan yazma denemesi (`s.saniye = 75;`) derleme hatasıyla durdurulur.

Şekildeki kesik kırmızı ok, sarmalamanın en somut kazancını gösterir. Alana doğrudan yazmak bir çalışma hatası değil, bir **derleme hatasıdır**. Program yanlış değeri hiçbir zaman göremez, çünkü böyle bir program derlenmez. Geçersiz değeri durdurma işi çalışma anına bırakılmaz; derleyici bunu kod yazılırken yakalar.

Sarmalama, nesne yönelimli programlamanın dört temel ilkesinden biridir. Ötekiler kalıtım, çok biçimlilik ve soyutlamadır. Dördü birbirini tamamlar: kalıtım ve çok biçimlilik sınıflar arasındaki ilişkiyi düzenler; sarmalama ve soyutlama ise tek bir sınıfın dışarıya nasıl göründüğünü belirler.

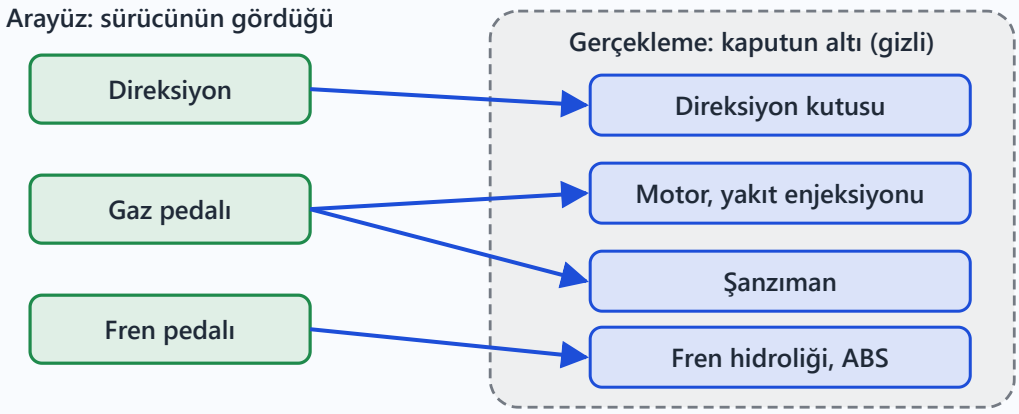
4.2.1.2 Bilgi saklama: ne ve nasıl

Tanım 4.2.2

Bilgi saklama (*information hiding*), bir sınıfın iç yapısını onu kullananlardan gizleme ilkesidir. Tasarım, bir işin *ne* yapıldığını *nasıl* yapıldığından ayırır. Kullanıcı yalnızca metodun ne işe yaradığını bilir; iç ayrıntıları bilmeden de metodu doğru kullanabilir.

Bilgi saklama, **soyutlamayla** (*abstraction*) iç içe bir kavramdır ve bazı kaynaklar ikisini aynı adla anar. Soyutlama ayrıntıyı bir kenara bırakıp yalnızca önemli özellikleri göstermektir. Bilgi saklama ise bu ayrıntıların kullanıcıya gerçekten ulaşmamasını sağlar. Biri bakış açısıdır, öteki o bakış açısını koruyan duvardır.

Günlük hayattaki en bilinen örnek arabadır. Sürücü direksiyonu, gaz pedalını ve freni görür ve kullanır. Direksiyon kutusunun, yakıt enjeksiyonunun ya da fren hidroliğinin nasıl çalıştığını bilmesi gerekmez. Üretici motoru değiştirse bile sürücü arabayı aynı biçimde kullanmaya devam eder. Şekil 4.2.2 bu ayrımı iki katman olarak çizer.



Şekil 4.2.2: Araba benzetmesi. Solda sürücünün gördüğü arayüz (direksiyon, gaz pedalı, fren pedalı), sağda kaputun altındaki gizli gerçekleme (direksiyon kutusu, motor ve yakıt enjeksiyonu, şanzıman, fren hidroliği ve ABS). Her kumanda, sürücünün bilmediği bir ya da birkaç iç parçayı harekete geçirir.

Aynı fikir Java'da da her gün kullanılır. `System.out.println` metodu sürekli çağrılır, ama karakterlerin ekrana nasıl gönderildiğini bilmek gerekmez. İşletim sistemiyle konuşan kod, karakter kodlaması, tampon bellek gibi ayrıntılar `println`'ın arkasında saklıdır. Kullanıcının bildiği tek şey, metodun ne yaptığıdır: verilen değeri yazar ve satır sonuna geçer.

4.2.1.3 Tek çağrının arkası

Bilgi saklamayı kodda görmek için bir veri tabanına kayıt ekleyen küçük bir sınıf ele alalım. `VeriIslemleri` sınıfı, bağlantıyı açma, veriyi yazma ve bağlantıyı kapatma adımlarını tek bir metodun içine toplar:

```
public class VeriIslemleri {
    private VeritabaniBaglantisi baglanti;
    // metodun iç yapısı bilinmeden de kullanılabilir.
    public void veriEkle(String veri) {
        baglanti = new VeritabaniBaglantisi();
        baglanti.baglan();
        baglanti.veriEkle(veri);
        baglanti.kapat();
    }
}
```

```
public class Ana {
    public static void main(String[] args) {
        VeriIslemleri islem = new VeriIslemleri();
        islem.veriEkle("Ayşe");
    }
}
```

Bu programı derlemek için `VeritabaniBaglantisi` sınıfı da gerekir. Gerçek bir uygulamada bu sınıf bir veri tabanı sürücüsüyle konuşurdu. Notun örneğinde yalnızca ekrana yazan basit bir sınıftır:

```
public class VeritabaniBaglantisi {
    public void baglan() { System.out.println("Bağlantı açıldı"); }
    public void veriEkle(String veri) { System.out.println("Eklendi: " + veri); }
    public void kapat() { System.out.println("Bağlantı kapatıldı"); }
}
```

```
Bağlantı açıldı
Eklendi: Ayşe
Bağlantı kapatıldı
```

`Ana` sınıfının yaptığı tek şey `islem.veriEkle("Ayşe")` çağrısıdır. Bu satır *ne* istendiğini söyler: "Ayşe kaydını ekle." Bağlantının açılması, verinin yazılması ve bağlantının kapatılması, yani işin *nasıl* yapıldığı, `VeriIslemleri`'nin içinde gizlidir. `baglanti` alanı `private` olduğu için `Ana` onu göremez de değiştiremez de. Yarın `VeritabaniBaglantisi` yerine gerçek bir sürücü kullanılsa, `Ana`'nın tek bir satırı bile değişmez.

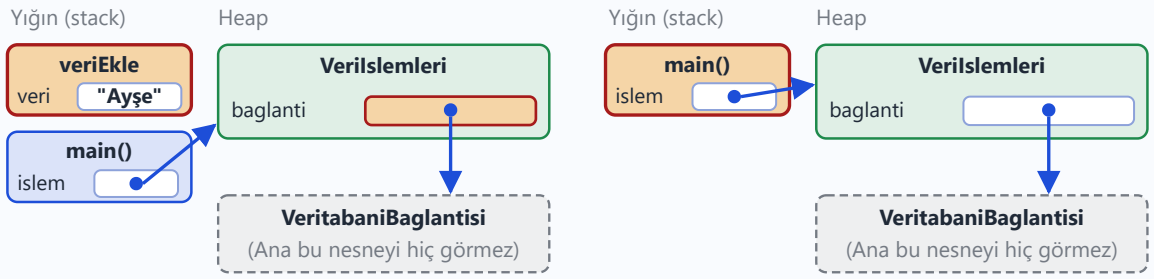
Çağrının arkasında neler olduğu Tablo 4.2.1'de adım adım verilmiştir. Satır numaraları, iki sınıfın art arda yazıldığı izleme kodunun düzenine göredir: 1–10. satırlar `VeriIslemleri`, 11–16. satırlar `Ana` sınıfıdır.

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	13: `VeriIslemleri islem = new VeriIslemleri();`	main()	islem=→VeriIslemleri	—
2	4: `public void veriEkle(String veri) {`	main() → islem.veriEkle("Ayşe")	veri="Ayşe"	—
3	5: `baglanti = new VeritabaniBaglantisi();`	main() → islem.veriEkle("Ayşe")	veri="Ayşe", baglanti=→nesne	—
4	6: `baglanti.baglan();`	main() → islem.veriEkle("Ayşe") → baglanti.baglan()	veri="Ayşe"	`Bağlantı açıldı`
5	7: `baglanti.veriEkle(veri);`	main() → islem.veriEkle("Ayşe") → baglanti.veriEkle(veri)	veri="Ayşe"	`Eklendi: Ayşe`
6	8: `baglanti.kapat();`	main() → islem.veriEkle("Ayşe") → baglanti.kapat()	veri="Ayşe"	`Bağlantı kapatıldı`
7	15: `}`	main()	islem=→VeriIslemleri	—

Tablo 4.2.1: `islem.veriEkle("Ayşe")` çağrısının izlenmesi. Çağrı yığını sütununda en sağdaki çerçeve en üsttedir; değişkenler sütunu o anda çalışan çerçeveyi gösterir.

İlk adımda `new VeriIslemleri()` heap'te bir nesne oluşturur. `islem` değişkeni bu nesneyi gösteren bir referanstır. Nesnenin `baglanti` alanı henüz `null`'dır, çünkü referans tipindeki alanların varsayılan değeri `null`'dır. İkinci adımda `veriEkle` çağrılır; yığına yeni bir çerçeve eklenir ve `veri` parametresi `"Ayşe"` değerini alır.

Üçüncü adımda gizli iş başlar. İkinci bir nesne oluşur ve `private` `baglanti` alanı onu gösterir. Şekil 4.2.3 bu anı ve çağrının sonunu yan yana verir. Dördüncü adımda `baglan()` çağrılır; yığında artık üç çerçeve vardır: `main`, `veriEkle` ve `baglan`. `Ana` bu çerçevelerin hiçbirinden haberdar değildir. Beşinci adımda veri, bağlantı nesnesinin `veriEkle` metoduna gider. İki metodun adı aynıdır, ama farklı sınıflara aittir; Java hangisinin çağrıldığını nesnenin tipinden bilir. Altıncı adımda bağlantı kapatılır.



Şekil 4.2.3: Tek çağrının iki anı. Solda `veriEkle` çalışıyor; `VeriIslemleri` nesnesinin özel `baglanti` alanı yeni oluşturulan `VeritabaniBaglantisi` nesnesini gösteriyor. Sağda çağrı bitmiş, yığında yalnızca `main` kalmış; bağlantı nesnesi kullanıcının hiç görmediği bir yerde duruyor.

Yedinci adımda akış `main`'e döner. Kullanıcı tek satır yazdı ve üç ayrı işlemin hiçbirini bilmedi. Bilgi saklamanın anlamı tam olarak budur. Bu düzenin bir maliyeti yoktur: çağrı yığınına eklenen her çerçeve sabit zamanda açılıp kapanır. Gizlenen iş, gizlenmese de yapılacak işin aynısıdır.

Sık yapılan hata

Gizli ayrıntıyı kullanan kodu dışarıda yazmak. `Ana` içinde bağlantıyı kendisi açıp kapatan bir programcı, `VeriIslemleri`'nin işini ikinci kez yapmış olur. Bağlantı biçimi değiştiğinde iki yerin birden düzeltilmesi gerekir. Bir sınıfın bir işi tek çağrıyla sunduğu yerde, o işin adımları dışarıda tekrar yazılmaz.

4.2.2 Ön ve son koşullar

Bilgi saklama, kullanıcının metodun içini bilmemesini ister. Ama kullanıcı bir şeyi mutlaka bilmelidir: metot hangi durumda doğru çalışır ve çalıştığında ne garanti eder? Bu bilgi kodun içinde değil, metodun dış yüzündedir. Bu kesim, metodun kullanıcıyla yaptığı anlaşmayı ve bu anlaşmanın kodda nasıl uygulandığını anlatır.

4.2.2.1 Metodun sözleşmesi

> **Tanım 4.2.3**
>
> **Ön koşul** (*precondition*), metot çağrılmadan önce doğru olduğu varsayılan koşuldur; onu sağlamak çağıranın sorumluluğudur. **Son koşul** (*postcondition*), ön koşul sağlandıysa metot bittikten sonra doğru olacağı garanti edilen durumdur; onu sağlamak metodun sorumluluğudur. İkisi birlikte metodun **sözleşmesini** (*contract*) oluşturur.

Sözleşme iki taraflıdır. Çağıran "Boş olmayan bir veri veriyorum" der, metot da "Ben de onu ekleyip `true` döndürüyorum" diye cevap verir. Kullanıcı yalnızca sözleşmeyi bilir, gövdeyi bilmez. Ön koşul bozulursa metot hiçbir şey garanti etmek zorunda değildir. Bu, sorumluluğun açıkça paylaşılması demektir: hata çağıranın verdiği değerdeyse çağıran düzeltir, metot doğru değerle yanlış sonuç veriyorsa metot düzeltilir.

Ön ve son koşul ilk adımda yorum satırı olarak yazılabilir. Sunumdaki iki küçük parça, aynı `veriEkle` metodunun önce ön koşulunu, sonra son koşulunu belirtir:

```
public class VeriIslemleri {
    // Ön koşul: veri != null
    public void veriEkle(String veri) {
        // verinin null olup olmadığını kontrol et
        // ...
    }
}
```

```
public class VeriIslemleri {
    // Son koşul: veri tabanına veri eklendiyse true döner
    public boolean veriEkle(String veri) {
        // veriyi ekleyip eklemediğini kontrol et
        // ...
        return true;
    }
}
```

İkinci parçada metodun dönüş tipi de değişmiştir: `void` yerine `boolean`. Son koşul "eklendiyse `true` döner" dediği için metodun bir değer döndürmesi gerekir. Sözleşmeyi yazmak, metodun imzası üzerinde de düşünmeyi zorunlu kılar.

4.2.2.2 Sözleşmeyi kodda uygulamak

Yorum satırı tek başına hiçbir şeyi denetlemez. Aşağıdaki sürüm sözleşmeyi hem yorumda yazar hem de gövdenin ilk satırında uygular:

```
public class VeriIslemleri {
  private int kayitSayisi = 0;

  /**
   * Veri tabanına bir kayıt ekler.
   * Ön koşul: veri null değil ve boş değil.
   * Son koşul: eklendiyse true döner,
   *            kayitSayisi 1 artar.
   */
  public boolean veriEkle(String veri) {
    if (veri == null || veri.isEmpty()) {
      return false;   // ön koşul bozuk
    }
    // ... veriyi veri tabanına ekle ...
    kayitSayisi++;
    return true;      // son koşul sağlandı
  }

  public int getKayitSayisi() {
    return kayitSayisi;
  }
}
```

Metodun başındaki `/** ... */` bir **Javadoc** yorumudur: metodun nasıl kullanılacağını anlatır ve araçlarla belgeye dönüştürülebilir. Gövdenin ilk işi ön koşulu denetlemektir. `veri` `null` ya da boşsa metot nesneye dokunmadan `false` döndürür. Ön koşul sağlanmışsa kayıt eklenir, `kayitSayisi` bir artar ve `true` döner; son koşul böylece yerine getirilir. Sunumdaki ilk `main` metodu üç çağrı yapar:

```
// main içinde:
VeriIslemleri islem = new VeriIslemleri();
System.out.println(islem.veriEkle("Ali"));
System.out.println(islem.veriEkle(""));
System.out.println(islem.veriEkle(null));
System.out.println(islem.getKayitSayisi());
```

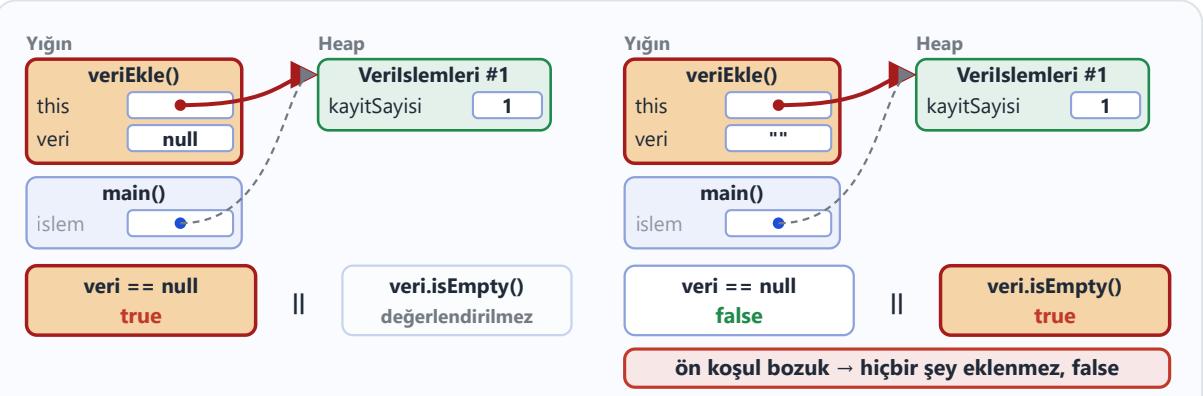
```
true
false
false
1
```

Dört çağrının izlemesi Tablo 4.2.2'de verilmiştir. Bu kez sırayla `"Ayşe"`, `null`, `""` ve `"Can"` eklenmeye çalışılır. Satır numaraları izleme kodunun düzenine göredir: 1–7. satırlar `veriEkle`, 9–14. satırlar `main` içindedir.

Adım	Satır	Değişkenler	Yeni çıktı
1	9: `VeriIslemleri islem = new VeriIslemleri();`	islem=→#1, kayitSayisi=0	—
2	10: `System.out.println(islem.veriEkle("Ayşe"));` 2: `if (veri == null || veri.isEmpty()) {`	veri="Ayşe", kayitSayisi=0	—
3	5: `kayitSayisi++;` 6: `return true; // son koşul sağlandı`	veri="Ayşe", kayitSayisi=1	`true`
4	11: `System.out.println(islem.veriEkle(null));` 2: `if (veri == null || veri.isEmpty()) {`	veri=null, kayitSayisi=1	—
5	3: `return false; // ön koşul bozuk`	veri=null, kayitSayisi=1	`false`
6	12: `System.out.println(islem.veriEkle(""));` 2: `if (veri == null || veri.isEmpty()) {` 3: `return false; // ön koşul bozuk`	veri="", kayitSayisi=1	`false`
7	13: `System.out.println(islem.veriEkle("Can"));` 5: `kayitSayisi++;` 6: `return true; // son koşul sağlandı`	veri="Can", kayitSayisi=2	`true`
8	14: `System.out.println(islem.getKayitSayisi());`	kayitSayisi=2	`2`

Tablo 4.2.2: `veriEkle` sözleşmesinin dört çağrıyla izlenmesi. Değişkenler sütunundaki `kayitSayisi`, nesnenin alanıdır.

İkinci adımda `"Ayşe"` ne `null` ne de boştur. `||` işlecinin iki tarafı da yanlış çıkar, `if` bloğu atlanır. Üçüncü adımda sayaç 1 olur ve `true` döner. Dördüncü adım en öğretici olanıdır: `veri == null` doğrudur ve `||` **kısa devre** yapar. Sol taraf doğru olduğu için sağ taraf hiç değerlendirilmez; `null.isEmpty()` çağrısı yapılmaz ve `NullPointerException` oluşmaz. Şekil 4.2.4 bu durumu boş dizgi durumuyla yan yana gösterir.



Şekil 4.2.4: Ön koşulun iki farklı yoldan bozulması. Solda `veri` `null` 'dır; `veri == null` doğru çıkar ve `veri.isEmpty()` hiç değerlendirilmez. Sağda `veri` boş dizgidir; ilk koşul yanlış, ikinci koşul doğrudur. İki durumda da `kayitSayisi` 1'de kalır.

Koşulların sırası bu yüzden önemlidir. `veri.isEmpty() || veri == null` biçiminde yazılsaydı, `null` argümanında önce `isEmpty()` çağrılır ve program `NullPointerException` ile dururdu. JDK 21 ile denendiğinde gerçekten de böyle olur. Önce `null` denetimi, sonra nesne üzerindeki çağrı: bu sıra Java'da her zaman korunur.

Altıncı adımda boş dizgi gelir. `""` bir nesnedir, `null` değildir; ama uzunluğu 0 olduğu için `isEmpty()` doğru döndürür. Yedinci adımda `"Can"` eklenir ve sayaç 2 olur. Dört çağrıdan yalnızca sözleşmeye uyan ikisi kayıt ekledi. Nesnenin durumu hiçbir çağrıda bozulmadı.

Bir ayrıntı daha: `" "` (tek boşluk) `isEmpty()` sınamasından geçer, çünkü uzunluğu 1'dir. Aynı nesneye `veriEkle(" ")` gönderildiğinde metot `true` döndürür ve kaydı sayar. Boşluklardan oluşan veriyi de reddetmek istenirse koşul `veri.trim().isEmpty()` ya da Java 11'den beri `veri.isBlank()` ile yazılır. Ön koşulun tam olarak ne dediği, kodda neyin denetlendiğini belirler.

Ön koşul denetiminin maliyeti küçüktür: iki karşılaştırma ve en çok bir metot çağrısı, yani sabit zaman. Buna karşılık kazancı büyüktür; geçersiz bir kayıt nesneye hiç giremez.

4.2.2.3 Ön koşul bozulunca

Ön koşul bozulduğunda metodun ne yapacağı da bir tasarım kararıdır. Tablo 4.2.3 üç yaygın yaklaşımı karşılaştırır.

Yaklaşım	Örnek	Çağıran ne fark eder?
Sessizce yok saymak	`setSaniye(75)` hiçbir şey yapmaz	Hiçbir şey; hata gözden kaçabilir.
Durum döndürmek	`veriEkle(null)` → `false`	Dönüş değerine bakarsa fark eder.
İstisna fırlatmak	`throw new IllegalArgumentException(...)`	Yakalamazsa program durur; hata kaybolmaz.

Tablo 4.2.3: Ön koşul bozulduğunda seçilebilecek üç yaklaşım.

İlk yaklaşım en basit olanıdır, ama hatayı gizler. Çağıran 75 yazdığını sanır, oysa saat değişmemiştir. İkinci yaklaşım hatayı bildirir, fakat çağıranın dönüş değerine bakmasına bağlıdır. Dönüş değerini yok sayan bir çağrı, ilk yaklaşımdaki kadar sessizdir. Üçüncü yaklaşım hatayı görmezden gelinemez hâle getirir. Aşağıdaki `setSaniye`, geçersiz değerde `IllegalArgumentException` fırlatır:

```
public void setSaniye(int saniye) {
    if (saniye < 0 || saniye > 59)
        throw new IllegalArgumentException(
            "saniye 0-59 olmalı: " + saniye);
    this.saniye = saniye;
}
// main içinde:
DijitalSaat s = new DijitalSaat();
try {
    s.setSaniye(30);
    s.setSaniye(75);
    System.out.println("buraya gelinmez");
} catch (IllegalArgumentException e) {
    System.out.println("Hata: " + e.getMessage());
}
System.out.println(s.getSaniye());
```

```
Hata: saniye 0-59 olmalı: 75
30
```

`s.setSaniye(30)` başarılıdır ve saniye 30 olur. `s.setSaniye(75)` çağrısında koşul doğru çıkar ve `throw` deyimi istisnayı fırlatır. Atama satırı çalışmaz; metot o noktada biter. Akış `try` bloğunun geri kalanını atlar, bu yüzden `"buraya gelinmez"` yazılmaz. Kontrol `catch` bloğuna geçer ve istisnanın iletisi yazılır. Son satırda saniye hâlâ 30'dur; nesnenin durumu bozulmamıştır.

Tanım 4.2.4

İstisna (*exception*), program çalışırken normal akışı bozan ve nesne olarak fırlatılan olaydır. `throw` deyimi bir istisna nesnesi fırlatır; `try` bloğunda fırlatılan istisna, tipine uyan `catch` bloğunda yakalanır.

İstisna yakalanmazsa ne olur? Aynı `setSaniye(75)` çağrısı `try` dışında yapıldığında JDK 21 şu çıktıyı verir:

```
Exception in thread "main" java.lang.IllegalArgumentException: saniye 0-59 olmalı: 75
        at DijitalSaat.setSaniye(DijitalSaat.java:6)
        at Yakalanmaz.main(Yakalanmaz.java:4)
```

İletinin ilk satırı istisnanın tipini ve iletisini, sonraki satırlar istisnanın nerede doğduğunu gösterir. `DijitalSaat.java:6` `throw` deyiminin bulunduğu satırdır; `Yakalanmaz.java:4` ise hatalı çağrının yapıldığı satırdır. İstisnanın güçlü yanı budur: hatalı değer, çağrının yapıldığı yerde görünür hâle gelir.

`public` metotlarda argüman hatası için Java'nın alışılmış seçimi `IllegalArgumentException`'dır. Java kütüphanesi de aynı yolu izler. Örneğin `Integer.parseInt("abc")` bir `NumberFormatException` fırlatır ve bu sınıf `IllegalArgumentException`'ın bir alt türüdür. Hangi yaklaşım seçilirse seçilsin, seçim sözleşmenin parçası olarak belgelenmelidir. Kullanıcı metodun bozuk argümanda ne yapacağını belgeden okuyabilmelidir.

4.2.2.4 Sözleşmeyi Javadoc ile yazmak

Javadoc yorumları, sözleşmeyi hem okunabilir hem de araçlarla işlenebilir biçimde yazmayı sağlar. Yorum `/**` ile başlar; ilk cümle metodun ne yaptığını özetler. Ardından gelen **etiketler** sözleşmenin parçalarını ayrı ayrı belirtir:

```
/**
 * Hesaptan para çeker.
 * @param miktar çekilecek tutar; {@code 0 < miktar <= bakiye} olmalı
 * @return işlemden sonraki bakiye (eski bakiye - miktar)
 * @throws IllegalArgumentException miktar bu aralıkta değilse
 */
public double paraCek(double miktar) {
    if (miktar <= 0 || miktar > bakiye)
        throw new IllegalArgumentException("Geçersiz miktar: " + miktar);
    bakiye -= miktar;
    return bakiye;
}
```

Etiket	Ne yazılır?	Örnekte
`@param`	Her parametre ve onun ön koşulu	`miktar` için `0 < miktar <= bakiye`
`@return`	Dönüş değeri, yani son koşul	işlemden sonraki bakiye
`@throws`	Sözleşme bozulunca ne olacağı	`IllegalArgumentException`

Tablo 4.2.4: Sözleşmeyi yazmakta kullanılan Javadoc etiketleri.

Yorumdaki `{@code ...}` yazımı bir ayrıntı değildir. Javadoc, yorumların içini HTML olarak işler. Ham `<` işareti bir HTML etiketinin başlangıcı sanılır. Aynı yorum `{@code}` olmadan `0 < miktar <= bakiye` diye yazıldığında JDK 21'in `javadoc` aracı iki `malformed HTML` hatası verir ve üretilen sayfada eşitsizliğin yerine `invalid input` uyarısı görünür. `{@code ...}` içindeki metin ise olduğu gibi, kod yazı tipiyle gösterilir.

`javadoc -d belge BankaHesabi.java` komutu bu yorumlardan `belge` klasörüne HTML sayfaları üretir. Java kütüphanesinin çevrim içi belgeleri de aynı yolla oluşturulur. `String` ya da `Math` sınıfının belgesinde okunan her açıklama, o sınıfların kaynak kodundaki Javadoc yorumlarından gelir.

Javadoc'a ne yazılacağı sorusunun cevabı bilgi saklamadan çıkar. Metodun **ne** yaptığı ve kullanım koşulları yazılır. **Nasıl** yaptığı yazılmaz; o, gizlenen gerçeklemedir ve değişebilir. Javadoc'un okuru, gövdeyi hiç görmeyecek olan kullanıcıdır. Bu bakışla sınıfın dış yüzü, `public` metotların imzaları ile bu yorumların toplamıdır.

Örnekteki `paraCek`, sözleşmeyi istisnayla uygular. JDK 21 ile 1000 TL bakiyeli bir hesapta denendiğinde `paraCek(300.0)` 700.0 döndürür. `paraCek(5000.0)` ve `paraCek(-20)` ise `Geçersiz miktar: 5000.0` ve `Geçersiz miktar: -20.0` iletileriyle istisna fırlatır. İkinci iletideki `-20.0`'a dikkat: `int` olarak verilen `-20` argümanı `double` parametreye genişletilmiştir.

4.2.2.5 Sınıf değişmezi ve assert

Ön ve son koşul tek bir metoda aittir. Bazı kurallar ise sınıfın bütün nesneleri için her an geçerli olmalıdır. Saatin saniyesi her zaman 0–59 arasında, banka hesabının bakiyesi her zaman sıfır ya da pozitif olmalıdır.

Tanım 4.2.5

Sınıf değişmezi (*class invariant*), bir sınıfın her nesnesinin durumunun her zaman sağlaması gereken kuraldır. Değişmez, nesne kurulduğunda sağlanır ve her `public` metot bittiğinde yeniden doğru olmalıdır.

Değişmezi geliştirme sırasında denetlemek için Java'nın `assert` deyimi kullanılır. `assert koşul : mesaj;` deyimi koşulu hesaplar; koşul yanlışsa mesajla birlikte bir `AssertionError` fırlatır. Aşağıdaki `Sayac` sınıfında bilerek bir hata bırakılmıştır: değer 9'dan sonra 10 olur, oysa değişmez 0–9 aralığını ister.

```
class Sayac {
    private int deger;   // değişmez: 0 <= deger <= 9
    void arttir() {
        deger = deger + 1;   // hata: 9'dan sonra 10
        assert deger >= 0 && deger <= 9 : "deger=" + deger;
    }
}
public class Ana {
    public static void main(String[] args) {
        Sayac s = new Sayac();
        for (int i = 0; i < 10; i++) s.arttir();
        System.out.println("bitti");
    }
}
```

Program iki kez çalıştırılır. Önce olağan biçimde, `java Ana` komutuyla:

```
bitti
```

Sonra doğrulamalar açılarak, `java -ea Ana` komutuyla:

```
Exception in thread "main" java.lang.AssertionError: deger=10
        at Sayac.arttir(Ana.java:5)
        at Ana.main(Ana.java:11)
```

İki çıktının farkı `assert`'ün en önemli özelliğini gösterir. Doğrulamalar varsayılan olarak **kapalıdır**. `-ea` (*enable assertions*) seçeneği verilmezse `assert` satırı hiç çalışmaz; koşul hesaplanmaz bile. Hata sessizce geçer ve program `bitti` yazar. `-ea` ile ise onuncu artırmada değer 10 olur, koşul yanlış çıkar ve `AssertionError: deger=10` fırlatılır. Yığın izinin ilk satırı hatanın `Sayac.arttir` içinde, `Ana.java` dosyasının 5. satırında doğduğunu gösterir.

Bu özellik `assert`'ün ne için kullanılacağını da belirler. `assert`, programcının "bu asla olmamalı" dediği iç varsayımları geliştirme sırasında denetler: sınıf değişmezleri, son koşullar, ulaşılmaması gereken kod dalları. `public` bir metodun argümanları ise `assert` ile denetlenmez. Doğrulamalar kapalıyken denetim hiç yapılmaz ve hatalı argüman nesneye girer. Kısaca: argüman için istisna, iç tutarlılık için `assert`.

Sık yapılan hata

Kullanıcı girdisini `assert` ile denetlemek. `assert yas >= 0;` satırı yalnızca `-ea` ile çalıştırılan programda işe yarar. Program olağan biçimde çalıştırıldığında negatif yaş hiçbir uyarı vermeden kabul edilir. Dışarıdan gelen değer için `if` ile denetim ve `IllegalArgumentException` kullanılır.

Sık yapılan hata

`assert` koşuluna yan etkili ifade yazmak. `assert liste.remove(x);` gibi bir satır, doğrulamalar kapalıyken hiç çalışmaz; eleman listeden silinmez. Programın davranışı `-ea` seçeneğine göre değişir. `assert` koşulu yalnızca okur, hiçbir şeyi değiştirmez.

4.2.3 Erişim belirleyiciler

Sarmalamanın "kapatma" yarısını Java'da erişim belirleyiciler sağlar. Bir alanı ya da metodu kimlerin kullanabileceğini, tanımın başına yazılan tek bir anahtar sözcük belirler. Kuralı derleyici uygular; programcının iyi niyetine bırakılmaz.

4.2.3.1 public ve private

Tanım 4.2.6

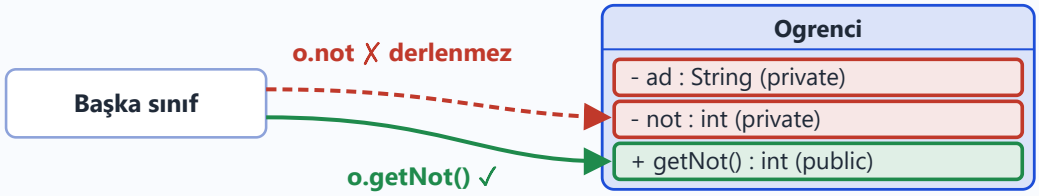
Erişim belirleyici (*access modifier*), bir sınıfa, alana ya da metoda kimin erişebileceğini belirleyen anahtar sözcüktür. `public` üyeye diğer sınıflar doğrudan erişebilir. `private` üyeye yalnızca sınıfın içinden erişilebilir; dışarıdan erişimi derleyici engeller.

Sınıf yazarken izlenen alışkanlık üç maddeyle özetlenir. Sınıflar çoğu zaman `public` tanımlanır. Sınıfın alanları `private` tanımlanır. Alanlara erişmek ve onları değiştirmek için `public` metotlar yazılır. Kısaca: alanlar gizli, davranışlar açık.

```
public class Ogrenci {      // herkese açık sınıf
    private String ad;      // gizli alan
    private int not;        // gizli alan

    public int getNot() {   // açık metot
        return not;
    }
}
```

`Ogrenci` sınıfında `ad` ve `not` alanları `private`'tır. Başka bir sınıf `o.not` yazarak notu okuyamaz; okumak için `o.getNot()` çağrılır. Şekil 4.2.5 bu iki yolu UML gösterimiyle karşılaştırır. UML sınıf diyagramlarında `-` işareti `private`, `+` işareti `public` üye demektir.



Şekil 4.2.5: Başka bir sınıf `Ogrenci`'nin `public` metodu `getNot()`'u çağırabilir (yeşil ok). `private` alan `not`'a doğrudan erişme denemesi ise derlenmez (kırmızı kesik ok).

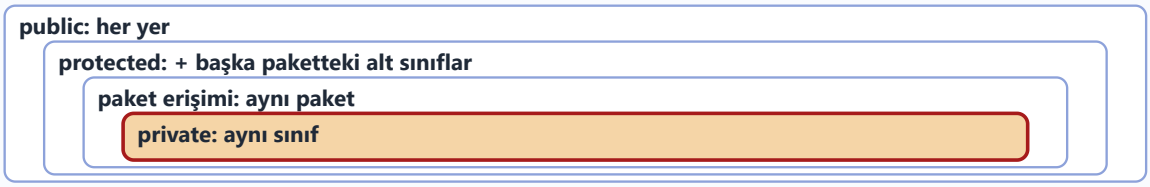
4.2.3.2 Dört erişim düzeyi

Java'da `public` ile `private` arasında iki ara düzey daha vardır. Dört düzeyin kimlere açık olduğu Tablo 4.2.5'te verilmiştir.

Belirleyici	Aynı sınıf	Aynı paket	Alt sınıf (başka paket)	Her yer
`public`	✓	✓	✓	✓
`protected`	✓	✓	✓	✗
(yazılmazsa) paket erişimi	✓	✓	✗	✗
`private`	✓	✗	✗	✗

Tablo 4.2.5: Dört erişim düzeyi. ✓ erişilebilir, ✗ erişilemez demektir.

Belirleyici hiç yazılmazsa **paket erişimi** (*package-private*, bazı kaynaklarda "default") geçerli olur: üyeye yalnızca aynı paketteki sınıflar erişir. `protected` ise alt sınıflar içindir. Aynı paketteki sınıflara ek olarak başka paketteki alt sınıflar da `protected` üyeye ulaşır. Paket, birbiriyle ilgili sınıfların aynı klasörde toplandığı gruptur. Alt sınıf ise `extends` ile başka bir sınıftan türetilen sınıftır. Düzeyler iç içe halkalar oluşturur (Şekil 4.2.6).



Şekil 4.2.6: Erişim düzeyleri iç içe alanlardır. En içte `private` (yalnız aynı sınıf), onu saran halkada paket erişimi (aynı paket) bulunur. Onun dışında `protected` (artı başka paketteki alt sınıflar), en dışta `public` (her yer) yer alır.

Tablodaki her hücre JDK 21 ile sınanmıştır. `p1` paketindeki `A` sınıfına dört alan konmuştur: `public int acik`, `protected int korumali`, belirleyicisiz `int paket` ve `private int gizli`. Dört alana dört farklı yerden yazılmaya çalışıldığında derleyici şu hataları verir:

```
p1/B.java:3: error: gizli has private access in A
    void dene(A a) { a.acik = a.korumali = a.paket = 1; a.gizli = 1; }
                                                        ^
p2/C.java:3: error: paket is not public in A; cannot be accessed from outside package
    void dene() { acik = korumali = 1; paket = 1; gizli = 1; }
                                       ^
p2/C.java:3: error: gizli has private access in A
    void dene() { acik = korumali = 1; paket = 1; gizli = 1; }
                                                  ^
p2/D.java:3: error: korumali has protected access in A
    void dene(p1.A a) { a.acik = 1; a.korumali = 1; a.paket = 1; a.gizli = 1; }
                                    ^
p2/D.java:3: error: paket is not public in A; cannot be accessed from outside package
    void dene(p1.A a) { a.acik = 1; a.korumali = 1; a.paket = 1; a.gizli = 1; }
                                                    ^
p2/D.java:3: error: gizli has private access in A
    void dene(p1.A a) { a.acik = 1; a.korumali = 1; a.paket = 1; a.gizli = 1; }
                                                                 ^
6 errors
```

`A`'nın kendi metodu dört alana da yazabilir; hiç hata yoktur. Aynı paketteki `B` yalnızca `gizli`'ye ulaşamaz. Başka paketteki alt sınıf `C`, `paket` ve `gizli`'ye ulaşamaz. Başka paketteki ilgisiz sınıf `D` ise yalnızca `acik`'e yazabilir. Bu sonuç tablonun satırlarıyla birebir aynıdır. İletiler de düzeye göre değişir: `has private access`, `is not public in A; cannot be accessed from outside package` ve `has protected access`.

Doğru düzeyi seçerken kural basittir: en kısıtlayıcı düzey tercih edilir. Önce `private` düşünülür; gerçekten gerekirse daha açık bir düzeye geçilir. Bir üyeyi sonradan açmak kolaydır ve kimseyi etkilemez. Açık bir üyeyi sonradan kapatmak ise onu kullanan bütün kodu bozar.

Son bir kural da sınıfların kendisiyle ilgilidir. Üst düzey bir sınıf, yani başka bir sınıfın içinde olmayan sınıf, yalnızca `public` ya da paket erişimli olabilir. `private class UstPrivate { }` yazıldığında derleyici `modifier private not allowed here` iletisini verir. `private` ve `protected` sınıflar ancak başka bir sınıfın içinde, iç sınıf olarak tanımlanabilir.

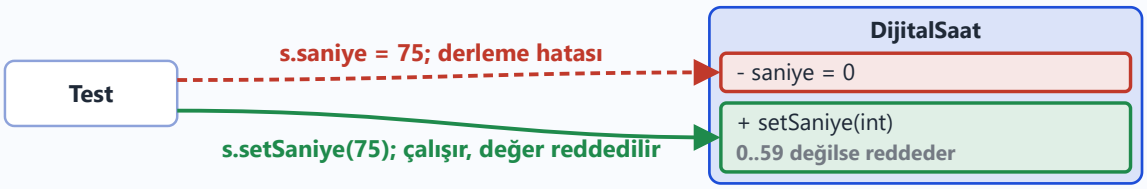
4.2.3.3 private alana erişim denemesi

`DijitalSaat` sınıfında `saniye` alanı `private`'tır. Başka bir sınıf bu alana doğrudan yazmaya çalışırsa ne olur?

```
public class Test {
    public static void main(String[] args) {
        DijitalSaat s = new DijitalSaat();
        s.saniye = 75;
    }
}
```

```
Test.java:4: error: saniye has private access in DijitalSaat
        s.saniye = 75;
         ^
1 error
```

İleti hatanın dosyasını ve satırını (`Test.java:4`), nedenini (`saniye has private access in DijitalSaat`) ve satırın içindeki yerini (`^` işareti) gösterir. Program hiç çalışmaz; `.class` dosyası üretilmez. Alan `public` olsaydı `s.saniye = 75;` sessizce kabul edilir ve saat geçersiz bir duruma düşerdi. Çözüm, değişikliği sınıfın kendi metoduna yaptırmaktır: `s.setSaniye(75);`. Metot geçersiz değeri reddeder (Şekil 4.2.7).



Şekil 4.2.7: `Test` sınıfının `DijitalSaat`'e iki yolu. Alana doğrudan yazmak (`s.saniye = 75;`) derleme hatasıdır. Değiştirici üzerinden gitmek (`s.setSaniye(75);`) derlenir ve çalışır; ama 0–59 dışındaki değer reddedilir, alan 0'da kalır.

İki yol arasındaki fark, kuralın nerede denetlendiğidir. Birinci yolda denetimi derleyici yapar ve erişimi tamamen yasaklar. İkinci yolda erişime izin verilir, ama değer sınıfın kendi kuralından geçer. Sarmalama ikisini birlikte kullanır: alana giden doğrudan yol kapalı, metoda giden denetimli yol açıktır.

4.2.3.4 private sınıf düzeyindedir

`private` sözcüğü "bu nesneye özel" anlamına gelmez. Erişim denetimi nesneye göre değil, **sınıfa göre** yapılır. Bir sınıfın kodu, o sınıftan oluşturulmuş *her* nesnenin `private` alanlarını okuyabilir.

```
class Kesir {
    private int pay, payda;
    Kesir(int pay, int payda) {
        this.pay = pay;
        this.payda = payda;
    }
    Kesir carp(Kesir diger) {
        // diger'in private alanları: izinli
        return new Kesir(pay * diger.pay,
                         payda * diger.payda);
    }
    @Override public String toString() {
        return pay + "/" + payda;
    }
}
// main içinde:
Kesir k = new Kesir(2, 3).carp(new Kesir(3, 4));
System.out.println(k);
```

```
6/12
```

`carp` metodu kendi alanlarını (`pay` , `payda`) ve parametre olarak gelen `diger` nesnesinin alanlarını (`diger.pay` , `diger.payda`) okur. İkinci nesne başka bir nesnedir, ama aynı sınıftandır; bu yüzden erişim serbesttir. 2/3 ile 3/4'ün çarpımı 6/12 bulunur. Kesir sadeleştirilmez, çünkü `carp` yalnızca pay ve paydaları çarpar.

`carp` , `equals` , `compareTo` gibi iki nesneyi birlikte işleyen metotlar bu sayede alan okumak için ayrı bir metoda gerek duymaz. Aynı satır başka bir sınıfta yazıldığında ise derlenmez:

```
Dis.java:3: error: pay has private access in Kesir
        int p = new Kesir(1, 2).pay;
                                ^
1 error
```

Sarmalamanın sınırı sınıfın kaynak kodudur. Sınıfın içi tek bir yazarın sorumluluğundadır ve o yazar sınıfın bütün kurallarını bilir. Korunan, sınıfın dışındaki kullanıcılardır.

4.2.4 Erişimci ve değiştirici metotlar

Alanlar `private` olunca onlara ulaşmanın tek yolu sınıfın metotlarıdır. Bu metotların en yaygın iki türü vardır: alanın değerini okuyan ve alanı denetimli biçimde değiştiren metotlar.

> **Tanım 4.2.7**
>
> **Erişimci** (*accessor*, *getter*), bir alanın değerini döndüren metottur. **Değiştirici** (*mutator*, *setter*), bir alana yeni değeri önce denetleyip geçerliyse atayan metottur. Java geleneğinde adları `get` ve `set` ile başlar.

4.2.4.1 DijitalSaat sınıfı

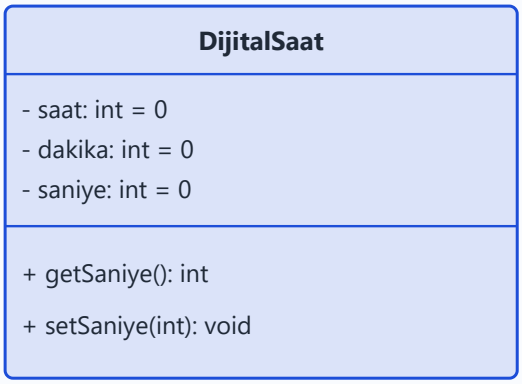
Sunumun ana örneği, saat, dakika ve saniyeyi tutan bir dijital saattir. İlk sürüm yalnızca saniye için erişimci ve değiştirici içerir:

```
public class DijitalSaat {
    private int saat = 0;
    private int dakika = 0;
    private int saniye = 0;

    public int getSaniye() {
        return this.saniye;
    }

    public void setSaniye(int saniye) {
        if (saniye >= 0 && saniye < 60) {
            this.saniye = saniye;
        }
    }
}
```

`getSaniye()` alanın değerini döndürür; hiçbir şeyi değiştirmez. `setSaniye(int saniye)` önce gelen değeri denetler. Değer 0 ile 59 arasındaysa atama yapılır, değilse metot hiçbir şey yapmadan biter. Bu değiştirici sessizce yok sayma yaklaşımını seçmiştir. Sınıfın UML diyagramı Şekil 4.2.8'de verilmiştir.



Şekil 4.2.8: `DijitalSaat` sınıfının UML diyagramı. Üstte sınıfın adı, ortada üç `private` alan (`-`) ve ilk değerleri, altta iki `public` metot (`+`) ve dönüş tipleri.

4.2.4.2 Değiştiricinin korumasını izlemek

Değiştiricinin geçersiz değeri nasıl durdurduğunu görmek için aşağıdaki program adım adım izlenir:

```
public class DijitalSaat {
  private int saniye = 0;
  public int getSaniye() {
    return this.saniye;
  }
  public void setSaniye(int saniye) {
    if (saniye >= 0 && saniye < 60) {
      this.saniye = saniye;
    }
  }
}
// SaatTest.main içinde:
DijitalSaat s = new DijitalSaat();
s.setSaniye(45);
s.setSaniye(75);
System.out.println(s.getSaniye());
```

```
45
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	13: `DijitalSaat s = new DijitalSaat();`	main()	s=→DijitalSaat	—
2	7: `if (saniye >= 0 && saniye < 60) {`	main() → s.setSaniye(45)	this=→nesne, saniye=45	—
3	8: `this.saniye = saniye;`	main() → s.setSaniye(45)	this=→nesne, saniye=45	—
4	7: `if (saniye >= 0 && saniye < 60) {`	main() → s.setSaniye(75)	this=→nesne, saniye=75	—
5	4: `return this.saniye;`	main() → s.getSaniye()	this=→nesne, dönüş=45	—

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
6	16: `System.out.println(s.getSaniye());`	main()	s=→DijitalSaat	`45`

Tablo 4.2.6: `setSaniye` çağrılarının izlenmesi. Satır numaraları: 1–11 sınıf, 13–16 `main` içindeki çağrılar.

Birinci adımda nesne oluşturulur ve `saniye` alanı 0 ile başlar. İkinci adımda `setSaniye(45)` çağrılır. Yeni çerçevede iki şey vardır: `saniye` parametresi 45'tir ve `this` metodu çağıran nesneyi gösterir. Koşul doğru çıkar. Üçüncü adımda `this.saniye = saniye;` çalışır. Soldaki `this.saniye` nesnenin alanıdır, sağdaki `saniye` parametredir; alan 45 olur.

Dördüncü adım korumanın kendisidir. `setSaniye(75)` çağrısında koşul yanlış çıkar ve atama satırı atlanır. Geçersiz değer nesneye giremez. Şekil 4.2.9 bu iki çağrıyı yan yana gösterir. Beşinci adımda `getSaniye()` alanın değerini döndürür: 45. Son adımda ekrana 45 yazılır. Nesnenin durumu baştan sona geçerli kaldı.



Şekil 4.2.9: Değiştiricinin iki çağrısı. Solda `setSaniye(45)` : koşul doğru, atama yapılacak. Sağda `setSaniye(75)` : koşul yanlış; heap'teki nesnenin `saniye` alanı 45'te kalır.

Erişimci ve değiştiricinin maliyeti sabittir: bir karşılaştırma, bir atama ya da bir okuma. Java sanal makinesi bu kadar küçük metotları çalışma sırasında çoğu zaman çağrı yerine yerleştirir; doğrudan alan erişimiyle aralarında ölçülebilir bir hız farkı kalmaz. Sarmalama, programı yavaşlatan bir bedel olarak görülmemelidir.

4.2.4.3 this ve gölgeleme

Değiştiricilerde parametrenin adı genellikle alanın adıyla aynı seçilir: `setSaniye(int saniye)`. Bu seçim okunabilirliği artırır, ama bir kuralın bilinmesini gerektirir.

Tanım 4.2.8

Gölgeleme (*shadowing*), iç kapsamdaki bir adın dış kapsamdaki aynı adı gizlemesidir. Parametrenin adı alanla aynıysa, metodun içinde o ad parametreyi gösterir; alan gölgede kalır. `this`, metodu çağıran nesnenin kendisine referanstır ve `this.saniye` her zaman alanı gösterir.

Doğru yazım `this.saniye = saniye;`'dir. `this` unutulduğunda ne olacağını `Kutu` sınıfı gösterir:

```
public class Kutu {
    private int en;
    public int getEn() {
        return en;
    }
    public void setEn(int en) {
        en = en;
    }
}
```

`setEn` metodunda `en = en;` satırının iki tarafı da parametredir. Parametre kendisine atanır, alan hiç değişmez. `k.setEn(5)` çağrısından sonra `k.getEn()` şunu döndürür:

```
0
```

Alan, `int` alanların varsayılan değeri olan 0'da kalmıştır. Bu hatanın en tehlikeli yanı derleyicinin hata vermemesidir. `en = en;` geçerli bir Java deyimidir. Hata ancak program çalıştırılıp çıktıya bakıldığında fark edilir. Bu yüzden değiştiricilerde `this.` yazma alışkanlığı önemlidir. Bazı geliştirme ortamları bu satırı "kendine atama" uyarısıyla işaretler, ama `javac` varsayılan ayarlarıyla sessiz kalır.

Sık yapılan hata

`final` alanda `this`'i unutmak. Alan `private final String ad;` ise ve yapıcıda `ad = ad;` yazılırsa sonuç farklıdır. Atama yine parametreye yapılır; alana hiçbir değer verilmemiş olur. `final` alanın yapıcıda mutlaka atanması gerektiği için derleyici bu kez yapıcının sonunu gösteren `variable ad might not have been initialized` hatasını verir. `final` olmayan alanda ise hiçbir uyarı çıkmaz.

4.2.4.4 Erişim metotları ve adlandırma

Saatin bütün alanları için erişimci ve değiştirici yazıldığında sınıf şu hâli alır:

```
public class DijitalSaat {
  private int saat = 0;      // 0..23
  private int dakika = 0;    // 0..59
  private int saniye = 0;    // 0..59

  public int getSaat() { return saat; }
  public int getDakika() { return dakika; }
  public int getSaniye() { return saniye; }

  public void setSaat(int saat) {
    if (saat >= 0 && saat < 24) this.saat = saat;
  }
  public void setDakika(int dakika) {
    if (dakika >= 0 && dakika < 60) this.dakika = dakika;
  }
  public void setSaniye(int saniye) {
    if (saniye >= 0 && saniye < 60) this.saniye = saniye;
  }

  public void goster() {
    System.out.printf("%02d:%02d:%02d%n",
        saat, dakika, saniye);
  }
}
```

```
DijitalSaat s = new DijitalSaat();
s.setSaat(9);
s.setDakika(5);
s.setSaniye(30);
s.goster();
s.setSaat(24);     // reddedilir
s.setDakika(-3);   // reddedilir
s.goster();
```

```
09:05:30
09:05:30
```

İlk üç çağrı geçerlidir ve saat 09:05:30 olur. `setSaat(24)` çağrısı reddedilir, çünkü geçerli saatler 0–23'tür. `setDakika(-3)` de reddedilir. İkinci `goster()` çağrısı bu yüzden aynı değeri yazar. `printf` içindeki `%02d` biçimi sayıyı en az iki basamağa, gerekirse başına sıfır ekleyerek yazar: 9 değeri `09` olarak görünür.

Erişim metotlarının adları bir geleneğe uyar. `dakika` alanının erişimcisi `getDakika()`, değiştiricisi `setDakika(int)` adını alır. Alan adının ilk harfi büyütülüp başına `get` ya da `set` eklenir. `boolean` bir alanın erişimcisi ise `is` ile başlar: `aktif` alanı için `isAktif()`. Bu gelenek yalnızca okunabilirlik için değildir. Birçok Java kütüphanesi ve geliştirme aracı, bir sınıfın özelliklerini bu adlardan çıkarır.

4.2.4.5 Neden public alan değil?

Bir alanı `private` yapıp iki metot yazmak, onu `public` yapmaktan daha zahmetlidir. Bu zahmetin karşılığı Tablo 4.2.7'de özetlenmiştir.

```
public class Saat {
    public int saniye;
}
// başka bir sınıfta:
s.saniye = -5;   // kabul edilir!
```

```
public class Saat {
    private int saniye;
    public void setSaniye(int s) {
        if (s >= 0 && s < 60) saniye = s;
    }
}
```

Soru	`public` alan	`private` alan + metot
Kim, hangi değeri yazabilir?	Her yerden, her değer	Yalnızca metodun kabul ettiği değer
Kural (0–59) nerede denetlenir?	Her kullanıcıya bırakılır; biri unutursa nesne bozulur	Tek bir yerde, sınıfın içinde
Salt okunur alan mümkün mü?	Hayır (`final` dışında)	Evet; yalnızca erişimci yazılır
Alanın adı ya da tipi değişirse?	Onu kullanan bütün sınıflar değişir	Metotların imzası aynı kalabilir

Tablo 4.2.7: `public` alan ile `private` alan ve metot yaklaşımlarının karşılaştırılması.

Tablonun son satırı en uzun vadeli kazancı gösterir: iç yapı değiştiğinde bile metotların imzası korunabilir. Saatin iki alanı tek bir `toplamSaniye` alanına indirildiğinde de kullanan kod hiç değişmez.

Sunumdaki etkileşimli deneme, bu karşılaştırmayı bir `DijitalSaat` nesnesi üzerinde yaptırır. Nesne 09:05:30 durumundadır. Saniyeye 75 yazılmak istendiğinde iki yol izlenebilir. Değiştirici yolu (`s.setSaniye(75)`) değeri reddeder ve saat 09:05:30'da kalır. Alan `public` olsaydı (`s.saniye = 75`) değer doğrudan yazılır ve saat 09:05:75 gibi anlamsız bir durumu gösterir. Geçerli aralıklar saat için 0–23, dakika ve saniye için 0–59'dur.

Yalnızca veri taşıyan ve hiç değişmeyecek sınıflar için Java 16'dan beri daha kısa bir yol vardır: **kayıt** (*record*). `record Nokta(int x, int y) { }` satırı, alanları otomatik olarak `private final` olan bir sınıf tanımlar. Erişimcilerin adı `get` ile başlamaz, alanın adını taşır: `x()` ve `y()`. Derleyici `toString`, `equals` ve `hashCode` metotlarını da kendisi yazar:

```
public class KayitTest {
  public static void main(String[] args) {
    Nokta n = new Nokta(3, 4);
    System.out.println(n.x() + " " + n.y());
    System.out.println(n);
    System.out.println(n.equals(new Nokta(3, 4)));
  }
}
```

```
3 4
Nokta[x=3, y=4]
true
```

Kaydın alanına dışarıdan yazma denemesi (`n.x = 5;`) `x has private access in Nokta` hatası verir. Alan hem `private` hem `final` olduğu için kayıt, değiştirilemez sınıfın hazır bir örneğidir.

4.2.5 Örnek: banka hesabı

Şimdiye kadarki parçalar tek bir sınıfta birleştirilebilir. Banka hesabı bunun için iyi bir örnektir: verisi değerlidir, kuralları açıktır ve kuralların bozulması kolayca fark edilir. Bakiye hiçbir zaman eksiye düşmemeli, para yalnızca belirli işlemlerle girip çıkmalıdır. `BankaHesabi` sınıfı dört parça hâlinde incelenir.

4.2.5.1 Alanlar ve yapıcı

```
public class BankaHesabi {
  private int hesapNumarasi;  // Hesap numarası
  private double bakiye;      // Hesap bakiyesi
  private String hesapTuru;   // "Vadesiz" / "Vadeli"

  public BankaHesabi(int hesapNumarasi,
      double bakiye, String hesapTuru) {
    this.hesapNumarasi = hesapNumarasi;
    this.bakiye = bakiye;
    this.hesapTuru = hesapTuru;
  }

  public int getHesapNumarasi() {
    return hesapNumarasi;
  }
  public double getBakiye() {
    return bakiye;
  }
  public String getHesapTuru() {
    return hesapTuru;
  }
  // ... para yatırma ve çekme metotları
```

Sınıfın üç alanı da `private`'tır: hesap numarası, bakiye ve hesap türü. Kimse hesap numarasını ya da bakiyeyi doğrudan değiştiremez. **Yapıcı** (*constructor*) nesneyi ilk değerleriyle kurar. Parametrelerin adları alanlarla aynı seçildiği için atamalar `this.bakiye = bakiye;` kalıbıyla yazılır.

Sınıfta yalnızca erişimciler vardır, değiştirici yoktur. `setBakiye` metodu bilinçli olarak yazılmamıştır. Bakiye yalnızca para yatırma, çekme, faiz ve transfer işlemleriyle değişir. Böylece bakiyeyi değiştiren her yol, kendi kuralını uygulayan bir metottan geçer. Hesap numarası ve hesap türü için de değiştirici yoktur; bu iki alan hesap açıldıktan sonra değişmez.

4.2.5.2 Yatırma ve çekme

```
// ...
// Para yatırma işlemi
public void paraYatir(double miktar) {
  if (miktar > 0) {
    bakiye += miktar;
    System.out.println(miktar
        + " TL yatırıldı. Yeni bakiye: " + bakiye);
  } else {
    System.out.println("Yatırılan miktar geçersiz.");
  }
}

// Para çekme işlemi
public void paraCek(double miktar) {
  if (miktar <= bakiye && miktar > 0) {
    bakiye -= miktar;
    System.out.println(miktar
        + " TL çekildi. Kalan bakiye: " + bakiye);
  } else if (miktar > bakiye) {
    System.out.println("Yetersiz bakiye.");
  } else {
    System.out.println("Çekilen miktar geçersiz.");
  }
}
```

Her işlem önce ön koşulu denetler. Para yatırmada koşul `miktar > 0`'dır. Para çekmede iki koşul vardır: `miktar > 0` ve `miktar ≤ bakiye`. Koşul bozuksa bakiye değişmez, yalnızca bir mesaj yazılır. Bu sınıf ön koşul bozulduğunda "sessizce yok saymak" ile "bildirmek" arasında bir yol seçmiştir: istisna fırlatmaz, ama ekrana neyin yanlış gittiğini yazar.

`paraCek`'teki `if` – `else if` – `else` zincirinin sırası önemlidir. Önce işlemin geçerli olduğu durum denetlenir. Geçerli değilse ilk olası neden, miktarın bakiyeyi aşmasıdır. Geriye kalan tek durum miktarın sıfır ya da negatif olmasıdır. Bu düzen sayesinde her geçersiz çağrı tam olarak bir mesaj alır.

İki metot birlikte sınıfın değişmezini korur: bakiye hiçbir yoldan eksiye düşemez. Yatırma bakiyeyi yalnızca artırır; çekme ise ancak yeterli bakiye varsa azaltır. Değişmez nesne kurulurken doğruysa, bu iki metottan sonra da doğru kalır.

4.2.5.3 Faiz

```
  // ...
  // Faiz ekleme (yalnızca vadeli hesap)
  public void faizEkle(double oran) {
    if (hesapTuru.equals("Vadeli") && oran > 0) {
      double faiz = bakiye * oran / 100;
      bakiye += faiz;
      System.out.println("Faiz ekleniyor: " + faiz
          + " TL. Yeni bakiye: " + bakiye);
    } else if (!hesapTuru.equals("Vadeli")) {
      System.out.println(
          "Faiz yalnızca vadeli hesaplarda uygulanabilir.");
    } else {
      System.out.println("Geçersiz faiz oranı.");
    }
  }
```

Faiz yalnızca vadeli hesapta ve pozitif oranla eklenir. Hesap türü bir metin olduğu için karşılaştırma `==` ile değil, `equals` ile yapılır. `==` iki referansın aynı nesneyi gösterip göstermediğine bakar; `equals` ise metinlerin içeriğini karşılaştırır. İki ayrı yoldan oluşturulmuş `"Vadeli"` metinleri `==` ile farklı çıkabilir.

`faiz` bir **yerel değişkendir**. Yalnızca metot çalışırken vardır ve nesnenin parçası değildir. Faiz tutarı bir kez hesaplanır, bakiyeye eklenir ve metot bitince yok olur. Nesnenin durumunu oluşturan, yalnızca üç alandır.

4.2.5.4 Transfer

```
  // ...
  // Başka bir hesaba transfer
  public void transfer(BankaHesabi hedefHesap,
                       double miktar) {
    if (miktar <= bakiye && miktar > 0) {
      bakiye -= miktar;
      hedefHesap.paraYatir(miktar);
      System.out.println(miktar
          + " TL transfer edildi. Yeni bakiye: " + bakiye);
    } else {
      System.out.println(
          "Yetersiz bakiye veya geçersiz miktar.");
    }
  }
}
```

`transfer` metodu, aynı sınıftan başka bir nesnenin metodunu çağırır: `hedefHesap.paraYatir(miktar)`. Kendi bakiyesinden düşer, hedef hesaba ise yatırma metoduyla ekler (Şekil 4.2.10).



Şekil 4.2.10: Transferin iki yarısı. `hesap1` kendi bakiyesinden 300 düşer. Ardından `hesap2` nesnesinin `paraYatir` metodunu çağırır; ekleme işini `hesap2` kendi kuralıyla yapar.

Hedef hesabın `bakiye` alanına doğrudan yazmak da mümkündü: `hedefHesap.bakiye += miktar;`. İki nesne aynı sınıftandır ve `private` sınıf düzeyindedir. Ama metot çağırmak kuralların tek yerde kalmasını sağlar. Yarın yatırma işlemine bir kural eklense, örneğin her yatırma kaydedilse, transfer de bu kuraldan otomatik olarak geçer. Alanı doğrudan değiştiren kod ise yeni kuralı atlardı.

4.2.5.5 İki hesabı adım adım izlemek

Sınıfın bütün parçaları aşağıdaki `main` metoduyla birlikte çalıştırılır:

```
// main içinde:
BankaHesabi hesap1 =
    new BankaHesabi(123456, 2000.0, "Vadeli");
BankaHesabi hesap2 =
    new BankaHesabi(789012, 1500.0, "Vadesiz");
hesap1.paraYatir(500.0);
hesap2.paraCek(200.0);
hesap1.faizEkle(5.0);
hesap1.transfer(hesap2, 300.0);
System.out.println(hesap1.getBakiye());
System.out.println(hesap2.getBakiye());
```

```
500.0 TL yatırıldı. Yeni bakiye: 2500.0
200.0 TL çekildi. Kalan bakiye: 1300.0
Faiz ekleniyor: 125.0 TL. Yeni bakiye: 2625.0
300.0 TL yatırıldı. Yeni bakiye: 1600.0
300.0 TL transfer edildi. Yeni bakiye: 2325.0
2325.0
1600.0
```

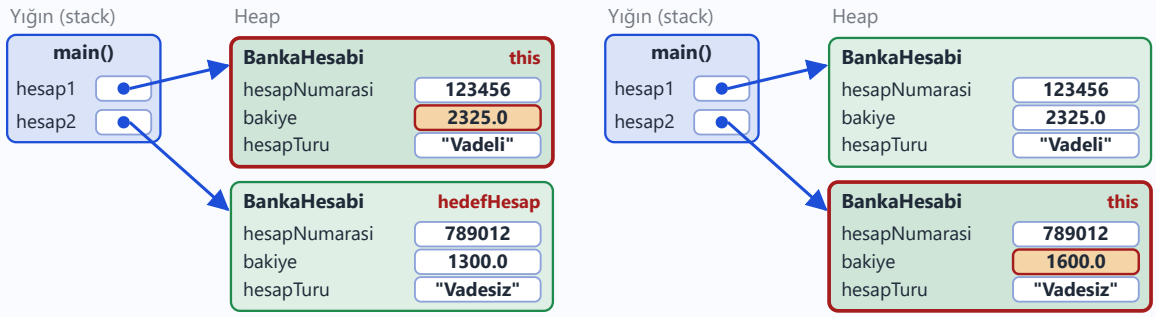
Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	2: `BankaHesabi hesap1 =` 3: `new BankaHesabi(123456, 2000.0, "Vadeli");`	main()	hesap1=→BankaHesabi	—
2	4: `BankaHesabi hesap2 =` 5: `new BankaHesabi(789012, 1500.0, "Vadesiz");`	main()	hesap1=→BankaHesabi, hesap2=→BankaHesabi	—
3	6: `hesap1.paraYatir(500.0);`	main() → hesap1.paraYatir(500.0)	miktar=500.0	`500.0 TL yatırıldı. Yeni bakiye: 2500.0`
4	7: `hesap2.paraCek(200.0);`	main() → hesap2.paraCek(200.0)	miktar=200.0	`200.0 TL çekildi. Kalan bakiye: 1300.0`
5	8: `hesap1.faizEkle(5.0);`	main() → hesap1.faizEkle(5.0)	oran=5.0, faiz=125.0	`Faiz ekleniyor: 125.0 TL. Yeni bakiye: 2625.0`
6	9: `hesap1.transfer(hesap2, 300.0);`	main() → hesap1.transfer(hesap2, 300.0)	miktar=300.0, hedefHesap=→hesap2	—
7	9: `hesap1.transfer(hesap2, 300.0);`	main() → hesap1.transfer(hesap2, 300.0) → hesap2.paraYatir(300.0)	miktar=300.0	`300.0 TL yatırıldı. Yeni bakiye: 1600.0`
8	9: `hesap1.transfer(hesap2, 300.0);`	main() → hesap1.transfer(hesap2, 300.0)	miktar=300.0	`300.0 TL transfer edildi. Yeni bakiye: 2325.0`
9	10: `System.out.println(hesap1.getBakiye());` 11: `System.out.println(hesap2.getBakiye());`	main()	hesap1=→BankaHesabi, hesap2=→BankaHesabi	`2325.0 ⏎ 1600.0`

Tablo 4.2.8: İki banka hesabının izlenmesi. Satır numaraları yukarıdaki `main` parçasına göredir (1. satır yorum).

İlk iki adımda iki bağımsız nesne kurulur. Her nesnenin kendi `bakiye` alanı vardır: `hesap1`'inki 2000.0, `hesap2`'ninki 1500.0. Üçüncü adımda `hesap1.paraYatir(500.0)` çağrılır. Bu çağrıda `this`, `hesap1`'i gösterir; `miktar > 0` doğrudur ve bakiye 2500.0 olur. Dördüncü adımda bu kez `this`, `hesap2`'dir. 200 ≤ 1500 olduğu için çekim yapılır ve bakiye 1300.0'a iner. `hesap1` bu işlemden etkilenmez.

Beşinci adımda faiz hesaplanır. Hesap vadelidir ve oran pozitiftir: faiz = 2500.0 × 5 / 100 = 125.0. Bakiye 2625.0 olur. Altıncı adımda transfer başlar. Koşul doğrudur; `hesap1` önce kendi bakiyesinden düşer: 2625.0 − 300.0 = 2325.0. Yedinci adımda `hedefHesap.paraYatir(miktar)` çağrılır. Yığında

artık üç çerçeve vardır: `main`, `transfer` ve `paraYatir`. Bu en üstteki çerçevede `this`, `hesap2`'yi gösterir ve onun bakiyesi 1600.0 olur (Şekil 4.2.11).



Şekil 4.2.11: Transferin içinden iki an. Solda `transfer` çalışıyor: `this` birinci hesabı, `hedefHesap` ikinci hesabı gösteriyor; birinci hesabın bakiyesi 2325.0'a inmiş. Sağda `paraYatir` çalışıyor: bu kez `this` ikinci hesap ve bakiyesi 1600.0.

Sekizinci adımda `paraYatir` biter ve `transfer` kaldığı yerden devam eder. Çıktıdaki sıraya dikkat: önce "300.0 TL yatırıldı" (iç çağrının mesajı), sonra "300.0 TL transfer edildi" (dış metodun mesajı) yazılır. Son adımda erişimciler iki bakiyeyi okur. Toplam para korunmuştur: 2325 + 1600 = 3925 = 2625 + 1300. Transfer para üretmez, yalnızca yer değiştirir.

Sınıfın her metodu sabit sayıda işlem yapar; döngü yoktur. Bu yüzden her çağrının zaman maliyeti O(1)'dir. Bellekte ise her hesap nesnesi yalnızca üç alan tutar; nesne sayısı arttıkça bellek doğrusal olarak artar.

Çıktıdaki sayılar `double` değerlerin Java'daki varsayılan yazımıdır: 500.0, 2625.0. Bu örnekte bütün tutarlar tam olduğu için sorun çıkmaz. Ama `double` ondalık kesirleri her zaman tam tutamaz. Aynı sınıfla 0.1 TL bakiyeli bir hesaba 0.2 TL yatırıldığında JDK 21 `Yeni bakiye: 0.30000000000000004` yazar. Gerçek bankacılık yazılımları bu yüzden parayı kuruş cinsinden bir tam sayıyla ya da `BigDecimal` sınıfıyla tutar.

4.2.5.6 Yapıcı da doğrulamalı

`BankaHesabi`'nın işlem metotları bakiyeyi korur. Ama yapıcısı gelen değeri hiç denetlemeden atar. Bu boşluk iki yoldan kötüye kullanılabilir. `new BankaHesabi(1, -500.0, "Vadesiz")` eksi bakiyeli bir hesap kurar; değişmez daha nesne doğarken bozulmuştur. `"vadeli"` (küçük harfle) yazılırsa hesap hiçbir zaman faiz alamaz ve kimse bunu fark etmez. JDK 21 ile denendiğinde iki nesne de sorunsuz kurulur; küçük harfli hesaba `faizEkle(5.0)` çağrıldığında "Faiz yalnızca vadeli hesaplarda uygulanabilir." mesajı çıkar.

Sınıf değişmezi nesnenin doğduğu anda sağlanmalıdır. Yapıcı da bir ön koşul denetleyicisidir:

```
public BankaHesabi(int hesapNumarasi,
        double bakiye, String hesapTuru) {
    if (bakiye < 0)
        throw new IllegalArgumentException(
            "Başlangıç bakiyesi negatif olamaz");
    if (!hesapTuru.equals("Vadeli")
            && !hesapTuru.equals("Vadesiz"))
        throw new IllegalArgumentException(
            "Bilinmeyen hesap türü: " + hesapTuru);
    this.hesapNumarasi = hesapNumarasi;
    this.bakiye = bakiye;
    this.hesapTuru = hesapTuru;
}
```

Bu yapıcıyla `new BankaHesabi(1, -500.0, "Vadesiz")` çağrısı `Başlangıç bakiyesi negatif olamaz`, `new BankaHesabi(2, 1000.0, "vadeli")` çağrısı ise `Bilinmeyen hesap türü: vadeli` iletisiyle istisna fırlatır. Hesap türü olarak `null` verilirse `hesapTuru.equals(...)` çağrısı `NullPointerException` fırlatır; o da nesnenin kurulmasını engeller, ama iletisi daha az açıklayıcıdır.

Yapıcı istisna fırlatırsa `new` ifadesi hiçbir referans üretmez. Geçersiz nesne kullanıcının eline hiç geçmez. Bu davranışı küçük bir `Sicaklik` sınıfıyla izlemek mümkündür:

```
class Sicaklik {
    private final double kelvin;
    public Sicaklik(double kelvin) {
        if (kelvin < 0)
            throw new IllegalArgumentException(
                "Kelvin negatif olamaz");
        this.kelvin = kelvin;
    }
    public double getKelvin() { return kelvin; }
}
// main içinde:
Sicaklik a = new Sicaklik(300);
Sicaklik b = null;
try {
    b = new Sicaklik(-5);
} catch (IllegalArgumentException e) {
    System.out.println("Reddedildi: "
        + e.getMessage());
}
System.out.println(a.getKelvin() + " " + b);
```

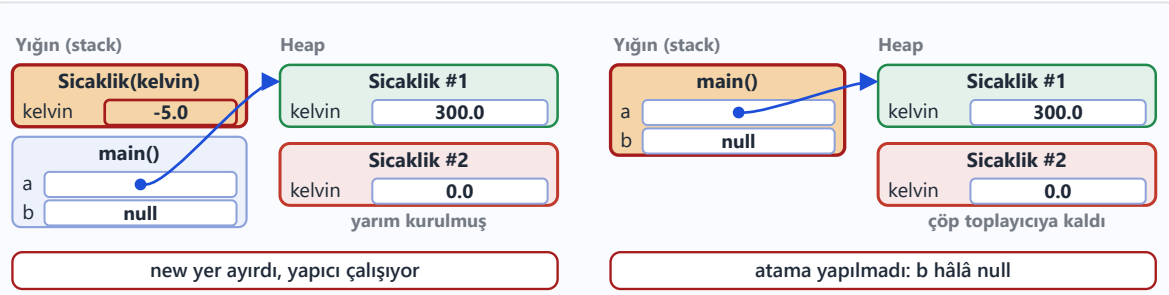
```
Reddedildi: Kelvin negatif olamaz
300.0 null
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	12: `Sicaklik a = new Sicaklik(300);`	main()	a=→#1	—
2	13: `Sicaklik b = null;`	main()	a=→#1, b=null	—
3	15: `b = new Sicaklik(-5);` 3: `public Sicaklik(double kelvin) {`	main() → Sicaklik(-5.0)	a=→#1, b=null, kelvin=-5.0	—
4	4: `if (kelvin < 0)` 5: `throw new IllegalArgumentException(`	main() → Sicaklik(-5.0)	a=→#1, b=null, kelvin=-5.0	—
5	16: `} catch (IllegalArgumentException e) {` 17: `System.out.println("Reddedildi: "` 18: `+ e.getMessage());`	main()	a=→#1, b=null	`Reddedildi: Kelvin negatif olamaz`
6	20: `System.out.println(a.getKelvin() + " " + b);`	main()	a=→#1, b=null	`300.0 null`

Tablo 4.2.9: Yapıcıda doğrulamanın izlenmesi. Satır numaraları: 1–10 `Sicaklik` sınıfı, 12–20 `main` içindeki kod.

Birinci adımda 300 geçerlidir; koşul yanlış çıkar, `kelvin` atanır ve `a` yeni nesneyi gösterir. İkinci adımda `b` açıkça `null` ile başlatılır. Bu atama gereklidir: `b` `catch` bloğundan sonra okunacaktır. Java, değer verilmemiş olabilecek bir yerel değişkenin okunmasına izin vermez.

Üçüncü adımda `new` ikinci nesneye yer ayırır ve yapıcıyı çağırır. Yer ayrılmış nesnenin `kelvin` alanı varsayılan değer olan 0.0'dır; parametre ise −5.0'dır. Dördüncü adımda ön koşul bozuk bulunur ve `throw` yapıcıyı keser. Atama satırı hiç çalışmaz. Şekil 4.2.12 bu iki anı gösterir.



Şekil 4.2.12: Yapıcıda doğrulama. Solda `new` ikinci nesneye yer ayırmış, yapıcı −5.0 parametresiyle çalışıyor; nesne yarım kurulmuştur. Sağda istisna yakalanmış, `b = ...` ataması hiç yapılmamış; `b` hâlâ `null` ve yarım nesneyi gösteren hiçbir referans yok.

Beşinci adımda kontrol `catch` bloğuna geçer ve `Reddedildi: Kelvin negatif olamaz` yazılır. `b = new Sicaklik(-5);` ataması yarıda kaldığı için `b` değişmemiştir. Yarım kurulmuş nesneye ulaşan hiçbir referans yoktur; çöp toplayıcı onu bir süre sonra temizler. Son adımda `300.0 null` yazılır.

Programda yalnızca geçerli `Sicaklik` nesneleri kullanılabilir: değişmez (`kelvin ≥ 0`) her nesnede doğrudur.

Yapıcı, nesnenin kurulduğu tek kapıdır. Kurallar orada bir kez uygulanırsa sınıfın öteki metotları gelen nesnenin geçerli olduğuna güvenebilir. Değiştiricisi olan sınıflarda yapıcının, aynı denetimi tekrar yazmak yerine değiştiriciyi çağırması da yaygın bir yoldur. Böylece kural tek bir metotta kalır.

Sık yapılan hata

Değiştiricileri doğrulayıp yapıcıyı unutmak. `setSaniye` 75'i reddeder, ama yapıcı `this.saniye = saniye;` yazıyorsa `new DijitalSaat(75)` geçersiz bir saat kurar. Nesneye değer giren her yol, yapıcı dahil, aynı kuraldan geçmelidir.

4.2.6 Sınıf tasarımı

Sarmalanmış bir sınıfın iki yüzü vardır: kullanıcının gördüğü dış yüz ve yalnızca yazarın bildiği iç yapı. Bu ayrım, sınıfı tasarlarken verilen kararların çoğunu yönlendirir. Bu kesim ayrımı kavram olarak tanımlar, Java'nın `interface` yapısıyla somutlaştırır ve bir sınıfın iç yapısını kullananları bozmadan nasıl değiştirdiğini gösterir.

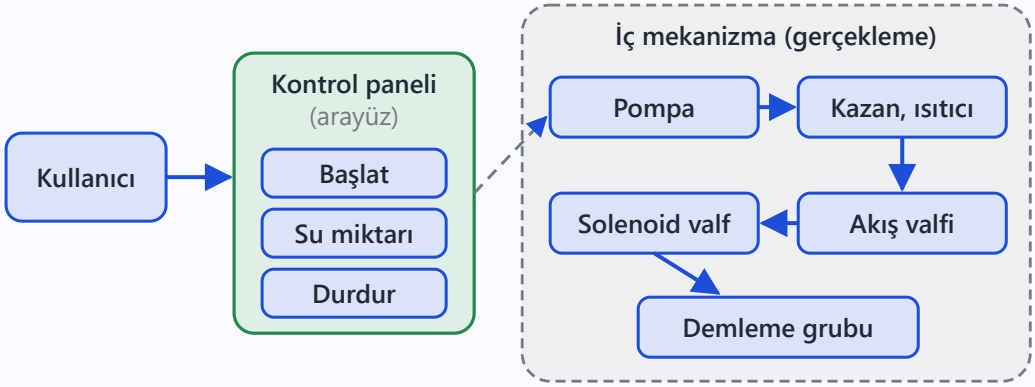
4.2.6.1 Arayüz ve gerçekleme

Tanım 4.2.9

Bir sınıf tanımı iki kısma ayrılır. **Arayüz** (*interface*), sınıfın ne yaptığını ve nasıl kullanılacağını belirtir: `public` metotların adları, parametreleri, dönüş tipleri ve yorumları. **Gerçekleme** (*implementation*), sınıfın iç mekanik ayrıntılarını ve `private` alanlarını içerir; bunlar yalnızca sınıfın içinde önemlidir.

Gerçekleme değişse bile arayüz değişmeyebilir, çünkü özel alanlar sınıfın dışından gizlidir. Bazı kaynaklar bu ayrımı üç unsurla anlatır. **Tanımlama** (*specification*), yapıcı ve metotların imzalarıdır. **Temsil** (*representation*), veriyi tutan `private` alanlardır. **Gerçekleme**, yapıcı ve metotların gövdeleridir. Kullanıcı yalnızca tanımlamayı bilir. Temsil ve gerçekleme değişse de kullanıcının kodu bozulmaz.

Bir espresso makinesi bu ayrımı iyi gösterir (Şekil 4.2.13). Kullanıcı kontrol panelindeki “Başlat”, “Su miktarı” ve “Durdur” düğmelerini bilir. Pompa, kazan, valfler ve demleme grubu panelin arkasındadır. Üretici pompayı daha sessiz bir modelle değiştirse bile panel aynı kalır ve kullanıcı farkı yalnızca sesten anlar.



Şekil 4.2.13: Espresso makinesi benzetmesi. Kullanıcı yalnızca kontrol panelini (arayüz) kullanır; pompa, kazan ve ısıtıcı, akış ve solenoid valfleri ile demleme grubu iç mekanizmadır (gerçekleme).

4.2.6.2 interface ve implements

Java, arayüz fikrine bir dil yapısı ayırmıştır. `interface` anahtar sözcüğüyle tanımlanan bir tip, bir sınıfın ne yapacağını söyleyen sözleşmedir. Metotların yalnızca imzası vardır, gövdesi yoktur:

```
public interface Araba {
    // Sınıf arayüzü: NE yapılır?
    public void hareketEt();

    public void dur();
}
```

Arayüzdeki metotlar örtük olarak `public` ve `abstract`'tır; örnekteki `public` sözcükleri yazılmasa da anlam değişmez. Arabayı kullanan kişi yalnızca `hareketEt()` ve `dur()` metotlarını bilir. Arayüzü uygulayan sınıf ise işin nasıl yapılacağını yazar:

```
public class SporAraba implements Araba {
  // Sınıf gerçeklemesi: NASIL yapılır?
  private String marka;
  private int hiz;

  public SporAraba(String marka) {
    this.marka = marka;
  }

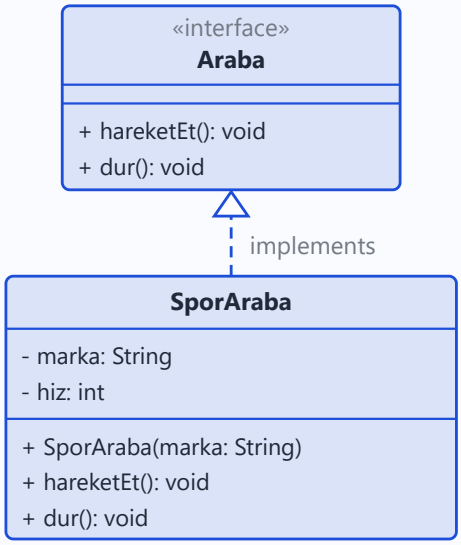
  public void hareketEt() {
    hiz = 100;
    System.out.println(marka + " hareket ediyor: "
        + hiz + " km/sa");
  }

  public void dur() {
    hiz = 0;
    System.out.println(marka + " durdu.");
  }
}
```

```
// main içinde:
Araba araba = new SporAraba("Rüzgar");
araba.hareketEt();
araba.dur();
```

```
Rüzgar hareket ediyor: 100 km/sa
Rüzgar durdu.
```

`implements Araba` ifadesi, `SporAraba`'nın `Araba` arayüzünü uyguladığını söyler. Bu bir sözdür ve derleyici sözün tutulup tutulmadığını denetler: sınıf, arayüzdeki her metodu yazmak zorundadır. Şekil 4.2.14 iki tip arasındaki ilişkiyi UML ile gösterir.



Şekil 4.2.14: UML gösteriminde `SporAraba` sınıfı `Araba` arayüzünü uygular. Kesik çizgi ve içi boş üçgen "uygular" ilişkisidir. Arayüzde yalnızca iki `public` metot, sınıfta ise ek olarak iki `private` alan ve bir yapıcı vardır.

Arayüz ile sınıf aynı pakette aynı adı taşıyamaz. Sınıfa da `Araba` adı verilmeye çalışılırsa derleyici `duplicate class: Araba` hatası verir; bu yüzden sınıfın adı `SporAraba`'dır. `main` metodu değişkenini `Araba` tipinde tanımlar: `Araba araba = new SporAraba("Rüzgar");`. Böylece bu kod yalnızca arayüze bağımlıdır. Yarın `SporAraba` yerine başka bir gerçekleme kullanılsa yalnızca `new` ifadesi değişir.

Bir metot yazılmazsa ne olur? `SporAraba` yalnızca `hareketEt()`'i yazıp `dur()`'u unuttuğunda JDK 21 şu iletiyi verir:

```
ArayuzEksik.java:5: error: SporAraba is not abstract and does not override abstract method
dur() in Araba
class SporAraba implements Araba {
^
1 error
```

İleti iki çıkış yolunu da söyler: sınıf ya eksik metodu yazar ya da kendisi `abstract` olarak işaretlenir. Burada önemli olan, arayüzün sarmalamadaki "dış yüz" fikrini derleyicinin denetlediği bir yapıya dönüştürmesidir.

4.2.6.3 Gerçeklemeyi değiştirmek

Sarmalamanın en güçlü kazancı, sınıfın iç yapısını kullananları bozmadan değiştirebilmektir. `DijitalSaat` bunu göstermek için iki sürümle yazılmıştır. Eski sürüm dakikayı ve saniyeyi iki ayrı alanda tutar:

```
public class DijitalSaat {
  private int dakika = 0;
  private int saniye = 0;
  public int getDakika() {
    return dakika;
  }
  public int getSaniye() {
    return saniye;
  }
  public void setDakika(int dakika) {
    if (dakika >= 0 && dakika < 60) {
      this.dakika = dakika;
    }
  }
  public void setSaniye(int saniye) {
    if (saniye >= 0 && saniye < 60) {
      this.saniye = saniye;
    }
  }
}
```

Yeni sürüm ise tek bir alan kullanır: gece yarısından beri geçen saniye sayısı. Dakika ve saniye artık saklanmaz, her istendiğinde bu sayıdan hesaplanır:

```
public class DijitalSaat {
  // gece yarısından beri geçen saniye
  private int toplamSaniye = 0;
  public int getDakika() {
    return toplamSaniye / 60 % 60;
  }
  public int getSaniye() {
    return toplamSaniye % 60;
  }
  public void setDakika(int dakika) {
    if (dakika >= 0 && dakika < 60) {
      int fark = dakika - getDakika();
      toplamSaniye += fark * 60;
    }
  }
  public void setSaniye(int saniye) {
    if (saniye >= 0 && saniye < 60) {
      toplamSaniye += saniye - getSaniye();
    }
  }
}
```

İki sürümün `public` metotları aynıdır: `getDakika()`, `getSaniye()`, `setDakika(int)`, `setSaniye(int)`. Kullanan kod hangisiyle çalıştığını bilemez. Yeni sürümdeki hesaplar biraz düşünmeyi gerektirir. `toplamSaniye / 60` toplam dakika sayısını verir; `% 60` bunun saat içindeki dakika kısmını ayırır. `toplamSaniye % 60` ise saniye kısmıdır. Değiştiriciler yeni değeri doğrudan yazamaz, çünkü öteki kısmı korumak zorundadır. Bunun yerine eski değerle yeni değer arasındaki farkı ekler.

Aşağıdaki kullanım kodu yeni sürümle adım adım izlenir:

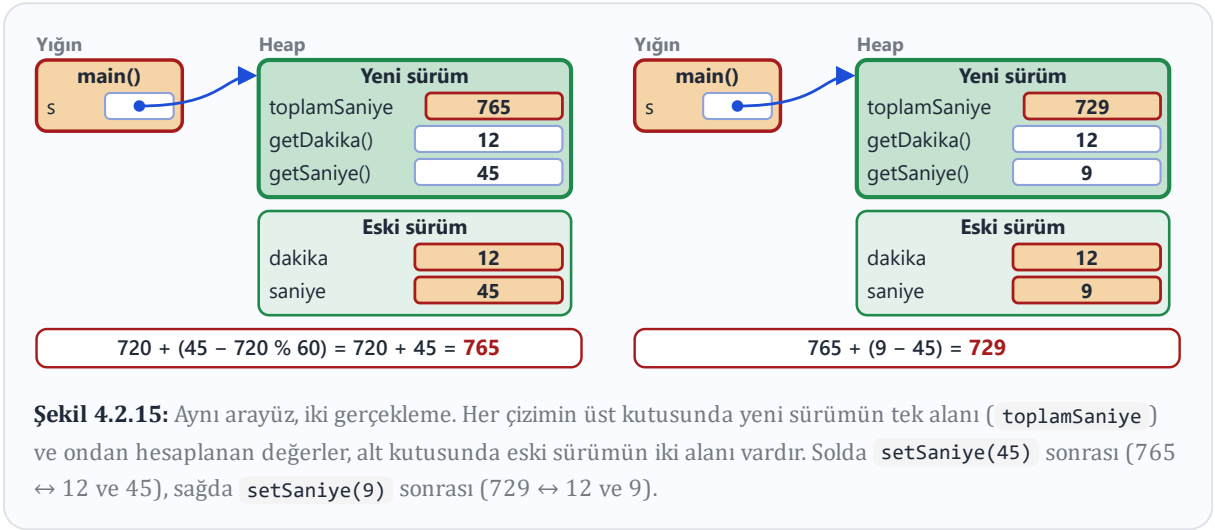
```
private int toplamSaniye = 0;  // tek alan
public int getSaniye() {
    return toplamSaniye % 60;
}
public void setSaniye(int saniye) {
    if (saniye >= 0 && saniye < 60)
        toplamSaniye += saniye - getSaniye();
}
// getDakika(): toplamSaniye / 60 % 60
DijitalSaat s = new DijitalSaat(); // main
s.setDakika(12);
s.setSaniye(45);
s.setSaniye(75);
s.setSaniye(9);
System.out.println(s.getDakika() + ":"
    + s.getSaniye());
```

Adım	Satır	Değişkenler	Yeni çıktı
1	10: `DijitalSaat s = new DijitalSaat(); // main`	toplamSaniye=0	—
2	11: `s.setDakika(12);`	toplamSaniye=720	—
3	12: `s.setSaniye(45);` 5: `public void setSaniye(int saniye) {` 6: `if (saniye >= 0 && saniye < 60)`	saniye=45, toplamSaniye=720	—
4	7: `toplamSaniye += saniye - getSaniye();` 2: `public int getSaniye() {` 3: `return toplamSaniye % 60;`	saniye=45, getSaniye()=0, toplamSaniye=765	—
5	13: `s.setSaniye(75);` 6: `if (saniye >= 0 && saniye < 60)`	saniye=75, toplamSaniye=765	—
6	14: `s.setSaniye(9);` 7: `toplamSaniye += saniye - getSaniye();`	saniye=9, getSaniye()=45, toplamSaniye=729	—
7	15: `System.out.println(s.getDakika() + ":"` 16: `+ s.getSaniye());`	getDakika()=12, getSaniye()=9	`12:9`

Tablo 4.2.10: Tek alanlı `DijitalSaat`'in izlenmesi. Satır numaraları yukarıdaki kısaltılmış koda göredir: 1–9 sınıftan parçalar, 10–16 `main`.

Başta `toplamSaniye` 0'dır; dışarıdan bakan için saat 00:00'dır. `setDakika(12)` çağrısında `getDakika()` 0 döndürür, fark 12 dakikadır ve 12 · 60 = 720 saniye eklenir. Eski sürüm aynı çağrıda yalnızca `dakika = 12` yazardı. `setSaniye(45)` çağrısında önce `getSaniye()` = 720 % 60 = 0 hesaplanır; fark 45'tir ve toplam 765 olur. Dışarıdan bakan için saat 12:45'tir.

`setSaniye(75)` çağrısı iki sürümde de reddedilir; kural (0–59) sınıfın içinde kalmıştır. `setSaniye(9)` çağrısında fark negatiftir: 9 − 45 = −36. Toplam 765 − 36 = 729 olur ve dakika kısmı yine 12'dir. Şekil 4.2.15 bu iki anı iki sürümle birlikte gösterir.



Şekil 4.2.15: Aynı arayüz, iki gerçekleme. Her çizimin üst kutusunda yeni sürümün tek alanı (`toplamSaniye`) ve ondan hesaplanan değerler, alt kutusunda eski sürümün iki alanı vardır. Solda `setSaniye(45)` sonrası (765 ↔ 12 ve 45), sağda `setSaniye(9)` sonrası (729 ↔ 12 ve 9).

Son satır `12:9` yazar; eski sürüm de aynısını verirdi. JDK 21 ile iki sürüm aynı çağrılarla yan yana çalıştırıldığında her adımda aynı dakika ve saniye değerleri bulunur: 12:45 ve 12:9. İç temsil değişti, `main`'in tek harfi değişmedi. Çıktıdaki `12:9` yazımı biçimlendirme yapılmadığı içindir; `printf("%d:%02d", ...)` ile `12:09` elde edilir.

Bu değişikliğin bir bedeli de vardır. Eski sürümde `getDakika()` yalnızca bir alan okur. Yeni sürümde bir bölme ve bir kalan işlemi yapar. İkisi de O(1)'dir; fark önemsizdir. Ama değiştiriciler artık önce erişimciyi çağırır. Tasarımcı bu tür takasları kullanıcıya hiç yansıtmadan yapabilir. Sarmalamanın sağladığı özgürlük budur.

4.2.6.4 Tasarım önerileri

Sarmalanmış bir sınıf yazarken dört öneri izlenir:

1. Sınıf içindeki bütün alanlar `private` tanımlanır.
2. Alanlara erişim için *gerektiği kadar* `public` metot sağlanır. Her alana otomatik olarak değiştirici yazmak şart değildir.
3. Her metodun nasıl kullanılacağını belirten yorumlar yazılır: ne yapar, ön ve son koşulları nelerdir.
4. Yalnızca sınıf içinde kullanılan yardımcı metotlar `private` tanımlanır.

Dördüncü öneri alanlar kadar metotlar için de sarmalamanın geçerli olduğunu hatırlatır:

```
public class Ogrenci {
    private int not;
    private boolean gecerliMi(int deger) {
        return deger >= 0 && deger <= 100;
    }
    public void setNot(int not) {
        if (gecerliMi(not)) this.not = not;
    }
}
```

`gecerliMi` sınıfın iç işidir. `setNot` onu kullanır, ama dışarıya açılması için bir neden yoktur. Başka bir sınıftan `o.gecerliMi(50)` çağrıldığında derleyici şu hatayı verir:

```
Dis.java:4: error: gecerliMi(int) has private access in Ogrenci
        boolean b = o.gecerliMi(50);
                      ^
1 error
```

Yardımcı metodu `private` tutmak, sınıfın dış yüzünü küçük tutar. Dış yüzdeki her metot bir sözdür ve sonradan geri alınamaz. `private` metot ise yazarın istediği zaman değiştirebileceği, silebileceği bir ayrıntıdır.

4.2.6.5 Sor değil, söyle

Her alana bir erişimci ve bir değiştirici yazmak sarmalama değildir. Kurallar yine sınıfın dışında uygulanıyorsa veri yalnızca bir adım uzağa taşınmıştır. Aşağıdaki iki kod parçası farkı gösterir. Birincisinde kullanan kod, nesneye verisini **sorar** ve kararı kendisi verir:

```
// her para çeken yerde aynı kontrol
if (h.getBakiye() >= 300) {
    h.setBakiye(h.getBakiye() - 300);
}
// ...ve not ekleyen her yerde
o.setNot(Math.min(100, o.getNot() + 5));
```

Bu yazımda kural, nesneyi kullanan her yerde tekrar eder. Biri unutursa nesne bozulur. Üstelik `setBakiye` yazıldığı anda herkes bakiyeyi istediği değere çekebilir; sınıfın değişmezi korumasız kalır. İkinci yazımda kullanan kod, nesneye ne istediğini **söyler**:

```
h.paraCek(300);   // kural BankaHesabi'nda
o.notEkle(5);     // 100 sınırı Ogrenci'de
```

Kararı, veriyi tutan nesne verir. 300 TL'lik çekimin yapılıp yapılmayacağına `BankaHesabi` karar verir; notun 100'ü aşmamasını `Ogrenci` sağlar. Değiştirici ancak alanın gerçekten serbestçe değişebildiği durumlarda yazılır. `BankaHesabi`'nda bu yüzden `setBakiye` yoktur.

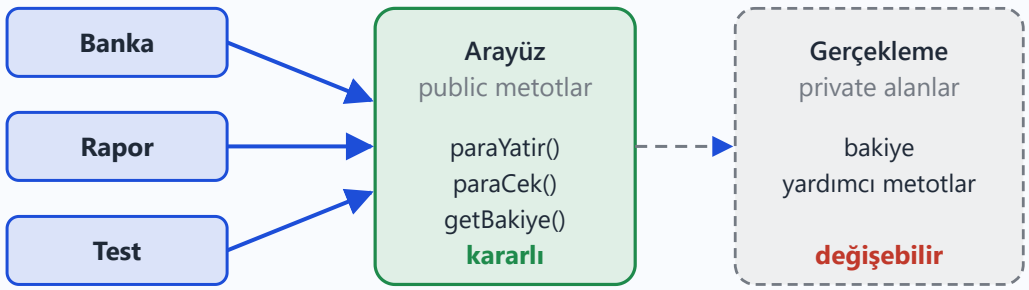
Bir metodun nereye ait olduğunu bulmak için basit bir soru yeter: "Bu kararı vermek için gereken veri kimde?" Cevap nesnenin kendisiyse, karar da onun metodunda olmalıdır.

4.2.6.6 Sarmalamanın kazançları

Notun buraya kadarki örnekleri sarmalamanın dört kazancını somut olarak göstermiştir:

- **Soyutlama.** Nesne ile kullanıcısı arasındaki ilişki sadeleşir; kullanıcı yalnızca metotları bilir. `islem.veriEkle("Ayşe")` tek satırdır.
- **Koruma.** Nesne, kullanıcının istenmeyen ya da hatalı erişimlerine karşı korunur. `s.saniye = 75;` derlenmez, `setSaniye(75)` reddedilir.
- **Değiştirilebilirlik.** Arayüz değiştirilmeden sınıfın iç yapısı değiştirilebilir; kullanan kod etkilenmez. İki alanlı saat, tek alanlı saate dönüştü ve `main` aynı kaldı.
- **Geçerli durum.** Nesnenin durumu her zaman kurallara uyar. Yalnızca pozitif koordinatlı noktalar, 0–59 arasında saniye ve dakika, negatif olmayan saat ya da bakiye. `paraCek` bakiyeyi asla eksiye düşürmez.

Bu kazançların büyük ölçekteki sonucu Şekil 4.2.16'da görülür. Bir programda `BankaHesabi`'nı kullanan birçok sınıf olabilir: banka işlemleri, raporlar, testler. Hepsi yalnızca arayüze bağlıdır.



Şekil 4.2.16: Bağımlılık yönü. `Banka`, `Rapor` ve `Test` sınıfları yalnızca arayüze (`paraYatir()`, `paraCek()`, `getBakiye()`) bağlıdır; arayüz kararlıdır. Gerçekleme (özel alanlar ve yardımcı metotlar) arayüzün arkasındadır ve değişebilir.

Arayüzler gerçeklemeye göre çok daha seyrek değişir; bu, kodun kararlı olmasını sağlar. Gerçekleme değiştiğinde ona doğrudan bağımlı olan kodların da değişmesi gerekir. Gerçeklemeyi gizlemek bu bağımlılığı ortadan kaldırır ve değişiklik sınıfın içinde kalır. Bu duruma **gevşek bağlılık** (*loose coupling*) denir. Gevşek bağlı bir programda bir sınıfı değiştirmek, ötekilere dokunmayı gerektirmez.

4.2.7 Değiştirilemez sınıflar

Alanları `private` yapmak ve değiştirici yazmamak, sarmalamanın tamamlandığı anlamına gelmez. Alan bir dizi ya da liste gibi değiştirilebilir bir nesneyi gösteriyorsa, o nesneye giden referans dışarı sızabilir. Bu kesim sızıntının nasıl oluştuğunu, nasıl kapatıldığını ve hiç değişmeyen nesnelerin neden bu sorunu baştan çözdüğünü anlatır.

4.2.7.1 Sızan referans

`Karne` sınıfı notları bir dizide tutar. Alan `private final`'dır ve değiştirici yoktur. İlk bakışta notlar korunmuş görünür:

```
public class Karne {
    private final int[] notlar;
    public Karne(int[] notlar) {
        this.notlar = notlar; // aynı dizi
    }
    public int[] getNotlar() {
        return notlar;        // iç dizi
    }
    public double ortalama() {
        int t = 0;
        for (int n : notlar) t += n;
        return (double) t / notlar.length;
    }
}
```

```
int[] dizi = {70, 80, 90};
Karne k = new Karne(dizi);
dizi[0] = 0;                // dışarıdaki dizi
k.getNotlar()[1] = -100;    // getter'dan gelen
System.out.println(k.ortalama());
```

```
-3.3333333333333335
```

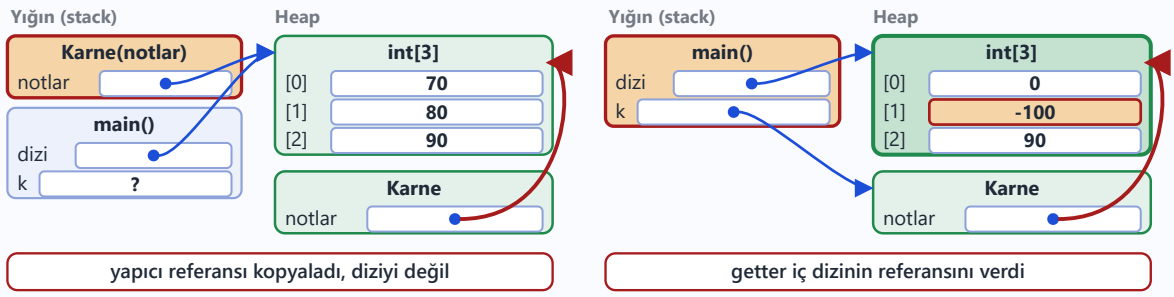
Notlar dışarıdan değişmiştir; üstelik −100 gibi geçersiz bir not içeri girmiştir. Bunun nedeni, dizinin bir nesne olmasıdır. Yapıcı ve erişimci diziyi değil, yalnızca referansını paylaşır. Aynı diziye üç ayrı yerden ulaşılabilir: `main`'deki `dizi` değişkeninden, `Karne`'nin alanından ve erişimcinin döndürdüğü referanstan. Tablo 4.2.11 bu durumun nasıl oluştuğunu adım adım gösterir.

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	8: `int[] dizi = {70, 80, 90};`	main()	dizi=→[70, 80, 90]	—
2	9: `Karne k = new Karne(dizi);` 2: `this.notlar = notlar;`	main() → Karne(notlar)	dizi=→[70, 80, 90], k=→Karne	—
3	10: `dizi[0] = 0;`	main()	dizi=→[0, 80, 90], k=→Karne	—
4	11: `k.getNotlar()[1] = -100;` 5: `return notlar;`	main()	dizi=→[0, -100, 90], k=→Karne	—
5	12: `System.out.println(k.ortalama());`	main()	dizi=→[0, -100, 90], k=→Karne	`-3.3333333333333335`

Tablo 4.2.11: Sızan referansın izlenmesi. Satır numaraları: 1–6 `Karne`'nin yapıcısı ve erişimcisi, 8–12 `main`.

Birinci adımda dizi heap'te oluşur; `dizi` değişkeni yalnızca onun referansını tutar. İkinci adımda yapıcı parametredeki referansı alana yazar. Artık `Karne`'nin `private` alanı ile `main`'in `dizi` değişkeni aynı diziyi gösterir. Yapıcı referansı kopyalamıştır, diziyi değil.

Üçüncü adımda `dizi[0] = 0` çalışır. Bu satır `Karne`'nin hiçbir metodundan geçmez, ama `Karne`'nin gördüğü not değişir. Dördüncü adımda `getNotlar()` iç dizinin kendisini döndürür ve dönen referansla −100 yazılır. Geçersiz not hiçbir denetimden geçmeden nesneye girmiştir (Şekil 4.2.17).



Şekil 4.2.17: Sızan referans. Solda yapıcı çalışıyor: `notlar` parametresi ve `Karne`'nin alanı, `main`'deki `dizi` ile aynı `int[3]` nesnesini gösteriyor. Sağda erişimci iç dizinin referansını vermiş ve `[1]` hücresine −100 yazılmış.

Son adımda ortalama (0 − 100 + 90) / 3 ≈ −3,33 bulunur. Java bu değeri `-3.3333333333333335` olarak yazar; son basamaktaki 5, `double`'ın ikili gösteriminden gelir. `private` yazmak yetmedi: iç nesneye giden referans dışarı sızdı.

`final` sözcüğü de bu durumu önlemez. `final` yalnızca referansın başka bir diziyi göstermesini engeller; dizinin içeriğini korumaz. Sınıfın içinde `notlar[0] = 0;` sorunsuz derlenir, `notlar = new int[2];` ise `cannot assign a value to final variable notlar` hatası verir. Değişmeyen, kutudaki referanstır; referansın gösterdiği nesne değil.

4.2.7.2 Savunmacı kopya

Tanım 4.2.10

Savunmacı kopya (*defensive copy*), değiştirilebilir bir nesnenin sınıfa girerken ve sınıftan çıkarken kopyalanmasıdır. İç nesneye ulaşan tek referans, sınıfın kendi alanı olarak kalır.

```
public class Karne {
    private final int[] notlar;
    public Karne(int[] notlar) {
        this.notlar = notlar.clone(); // girişte
    }
    public int[] getNotlar() {
        return notlar.clone();        // çıkışta
    }
    // ortalama() aynı
}
```

Aynı `main` metoduyla bu sürüm şunu yazar:

```
80.0
```

Yapıcıdaki `notlar.clone()`, çağıranın elindeki diziyi sonradan değiştirmesine karşı korur. `dizi[0] = 0` artık yalnızca `main`'in dizisini değiştirir; `Karne`'nin kopyası 70'te kalır. Erişimcideki kopya ise dönen dizinin değiştirilmesine karşı korur. `k.getNotlar()[1] = -100` geçici bir kopyaya yazar ve o kopya hemen kaybolur. Ortalama (70 + 80 + 90) / 3 = 80.0'dır.

Listeler için iki seçenek vardır. `new ArrayList<>(liste)` bağımsız bir kopya üretir. `Collections.unmodifiableList(liste)` ise kopyalamaz; aynı listeye salt okunur bir **görünüm** verir. İkisinin farkı JDK 21 ile şöyle görülür:

```
import java.util.*;
public class Liste {
  public static void main(String[] args) {
    List<Integer> ic = new ArrayList<>(List.of(70, 80));
    List<Integer> gorunum = Collections.unmodifiableList(ic);
    List<Integer> kopya = new ArrayList<>(ic);
    ic.add(90);
    System.out.println(gorunum + " " + kopya);
    try {
      gorunum.add(100);
    } catch (UnsupportedOperationException e) {
      System.out.println(e.getClass().getName());
    }
    List<Integer> sabit = List.copyOf(ic);
    ic.set(0, 0);
    System.out.println(sabit + " " + gorunum);
  }
}
```

```
[70, 80, 90] [70, 80]
java.lang.UnsupportedOperationException
[70, 80, 90] [0, 80, 90]
```

İç listeye 90 eklendikten sonra görünüm `[70, 80, 90]`'ı, kopya ise `[70, 80]`'i gösterir. Görünüm, iç listenin değişikliklerini yansıtır; yalnızca görünüm üzerinden değişiklik yapılamaz. `gorunum.add(100)` bir `UnsupportedOperationException` fırlatır. Java 10'dan beri bulunan `List.copyOf` ise değiştirilemez bir kopya üretir: iç listenin ilk elemanı 0 yapıldığında kopya etkilenmez.

`String` ve `Integer` gibi değiştirilemez nesneler kopyalanmaz. Onları değiştirmenin bir yolu olmadığı için paylaşılmaları zararsızdır. Kopyalamanın bir maliyeti de vardır: n elemanlı bir dizinin kopyası O(n) zaman ve O(n) ek bellek ister. Erişimci her çağrıldığında kopya üretildiği için sık çağrılan büyük dizilerde bu maliyet birikir. Böyle durumlarda salt okunur bir görünüm ya da baştan değiştirilemez bir tip seçmek daha ucuzdur.

4.2.7.3 Değiştirilemez sınıf tasarlamak

Tanım 4.2.11

Değiştirilemez (*immutable*) bir nesne, oluşturulduktan sonra durumu hiç değişmeyen nesnedir. `String` bunun en bilinen örneğidir.

Sızan referans sorununu kökünden çözmenin yolu, nesnenin hiç değişmemesidir. Aşağıdaki `Para` sınıfı tutarı kuruş cinsinden bir `long` ile tutar:

```
public final class Para {        // alt sınıf yok
    private final long kurus;     // bir kez atanır
    private final String birim;
    public Para(long kurus, String birim) {
        this.kurus = kurus;
        this.birim = birim;
    }
    public Para ekle(Para diger) {   // setter yerine
        if (!birim.equals(diger.birim))
            throw new IllegalArgumentException("Birim farklı");
        return new Para(kurus + diger.kurus, birim);
    }
    @Override public String toString() {
        return kurus / 100 + ","
            + String.format("%02d", kurus % 100) + " " + birim;
    }
}
```

Değiştirilemez bir sınıf beş kurala uyar:

1. Değiştirici yoktur; durum yalnızca yapıcıda kurulur.
2. Alanlar `private final`'dır.
3. Sınıf `final`'dır, böylece değiştirilebilir bir alt sınıf yazılamaz.
4. “Değiştiren” metotlar nesneyi değiştirmez, yeni bir nesne döndürür.
5. Değiştirilebilir alanlar varsa savunmacı kopya yapılır.

Her kuralı derleyici de destekler. `final` bir alana yapıcı dışında değer atanmaya çalışıldığında, örneğin `kurus += k;` yazıldığında, `cannot assign a value to final variable kurus` hatası çıkar. `final` sınıftan türetme denemesi ise `cannot inherit from final Para` hatası verir. Üçüncü kural gereksiz görünebilir. Ama alt sınıf yazılabilseydi, o alt sınıf kendi değiştirilebilir alanlarını ekleyip `Para` yerine kullanılabilirdi.

Değiştirilemez sınıfın kazançları büyüktür. Değişmez yalnızca yapıcıda denetlenir; sonra hiç bozulamaz. Nesne kopyalanmadan paylaşılabilir; savunmacı kopya gerekmez. `HashMap` gibi yapılarda anahtar olarak güvenle kullanılabilir, çünkü anahtar değişip tablodaki yerini kaybetmez. İş parçacıkları arasında da güvenlidir; aynı nesneyi okuyan iki iş parçacığı birbirini rahatsız edemez. `String` ve `record` bu kuralların hazır örnekleridir.

`toString` metodu tutarı TL ve kuruş olarak yazar: `kurus / 100` tam lira kısmını, `kurus % 100` kuruş kısmını verir. `String.format("%02d", ...)` kuruşu iki basamağa tamamlar; 5 kuruş `0,05 TL` olarak görünür. Bu yazım negatif tutarlarda bozulur: `new Para(-50, "TL")` `0,-50 TL` yazar, çünkü

−50 / 100 = 0 ve −50 % 100 = −50'dir. Sınıf negatif tutara izin verecekse `toString`'in bu durumu ayrıca ele alması gerekir.

4.2.7.4 Para sınıfını izlemek

```
public Para ekle(Para diger) {
    if (!birim.equals(diger.birim))
        throw new IllegalArgumentException(
            "Birim farklı");
    return new Para(kurus + diger.kurus, birim);
}
// main içinde:
Para a = new Para(1250, "TL");
a.ekle(new Para(100, "TL"));  // sonuç atıldı
Para b = a.ekle(new Para(375, "TL"));
System.out.println(a + " | " + b);
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	8: `Para a = new Para(1250, "TL");`	main()	a=12,50 TL	—

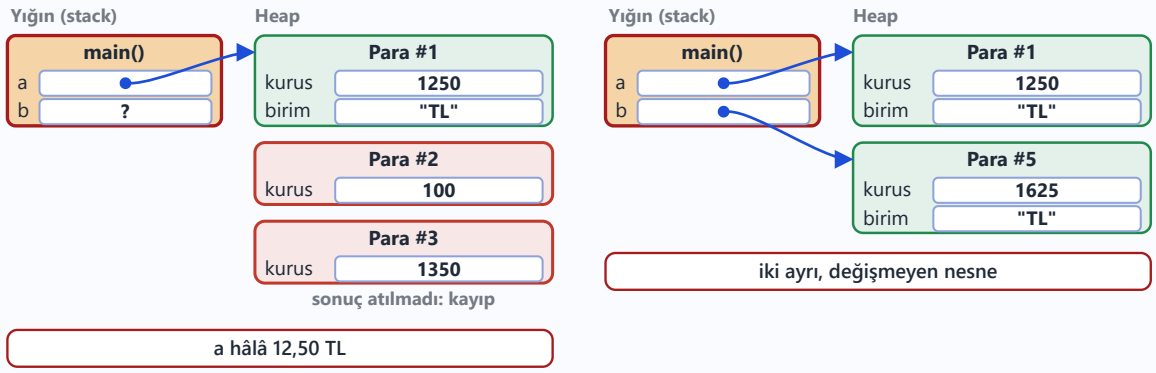
2	9: `a.ekle(new Para(100, "TL")); // sonuç atıldı`
	5: `return new Para(kurus + diger.kurus, birim);`
		main() → a.ekle(diger)	a=12,50 TL	—

3	9: `a.ekle(new Para(100, "TL")); // sonuç atıldı`	main()	a=12,50 TL	—
4	10: `Para b = a.ekle(new Para(375, "TL"));` 5: `return new Para(kurus + diger.kurus, birim);`	main()	a=12,50 TL, b=16,25 TL	—
5	11: `System.out.println(a + " | " + b);`	main()	a=12,50 TL, b=16,25 TL	`12,50 TL	16,25 TL`

Tablo 4.2.12: Değiştirilemez `Para` nesnelerinin izlenmesi. Satır numaraları: 1–6 `ekle` metodu, 8–11 `main`.

Birinci adımda ilk nesne kurulur: 1250 kuruş, yani 12,50 TL. Alanları `final` olduğu için bundan sonra hiçbir metot onları değiştiremez. İkinci adımda `a.ekle(new Para(100, "TL"))` çağrılır. Argüman olarak ikinci bir nesne (#2) kurulur. `ekle`, `a`'yı değiştirmez; toplamı taşıyan yeni bir nesne (#3) kurup döndürür.

Üçüncü adım en sık yapılan hatayı gösterir. Dönen değer bir değişkene atanmamıştır; #3 kaybolur ve `a` değişmemiştir (Şekil 4.2.18). Derleyici bu satır için uyarı vermez; değer döndüren bir metodun sonucunu yok saymak Java'da serbesttir. Dördüncü adımda sonuç `b`'ye atanır: 1250 + 375 = 1625 kuruş. Argüman olarak kurulan #4 de artık kullanılmaz. Son adımda `a` ilk değerini korumuştur.



Şekil 4.2.18: Değiştirilemez nesneler. Solda `ekle`'nin sonucu atılmış: #3 nesnesine (1350 kuruş) hiçbir değişken bağlı değil, `a` hâlâ 12,50 TL. Sağda programın sonu: `a` 1250 kuruşu, `b` 1625 kuruşu gösteriyor; iki ayrı, değişmeyen nesne.

```
12,50 TL | 16,25 TL
```

`String`'in metotları da aynı biçimde çalışır. Aşağıdaki deneme `Para` sınıfının öteki davranışlarını da gösterir:

```
public class ParaDeneme {
  public static void main(String[] args) {
    String s = "java";
    s.toUpperCase();
    System.out.println(s);
    s = s.toUpperCase();
    System.out.println(s);
    Para a = new Para(1250, "TL");
    try {
      a.ekle(new Para(100, "USD"));
    } catch (IllegalArgumentException e) {
      System.out.println(e.getMessage());
    }
    System.out.println(new Para(5, "TL") + " " + new Para(-50, "TL"));
  }
}
```

```
java
JAVA
Birim farklı
0,05 TL 0,-50 TL
```

`s.toUpperCase();` tek başına `s`'yi değiştirmez ve `java` yazılır; `s = s.toUpperCase();` yazıldığında `JAVA` elde edilir. Farklı birimlerin toplanması `Birim farklı` iletisiyle reddedilir. Son satır, `toString`'in 5 kuruşu doğru, −50 kuruşu ise yanlış yazdığını gösterir. Doğru kalıp her zaman aynıdır: `a = a.ekle(p);`.

Sık yapılan hata

Değiştirilemez nesnenin metodunu değiştirici sanmak. `a.ekle(p);`, `s.toUpperCase();`, `s.trim();` gibi satırlar tek başına hiçbir şeyi değiştirmez. Metot yeni bir nesne döndürür ve sonuç atanmazsa kaybolur. Derleyici uyarmaz; hata ancak çıktıda görülür.

4.2.8 Paketler

Sarmalama tek bir sınıfın sınırında durmaz. Büyük bir program yüzlerce sınıftan oluşur ve bu sınıfların bir kısmı yalnızca birkaç komşu sınıfın işine yarar. Paketler sınıfları gruplar, ad çakışmalarını önler ve sınıf düzeyinin bir üstünde ikinci bir sarmalama katmanı kurar.

4.2.8.1 Paket ve klasör

Tanım 4.2.12

Paket (*package*), bir klasör içinde gruplanmış sınıflar topluluğudur; klasörün adı paketin adıdır. Bir dosyanın hangi pakete ait olduğu, dosyanın ilk satırındaki `package paket_adi;` bildirimiyle belirtilir.

Paketlerle çalışırken birkaç kural geçerlidir. Her `public` sınıf ayrı bir dosyada, sınıfla aynı adı taşıyan dosyada tanımlanır. `package` bildirimi dosyanın ilk satırıdır; ondan önce yalnızca yorum gelebilir. İç içe klasörlerde `/` ya da `\` yerine nokta kullanılır: `okul/yonetim` klasörü `okul.yonetim` paketidir. Paket adları küçük harfle yazılır. Java'nın kendi paketleri de böyle adlandırılmıştır: `java.lang`, `java.util`, `java.io`.

Aşağıdaki proje yapısında iki paket vardır. `banka` paketi hesap işlerini, `uygulama` paketi programın giriş noktasını içerir. Kaynak dosyalar `src/`, derlenmiş dosyalar `out/` altındadır:

```
proje/
├── src/
│   ├── banka/
│   │   ├── Hesap.java      // package banka;
│   │   └── Yardimci.java   // package banka;
│   └── uygulama/
│       └── Ana.java        // package uygulama;
└── out/                    // derleme çıktısı
    ├── banka/
    │   ├── Hesap.class
    │   └── Yardimci.class
    └── uygulama/
        └── Ana.class
```

Bir sınıfın **tam adı** paketini de içerir: `banka.Hesap`, `uygulama.Ana`. Derleyici ve Java sanal makinesi paketleri **sınıf yolu** (*classpath*) köklerinden başlayarak arar. Paket adı bu kökten itibaren klasör yolunu söyler: `banka.Hesap` sınıfı, kökün altındaki `banka/Hesap.class` dosyasıdır. Varsayılan sınıf yolu, komutun çalıştırıldığı klasördür (`.`).

> **Sık yapılan hata**
>
> `public` **sınıfı başka adlı dosyaya yazmak.** `public class Hesap`, `Banka.java` dosyasına yazıldığında derleyici `class Hesap is public, should be declared in a file named Hesap.java` hatası verir. Dosya adı, `public` sınıfın adıyla büyük-küçük harf dahil birebir aynı olmalıdır.

4.2.8.2 Paketli programı derlemek ve çalıştırmak

İki paketli küçük bir program yazalım. `banka.Hesap` sınıfı yatırma ve bakiye okuma işlerini yapar; `uygulama.Ana` bu sınıfı kullanır:

```
package banka;

public class Hesap {
    private double bakiye;

    public void yatir(double miktar) {
        if (miktar > 0) {
            bakiye += miktar;
        }
    }

    public double getBakiye() {
        return bakiye;
    }
}
```

```
package uygulama;

import banka.Hesap;

public class Ana {
    public static void main(String[] args) {
        Hesap h = new Hesap();
        h.yatir(250.0);
        System.out.println("Bakiye: " + h.getBakiye());
    }
}
```

`Ana`, başka bir paketteki `Hesap`'ı kısa adıyla kullanabilmek için `import banka.Hesap;` satırını yazar. Proje klasöründe derleme ve çalıştırma komutları şöyledir:

```
> javac -d out src/banka/Hesap.java src/uygulama/Ana.java
> java -cp out uygulama.Ana
Bakiye: 250.0
```

`-d out` seçeneği, `.class` dosyalarını paket klasörleriyle birlikte `out/` altına yazar. JDK 21 ile çalıştırıldığında `out/banka/Hesap.class` ve `out/uygulama/Ana.class` dosyaları oluşur. Proje yapısında görünen `Yardimci.class` ancak `Yardimci.java` da derleme komutuna eklenirse oluşur. `-cp out` ise sınıf yolunu `out` klasörü yapar. Çalıştırırken sınıfın dosya yolu değil, **tam adı** verilir: `uygulama.Ana`.

Bu ayrım yeni başlayanların en çok takıldığı noktadır. Aynı projede yanlış komutlar denendiğinde JDK 21 şu iletileri verir:

Komut	Sonuç	Neden
`java -cp out uygulama.Ana`	`Bakiye: 250.0`	Sınıf yolu ve tam ad doğru.
`java -cp out Ana`	`ClassNotFoundException: Ana`	Paket adı eksik; kökte `Ana.class` yok.
`java -cp out/uygulama Ana`	`NoClassDefFoundError: Ana (wrong name: uygulama/Ana)`	Dosya bulundu, ama içindeki sınıfın adı `uygulama.Ana`.
`java -cp out/uygulama Ana.class`	`ClassNotFoundException: Ana.class`	Sınıf adı yerine dosya adı verilmiş.

Tablo 4.2.13: Paketli `uygulama.Ana` sınıfını çalıştırma denemeleri (proje klasöründe, JDK 21).

Üçüncü satırdaki ileti öğreticidir. Java `Ana.class` dosyasını bulur, ama dosyanın içindeki sınıf kendini `uygulama.Ana` olarak tanıtır. Paket adı klasör yapısıyla uyuşmadığı için sınıf yüklenmez. Paket, yalnızca bir klasör değil, sınıfın adının bir parçasıdır.

4.2.8.3 İçe aktarma (import)

Tanım 4.2.13

İçe aktarma (*import*), başka bir paketteki sınıfı kısa adıyla kullanmayı sağlayan bildirimdir. `import paket_adi.SinifAdi;` tek bir sınıfı, `import paket_adi.*;` paketteki bütün sınıfları içe aktarır.

`import` zorunlu değildir; bir sınıf her zaman tam adıyla da yazılabilir: `java.util.Random`. `import` yalnızca uzun adı kısaltır. Bazı sınıflar ise hiç `import` gerektirmez. `java.lang` paketindeki sınıflar (`String`, `Math`, `System`, `Integer`) ve aynı paketteki sınıflar her dosyada otomatik olarak görünür.

Bir kaynak dosyada sıra bellidir: önce `package`, sonra `import`'lar, en son sınıf. `import static` biçimi ise bir sınıfın `static` üyesini sınıf adı yazmadan kullandırır. Aşağıdaki program üç yolu bir arada gösterir:

```
import java.util.Scanner;
import static java.lang.Math.sqrt;

public class ImportOrnek {
  public static void main(String[] args) {
    Scanner giris = new Scanner(System.in);
    int sayi = giris.nextInt();
    java.util.Random rastgele =
        new java.util.Random(42);
    System.out.println(sqrt(sayi) + " "
        + Math.max(sayi, 10) + " "
        + rastgele.nextInt(100));
  }
}
```

Girdi olarak 49 verildiğinde çıktı şöyledir:

```
7.0 49 30
```

`Scanner`, `import java.util.Scanner;` sayesinde kısa adıyla kullanılır. `java.util.Random` ise `import` edilmeden tam adıyla yazılmıştır. `sqrt`, `import static java.lang.Math.sqrt;` sayesinde `Math.` yazılmadan çağrılır; 49'un karekökü `7.0`'dır. `Math.max` ise `java.lang`'tan geldiği için `import` gerektirmez ve 49 ile 10'un büyüğünü verir. Son sayı, 42 tohumuyla başlatılan rastgele sayı üretecinin ilk `nextInt(100)` değeridir. Aynı tohum her çalıştırmada aynı diziyi ürettiği için bu değer her zaman 30'dur.

`import` unutulduğunda derleyici sınıfı tanımaz. `Scanner`'ı içe aktarmadan kullanan bir dosyada JDK 21, satırdaki iki `Scanner` geçişi için ayrı ayrı hata verir:

```
ScannerYok.java:3: error: cannot find symbol
        Scanner giris = new Scanner(System.in);
        ^
  symbol:   class Scanner
  location: class ScannerYok
ScannerYok.java:3: error: cannot find symbol
        Scanner giris = new Scanner(System.in);
                            ^
  symbol:   class Scanner
  location: class ScannerYok
2 errors
```

Çözüm dosyanın başına `import java.util.Scanner;` eklemektir. `import` yalnızca derleme zamanına ait bir kolaylıktır; programın çalışmasını yavaşlatmaz ve `.class` dosyasına ek kod koymaz. `import java.util.*;` yazmak da paketteki bütün sınıfları programa "yüklemez"; derleyiciye yalnızca adları nerede arayacağını söyler.

4.2.8.4 Paket erişimi

Erişim belirleyicisi yazılmayan sınıf ve üyeler **paket erişimlidir**: yalnızca kendi paketlerinden görülürler. Bu kural sınıflar için de geçerlidir. `banka` paketindeki `Yardimci` sınıfı `public` değildir:

```
package banka;

class Yardimci {     // public değil
    static double kdv(double tutar) {
        return tutar * 0.20;
    }
}
```

```
package uygulama;

import banka.Yardimci;

public class Ana2 {
}
```

```
uygulama/Ana2.java:3: error: Yardimci is not public in banka; cannot be accessed from
outside package
import banka.Yardimci;
            ^
1 error
```

`Yardimci`, `banka` paketinin iç işidir. Aynı paketteki `Hesap` onu kullanabilir, `uygulama` paketi kullanamaz. Paket de böylece bir sarmalama düzeyi olur. Sınıflar dışarıya kapatılır; yalnızca `public` olanlar paketin dış yüzünü oluşturur. Bir sınıf `private` alanlarla kendi içini sakladığı gibi, bir paket de `public` olmayan sınıflarla kendi içini saklar.

Bir sınıfı paket dışından kullanılabilir yapmak için yalnızca sınıfı `public` yapmak yetmeyebilir. Kullanılan yapıcı ve metotlar da `public` olmalıdır. `okul.Ogrenci` sınıfı `public` yapıldığı hâlde `getAd()` metodu belirleyicisiz kalmışsa, `uygulama` paketinden yapılan `o.getAd()` çağrısı `getAd() is not public in Ogrenci; cannot be accessed from outside package` hatası verir.

4.2.8.5 Paket adlandırma ve alt paketler

Dünyada milyonlarca Java sınıfı yazılmıştır ve `banka` gibi basit bir paket adı başkasıyla kolayca çakışabilir. Çakışmayı önlemek için kurumlar paket adına ters çevrilmiş alan adıyla başlar. Giresun Üniversitesi'nin alan adı `giresun.edu.tr` olduğu için paket adı `tr.edu.giresun.banka` olur. Noktalı adlar klasörlere karşılık gelir: `tr/edu/giresun/banka/`.

Adların iç içe görünmesi bir erişim hakkı vermez. **Alt paket ayrı bir pakettir.** `tr.edu.giresun.banka.rapor` paketi, `tr.edu.giresun.banka` paketinin paket erişimli üyelerini göremez:

```
// tr/edu/giresun/banka/Yardimci.java
package tr.edu.giresun.banka;
class Yardimci {                  // paket erişimli
    static double kdv(double t) { return t * 0.20; }
}
```

```
// tr/edu/giresun/banka/rapor/Rapor.java
package tr.edu.giresun.banka.rapor;
import tr.edu.giresun.banka.Yardimci;
public class Rapor {
    double hesapla() { return Yardimci.kdv(100); }
}
```

```
tr/edu/giresun/banka/rapor/Rapor.java:3: error: Yardimci is not public in
tr.edu.giresun.banka; cannot be accessed from outside package
import tr.edu.giresun.banka.Yardimci;
                          ^
tr/edu/giresun/banka/rapor/Rapor.java:5: error: Yardimci is not public in
tr.edu.giresun.banka; cannot be accessed from outside package
    double hesapla() { return Yardimci.kdv(100); }
                              ^
2 errors
```

Aynı kural `import` için de geçerlidir. `import tr.edu.giresun.banka.*;` alt paketleri içe aktarmaz. `import java.util.*;` da `java.util.function` paketindeki sınıfları getirmez. `Function` arayüzü kullanılmak istendiğinde derleyici sınıfı bulamaz:

```
FunctionYok.java:5: error: cannot find symbol
        Function<Integer, Integer> f = x -> x * 2;
        ^
  symbol:   class Function
  location: class FunctionYok
1 error
```

Çözüm `import java.util.function.Function;` satırını ayrıca yazmaktır. Paket adı ile klasör yolu birebir eşleşmezse derleyici ve sanal makine sınıfı bulamaz. Geliştirme ortamları bu yüzden klasörleri paket adından kendileri oluşturur.

4.2.8.6 Sınıf adı çakışmaları

Farklı paketlerdeki iki sınıf aynı adı taşıyabilir. Java kütüphanesinde bunun bilinen örneği `java.util.Date` ile `java.sql.Date`'tir. İki paket de `*` ile içe aktarılırsa `Date` adı belirsiz kalır:

```
import java.util.*;
import java.sql.*;

public class Belirsiz {
    public static void main(String[] args) {
        Date tarih = null;
    }
}
```

```
Belirsiz.java:6: error: reference to Date is ambiguous
        Date tarih = null;
        ^
  both class java.sql.Date in java.sql and class java.util.Date in java.util match
1 error
```

Derleyici iki sınıfı da bulur ve hangisinin kastedildiğini seçemez. Belirsizliği gidermek için paket adı kullanılır. Biri tek sınıflık `import` ile içe aktarılır, öteki tam adıyla yazılır:

```
import java.util.Date;

public class TarihOrnegi {
  public static void main(String[] args) {
    Date baslangic = new Date(0);  // java.util
    java.sql.Date kayit =          // tam ad
        java.sql.Date.valueOf("2024-09-16");
    System.out.println(baslangic.getTime());
    System.out.println(kayit);
  }
}
```

```
0
2024-09-16
```

Tek sınıflık `import java.util.Date;`, `*` ile yapılan içe aktarmalardan önce gelir. `Date` artık yalnızca `java.util.Date` anlamına gelir. `new Date(0)`, 1 Ocak 1970 gece yarısını (UTC) gösteren nesneyi kurar; `getTime()` bu andan beri geçen milisaniyeyi verdiği için 0 yazılır. `java.sql.Date` tam adıyla kullanılmıştır; `valueOf` metodu `yyyy-aa-gg` biçimindeki metinden tarih üretir ve aynı biçimde yazdırılır.

4.2.9 Örnek problemler

Bu kesimdeki problemler sunumun örnek problemleri ve alıştırmalarıdır. Her birinin cevabı JDK 21 ile çalıştırılarak doğrulanmıştır.

Örnek 4.2.1

Erişim metotlarıyla yazılmış `DijitalSaat` sınıfı kullanılıyor (saat 0–23, dakika ve saniye 0–59). Aşağıdaki iki kod parçası ne yazdırır? İkinci parçada kaç `DijitalSaat` nesnesi oluşturulur?

```
DijitalSaat t = new DijitalSaat();
t.setSaat(25);
t.setDakika(30);
t.setSaniye(-1);
t.setSaniye(59);
t.goster();
```

```
DijitalSaat a = new DijitalSaat();
DijitalSaat b = a;
a.setDakika(15);
b.setDakika(45);
b = new DijitalSaat();
b.setDakika(10);
System.out.println(a.getDakika() + " " + b.getDakika());
```

```
00:30:59
45 10
```

İlk parçada `setSaat(25)` ve `setSaniye(-1)` değiştiricilerce reddedilir. Saat başlangıç değeri olan 0'da kalır; dakika 30, saniye ise son geçerli çağrıyla 59 olur. Sonuç `00:30:59`'dur.

İkinci parçada `b = a` ataması yeni nesne oluşturmaz; iki değişken aynı nesneyi gösterir. `a.setDakika(15)` ve `b.setDakika(45)` aynı nesneye uygulanır ve dakika 45 olur. Sonra `b` yeni bir nesneye bağlanır ve onun dakikası 10 yapılır. `a` ilk nesneyi göstermeye devam eder. Çıktı `45 10`'dur. Toplam iki nesne oluşturulmuştur, çünkü `new` iki kez çalışmıştır.

Örnek 4.2.2

Aşağıdaki A–E satırlarından hangileri derleme hatası verir? Neden?

```
public class Test {
    public static void main(String[] args) {
        DijitalSaat s = new DijitalSaat();
        s.setSaniye(30);                    // A
        int x = s.saniye;                   // B
        System.out.println(s.getSaniye());  // C
        s.getSaniye() = 10;                 // D
        s.setSaniye("40");                  // E
    }
}
```

```
Test.java:5: error: saniye has private access in DijitalSaat
        int x = s.saniye;                   // B
                 ^
Test.java:7: error: unexpected type
        s.getSaniye() = 10;                 // D
                      ^
  required: variable
  found:    value
Test.java:8: error: incompatible types: String cannot be converted to int
        s.setSaniye("40");                  // E
                    ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
3 errors
```

B satırı `private` alanı okumaya çalışır; doğrusu `s.getSaniye()`'dir. D satırında bir metot çağrısının sonucuna atama yapılmak istenmiştir. Metot çağrısı bir değerdir, değişken değildir; iletideki `required: variable, found: value` bunu söyler. Doğrusu `s.setSaniye(10)`'dur. E satırında `"40"` bir `String`'dir ve `int` parametreye otomatik dönüştürülmez. A ve C doğrudur, çünkü `public` metotları çağırır.

Örnek 4.2.3

`BankaHesabi` sınıfıyla aşağıdaki kod ne yazdırır? Son satırda hangi bakiyeler görülür?

```
BankaHesabi h = new BankaHesabi(111, 1000.0, "Vadesiz");
h.paraCek(5000.0);
h.paraYatir(-50.0);
h.faizEkle(3.0);
h.paraCek(250.0);
BankaHesabi v = new BankaHesabi(222, 100.0, "Vadeli");
h.transfer(v, 1000.0);
h.transfer(v, 750.0);
System.out.println(h.getBakiye() + " " + v.getBakiye());
```

```
Yetersiz bakiye.
Yatırılan miktar geçersiz.
Faiz yalnızca vadeli hesaplarda uygulanabilir.
250.0 TL çekildi. Kalan bakiye: 750.0
Yetersiz bakiye veya geçersiz miktar.
750.0 TL yatırıldı. Yeni bakiye: 850.0
750.0 TL transfer edildi. Yeni bakiye: 0.0
0.0 850.0
```

Bakiye 1000.0'dır. 5000.0 bakiyeyi aştığı için ilk çekim reddedilir. −50.0 geçersiz bir yatırma miktarıdır. Hesap vadesiz olduğu için faiz eklenmez. 250.0'lık çekim geçerlidir ve bakiye 750.0'a iner. İkinci hesap vadeli ve 100.0 bakiyelidir. 1000.0'lık transfer bakiyeyi aştığı için reddedilir. 750.0'lık transfer ise tam bakiye kadardır ve `miktar ≤ bakiye` koşulunu sağlar. Önce hedef hesabın mesajı (850.0), sonra transferin mesajı (0.0) yazılır. Son satır `0.0 850.0`'dır.

Örnek 4.2.4

(a) Kopyalamayan `Karne` ile aşağıdaki ilk kod ne yazdırır? Savunmacı kopyalı `Karne` ile ne yazdırır? (b) `Para` sınıfıyla ikinci kod ne yazdırır? (c) On kez `arttir()` çağıran `Sayac` programı `-ea` olmadan ve `-ea` ile çalıştırılırsa ne olur? (d) `Kesir` dışındaki bir sınıfta `new Kesir(1, 2).pay` derlenir mi?

```
int[] d = {50, 60};
Karne k = new Karne(d);
int[] g = k.getNotlar();
g[0] = 100;
d[1] = 100;
System.out.println(k.ortalama());
```

```
Para p = new Para(100, "TL");
p.ekle(p);
p = p.ekle(p);
p.ekle(new Para(1, "TL"));
System.out.println(p);
```

(a) Kopyalamayan sürümde `d`, `g` ve alan aynı diziyi gösterir. İki atama da nesneye ulaşır: dizi `{100, 100}` olur ve ortalama `100.0`'dır. Savunmacı kopyalı sürümde iki atama da kopyalara yapılır; nesnenin dizisi `{50, 60}` kalır ve ortalama `55.0`'dır. JDK 21 iki sürümde de bu değerleri verir.

(b) `p.ekle(p)` ve `p.ekle(new Para(1, "TL"))` çağrılarının sonuçları atılır. Yalnızca `p = p.ekle(p)` saklanır: 100 + 100 = 200 kuruş. Çıktı `2,00 TL`'dir. Bir nesnenin kendisini argüman olarak alması da sorun çıkarmaz, çünkü `ekle` hiçbir nesneyi değiştirmez.

(c) `-ea` olmadan `assert` satırı çalışmaz ve program `bitti` yazar; hata görünmez. `-ea` ile onuncu artırmada değer 10 olur ve `AssertionError: deger=10` fırlatılır. Hatayı yalnızca ikinci çalıştırma gösterir.

(d) Hayır. Derleyici `pay has private access in Kesir` hatası verir. `private` erişim yalnızca `Kesir`'in kendi kodunda serbesttir.

Örnek 4.2.5

Dört küçük sınıf yazalım. `Ogrenci`: `private` ad ve not; not yalnızca 0–100 arasında kabul edilsin; `gectiMi()` not ≥ 50 ise `true` döndürsün. `Nokta`: yalnızca negatif olmayan koordinatları kabul etsin; `uzaklik()` orijine uzaklığı döndürsün. `Termometre`: santigrat değeri saklasın, −273,15'in altını reddetsin; `getFahrenheit()` ayrı bir alan olmadan hesaplasın. `Sayac`: yalnızca `arttir()`, `sifirla()` ve `getDeger()` metotları olsun.

```
public class Ogrenci {
  private String ad;
  private int not;
  public Ogrenci(String ad) { this.ad = ad; }
  public void setNot(int not) {
    if (not >= 0 && not <= 100) this.not = not;
  }
  public int getNot() { return not; }
  public boolean gectiMi() { return not >= 50; }
}
```

```
public class Nokta {
  private double x, y;
  public Nokta(double x, double y) { setX(x); setY(y); }
  public void setX(double x) { if (x >= 0) this.x = x; }
  public void setY(double y) { if (y >= 0) this.y = y; }
  public double getX() { return x; }
  public double uzaklik() { return Math.sqrt(x * x + y * y); }
}
```

```
public class Termometre {
  private double santigrat;
  public Termometre(double c) { setSantigrat(c); }
  public void setSantigrat(double c) { if (c >= -273.15) santigrat = c; }
  public double getSantigrat() { return santigrat; }
  public double getFahrenheit() { return santigrat * 9 / 5 + 32; }
}
```

```
public class Sayac {
  private int deger;
  public void arttir() { deger++; }
  public void sifirla() { deger = 0; }
  public int getDeger() { return deger; }
}
```

```
public class Alistirma {
  public static void main(String[] args) {
    Ogrenci o = new Ogrenci("Ece");
    o.setNot(120);
    System.out.println(o.getNot() + " " + o.gectiMi());
    o.setNot(75);
    System.out.println(o.getNot() + " " + o.gectiMi());
    Nokta n = new Nokta(3, 4);
    System.out.println(n.uzaklik());
    n.setX(-7);
    System.out.println(n.getX());
    Termometre t = new Termometre(25);
    System.out.println(t.getFahrenheit());
    t.setSantigrat(-300);
    System.out.println(t.getSantigrat());
    Sayac s = new Sayac();
    s.arttir(); s.arttir(); s.arttir();
    System.out.print(s.getDeger() + " ");
    s.sifirla();
    System.out.println(s.getDeger());
  }
}
```

```
0 false
75 true
5.0
3.0
77.0
25.0
3 0
```

`setNot(120)` reddedilir ve not 0'da kalır: `0 false`. `setNot(75)` sonrasında `75 true` yazılır. `Nokta`'nın yapıcısı değiştiricileri çağırır; böylece kural tek yerde kalır. `new Nokta(3, 4)` için uzaklık √(9 + 16) = 5.0'dır. `setX(-7)` reddedilir ve x 3.0'da kalır. 25 °C, 25 × 9 / 5 + 32 = 77.0 °F'tır. Türetilen değer ayrı bir alanda tutulmaz; tutulsaydı iki alanın her değişiklikte uyumlu tutulması gerekirdi. `setSantigrat(-300)` sonrasında değer 25.0 kalır.

`Sayac`'ta değiştirici yoktur, çünkü sayacın değeri yalnızca birer birer artmalı ya da sıfırlanmalıdır. `setDeger` bu kuralı delerdi. Üç `arttir()` çağrısı değeri 3 yapar; `sifirla()` 0'a döndürür.

Yapıcının değiştirici çağırmasıyla ilgili küçük bir not: `javac -Xlint:all` ile derlendiğinde JDK 21 `Nokta` ve `Termometre` için `possible 'this' escape` uyarısı verir. Uyarı, bir alt sınıfın bu metotları değiştirip yarım kurulmuş nesneyi görebileceğini söyler. Sınıf `final` yapılırsa ya da çağrılan metotlar `final` ya da `private` olursa uyarı kaybolur.

Örnek 4.2.6

(a) `Ogrenci` sınıfı `okul` paketine taşınıp `uygulama.Ana`'dan kullanılacak. Hangi satırlar eklenir; derleme ve çalıştırma komutları nelerdir? (b) `BankaHesabi`'na yalnızca erişimcisi olan bir `islemSayisi` alanı eklenecek; başarılı her yatırma ve çekmede bir artacak. İki hesaplı izleme örneğinin sonunda iki hesabın işlem sayısı kaçtır?

(a) `Ogrenci.java` dosyasının başına `package okul;`, `Ana.java`'nın başına `package uygulama;` ve `import okul.Ogrenci;` yazılır. Dosyalar `src/okul/` ve `src/uygulama/` klasörlerine konur. Komutlar: `javac -d out src/okul/Ogrenci.java src/uygulama/Ana.java` ve `java -cp out uygulama.Ana`. `Ogrenci` sınıfının ve kullanılan metotlarının `public` olması gerekir.

```
1 2
```

(b) Sayaç `paraYatir` ve `paraCek` içinde, yalnızca başarılı dallarda artırılır. `hesap1`'in sayacı 1'dir: yalnızca `paraYatir(500.0)` sayılır. `faizEkle` bir yatırma değildir; `transfer` ise kendi bakiyesinden doğrudan düşer, kendi `paraCek`'ini çağırmaz. `hesap2`'nin sayacı 2'dir: `paraCek(200.0)` ve transferin içindeki `paraYatir`. Bu sonuç, transferin kendi tarafındaki işlemi `paraCek` üzerinden yapmamasının bir yan etkisini de gösterir. Kurallar tek yerde toplanmak isteniyorsa `transfer` de `paraCek`'i çağırmalıdır.

4.2.10 Sık yapılan hatalar

Sarmalamayla ilgili hataların bir kısmını derleyici yakalar ve iletisiyle yerini gösterir. Ötekiler derlenen ama nesneyi sessizce bozan programlara yol açar. İkinci gruptakiler daha tehlikelidir.

Sık yapılan hata

`private` alana dışarıdan erişmek. İleti: `saniye has private access in DijitalSaat`. `s.saniye = 75;` yerine `s.setSaniye(75);` yazılır; okumak için `s.getSaniye()` kullanılır.

Sık yapılan hata

Değiştiricide `this`'i unutmak. `en = en;` derlenir, ama parametreyi kendisine atar; alan değişmez ve `getEn()` 0 döndürür. Doğrusu `this.en = en;`'dir.

Sık yapılan hata

Erişimciyi `void` yazmak. `public void getEn() { return en; }` satırı `incompatible types: unexpected return value` hatası verir. `void` metot değer döndüremez; dönüş tipi alanın tipi, yani `int` olmalıdır.

Sık yapılan hata

Değiştiriciyi doğrulamasız yazmak. `public void setSaniye(int s) { saniye = s; }` alanı `public` yapmaktan farksızdır. Geçerlilik kuralı değiştiricide denetlenir.

Sık yapılan hata

Metot adını parantezsiz yazmak. `int d = s.getDakika;` satırında derleyici `getDakika` adlı bir alan arar: `cannot find symbol ... variable getDakika`. Metot çağrısı her zaman parantezle yazılır.

Sık yapılan hata

Arayüz metodunu yazmamak. İleti: `SporAraba is not abstract and does not override abstract method dur() in Araba`. `implements` diyen sınıf arayüzdeki bütün metotları yazmalıdır.

Sık yapılan hata

İç nesneyi erişimciyle vermek. `return notlar;` değiştirilebilir diziyi ya da listeyi dışarı açar. Kopya ya da salt okunur görünüm döndürülür. Yapıcıda gelen dizi de kopyalanır.

Sık yapılan hata

`final` alanı değiştirilemez sanmak. `private final int[] notlar` yazıldığında sabit olan referanstır, dizinin elemanları değil. `notlar[0] = 0;` derlenir.

Sık yapılan hata

Yapıcıda doğrulamamak. Değiştiriciler denetlese de `new BankaHesabi(1, -500.0, ...)` geçersiz bir nesne kurar. Yapıcı da kuralı uygulamalıdır.

Sık yapılan hata

`import`**'u unutmak ya da alt paketin içe aktarıldığını sanmak.** İleti: `cannot find symbol ... class Scanner`. `import java.util.*;` sonrasında `Function` için de aynı hata çıkar; `java.util.function` ayrıca içe aktarılır.

Sık yapılan hata

Paket erişimli sınıfı dışarıdan kullanmak. İleti: `Yardimci is not public in banka; cannot be accessed from outside package`. Sınıf ya `public` yapılır ya da yalnız aynı paketten kullanılır. Kullanılan metotların da `public` olması gerekir.

Sık yapılan hata

Her alana değiştirici yazmak. Kural kullanan koda kayar. Önce davranış (`paraCek`, `notEkle`) tasarlanır; çoğu zaman değiştirici gerekmez.

Sık yapılan hata

Hatayı sessizce yutmak. Geçersiz değeri hiçbir iz bırakmadan yok saymak hatayı gizler. Davranış ya sözleşmede açıkça belirtilir ya da istisna fırlatılır.

4.2.11 Özet

- **Sarmalama**, verinin ve onu işleyen metotların bir sınıfta toplanması ve verinin dışarıya kapatılmasıdır. Java'da alanlar `private`, davranışlar `public` yazılır.
- **Bilgi saklama** bir işin "ne"sini "nasıl"ından ayırır. Kullanıcı yalnızca metot imzasını ve sözleşmesini bilir; `islem.veriEkle("Ayşe")` çağrısının arkasındaki adımları bilmez.
- **Ön koşul** çağrıdan önce doğru olduğu varsayılan, **son koşul** çağrıdan sonra garanti edilen durumdur. İkisi metodun sözleşmesidir ve Javadoc'ta `@param`, `@return`, `@throws` etiketleriyle yazılır. Yorumdaki `<` işaretleri `{@code ...}` içine alınır.
- Ön koşul bozulduğunda metot değeri yok sayabilir, durum döndürebilir ya da istisna fırlatabilir. `public` metotlarda alışılmış seçim `IllegalArgumentException`'dır.
- **Sınıf değişmezi** her nesnenin her an sağladığı kuraldır. `assert` değişmezi geliştirme sırasında denetler, ama yalnızca `-ea` ile çalışır; argüman denetimi için kullanılmaz.
- Dört **erişim düzeyi** vardır: `public` (her yer), `protected` (paket ve alt sınıflar), paket erişimi (yalnız paket) ve `private` (yalnız sınıf). En kısıtlayıcı düzey tercih edilir. `private` denetimi nesneye göre değil, sınıfa göre yapılır.

- **Erişimci** alanı döndürür, **değiştirici** yeni değeri denetleyip atar. Parametre alanı gölgelediği için atama `this.alan = alan;` biçiminde yazılır.
- Yapıcı da doğrulamalıdır. Yapıcı istisna fırlatırsa `new` hiçbir referans üretmez ve geçersiz nesne kullanıcıya ulaşmaz.
- **Arayüz** sınıfın dış yüzü, **gerçekleme** iç mekanizmasıdır. `interface` ve `implements` bu ayrımı derleyicinin denetlediği bir yapıya dönüştürür. Gerçekleme değişse de arayüz aynı kalabilir: iki alanlı ve tek alanlı saat aynı çıktıyı verir.
- "Sor değil, söyle": kararı veriyi tutan nesne verir; değiştirici yalnızca alan gerçekten serbestçe değişebiliyorsa yazılır.
- Değiştirilebilir bir iç nesnenin referansı dışarı **sızabilir**. Yapıcıda ve erişimcide **savunmacı kopya** yapılır ya da salt okunur görünüm verilir. `final` yalnızca referansı sabitler.
- **Değiştirilemez sınıfta** değiştirici yoktur, alanlar `private final`'dır, sınıf `final`'dır ve "değiştiren" metotlar yeni nesne döndürür. Sonucu atamayı unutmak (`a.ekle(p);`) en sık yapılan hatadır.
- **Paket** klasördeki sınıf grubudur; sınıfın tam adı paketini içerir. Paketli program `javac -d out ...` ile derlenir, `java -cp out paket.Sinif` ile çalıştırılır. `import` başka paketteki sınıfı kısa adıyla kullandırır; alt paketleri getirmez. Aynı adlı iki sınıftan biri `import` edilir, öteki tam adıyla yazılır.

4.2.12 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

sarmalama *encapsulation* · bilgi saklama *information hiding* · veri gizleme *data hiding* · soyutlama *abstraction* · ön koşul / son koşul *precondition / postcondition* · sınıf değişmezi *class invariant* · doğrulama (assert) *assertion* · istisna *exception* · erişim belirleyici *access modifier* · erişimci (getter) *accessor* · değiştirici (setter) *mutator* · gölgeleme *shadowing* · this *this keyword* · yapıcı *constructor* · alan *field* · referans *reference* · arayüz *interface* · arayüz / gerçekleme *interface vs. implementation* · uygulamak *implement* · savunmacı kopya *defensive copy* · değiştirilemez *immutable* · kayıt (record) *record* · paket *package* · içe aktarma (import) *import*

4.2.13 Kendini Dene

1. Erişim metotlarıyla yazılmış `DijitalSaat` için şu kod çalıştırılıyor: `DijitalSaat a = new DijitalSaat(); DijitalSaat b = a; a.setSaat(23); b.setSaat(a.getSaat() + 1);`

`a.setDakika(a.getDakika() - 1); b.setSaniye(59); a.goster();`. Ne yazılır?

Cevap

`a` ve `b` aynı nesneyi gösterir. Saat 23 olur. `b.setSaat(24)` reddedilir, çünkü geçerli saatler 0–23'tür. `setDakika(-1)` de reddedilir ve dakika 0'da kalır. Saniye 59 olur. Çıktı: `23:00:59`.

2. `p1` paketindeki `public class A` sınıfında `public int acik`, `protected int korumali`, belirleyicisiz `int paket` ve `private int gizli` alanları var. `p2` paketinde `class C extends p1.A` sınıfının bir metodu dört alana da kendi adıyla değer atıyor. Hangi atamalar derlenmez?

Cevap

`acik` ve `korumali` atamaları derlenir: `public` her yerden, `protected` başka paketteki alt sınıftan erişilebilir. `paket` için `paket is not public in A; cannot be accessed from outside package`, `gizli` için `gizli has private access in A` hatası çıkar.

3. Tek alanlı (`toplamSaniye`) `DijitalSaat` yeni oluşturulmuş. Sırayla `setDakika(59)`, `setSaniye(59)` ve `setDakika(0)` çağrılıyor. Her çağrıdan sonra `toplamSaniye` kaçtır? Sonunda `getDakika() + ":" + getSaniye()` ne verir?

Cevap

İlk çağrıda fark 59 dakikadır: 59 · 60 = 3540. İkincide `getSaniye()` 0'dır, 59 eklenir: 3599. Üçüncüde `getDakika()` 3599 / 60 % 60 = 59'dur; fark −59 dakika, yani −3540 saniyedir: 59. Sonuç `0:59`. Saniye kısmı dakika değişikliğinden etkilenmez.

4. `veriEkle` metodundaki koşul yanlışlıkla `if (veri.isEmpty() || veri == null)` diye yazılmış. `veriEkle(null)` çağrısında ne olur? Koşul `veri == null || veri.trim().isEmpty()` yapılırsa `" "`, `null` ve `" a b "` argümanlarından kaçı kayıt ekler?

Cevap

İlk yazımda `null` üzerinde `isEmpty()` çağrılır ve program `NullPointerException` ile durur; `||`'nin kısa devresi ancak sol taraf doğruysa işe yarar. İkinci yazımda `" "` kırpılınca boş kalır ve reddedilir, `null` ilk koşulda reddedilir; yalnızca `" a b "` eklenir. JDK 21 çıktısı: `false false true 1`.

5. Bir Javadoc yorumunda `@param n 0 < n < 10 olmalı` yazılmış. `javadoc` komutu ne yapar?

Yorum nasıl düzeltilir?

Cevap

Javadoc yorumu HTML olarak işler ve ham `<` işaretini bir etiketin başlangıcı sanır. JDK 21 `malformed HTML` hatası verir. Düzeltme: `@param n {@code 0 < n < 10} olmalı`. `{@code ...}` içindeki metin olduğu gibi gösterilir. `<` yazmak da çalışır, ama kaynakta okunması zordur.

6. `List<Integer> ic = new ArrayList<>(List.of(70, 80));` listesinden `gorunum = Collections.unmodifiableList(ic)` ve `kopya = new ArrayList<>(ic)` üretiliyor; sonra `ic.add(90)` çalışıyor. `gorunum` ve `kopya` ne içerir? `gorunum.add(100)` ne yapar?

Cevap

Görünüm kopyalamaz, iç listeyi gösterir: `[70, 80, 90]`. Kopya bağımsızdır: `[70, 80]`. `gorunum.add(100)` `UnsupportedOperationException` fırlatır. Görünüm dışarıdan değişikliği engeller, ama sınıfın kendi içindeki değişiklikleri yansıtır.

7. `Para` sınıfıyla `System.out.println(new Para(-50, "TL"));` ne yazar? Bu bir hata mıdır; sınıfa hangi kural eklenebilir?

Cevap

−50 / 100 = 0 ve −50 % 100 = −50 olduğu için `0,-50 TL` yazılır. Tutar anlamsız görünür. Sınıf negatif tutara izin vermeyecekse yapıcı `kurus < 0` için `IllegalArgumentException` fırlatmalıdır. İzin verecekse `toString` işareti ayrı yazıp mutlak değerle çalışmalıdır.

8. Proje klasöründe `javac -d out src/banka/Hesap.java src/uygulama/Ana.java` çalıştırıldı. Ardından `java -cp out/uygulama Ana` denendi. Ne olur, neden?

Cevap

Java `out/uygulama/Ana.class` dosyasını bulur, ama dosyadaki sınıfın tam adı `uygulama.Ana`'dır. Sonuç: `NoClassDefFoundError: Ana (wrong name: uygulama/Ana)`. Doğru komut `java -cp out uygulama.Ana`'dır: sınıf yolu paketlerin kökü, sınıf adı ise tam addır.

9. `Urun` sınıfının yapıcısı fiyatı `setFiyat` ile atıyor; `setFiyat` pozitif olmayan değeri sessizce yok

sayıyor. `new Urun("Kalem", -5).getFiyat()` ne döndürür? Bu tasarımın sorunu nedir?

Cevap

Atama yapılmaz ve alan varsayılan değer olan `0.0`'da kalır. Nesne kurulmuştur, ama fiyatı 0 olan bir ürün "yalnızca pozitif fiyat" değişmezini bozar. Sessiz reddetme yapıcıda işe yaramaz. Yapıcının, geçersiz değerde `IllegalArgumentException` fırlatıp nesnenin hiç kurulmamasını sağlaması gerekir.

4.2.14 İleri okuma

- J. Bloch, *Effective Java* (3. baskı, Addison-Wesley, 2018), Madde 15–17. Sınıfların ve üyelerin erişimini, değiştirilebilirliği en aza indirmeyi ve `public` alan yerine erişim metodu kullanmayı kısa, gerekçeli kurallarla anlatır.
- Oracle, *The Java Tutorials*, "Controlling Access to Members of a Class" ve "Creating and Using Packages": erişim düzeylerini ve paketli programların derlenmesini resmî örneklerle anlatır.
- A. B. Downey ve C. Mayfield, *Think Java* (2. baskı, O'Reilly, 2019), 11. bölüm "Designing Classes": bir sınıfı alanları, yapıcıları, erişim metotları ve değişmezleriyle adım adım kurar; çevrim içi olarak ücretsiz okunabilir.
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21, §6.6 ve §7: erişim denetiminin ve paketlerin kesin kuralları; özellikle `protected` erişimin ayrıntıları için başvuru kaynağı.

4.2.15 Kaynaklar

- Oracle, *The Java Tutorials*: "Controlling Access to Members of a Class", "Using the this Keyword", "Packages", "Interfaces".
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21 Edition (6.6 Access Control, 7 Packages and Modules, 7.5 Import Declarations, 15.12.2 metot seçimi).
- P. Deitel ve H. Deitel, *Java: How to Program, Early Objects*, 11. baskı, Pearson, 2017 (Bölüm 3 ve 8: Classes and Objects).
- K. Sierra ve B. Bates, *Head First Java*, 3. baskı, O'Reilly, 2022 (Bölüm 4: encapsulation).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, 2022 (Bölüm 4: Objects and Classes).
- J. Bloch, *Effective Java*, 3. baskı, Addison-Wesley, 2018 (Madde 15–16: erişimi en aza indirmek, `public` alan yerine erişim metodu).
- A. B. Downey ve C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019 (Bölüm 11: Designing Classes).

KISIM II · BÖLÜM 4.3

Kalıtım

Bir yazılımda birbirine çok benzeyen sınıflar sık görülür. Öğrenci ile öğretmenin adı ve yaşı vardır; kedi, köpek ve kuş yer, uyur ve ses çıkarır. Bu ortak kısmı her sınıfta yeniden yazmak hem emek ister hem de hataya kapı açar: bir yerde yapılan düzeltme öbür yerde unutulur. Kalıtım bu sorunu çözer. Ortak alan ve metotlar bir kez, bir üst sınıfta yazılır; alt sınıflar onları kalıtımla alır ve yalnız kendi farklarını ekler. Not önce kalıtımın ne olduğunu, `extends` söz dizimini ve alt sınıfa nelerin geçtiğini anlatır. Ardından `super` anahtar sözcüğünü, yapıcı zincirini ve ilklendirme sırasını adım adım izler. Metot geçersiz kılma ve kuralları, `protected` erişim ve `final` ile kalıtımı sınırlama bunu izler. Son kesimler sınıflar arasındaki is-a ve has-a ilişkilerini, kalıtımın tasarım tuzaklarını ve her sınıfın atası olan `Object` sınıfının `toString`, `equals` ve `hashCode` metotlarını işler.

ÖĞRENME ÇIKTILARI

Kavram
Kalıtımı, üst sınıf ve alt sınıf kavramlarını tanımlamak; is-a ilişkisini tanımak.

Söz dizimi
`extends` ile alt sınıf yazmak; neyin kalıtımla alındığını, neyin alınmadığını açıklamak.

Yapıcılar
`super(...)` ve `super.metot()` kullanmak; yapıcı zincirinin ve ilklendirmenin çalışma sırasını izlemek.

Geçersiz kılma
Metodu geçersiz kılmak, aşırı yüklemeden ayırmak, `@Override` kullanmak ve kuralların gerekçesini açıklamak.

Erişim ve sınırlama
`protected` erişimini ve `final` ile kalıtımın nasıl sınırlandığını açıklamak.

Tasarım
Kalıtım ile oluşum arasında seçim yapmak; kırılgan üst sınıf ve yerine koyma ilkesi sorunlarını tanımak.

Object sınıfı
`toString`, `equals` ve `hashCode` metotlarını doğru yazmak; `==` ile `equals` farkını bilmek.

4.3.1 Kalıtım kavramı

Sınıf, aynı yapıdaki nesnelerin kalıbıdır: hangi verileri tuttuklarını ve hangi işleri yaptıklarını söyler. Gerçek bir yazılımda sınıflar birbirinden bağımsız durmaz. Bir kısmı ötekilerin daha özel bir biçimidir. Her dikdörtgen bir şekildir, her öğrenci bir kişidir, her kedi bir hayvandır. Kalıtım, sınıflar arasındaki bu "daha özel biçimi" ilişkisini koda döker. Bu kesim kalıtımı tanımlar, Java'daki söz dizimini verir, kalıtımla neyin alındığını ve bir alt sınıf nesnesinin bellekte nasıl durduğunu anlatır.

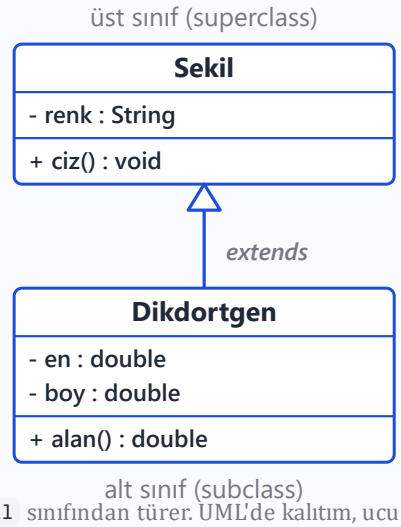
4.3.1.1 Kalıtım nedir?

Tanım 4.3.1

Kalıtım (*inheritance*), var olan bir sınıfın alan ve metotlarını alıp onları yeni ya da değiştirilmiş yeteneklerle zenginleştirerek yeni bir sınıf oluşturmaktır. Var olan sınıfa **üst sınıf** (*superclass*), yeni sınıfa **alt sınıf** (*subclass*) denir.

Tanımdaki iki fiil önemlidir: *almak* ve *zenginleştirmek*. Alt sınıf, üst sınıfın alan ve metotlarını yeniden yazmadan alır. Bu alınan üyeler alt sınıfta tekrar tanımlanmaz; alt sınıfın kendi üyeleri gibi kullanılır. Alt sınıf bunların üstüne yeni alanlar ve yeni metotlar ekleyebilir. Üst sınıftan aldığı bir metodun davranışını da değiştirebilir.

Gündelik dildeki bir cümle ilişkiyi iyi özetler: "Her dikdörtgen bir şekildir." Bir çizim programında her şeklin bir rengi vardır ve her şekil çizilebilir. Dikdörtgen bu ortak özelliklerin hepsini taşır. Üstüne kendi eni ve boyu vardır; alanı da bu ikisinden hesaplanır. Bu ilişki bir diyagramla da gösterilebilir. Şekil 4.3.1 onu UML (*Unified Modeling Language*, birleşik modelleme dili) sınıf diyagramıyla gösterir.



Şekil 4.3.1: `Dikdortgen` sınıfı `Sekil` sınıfından türer. UML'de kalıtım, ucu üst sınıfa dönük, içi boş üçgen okla çizilir. Alt sınıfın kutusunda yalnız kendi eklediği üyeler yazılır; `renk` ve `ciz()` kalıtımla gelir.

Diyagramda `Dikdortgen`'in kutusunda `renk` alanı ve `ciz()` metodu yoktur. Bunlar yazılmamıştır, ama her `Dikdortgen` nesnesinde bulunurlar. Kalıtım tam da budur: ortak kısım bir kez, üst sınıfta yazılır. Her alt sınıf yalnız kendi farkını taşır. Oku izleyen bir okuyucu "`Dikdortgen` bir `Sekil`'dir" cümlesini kurar. Okun yönü bu cümlenin yönüdür.

4.3.1.2 `extends` ile alt sınıf

Java'da bir sınıfın başka bir sınıftan türediği `extends` anahtar sözcüğüyle bildirilir. Söz dizimi `class AltSinif extends UstSinif` biçimindedir. `extends` İngilizcede "genişletir" demektir: alt sınıf, üst sınıfı genişletir.

```
// Temel sınıf (üst sınıf)
public class Sekil {
    // Temel sınıfa ait nitelik ve metotlar
    // ...
}

// Yeni sınıf (alt sınıf), Sekil'den türer
public class Dikdortgen extends Sekil {
    // Yeni sınıfa ait nitelik ve metotlar
    // ...
}
```

Bu iki sınıfın gövdesi henüz boştur, ama ilişki kurulmuştur. `Dikdortgen` türünden her nesne artık bir `Sekil`'dir. `Sekil`'e yazılacak her `public` ve `protected` üye `Dikdortgen`'de de kullanılabilir. Alt sınıfın kodu bu üyeleri kendi üyeleriymiş gibi, nesne adı ya da ön ek olmadan çağırır.

`extends` sözcüğünden sonra yalnız **bir** sınıf yazılabilir. Java sınıflarda çoklu kalıtıma izin vermez. Bir sınıfta `extends` hiç yazılmamışsa sınıfın yine de bir üst sınıfı vardır. Derleyici o sınıfı gizlice `java.lang.Object`'ten türetir. Şekil 4.3.2 bu iki adımlı zinciri gösterir.



Şekil 4.3.2: `Dikdortgen`, `Sekil`'i açıkça genişletir. `Sekil`'de `extends` yazılmadığı için derleyici onu `Object`'ten türetir. Zincir her zaman `Object`'te biter.

Alt sınıfta yapılmaması gereken bir şey de vardır. Üst sınıftaki bir alanla aynı adda bir alan alt sınıfta yeniden tanımlanırsa Java hata vermez. Ama bu iki alan aynı değişken değildir. Nesnede iki ayrı değişken oluşur; alt sınıfın kodu kendi alanını, üst sınıfın kodu kendininkini görür. Bu duruma **alan gizleme** (*field hiding*) denir. Hangi kodun hangi değeri okuduğunu izlemek zorlaşır ve hatalar sessizce ortaya çıkar. Kalıtımla gelen alan alt sınıfta yeniden tanımlanmaz.

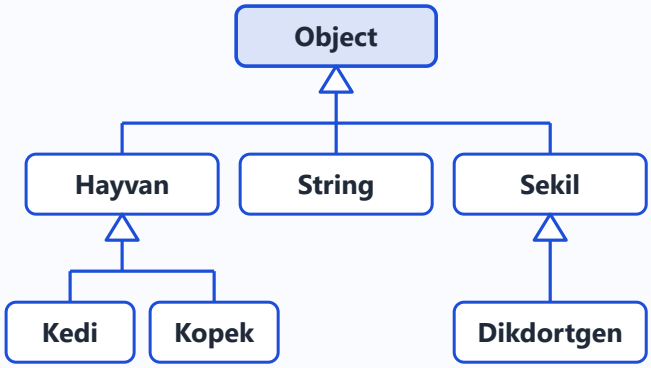
Sık yapılan hata

Üst sınıfta `String renk` varken alt sınıfa da `String renk` eklemek. Alt sınıfın yapıcısında `renk = "kırmızı"` yazılır; üst sınıfın `ciz()` metodu ise kendi `renk` alanını okur ve `null` görür. Doğrusu, alanı yalnız üst sınıfta tutmak ve değerini üst sınıfın yapıcısıyla vermektir.

4.3.1.3 Alt ve üst sınıf: özelleştirme

Alt sınıf, üst sınıfın bütün davranışlarını sergiler. Buna ek olarak kendine özgü davranışlar da sergileyebilir. Bir kedi bir hayvanın yaptığı her şeyi yapar: yer, uyur, nefes alır. Ayrıca miyavlar ve tırmanır. Bu yüzden kalıtıma **özelleştirme** (*specialization*) de denir. Alt sınıf, üst sınıfın temsil ettiği grubun içindeki daha özel bir grubu temsil eder.

İlişki tek yönlüdür. Her kedi bir hayvandır, ama her hayvan kedi değildir. Kod düzeyinde bunun karşılığı açıktır. `Hayvan` tipinde bir değişkene bir `Kedi` nesnesi atanabilir, çünkü bir kedi hayvandan beklenen her şeyi yapar. Tersi ise olmaz. Bir `Kedi` değişkenine sıradan bir `Hayvan` nesnesi atanamaz; o nesnenin miyavlayabileceği bilinmez.



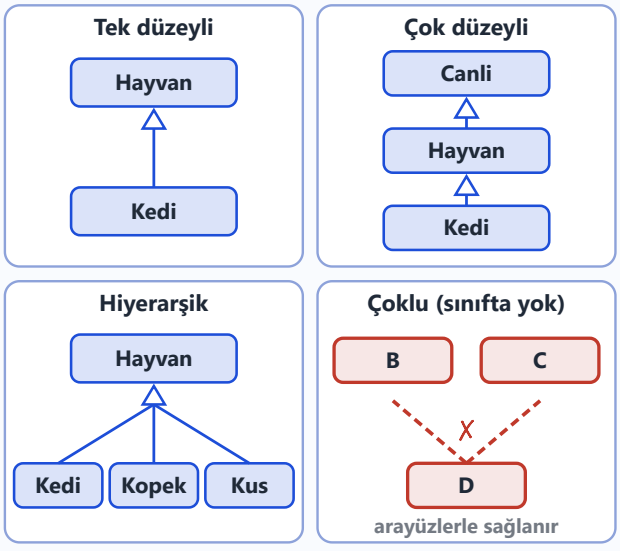
yukarı: genel · aşağı: özel

Şekil 4.3.3: Java sınıf hiyerarşisinin küçük bir parçası. En tepede `Object` vardır. Yukarı çıkıldıkça sınıflar genelleşir, aşağı inildikçe özelleşir.

Java'nın sınıf hiyerarşisi `java.lang` paketindeki `Object` sınıfıyla başlar. Şekil 4.3.3 bu hiyerarşiden bir kesit verir. `Hayvan`, `String` ve `Sekil` doğrudan `Object`'ten türer; `Kedi` ve `Kopek` ise `Hayvan` üzerinden dolaylı olarak türer. Her sınıf, doğrudan ya da dolaylı olarak `Object`'ten kalıtım alır. Bu yüzden her nesnede `Object`'in metotları hazır bulunur.

4.3.1.4 Kalıtım türleri

Sınıflar arasındaki kalıtım ilişkileri dört biçimde düzenlenebilir. Şekil 4.3.4 dördünü yan yana gösterir.



Şekil 4.3.4: Kalıtım türleri. Tek düzeyli, çok düzeyli ve hiyerarşik kalıtım Java'da vardır. Bir sınıfın iki üst sınıfı olduğu çoklu kalıtım sınıflarda yoktur; aynı ihtiyaç arayüzlerle karşılanır.

- **Tek düzeyli** (*single*) kalıtımda bir alt sınıf ve onun bir üst sınıfı vardır: `Kedi extends Hayvan`.

- **Çok düzeyli** (*multilevel*) kalıtımda sınıflar bir zincir oluşturur: `Hayvan extends Canli` ve `Kedi extends Hayvan`. `Kedi`, `Canli`'nin üyelerini `Hayvan` üzerinden alır. Her düzey bir öncekinin her şeyini taşır.
- **Hiyerarşik** (*hierarchical*) kalıtımda bir üst sınıftan birçok alt sınıf türer: `Kedi`, `Kopek` ve `Kus` üçü de `Hayvan`'dan türer. Ortak kod bir kez yazılır; her alt sınıf yalnız kendi farkını ekler.
- **Çoklu** (*multiple*) kalıtımda bir sınıfın birden çok üst sınıfı olur. Java sınıflarda buna izin vermez. Aynı ihtiyaç arayüzlerle karşılanır: bir sınıf tek bir sınıfı genişletirken birçok arayüzü uygulayabilir.

Bu kuralların sonucu olarak Java'daki sınıf hiyerarşisi bir **ağaç**tır. Ağacın kökü `Object`'tir. `Object` dışındaki her sınıfın tam olarak bir doğrudan üst sınıfı vardır. Zincirin derinliği için bir sınır yoktur. Ama derin zincirler okumayı ve değiştirmeyi zorlaştırır: bir sınıfın davranışını anlamak için yukarıdaki bütün sınıflara bakmak gerekir. Uygulamada iki üç düzeyden derin hiyerarşiler ancak iyi bir gerekçeyle kurulur.

4.3.1.5 Neler kalıtımla alınır?

"Alt sınıf, üst sınıfın her şeyini alır" cümlesi kısmen doğrudur. Doğru olan kısmı şudur: alt sınıf *nesnesinin içinde* üst sınıfın bütün alanları vardır. Ama bu alanlara ve metotlara alt sınıfın kodundan doğrudan erişilip erişilemeyeceğini erişim belirleyicisi belirler. Yapıcılar ise hiç kalıtımla alınmaz. Tablo 4.3.1 durumu üye türüne göre özetler.

Üye	Alt sınıfa geçer mi?	Not
`public` / `protected` alan ve metotlar	Evet	Alt sınıfta doğrudan kullanılır.
Belirleyicisiz (paket erişimli) üyeler	Yalnız aynı paketteyse	Başka paketteki alt sınıf erişemez.
`private` alanlar	Nesnede yer alır, ama doğrudan erişilemez	Üst sınıfın `public` / `protected` metotlarıyla (getter, setter) kullanılır.
`private` metotlar	Hayır	Görünmez; geçersiz kılınamaz.
Yapıcılar (*constructors*)	Hayır	Alt sınıf yapıcısı üst sınıfınkini `super(...)` ile çağırır.
`static` üyeler	Erişilebilir	Sınıfa aittir; geçersiz kılınmaz, yalnız gizlenebilir.

Tablo 4.3.1: Üst sınıfın üyeleri alt sınıfa geçer mi?

Tablonun en çok karıştırılan satırı `private` alanlarınkidir. Bir `Hayvan`'ın `private double saglik` alanı olsun. `Kedi extends Hayvan` yazıldığında her `Kedi` nesnesinde bir `saglik` alanı bulunur. Kedi uyuduğunda sağlığı artar, değer bu alanda saklanır. Ama `Kedi` sınıfının kodunda `saglik += 5` yazılırsa derleyici `saglik has private access in Hayvan` iletisiyle hata verir. Alan oradadır, ama adıyla erişim yalnız `Hayvan`'ın kendi koduna açıktır. `Kedi` o alana `Hayvan`'ın sunduğu metotlar üzerinden ulaşır.

Yapıcıların kalıtımla alınmamasının da mantıklı bir nedeni vardır. Yapıcının adı sınıfın adıdır. `Hayvan(String isim)` yapıcısı `Kedi`'ye geçseydi `new Kedi("Mavi")` çağrısı yalnız `Hayvan` kısmını kurar, `Kedi`'nin kendi alanları kurulmadan kalırdı. Bu yüzden her sınıf kendi yapıcısını yazar. Alt sınıfın yapıcısı, üst sınıf kısmının kurulması için üst sınıfın yapıcısını `super(...)` ile çağırır.

Örnek 4.3.1

`Hayvan` sınıfında şu üyeler olsun: `private String isim`, `protected int yas`, paket erişimli `void yemekYe()`, `public String getIsim()` ve `private void sindir()`. `Kedi extends Hayvan` başka bir pakette yazılıyor.

`Kedi`'nin kodunda `yas` (`protected`) ve `getIsim()` (`public`) doğrudan kullanılır. `isim` nesnede vardır, ama ona yalnız `getIsim()` ile ulaşılır. `yemekYe()` paket erişimlidir ve `Kedi` başka pakette olduğu için görünmez. `sindir()` ise `private`'tır, hiç görünmez. `Kedi` aynı paketteki bir sınıf olsaydı `yemekYe()` de kullanılabilirdi.

4.3.1.6 Alt sınıf nesnesinin yapısı

Kalıtımı ilk öğrenenler çoğu zaman şöyle bir resim kurar: `new Kedi(...)` yazılınca bir `Kedi` nesnesi ve onun içinde ya da yanında bir `Hayvan` nesnesi oluşur. Bu resim yanlıştır. `new Kedi(...)` **tek bir nesne** oluşturur. Üst sınıf için ayrı bir nesne oluşmaz.



Şekil 4.3.5: Bir `Kedi` nesnesi. `Object`, `Hayvan` ve `Kedi` kısımları aynı nesnenin içindedir. Yığındaki `k` değişkeni bu tek nesneyi gösterir. `this` ve `super` aynı nesneye bakar.

Şekil 4.3.5 nesnenin içini gösterir. Nesne katmanlardan oluşur. En içte `Object`'ten gelen kısım durur. Onun üstünde `Hayvan`'ın alanları (`isim`, `saglik`), en üstte de `Kedi`'nin kendi alanı (`evcil`) yer alır. `Hayvan`'ın alanları `private` olsa bile bu nesnenin içindedir. Yalnız `Kedi`'nin kodu onlara adıyla erişemez.

`super` anahtar sözcüğü bu resimde ayrı bir nesneyi göstermez. `super`, aynı nesnenin üst sınıf kısmına üst sınıfın kodu üzerinden ulaşmanın yoludur. `this` ile `super` aynı adresi taşır; fark, hangi sınıfın metodunun çağrılacağındadır. Bu ayrım yapıcı zincirini ve geçersiz kılmayı anlamanın anahtarıdır.

Bunun bir kanıtı vardır. `Object`'in `getClass()` metodu, nesnenin çalışma zamanındaki gerçek sınıfını döndürür. Aşağıdaki programda `getClass()` çağrısı `Hayvan`'ın yapıcısının içindedir:

```
class Hayvan {
    Hayvan() {
        System.out.println("Hayvan() içinde: "
            + getClass().getSimpleName());
    }
}
class Kedi extends Hayvan { }
// main içinde:
new Kedi();
new Hayvan();
```

```
Hayvan() içinde: Kedi
Hayvan() içinde: Hayvan
```

İlk satır şaşırtıcı görünebilir. Çalışan kod `Hayvan`'ın yapıcısıdır, ama nesnenin sınıfı `Kedi` çıkar. Nedeni şudur: `new Kedi()` önce bir `Kedi` nesnesi için yer ayırır, sonra bu nesnenin parçalarını sırayla kurar. `Hayvan()` çalışırken kurduğu şey bir `Kedi`'nin `Hayvan` kısmıdır. Ayrı bir `Hayvan` nesnesi yoktur. İkinci satırda ise `new Hayvan()` gerçekten bir `Hayvan` nesnesi oluşturur. `Kedi` sınıfında yapıcı yazılmamıştır; derleyici ona parametresiz bir varsayılan yapıcı ekler ve bu yapıcı `Hayvan()`'ı çağırır.

4.3.1.7 Ortak kodu üst sınıfa taşımak

Kalıtım çoğu zaman tasarımın başında kurulmaz. Önce birkaç sınıf yazılır, sonra aralarında tekrar eden kod fark edilir. Bu tekrarı bulup ortak bir üst sınıfa taşımaya **genelleştirme** (*generalization*) denir. Genelleştirme, özelleştirmenin ters yönüdür. Özelleştirmede genel bir sınıftan özel bir sınıfa inilir; genelleştirmede özel sınıflardan ortak bir genel sınıfa çıkılır.

Bir okul yönetim programında başlangıçta şu iki sınıf olsun:

```
class Ogrenci {
    private String ad;
    private int yas;
    private double ortalama;
    String kimlik() { return ad + " (" + yas + ")"; }
}
class Ogretmen {
    private String ad;
    private int yas;
    private String brans;
    String kimlik() { return ad + " (" + yas + ")"; }
}
```

İki sınıfta `ad` ve `yas` alanları da `kimlik()` metodu da birebir aynıdır. Bu tekrarın bedeli değişiklik anında ödenir. Yaşın gösterim biçimi değişsin; örneğin "(20)" yerine "20 yaş" yazılsın. İki sınıfın da düzeltilmesi gerekir. Biri unutulursa program aynı bilgiyi iki farklı biçimde yazar. Sınıf sayısı arttıkça unutma olasılığı da artar.

Ortak kısım bir `Kisi` sınıfına taşınınca kod şöyle olur:

```
class Kisi {
    private String ad;
    private int yas;
    Kisi(String ad, int yas) { this.ad = ad; this.yas = yas; }
    String kimlik() { return ad + " (" + yas + ")"; }
}
class Ogrenci extends Kisi {
    private double ortalama;
    Ogrenci(String ad, int yas, double ortalama) {
        super(ad, yas);
        this.ortalama = ortalama;
    }
    @Override String kimlik() {
        return super.kimlik() + ", ort. " + ortalama;
    }
}   // Ogretmen de aynı biçimde: brans eklenir
```

`Ogrenci` artık yalnız kendi farkını taşır: `ortalama` alanını. Adı ve yaşı kendisi saklamaz; yapıcısında bunları `super(ad, yas)` ile `Kisi`'nin yapıcısına verir. `kimlik()` metodunu geçersiz kılar, ama ortak kısmı yeniden yazmaz. `super.kimlik()` ile `Kisi`'nin sürümünü çağırır ve sonuna ortalamayı ekler. `Ogretmen` de aynı biçimde yazılır: alanı `brans`'tır ve `kimlik()`'in sonuna branşı ekler. Program çalışınca iki satır yazılır:

```
Ece (20), ort. 3.4
Murat (41), Fizik
```

Artık yaşın gösterim biçimi tek bir yerde, `Kisi.kimlik()`'te değişir; iki alt sınıf da yeni biçimi kendiliğinden kullanır.

Genelleştirmeden önce sorulacak bir soru vardır: "Her öğrenci bir kişi midir?" Cevap evetse genelleştirme doğrudur. İki sınıfın yalnız birkaç alanı benziyorsa ve aralarında "bir türüdür" ilişkisi yoksa kalıtım yanlış araçtır. Örneğin bir `Ogrenci` ile bir `Ders` sınıfının ikisinde de `ad` alanı olabilir. Ama bir ders bir öğrenci değildir. Böyle durumlarda ortak kod ayrı bir yardımcı sınıfa konur ve oluşumla kullanılır.

Örnek 4.3.2

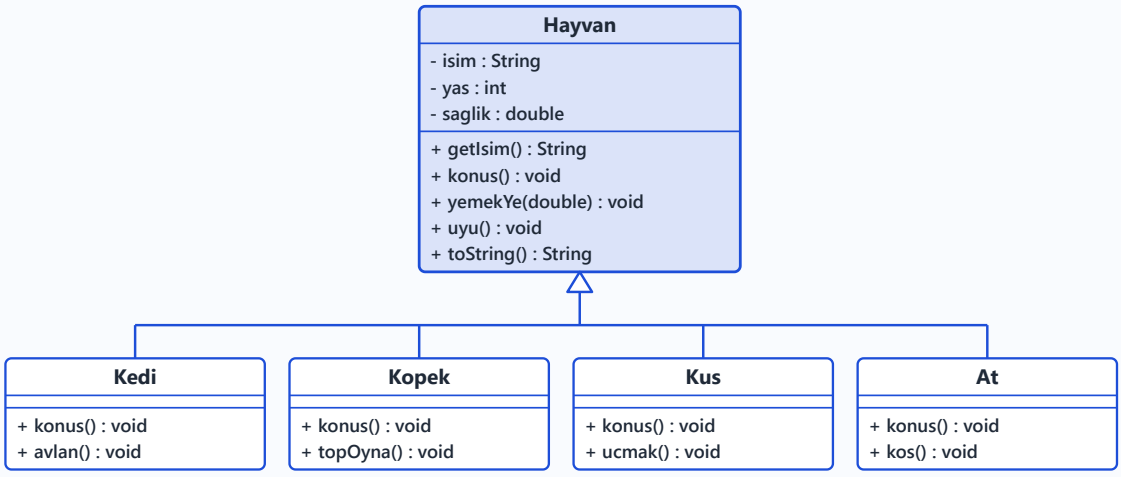
Bir elektrik devresi benzetim programında `Direnc`, `Kondansator` ve `Bobin` sınıfları olsun. Üçünde de iki düğüm numarası (`dugumA`, `dugumB`), bir ad ve `bagla(int a, int b)` metodu tekrar ediyor. Her birinin kendine özgü değeri ise farklı: direnç, sığa ve endüktans.

"Her direnç bir devre elemanı mıdır?" sorusunun cevabı evettir; diğer ikisi için de öyledir. Ortak alanlar ve `bagla` metodu bir `DevreElemani` üst sınıfına taşınır. Her alt sınıfta yalnız kendi değeri ve o değere bağlı hesap kalır; örneğin bir `empedans(double frekans)` metodu. Düğüm numaralarının denetimi değişirse yalnız `DevreElemani` düzeltilir.

4.3.2 Hayvan örneği

Kalıtımın parçaları tek bir örnek üzerinde birlikte görüldüğünde daha kolay anlaşılır. Bu kesim bir hayvan hiyerarşisi kurar. Ortak üyeler bir üst sınıfta toplanır, alt sınıflar bir davranışı değiştirir ve kendilerine özgü davranış ekler.

4.3.2.1 Hayvan sınıf hiyerarşisi



Şekil 4.3.6: `Kedi`, `Kopek`, `Kus` ve `At` sınıfları `Hayvan`'dan türer. Her biri `konus()` metodunu kendine göre geçersiz kılar ve kendine özgü bir metot ekler. `-` işareti `private`, `+` işareti `public` üyeyi gösterir.

Şekil 4.3.6 hiyerarşik kalıtımın tipik bir örneğidir. `Hayvan` sınıfı bütün hayvanlarda ortak olan alanları taşır: `isim`, `yas`, `saglik`. Ortak davranışlar da onda yazılıdır: `konus`, `yemekYe`, `uyu`, `toString`. Bu üyeler bir kez, `Hayvan`'da yazılır. Dört alt sınıfın kutusunda ise yalnız iki metot vardır. Biri `konus()`'tur; her hayvan farklı ses çıkardığı için her alt sınıf bu metodu kendine göre yeniden yazar. Öteki o hayvana özgü bir davranıştır: kedi avlanır, köpek top oynar, kuş uçar, at koşar.

Diyagramdan bir sonuç daha çıkar. Yeni bir hayvan türü eklemek için `Hayvan`'a dokunmak gerekmez. Örneğin bir `Inek` sınıfı `Hayvan`'dan türetilir, `konus()`'u geçersiz kılar ve bir `sagil()` metodu ekler. Uyuma, yemek yeme ve sağlık takibi kendiliğinden gelir.

4.3.2.2 `Hayvan` ve `Kopek` sınıfları

Hiyerarşinin kodu iki sınıfla başlar. Üst sınıf `Hayvan`, bütün hayvanlarda ortak olan alanları ve davranışları içerir:

```
public class Hayvan {
    private String isim;     // ortak nitelikler
    private int yas;
    private double saglik;

    public Hayvan(String isim, int yas) {
        this.isim = isim;
        this.yas = yas;
        this.saglik = 100.0;
    }
    public String getIsim() { return isim; }   // private isim'e erişim yolu
    public void konus() {
        System.out.println("Bu hayvan konuşur.");
    }
    public void uyu() {
        saglik += 5;
        System.out.println(isim + " uyudu. Sağlık: " + saglik);
    }
}
```

Üç alan da `private`'tır. Dışarıdan ve alt sınıflardan doğrudan erişilemezler. `isim`'e erişim yolu `getIsim()` metodudur. Yapıcı iki değer alır ve sağlığı 100 ile başlatır. `konus()` genel bir ileti yazar; hangi hayvanın nasıl konuştuğu henüz bilinmez. `uyu()` sağlığı 5 artırır ve durumu yazar.

Alt sınıf `Kopek` yalnız farkı yazar:

```
public class Kopek extends Hayvan {
    public Kopek(String isim, int yas) {
        super(isim, yas);   // üst sınıf yapıcısı alanları doldurur
    }
    @Override
    public void konus() {   // miras alınan davranışı değiştirir
        System.out.println(getIsim() + " havlar.");
    }
    public void topOyna() { // yalnızca Kopek'te olan davranış
        System.out.println(getIsim() + " top oynuyor!");
    }
}
```

Sınıf üç şey yapar. Yapıcısı, aldığı adı ve yaşı `super(isim, yas)` ile `Hayvan`'ın yapıcısına verir; alanları `Hayvan` doldurur. `konus()` metodunu geçersiz kılar: `@Override` etiketi, bu metodun üst sınıftan gelen bir metodu yeniden yazdığını bildirir. Son olarak yalnız köpeklerde olan `topOyna()` metodunu ekler. Dikkat edilirse `Kopek`'in kodu `isim` alanına doğrudan dokunmaz; adı `getIsim()` ile alır. `isim` `private` olduğu için başka yol da yoktur.

```
Kopek kopek = new Kopek("Karabaş", 4);
kopek.konus();     // Kopek'in sürümü
kopek.topOyna();   // Kopek'e özgü
kopek.uyu();       // Hayvan'dan miras
```

```
Karabaş havlar.
Karabaş top oynuyor!
Karabaş uyudu. Sağlık: 105.0
```

Üç çağrı üç farklı durumu gösterir. `kopek.konus()` `Kopek`'in geçersiz kıldığı sürümü çalıştırır; "Bu hayvan konuşur." yazılmaz. `kopek.topOyna()` yalnız `Kopek`'te olan metottur. `kopek.uyu()` ise `Kopek`'te hiç yazılmamıştır. Java bu metodu önce `Kopek`'te arar, bulamayınca bir üst sınıfa, `Hayvan`'a çıkar ve orada bulur. Metot `Hayvan`'ın kodudur, ama bu köpek nesnesinin `saglik` alanını 100'den 105'e çıkarır.

Metot arama kuralı kalıtımın temel işleyişidir. Bir nesne üzerinde bir metot çağrıldığında arama nesnenin *gerçek* sınıfından başlar. Metot orada yoksa hiyerarşide yukarı çıkılır. İlk bulunan sürüm çalışır. `Object`'e kadar çıkıp bulunamayan bir metodu ise derleyici zaten kabul etmez.

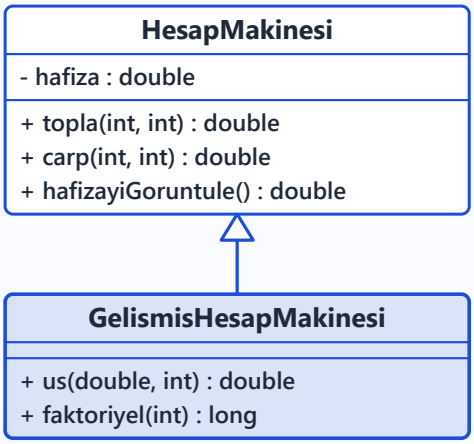
4.3.2.3 Kalıtımın avantajları

Kalıtımın üç yararı vardır.

- **Yeniden kullanım.** Üst sınıfın metot ve alanları alt sınıfta yeniden yazılmadan kullanılır. Ortak kod tek yerde durur; bir düzeltme bütün alt sınıflara yansır.
- **Genişletme.** Var olan bir sınıf, kodu değiştirilmeden yeni yeteneklerle genişletilir. Üst sınıf başka bir ekibin ya da bir kütüphanenin olabilir; kaynak koduna dokunmak gerekmez.
- **Değiştirme.** Üst sınıfın metotları alt sınıfta geçersiz kılınarak (*overriding*) değiştirilebilir. Alt sınıf, ortak davranışın yalnız kendisine uymayan kısmını yeniden yazar.

Genişletmeye bir hesap makinesi örneği verilebilir. Elimizde dört işlem yapan, son sonucu bir hafıza alanında tutan bir `HesapMakinesi` sınıfı olsun. Sunumda yalnız diyagramı verilen bu sınıf şöyle yazılabilir:

```
class HesapMakinesi {
    private double hafiza;
    public double topla(int a, int b) { hafiza = a + b; return hafiza; }
    public double carp(int a, int b) { hafiza = a * b; return hafiza; }
    public double hafizayiGoruntule() { return hafiza; }
}
```



Şekil 4.3.7: `GelismisHesapMakinesi`, `HesapMakinesi`'nden türer ve `us` ile `faktoriyel` metotlarını ekler. Üst sınıfın kodu değişmez.

Bilimsel hesap gereken bir kullanıcı için üs alma ve faktöriyel eklenecek olsun. Şekil 4.3.7 çözümü gösterir. Yeni sınıf eskisini genişletir; dört işlem ve hafıza kendiliğinden gelir:

```
public class GelismisHesapMakinesi
        extends HesapMakinesi {
    // Yeni yetenek: üs alma
    public double us(double taban, int us) {
        double sonuc = 1.0;
        for (int i = 0; i < us; i++) {
            sonuc *= taban;
        }
        return sonuc;
    }
    // faktoriyel(int) benzer biçimde eklenir
}
// main içinde
GelismisHesapMakinesi h = new GelismisHesapMakinesi();
System.out.println(h.topla(5, 3));         // 8.0  (miras)
System.out.println(h.hafizayiGoruntule()); // 8.0  (miras)
System.out.println(h.us(2, 5));            // 32.0 (yeni)
```

```
8.0
8.0
32.0
```

`h.topla(5, 3)` ve `h.hafizayiGoruntule()` çağrıları `GelismisHesapMakinesi`'nde yazılmamıştır; ikisi de kalıtımla gelir ve 8.0 verir. `us(2, 5)` yeni metottur. Döngü beş tur döner ve `sonuc` sırasıyla 2, 4, 8, 16, 32 olur. `us` parametresinin metodun adıyla aynı olması Java'da sorun değildir. Metot adları ile değişken adları ayrı ad alanlarında durur. Okunurluk için yine de farklı adlar seçmek daha iyidir.

Maliyet açısından bakıldığında kalıtımla gelen bir metodu çağırmak, sınıfın kendi metodunu çağırmakla aynı maliyettedir. Java sanal makinesi her sınıf için bir metot tablosu tutar. Arama, tablodaki bir girişe bakmaktan ibarettir; hiyerarşinin derinliği her çağrıda yeniden taranmaz. Yani kalıtımın asıl bedeli çalışma süresinde değil, tasarımda ödenir. Bu bedel, sınıflar arasındaki sıkı bağlılıktır.

4.3.3 Yapıcılar ve `super`

Bir alt sınıf nesnesi kurulurken üst sınıf kısmı da kurulmalıdır. Üst sınıfın alanlarına ilk değer verme işi ise üst sınıfa aittir. Bu kesim alt sınıf ile üst sınıf arasındaki bu işbirliğinin aracını, `super` anahtar sözcüğünü anlatır. Ardından yapıcıların hangi sırayla çalıştığını üç ayrı kod izlemesiyle adım adım gösterir.

4.3.3.1 `super` anahtar sözcüğü

> **Tanım 4.3.2**
>
> `super`, alt sınıfın kodu içinde üst sınıfı gösteren anahtar sözcüktür. İki biçimde kullanılır. `super(...)` alt sınıfın yapıcısı içinde üst sınıfın yapıcısını çağırır. `super.metot()` ise üst sınıftaki bir metodun sürümünü çağırır; genellikle geçersiz kılınmış bir metodun içinden kullanılır.

`super(...)` ile ilgili iki kural vardır. Birincisi, `super(...)` yapıcının **ilk deyimi** olmalıdır. Bu kural Java 21 dahil bütün yaygın sürümlerde geçerlidir. Java 25 kuralı biraz gevşetmiştir: `super(...)`'dan önce, kurulmakta olan nesneye dokunmayan deyimler yazılabilir. Örneğin parametreler denetlenebilir. Ama bu not Java 21'i esas alır. İkincisi, yapıcıda `super(...)` ya da `this(...)` yazılmazsa derleyici gizlice `super();` ekler. Yani bir üst sınıf yapıcısı *her zaman* çağrılır; programcının tek seçimi hangisinin çağrılacağıdır.

Bu kuralların arkasında bir tasarım ilkesi yatar: her sınıf yalnız kendi verilerini yönetir. `Kedi`, `Hayvan`'ın `isim` alanına kendisi atama yapmaz; işi `super(isim)`'e bırakır. `Hayvan` kendi alanlarının nasıl ilk değer alacağını en iyi bilen sınıftır. Belki adın boş olmadığını denetler, belki sağlığı 100'den başlatır. Alt sınıf bu kuralları bilmek zorunda kalmaz.

```
class Kedi extends Hayvan {
    private boolean evcil;

    Kedi(String isim, boolean evcil) {
        super(isim);         // Hayvan(String)
        this.evcil = evcil;  // kendi verisi
    }

    @Override
    public String toString() {
        return super.toString()   // Hayvan'ınki
               + ", Evcil: " + evcil;
    }
}
// main içinde
System.out.println(new Kedi("Mavi", true));
```

```
İsim: Mavi, Sağlık: 100.0, Evcil: true
```

Burada `Hayvan(String isim)` yapıcısı adı alanına yazar ve sağlığı 100.0 yapar. `Hayvan`'ın `toString()` metodu da ad ve sağlık bilgisini "İsim: Mavi, Sağlık: 100.0" biçiminde döndürür. `Kedi`'nin yapıcısı ilk satırda `super(isim)` ile bu yapıcıyı çağırır, sonra kendi alanı `evcil`'i doldurur. `Kedi`'nin `toString()`'i ise üst sınıfın ürettiği metni yeniden üretmez. `super.toString()` ile o metni alır ve sonuna yalnız kendi alanını ekler. `println`, kendisine verilen nesnenin `toString()` metodunu çağırır. Nesne bir `Kedi` olduğu için `Kedi`'nin sürümü çalışır.

Bu kalıp çok yaygındır: geçersiz kılınan metot önce `super` ile üst sınıfın işini yaptırır, sonra kendi farkını ekler. Böylece üst sınıfın davranışı değiştiğinde, örneğin `Hayvan.toString()` yaşı da yazmaya başladığında, `Kedi`'nin çıktısı da kendiliğinden güncellenir.

4.3.3.2 Kod izleme: `super.tanit()` zinciri

`super.metot()` çağrıları zincir hâlinde de kullanılabilir. Üç düzeyli bir hiyerarşide her sınıf `tanit()` metodunu geçersiz kılsın ve her biri önce üst sınıfınkini çağırsın:

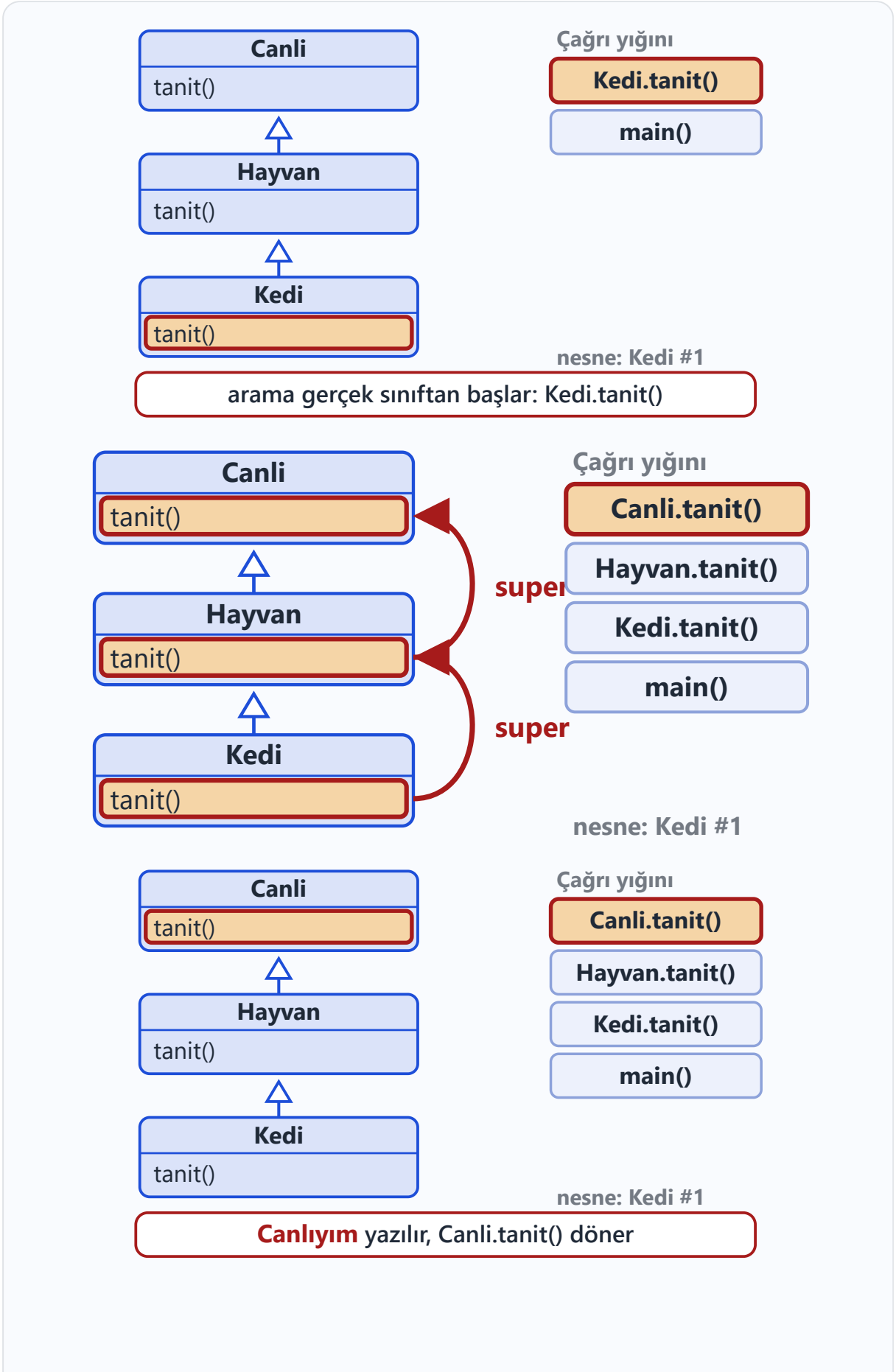
```
class Canli {
    void tanit() { System.out.println("Canlıyım"); }
}
class Hayvan extends Canli {
    void tanit() {
        super.tanit();
        System.out.println("Hayvanım");
    }
}
class Kedi extends Hayvan {
    void tanit() {
        super.tanit();
        System.out.println("Kediyim");
    }
}
new Kedi().tanit();   // main
```

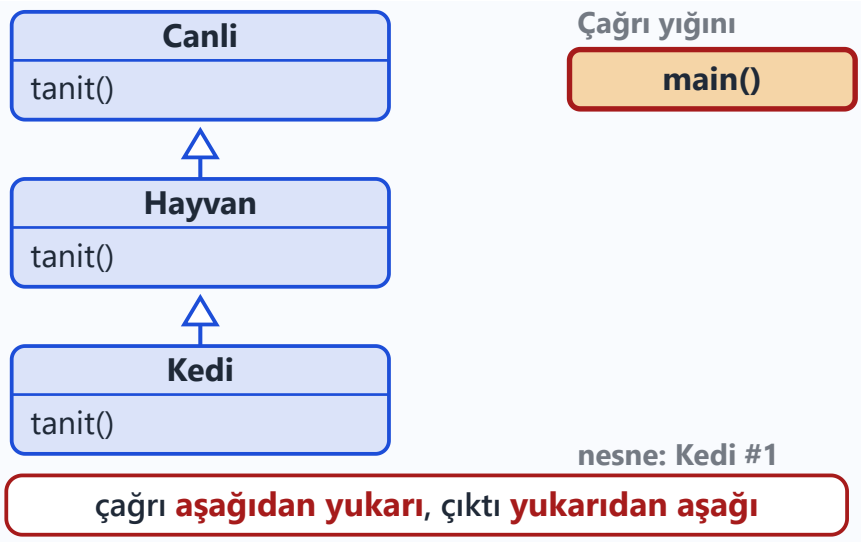
```
Canlıyım
Hayvanım
Kediyim
```

Çıktının sırası ilk bakışta tersmiş gibi görünür. Çağrı `Kedi`'den başlar, ama ilk yazılan satır `Canli`'ninkidir. Tablo 4.3.2 çağrıları adım adım izler.

Adım	Olay	Çağrı yığını	Çıktı
1	Nesne bir `Kedi`; arama en alttan başlar, `Kedi.tanit()` bulunur.	main, Kedi.tanit	—
2	`super.tanit()`: `Hayvan.tanit()` çağrılır. `Kedi.tanit` bekler.	main, Kedi.tanit, Hayvan.tanit	—
3	`Hayvan` içindeki `super` sabit olarak `Canli`'yi gösterir: `Canli.tanit()`.	main, Kedi.tanit, Hayvan.tanit, Canli.tanit	—
4	En üstteki sürüm yazar ve döner.	main, Kedi.tanit, Hayvan.tanit	Canlıyım
5	`Hayvan.tanit()` kaldığı yerden sürer, yazar ve döner.	main, Kedi.tanit	Hayvanım
6	`Kedi.tanit()` kendi satırını yazar ve döner.	main	Kediyim

Tablo 4.3.2: `new Kedi().tanit()` çağrısının izlemesi. Yığın sütununda en üstteki çerçeve sağdadır.


Canli
tanit()
Hayvan
tanit()
Kedi
tanit()
Çağrı yığını
Kedi.tanit()
main()
nesne: Kedi #1
arama gerçek sınıftan başlar: Kedi.tanit()
Canli
tanit()
Hayvan
tanit()
Kedi
tanit()
super
super
Çağrı yığını
Canli.tanit()
Hayvan.tanit()
Kedi.tanit()
main()
nesne: Kedi #1
Canli
tanit()
Hayvan
tanit()
Kedi
tanit()
Çağrı yığını
Canli.tanit()
Hayvan.tanit()
Kedi.tanit()
main()
nesne: Kedi #1
Canlıyım yazılır, Canli.tanit() döner



Şekil 4.3.8: `super.tanit()` zincirinden dört an. (a) `Kedi.tanit()` çalışır. (b) İki `super` çağrısıyla yığına `Hayvan.tanit` ve `Canli.tanit` eklenir. (c) `Canli` ilk satırı yazar. (d) Bütün çağrılar döndü; çıktılar ters sırada geldi.

Üçüncü adım ince bir noktayı taşır. Nesne hâlâ bir `Kedi`'dir. Normal bir metot çağrısında arama nesnenin gerçek sınıfından başlar. Bu kurala uyulsaydı `Hayvan`'ın içindeki çağrı yeniden `Kedi.tanit()`'e gider ve sonsuz bir döngü oluşurdu. `super` bu kurala uymaz. `Hayvan`'ın kodunda yazılan `super`, nesnenin sınıfı ne olursa olsun, her zaman `Hayvan`'ın üst sınıfını, yani `Canli`'yi gösterir. Bu bağ derleme anında kurulur; çalışma zamanında nesneye bakılarak seçilmez.

Şekil 4.3.8 zincirin genel biçimini gösterir. Çağrılar `Kedi` → `Hayvan` → `Canli` yönünde gider, çıktılar ters sırada gelir. Her sınıf önce üst sınıfın işini yaptırır, sonra kendi farkını ekler. Yığın en çok dört çerçeveye ulaşır: `main` ve üç `tanit`. Zincirin her düzeyi bir çağrı ve bir çerçeve ekler. Yani *d* düzeyli bir zincirin süresi ve yığın kullanımı *d* ile orantılıdır, O(d)'dir. `new Hayvan().tanit()` çağrılsaydı zincir bir düzey kısa olur ve yalnız “Canlıyım” ile “Hayvanım” yazılırdı.

4.3.3.3 Kod izleme: yapıcı zinciri

Yapıcılar da benzer bir zincir oluşturur. Aşağıdaki kodda `Hayvan`'ın yapıcısı `super()`'ı açıkça yazar. Yazılmasaydı derleyici aynı satırı kendisi ekleyecekti.

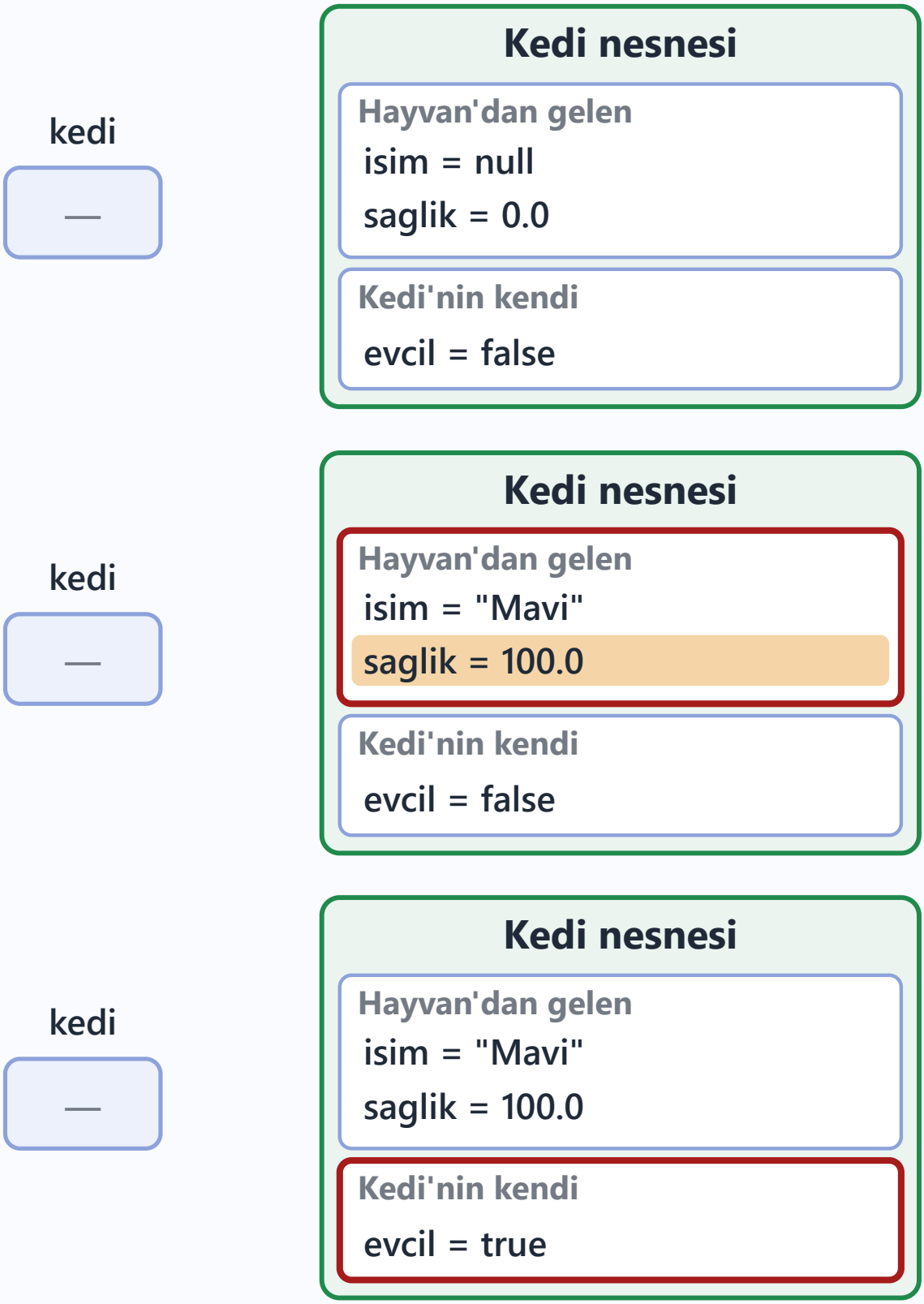
```
class Hayvan {
    private String isim;
    private double saglik;
    Hayvan(String isim) {
        super();   // Object() çağrılır
        this.isim = isim;
        saglik = 100.0;
        System.out.println("Hayvan yapıcısı");
    }
}
class Kedi extends Hayvan {
    private boolean evcil;
    Kedi(String isim, boolean evcil) {
        super(isim);
        this.evcil = evcil;
        System.out.println("Kedi yapıcısı");
    }
}
// main içinde
Kedi kedi = new Kedi("Mavi", true);
System.out.println("Nesne hazır");
```

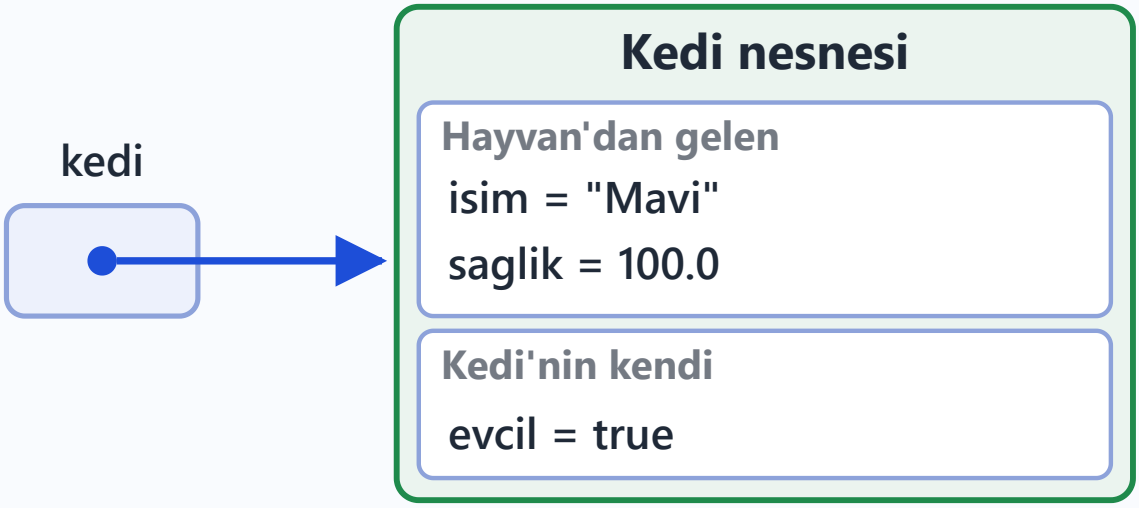
```
Hayvan yapıcısı
Kedi yapıcısı
Nesne hazır
```

`new Kedi("Mavi", true)` ifadesinin değerlendirilmesi on bir adımda incelenebilir. Tablo 4.3.3 bu adımları, nesnenin alanlarının o anki değerleriyle birlikte verir.

Adım	Çalışan kod	isim	saglik	evcil	Çıktı
1	`new` : nesneye yer ayrılır	null	0.0	false	
2	`Kedi(...)` çağrılır	null	0.0	false	
3	`super(isim)` : `Hayvan(...)` ; `Kedi` bekler	null	0.0	false	
4	`super()` : `Object()`	null	0.0	false	
5	`Object()` döndü; `this.isim = isim`	"Mavi"	0.0	false	
6	`saglik = 100.0`	"Mavi"	100.0	false	
7	`Hayvan` yazar ve döner	"Mavi"	100.0	false	Hayvan yapıcısı
8	`Kedi` sürer: `this.evcil = evcil`	"Mavi"	100.0	true	
9	`Kedi` yazar	"Mavi"	100.0	true	Kedi yapıcısı
10	Nesnenin adresi `kedi` 'ye atanır	"Mavi"	100.0	true	
11	`main` sürer	"Mavi"	100.0	true	Nesne hazır

Tablo 4.3.3: `new Kedi("Mavi", true)` izlemesi. Alanlar sırasıyla `isim` , `saglik` , `evcil` 'dir.


kedi
—
Kedi nesnesi
Hayvan'dan gelen
isim = null
saglik = 0.0
Kedi'nin kendi
evcil = false
kedi
—
Kedi nesnesi
Hayvan'dan gelen
isim = "Mavi"
saglik = 100.0
Kedi'nin kendi
evcil = false
kedi
—
Kedi nesnesi
Hayvan'dan gelen
isim = "Mavi"
saglik = 100.0
Kedi'nin kendi
evcil = true



Şekil 4.3.9: Yapıcı zincirinden dört an. (a) `new` nesneye yer ayırdı; bütün alanlar varsayılan değerde. (b) `Hayvan` kendi alanlarını doldurdu. (c) `Kedi` kendi alanını doldurdu. (d) Nesnenin adresi `kedi` değişkenine atandı.

İzlemenin ilk adımı önemlidir. `new`, yapıcı çalışmadan *önce* nesnenin bütün alanlarına yer ayırır. `Hayvan`'dan gelen `isim` ve `saglik` da bunlara dahildir. Bütün alanlar varsayılan değerlerini alır: referanslar `null`, sayılar 0, `boolean` `false`. Şekil 4.3.9'un ilk karesi bu durumu gösterir. Yapıcılar alan oluşturmaz, yalnız var olan alanlara değer yazar.

Sonra çağrılar aşağıdan yukarı gider: `Kedi` → `Hayvan` → `Object`. Her yapıcı ilk iş olarak bir üst sınıfın yapıcısını çağırdığı için hiçbiri gövdesine geçemeden bekler. Zincir her zaman `Object`'te biter. `Object()`'in üst sınıfı yoktur, gövdesi de boştur; hemen döner. Bundan sonra gövdeler yukarıdan aşağı tamamlanır: önce `Hayvan` kendi alanlarını doldurup iletisini yazar, sonra `Kedi`. Sonuç tek cümleyle söylenebilir: yapıcılar **yukarıdan aşağıya tamamlanır**, `Object` → `Hayvan` → `Kedi`.

Bu sıranın bir gerekçesi vardır. Alt sınıfın yapıcısı, üst sınıftan gelen alanları ve metotları kullanabilmelidir. Örneğin `Kedi`'nin yapıcısı `getIsim()`'i çağırıp bir ileti yazmak isteyebilir. Bunun güvenli olması için `Hayvan` kısmı o anda kurulmuş olmalıdır. Üst sınıfın önce tamamlanması bunu garanti eder.

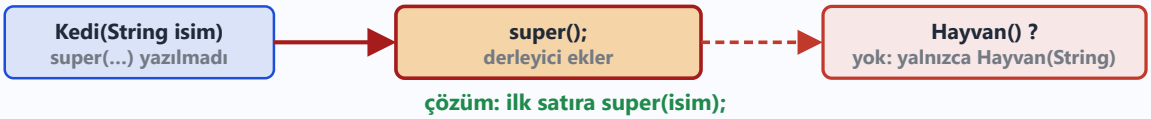
Son adımda nesne hazırdır ve adresi `kedi` değişkenine atanır. Yapıcı sürerken `kedi` değişkeni henüz bir değer taşımaz. Yarım kurulmuş bir nesne `main`'deki değişkene hiç ulaşmaz.

4.3.3.4 Gizli `super()` ve hatalar

Derleyicinin gizlice eklediği `super();` çağrısı çoğu zaman yararlıdır, ama yeni başlayanların en sık karşılaştığı derleme hatasının da kaynağıdır. Üst sınıfta parametresiz bir yapıcı yoksa gizli çağrının karşılığı bulunmaz:

```
class Hayvan {
    private String isim;
    Hayvan(String isim) { this.isim = isim; }
}
class Kedi extends Hayvan {
    Kedi(String isim) {
        // derleyici buraya super(); ekler
        System.out.println("Kedi");
    }
}
```

```
error: constructor Hayvan in class Hayvan
  cannot be applied to given types;
  required: String
  found:    no arguments
```



Şekil 4.3.10: `Kedi`'nin yapıcısında `super(...)` yazılmadığı için derleyici `super();` ekler. `Hayvan`'da yalnız `Hayvan(String)` olduğu için bu çağrının gideceği yapıcı yoktur.

İleti ilk bakışta kafa karıştırır. Programcı `Hayvan`'ın yapıcısını çağırmamıştır, ama derleyici bir çağrının yanlış argümanlarla yapıldığını söyler. İletinin söylediği şudur: `Hayvan`'ın yapıcısı bir `String` bekler (`required: String`), oysa çağrıda hiç argüman yoktur (`found: no arguments`). Argümansız çağrı, Şekil 4.3.10'da görüldüğü gibi, derleyicinin eklediği `super();`'dır.

Aynı hata, alt sınıfta hiç yapıcı yazılmadığında da çıkar:

```
class Hayvan {
    private String isim;
    Hayvan(String isim) { this.isim = isim; }
}
class Kedi extends Hayvan { }
```

```
Main.java:5: error: constructor Hayvan in class Hayvan cannot be applied to given types;
class Kedi extends Hayvan { }
^
  required: String
  found:    no arguments
```

Hiç yapıcı yazılmamış bir sınıfa derleyici parametresiz bir **varsayılan yapıcı** (*default constructor*) ekler. Bu yapıcının gövdesi yalnız `super();`'dır. Bu yüzden `Kedi` için eklenen yapıcı da parametresiz `Hayvan()`'ı arar ve bulamaz. Varsayılan yapıcı yalnız sınıfta *hiçbir* yapıcı yoksa eklenir. `Hayvan`'a `Hayvan(String)` yazıldığı anda onun parametresiz yapıcısı ortadan kalkmıştır.

İkinci tür hata `super(...)`'ın yerini yanlış seçmektir:

```
Kedi(String isim) {
    System.out.println("Kedi");
    super(isim);
}
```

```
error: call to super must be first
  statement in constructor
```

Java 21'de `super(...)` yapıcının ilk deyimi olmak zorundadır. Önüne bir `println` bile yazılamaz. Çözüm iki hatada da aynıdır: alt sınıf yapıcısının ilk satırında uygun üst sınıf yapıcısı açıkça çağrılır, yani `super(isim);` yazılır. Bir başka çözüm üst sınıfa parametresiz bir yapıcı eklemektir. Ama bu yol, `Hayvan`'ın adsız bir hayvan oluşturmaya izin vermesi demektir; bu çoğu zaman istenmez.

Sık yapılan hata

`constructor Hayvan in class Hayvan cannot be applied to given types` iletisini gören programcı genellikle `Kedi`'yi oluşturan `new` satırına bakar ve orada hata arar. Hata ise `Kedi` sınıfının yapıcısındadır: üst sınıfın parametreli yapıcısı çağrılmamıştır. İletinin satır numarası bunu gösterir; satır, `Kedi`'nin yapıcı bildirimidir.

Java 25 ile gelen değişiklik bu ikinci hatayı ortadan kaldırır. Java 25'te `super(isim)`'den önce `System.out.println("Kedi")` yazılabilir; bu deyim kurulmakta olan nesneye dokunmaz. Ama o deyimlerde de `this` kullanılamaz, nesnenin metotları çağrılamaz ve alanları okunamaz. Kuralın özü değişmez: üst sınıf kısmı kurulmadan nesne kullanılmaz. İlk hatayı ise hiçbir sürüm çözmez; üst sınıfın parametreli yapıcısı yine açıkça çağrılmalıdır.

4.3.3.5 Kod izleme: `this(...)` ve `super(...)`

Bir yapıcı, aynı sınıfın başka bir yapıcısını `this(...)` ile çağırabilir. Bu da **yapıcı zincir**inin (*constructor chaining*) bir biçimidir ve tekrar eden ilk değer verme kodunu tek yapıcıda toplar. `this(...)` da `super(...)` gibi ilk deyim olmalıdır. Bu yüzden bir yapıcı ya `this(...)` ile ya `super(...)` ile başlar, ikisiyle birden değil. Aşağıdaki kodda iki sınıfın ikisinde de iki yapıcı vardır:

```
class Calisan {
    private String ad;  private double maas;
    Calisan(String ad) {
        this(ad, 20000.0);
        System.out.println("Calisan(String)");
    }
    Calisan(String ad, double maas) {
        this.ad = ad;  this.maas = maas;
        System.out.println("Calisan(String, double)");
    }
    double getMaas() { return maas; }
}
class Yonetici extends Calisan {
    private double prim;
    Yonetici(String ad) {
        this(ad, 5000.0);
        System.out.println("Yonetici(String)");
    }
    Yonetici(String ad, double prim) {
        super(ad);
        this.prim = prim;
        System.out.println("Yonetici(String, double)");
    }
    double gelir() { return getMaas() + prim; }
}
Yonetici y = new Yonetici("Ayşe");   // main
System.out.println(y.gelir());
```

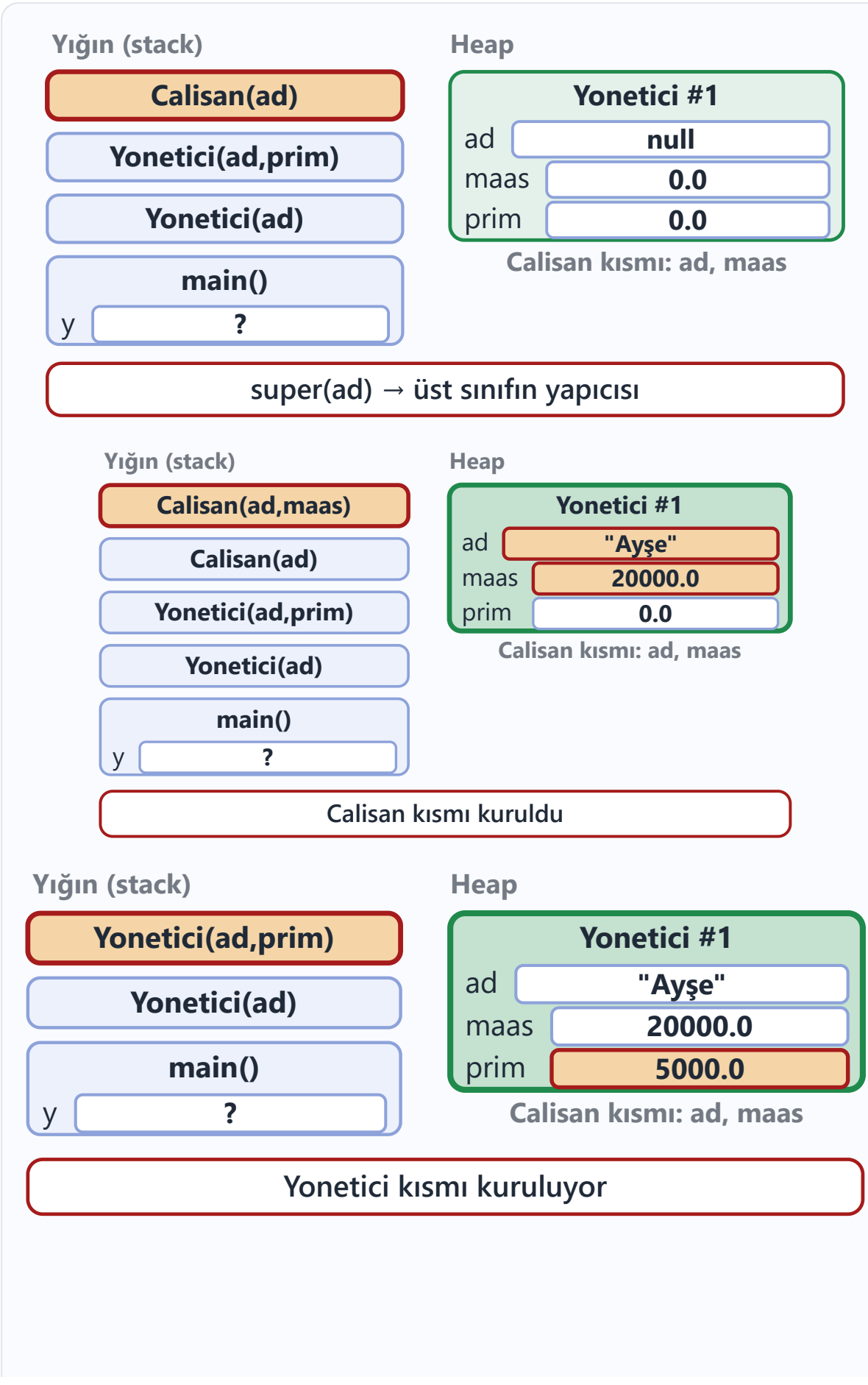
```
Calisan(String, double)
Calisan(String)
Yonetici(String, double)
Yonetici(String)
25000.0
```

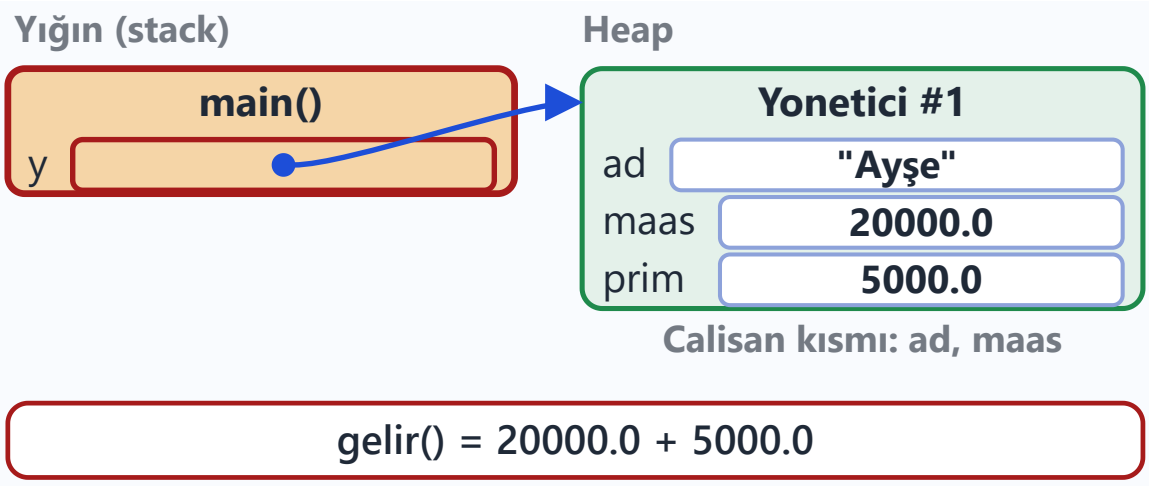
Tek bir `new` dört yapıcıyı çalıştırır. Çağrılar şu sırayla gider: `Yonetici(String)` → `this(ad, 5000.0)` ile `Yonetici(String, double)` → `super(ad)` ile `Calisan(String)` → `this(ad, 20000.0)` ile `Calisan(String, double)` → gizli `super()` ile `Object()`. Zincirin son yapıcısı `Calisan(String, double)` 'dır; ilk satırı `this` ya da `super` olmadığı için derleyici oraya `super()` ekler.

Adım	Olay	ad	maas	prim	Çıktı
1	`new` ; `Yonetici(String)` çağrılır	null	0.0	0.0	
2	`this(ad, 5000.0)`	null	0.0	0.0	
3	`super(ad)` : `Calisan(String)`	null	0.0	0.0	
4	`this(ad, 20000.0)` , gizli `super()` : `Object()` döner	null	0.0	0.0	
5	`Calisan(String, double)` gövdesi	"Ayşe"	20000.0	0.0	Calisan(String, double)

Adım	Olay	ad	maas	prim	Çıktı
6	`Calisan(String)` gövdesinin kalanı	"Ayşe"	20000.0	0.0	Calisan(String)
7	`Yonetici(String, double)` : `prim = 5000.0`	"Ayşe"	20000.0	5000.0	Yonetici(String, double)
8	`Yonetici(String)` biter	"Ayşe"	20000.0	5000.0	Yonetici(String)
9	`y.gelir()` = 20000.0 + 5000.0	"Ayşe"	20000.0	5000.0	25000.0

Tablo 4.3.4: `new Yonetici("Ayşe")` izlemesi. Alanlar `ad`, `maas`, `prim`'dir.


Yığın (stack)
Heap
Calisan(ad)
Yonetici(ad,prim)
Yonetici(ad)
main()
y
?
Yonetici #1
ad
null
maas
0.0
prim
0.0
Calisan kısmı: ad, maas
super(ad) → üst sınıfın yapıcısı
Yığın (stack)
Heap
Calisan(ad,maas)
Calisan(ad)
Yonetici(ad,prim)
Yonetici(ad)
main()
y
?
Yonetici #1
ad
"Ayşe"
maas
20000.0
prim
0.0
Calisan kısmı: ad, maas
Calisan kısmı kuruldu
Yığın (stack)
Heap
Yonetici(ad,prim)
Yonetici(ad)
main()
y
?
Yonetici #1
ad
"Ayşe"
maas
20000.0
prim
5000.0
Calisan kısmı: ad, maas
Yonetici kısmı kuruluyor



Şekil 4.3.11: `this(...)` ve `super(...)` izlemesinden dört an. (a) `super(ad)` : `Yonetici` kısmı, `Calisan` kısmı bitene kadar bekler. (b) İlk gövde en üstte çalışır; `ad` ve `maas` dolar. (c) `super(ad)` tamamlandı, şimdi `prim` dolar. (d) Nesnenin adresi `y`'ye atanır.

Tablo 4.3.4 ve Şekil 4.3.11 aynı kuralı bir kez daha gösterir. Çağrılar aşağıdan yukarı gider, gövdeler yukarıdan başlayarak tamamlanır. Dört yapıcı çalıştı, ama yığında hangi anda kaç yapıcı olursa olsun tek bir nesne kuruldu. `prim` alanı yedinci adıma kadar 0.0'da kalır. Alt sınıfın kendi alanı, üst sınıf kısmı tamamen bitmeden dolmaz.

Son satırdaki `gelir()` metodu kalıtımla gelen `getMaas()`'ı ve kendi `prim` alanını toplar. `maas` `private` olduğu için `Yonetici` onu adıyla okuyamaz; getter kullanır.

Yapıcı zincirinin bir sınırı vardır: döngü kurulamaz. `Calisan(String)` `this(ad, 1.0)`'ı, o da `this(ad)`'ı çağırsaydı zincir hiç `Object`'e ulaşmazdı. Derleyici bunu yakalar:

```
class Calisan {
    Calisan(String ad) { this(ad, 1.0); }
    Calisan(String ad, double m) { this(ad); }
}
```

```
error: recursive constructor invocation
```

Aynı yapıcıda hem `this(...)` hem `super(...)` yazmak da yasaktır. İkisi de ilk deyim olmak ister:

```
Yonetici(String ad) {
    this(ad, 5000.0);
    super(ad);
}
```

```
error: call to super must be first statement in constructor
```

Zincirdeki yapıcılardan yalnız sonuncusu `super(...)` çağırır. Öbürleri `this(...)` ile ona devreder.

4.3.3.6 Kalıtımda ilklendirme sırası

Yapıcılar, bir nesne kurulurken çalışan tek kod değildir. Alan bildirimindeki ilk değerler (`int a = deger(...)`) ve `static` bloklar da çalışır. Kalıtım işin içine girince bu parçaların sırası önem kazanır. Aşağıdaki kod sırayı görünür kılar. Her parça çalışırken bir iz yazar:

```
class Ust {
    static { System.out.println("Ust static"); }
    int a = deger("Ust alanı");
    Ust() { System.out.print("Ust(), "); }
    static int deger(String s) {
        System.out.print(s + ", ");
        return 1;
    }
}
class Alt extends Ust {
    static { System.out.println("Alt static"); }
    int b = deger("Alt alanı");
    Alt() { System.out.println("Alt()"); }
}
// main içinde:
new Alt();
new Alt();
```

```
Ust static
Alt static
Ust alanı, Ust(), Alt alanı, Alt()
Ust alanı, Ust(), Alt alanı, Alt()
```

Çıktı iki aşamayı ayırır: sınıf başlatma ve nesne kurma.

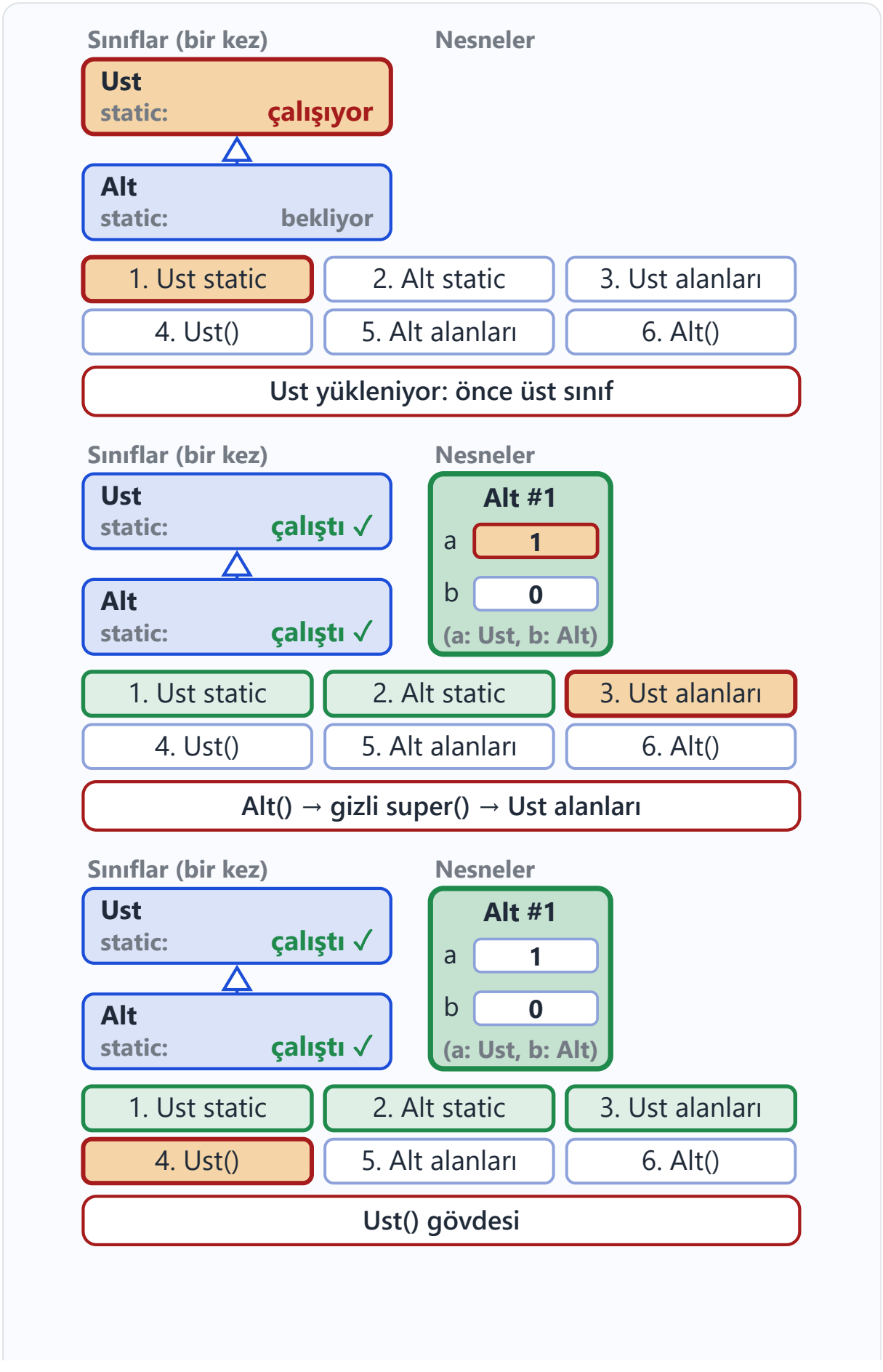
Sınıf başlatma. JVM (*Java Virtual Machine*, Java sanal makinesi) bir sınıfı ilk kez kullanmadan önce onu başlatır. Bu sırada sınıfın `static` blokları ve `static` alanların ilk değerleri çalışır. Bir alt sınıfı başlatmadan önce üst sınıf başlatılır. Bu yüzden önce "Ust static", sonra "Alt static" yazılır. Sınıf başlatma program boyunca **bir kez** yapılır.

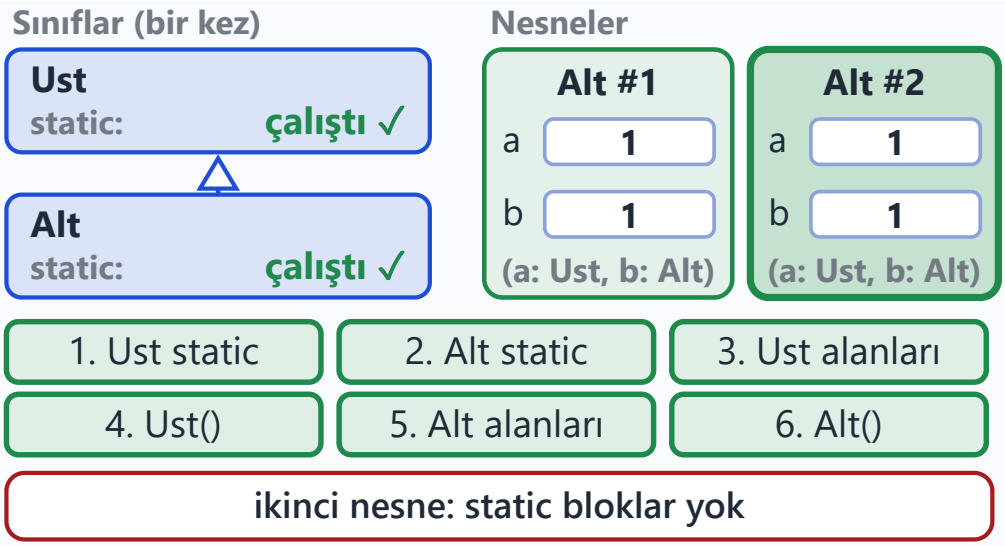
Nesne kurma. Her `new` için şu dört adım bu sırayla çalışır:

1. Üst sınıfın alan ilk değerleri (`a = deger("Ust alanı")`),
2. üst sınıfın yapıcı gövdesi (`Ust()`),

3. alt sınıfın alan ilk değerleri (`b = deger("Alt alanı")`),
4. alt sınıfın yapıcı gövdesi (`Alt()`).

Bunun nedeni derleyicinin alan ilk değerlerini nereye koyduğudur. Derleyici, alan bildirimlerindeki atamaları her yapıcının içine, `super(...)` çağrısından hemen sonraya yerleştirir. Yani `Alt()` aslında şöyle derlenir: önce `super()`, sonra `b = deger("Alt alanı")`, sonra gövde. `Ust()` da aynı biçimde: önce `super()`, yani `Object()`, sonra `a = ...`, sonra gövde.


Sınıflar (bir kez)
Nesneler
Ust
static: çalışıyor
Alt
static: bekliyor
1. Ust static
2. Alt static
3. Ust alanları
4. Ust()
5. Alt alanları
6. Alt()
Ust yükleniyor: önce üst sınıf
Sınıflar (bir kez)
Nesneler
Ust
static: çalıştı ✓
Alt
static: çalıştı ✓
Alt #1
a 1
b 0
(a: Ust, b: Alt)
1. Ust static
2. Alt static
3. Ust alanları
4. Ust()
5. Alt alanları
6. Alt()
Alt() → gizli super() → Ust alanları
Sınıflar (bir kez)
Nesneler
Ust
static: çalıştı ✓
Alt
static: çalıştı ✓
Alt #1
a 1
b 0
(a: Ust, b: Alt)
1. Ust static
2. Alt static
3. Ust alanları
4. Ust()
5. Alt alanları
6. Alt()
Ust() gövdesi



Şekil 4.3.12: İlklendirme sırasından dört an. (a) İlk `new Alt()` için önce `Ust`'un `static` bloğu çalışır. (b) Nesne ayrıldı; `Ust`'un alanı `a` ilk değerini aldı. (c) `Ust()` gövdesi çalışırken `b` hâlâ 0'dır. (d) İkinci `new Alt()`: `static` bloklar yeniden çalışmaz.

Şekil 4.3.12'nin üçüncü karesi bu sıranın tehlikeli bir sonucunu gösterir. `Ust()` gövdesi çalışırken alt sınıfın alanı `b` hâlâ 0'dır. Alt kısım henüz hiç ilk değer almamıştır. Bu ayrıntı, üst sınıfın yapıcısı alt sınıfın geçersiz kıldığı bir metodu çağırdığında önem kazanır:

```
class Ust {
    Ust() { yaz(); }
    void yaz() { System.out.println("Ust.yaz"); }
}
class Alt extends Ust {
    int b = 7;
    String s = "hazır";
    @Override void yaz() {
        System.out.println("Alt.yaz b=" + b + " s=" + s);
    }
}
// main içinde:
Alt a = new Alt();
a.yaz();
```

```
Alt.yaz b=0 s=null
Alt.yaz b=7 s=hazır
```

`new Alt()` önce `Ust()`'u çalıştırır. `Ust()` içindeki `yaz()` çağrısı, nesne bir `Alt` olduğu için `Alt.yaz()`'a gider. Ama o anda `Alt`'ın alan ilk değerleri henüz atanmamıştır: `b` 0, `s` `null`'dır. Kaynak kodda `int b = 7` yazdığı hâlde metot 0 görür. Nesne kurulduktan sonra yapılan ikinci çağrı ise doğru değerleri görür.

Sık yapılan hata

Üst sınıfın yapıcısında geçersiz kılınabilen bir metodu çağırmak. Alt sınıf bu metodu geçersiz kılarsa metot, alt sınıfın alanları daha ilk değer almadan çalışır. Sonuç varsayılan değerlerle yapılan yanlış bir hesap ya da bir `NullPointerException` 'dır. Yapıcıdan çağrılan metotlar `private` , `final` ya da `static` yapılırsa geçersiz kılınamaz ve sorun ortadan kalkar.

Örnek 4.3.3

Örnek ilklendirici bloklar (`{ ... }`) alan ilk değerleriyle aynı yerde, `super(...)` 'dan hemen sonra çalışır. Şu sınıflar verilsin:

```
class A {
    static { System.out.print("As "); }
    { System.out.print("Ab "); }
    A() { System.out.print("A "); }
}
class B extends A {
    static { System.out.print("Bs "); }
    { System.out.print("Bb "); }
    B() { this(0); System.out.print("B "); }
    B(int x) { System.out.print("B" + x + " "); }
}
```

Programdaki ilk `new B()` ne yazar? Önce sınıflar başlatılır: `As Bs` . Sonra `B()` çağrılır. İlk satırı `this(0)` olduğu için gövdesine geçmeden `B(int)` 'e gider. `B(int)` 'in ilk satırında gizli `super()` vardır: `A` 'nın örnek bloğu ve gövdesi çalışır, `Ab A` yazılır. `super()` döndükten sonra `B` 'nin örnek bloğu çalışır: `Bb` . Ardından `B(int)` 'in gövdesi `B0` yazar, en son `B()` 'nin gövdesi `B` yazar. Sonuç: `As Bs Ab A Bb B0 B` .

Örnek bloğu `B()` 'de değil, yalnız `super(...)` çağıran yapıcıda, yani `B(int)` 'te çalışır. Bu yüzden `Bb` bir kez yazılır. Ardından `new B(7)` çağrılırsa `static` bloklar yeniden çalışmaz: `Ab A Bb B7` .

4.3.4 Geçersiz kılma

Alt sınıf, üst sınıftan aldığı bir davranışı her zaman olduğu gibi kullanmak istemez. Köpek “Bu hayvan konuşur.” demez, havlar. Dikdörtgen genel bir şekil gibi değil, dikdörtgen olarak çizilir. Bu kesim alt sınıfın kalıtımla aldığı bir metodu nasıl yeniden yazdığını, bunun hangi kurallara bağlı olduğunu ve bu kuralların neden konduğunu anlatır.

4.3.4.1 Metodu geçersiz kılma

Tanım 4.3.3

Geçersiz kılma (*overriding*), alt sınıfın üst sınıftan kalıtımla aldığı bir metodu **aynı imzayla** yeniden yazarak kendi davranışını vermesidir. Metodun imzası, adı ve parametre listesidir (parametrelerin sayısı, tipleri ve sırası).

Geçersiz kılmanın dört temel koşulu vardır:

- Ad ve parametre listesi aynı olmalıdır.
- Dönüş tipi aynı ya da onun bir alt tipi olmalıdır.
- Erişim daraltılamaz: `public` bir metot alt sınıfta da `public` kalmalıdır.
- `final`, `static` ve `private` metotlar geçersiz kılınamaz.

```
class Sekil {
    public void ciz() {
        System.out.println("Şekil çiziliyor.");
    }
}
class Dikdortgen extends Sekil {
    @Override
    public void ciz() {
        System.out.println("Dikdörtgen çiziliyor.");
    }
}
// main içinde
new Sekil().ciz();
new Dikdortgen().ciz();
```

```
Şekil çiziliyor.
Dikdörtgen çiziliyor.
```

`Dikdortgen`, `ciz()`'i aynı ad, aynı (boş) parametre listesi, aynı dönüş tipi ve aynı erişimle yeniden yazar. `new Sekil().ciz()` üst sınıfın sürümünü, `new Dikdortgen().ciz()` alt sınıfınkini çalıştırır. Hangi sürümün çalışacağına nesnenin gerçek sınıfı karar verir. Değişkenin tipi `Sekil` olsa bile durum değişmez: `Sekil s = new Dikdortgen(); s.ciz();` yine "Dikdörtgen çiziliyor." yazar. Bu davranışa **dinamik bağlama** denir ve çok biçimliliğin temelidir.

Koşullardan biri bozulursa derleyici hata verir. Alt sınıfta `public` yazmayı unutmak erişimi paket düzeyine daraltır:

```
// alt sınıfta: void ciz()
error: ciz() in Dikdortgen cannot override
  ciz() in Sekil
  attempting to assign weaker access
  privileges; was public
```

Dönüş tipini değiştirmek de geçersiz kılmayı bozar:

```
// alt sınıfta: public int ciz()
error: ciz() in Dikdortgen cannot override
  ciz() in Sekil
  return type int is not compatible
  with void
```

İki iletinin ilk satırı aynıdır: `ciz() in Dikdortgen cannot override ciz() in Sekil`. Derleyici aynı imzalı iki metot görmüş ve bunu geçersiz kılma girişimi saymıştır. İkinci satır hangi kuralın bozulduğunu söyler. İletiyi okurken asıl bilgi bu ikinci satırdadır.

4.3.4.2 Geçersiz kılma ve aşırı yükleme

Geçersiz kılma, adı benzeyen başka bir kavramla sık karıştırılır: **aşırı yükleme** (*overloading*). Aşırı yüklemede aynı adlı, ama parametre listesi farklı birden çok metot yazılır. İkisinin farkı Tablo 4.3.5'te özetlenir.

	Geçersiz kılma (*overriding*)	**Aşırı yükleme (*overloading*)**
Nerede?	Alt sınıfta, kalıtımla alınan metot için	Aynı sınıfta (ya da alt sınıfta)
Ad	Aynı	Aynı
Parametreler	Aynı	Farklı (sayı ya da tip)
Dönüş tipi	Aynı ya da alt tipi	Serbest
Amaç	Kalıtımla alınan davranışı değiştirmek	Aynı işi farklı girdilerle yapmak
Seçim zamanı	Çalışma zamanı (nesnenin gerçek tipi)	Derleme zamanı (argüman tipleri)

Tablo 4.3.5: Geçersiz kılma ile aşırı yüklemenin karşılaştırması.

Aynı sınıfta ikisi birlikte de bulunabilir:

```
class Dikdortgen extends Sekil {
    @Override
    public void ciz() { ... }             // geçersiz kılma
    public void ciz(String renk) { ... }  // aşırı yükleme
}
```

İlk `ciz()`, `Sekil`'deki metodu geçersiz kılar. İkincisi `ciz(String renk)`'tir; parametre listesi farklı olduğu için yeni bir metottur ve `ciz()`'i aşırı yükler. `d.ciz("Mavi")` çağrısında derleyici argüman listesine bakar: bir `String` verilmiştir, dolayısıyla `ciz(String)` seçilir. Bu metodun gövdesi rengi iletinin başına ekler:

```
Mavi dikdörtgen çiziliyor.
```

Tablonun son satırı en önemli farkı söyler. Aşırı yüklemede hangi metodun çağrılacağına derleyici, argümanların *derleme anındaki* tiplerine bakarak karar verir. Geçersiz kılmada ise karar program çalışırken, nesnenin gerçek sınıfına bakılarak verilir. İki kavram da çok biçimliliğin (*polymorphism*) temelidir. Birincisine derleme zamanı, ikincisine çalışma zamanı çok biçimliliği denir.

4.3.4.3 `@Override` etiketi

Tanım 4.3.4

`@Override` bir **etiket**tir (*annotation*). Metodun, üst sınıftan kalıtımla alınan bir metodu geçersiz kıldığını bildirir. Derleyici bu bildirimi denetler: üst sınıflarda ya da arayüzlerde aynı imzalı bir metot yoksa derleme hatası verir.

```
public class Dikdortgen extends Sekil {
    // miras alınan metodu geçersiz kıl
    @Override
    public void ciz() {
        System.out.println("Dikdörtgen çiziliyor.");
    }
}
```

Etiket zorunlu değildir; yazılmasa da geçersiz kılma çalışır. Ama her zaman yazılması önerilir. İki yararı vardır. Birincisi, kodu okuyan biri metodun üst sınıftan geldiğini hemen görür. İkincisi ve daha önemlisi, yazım hatalarını derleme anında yakalar. Metodun adı yanlış yazılmış olsun:

```
    @Override
    public void Ciz() { ... }
```

```
error: method does not override or implement
  a method from a supertype
```

`Ciz` büyük harfle başlar. Java büyük ve küçük harfi ayırt ettiği için `Ciz()` ile `ciz()` farklı metotlardır. `@Override` olmasaydı ne olurdu? Derleyici hata vermezdi; `Ciz()` sessizce yeni bir metot olurdu. Program çalışır, ama yanlış davranırdı:

```
class Dikdortgen extends Sekil {
    public void Ciz() {          // @Override yok
        System.out.println("Dikdörtgen çiziliyor.");
    }
}
// main içinde:
Sekil s = new Dikdortgen();
s.ciz();
```

```
Şekil çiziliyor.
```

`s.ciz()` çağrısı için `Dikdortgen`'de geçersiz kılınmış bir `ciz()` yoktur; arama `Sekil`'e çıkar ve genel sürüm çalışır. Bu tür bir hata ancak program denenirken, belki çok sonra fark edilir. `@Override` yazılmış olsaydı derleme anında yakalanırdı. Aynı durum parametre tipini yanlış yazmakta, örneğin `equals(Object)` yerine `equals(Nokta)` yazmakta da görülür.

4.3.4.4 Geçersiz kılma kuralları

Geçersiz kılmanın kuralları tek tek ezberlenebilir, ama hepsi tek bir gerekçeden türer. Tablo 4.3.6 kuralları, bozulduklarında javac'ın verdiği iletilerle birlikte verir. İletiler JDK 21 ile alınmıştır.

Özellik	Alt sınıftaki metotta izin verilen	Kural bozulursa javac
Ad ve parametreler	Birebir aynı	Hata yok: yeni bir metot (aşırı yükleme) olur
Dönüş tipi	Aynı ya da bir alt tipi (kovaryant)	`return type String is not compatible with Hayvan`
Erişim	Aynı ya da daha geniş: `protected` → `public`	`attempting to assign weaker access privileges`
Kontrol edilen istisna	Aynısı, daha özeli ya da hiç	`overridden method does not throw Exception`
`static`	Örnek metodu `static` olamaz; `static` metot örnek metodu olamaz	`overriding method is static` / `overridden method is static`
`final` metot	Geçersiz kılınamaz	`overridden method is final`
`private` metot	Görünmez; aynı adlı metot ilgisiz yeni bir metottur	Hata yok (`@Override` varsa hata)

Tablo 4.3.6: Geçersiz kılma kuralları ve kural bozulunca alınan javac iletileri.

Ortak gerekçe şudur: alt sınıf nesnesi, üst sınıf nesnesinin beklendiği her yerde kullanılabilmelidir. `Sekil` tipinde bir değişkenle çalışan bir kod, o değişkenin gösterdiği nesnenin aslında `Dikdortgen` mi `Daire` mi olduğunu bilmez. Bu kod yalnız `Sekil` 'in verdiği söze güvenir. Üst sınıfın sözü üç şeyden oluşur: metot ne döndürecek, kime açık olacak, hangi istisnaları fırlatabilecek. Alt sınıf bu sözü koruyabilir ya da genişletebilir, ama daraltamaz.

Kurallar bu gerekçeyle tek tek açıklanır. Üst sınıfın metodu `public` ise her yerden çağrılabilir. Alt sınıf onu paket erişimine indirseydi, `Sekil` değişkeni üzerinden başka bir paketten yapılan çağrı bir `Dikdortgen` nesnesine ulaştığında erişim yasak olurdu. Erişimi genişletmek ise sorun yaratmaz. `protected` bir metodu `public` yapmak, eski çağıranların hiçbirini bozmaz.

İstisnalarda da durum aynıdır. Üst sınıfın `f()` metodu hiç kontrol edilen istisna bildirmiyorsa, onu çağıran kod `try` - `catch` yazmamıştır. Alt sınıfın sürümü `throws Exception` derse bu kod hazırlıksız yakalanır. Derleyici bunu önler:

```
class Ust { void f() { } }
class Alt extends Ust {
    @Override void f() throws Exception { }
}
```

```
error: f() in Alt cannot override f() in Ust
  overridden method does not throw Exception
```

Tersi serbesttir. Üst sınıf `throws Exception` bildiriyorsa alt sınıf daha özel bir istisna (`throws IOException`) bildirebilir ya da hiç bildirmeyebilir. Çağıran zaten `Exception` 'ı yakalamaya hazırdır; daha özel bir istisna da bunun içindedir. `RuntimeException` ve alt sınıfları kontrol edilmeyen istisnalardır; bu kural onlara uygulanmaz.

Örnek 4.3.4

`class U { protected void a() { } public void b() { } }` verilsin. `class V extends U` içinde `@Override public void a() { }` ve `@Override void b() { }` yazılıyor. Hangisi derlenir?

`a()` üst sınıfta `protected` 'tır; alt sınıfta `public` yapmak erişimi genişletir, derlenir. `b()` üst sınıfta `public` 'tir; belirleyicisiz yazmak erişimi paket düzeyine indirir. javac `b() in V cannot override b() in U` ve `attempting to assign weaker access privileges; was public` iletilerini verir. Erişim sırası dardan genişe şöyledir: `private` , paket, `protected` , `public` .

4.3.4.5 Kovaryant dönüş tipi

Dönüş tipi kuralı ilk okumada katı görünür, ama bir esnekliği vardır. Geçersiz kılan metot, üst sınıftaki dönüş tipinin bir **alt tipini** döndürebilir:

```
class Hayvan {
    Hayvan yavrula() { return new Hayvan(); }
    public String toString() { return "Hayvan"; }
}
class Kedi extends Hayvan {
    @Override
    Kedi yavrula() { return new Kedi(); }  // alt tip
    void miyavla() { System.out.println("miyav"); }
    public String toString() { return "Kedi"; }
}
// main içinde:
Kedi k = new Kedi();
k.yavrula().miyavla();   // dönüşüm gerekmez
Hayvan h = k;
System.out.println(h.yavrula());
```

```
miyav
Kedi
```

Tanım 4.3.5

Kovaryant dönüş tipi (*covariant return type*), geçersiz kılan metodun, üst sınıftaki dönüş tipinin bir alt tipini döndürmesidir.

`Hayvan.yavrula()` bir `Hayvan` döndürür. `Kedi.yavrula()` ise bir `Kedi` döndürür; `Kedi`, `Hayvan`'ın alt tipi olduğu için bu geçerli bir geçersiz kılmadır. Yararı ilk çağrıda görülür. `Kedi` referansıyla çağıran, sonucu doğrudan `Kedi` olarak alır. `k.yavrula().miyavla()` yazılabilir; `(Kedi)` dönüşümüne gerek kalmaz. Dönüş tipi `Hayvan` kalsaydı `miyavla()` çağrısı derlenmezdi, çünkü `Hayvan`'da böyle bir metot yoktur.

`Hayvan` referansıyla çağıran için ise söz bozulmaz. `h.yavrula()`'nın derleme anındaki tipi `Hayvan`'dır ve gelen nesne gerçekten bir hayvandır, çünkü bir `Kedi` de bir `Hayvan`'dır. Çağrı nesne bir `Kedi` olduğu için `Kedi.yavrula()`'ya gider; `println` yeni kedinin `toString()`'ini çağırır ve "Kedi" yazar.

Kovaryans yalnız referans tiplerinde geçerlidir. `int` döndüren bir metodu `long` döndürecek biçimde geçersiz kılmak olmaz; ilkel tipler arasında alt tip ilişkisi yoktur. İlgisiz bir referans tipi de olmaz:

```
error: yavrula() in Kedi cannot override
  yavrula() in Hayvan
  return type String is not compatible with Hayvan
```

`String`, `Hayvan`'ın alt tipi değildir. `Hayvan` değişkeniyle `yavrula()`'yı çağıran kod bir hayvan bekler, eline bir dizgi geçer. Derleyici buna izin vermez.

4.3.4.6 `private` ve `static` metotlar

`private` bir metot alt sınıfa görünmez. Alt sınıfta aynı adla yazılan bir metot onu geçersiz kılmaz; onunla ilgisiz, yeni bir metottur. Bu bazen beklenmedik sonuçlar doğurur:

```
class Ust {
    private void selam() { System.out.println("Ust.selam"); }
    void cagir() { selam(); }
}
class Alt extends Ust {
    void selam() { System.out.println("Alt.selam"); } // yeni
}
// main içinde:
Alt a = new Alt();
a.cagir();
a.selam();
```

```
Ust.selam
Alt.selam
```

`a.selam()` beklendiği gibi `Alt`'ın metodunu çağırır. Şaşırtıcı olan `a.cagir()`'ın sonucudur. `cagir()`, `Ust`'ta yazılmıştır ve `Alt` onu kalıtımla alır. İçindeki `selam()` çağrısı derleme sırasında `Ust`'un kendi `private` metoduna bağlanır. `private` metotlar geçersiz kılınamadığı için derleyici bu çağrıyı çalışma zamanına bırakmaz. Nesne bir `Alt` olsa da "Ust.selam" yazılır.

`Ust`'taki `selam()` `private` olmasaydı durum tersine dönerdi:

```
class Ust {
    void selam() { System.out.println("Ust.selam"); }
    void cagir() { selam(); }
}
class Alt extends Ust {
    @Override void selam() { System.out.println("Alt.selam"); }
}
// main içinde:
new Alt().cagir();
```

```
Alt.selam
```

Burada `Alt.selam()` gerçekten geçersiz kılar. `cagir()`'daki çağrı nesnenin gerçek sınıfına göre seçilir ve `Alt`'ın sürümü çalışır. İki program arasındaki tek fark `private` sözcüğüdür, ama davranış tamamen değişir.

Programcı `Alt.selam()`'ın geçersiz kıldığını sanıyorsa yanılgıyı `@Override` alışkanlığı ortaya çıkarır. `private` örnekteki `Alt.selam()`'a `@Override` eklenirse derleyici `method does not override or implement a method from a supertype` iletisini verir. Etiket, "geçersiz kıldığımı sanıyorum, denetle" demektir; derleyici de denetimin sonucunu bildirir.

Aynı durum `static` metotlarda da görülür. Alt sınıftaki aynı imzalı `static` metot üst sınıfınkini geçersiz kılmaz, yalnız **gizler** (*hiding*). Hangisinin çalışacağına nesne değil, değişkenin derleme anındaki tipi karar verir:

```
class Ust {
    static void f() { System.out.println("Ust.f"); }
    void g() { System.out.println("Ust.g"); }
}
class Alt extends Ust {
    static void f() { System.out.println("Alt.f"); }
    @Override void g() { System.out.println("Alt.g"); }
}
// main içinde:
Ust u = new Alt();
u.f();
u.g();
```

```
Ust.f
Alt.g
```

`u` bir `Alt` nesnesini gösterir, ama tipi `Ust`'tur. `static` metot `f()` sınıfa aittir. Derleyici `u.f()`'yi `Ust.f()` olarak okur ve nesneye hiç bakmaz. Örnek metodu `g()` ise geçersiz kılınmıştır; nesnenin gerçek sınıfı seçimi yapar. `static` metodu bir nesne değişkeni üzerinden çağırmak bu yüzden yanıltıcıdır. Doğru yazım `Ust.f()` ya da `Alt.f()`'dir. Bir örnek metodu `static` metotla, ya da tersine, aynı imzayla yazmak ise derleme hatasıdır:

```
error: f() in Alt cannot override f() in Ust
  overriding method is static
error: f() in Alt cannot override f() in Ust
  overridden method is static
```

4.3.4.7 Örnek: ürün hiyerarşisi

Geçersiz kılmanın gücü, üst sınıfın kendi içinden yaptığı çağrılarda ortaya çıkar. Bir mağaza programında ürünlerin satış fiyatı, KDV (katma değer vergisi) eklenerek hesaplansın. Gıda ürünlerinde KDV oranı farklıdır; bazı ürünlerde de indirim vardır.

```
class Urun {
    private String ad;
    private double fiyat;
    Urun(String ad, double fiyat) {
        this.ad = ad; this.fiyat = fiyat;
    }
    double kdvOrani() { return 0.20; }
    double satisFiyati() { return fiyat * (1 + kdvOrani()); }
    @Override public String toString() {
        double f = Math.round(satisFiyati() * 100) / 100.0;
        return ad + ": " + f + " TL";
    }
}
class GidaUrunu extends Urun {
    GidaUrunu(String ad, double fiyat) { super(ad, fiyat); }
    @Override double kdvOrani() { return 0.01; }
}
class IndirimliUrun extends Urun {
    private double indirim;
    IndirimliUrun(String ad, double fiyat, double indirim) {
        super(ad, fiyat);
        this.indirim = indirim;
    }
    @Override double satisFiyati() {
        return super.satisFiyati() * (1 - indirim);
    }
    @Override public String toString() {
        return super.toString()
            + " (%" + (int) (indirim * 100) + " indirim)";
    }
}
```

`Urun` sınıfı fiyat hesabını iki adıma böler. `kdvOrani()` oranı verir, `satisFiyati()` bu oranı kullanarak hesap yapar. `toString()` ise satış fiyatını iki ondalığa yuvarlar ve adla birlikte yazar. Yuvarlamada `Math.round(x * 100) / 100.0` kalıbı kullanılır. Bu kalıp, kayan nokta hesabının doğurabileceği uzun kuyrukları temizler; örneğin `0.1 + 0.2` işleminin sonucu 0.30000000000000004'tür.

İki alt sınıf iki farklı yol izler. `GidaUrunu` yalnız `kdvOrani()`'nı değiştirir. `IndirimliUrun` ise `satisFiyati()`'nı geçersiz kılar, ama üst sınıfın hesabını değiştirmez; onu **genişletir**. `super.satisFiyati()` sonucuna indirimi uygular. `toString()`'i de genişletir ve sonuna indirim oranını ekler.

```
Urun[] sepet = {new Urun("Kalem", 50),
    new GidaUrunu("Ekmek", 10),
    new IndirimliUrun("Kulaklık", 1000, 0.25)};
for (Urun u : sepet) System.out.println(u);
```

```
Kalem: 60.0 TL
Ekmek: 10.1 TL
Kulaklık: 900.0 TL (%25 indirim)
```

Ürün	`kdvOrani()`	`satisFiyati()`	Hesap	Sonuç
Kalem (`Urun`)	`Urun` : 0.20	`Urun`	50 × 1,20	60.0
Ekmek (`GidaUrunu`)	`GidaUrunu` : 0.01	`Urun`	10 × 1,01	10.1
Kulaklık (`IndirimliUrun`)	`Urun` : 0.20	`IndirimliUrun` → `super`	1000 × 1,20 × 0,75	900.0

Tablo 4.3.7: Sepetteki her ürün için hangi metodun çalıştığı ve hesap.

Tablo 4.3.7'nin ikinci satırı örneğin en öğretici kısmıdır. `GidaUrunu`'nda `satisFiyati()` ve `toString()` yazılmamıştır. İkisi de `Urun`'dan gelir. Ama `satisFiyati()`'nın içindeki `kdvOrani()` çağrısı `this.kdvOrani()` demektir ve `this` bir `GidaUrunu`'dur. Çağrı nesnenin sınıfına göre seçilir; %1 oran kullanılır. Üst sınıfın kodu, alt sınıfın yazdığı bir metodu çağırmış olur. Üst sınıf yazılırken `GidaUrunu` belki henüz yoktu.

Üçüncü satırda zincir iki düzeylidir. `toString()` önce `IndirimliUrun`'unkidir ve `super.toString()` ile `Urun`'unkini çağırır. `Urun.toString()`, `satisFiyati()`'nı çağırır; nesne bir `IndirimliUrun` olduğu için `IndirimliUrun.satisFiyati()` çalışır. O da `super.satisFiyati()` ile `Urun`'un hesabını yaptırır: 1000 × 1,20 = 1200. Sonra indirimi uygular: 1200 × 0,75 = 900. İndirim oranı `(int)(indirim * 100)` ile tam sayıya çevrilir ve "%25" yazılır.

Örnek 4.3.5

Fiyatı 200 TL olan üç ürün düşünülsün: düz bir `Urun`, bir `GidaUrunu` ve %10 indirimli bir `IndirimliUrun`. Satış fiyatları nedir?

`Urun`: 200 × 1,20 = 240.0. `GidaUrunu`: `kdvOrani()` 0.01 döner, 200 × 1,01 = 202.0. `IndirimliUrun`: önce `super.satisFiyati()` = 240, sonra 240 × 0,90 = 216.0. İndirim KDV'li fiyata uygulanır. KDV'siz fiyata uygulansaydı sonuç değişmezdi, çünkü çarpma sırası sonucu etkilemez: 200 × 0,90 × 1,20 de 216'dır.

Bu tasarım, üst sınıfın yeni ürün türlerine kapalı olmadan genişletilebildiğini gösterir. Lüks tüketim vergisi olan bir ürün eklenecekse yeni bir alt sınıf `kdvOrani()`'nı ya da `satisFiyati()`'nı geçersiz kılar. `Urun`'a ve sepette gezinen döngüye dokunmak gerekmez.

4.3.5 Erişim ve sınırlama

Kalıtım, üst sınıfın içini alt sınıflara bir ölçüde açar. Bu açıklığın ne kadar olacağı tasarımın bir parçasıdır. Bu kesim önce alt sınıflara özel bir erişim düzeyi olan `protected`'ı anlatır. Ardından kalıtımı tamamen kapatan `final`'ı ve nesnenin görünümünü üst sınıf tipiyle daraltmayı işler.

4.3.5.1 `protected` erişim düzeyi

Java'da dört erişim düzeyi vardır. `public` üyelere her yerden erişilir. `private` üyelere yalnız o sınıfın içinden erişilir. Belirleyici yazılmayan üyeler **paket erişimlidir**; aynı paketteki bütün sınıflar onları görür. `protected` ise paket erişimi ile `public` arasında bir düzeydir.

Belirleyici	Aynı sınıf	Aynı paket	Alt sınıf (başka paket)	Her yer
`private`	✓	✗	✗	✗
(belirleyici yok: paket)	✓	✓	✗	✗
`protected`	✓	✓	✓	✗
`public`	✓	✓	✓	✓

Tablo 4.3.8: Erişim belirleyicilerinin kapsamı (✓ erişilebilir, ✗ erişilemez).

> **Tanım 4.3.6**
>
> `protected` bir üyeye sınıfın kendisi, aynı paketteki bütün sınıflar ve başka paketlerdeki alt sınıflar erişebilir.

Tablo 4.3.8 her satırda bir öncekinden daha geniş bir kapsam gösterir. `protected` sözcüğü "korumalı" demektir, ama tablodan görüldüğü gibi paket erişiminden *daha açıktır*. Bu sık karıştırılır. `protected`, paket erişiminin üstüne bir izin daha ekler: başka paketteki alt sınıflara açılır.

Bir sınıfın `private` alanlarına alt sınıf yalnız üst sınıftan kalıtımla aldığı `public` ya da `protected` metotlar aracılığıyla erişebilir. İyi uygulama şudur: alanlar yine `private` kalır; alt sınıfların ihtiyaç duyduğu işlemler `protected` metotlar olarak açılır. Alanı `protected` yapmak kolay görünür, ama alanın değerini denetleme sorumluluğu artık bütün alt sınıflara dağılır. Bir alt sınıf sağlığı −50 yapabilir ve `Hayvan`'ın bunu önleme şansı kalmaz.

4.3.5.2 `protected`: kod örneği

Aşağıdaki iki sınıf farklı paketlerdedir. `Hayvan` `canli` paketinde, `Kedi` `evcil` paketindedir.

```
package canli;

public class Hayvan {
    protected String isim;  // alt sınıflar
    private double saglik;  // yalnızca Hayvan

    public Hayvan(String isim) {
        this.isim = isim;
        saglik = 100.0;
    }
    public double getSaglik() { return saglik; }
    protected void saglikEkle(double miktar) {
        saglik += miktar;
    }
}
```

```
package evcil;     // başka paket
import canli.Hayvan;

public class Kedi extends Hayvan {
    public Kedi(String isim) { super(isim); }
    public void uyu() {
        // protected alan: izin var
        System.out.println(isim + " uyuyor.");
        // saglik += 5;  // private: hata
        saglikEkle(5);   // protected metot
        System.out.println("Sağlık: "
                           + getSaglik());
    }
}
```

`Hayvan`'ın iki alanı iki farklı tasarım seçimini gösterir. `isim` `protected`'tır; alt sınıflar onu doğrudan okuyabilir. `saglik` ise `private`'tır. Ona yalnız iki yoldan ulaşılır: herkese açık `getSaglik()` okur, yalnız alt sınıflara açık `saglikEkle(double)` değiştirir.

`Kedi`'nin `uyu()` metodu bu iki yolu kullanır. `isim`'i doğrudan yazar; `Kedi` başka pakette olsa da `Hayvan`'ın alt sınıfıdır. Sağlığı `saglikEkle(5)` ile artırır ve `getSaglik()` ile okur. `new Kedi("Mavi").uyu();` şunu yazar:

```
Mavi uyuyor.
Sağlık: 105.0
```

Yorum satırındaki `saglik += 5;` açılırsa derleme durur:

```
error: saglik has private access in Hayvan
```

Bu tasarımın yararı somuttur. `Hayvan`'ın sahibi ileride sağlığın 100'ü aşmamasını isteyebilir. Bu kuralı yalnız `saglikEkle`'ye yazar; bütün alt sınıflar kendiliğinden uyar. Alan `protected` olsaydı her alt sınıfın kodunu tek tek denetlemek gerekirdi.

Alt sınıf olmayan bir sınıf ise başka paketten `protected` üyeye erişemez:

```
package diger;
import canli.Hayvan;
public class Bakici {             // alt sınıf değil
    String ad(Hayvan h) { return h.isim; }
}
```

```
diger/Bakici.java:4: error: isim has protected access in Hayvan
```

`Bakici` ne `canli` paketindedir ne de `Hayvan`'ın alt sınıfıdır. Tablodaki dört sütunun hiçbiri ona izin vermez. `Bakici` `canli` paketine taşınsaydı derlenirdi: aynı paketteki sınıflar `protected` üyelere erişir.

4.3.5.3 `protected` erişimin ince kuralı

`protected`'ın bir kuralı tabloya sığmaz. Başka paketteki alt sınıf, `protected` üyeye yalnız **kendi tipinden** (ya da kendi alt tiplerinden) bir referans üzerinden erişebilir:

```
package canli;
public class Hayvan {
    protected String isim;
    public Hayvan(String isim) {
        this.isim = isim;
    }
}
```

```
package evcil;
import canli.Hayvan;
public class Kedi extends Hayvan {
    public Kedi(String isim) { super(isim); }
    void karsilastir(Kedi k, Hayvan h) {
        System.out.println(isim);    // kendi alanı
        System.out.println(k.isim);  // Kedi üzerinden
        System.out.println(h.isim);  // Hayvan üzerinden
    }
}
```

```
evcil/Kedi.java:8: error: isim has protected access in Hayvan
        System.out.println(h.isim);
```

`karsilastir` metodundaki üç satır üç farklı referans kullanır. `isim` tek başına yazıldığında `this.isim` demektir; `this` bir `Kedi`'dir ve erişim serbesttir. `k.isim`'de `k`'nin tipi `Kedi`'dir; bu da serbesttir. `h.isim`'de ise `h`'nin tipi `Hayvan`'dır ve derleme durur.

Kuralın gerekçesi `h`'nin neyi gösterebileceğindedir. `h` herhangi bir `Hayvan` olabilir, örneğin bir `Kopek`. `protected`'ın verdiği izin "alt sınıf, kendi kalıtımla aldığı kısma dokunabilir" iznidir. Bir kedinin bir köpeğin iç verisine dokunması bu iznin dışındadır. Derleyici bu yüzden `Hayvan` tipli referansa izin vermez. Aynı nedenle `Kopek` tipli bir parametre de geçmez: `Kedi` içinde `k.isim`, `k` bir `Kopek` ise yine `isim has protected access in Hayvan` iletisini doğurur.

Aynı paketteki sınıflar için bu kısıt yoktur. Orada `protected`, paket erişimini de kapsar. `canli` paketindeki bir `Veteriner` sınıfı, alt sınıf olmasa da herhangi bir `Hayvan`'ın `isim`'ini okuyabilir ve `saglikEkle`'yi çağırabilir.

4.3.5.4 `final` ile kalıtımı sınırlamak

`final` anahtar sözcüğü kalıtımı sınırlamak için kullanılır. Bir metot `final` tanımlanırsa hiçbir alt sınıfta geçersiz kılınamaz. Bir sınıf `final` tanımlanırsa ondan hiçbir alt sınıf türetilemez.

```
class Hayvan {
    public final void nefesAl() { ... }
}
class Kedi extends Hayvan {
    public void nefesAl() { ... }
}
```

```
error: nefesAl() in Kedi cannot override
  nefesAl() in Hayvan
  overridden method is final
```

```
final class Sabit { ... }
class Deneme extends Sabit { }
```

```
error: cannot inherit from final Sabit
```



Şekil 4.3.13: `final` sınıftan alt sınıf türetilemez; `final` metot alt sınıfta geçersiz kılınamaz.

Java kütüphanesinde pek çok `final` sınıf vardır: `String`, `Math` ve `Integer` bunlardandır. `class BenimDizgim extends String { }` yazmak `cannot inherit from final String` hatasını verir. Nedeni güvenlik ve tutarlılıktır. `String` nesneleri değiştirilemez olmalıdır. Bir metot bir `String` aldığında o metnin sonradan değişmeyeceğine güvenir. Örneğin dosya adları, ağ adresleri ve `HashMap` anahtarları bu güvene dayanır. `String`'in değiştirilebilen bir alt sınıfı olsaydı bu söz bozulurdu. `final` bu kapıyı kapatır.

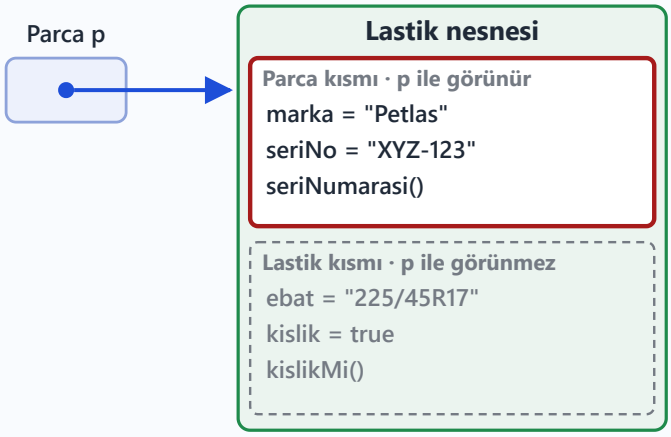
`final` metot da benzer bir gerekçeyle kullanılır. `Hayvan`'ın `nefesAl()` metodu sağlık hesabının doğruluğu için kritikse, hiçbir alt sınıfın onu değiştirmemesi istenir. Yapıcıdan çağrılan metotları `final` yapmak da, alt sınıfın henüz ilk değer almamış alanlarını okuyan bir geçersiz kılmayı önler.

Değişkenlerde `final` başka bir anlam taşır: değişkene değer bir kez atanır ve değişken sabit olur. Aynı sözcük üç yerde üç anlam taşır. Ortak noktaları "bundan sonra değişmez"dir: değişkenin değeri, metodun gövdesi, sınıfın hiyerarşideki yeri.

4.3.5.5 Nesnenin görünümünü kısıtlama

Bir alt sınıf nesnesi, üst sınıf tipinde bir değişkene atanabilir. Bu atamaya **yukarı dönüşüm** (*upcasting*) denir. Atama kendiliğindendir, dönüşüm işleci gerekmez, çünkü her `Lastik` bir `Parca`'dır.

```
Parca p = new Lastik("Petlas", "XYZ-123", "225/45R17", true);
System.out.println(p.seriNumarasi());   // izin verilir → XYZ-123
// p.kislikMi()                         // izin verilmez
```



Şekil 4.3.14: `Parca` tipindeki `p` referansı bir `Lastik` nesnesini gösterir. `p` üzerinden nesnenin yalnız `Parca` kısmı görünür; `ebat`, `kislik` ve `kislikMi()` gizli kalır.

Şekil 4.3.14 önemli bir ayrımı gösterir. Nesnenin kendisi değişmemiştir; hâlâ bütün alanlarıyla bir `Lastik`'tir. Değişen yalnız ona bakan pencerenin genişliğidir. Derleyici hangi metotların çağrılabileceğine **değişkenin tipine** bakarak karar verir. `p`'nin tipi `Parca`'dır; `Parca`'da `seriNumarasi()` vardır, `kislikMi()` yoktur:

```
error: cannot find symbol
  symbol:   method kislikMi()
  location: variable p of type Parca
```

Bu kısıtlama bir tasarım aracıdır. Bir metot parametre olarak `Parca` alırsa her türlü parçayla çalışır ve yalnız parçalara özgü işlemleri kullanabilir. Böylece dışarıya nesnenin yalnız gereken yüzü açılır. Stok sayan bir metot, lastiğin kışlık olup olmadığını bilmek zorunda kalmaz.

`Lastik`'e özgü bir metoda ulaşmak gerekirse **aşağı dönüşüm** (*downcasting*) yapılır. Aşağı dönüşüm açıkça yazılır ve çalışma anında denetlenir. Bu yüzden önce `instanceof` ile tip kontrolü yapılır:

```
Parca p = new Lastik("Petlas", "XYZ-123", "225/45R17", true);
if (p instanceof Lastik) {
    Lastik l = (Lastik) p;          // güvenli
    System.out.println(l.kislikMi());
}
Parca q = new Parca("Bosch", "AB-9");
Lastik l2 = (Lastik) q;             // q bir Lastik değil
```

```
true
Exception in thread "main" java.lang.ClassCastException: class Parca cannot be cast to
class Lastik
```

`p instanceof Lastik` doğru olduğu için `(Lastik) p` güvenlidir ve `true` yazılır. `q` ise düz bir `Parca`'dır. Derleyici `(Lastik) q`'yu kabul eder, çünkü bir `Parca` değişkeni bir `Lastik`'i gösteriyor olabilir. Çalışma anında JVM nesneye bakar, onun bir `Lastik` olmadığını görür ve `ClassCastException` fırlatır. Aşağı dönüşüm yeni bir nesne üretmez. Aynı nesneye daha geniş bir pencereden bakmayı sağlar; nesne o pencereye uymuyorsa istisna oluşur.

Sık yapılan hata

Kontrolsüz aşağı dönüşüm: `(Lastik) p` yazmak, `p` bir `Lastik` değilse `ClassCastException` doğurur. Dönüşümden önce `if (p instanceof Lastik)` denetimi yapılır. Java 16'dan beri bu ikisi tek adımda yazılabilir: `if (p instanceof Lastik l) { … l.kislikMi() … }`. Burada `l`, denetim doğruysa dönüştürülmüş referansı tutar.

4.3.6 Sınıflar arası ilişkiler

Kalıtım, iki sınıfı ilişkilendirmenin tek yolu değildir. Bir sınıf başka bir sınıfın nesnesini alan olarak da tutabilir. Bu kesim iki ilişkiyi karşılaştırır, Java'nın çoklu kalıtıma neden izin vermediğini açıklar ve kalıtımın yanlış kullanıldığında doğurduğu iki sorunu kod izlemeleriyle gösterir.

4.3.6.1 is-a ve has-a ilişkileri

Tanım 4.3.7

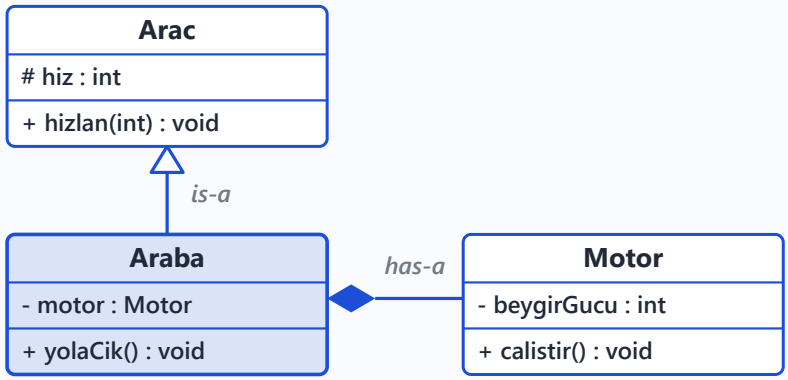
is-a ("bir türüdür") ilişkisi kalıtımdır: bir sınıf başka bir sınıftan türetilir ve onun bir türü olur. Alt sınıf nesnesi üst sınıfın nesnesi gibi de kullanılabilir.

has-a ("sahiptir") ilişkisi ise **oluşum**dur (*composition*): bir sınıf, başka bir sınıf tipinde bir alan içerir. Nesne, içinde başka nesnelere referanslar tutar.

İki ilişkiyi ayırmanın en kolay yolu cümle kurmaktır. "Araba bir taşıttır" doğru bir cümledir; `Araba extends Arac` yazılır. "Araba bir motordur" yanlıştır. Doğru cümle "arabanın bir motoru vardır"dır; motor arabanın bir alanı olur. Gerçek sınıflar çoğu zaman iki ilişkiyi birlikte taşır:

```
// Is-a ilişkisi (kalıtım)
public class Araba extends Arac {
    // Has-a ilişkisi (oluşum)
    private Motor motor;

    public Araba(int beygirGucu) {
        motor = new Motor(beygirGucu);
    }
    public void yolaCik() {
        motor.calistir();   // Motor'a devret
        hizlan(50);         // Arac'tan miras
        System.out.println("Hız: " + hiz
                           + " km/sa");
    }
}
// Motor: beygirGucu alanı ve calistir():
//   "Motor çalıştı: " + beygirGucu + " BG"
```



Şekil 4.3.15: `Araba` , `Arac` 'tan türer (is-a). `Araba` bir `Motor` nesnesi içerir (has-a). UML'de oluşum, içeren sınıf tarafında dolu bir eşkenar dörtgenle çizilir.

`Arac` sınıfında bir `hiz` alanı ve `hizlan(int)` metodu vardır; `hiz` alt sınıfların erişebilmesi için `protected` 'tır. `Motor` sınıfı beygir gücünü tutar ve `calistir()` metodu "Motor çalıştı: ... BG" yazar. `Araba` yapıcısında kendi motorunu oluşturur. `yolaCik()` metodu iki ilişkiyi birlikte kullanır: motoru çalıştırma işini `Motor` nesnesine devreder, hızlanmayı `Arac` 'tan kalıtımla aldığı metotla yapar. `new Araba(110).yolaCik();` şunu yazar:

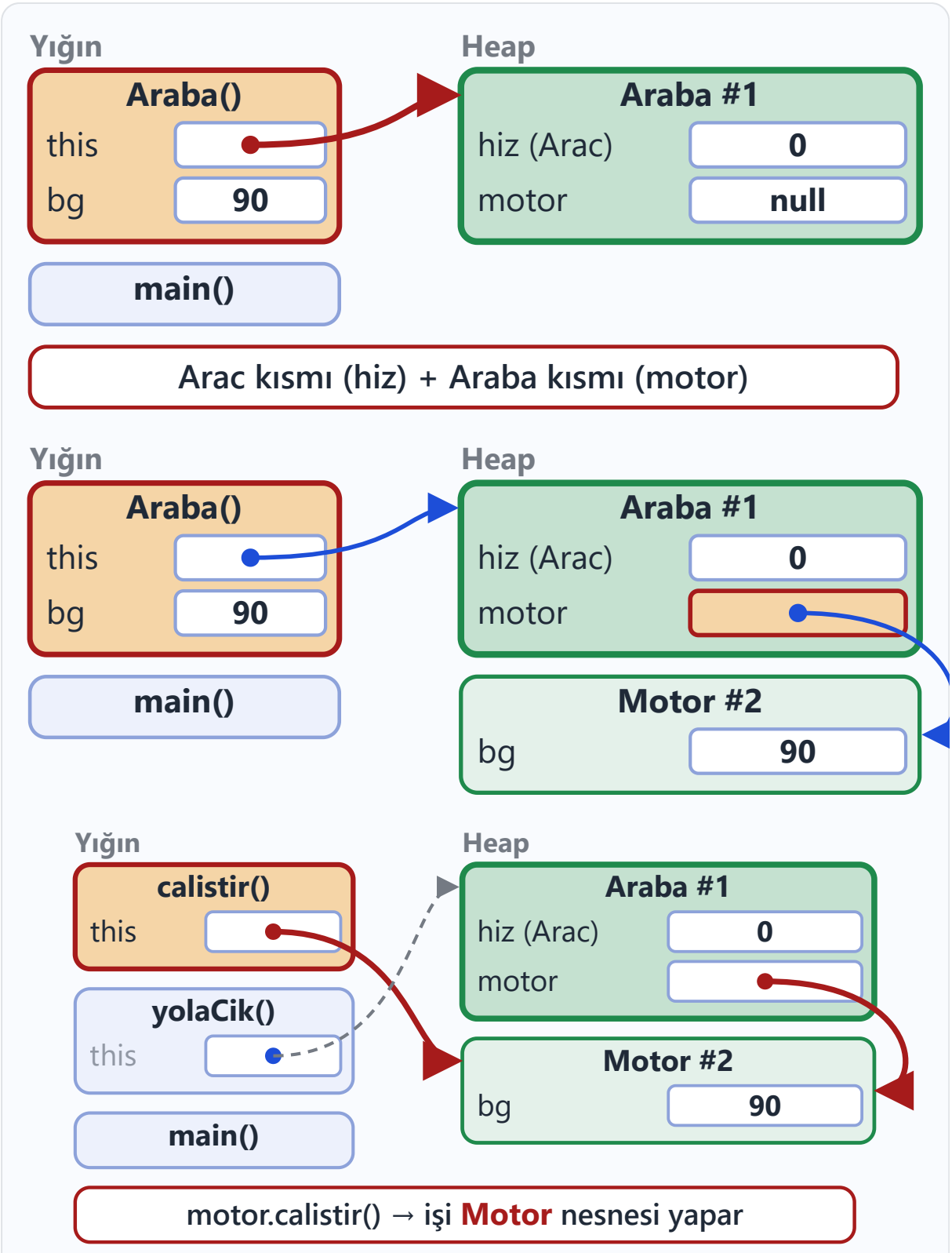
```
Motor çalıştı: 110 BG
Hız: 50 km/sa
```

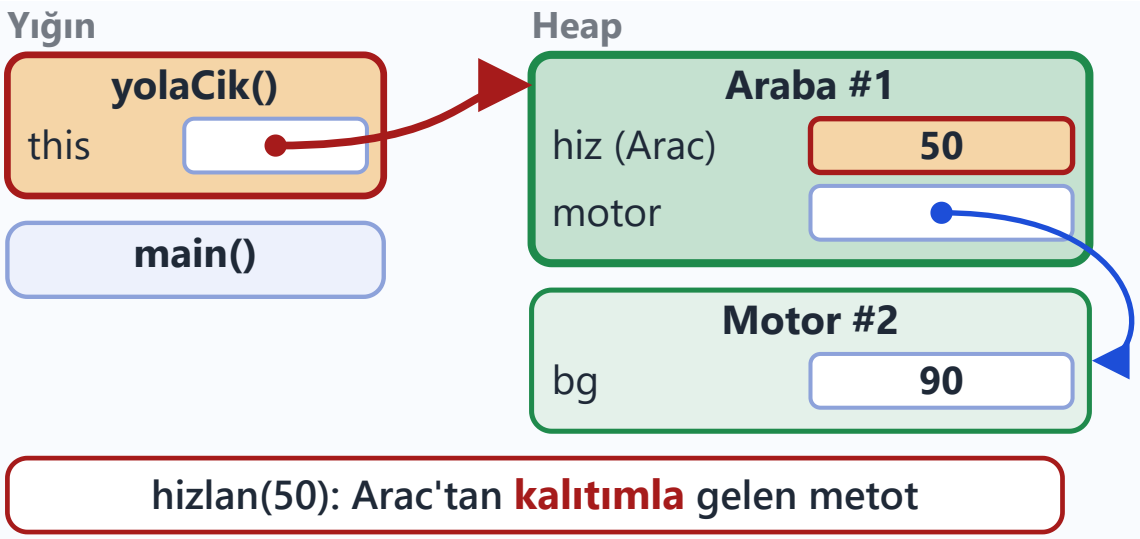
4.3.6.2 Kod izleme: araba ve motor

İki ilişkinin bellekteki farkı bir izlemeyle görülür. Sunumdaki izleme, aynı sınıfların kısaltılmış bir sürümünü kullanır:

```
class Motor {
    private int bg;
    Motor(int bg) { this.bg = bg; }
    void calistir() {
        System.out.println("Motor: " + bg + " BG");
    }
}
class Araba extends Arac {        // is-a
    private Motor motor;           // has-a
    Araba(int bg) { motor = new Motor(bg); }
    void yolaCik() {
        motor.calistir();  hizlan(50);
        System.out.println("Hız: " + hiz);
    }
}
new Araba(90).yolaCik();   // main
```

```
Motor: 90 BG
Hız: 50
```


Yığın
Heap
Araba()
this
bg
90
Araba #1
hiz (Arac)
0
motor
null
main()
Arac kısmı (hiz) + Araba kısmı (motor)
Yığın
Heap
Araba()
this
bg
90
Araba #1
hiz (Arac)
0
motor
main()
Motor #2
bg
90
Yığın
Heap
calistir()
this
yolaCik()
this
main()
Araba #1
hiz (Arac)
0
motor
Motor #2
bg
90
motor.calistir() → işi Motor nesnesi yapar



Şekil 4.3.16: is-a ve has-a izlemesinden dört an. (a) `new Araba(90)` tek bir nesne oluşturur; `Arac`'tan gelen `hiz` bu nesnenin içindedir. (b) Yapıcı ikinci bir nesne, bir `Motor` oluşturur; `motor` alanı onu gösterir. (c) `motor.calistir()` : iş referans izlenerek `Motor`'a devredilir. (d) `hizlan(50)` aynı `Araba` nesnesinin `hiz` alanını 50 yapar.

Şekil 4.3.16 iki ilişkinin bellekteki karşılığını açıkça ayırır. Kalıtım **tek bir nesne** verir. `hiz` alanı ayrı bir `Arac` nesnesinde değil, `Araba` nesnesinin kendisindedir. Oluşum ise **iki ayrı nesne** verir. `Araba` nesnesindeki `motor` alanı, heap'teki başka bir nesneyi gösteren bir referanstır. Araba bir motor değildir, motoru vardır; bağlantı bir referanstır.

Çağrıların akışı da farklıdır. `motor.calistir()` çağrısında `Araba` işi kendisi yapmaz; referansı izleyip `Motor` nesnesine devreder. Yığına `calistir()` çerçevesi eklenir ve bu çerçevedeki `this`, `Motor` nesnesini gösterir. `hizlan(50)` çağrısında ise `this` değişmez; metot `Arac`'ta yazılmıştır, ama aynı `Araba` nesnesi üzerinde çalışır.

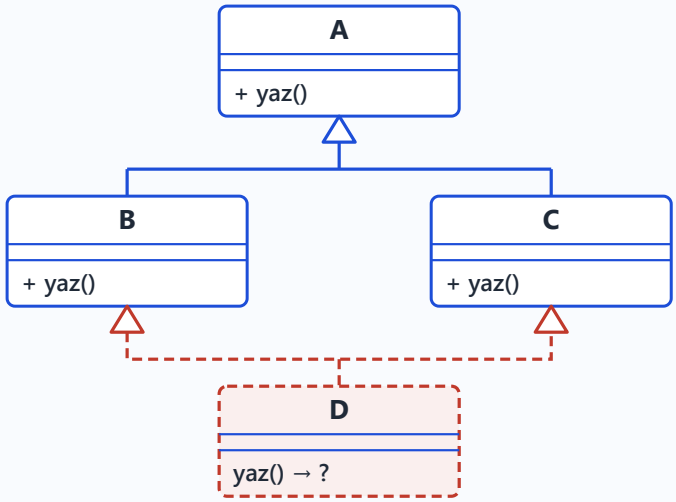
4.3.6.3 Çoklu kalıtım

Java'da bir sınıf yalnız bir sınıftan türeyebilir; doğrudan çoklu kalıtım desteklenmez:

```
class D extends B, C { }   // geçersiz
```

```
error: '{' expected
class D extends B, C { }
                 ^
```

İleti biraz garip görünür. Derleyici "çoklu kalıtım yasaktır" demez, söz dizimi hatası verir. `extends`'ten sonra tek bir sınıf adı bekler ve adın hemen ardından sınıf gövdesinin `{`'ini arar. Virgülü görünce durur.



Şekil 4.3.17: Elmas problemi. `D` hem `B`'den hem `C`'den türeseydi, ikisi de `A`'nın `yaz()` metodunu farklı biçimde geçersiz kıldığında `D`'nin hangisini alacağı belirsiz olurdu.

Yasağın gerekçesi **elmas problemi**dir (*diamond problem*). Şekil 4.3.17 durumu gösterir: `A`'dan `B` ve `C` türer, ikisi de `yaz()`'ı farklı geçersiz kılar. `D` ikisinden birden türeyebilseydi `d.yaz()` hangi sürümü çalıştırırdı? Diyagram bir elmasa benzediği için sorun bu adı alır. Sorunun ikinci bir yüzü de vardır: `A`'nın alanları `D` nesnesinde bir kez mi, iki kez mi bulunmalı? C++ gibi çoklu kalıtıma izin veren diller bu sorulara karmaşık kurallarla cevap verir. Java'nın tasarımcıları sınıflarda çoklu kalıtımı hiç almamayı seçmiştir.

Birden fazla sınıfın yetenekleri gerekiyorsa iki yol vardır. Birincisi oluşumdur: sınıf, yeteneklerini almak istediği nesneleri alan olarak tutar. İkincisi **arayüz**lerdir (*interface*). Bir sınıf tek bir sınıfı genişletirken birçok arayüzü uygulayabilir:

```
interface Sarj { void sarjEt(); }
class ElektrikliAraba extends Araba
                      implements Sarj {
    @Override
    public void sarjEt() {
        System.out.println("Şarj ediliyor.");
    }
}
```

`ElektrikliAraba` hem bir `Araba`'dır hem de şarj edilebilir bir şeydir. Arayüz yalnız bir söz verir: `sarjEt()` diye bir metot olacaktır. Gövdeyi sınıf yazar. Klasik arayüzlerde alan olmadığı için elmas probleminin alan yüzü ortaya çıkmaz.

4.3.6.4 Tasarım ilkeleri: kalıtım mı, oluşum mu?

Kalıtımla tasarımda dört ilke yol gösterir.

- **is-a testi.** "Her X bir Y midir?" cümlesi doğru değilse kalıtım kullanılmaz. Araba bir motor *değildir*, motora *sahiptir*; doğru ilişki oluşumdur.
- **Her sınıf kendi verisini yönetir.** Üst sınıfın alanlarına `super(...)` ve üst sınıfın metotlarıyla değer verilir; alanlar `private` tutulur.
- **Genel metotlar geçersiz kılınır.** `toString` ve `equals`, onunla birlikte de `hashCode`, her anlamlı sınıfta geçersiz kılınmalıdır.
- **Ortak kavramlar soyut sınıfa.** "Hayvan" tek başına bir nesne olmamalıysa, yani hiç kimse belirsiz bir "hayvan" oluşturmamalıysa, alt sınıfların ortak kavramı soyut (*abstract*) bir sınıfla temsil edilir.

Kalıtım ile oluşum arasındaki seçim ilk ilkenin uygulanmasıdır. Tablo 4.3.9 iki yolu birkaç ölçüte göre karşılaştırır.

	Kalıtım (`extends`)	**Oluşum (alan olarak nesne)**
İlişki	is-a: "her X bir Y'dir"	has-a: "X'in bir Y'si vardır"
Bağlılık	Sıkı: alt sınıf, üst sınıfın iç işleyişine de bağlıdır	Gevşek: yalnız içteki nesnenin `public` metotlarına bağlıdır
Yeniden kullanım	Otomatik: bütün `public` metotlar gelir (istenmeyenler de)	Seçerek: yalnız iletilmek istenen metotlar yazılır
Değişiklik	Üst sınıf derleme zamanında sabittir	Alan, çalışma zamanında başka bir nesneyle değiştirilebilir
Üst tip yerine kullanım	Doğrudan: `Hayvan h = new Kedi()`	Ortak bir arayüz uygulanırsa

Tablo 4.3.9: Kalıtım ile oluşumun karşılaştırması.

Kalıtım, is-a ilişkisi gerçekten doğruysa ve üst sınıf genişletilmek üzere tasarlanıp belgelendiyse uygundur. `Kedi` ile `Hayvan` bu duruma uyar. Oluşum, amaç yalnız başka bir sınıfın kodunu kullanmaksa uygundur. Bir yığın (*stack*) sınıfı örnek verilebilir. `Yigin`, öğelerini tutmak için bir `ArrayList`'ten yararlanabilir. Ama bir yığın bir liste *değildir*. `Yigin extends ArrayList` yazılırsa yığın, listenin bütün `public` metotlarını da alır. Biri `add(0, x)` ile yığının ortasına eleman ekleyebilir. Bu, yığının "yalnız tepeden ekle ve çıkar" kuralını bozar. Doğrusu, listeyi bir `private` alanda tutmak ve dışarıya yalnız `ekle` ile `cikar` açmaktır.

Değişiklik satırı da önemli bir fark söyler. `Araba`'nın motoru bir alan olduğu için çalışma sırasında değiştirilebilir. Benzinli motor sökülür, yerine elektrikli motor takılır; ikisi de ortak bir `Motor` arayüzünü uyguluyorsa `Araba`'nın kodu değişmez. Kalıtımda ise `Araba`'nın üst sınıfı kaynak kodda yazılıdır ve bir kez derlendikten sonra değişmez.

4.3.6.5 Kod izleme: kırılgan üst sınıf

Kalıtımın sıkı bağlılığı bazen beklenmedik hatalar doğurur. Aşağıdaki örnekte bir liste sınıfı vardır. Amaç, ona eklenen öğe sayısını tutan bir alt sınıf yazmaktır.

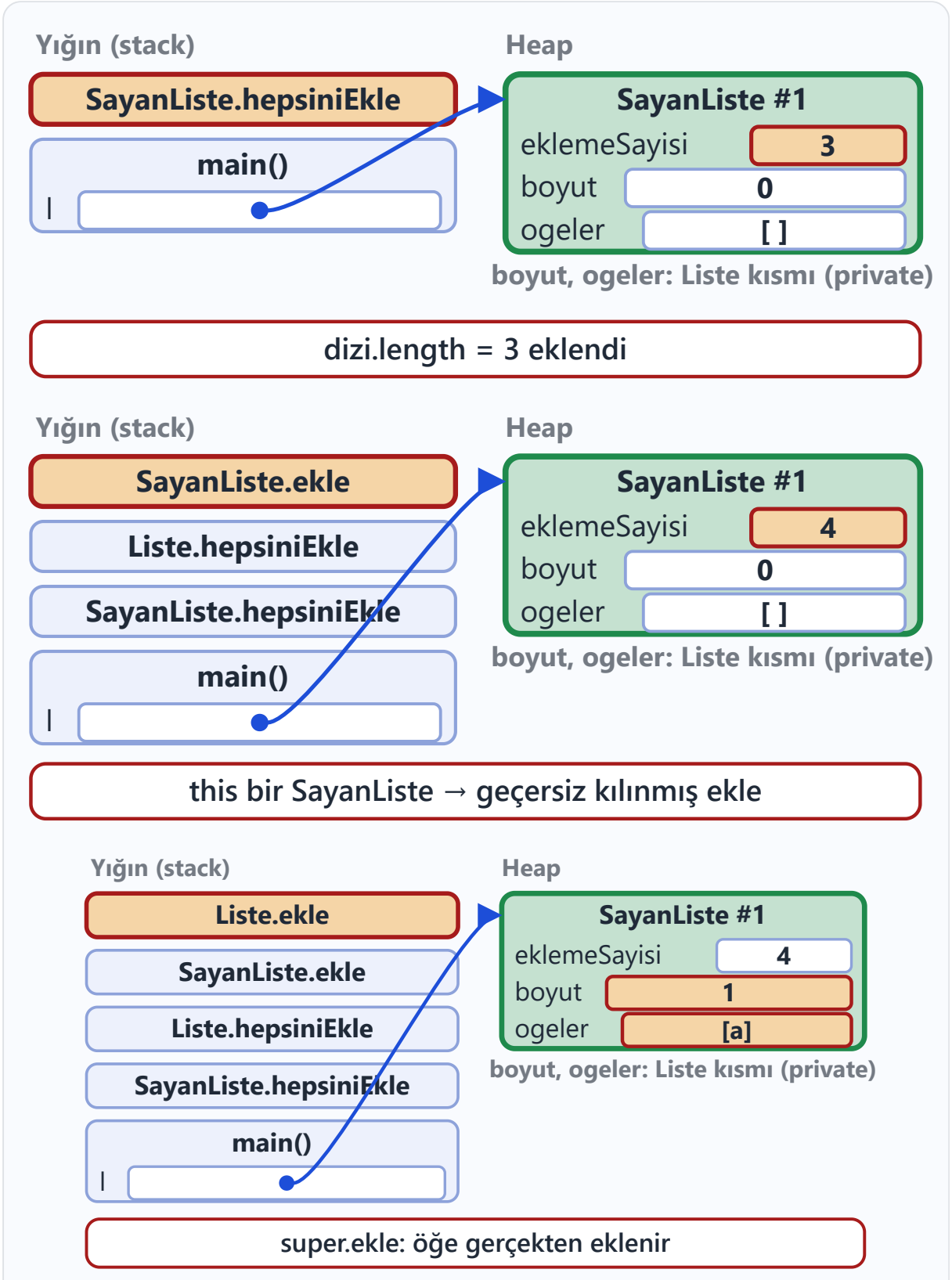
```
class Liste {
    private String[] ogeler = new String[10];
    private int boyut;
    void ekle(String s) { ogeler[boyut++] = s; }
    void hepsiniEkle(String[] dizi) {
        for (String s : dizi) ekle(s);
    }
    int boyut() { return boyut; }
}
class SayanListe extends Liste {
    int eklemeSayisi = 0;
    @Override void ekle(String s) {
        eklemeSayisi++;
        super.ekle(s);
    }
    @Override void hepsiniEkle(String[] dizi) {
        eklemeSayisi += dizi.length;
        super.hepsiniEkle(dizi);
    }
}
SayanListe l = new SayanListe();   // main
l.hepsiniEkle(new String[] {"a", "b", "c"});
System.out.println(l.eklemeSayisi + " " + l.boyut());
```

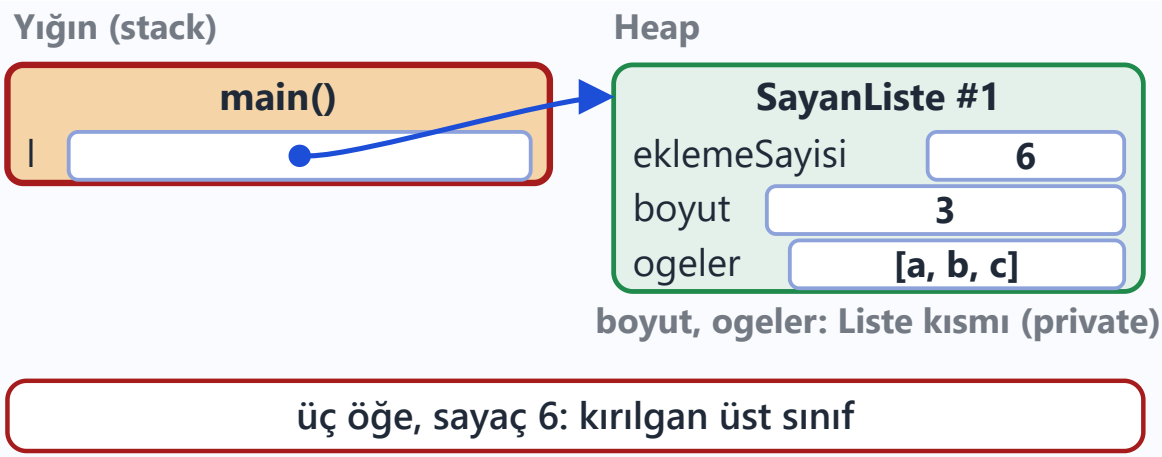
```
6 3
```

Üç öğe eklendi ve listede gerçekten üç öğe var, ama sayaç 6 diyor. `SayanListe`'nin iki metodu da tek başına doğru görünür. `ekle` sayacı bir artırıp işi üst sınıfa bırakır. `hepsiniEkle` sayaca dizinin uzunluğunu ekleyip işi üst sınıfa bırakır. Hatanın kaynağını Tablo 4.3.10 adım adım gösterir.

Adım	Çalışan metot	Olay	eklemeSayisi	boyut
1	`SayanListe.hepsiniEkle`	Nesne bir `SayanListe`; alt sınıfın sürümü çağrılır.	0	0
2	`SayanListe.hepsiniEkle`	`eklemeSayisi += dizi.length`	3	0
3	`Liste.hepsiniEkle`	`super.hepsiniEkle(dizi)`; döngü `ekle("a")`, yani `this.ekle("a")` çağırır.	3	0
4	`SayanListe.ekle`	`this` bir `SayanListe`: geçersiz kılınmış `ekle` çalışır, sayaç artar.	4	0
5	`Liste.ekle`	`super.ekle(s)` öğeyi diziye yazar.	4	1
6	(döngü)	"b" ve "c" için 3–5 yinelenir.	6	3
7	`main`	`println`: yanlış sonuç	6	3

Tablo 4.3.10: `l.hepsiniEkle({"a", "b", "c"})` izlemesi.


Yığın (stack)
Heap
SayanListe.hepsiniEkle
main()
SayanListe #1
eklemeSayisi
3
boyut
0
ogeler
[ ]
boyut, ogeler: Liste kısmı (private)
dizi.length = 3 eklendi
Yığın (stack)
Heap
SayanListe.ekle
Liste.hepsiniEkle
SayanListe.hepsiniEkle
main()
SayanListe #1
eklemeSayisi
4
boyut
0
ogeler
[ ]
boyut, ogeler: Liste kısmı (private)
this bir SayanListe → geçersiz kılınmış ekle
Yığın (stack)
Heap
Liste.ekle
SayanListe.ekle
Liste.hepsiniEkle
SayanListe.hepsiniEkle
main()
SayanListe #1
eklemeSayisi
4
boyut
1
ogeler
[a]
boyut, ogeler: Liste kısmı (private)
super.ekle: öğe gerçekten eklenir



Şekil 4.3.18: Kırılgan üst sınıf izlemesinden dört an. (a) Alt sınıfın `hepsiniEkle`'si sayaca 3 ekler. (b) Üst sınıfın döngüsündeki `ekle(s)`, nesne bir `SayanListe` olduğu için alt sınıfa döner; aynı öğe ikinci kez sayılır. (c) `super.ekle` öğeyi yazar: `boyut` = 1. (d) Sonuç 6 3.

Hata üçüncü adımda doğar. `Liste.hepsiniEkle`, işini her öğe için `ekle(s)` çağırarak yapar. Bu, `Liste`'nin bir **gerçekleme ayrıntısıdır**; sınıfın belgesinde yazmayabilir. `Liste`'nin kodundaki `ekle(s)` aslında `this.ekle(s)`'dir ve `this` bir `SayanListe`'dir. Çağrı geçersiz kılınmış sürüme gider. Her öğe bir kez `hepsiniEkle`'de, bir kez de `ekle`'de sayılır.

Tanım 4.3.8

Kırılgan üst sınıf (*fragile base class*) sorunu, alt sınıfın doğruluğunun üst sınıfın iç işleyişine bağlı olmasıdır. Üst sınıfta, dışarıdan görünen davranışı değiştirmeyen bir düzenleme bile alt sınıfı bozabilir.

Sorunun kırılganlık yönü şudur. `SayanListe`'nin programcısı hatayı fark edip `hepsiniEkle`'nin geçersiz kılınmasını silebilir. Sayaç o zaman doğru çıkar, çünkü her öğe yalnız `ekle`'de sayılır. Ama bu düzeltme `Liste`'nin o günkü koduna dayanır. `Liste`'nin yeni bir sürümü hız için `hepsiniEkle`'yi diziyi doğrudan kopyalayacak biçimde yazarsa `ekle` hiç çağrılmaz ve sayaç 0'da kalır. İki durumda da `Liste`'nin kullanıcılarına sunduğu davranış aynıdır, ama `SayanListe` bozulur.

Örnek 4.3.6

Yeni bir `SayanListe` üzerinde önce `ekle("x")`, sonra `hepsiniEkle(new String[] {"y", "z"})` çağrılıyor. `eklemeSayisi` ve `boyut()` nedir?

`ekle("x")` doğrudan alt sınıfın `ekle`'sini çalıştırır: sayaç 1, boyut 1. `hepsiniEkle` önce sayaca 2 ekler: sayaç 3. Sonra üst sınıfın döngüsü iki kez `ekle` çağırır ve her biri alt sınıfa döner: sayaç 5, boyut 3. Sonuç **5 3**'tür. Yalnız `ekle` kullanılsaydı sonuç doğru çıkardı; hata `hepsiniEkle` ile `ekle`'nin birlikte geçersiz kılınmasından doğar.

4.3.6.6 Oluşumla çözüm: iletme

Kırılgan üst sınıf sorununun temiz çözümü kalıtımı bırakıp oluşuma geçmektir:

```
class SayanListe2 {
    private final Liste liste = new Liste(); // has-a
    private int eklemeSayisi = 0;
    void ekle(String s) {
        eklemeSayisi++;
        liste.ekle(s);            // iletir
    }
    void hepsiniEkle(String[] dizi) {
        eklemeSayisi += dizi.length;
        liste.hepsiniEkle(dizi);  // iletir
    }
    int getEklemeSayisi() { return eklemeSayisi; }
    int boyut() { return liste.boyut(); }
}
```

```
SayanListe2 l2 = new SayanListe2();
l2.hepsiniEkle(new String[] {"a", "b", "c"});
System.out.println(l2.getEklemeSayisi()
    + " " + l2.boyut());
```

```
3 3
```

Tanım 4.3.9

İletme (*forwarding*), oluşumda gelen bir çağrının içteki nesnenin metoduna aktarılmasıdır. Dıştaki sınıfın her metodu önce kendi işini yapar, sonra çağrıyı içteki nesneye iletir.

Yeni sınıf bir `Liste`'den türemez; bir `Liste`'yi içinde tutar. `liste.hepsiniEkle(...)` çağrısının içindeki `ekle` çağrıları, düz bir `Liste` nesnesi üzerinde çalışır. O nesnenin `this`'i bir `Liste`'dir; çağrı `SayanListe2`'ye geri dönmez. Sayaç doğrudur: 3. `Liste`'nin iç ayrıntısı değişse de sonuç değişmez. `SayanListe2` yalnız `Liste`'nin `public` metotlarına bağlıdır.

Bu çözümün bir bedeli vardır. `SayanListe2` bir `Liste` değildir; bir `Liste` beklenen yere verilemez. Gerekirse `Liste` ile `SayanListe2` ortak bir arayüzü uygular ve kod o arayüz tipiyle yazılır. İletme yöntemi Java kütüphanesinde de yaygındır. Örneğin `Collections.unmodifiableList(...)` bir listeyi saran ve okuma çağrılarını ona ileten bir nesne döndürür.

4.3.6.7 Yerine koyma ilkesi

is-a testi gündelik dilde yapıldığında yanıltıcı olabilir. Matematikte her kare bir dikdörtgendir. Bu cümleye dayanarak `Kare extends Dikdortgen` yazmak doğal görünür:

```
class Dikdortgen {
    protected int en, boy;
    void setEn(int en) { this.en = en; }
    void setBoy(int boy) { this.boy = boy; }
    int alan() { return en * boy; }
}
class Kare extends Dikdortgen {
    @Override void setEn(int k) { en = k; boy = k; }
    @Override void setBoy(int k) { en = k; boy = k; }
}
static void genislet(Dikdortgen d) {
    d.setEn(5);
    d.setBoy(4);
    System.out.println("Beklenen 20, bulunan "
        + d.alan());
}
// main içinde:
genislet(new Dikdortgen());
genislet(new Kare());
```

```
Beklenen 20, bulunan 20
Beklenen 20, bulunan 16
```

Tanım 4.3.10

Liskov yerine koyma ilkesi (*Liskov substitution principle*, LSP): üst sınıf tipiyle yazılmış kod, bir alt sınıfın nesnesiyle de doğru çalışmalıdır. Kodun derlenmesi yetmez; davranışın da doğru kalması gerekir.

`genislet` metodu bir dikdörtgenin enini 5, boyunu 4 yapar ve alanın 20 olmasını bekler. Düz bir `Dikdortgen` ile beklenti tutar. Bir `Kare` ile ise `setEn(5)` iki kenarı da 5 yapar, ardından `setBoy(4)` iki kenarı da 4 yapar. Alan 16 çıkar. Kod derlendi, hiçbir istisna oluşmadı, ama sonuç yanlış.

Hata nerededir? `genislet` hatalı değildir. `Dikdortgen` sınıfı “en ve boy birbirinden bağımsız değişir” sözünü verir: `setEn` boya dokunmaz. `genislet` bu söze güvenmiştir. `Kare` ise bu sözü tutamaz, çünkü karenin iki kenarı birbirinden bağımsız olamaz. Hata, sözü bozan alt sınıftadır. Matematiksel ilişki doğrudur, ama sınıflar arasındaki ilişki yanlıştır.

Bu sorunun iki yaygın çözümü vardır. Birincisi `Kare`'yi `Dikdortgen`'den türetmemektir; ikisi de ortak bir `Sekil`'den türer ve her biri kendi `alan()`'ını yazar. İkincisi sınıfları değiştirilemez yapmaktır. En ve boy yalnız yapıcıda verilir, `setEn` ve `setBoy` hiç olmaz. O zaman "en değişince boy değişmez" diye bir söz de olmaz ve kare gerçekten bir dikdörtgen gibi davranabilir.

Ders şudur: is-a testi gerçek dünyaya değil, sınıfın **davranışına** uygulanır. Doğru soru "kare bir dikdörtgen midir?" değil, "`Kare`, `Dikdortgen`'in her metoduna onun gibi cevap verebilir mi?" sorusudur.

Örnek 4.3.7

`Dikdortgen d = new Kare(); d.setEn(3); d.setBoy(7);` satırlarından sonra `d.alan()` kaçtır?

Değişkenin tipi `Dikdortgen`, nesne bir `Kare`'dir. Çağrılar geçersiz kılınmış sürümlere gider. `setEn(3)`: en = boy = 3. `setBoy(7)`: en = boy = 7. Alan 7 × 7 = **49**'dur. Son çağrı her iki kenarı belirler. Düz bir dikdörtgende sonuç 3 × 7 = 21 olurdu.

4.3.7 `Object` sınıfı

Java'daki her sınıf, yazılsın ya da yazılmasın, `Object`'ten türer. Bu kesim `Object`'in her nesneye verdiği metotları ve bunların en önemli üçünü, `toString`, `equals` ve `hashCode`'u, doğru geçersiz kılmayı anlatır.

4.3.7.1 Her sınıfın atası

Her sınıf `java.lang.Object` sınıfından türer. `extends` yazılmamışsa dolaylı olarak:

```
class Nokta { }
// derleyicinin gördüğü:
class Nokta extends Object { }
```

Bu yüzden her nesnede bazı metotlar hazır bulunur. Tablo 4.3.11 en çok kullanılan dördünü verir. Bu metotların varsayılan sürümleri genel amaçlıdır ve çoğu zaman işe yaramaz. Anlamlı sınıflarda ilk üçü geçersiz kılınır.

Metot	Varsayılan davranış
`String toString()`	Sınıf adı + `@` + `hashCode` değerinin onaltılık (hex) yazımı
`boolean equals(Object o)`	Referansları karşılaştırır: `this == o`
`int hashCode()`	Nesneye özgü bir tam sayı (genellikle her nesnede farklı)

Metot	Varsayılan davranış
`Class getClass()`	Nesnenin çalışma zamanındaki sınıfı (`final`; geçersiz kılınamaz)

Tablo 4.3.11: `Object` sınıfının en sık kullanılan metotları.

`Object`'in başka metotları da vardır: iş parçacıklarında kullanılan `wait`, `notify` ve `notifyAll`, nesne kopyalamaya yarayan `clone` gibi. Bunlar bu notun konusu değildir. `Object` tipindeki bir değişken her nesneyi gösterebilir. Bu yüzden `equals`'ın parametresi `Object`'tir: bir nesne, herhangi bir başka nesneyle karşılaştırılabilmelidir.

4.3.7.2 `toString` metodu

`public String toString()` nesneyi metin olarak gösterir. `System.out.println(nesne)` ve `"..." + nesne` biçimindeki birleştirme bu metodu kendiliğinden çağırır. Geçersiz kılınmazsa sınıf adını ve nesnenin `hashCode` değerini içeren bir metin döner:

```
class Nokta {
    private int x, y;
    Nokta(int x, int y) {
        this.x = x; this.y = y;
    }
}
// main içinde
System.out.println(new Nokta(3, 4));
```

```
Nokta@28a418fc
```

`@` işaretinden sonraki `28a418fc`, `hashCode()` değerinin onaltılık yazımıdır. Değer her çalıştırmada ya da her JVM'de farklı olabilir. Nesnenin içeriği hakkında hiçbir şey söylemez; hata ayıklarken bile işe yaramaz. Geçersiz kılınınca nesne okunaklı yazılır:

```
class Nokta {
    private int x, y;
    Nokta(int x, int y) {
        this.x = x; this.y = y;
    }
    @Override
    public String toString() {
        return "Nokta[x=" + x + ", y=" + y + "]";
    }
}
// main içinde
Nokta n = new Nokta(3, 4);
System.out.println(n);
System.out.println("Konum: " + n);
```

```
Nokta[x=3, y=4]
Konum: Nokta[x=3, y=4]
```

İkinci satırda `+` işlecinin bir yanı `String` olduğu için Java öbür yanı da metne çevirir ve bunun için `toString()`'i çağırır. `toString()`'in dönüş tipi `String`, erişimi `public`'tir. Geçersiz kılarken `public` yazmayı unutmak, Tablo 4.3.6'da görülen erişimi daraltma hatasını verir.

Bir ayrıntı daha vardır. `null` bir referans yazdırılırsa `toString` çağrılmaz; "null" yazılır:

```
Nokta n = null;
System.out.println(n);
System.out.println("Konum: " + n);
```

```
null
Konum: null
```

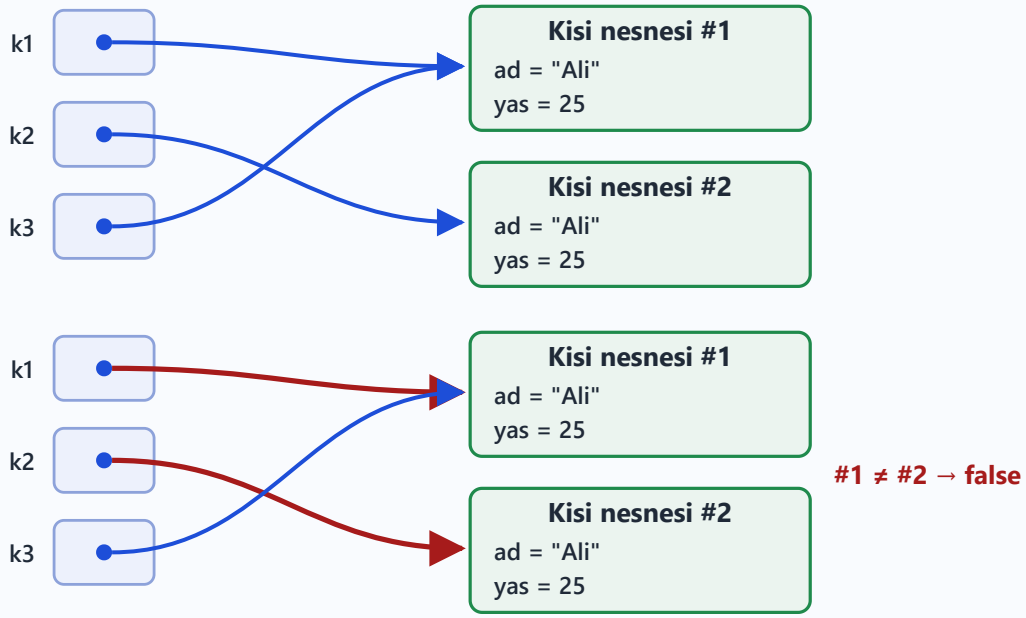
`println` ve dizgi birleştirme `null`'ı özel olarak ele alır. Ama `n.toString()` açıkça çağrılsaydı `NullPointerException` oluşurdu; `null` referans üzerinde metot çağrılamaz.

4.3.7.3 `==` mi, `equals` mı?

İki nesneyi karşılaştırmanın iki yolu vardır ve ikisi farklı sorular sorar. `==`, "bu iki referans aynı nesneyi mi gösteriyor?" diye sorar. `equals` ise "bu iki nesne eşdeğer mi?" diye sorar. Aşağıdaki `Kisi` sınıfında `equals`, adı ve yaşı karşılaştıracak biçimde geçersiz kılınmış olsun:

```
Kisi k1 = new Kisi("Ali", 25);
Kisi k2 = new Kisi("Ali", 25);
Kisi k3 = k1;
System.out.println(k1 == k2);
System.out.println(k1 == k3);
System.out.println(k1.equals(k2));
```

```
false
true
true
```



Şekil 4.3.19: `==` ile `equals` izlemesinden iki an. (a) `k3 = k1` yalnız adresi kopyalar; `k1` ve `k3` aynı nesneyi gösterir. (b) `k1 == k2` : iki ayrı nesne, sonuç `false` .

Satırları sırayla izleyelim. İlk iki `new` heap'te içeriği aynı iki ayrı nesne oluşturur; `k1` ve `k2` bunların adreslerini tutar. `k3 = k1` yeni nesne oluşturmaz; yalnız adresi kopyalar. Artık `k1` ve `k3` aynı nesneyi gösterir (Şekil 4.3.19). `k1 == k2` iki farklı adresi karşılaştırır ve `false` verir. `k1 == k3` aynı adresi karşılaştırır ve `true` verir. `k1.equals(k2)` ise içerikleri karşılaştırır: iki nesnenin de adı "Ali", yaşı 25'tir, sonuç `true` 'dur.

`equals` geçersiz kılınmamış olsaydı son satır `false` verirdi. `Object` 'in `equals` 'ı `this == o` 'dan ibarettir. `String` karşılaştırmasında da kural aynıdır: `s1 == s2` değil, `s1.equals(s2)` yazılır. `new String("Java") == new String("Java")` `false` , `equals` ise `true` verir.

4.3.7.4 `equals` sözleşmesi

`public boolean equals(Object other)`, iki nesnenin eşdeğer olup olmadığını söyler. Varsayılan olarak yalnız referansları karşılaştırır; geçersiz kılınmazsa `==` gibi davranır. Geçersiz kılınınca metot iki nesnenin durumunu, yani alanlarını karşılaştırır ve eşitse `true` döner.



Şekil 4.3.20: İki ayrı nesne aynı içerikte. `==` ve varsayılan `equals` adresleri karşılaştırır ve `false` verir. Geçersiz kılınmış `equals` alanları karşılaştırır ve `true` verir.

`equals` keyfî yazılamaz. Java kütüphanesi, özellikle koleksiyonlar, `equals`'ın belirli kurallara uyduğunu varsayar. Bu kurallara `equals` sözleşmesi denir (Tablo 4.3.12).

Özellik	Kural
Yansıma	`x.equals(x)` → `true`
Simetri	`x.equals(y)` ⇔ `y.equals(x)`
Geçişme	`x` = `y` ve `y` = `z` ise `x` = `z`
Tutarlılık	Alanlar değişmedikçe sonuç değişmez
null	`x.equals(null)` → `false`

Tablo 4.3.12: `equals` sözleşmesi (`x`, `y`, `z` `null` olmayan referanslar).

Bu kurallar matematikteki eşitliğin özellikleridir. Bozulduklarında hatalar dolaylı yoldan ortaya çıkar. Simetri bozuksa `liste.contains(x)` bir sırayla `true`, öbür sırayla `false` verebilir. Tutarlılık bozuksa bir `HashSet`'e eklenen nesne sonradan bulunamaz.

4.3.7.5 `instanceof` ile `equals`

`equals`'ın parametresi `Object` tipindedir. Gelen nesne bir `Nokta` olmayabilir; bir dizgi, bir kişi ya da `null` olabilir. Bu yüzden alanlara geçmeden önce tip kontrolü yapılır:

```
public boolean equals(Object nesne) {
    if (nesne instanceof Nokta) {
        // nesne bir Nokta nesnesidir;
        // dönüştürüp karşılaştır
        Nokta diger = (Nokta) nesne;
        return x == diger.x && y == diger.y;
    } else {
        // nesne bir Nokta değildir;
        // eşit olamaz
        return false;
    }
}
```

Metot önce `nesne instanceof Nokta` ile gelen nesnenin bir `Nokta` olup olmadığını sorar. Öyleyse `(Nokta) nesne` ile aşağı dönüşüm yapar. Dönüşüm yalnız tip kontrolünden sonra güvenlidir. Sonra iki koordinatı karşılaştırır. Nesne bir `Nokta` değilse eşit olamaz ve `false` döner.

```
Nokta a = new Nokta(3, 4);
Nokta b = new Nokta(3, 4);
System.out.println(a == b);
System.out.println(a.equals(b));
System.out.println(a.equals("3,4"));
System.out.println(a.equals(null));
```

```
false
true
false
false
```

İlk satır iki ayrı nesneyi `==` ile karşılaştırır: `false`. İkincisi içerikleri karşılaştırır: `true`. Üçüncüsünde `"3,4"` bir `Nokta` değil, bir `String`'dir; `instanceof` yanlış verir ve `false` döner. Dönüşüme hiç gelinmediği için `ClassCastException` da oluşmaz. Dördüncüsünde `null instanceof Nokta` yanlıştır. `instanceof`, `null` için her zaman `false` verir. Bu yüzden `instanceof` kullanan bir `equals`'ta ayrı bir `null` kontrolüne gerek kalmaz.

Parametre tipi `Object` olmalıdır. `equals(Nokta n)` yazmak geçersiz kılma değil, aşırı yüklemedir. Bu hata derleyicinin gözünden kaçar ve sinsi sonuçlar doğurur:

```
class NoktaA {
    int x, y;
    NoktaA(int x, int y) { this.x = x; this.y = y; }
    public boolean equals(NoktaA n) {   // aşırı yükleme!
        return x == n.x && y == n.y;
    }
}
// main içinde:
NoktaA p = new NoktaA(1, 2), q = new NoktaA(1, 2);
Object qo = q;
System.out.println(p.equals(q) + " " + p.equals(qo));
List<NoktaA> liste = new ArrayList<>();
liste.add(p);
System.out.println(liste.contains(q));
```

```
true false
false
```

`p.equals(q)` doğru sonucu verir, çünkü `q`'nun derleme anındaki tipi `NoktaA`'dır ve derleyici `equals(NoktaA)`'yı seçer. `p.equals(qo)`'da ise argümanın tipi `Object`'tir. Derleyici `Object`'ten gelen `equals(Object)`'i seçer, o da adresleri karşılaştırır: `false`. Kütüphane sınıfları her zaman `equals(Object)`'i çağırır. `ArrayList.contains` bu yüzden aynı içerikli noktayı bulamaz. `@Override` yazılsaydı hata derleme anında yakalanırdı: `method does not override or implement a method from a supertype`.

4.3.7.6 `getClass()` ile `equals` ve izleme

Tip kontrolünün ikinci bir yolu `getClass()`'tır:

```
public class Kisi {
    private String ad;
    private int yas;
    // ... yapıcı ...

    @Override
    public boolean equals(Object obj) {
        // aynı referanssa eşittir
        if (this == obj) return true;
        // null ya da farklı sınıf: eşit değil
        if (obj == null
                || getClass() != obj.getClass())
            return false;
        // alanları karşılaştır
        Kisi kisi = (Kisi) obj;
        return yas == kisi.yas
               && ad.equals(kisi.ad);
    }
}
```

Bu kalıp üç adımlıdır. Önce `this == obj` denetlenir. Aynı nesneyse alanlara bakmaya gerek yoktur ve hemen `true` döner; bu hızlı bir kestirmedir. Sonra `obj == null` ve sınıf denetimi yapılır. `getClass()` nesnenin gerçek sınıfını verir; iki tarafın sınıfı birebir aynı olmalıdır. Son olarak alanlar karşılaştırılır. İlkel tipler (`int yas`) `==` ile, nesne tipindeki alanlar (`String ad`) `equals` ile karşılaştırılır.

İki kalıp alt sınıf nesneleri için farklı davranır (Tablo 4.3.13).

Tip kontrolü	Alt sınıf nesnesi eşit sayılabilir mi?
`instanceof`	Evet (bir `Kedi` nesnesi bir `Hayvan` 'dır)
`getClass()`	Hayır (sınıflar birebir aynı olmalı)

Tablo 4.3.13: `instanceof` ve `getClass()` ile tip kontrolü.

```
class Ogrenci extends Kisi {         // Kisi'nin equals'ı getClass() kullanır
    Ogrenci(String ad, int yas) { super(ad, yas); }
}
// main içinde:
System.out.println(new Kisi("Ali", 20).equals(new Ogrenci("Ali", 20)));
```

```
false
```

`Kisi`'nin `equals`'ı `getClass()` kullandığı için aynı ad ve yaştaki bir `Ogrenci`'yi eşit saymaz. `instanceof` kullanılsaydı `true` verirdi. Hangisinin doğru olduğu tasarıma bağlıdır. `getClass()` simetriyi kendiliğinden korur. `instanceof` ise alt sınıflar `equals`'ı yeniden yazmadıkça güvenlidir; alt sınıf yeni alanlarla `equals`'ı geçersiz kılarsa simetri bozulabilir.

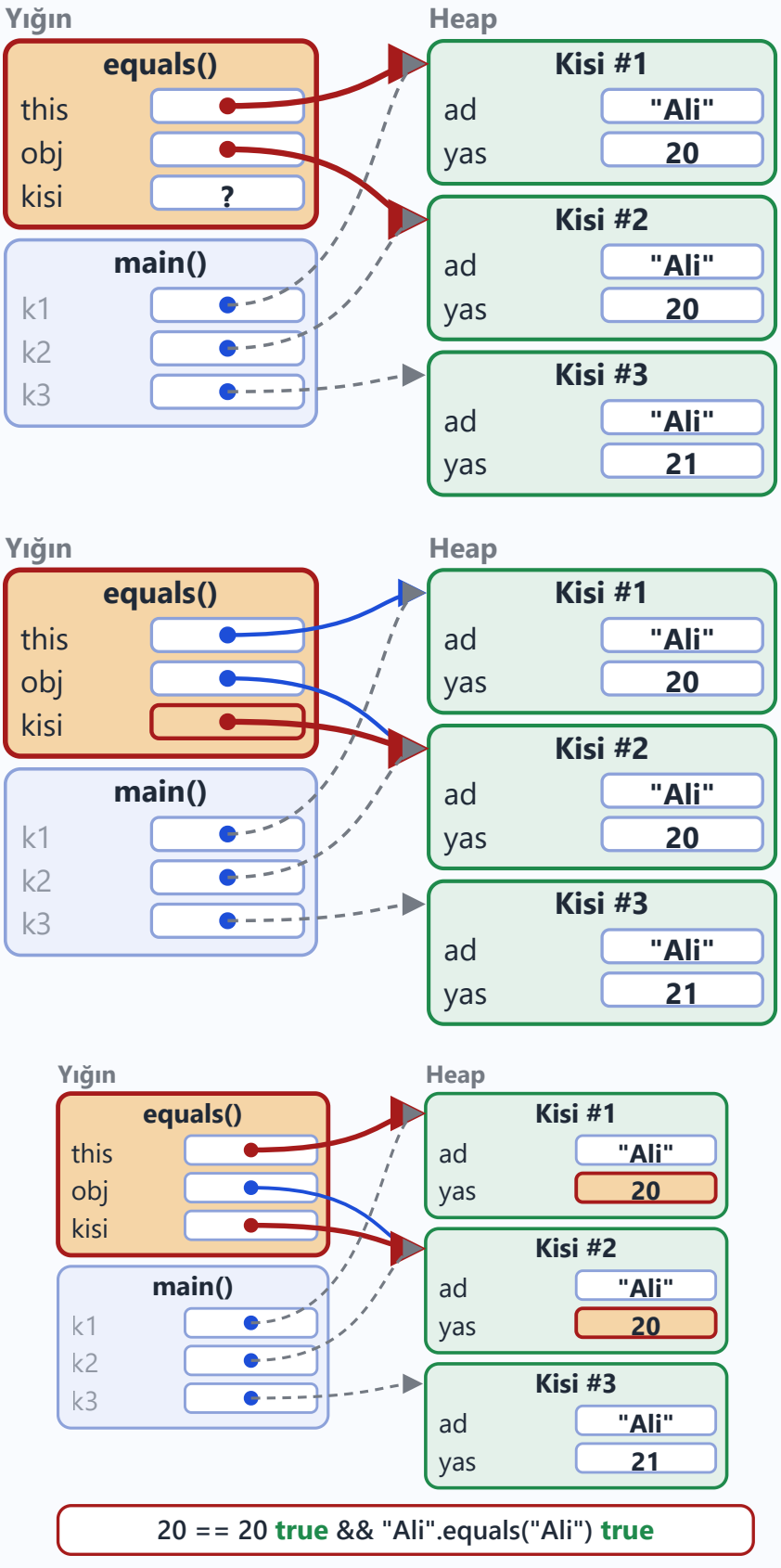
Şimdi `getClass()`'lı `equals`'ı bir izlemeyle çalıştıralım:

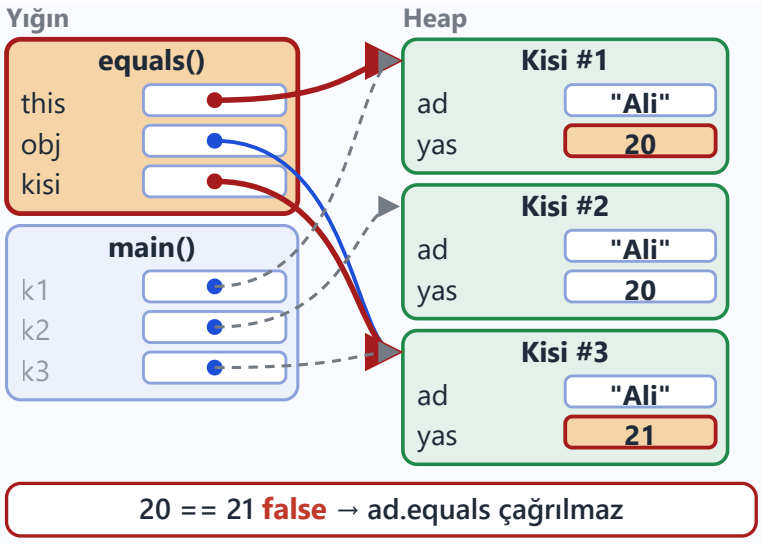
```
public boolean equals(Object obj) {
    if (this == obj) return true;
    if (obj == null || getClass() != obj.getClass())
        return false;
    Kisi kisi = (Kisi) obj;
    return yas == kisi.yas && ad.equals(kisi.ad);
}
// main içinde:
Kisi k1 = new Kisi("Ali", 20);
Kisi k2 = new Kisi("Ali", 20);
Kisi k3 = new Kisi("Ali", 21);
System.out.println(k1.equals(k2));
System.out.println(k1.equals(k3));
System.out.println(k1.equals(null));
```

```
true
false
false
```

Çağrı	`this == obj`	`null` / sınıf denetimi	`yas == kisi.yas`	`ad.equals(...)`	Sonuç
`k1.equals(k2)`	#1 ≠ #2: yanlış	geçti (`Kisi` = `Kisi`)	20 == 20: doğru	"Ali" = "Ali": doğru	`true`
`k1.equals(k3)`	#1 ≠ #3: yanlış	geçti	20 == 21: yanlış	çağrılmaz (kısa devre)	`false`
`k1.equals(null)`	yanlış	`obj == null` doğru; `getClass` çağrılmaz	—	—	`false`

Tablo 4.3.14: Üç `equals` çağrısının izlemesi. #1, #2 ve #3 nesnelerin adresleridir.


Yığın
equals()
this
obj
kisi
?
main()
k1
k2
k3
Heap
Kisi #1
ad
"Ali"
yas
20
Kisi #2
ad
"Ali"
yas
20
Kisi #3
ad
"Ali"
yas
21
Yığın
equals()
this
obj
kisi
main()
k1
k2
k3
Heap
Kisi #1
ad
"Ali"
yas
20
Kisi #2
ad
"Ali"
yas
20
Kisi #3
ad
"Ali"
yas
21
Yığın
equals()
this
obj
kisi
main()
k1
k2
k3
Heap
Kisi #1
ad
"Ali"
yas
20
Kisi #2
ad
"Ali"
yas
20
Kisi #3
ad
"Ali"
yas
21
20 == 20 true && "Ali".equals("Ali") true



Şekil 4.3.21: `equals` izlemesinden dört an. (a) `k1.equals(k2)` : yeni çerçevede `this` = #1, `obj` = #2. (b) Aşağı dönüşüm yeni nesne üretmez; `kisi` aynı #2'yi `Kisi` tipiyle gösterir. (c) Yaş ve ad eşit: `true` . (d) `k1.equals(k3)` : yaşlar farklı, `&&` kısa devre yapar.

İzlemenin üç ayrıntısı önemlidir. Birincisi, `obj` 'nin tipi `Object` 'tir. `obj.yas` yazılamaz, çünkü `Object` 'te `yas` diye bir alan yoktur. Aşağı dönüşüm, aynı nesneye `Kisi` penceresinden bakan `kisi` referansını verir (Şekil 4.3.21, b). İkincisi, `&&` kısa devre yapar. Yaşlar farklıysa `ad.equals` hiç çağrılmaz. Ucuz karşılaştırmanın önce yazılması bu yüzden iyi bir alışkanlıktır. Üçüncüsü, `||` da kısa devre yapar. `obj == null` doğruysa sağ taraf değerlendirilmez. Sıra ters yazılsaydı (`getClass() != obj.getClass() || obj == null`) `null` argümanla `obj.getClass()` çalışır ve `NullPointerException` oluşurdu.

Maliyet açısından `equals` sabit sayıda alan karşılaştırır. Alanlardan biri `String` ise onun `equals` 'ı dizginin uzunluğuyla orantılı iş yapar. `this == obj` kestirmesi, bir nesnenin kendisiyle karşılaştırıldığı sık durumda bu işi tamamen atlar.

4.3.7.7 `equals` ve `hashCode`

`equals` 'ı geçersiz kılmanın bir yükümlülüğü vardır. Kural şudur: `equals` geçersiz kılınıyorsa `hashCode` da geçersiz kılınmalıdır. Eşit nesnelerin `hashCode` değerleri aynı olmalıdır.

Tanım 4.3.11

Hash kodu (*hash code*), `hashCode()` metodunun bir nesne için döndürdüğü tam sayıdır. Sözleşmesi: `a.equals(b)` doğruysa `a.hashCode() == b.hashCode()` olmalıdır. Tersi gerekmez; eşit olmayan iki nesnenin hash kodu aynı olabilir.

```
import java.util.Objects;
public class Nokta {
    private int x, y;
    public Nokta(int x, int y) {
        this.x = x; this.y = y;
    }
    @Override public boolean equals(Object o) {
        return (o instanceof Nokta n)
               && x == n.x && y == n.y;
    }
    @Override public int hashCode() {
        return Objects.hash(x, y);
    }
    @Override public String toString() {
        return "Nokta[x=" + x + ", y=" + y + "]";
    }
}
```

```
HashSet<Nokta> kume = new HashSet<>();
kume.add(new Nokta(3, 4));
kume.add(new Nokta(3, 4));
System.out.println(kume.size() + " "
    + kume.contains(new Nokta(3, 4)));
```

```
1 true
```

Bu `Nokta` sınıfı üç metodu da geçersiz kılar. `equals`'ta `o instanceof Nokta n` biçimi kullanılmıştır. Bu, Java 16'dan beri bulunan **örüntü eşleme**dir: denetim doğruysa `n` dönüştürülmüş referansı tutar. Klasik biçimi `if (o instanceof Nokta) { Nokta n = (Nokta) o; … }`'tır. `hashCode` ise `java.util.Objects.hash(x, y)` ile alanlardan bir değer üretir.

Neden iki metot birlikte gerekir? `HashSet` ve `HashMap` bir öğeyi ararken önce `hashCode` ile bir "kova" seçer, sonra yalnız o kovadaki öğelerle `equals` çağırır. `hashCode` farklıysa eşit nesneler farklı kovalara düşer ve hiç karşılaştırılmaz. Tablo 4.3.15 iki durumu karşılaştırır.

`Nokta` sınıfında	Çıktı	
yalnız `equals`	`2 false`	✗
`equals` + `hashCode`	`1 true`	✓

Tablo 4.3.15: `kume.size() + " " + kume.contains(new Nokta(3, 4))` çıktısı.

Yalnız `equals` yazıldığında `Object`'in `hashCode`'u kullanılır ve iki ayrı nesneye pratikte farklı değerler verir. İkinci `new Nokta(3, 4)` başka bir kovaya düşer. Küme onu ilkiyle karşılaştırmaz ve ikinci kez ekler: boyut 2. `contains` da üçüncü bir kovaya bakar ve bulamaz: `false`. Derleme hatası yoktur; derleyici iki metodun uyumunu denetlemez. Hata yalnız çalışma anında, yanlış sonuçla görülür.

> **Sık yapılan hata**
>
> `equals`'ı geçersiz kılıp `hashCode`'u unutmak. Sınıf bir `ArrayList`'te sorunsuz çalışır, çünkü `ArrayList.contains` yalnız `equals` kullanır. Aynı nesneler bir `HashSet`'e ya da bir `HashMap`'in anahtarına konunca hatalar başlar: yinelenen öğeler, bulunamayan anahtarlar.

4.3.7.8 Kod izleme: `HashSet` içinde

Kovaların nasıl seçildiğini somut sayılarla izlemek mümkündür. Aşağıdaki kod klasik `equals` biçimini kullanır:

```
public boolean equals(Object o) {
    if (!(o instanceof Nokta)) return false;
    Nokta n = (Nokta) o;
    return x == n.x && y == n.y;
}
public int hashCode() {
    return Objects.hash(x, y);  // 31·(31 + x) + y
}
// main içinde:
HashSet<Nokta> kume = new HashSet<>();
kume.add(new Nokta(3, 4));
kume.add(new Nokta(4, 3));
kume.add(new Nokta(3, 4));
System.out.println(kume.size());
System.out.println(kume.contains(new Nokta(4, 3)));
```

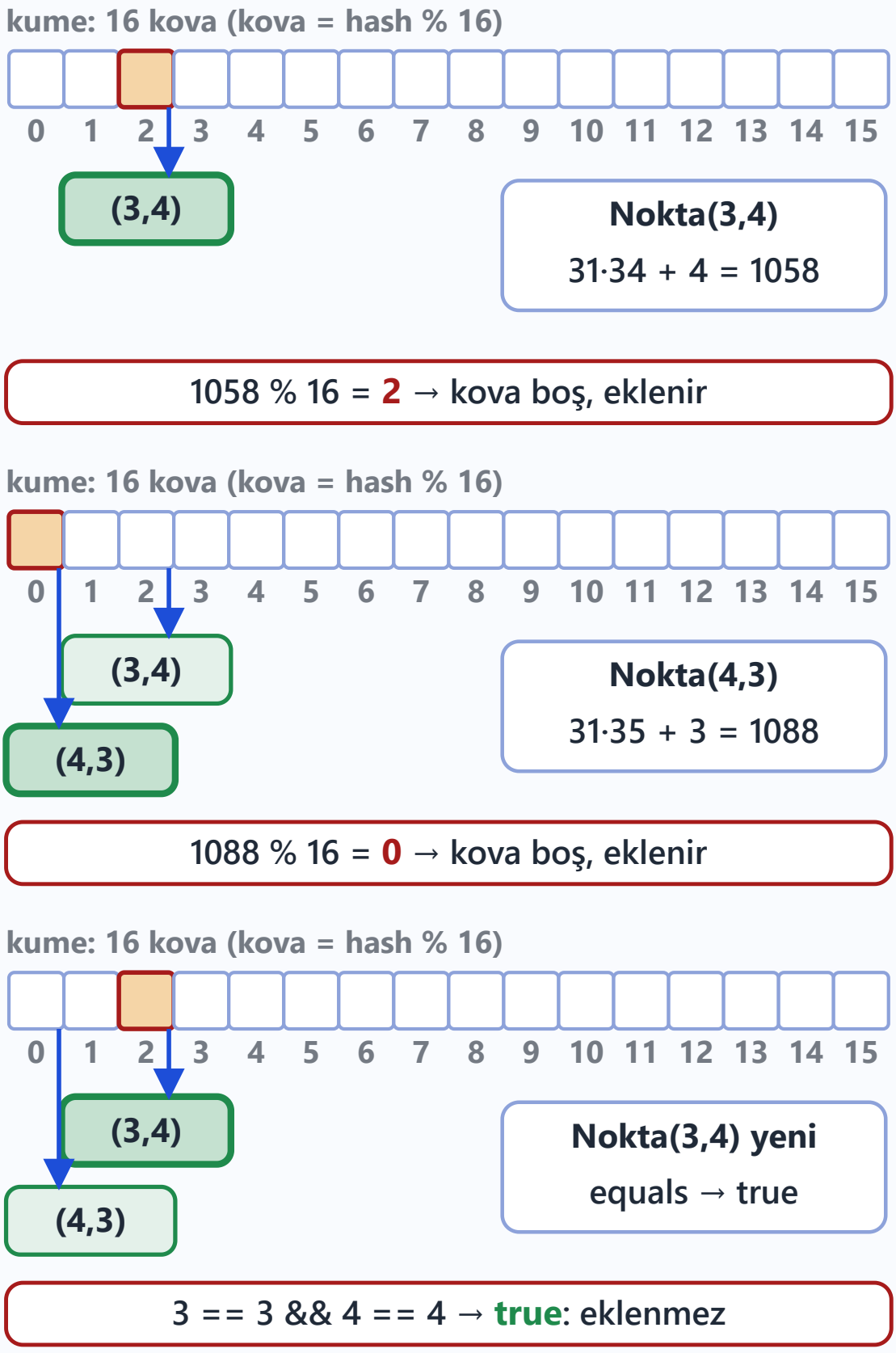
```
2
true
```

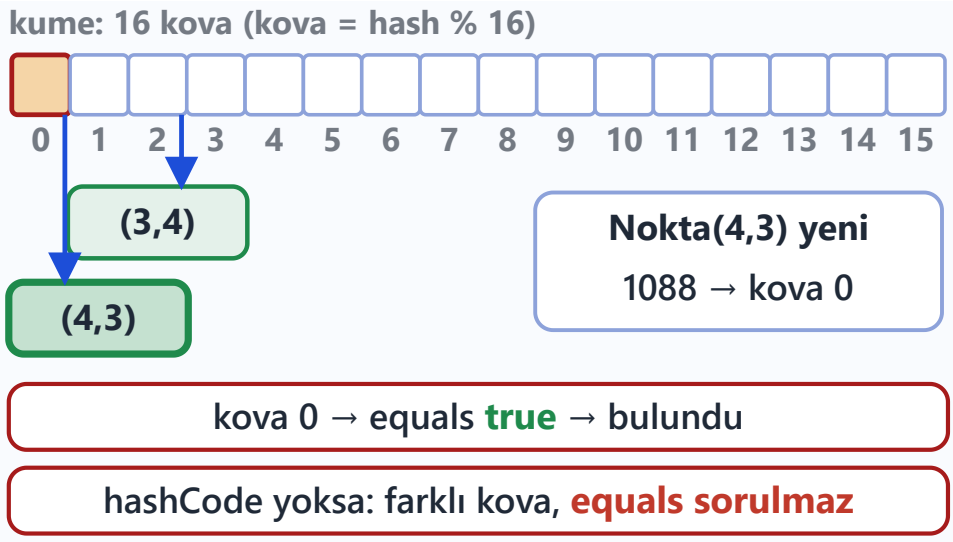
`Objects.hash(x, y)`, iki alan için *31 · (31 + x) + y* değerini hesaplar. Boş bir `HashSet` 16 kovayla başlar. Öğenin kovası, küçük hash değerleri için *hash % 16*'dır. (Kütüphane gerçekte hash değerinin üst bitlerini alt bitlerle karıştırır ve *hash & 15* alır. 65.536'dan küçük pozitif değerlerde sonuç *hash % 16* ile aynıdır.)

İşlem	hash	kova	Kovada ne var?	Sonuç	size
`add(Nokta(3, 4))`	31 · 34 + 4 = 1058	2	boş	eklendi	1
`add(Nokta(4, 3))`	31 · 35 + 3 = 1088	0	boş	eklendi	2

İşlem	hash	kova	Kovada ne var?	Sonuç	size
`add(Nokta(3, 4))` (yeni nesne)	1058	2	(3, 4); `equals` doğru	eklenmedi, `add` `false`	2
`contains(Nokta(4, 3))`	1088	0	(4, 3); `equals` doğru	`true`	2

Tablo 4.3.16: `HashSet` izlemesi. Hash = 31 · (31 + x) + y, kova = hash % 16.


kume: 16 kova (kova = hash % 16)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(3,4)
Nokta(3,4)
31·34 + 4 = 1058
1058 % 16 = 2 → kova boş, eklenir
kume: 16 kova (kova = hash % 16)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(3,4)
(4,3)
Nokta(4,3)
31·35 + 3 = 1088
1088 % 16 = 0 → kova boş, eklenir
kume: 16 kova (kova = hash % 16)
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
(3,4)
(4,3)
Nokta(3,4) yeni
equals → true
3 == 3 && 4 == 4 → true: eklenmez



Şekil 4.3.22: `HashSet` izlemesinden dört an. (a) (3, 4): hash 1058, kova 2. (b) (4, 3): hash 1088, kova 0. (c) Yeni bir (3, 4) yine kova 2'ye düşer; `equals` doğru olduğu için eklenmez. (d) `contains` önce kovayı bulur, sonra `equals` sorar: `true`.

Tablo 4.3.16 iki önemli noktayı gösterir. Birincisi, aynı sayıların farklı sırası farklı hash verir: (3, 4) için 1058, (4, 3) için 1088. Formüldeki 31 çarpanı bu yüzden vardır; düz toplama (x + y) iki noktayı aynı değere götürürdü. İkincisi, küme her işlemde yalnız bir kovaya bakar. 16 kovaya dağılmış öğeler arasında arama, bütün öğeleri tek tek gezmekten çok daha hızlıdır. İyi dağıtan bir `hashCode` ile `add` ve `contains` ortalama O(1) sürer.

Farklı nesneler aynı kovaya düşebilir; buna **çakışma** denir. (1, 2) noktasının hash'i 31 · 32 + 2 = 994'tür ve 994 % 16 = 2'dir. Bu nokta da (3, 4) ile aynı kovaya düşer. Küme bu durumda `equals` sorar, yanlış cevabını alır ve yeni noktayı aynı kovaya ikinci öğe olarak ekler. Hatta hash değerleri bile eşit olabilir: (2, 65) noktasının hash'i de (4, 3)'ünki gibi 1088'dir. Sözleşme buna izin verir; eşit hash, eşit nesne demek değildir. `equals` son sözü söyler.

Örnek 4.3.8

Bir sınıfın `hashCode`'u her nesne için 42 döndürsün; `equals` doğru yazılmış olsun. Bir `HashSet`'e (1, 2), (1, 2) ve (2, 1) eklenince boyut kaç olur? Küme doğru çalışır mı?

Sözleşme bozulmaz: eşit nesnelerin hash'i aynıdır, çünkü bütün nesnelerinki aynıdır. Bütün öğeler aynı kovaya düşer. İkinci (1, 2) `equals` ile yakalanır ve eklenmez; (2, 1) eşit olmadığı için eklenir. Boyut **2**'dir, `contains` doğru cevap verir. Ama her işlem o tek kovadaki bütün öğeleri gezmek zorundadır. Küme doğru çalışır, ama ortalama O(1) yerine O(n) sürer. Kötü bir `hashCode` doğruluğu değil, hızı bozar.

4.3.8 Sık yapılan hatalar

Kalıtımla ilgili hatalar iki gruba ayrılır. Derleme hataları javac'ın iletileriyle hemen görünür. İletileri okumayı öğrenmek bu hataların çoğunu dakikalar içinde çözdürür. Mantık hataları ise derleyiciden geçer ve ancak program yanlış sonuç verince fark edilir. Aşağıdaki iki tablo, notun gövdesinde ayrı ayrı işlenen hataları bir arada verir. İletiler JDK 21'in javac ve java araçlarıyla alınmıştır.

Hata	javac iletisi	Düzeltme
Üst sınıfta parametresiz yapıcı yokken `super(...)` çağırmamak	`constructor Hayvan in class Hayvan cannot be applied to given types`	İlk satıra `super(isim);`
`super(...)` 'ı ilk satıra yazmamak; `this(...)` ile birlikte yazmak	`call to super must be first statement in constructor`	Tek bir `this` / `super` çağrısı, ilk satırda
`private` alana alt sınıftan erişmek	`saglik has private access in Hayvan`	Getter ya da `protected` metot
Başka paketten üst tip referansıyla `protected` üyeye erişmek	`isim has protected access in Hayvan`	Alt sınıfın kendi tipi üzerinden erişim
Geçersiz kılarken erişimi daraltmak	`attempting to assign weaker access privileges; was public`	Aynı ya da daha geniş erişim
Dönüş tipini değiştirmek	`return type String is not compatible with Hayvan`	Aynı tip ya da alt tipi
Kontrol edilen istisna eklemek	`overridden method does not throw Exception`	İstisnayı metot içinde yakalamak
Yanlış imzada `@Override` : `Ciz()` , `equals(Nokta)` , `private` metot	`method does not override or implement a method from a supertype`	Ad ve parametre listesi birebir aynı
`final` metodu geçersiz kılmak	`overridden method is final`	Başka adla yeni metot
`final` sınıftan türetmek	`cannot inherit from final Sabit`	Oluşum
İki sınıftan türetmek	`'{' expected`	Oluşum ya da arayüz
Üst tip değişkeniyle alt sınıf metodu çağırmak	`cannot find symbol … method kislikMi()`	`instanceof` ve aşağı dönüşüm

Tablo 4.3.17: Kalıtımla ilgili sık derleme hataları.

Hata	Sonuç	Düzeltme
Nesneleri `==` ile karşılaştırmak	İçeriği aynı iki nesne için `false`	`equals`
`equals` var, `hashCode` yok	`HashSet` 'e aynı nokta iki kez eklenir: `2` `false`	`Objects.hash(x, y)`
`equals(Nokta)` yazmak (`@Override` 'sız)	`contains` eşit nesneyi bulamaz	`equals(Object)` ve `@Override`
`@Override` 'sız yazım hatası (`Ciz`)	Üst sınıfın sürümü sessizce çalışır	Her geçersiz kılmada `@Override`
`private` metodu geçersiz kıldığını sanmak	Üst sınıfın kodu kendi `private` sürümünü çağırır	`@Override` bu yanılgıyı yakalar
Kontrolsüz aşağı dönüşüm	`ClassCastException`	Önce `instanceof`

Hata	Sonuç	Düzeltme
Üst yapıcıdan geçersiz kılınabilen metot çağırmak	Alt sınıfın alanları henüz varsayılan değerde: `b=0 s=null`	Yapıcıdan yalnız `private` / `final` metot
Kendini çağıran metotları birlikte geçersiz kılmak	Sayaç iki kez artar: `6 3`	Oluşum ve iletme
Kod paylaşmak için kalıtım (`Yigin extends ArrayList`)	İstenmeyen `public` metotlar gelir, ortadan ekleme yapılabilir	Oluşum
Sözü bozan alt sınıf (`Kare extends Dikdortgen`)	Üst sınıf için yazılmış kod yanlış sonuç verir: 16	Ortak üst sınıf ya da değiştirilemez sınıflar

Tablo 4.3.18: Kalıtımla ilgili sık çalışma zamanı ve mantık hataları.

Bu hataların büyük kısmı iki alışkanlıkla önlenir. Birincisi, her geçersiz kılmada `@Override` yazmaktır; ikinci tablodaki ilk beş hatadan üçünü derleme hatasına çevirir. İkincisi, kalıtım kurmadan önce is-a testini sınıfın davranışına uygulamaktır. Test olumsuzsa ya da şüpheliyse oluşum seçilir.

4.3.9 Özet

- Kalıtım, var olan bir sınıfın üyelerini alıp yenilerini ekleyerek yeni bir sınıf oluşturmaktır. `class Alt extends Ust` yazılır; `extends`'ten sonra tek bir sınıf gelir. `extends` yazılmayan sınıf `Object`'ten türer.
- Alt sınıf üst sınıfın `public` ve `protected` üyelerini kendi üyeleri gibi kullanır. `private` alanlar nesnede vardır, ama doğrudan erişilemez. Yapıcılar kalıtımla alınmaz.
- `new` tek bir nesne oluşturur; üst sınıf kısmı bu nesnenin içindedir. `super` ayrı bir nesne değil, aynı nesneye üst sınıfın kodu üzerinden ulaşma yoludur.
- `super(...)` üst sınıf yapıcısını çağırır ve Java 21'de yapıcının ilk deyimidir. Yazılmazsa derleyici `super();` ekler. `super.m()` üst sınıfın sürümünü çağırır ve derleme anında bağlanır.
- Yapıcı çağrıları aşağıdan yukarı gider, gövdeler `Object` → … → `Ust` → `Alt` sırasıyla tamamlanır. Nesne kurmanın sırası şudur: üst sınıf alanları, üst yapıcı, alt sınıf alanları, alt yapıcı. `static` bloklar sınıf ilk kullanılırken bir kez, üstten alta çalışır.
- Geçersiz kılma, alt sınıfta aynı imzalı metodu yeniden yazmaktır. Hangi sürümün çalışacağına nesnenin gerçek sınıfı karar verir. Aşırı yükleme ise farklı parametre listesidir ve derleme anında seçilir.
- Geçersiz kılan metot üst sınıfın sözünü daraltamaz. Dönüş tipi aynı ya da alt tip, erişim aynı ya da geniş, kontrol edilen istisnalar aynı ya da dar olur. `@Override` bu kuralları derleyiciye denetletir.
- `private`, `static` ve `final` metotlar geçersiz kılınmaz. `private` ile aynı adlı metot yeni bir metottur; `static` metot yalnız gizlenir.
- `protected` = sınıf + aynı paket + alt sınıflar. Başka paketteki alt sınıf `protected` üyeye yalnız kendi tipinden bir referansla erişir. Alanlar `private` kalır, alt sınıflara `protected` metotlar açılır.
- `final` metot geçersiz kılınamaz, `final` sınıftan alt sınıf türetilemez (`String`, `Math`, `Integer`).

- Yukarı dönüşüm kendiliğindendir ve nesnenin görünümünü daraltır: çağrılabilecek metotları değişkenin tipi belirler. Aşağı dönüşüm açıkça yazılır ve önce `instanceof` ile denetlenir.
- is-a kalıtımdır, has-a oluşumdur. Java sınıflarda çoklu kalıtıma izin vermez (elmas problemi); yerine oluşum ve arayüzler kullanılır.
- Kalıtım sıkı bağlılık getirir: kırılgan üst sınıf sorunu ve yerine koyma ilkesinin bozulması bunun sonuçlarıdır. Amaç yalnız kod paylaşmaksa oluşum ve iletme seçilir.
- `Object` her sınıfın atasıdır. `toString` nesneyi okunur yazar. `equals` içerik karşılaştırır ve sözleşmeye uyar. `equals` geçersiz kılınıyorsa `hashCode` da kılınır; eşit nesnelerin hash kodu aynıdır.
- `==` referansları, `equals` içerikleri karşılaştırır. `HashSet` önce `hashCode` ile kova seçer, sonra `equals` sorar; iyi bir `hashCode` ile işlemler ortalama O(1) sürer.

4.3.10 Kavram listesi

kalıtım *inheritance* · üst sınıf / alt sınıf *superclass / subclass* · super *super keyword* · yapıcı zinciri *constructor chaining* · varsayılan yapıcı *default constructor* · ilklendirici blok *initializer block* · geçersiz kılma *method overriding* · aşırı yükleme *method overloading* · metot imzası *method signature* · etiket *annotation* · kovaryant dönüş tipi *covariant return type* · erişim belirleyici *access modifier* · yukarı dönüşüm / aşağı dönüşüm *upcasting / downcasting* · dinamik bağlama *dynamic binding* · is-a / has-a ilişkisi *is-a / has-a relationship* · oluşum *composition* · çoklu kalıtım *multiple inheritance* · kırılgan üst sınıf *fragile base class* · iletme *forwarding* · yerine koyma ilkesi *Liskov substitution principle* · Object sınıfı *Object class* · hash kodu *hash code*

4.3.11 Kendini Dene

1. `class X { X() { print("X "); } }`, `class Y extends X { Y() { this(1); print("Y "); } Y(int a) { print("Y" + a + " "); } }` ve `class Z extends Y { Z() { super(5); print("Z "); } }` verilsin. `new Z()` ne yazar?

   Cevap

   `Z()`'nin ilk satırı `super(5)`'tir ve doğrudan `Y(int)`'i çağırır; `Y()` hiç çalışmaz. `Y(int)`'te gizli `super()` vardır: önce `X()` çalışır ve `X` yazar. Sonra `Y(int)`'in gövdesi `Y5`, en son `Z()`'nin gövdesi `Z` yazar. Sonuç: `X Y5 Z`. `Y` yazısı çıkmaz, çünkü `this(1)` içeren yapıcı çağrılmamıştır.

2. `class P { int f() { return 1; } int g() { return f() * 10; } }` ve `class Q extends P { int f() { return 2; } int g() { return super.g() + 1; } }` verilsin. `P p = new Q();` için `p.g()` kaçtır?

   Cevap

   Nesne bir `Q`'dur; `Q.g()` çalışır ve `super.g()` ile `P.g()`'yi çağırır. `P.g()`'nin içindeki `f()`, `this.f()`'tir ve `this` hâlâ bir `Q`'dur: `Q.f()` = 2 döner. `P.g()` = 2 · 10 = 20, `Q.g()` = 20 + 1 = **21**. `super` yalnız `g`'nin hangi sürümünün çağrılacağını belirler; o sürümün içindeki çağrılar yine nesneye göre seçilir. `new P().g()` ise 10 verir.

3. Hangi çiftler is-a, hangileri has-a ilişkisidir? Transistör–Yarı iletken eleman, Devre–Direnç, Telefon–Batarya, Elektrikli süpürge–Ev aleti.

   Cevap

   is-a: Transistör–Yarı iletken eleman ("her transistör bir yarı iletken elemandır") ve Elektrikli süpürge–Ev aleti. **has-a**: Devre–Direnç (bir devrenin dirençleri vardır; dirençler bir liste alanında tutulur) ve Telefon–Batarya (telefonun bir bataryası vardır; batarya değiştirilebilir, bu da oluşumun alanın sonradan değiştirilebilmesi özelliğine uyar).

4. `class U { U() { System.out.println("U(): " + ad()); } String ad() { return "U"; } }` ve `class V extends U { String s = "V"; String ad() { return s; } }` verilsin. `new V()` ne yazar?

   Cevap

   `U(): null`. `V()`'nin gizli `super()`'ı `U()`'yu çalıştırır. `U()` içindeki `ad()` geçersiz kılınmıştır ve nesne bir `V` olduğu için `V.ad()`'a gider. Ama alt sınıfın alan ilk değerleri `super()` döndükten sonra atanır. O anda `s` hâlâ `null`'dır.

5. `KisiI` sınıfının `equals`'ı `getClass()` yerine `instanceof KisiI` kullanıyor ve adı ile yaşı karşılaştırıyor. `OgrenciI extends KisiI` yeni alan eklemiyor ve `equals`'ı geçersiz kılmıyor. `new KisiI("Ali", 20).equals(new OgrenciI("Ali", 20))` ve ters sıradaki çağrı ne verir?

   Cevap

   İkisi de `true`'dur. Bir `OgrenciI` bir `KisiI`'dir; `instanceof` iki yönde de doğru çıkar ve alanlar eşittir. Simetri korunur, çünkü iki çağrı da aynı `equals`'ı çalıştırır. `getClass()` kullanılsaydı iki çağrı da `false` verirdi.

6. `Ust`'te `protected Object kopya()` var. `Alt extends Ust` içinde `@Override public Alt`

`kopya()` derlenir mi? `@Override Alt kopya()` (belirleyicisiz) derlenir mi?

> **Cevap**
>
> Birincisi **derlenir**. `Alt`, `Object`'in alt tipidir (kovaryant dönüş) ve erişim `protected`'tan `public`'e genişler. İkincisi **derlenmez**: belirleyicisiz yazmak paket erişimidir ve `protected`'tan daha dardır. javac `attempting to assign weaker access privileges; was protected` der.

7. `class A { final void f() { … } }` ve `class B extends A { void f(int x) { … } }` derlenir mi?

> **Cevap**
>
> **Derlenir**. `f(int)`'in parametre listesi farklıdır; `A.f()`'yi geçersiz kılmaz, aşırı yükler. `final` yalnız geçersiz kılmayı yasaklar. `B` nesnesinde iki metot da vardır: `b.f()` `A`'nın sürümünü, `b.f(3)` `B`'ninkini çalıştırır.

8. `canli.Hayvan`'da `protected String isim` var; `canli.Kopek extends Hayvan`. Başka paketteki `evcil.Kedi extends Hayvan` sınıfına `String f(Kopek k) { return k.isim; }` yazılıyor. Derlenir mi?

> **Cevap**
>
> **Derlenmez**: `isim has protected access in Hayvan`. Başka paketteki alt sınıf, `protected` üyeye yalnız kendi tipinden (`Kedi` ya da alt tipleri) bir referansla erişebilir. `k` bir `Kopek`'tir; bir kedinin bir köpeğin iç verisine dokunmasına izin verilmez. `Kedi` `canli` paketinde olsaydı derlenirdi.

9. Hash kodu `Objects.hash(x, y)` olan bir `Nokta` sınıfında, boş bir `HashSet`'e önce (3, 4), sonra (1, 2) ekleniyor. İki nokta hangi kovalara düşer? `size()` kaçtır?

> **Cevap**
>
> (3, 4): 31 · 34 + 4 = 1058, 1058 % 16 = 2. (1, 2): 31 · 32 + 2 = 994, 994 % 16 = 2. İki nokta **aynı kovaya** düşer. İkinci eklemede küme kovadaki (3, 4) ile `equals` sorar, `false` alır ve (1, 2)'yi de ekler. İki `add` da `true` döner; `size()` = **2**. Aynı kova eşit nesne demek değildir.

4.3.12 İleri okuma

- Oracle, *The Java Tutorials*, "Inheritance" bölümü: alt sınıf, geçersiz kılma ve gizleme, `super`, `Object` sınıfı ve `final` konularını kısa örneklerle anlatır. Java'nın resmî ve ücretsiz başlangıç kaynağıdır.

- J. Bloch, *Effective Java* (3. baskı), Madde 10–11 ve 18–19: `equals` ile `hashCode`'un nasıl yazılacağını anlatır. Oluşumun neden kalıtıma tercih edildiğini ve kalıtım için tasarlamanın kurallarını da tartışır. Kırılgan üst sınıf örneğinin kaynağı da bu kitaptır.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals* (12. baskı), Bölüm 5 "Inheritance": `equals`'ta `getClass()` ile `instanceof` seçimini ve simetri sorununu ayrıntılı tartışır.
- OpenJDK, JEP 513 "Flexible Constructor Bodies": Java 25'te `super(...)`'dan önce deyim yazmaya izin veren değişikliği ve gerekçesini anlatır. Yapıcı kurallarının neden konduğunu da açıklar.

4.3.13 Kaynaklar

1. Oracle, *The Java Tutorials*: Inheritance; Overriding and Hiding Methods; Using the Keyword super; Object as a Superclass.
2. J. Gosling ve diğerleri, *The Java Language Specification, Java SE 21 Edition*, Oracle, 2023 (8.1.4 Superclasses and Subclasses, 8.4.8 Inheritance, Overriding, and Hiding).
3. P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (Bölüm 9: Object-Oriented Programming: Inheritance).
4. K. Sierra ve B. Bates, *Head First Java*, 3. baskı, O'Reilly, 2022 (Bölüm 7: Inheritance and Polymorphism).
5. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson, 2022 (Bölüm 5: Inheritance).
6. J. Bloch, *Effective Java*, 3. baskı, Addison-Wesley, 2018 (Madde 10–11: equals ve hashCode; Madde 18: oluşumu kalıtıma tercih etmek).
7. OpenJDK, *JEP 513: Flexible Constructor Bodies*, JDK 25, 2025. https://openjdk.org/jeps/513

KISIM II · BÖLÜM 4.4

Çok Biçimlilik

Bir hayvanat bahçesinde bakıcı “herkes ses çıkarsın” der; köpek havlar, kedi miyavlar. Bakıcı her hayvana ayrı komut vermez, her hayvan aynı komuta kendi bildiği gibi karşılık verir. Nesne yönelimli programlamada bu fikrin adı çok biçimliliktir. Kod üst sınıf tipiyle bir kez yazılır; çalışma sırasında her nesne çağrıya kendi sürümüyle cevap verir. Not önce referans tipi ile nesne tipi arasındaki farkı kurar: derleyici birine, Java sanal makinesi (JVM) ötekine bakar. Ardından yukarı ve aşağı dönüşümü, `ClassCastException`'ı ve `instanceof` sınamasını işler. Dinamik bağlama, metot araması ve metot tablosu notun merkezindedir. Alanların, `static` metotların ve aşırı yüklemenin neden derleme zamanında seçildiği ayrıca incelenir. Not şekil hiyerarşisi, çok biçimli parametreler, bir tanksavar oyunu ve `equals` metodunun simetri tuzağıyla biter.

ÖĞRENME ÇIKTILARI

Kavram
Çok biçimliliği tanımlamak; derleme zamanı ve çalışma zamanı çok biçimliliğini ayırt etmek.

Referans kuralı
Üst sınıf referansına alt sınıf nesnesi atanabildiğini, tersinin neden derleme hatası verdiğini açıklamak.

Tip dönüşümü
Yukarı ve aşağı dönüşümü yazmak; `ClassCastException`'ı `instanceof` ile önlemek; dizi kovaryansının doğurduğu `ArrayStoreException`'ı tanımak.

Dinamik bağlama
Çağrılan metodun nesnenin gerçek sınıfına göre seçildiğini adım adım izlemek; metot aramasını hiyerarşide ve metot tablosunda göstermek.

Aşırı yükleme ve geçersiz kılma
Aşırı yükleme ile geçersiz kılmayı karşılaştırmak; hangisini derleyicinin, hangisini JVM'in seçtiğini bilmek; aşırı yükleme seçiminin üç aşamasını uygulamak.

Tasarım
Şekil dizisi ve tanksavar oyunu örnekleriyle üst sınıf tipiyle genel kod yazmak; `equals` metodunun simetri tuzağını tanımak.

4.4.1 Çok biçimlilik kavramı

Nesne yönelimli programlamanın dört temel ilkesi sarmalama, kalıtım, çok biçimlilik ve soyutlamadır. Sarmalama veriyi ve onu işleyen metotları bir sınıfta toplar. Kalıtım, bir sınıfın başka bir sınıfı genişletmesini sağlar. Çok biçimlilik bu ikisinin üzerine kurulur: bir sınıf hiyerarşisi olmadan çalışma zamanı çok biçimliliği de olmaz. Bu yüzden kalıtımın temel kavramları burada da kullanılır: üst sınıf, alt sınıf, `extends`, geçersiz kılma ve `@Override`.

4.4.1.1 Çok biçimlilik nedir?

> **Tanım 4.4.1**
>
> **Çok biçimlilik** (*polymorphism*), aynı ifadenin farklı nesnelerde farklı davranışa yol açmasıdır. Sözcük Yunanca *poly* (çok) ve *morph* (biçim) sözcüklerinden gelir. Java'da bir referans, kendi tipindeki ya da bir alt sınıfa ait bir nesneyi gösterebilir. Bu referans üzerinden çağrılan geçersiz kılınmış metodun hangi sürümünün çalışacağını, gösterilen nesnenin gerçek sınıfı belirler.

Tanımın özü tek bir satırda görülür. `hayvan.sesCikar()` çağrısı `hayvan` bir köpeği gösteriyorsa `Hav hav!`, bir kediyi gösteriyorsa `Miyav!` yazdırır. Satır aynıdır, davranış nesneye göre değişir. Kod üst sınıf tipiyle bir kez yazılır; her nesne çağrıya kendi sürümüyle cevap verir.

Gündelik hayatta bu fikrin pek çok örneği vardır. Uzaktan kumandadaki aynı “aç” düğmesi televizyonu, klimayı ve projektörü açar; her cihaz bu işi kendi bildiği gibi yapar. B sınıfı sürücü belgesiyle her marka otomobil sürülür. Sürücü yalnız “otomobil” arayüzünü kullanır: direksiyon, pedallar, vites. Motorun ayrıntısı ise markaya aittir. Hayvanat bahçesindeki bakıcı da tek komut verir ve her hayvan kendi sesini çıkarır. Üç örnekte de çağıran taraf ortak bir arayüzü bilir; ayrıntıyı nesne kendisi tamamlar.

```
Hayvan h1 = new Kopek();
Hayvan h2 = new Kedi();

h1.sesCikar(); // Hav hav!
h2.sesCikar(); // Miyav!
```

Bu dört satırda iki değişken de `Hayvan` tipindedir. `h1` bir `Kopek`, `h2` bir `Kedi` nesnesini gösterir. Derleyici iki çağrıyı da aynı biçimde, `Hayvan` sınıfının `sesCikar()` metoduna yapılmış çağrı olarak derler. Program çalışırken ise JVM her çağrıda nesneye bakar ve köpek için `Kopek`'in, kedi için `Kedi`'nin sürümünü çalıştırır. Çıktı kodun yorumlarında yazıldığı gibidir: önce `Hav hav!`, sonra `Miyav!`.

4.4.1.2 Çok biçimliliğin iki türü

Java'da "aynı ad, farklı davranış" iki farklı mekanizmayla elde edilir. Birincisi aşırı yüklemedir: aynı sınıfta, aynı adlı ama parametreleri farklı birden çok metot yazılır. Hangisinin çağrılacağına derleyici karar verir. İkincisi geçersiz kılmadır: alt sınıf, üst sınıftan aldığı metodu aynı imzayla yeniden yazar. Hangi sürümün çalışacağına program çalışırken JVM karar verir.

Tanım 4.4.2

Derleme zamanı çok biçimliliği (statik çok biçimlilik), aşırı yüklenmiş metotlar arasındaki seçimi derleyicinin, argümanların sayısına ve bildirilen tiplerine bakarak yapmasıdır. **Çalışma zamanı çok biçimliliği** (dinamik çok biçimlilik) ise geçersiz kılınmış metotlar arasındaki seçimin program çalışırken, nesnenin gerçek sınıfına bakılarak yapılmasıdır. Bu ikinci seçime **dinamik bağlama** denir.

İki tür aşağıdaki iki kısa örnekte yan yana görülür. Soldaki örnekte tek bir sınıfta iki `topla` metodu vardır; argümanlar tam sayıysa `int` sürümü, ondalıklıysa `double` sürümü seçilir. Bu seçim program derlenirken yapılır ve bayt koduna yazılır. Sağdaki örnekte ise seçim nesneye bağlıdır ve ancak program çalışırken belli olur.

```
int topla(int a, int b) { ... }
double topla(double a, double b) { ... }

topla(5, 3);     // int sürümü
topla(3.5, 2.7); // double sürümü
```

	Derleme zamanı (statik)	Çalışma zamanı (dinamik)
Mekanizma	Aşırı yükleme	Geçersiz kılma + dinamik bağlama
Seçimi yapan	Derleyici (`javac`)	JVM, program çalışırken
Neye bakılır?	Argümanların sayısı ve bildirilen tipleri	Nesnenin gerçek sınıfı
Kalıtım gerekir mi?	Hayır, tek sınıfta olur	Evet, sınıf hiyerarşisi gerekir
Kısa tarif	Aynı ad, farklı parametreler	Aynı imza, farklı sınıflar

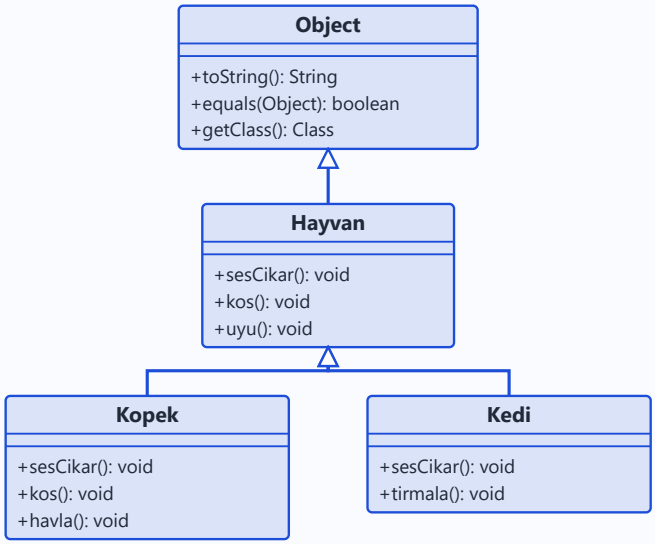
Tablo 4.4.1: Çok biçimliliğin iki türünün karşılaştırması.

Günlük konuşmada "çok biçimlilik" denince çoğunlukla çalışma zamanı çok biçimliliği kastedilir. Notun büyük kısmı da onu anlatır. Aşırı yükleme de ayrıca incelenir, çünkü iki mekanizma aynı çağrıda birlikte çalışabilir ve en çok karışıklık bu durumda çıkar.

4.4.1.3 Örnek sınıf hiyerarşisi

Notun büyük kısmında aynı küçük hiyerarşi kullanılır. `Hayvan` sınıfının üç metodu vardır: `sesCikar()`, `kos()` ve `uyu()`. `Kopek` ve `Kedi` bu sınıftan türer. Her sınıf gibi `Hayvan` da Java'daki bütün sınıfların atası olan `Object`'ten türer. Şekil 4.4.1 bu yapıyı UML sınıf diyagramıyla gösterir.

```
class Hayvan {
  public void sesCikar() { System.out.println("Hayvan sesi"); }
  public void kos()      { System.out.println("Hayvan koşuyor"); }
  public void uyu()      { System.out.println("Hayvan uyuyor"); }
}
class Kopek extends Hayvan {
  @Override public void sesCikar() { System.out.println("Hav hav!"); }
  @Override public void kos() { System.out.println("Köpek koşuyor"); }
  public void havla() { System.out.println("Köpek havlıyor"); }
}
class Kedi extends Hayvan {
  @Override public void sesCikar() { System.out.println("Miyav!"); }
  public void tirmala() { System.out.println("Kedi tırmalıyor"); }
}
```



Şekil 4.4.1: Örnek sınıf hiyerarşisi. `Kopek` ve `Kedi`, `Hayvan`'dan; `Hayvan` da `Object`'ten türer. Oklar alt sınıftan üst sınıfa yönelir.

İki alt sınıfın üst sınıfla ilişkisi farklıdır. `Kopek`, `sesCikar()` ve `kos()` metotlarını geçersiz kılar ve kendine özgü `havla()` metodunu ekler. `Kedi` yalnız `sesCikar()`'ı geçersiz kılar ve `tirmala()`'yı ekler. `uyu()`'yu iki alt sınıf da geçersiz kılmaz; ikisi de onu `Hayvan`'dan olduğu gibi alır. `kos()` ise yalnız köpekte değişmiştir; kedi `Hayvan`'ın sürümünü kullanır.

Bu ayrıntıların her biri bir kurala karşılık gelir. `havla()` yalnız `Kopek`'te bulunduğu için `Hayvan` tipindeki bir referansla çağrılamaz. `uyu()` alt sınıflarda yazılmadığı için metot araması `Hayvan`'a kadar çıkar. `@Override` etiketi (*annotation*) derleyiciye "bu metot üst sınıftaki bir metodu geçersiz kılıyor" bilgisini verir. Ad ya da parametre yanlış yazılırsa derleyici hata verir.

Kalıtımdaki is-a (bir türüdür) ilişkisi buradaki bütün kuralların temelidir. Köpek de kedi de bir hayvandır. Bu yüzden ikisi de `Hayvan` tipindeki bir değişkende tutulabilir. Tersi doğru değildir: her hayvan bir köpek değildir.

4.4.2 Referanslar ve tip uyumu

Çok biçimliliği anlamak için önce Java'daki değişkenlerin nesneyi değil, nesneye bir referansı tuttuğunu hatırlamak gerekir. Nesne heap'te yaşar. Değişken ise yığındaki bir çerçevede durur ve heap'teki nesnenin yerini gösterir. Bir nesneyi birden çok referans gösterebilir; bu referansların tipleri de farklı olabilir.

4.4.2.1 Bir nesne, üç referans

Aşağıdaki kod tek bir `Kopek` nesnesi oluşturur ve onu üç farklı tipte referansla gösterir.

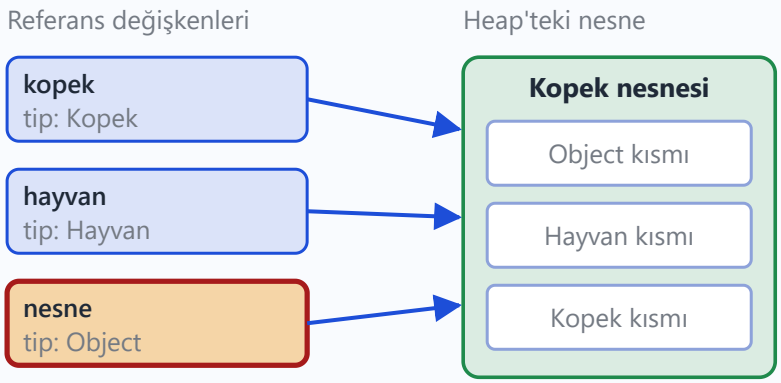
```
// Köpek nesnesine referans
Kopek kopek = new Kopek();
// Hayvan referansı Kopek'i gösterebilir
Hayvan hayvan = kopek;
// Object referansı da gösterebilir
Object nesne = hayvan;
System.out.println(kopek == hayvan);
System.out.println(hayvan == nesne);
System.out.println(nesne.getClass());
```

Program şunları yazar:

```
true
true
class Kopek
```

Kodu satır satır izleyelim. İlk satırdaki `new Kopek()`, heap'te bir nesne oluşturur. `kopek` referansı bu nesneyi gösterir; referansın tipi `Kopek`'tir. İkinci atama yeni bir nesne oluşturmaz. `hayvan`, aynı nesneyi gösteren ikinci bir referanstır. Atama geçerlidir, çünkü köpek bir hayvandır. Üçüncü atamada `Object` tipindeki `nesne` de aynı nesneyi gösterir. Her sınıf `Object`'ten türediği için bu da geçerlidir.

Sonraki iki satır `==` işlecini kullanır. Referans tiplerinde `==` nesnelerin içeriğini değil, referansların aynı nesneyi gösterip göstermediğini karşılaştırır. Üç referans da aynı nesneyi gösterdiği için iki karşılaştırma da `true` verir. Son satırdaki `getClass()`, nesnenin çalışma zamanındaki sınıfını döndürür. Referansın tipi `Object` olsa da nesnenin sınıfı değişmemiştir: `class Kopek`.



Şekil 4.4.2: Üç referans, tek nesne. `kopek`, `hayvan` ve `nesne` heap'teki aynı `Kopek` nesnesini gösterir. Nesnenin içinde `Object`'ten, `Hayvan`'dan ve `Kopek`'ten gelen kısımlar vardır.

Şekil 4.4.2 nesnenin iç yapısını da gösterir. Bir `Kopek` nesnesi, kalıtım zincirindeki her sınıftan gelen kısımları taşır: `Object` kısmı, `Hayvan` kısmı ve `Kopek` kısmı. Bu yüzden nesne, zincirdeki her tipin referansıyla gösterilebilir. Her referans nesnenin tamamını gösterir, ama nesneye kendi tipinin penceresinden bakar.

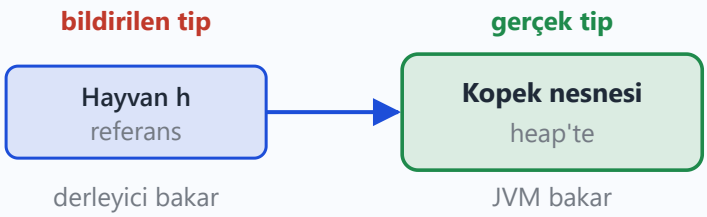
4.4.2.2 Referans tipi ve nesne tipi

Önceki örnek, her referans değişkeninin iki ayrı tipi olduğunu gösterir. Biri değişkenin tanımında yazan tiptir. Öteki, değişkenin o anda gösterdiği nesnenin sınıfıdır.

Tanım 4.4.3

Bir referans değişkeninin **bildirilen tipi** (*declared type*), tanımında yazan tiptir ve değişkenin ömrü boyunca değişmez. **Gerçek tipi** (*actual type*) ise değişkenin o anda gösterdiği nesnenin sınıfıdır ve atamayla değişebilir. Bildirilen tipe kısaca referans tipi, gerçek tipe nesne tipi de denir.

```
Hayvan h = new Kopek();
// bildirilen tür: Hayvan
// gerçek tür:     Kopek
```



Şekil 4.4.3: `Hayvan h = new Kopek();` satırının iki tipi. Bildirilen tip `Hayvan`'dır ve ona derleyici bakar. Gerçek tip `Kopek`'tir ve ona JVM bakar.

İki tipin işlevi farklıdır ve bu ayrım notun geri kalanının anahtarıdır. Derleyici program çalışmadan önce kodu denetler. O anda hiçbir nesne yoktur; derleyici yalnız değişkenlerin tanımlarını bilir. Bu yüzden derleyicinin bütün kararları bildirilen tipe dayanır. JVM ise program çalışırken gerçek nesneyle karşılaşır ve nesnenin hangi sınıftan olduğunu bilir. Tablo 4.4.2 üç temel soruyu ve cevabı veren tarafı özetler.

Soru	Kim karar verir?
Bu metot çağrılabilir mi?	Derleyici, referans tipine bakar.
Hangi sürüm çalışır?	JVM, nesne tipine bakar.
Hangi atamalar geçerli?	Derleyici, referans tipine bakar.

Tablo 4.4.2: Referans tipi ve nesne tipi: hangi soruya kim karar verir?

İlk sorunun sonucu aşağıdaki kodda görülür. Nesne gerçekten bir köpektir, ama değişkenin tipi `Hayvan`'dır.

```
Hayvan hayvan = new Kopek();
hayvan.havla(); // derleme hatası
```

```
error: cannot find symbol
  symbol:   method havla()
  location: variable hayvan of type Hayvan
```

Derleyici `hayvan.havla()` satırına geldiğinde yalnız şunu bilir: `hayvan`, `Hayvan` tipinde bir değişkendir. `Hayvan` sınıfında ve onun üst sınıfı `Object`'te `havla()` adlı bir metot yoktur. Derleyici, değişkenin hangi nesneyi gösterdiğini bilemez. Program çalışırken o satıra gelindiğinde değişken bir kediyi de gösteriyor olabilir. Bu yüzden çağrıya izin vermez. İletideki `location: variable hayvan of type Hayvan` satırı da kararın değişkenin tipine göre verildiğini açıkça söyler.

Uzaktan kumanda benzetmesi bu durumu iyi anlatır. Referans tipi bir kumandadır; kumandada hangi düğmeler varsa yalnız onlar kullanılabilir. Nesne bir köpek olsa da `Hayvan` kumandasında `havla()` düğmesi yoktur. Düğmeye basıldığında cihazın ne yapacağını ise cihaz, yani nesne belirler. `sesCikar()` düğmesi `Hayvan` kumandasında vardır; nesne köpekse basılınca havlama, kediyse miyavlama duyulur.

Sık yapılan hata

"Nesne köpek olduğuna göre `havla()` çağrılabilir" diye düşünmek. Çağrının derlenip derlenmeyeceğine nesne değil, referansın bildirilen tipi karar verir. `havla()` çağrılmak isteniyorsa referans ya baştan `Kopek` tipinde tanımlanır ya da aşağı dönüşümle `Kopek` tipine çevrilir.

4.4.2.3 Referans kuralı

Atamaların geçerliliği de bildirilen tiplere göre denetlenir. Java'nın bu konudaki kuralı kısadır ve kalıtımdaki is-a ilişkisinden doğar.

Tanım 4.4.4

Referans kuralı: Alt sınıfın nesnesi, üst sınıf tipindeki bir referansa atanabilir. Üst sınıfın nesnesi ise alt sınıf tipindeki bir referansa atanamaz. Kısaca, alt sınıf referansından üst sınıf referansına atama otomatiktir; tersi açık dönüşüm ister.

```
// Kopek, Hayvan'ın alt türüdür
Hayvan hayvan = new Kopek();
Object nesne = new Kedi();
```

Bu iki atama geçerlidir. Her köpek bir hayvandır: bir hayvandan beklenen her şeyi, yani `sesCikar()`, `kos()` ve `uyu()`'yu köpek de yapabilir. Her kedi de bir `Object`'tir. Üst tipteki referans, alt tipteki nesnenin yapamayacağı bir şey isteyemez. Bu yüzden atama güvenlidir.

```
// Hayvan, Kopek'in üst sınıfıdır
Kopek kopek = new Hayvan();
```

```
error: incompatible types:
  Hayvan cannot be converted to Kopek
```

Bu atama ise derlenmez. `kopek` değişkeni üzerinden ileride `kopek.havla()` çağrılabilir. Düz bir `Hayvan` nesnesi ise `havla()` bilmez. Derleyici böyle bir atamaya izin verseydi, program çalışırken var olmayan bir metot çağrılmaya çalışılırdı. Java bu tür hataları baştan, derleme zamanında engeller.



Şekil 4.4.4: Referans kuralı. `Hayvan` referansı bir `Kopek` nesnesini gösterebilir; `Kopek` referansı bir `Hayvan` nesnesini gösteremez.

Kuralın en çok şaşırtan sonucu şu iki satırda görülür:

```
Hayvan h = new Kopek();
Kopek k = h;   // derleme hatası
```

```
error: incompatible types: Hayvan cannot be converted to Kopek
```

Burada `h`'nin gösterdiği nesne gerçekten bir köpektir. Yine de ikinci satır derlenmez ve aynı ileti çıkar. Derleyici `h`'nin gösterdiği nesneyi değil, yalnız bildirilen tipini (`Hayvan`) görür. Bir önceki satırı okuyup "demek ki bu bir köpek" çıkarımını yapmaz; yapması da her durumda mümkün değildir. Çözüm açık dönüşümdür: `Kopek k = (Kopek) h;`. Bu yazımla programcı derleyiciye "bu referansın bir köpeği gösterdiğinden eminim" der. Dönüşümün doğru olup olmadığı ise çalışma zamanında denetlenir.

Örnek 4.4.1

`Kopek` ve `Kedi`, `Hayvan`'dan türüyor. Aşağıdaki dört atamadan hangileri derlenir?

(a) `Hayvan h = new Kedi();` — Derlenir. Kedi bir hayvandır; alt sınıf nesnesi üst sınıf referansına atanır.

(b) `Object o = new Kopek();` — Derlenir. Her sınıf `Object`'in alt sınıfıdır.

(c) `Kedi c = new Kopek();` — Derlenmez: `incompatible types: Kopek cannot be converted to Kedi`. `Kopek` ile `Kedi` kardeş sınıflardır; aralarında is-a ilişkisi yoktur.

(d) `Kopek k = new Hayvan();` — Derlenmez: `incompatible types: Hayvan cannot be converted to Kopek`. Üst sınıf nesnesi alt sınıf referansına atanamaz.

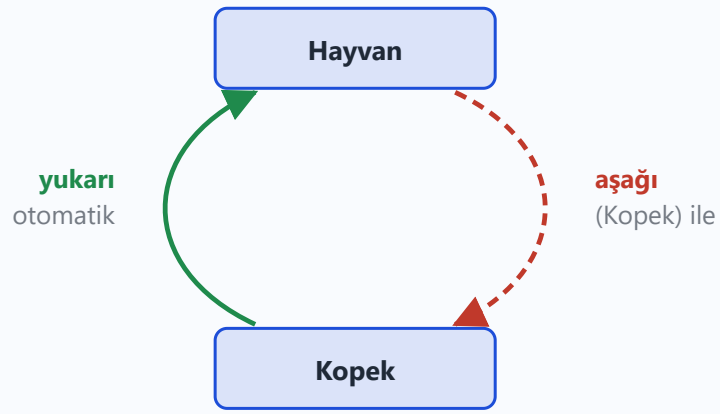
4.4.3 Tip dönüşümü

İlkel tiplerde dönüşüm tanıdık bir işlemdir. `int`'ten `double`'a geçiş kendiliğinden olur; `(int) 3.7` ise açık dönüşümdür ve değeri değiştirir. Referans tiplerinde de benzer iki yön vardır, ama önemli bir fark taşır: referans dönüşümü nesneyi hiçbir zaman değiştirmez. Değişen yalnız nesneye bakan referansın tipidir.

4.4.3.1 Yukarı ve aşağı dönüşüm

Tanım 4.4.5

Yukarı dönüşüm (*upcasting*), alt sınıf tipindeki bir referansın üst sınıf tipine çevrilmesidir. Otomatiktir ve her zaman güvenlidir. **Aşağı dönüşüm** (*downcasting*), üst sınıf tipindeki bir referansın alt sınıf tipine çevrilmesidir. Parantez içinde hedef tip yazılarak açıkça yapılır: `(Kopek) h`. Başarısı, referansın gösterdiği nesneye bağlıdır.



Şekil 4.4.5: Yukarı dönüşüm `Kopek`'ten `Hayvan`'a kendiliğinden olur. Aşağı dönüşüm `Hayvan`'dan `Kopek`'e açık dönüşümle, `(Kopek)` yazılarak yapılır.

```
Kopek kopek = new Kopek();
Hayvan h = kopek;    // yukarı: otomatik
Hayvan h1 = (Hayvan) kopek; // gereksiz
```

İkinci satırda yukarı dönüşüm kendiliğinden yapılır; referans kuralının ta kendisidir. Üçüncü satırdaki `(Hayvan)` yazımı da geçerlidir, ama gereksizdir. Derleyici bu dönüşümü zaten kendisi yapar.

```
Hayvan h = new Kopek();
Kopek k = (Kopek) h; // aşağı: açık
k.havla();           // Köpek havlıyor
```

Burada ters yöne gidilir. `h`'nin bildirilen tipi `Hayvan`'dır; `(Kopek) h` ifadesinin tipi ise `Kopek`'tir. Bu ifade `Kopek` tipindeki `k`'ye atanabilir ve `k.havla()` derlenir. Program çalışırken JVM, `h`'nin gösterdiği nesnenin gerçekten bir `Kopek` olup olmadığını denetler. Nesne bir köpek olduğu için dönüşüm başarılı olur ve `Köpek havlıyor` yazılır.

	Yukarı dönüşüm	**Aşağı dönüşüm**
Yön	Alt → üst	Üst → alt
Yazım	Otomatik (örtük)	Açık: `(Kopek) h`
Güvenli mi?	Her zaman	Nesneye bağlı
Diğer adı	Genişletme dönüşümü (*widening*)	Daraltma dönüşümü (*narrowing*)

Tablo 4.4.3: Yukarı ve aşağı dönüşümün karşılaştırması.

Tablodaki “genişletme” ve “daraltma” adları ilkel tiplerden tanıdıktır. Java dil belirtimi referans dönüşümlerini de bu adlarla anar. Yukarı dönüşüm, referansın görebildiği metot kümesini daraltsa da “genişletme” sayılır. Ad, tipin kapsadığı nesne kümesinin genişlemesinden gelir: `Hayvan` tipi bütün köpekleri, kedileri ve öteki hayvanları kapsar.

> **Sık yapılan hata**
>
> “Dönüşüm nesneyi değiştirir” sanmak. `(Hayvan) kopek` yazmak köpeği düz bir hayvana çevirmez. `((Hayvan) kopek).sesCikar()` çağrısı yine `Hav hav!` yazar, çünkü çalışacak sürümü nesne belirler. Dönüşüm yalnız derleyicinin o ifadeyi hangi tipte gördüğünü değiştirir.

4.4.3.2 Aşağı dönüşüm ve öncelik tuzağı

Aşağı dönüşümün amacı, üst sınıfta bulunmayan, yalnız alt sınıfa ait metotları çağırmaktır. Üst sınıf referansı ancak bu dönüşümle bir alt sınıf değişkenine atanabilir. Dönüşüm iki biçimde yazılır: sonucu bir değişkene atanır ya da aynı satırda doğrudan kullanılır.

```
Hayvan h2 = new Kopek();
Kopek k = (Kopek) h2;
k.havla();
((Kopek) h2).havla(); // tek satırda
```

```
Köpek havlıyor
Köpek havlıyor
```

Program iki kez `Köpek havlıyor` yazar. İlk yazımda dönüşümün sonucu `k`'ye atanır ve `k.havla()` çağrılır. İkinci yazımda `((Kopek) h2)` ifadesi bir `Kopek` referansı üretir ve `.havla()` doğrudan bu ifadeye uygulanır. Dıştaki parantez çifti zorunludur. Nedeni, Java'daki işleç önceliğidir.

Tanım 4.4.6

Öncelik kuralı: Üye erişimi ve metot çağrısı (`.`), dönüşüm işlecinden önce uygulanır. `(Kopek) h2.havla()` ifadesi `(Kopek) (h2.havla())` olarak okunur: dönüşüm, metot çağrısının sonucuna uygulanmak istenir.

```
(Kopek) h2.havla();   // hatalı
```

```
error: not a statement
```

Derleyici bu satırı ayrıştırırken önce `h2.havla()` çağrısını, sonra onun sonucunun `Kopek`'e dönüştürülmesini görür. Böyle bir dönüşüm ifadesi tek başına bir deyim olamaz; derleyici `not a statement` der. Satır başka bir biçimde düzeltilse bile ikinci bir sorun kalır: `h2`'nin bildirilen tipi `Hayvan`'dır ve `Hayvan`'da `havla()` yoktur.

İfade bir atamanın sağında kullanıldığında ikinci sorun doğrudan görünür:

```
String s = (Kopek) h2.isim();
// h2.isim() önce aranır → Hayvan'da yok
```

```
error: cannot find symbol
  symbol:   method isim()
  location: variable h2 of type Hayvan
```

Burada da önce `h2.isim()` çözülmeye çalışılır. Derleyici `Hayvan` sınıfında `isim()` bulamaz ve hata verir. JDK 21'in javac'ı bu satır için ayrıca ikinci bir ileti de verir: `incompatible types: Kopek cannot be converted to String`. Dönüşümün sonucu bir `Kopek`'tir ve `String` değişkenine atanamaz. Doğrusu her durumda aynıdır: dönüşüm parantez içine alınır, nokta parantezin dışına yazılır: `((Kopek) h2).havla();`.

4.4.3.3 `ClassCastException`

Aşağı dönüşüm, derleyiciye verilmiş bir sözdür. Derleyici bu sözün doğru olup olmadığını bilemez ve kabul eder. Söz tutulmazsa hata program çalışırken ortaya çıkar.

```
Hayvan h4 = new Kedi();
Kopek k4 = (Kopek) h4; // derlenir!
k4.havla();
```

```
Exception in thread "main" java.lang.ClassCastException:
class Kedi cannot be cast to class Kopek
```

İkinci satır derlenir. Derleyicinin bakış açısından bunda bir sorun yoktur: `Hayvan` tipindeki bir referans pekâlâ bir `Kopek`'i gösteriyor olabilir. Program çalışırken ise JVM dönüşüm anında nesnenin gerçek sınıfını denetler. Nesne bir `Kedi`'dir ve kedi bir köpek değildir. JVM bu durumda `ClassCastException` fırlatır; yakalayan olmadığı için program durur. Üçüncü satır hiç çalışmaz.

Tanım 4.4.7

`ClassCastException`, bir referansın, gösterdiği nesnenin sınıfıyla uyumsuz bir tipe açıkça dönüştürülmek istendiğinde JVM'in fırlattığı çalışma zamanı istisnasıdır. `(T) r` dönüşümü, ancak `r`'nin gösterdiği nesne `T`'den ya da `T`'nin bir alt sınıfından ise başarılı olur. `r` `null` ise dönüşüm her zaman başarılıdır ve sonuç yine `null`'dır.

JDK 21'de iletinin tamamı biraz daha uzundur: `class Kedi cannot be cast to class Kopek (Kedi and Kopek are in unnamed module of loader 'app')`. Ayraç içindeki kısım sınıfların hangi modülden ve hangi sınıf yükleyiciden geldiğini söyler; bu derste önemli değildir. İletinin baş kısmı ise her şeyi anlatır: hangi sınıftan hangi sınıfa dönüşüm istendiği.

Aynı istisna, düz bir üst sınıf nesnesi alt sınıfa dönüştürülmek istendiğinde de çıkar:

```
Kopek k5 = (Kopek) new Hayvan(); // derlenir!
```

```
java.lang.ClassCastException:
class Hayvan cannot be cast to class Kopek
```

Bu satır da derlenir, çünkü ifadenin tipi `Hayvan`'dır ve `Hayvan`'dan `Kopek`'e dönüşüm derleyicinin gözünde mümkündür. Ama oluşturulan nesne bir köpek değildir ve hiçbir zaman köpek olmayacaktır. Buradan çıkan genel sonuç şudur: üst sınıfın kendi nesnesi alt sınıf referansına hiçbir zaman atanamaz. Dönüşüm, yalnız nesne gerçekten o alt sınıftansa başarılı olur.

Derleyicinin hiç kabul etmediği bir dönüşüm türü de vardır. Aralarında kalıtım ilişkisi olmayan iki tip arasında dönüşüm yazılırsa derleyici bunun hiçbir nesne için doğru olamayacağını bilir:

```
Kopek kopek = new Kopek();
Kedi c = (Kedi) kopek;   // derleme hatası
```

```
error: incompatible types: Kopek cannot be converted to Kedi
```

`Kopek` ile `Kedi` kardeş sınıflardır. Bir nesne aynı anda hem köpek hem kedi olamaz; Java'da bir sınıfın yalnız bir üst sınıfı vardır. Bu yüzden derleyici dönüşümü derleme zamanında reddeder. Sınıflar için kural kısaca şöyledir: dönüşüm, kaynak ve hedef tipten biri ötekinin alt tipi değilse derlenmez; alt tipiyse derlenir ve çalışma zamanında denetlenir.

Örnek 4.4.2

`Hayvan h4 = new Kedi();` verilsin. Aşağıdaki üç dönüşüm için derleyicinin ve JVM'in kararı nedir?

(a) `(Kedi) h4` : `Kedi` , `Hayvan` 'ın alt tipidir; derlenir. Nesne bir kedidir; JVM onaylar. Başarılı.

(b) `(Kopek) h4` : `Kopek` , `Hayvan` 'ın alt tipidir; derlenir. Nesne bir kedidir; JVM `ClassCastException` fırlatır.

(c) `(Object) h4` : yukarı dönüşümdür, derlenir ve her zaman başarılıdır. Nesne değişmez; referansın tipi `Object` olur.

4.4.3.4 Kod izleme: hatalı aşağı dönüşüm

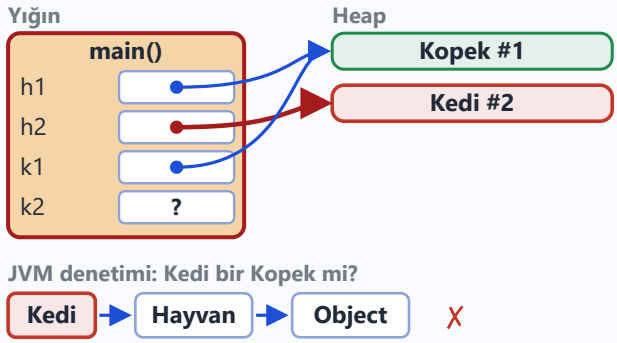
Aşağıdaki program bir başarılı ve bir başarısız aşağı dönüşüm içerir. Satırlar 1'den 7'ye numaralanırsa izleme daha kolay okunur.

```
Hayvan h1 = new Kopek();
Hayvan h2 = new Kedi();
Kopek k1 = (Kopek) h1;
k1.havla();
Kopek k2 = (Kopek) h2;
k2.havla();
System.out.println("Bitti");
```

Satır	Derleyici	JVM (çalışırken)	Durum / çıktı
1	Yukarı dönüşüm, geçerli	`Kopek` nesnesi #1 oluşur	`h1` → #1

Satır	Derleyici	JVM (çalışırken)	Durum / çıktı
2	Yukarı dönüşüm, geçerli	`Kedi` nesnesi #2 oluşur	`h2` → #2
3	`Hayvan` → `Kopek` mümkün, derlenir	#1 bir `Kopek` mi? Evet	`k1` → #1
4	`Kopek` 'te `havla()` var	`Kopek.havla()`	`Köpek havlıyor`
5	`Hayvan` → `Kopek` mümkün, derlenir	#2 bir `Kopek` mi? `Kedi` → `Hayvan` → `Object` : hayır	`ClassCastException` , `k2` 'ye atama yapılmaz
6	Derlenir	Çalışmaz	—
7	Derlenir	Çalışmaz	`Bitti` yazılmaz

Tablo 4.4.4: Hatalı aşağı dönüşüm programının izlemesi.



Şekil 4.4.6: 5. satırda JVM'in denetimi. `h2` 'nin gösterdiği nesnenin sınıfından yukarı çıkan zincirde (`Kedi` → `Hayvan` → `Object`) `Kopek` yoktur; dönüşüm başarısız olur ve `k2` boş kalır.

Tablonun en önemli satırı 5. satırdır. Derleyici bu satırı 3. satırdan ayırt edemez; ikisinde de bildirilen tip `Hayvan` , hedef tip `Kopek` 'tir. Fark yalnız çalışma zamanında, nesneye bakılınca ortaya çıkar. JVM'in denetimi Şekil 4.4.6'da görülen zincir yürüyüşüdür: nesnenin sınıfından başlanır ve üst sınıflarda sırayla `Kopek` aranarak gezinilir. `Object` 'e kadar `Kopek` bulunmazsa dönüşüm reddedilir.

Programın çıktısı yalnız `Köpek havlıyor` satırı ve ardından istisna iletisidir. `Bitti` yazılmaz, çünkü istisna yakalanmadığında `main` metodu oracıkta sona erer.

4.4.3.5 `instanceof` ile güvenli dönüşüm

Bir referansın hangi sınıftan bir nesneyi gösterdiği önceden bilinmiyorsa aşağı dönüşümden önce sınama yapılır. Java bunun için `instanceof` işlecini sunar.

Tanım 4.4.8

`referans instanceof Tur` ifadesi, referansın gösterdiği nesne `Tur` 'den ya da onun bir alt sınıfından ise `true` , değilse `false` verir. `null instanceof Tur` her zaman `false` 'tur. Referansın bildirilen tipiyle `Tur` arasında hiçbir kalıtım ilişkisi yoksa ifade derlenmez.

Aşağıdaki döngü, `hayvanlar = {new Kopek(), new Kedi(), new Hayvan()}` dizisindeki her öğe için önce tipi sınar, sonra dönüştürür.

```
for (Hayvan x : hayvanlar) {
    if (x instanceof Kopek) {
        Kopek kk = (Kopek) x;
        kk.havla();
    } else if (x instanceof Kedi) {
        ((Kedi) x).tirmala();
    } else {
        System.out.println("Özel davranış yok");
    }
}
```

```
Köpek havlıyor
Kedi tırmalıyor
Özel davranış yok
```

Önce sınama, sonra dönüşüm yapıldığı için `ClassCastException` oluşamaz. `x instanceof Kopek` doğruysa `x`'in gösterdiği nesne kesinlikle bir köpektir ve `(Kopek) x` başarılı olur. Üçüncü öğe düz bir `Hayvan`'dır; iki sınama da yanlış çıkar ve `else` dalı çalışır.

Sınamanın sırası da önemlidir. `instanceof` alt sınıfları da kapsar. `x instanceof Hayvan` sınaması köpek için de, kedi için de `true` verir. Bu yüzden zincirde önce en alttaki, en özel sınıflar sınanır. Sıra ters kurulursa genel dal her nesneyi yakalar ve özel dallara hiç sıra gelmez.

Java 16 ile gelen **örüntü eşleme** (*pattern matching*) bu kalıbı kısaltır. Sınama doğru çıkarsa dönüştürülmüş değer doğrudan yeni bir değişkene bağlanır:

```
if (x instanceof Kopek kk) {
    kk.havla(); // dönüşüm otomatik
}
```

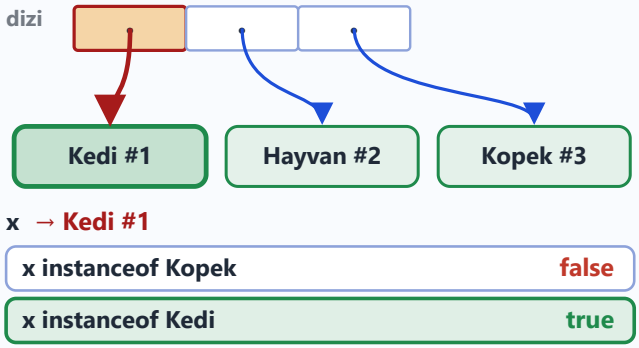
`kk` değişkeni yalnız `if`'in doğru dalında tanımlıdır ve tipi `Kopek`'tir. Dönüşümü derleyici yazar; programcının ayrıca `(Kopek) x` yazmasına gerek kalmaz. Sınama ile dönüşüm ayrı satırlarda olmadığı için birinin değiştirilip ötekinin unutulması gibi bir hata da yapılamaz.

Aynı döngünün öğe sırası değiştirilmiş bir sürümü izlenirse sınamaların her turda nasıl çalıştığı daha açık görülür.

```
Hayvan[] dizi = {new Kedi(), new Hayvan(), new Kopek()};
for (Hayvan x : dizi) {
    if (x instanceof Kopek) {
        ((Kopek) x).havla();
    } else if (x instanceof Kedi) {
        ((Kedi) x).tirmala();
    } else {
        System.out.println("Özel davranış yok");
    }
}
```

Tur	`x` → nesne	`instanceof Kopek`	`instanceof Kedi`	Çalışan dal	Çıktı
1	`Kedi` #1	false	true	`((Kedi) x).tirmala()`	`Kedi tırmalıyor`
2	`Hayvan` #2	false	false	`else`	`Özel davranış yok`
3	`Kopek` #3	true	— (sınanmaz)	`((Kopek) x).havla()`	`Köpek havlıyor`

Tablo 4.4.5: `instanceof` zincirinin izlemesi. Dizi: `{Kedi, Hayvan, Kopek}` .



Şekil 4.4.7: İlk tur. `x` bir `Kedi` 'yi gösterir; `Kopek` sınaması yanlış, `Kedi` sınaması doğru çıkar. Önce sınandığı için `(Kedi) x` dönüşümü güvenlidir.

İkinci turda düz bir `Hayvan` nesnesi vardır. Bu nesne hiçbir alt sınıfın örneği değildir; iki sınama da yanlış çıkar. `else` dalında hiçbir dönüşüm denenmediği için hata riski de yoktur. Üçüncü turda ilk sınama doğru çıkar; `else if` zincirinde sonraki koşul sınanmaz. Üç turun hiçbirinde `ClassCastException` oluşmaz.

Uzun `instanceof` zincirleri ise çoğunlukla bir tasarım uyarısıdır. Yeni bir alt sınıf eklendiğinde (örneğin `Kus`) zincire yeni bir dal eklenmesi gerekir; unutulursa kuş sessizce `else` dalına düşer. Daha iyi çözüm, davranışı üst sınıfta tanımlanıp alt sınıflarda geçersiz kılınan bir metoda taşımaktır.

4.4.3.6 Diziler ve `ArrayStoreException`

Referans kuralı dizilere de uygulanır. `Kopek` bir `Hayvan` olduğuna göre `Kopek[]` de bir `Hayvan[]` sayılır. Java dizilerinin bu özelliğine kovaryans denir.

Tanım 4.4.9

Diziler **kovaryanttır** (*covariant*): `A`, `B`'nin alt tipiyse `A[]` de `B[]`'nin alt tipidir. Bu yüzden `Kopek[]` nesnesi `Hayvan[]` referansına atanabilir. Dizi nesnesi ise kendi gerçek öğe tipini bilir. Diziye yazılan her öğe çalışma zamanında bu tiple denetlenir; uymayan öğe için `ArrayStoreException` fırlatılır.

```
Kopek[] kopekler = new Kopek[2];
Hayvan[] hayvanlar = kopekler;   // derlenir
hayvanlar[0] = new Kopek();
System.out.println("ilk atama tamam");
hayvanlar[1] = new Kedi();       // derlenir ama...
System.out.println("buraya gelinmez");
```

```
ilk atama tamam
Exception in thread "main" java.lang.ArrayStoreException: Kedi
```

İkinci satırda yukarı dönüşüm yapılır ve derlenir. Dizi nesnesi ise gerçekte bir `Kopek[]`'dir; `hayvanlar` yalnız ona bakan ikinci bir referanstır. Üçüncü satırda diziye bir köpek yazılır; JVM'in denetimi geçer ve `ilk atama tamam` yazılır. Beşinci satırda `hayvanlar[1] = new Kedi();` derleyicinin gözünde geçerlidir: `Hayvan[]` tipindeki bir diziye bir `Hayvan` yazılır. Ama JVM yazma anında dizinin gerçek öğe tipine bakar. Bir `Kopek[]` dizisine kedi konamaz; `ArrayStoreException: Kedi` fırlatılır ve son satır çalışmaz.

Bu durum dizilerin tip güvenliğinde bilinen bir açıktır. Derleyicinin yakalayamadığı bir hatayı JVM her yazmada yaptığı denetimle yakalar. Okuma ise güvenlidir: `Kopek[]` dizisinden okunan her öğe bir `Hayvan`'dır. Sorun yalnız yazarken çıkar.

Jenerik koleksiyonlar aynı açığı derleme zamanında kapatır. `List<Kopek>`, `List<Hayvan>`'ın alt tipi değildir:

```
List<Hayvan> l = new ArrayList<Kopek>();
```

```
error: incompatible types: ArrayList<Kopek>
  cannot be converted to List<Hayvan>
```

Derleyici bu atamaya izin verseydi listeye `l.add(new Kedi())` ile bir kedi eklenebilirdi. Jenerik tip bilgisi çalışma zamanında tutulmadığı için JVM bunu yakalayamazdı. Java bu yüzden jenerikleri kovaryant yapmaz ve sorunu baştan derleme hatasına çevirir.

Sık yapılan hata

Dizi kovaryansına güvenmek: `Hayvan[] d = new Kopek[2]; d[1] = new Kedi();` derlenir, ama çalışırken `ArrayStoreException: Kedi` verir. Farklı alt sınıflardan nesneler tutacak bir dizi baştan üst sınıf tipinde oluşturulur: `new Hayvan[2]` .

4.4.4 Dinamik bağlama

Bir metot çağrısının, çalışacak metot gövdesiyle eşleştirilmesine bağlama denir. Bu eşleştirme iki zamanda yapılabilir. Derleme sırasında yapılırsa statik (erken) bağlama, program çalışırken yapılırsa dinamik (geç) bağlama adını alır. Java'da geçersiz kılınabilen örnek metotları dinamik bağlanır ve çok biçimliliği mümkün kılan da budur.

4.4.4.1 Derleyici “var mı?”, JVM “hangisi?” diye sorar

Tanım 4.4.10

Dinamik bağlama (*dynamic binding*), çağrılacak metot sürümünün değişkenin bildirilen tipine göre değil, nesnenin gerçek sınıfına göre ve çalışma zamanında seçilmesidir. Referans bir alt sınıf nesnesini gösteriyorsa ve alt sınıf metodu geçersiz kılmışsa alt sınıfın sürümü çalışır.

```
Hayvan h = new Kopek();
h.kos();      // Kopek sınıfının metodu
h.sesCikar(); // Kopek sınıfının metodu
h.uyu();      // Kopek'te yok → Hayvan'ınki
```

```
Köpek koşuyor
Hav hav!
Hayvan uyuyor
```

Üç çağrı da `Hayvan` tipindeki `h` üzerinden yapılır. `h.kos()` ve `h.sesCikar()` için `Kopek` 'in sürümleri çalışır, çünkü nesne bir köpektir ve bu iki metot `Kopek` 'te geçersiz kılınmıştır. `h.uyu()` için ise `Hayvan` 'ın sürümü çalışır. `Kopek` , `uyu()` 'yu geçersiz kılmamıştır; kalıtımla aldığı sürüm `Hayvan` 'ınkidir.

Her çağrı iki aşamada çözülür. Şekil 4.4.8 `h.kos()` için bu iki aşamayı gösterir.



Şekil 4.4.8: `h.kos()` çağrısının iki aşaması. Derleyici `Hayvan`'da `kos()` olup olmadığına bakar; JVM ise nesnenin sınıfına bakar ve `Kopek.kos()`'u çalıştırır.

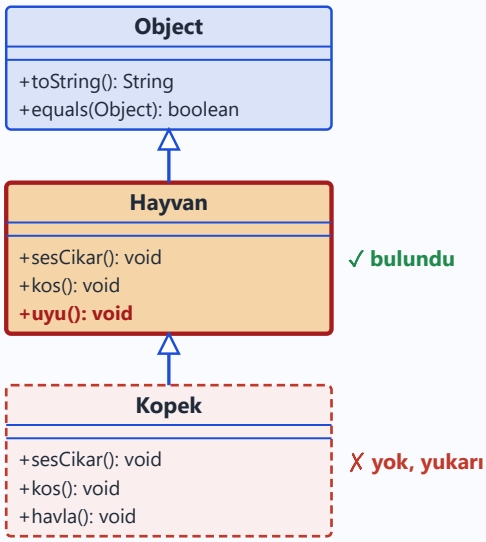
1. **Derleme zamanı.** Derleyici `h`'nin bildirilen tipine (`Hayvan`) bakar ve "`Hayvan`'da `kos()` var mı?" diye sorar. Varsa çağrı derlenir; yoksa derleme hatası verilir. Bu aşamada çalışacak gövde seçilmez; yalnız çağrının geçerli olduğu kesinleşir.
2. **Çalışma zamanı.** JVM, `h`'nin gösterdiği nesneye bakar ve "bu nesnenin sınıfında hangi `kos()` geçerli?" diye sorar. Gerçek sınıf `Kopek`'tir; `Kopek.kos()` çalışır.

Kural kısaca şöyle özetlenir: derleyici "var mı?" diye, JVM "hangisi?" diye sorar. Derleyicinin onayı, çalışma zamanında en az bir sürümün bulunacağını garanti eder. Çünkü `h` hangi alt sınıfın nesnesini gösterirse göstersin, o sınıf `Hayvan`'ın `kos()` metodunu ya kalıtımla almış ya da geçersiz kılmıştır.

4.4.4.2 Metot araması aşağıdan yukarı

JVM'in "hangisi?" sorusunu nasıl cevapladığı en kolay sınıf hiyerarşisi üzerinde gösterilir. Arama her zaman en alttan, nesnenin gerçek sınıfından başlar. Metot orada bulunursa arama biter. Bulunmazsa bir üstteki sınıfa çıkılır. Bu yürüyüş `Object`'e kadar sürebilir.

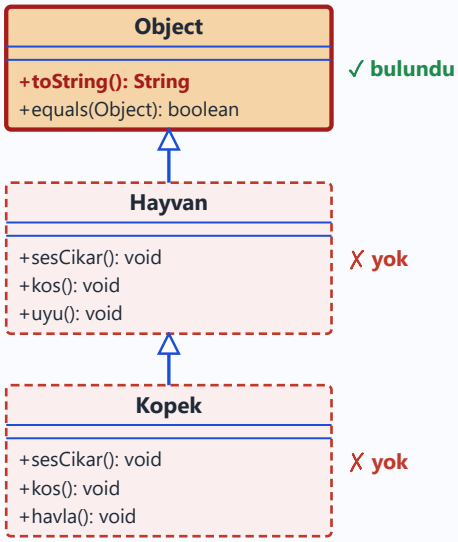
```
Hayvan h = new Kopek();
h.kos();
h.uyu();
System.out.println(h.toString());
// h.havla(); → derleme hatası
```



Şekil 4.4.9: `h.uyu()` için metot araması. Nesne bir `Kopek` 'tir; `Kopek` 'te `uyu()` yoktur. Arama bir üste çıkar ve `Hayvan` 'da biter.

Programdaki dört satırın her biri farklı bir durumu gösterir:

- `h.kos()` : arama `Kopek` 'te başlar. `kos()` orada vardır; arama biter ve `Köpek koşuyor` yazılır.
- `h.uyu()` : `Kopek` 'te `uyu()` yoktur. Bir üste çıkılır ve metot `Hayvan` 'da bulunur: `Hayvan uyuyor` (Şekil 4.4.9).
- `h.toString()` : ne `Kopek` 'te ne `Hayvan` 'da vardır. Arama en tepeye, `Object` 'e kadar çıkar. `Object` 'in `toString()` 'i sınıf adını, `@` işaretini ve onaltılık bir sayıyı (nesnenin hash kodunu) birleştirir; örneğin `Kopek@13221655` . Sayı çalıştırmaya göre değişir (Şekil 4.4.10).
- `h.havla()` : derlenmez. Bu kez aramayı derleyici yapar ve referans tipi `Hayvan` 'dan başlar. `Hayvan` 'da ve `Object` 'te `havla()` bulunmaz. `Kopek` 'e hiç bakılmaz, çünkü derleyici aşağı inmez.



Şekil 4.4.10: `h.toString()` için metot araması. `Kopek`'te ve `Hayvan`'da bulunmayan metot, zincirin tepesindeki `Object`'te bulunur.

Son madde iki aramanın farkını açıkça gösterir. Derleyicinin araması referans tipinden başlar ve yalnız yukarı gider. JVM'in araması nesnenin sınıfından başlar ve o da yalnız yukarı gider. JVM'in başladığı nokta derleyicininkinden daha aşağıda ya da onunla aynı yerdedir. Bu yüzden derleyicinin bulduğu her metodu JVM de bulur; belki daha aşağıda, geçersiz kılınmış bir sürümünü.

Örnek 4.4.3

`Kedi` nesnesini gösteren `Hayvan c = new Kedi();` için `c.sesCikar()`, `c.kos()` ve `c.uyu()` çağrılarında arama nerede biter?

`c.sesCikar()`: arama `Kedi`'de başlar ve orada biter, çünkü `Kedi` bu metodu geçersiz kılmıştır: `Miyav!`.

`c.kos()`: `Kedi`'de `kos()` yoktur; arama `Hayvan`'da biter: `Hayvan koşuyor`. Aynı çağrı bir köpekte `Köpek koşuyor` yazar.

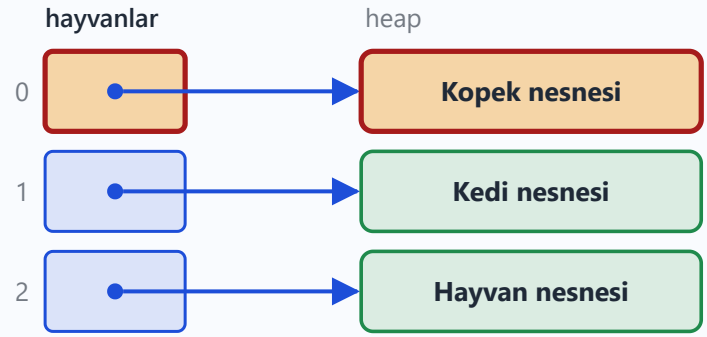
`c.uyu()`: yine `Hayvan`'da biter: `Hayvan uyuyor`.

4.4.4.3 Kod izleme: çok biçimli dizi

Çok biçimliliğin en sık kullanıldığı yer, üst sınıf tipinde bir dizi ya da listedir. Dizinin öğeleri farklı alt sınıfların nesnelerini gösterir; tek bir döngü hepsine aynı çağrıyı yapar.

```
Hayvan[] hayvanlar = new Hayvan[3];
hayvanlar[0] = new Kopek();
hayvanlar[1] = new Kedi();
hayvanlar[2] = new Hayvan();
for (int i = 0; i < hayvanlar.length; i++) {
    hayvanlar[i].sesCikar();
}
```

İlk satır, `Hayvan` tipinde üç öğeli bir dizi oluşturur. Öğeler nesne değil referanstır ve başlangıçta hepsi `null`'dır. Sonraki üç atama her öğeye farklı bir nesne bağlar. 0. öğe bir `Kopek`'i, 1. öğe bir `Kedi`'yi gösterir; 2. öğe ise düz bir `Hayvan`'dır. Her atamada yukarı dönüşüm kendiliğinden yapılır. Dizi artık üç farklı biçim tutar.



Şekil 4.4.11: Döngünün ilk turu. `hayvanlar[0]` bir `Kopek` nesnesini gösterir; JVM `Kopek.sesCikar()`'ı çalıştırır.

`i`	`i < hayvanlar.length`	`hayvanlar[i]` → nesne	Çalışan sürüm	Çıktı
0	0 < 3 doğru	`Kopek`	`Kopek.sesCikar()`	`Hav hav!`
1	1 < 3 doğru	`Kedi`	`Kedi.sesCikar()`	`Miyav!`
2	2 < 3 doğru	`Hayvan`	`Hayvan.sesCikar()`	`Hayvan sesi`
3	3 < 3 yanlış	—	—	döngü biter

Tablo 4.4.6: Çok biçimli dizi döngüsünün izlemesi.

Döngü gövdesindeki `hayvanlar[i].sesCikar()` satırı üç turda da aynıdır. Derleyici bu satırı bir kez, `Hayvan.sesCikar()` çağrısı olarak derler. JVM ise her turda nesnenin sınıfına bakar ve farklı bir gövde seçer. Tek satır kod, üç farklı davranış üretir. Diziye yarın bir `Kus` nesnesi eklense döngüde hiçbir değişiklik gerekmez; `Kus`'un `sesCikar()` sürümü kendiliğinden çalışır.

Sık yapılan hata

Diziyi oluşturup öğelere nesne atamadan çağrı yapmak. `new Hayvan[3]` üç hayvan oluşturmaz; üç boş referans oluşturur. Atama yapılmadan `hayvanlar[0].sesCikar()` çağrılırsa program `NullPointerException` ile durur.

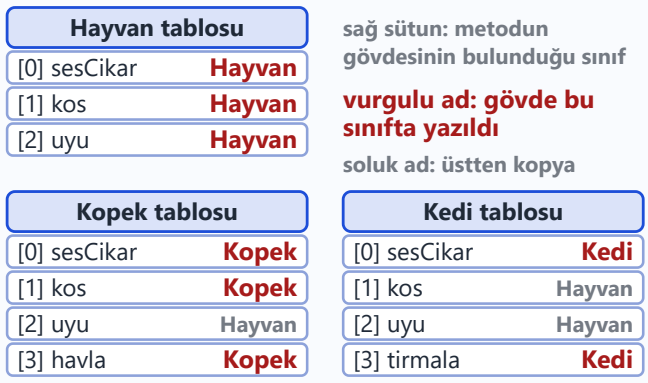
4.4.4.4 Perde arkası: metot tablosu

Hiyerarşide aşağıdan yukarı arama, dinamik bağlamayı anlamak için doğru bir modeldir. JVM'ler ise her çağrıda sınıflarda tek tek gezinmez; bu yavaş olurdu. Gerçekte arama işi büyük ölçüde sınıf yüklenirken, bir kez yapılır.

Tanım 4.4.11

Metot tablosu (*virtual method table*), JVM'in dinamik bağlamada kullandığı, sınıf başına bir tane tutulan tablodur. Geçersiz kılınabilir her metodun tabloda sabit bir sıra numarası, yani **yuvası** vardır. Yuvada, o sınıfın nesnelerinde çalışacak metot gövdesinin adresi durur.

Tablolar kalıtım boyunca şöyle kurulur. Alt sınıfın tablosu üst sınıfınkinin kopyasıyla başlar. Geçersiz kılınan metotların yuvalarına yeni gövde yazılır. Alt sınıfın eklediği yeni metotlar ise tablonun sonuna eklenir. Böylece bir metodun yuva numarası bütün alt sınıflarda aynı kalır. Şekil 4.4.12 örnek hiyerarşinin tablolarını gösterir.



Şekil 4.4.12: `Hayvan`, `Kopek` ve `Kedi` sınıflarının metot tabloları. Sağ sütun, metodun gövdesinin bulunduğu sınıftır. Vurgulu ad, gövdenin o sınıfta yazıldığını; soluk ad, üst sınıftan kopyalandığını gösterir.

Şekildeki tablolar sadeleştirilmiştir. Gerçek tablolar `Object`'in geçersiz kılınabilir metotlarıyla (`toString`, `equals`, `hashCode` gibi) başlar ve yuva numaraları JVM'e göre değişir. İlke ise aynıdır. `Kopek` tablosunda 0 ve 1 numaralı yuvalar (`sesCikar`, `kos`) `Kopek`'in gövdelerini gösterir. 2 numaralı `uyu` yuvası `Hayvan`'dan kopyalanmıştır. `havla` ise sona eklenmiştir. `Kedi` tablosunda yalnız `sesCikar` değişmiş, `tirmala` sona eklenmiştir.

Derleyicinin ürettiği bayt kodu (ara kod, *bytecode*) bu yapıyı kullanır. Aşağıdaki iki satır, `javap -c Ana` komutunun çıktısından alınmıştır. Bu komut, derlenmiş bir sınıfın bayt kodunu okunabilir biçimde gösterir.

```
47: invokevirtual #15  // Method Hayvan.kos:()V
52: invokevirtual #18  // Method Hayvan.sesCikar:()V
```

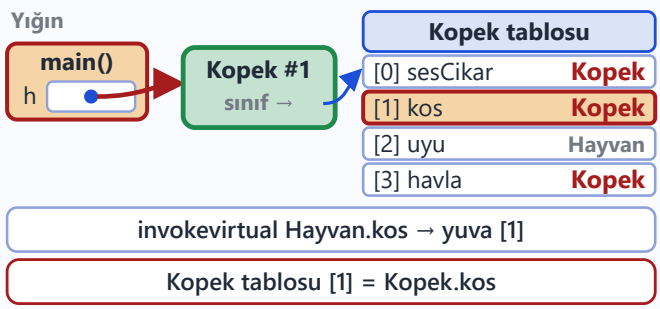
İki satırda da derleyici yalnız `Hayvan.kos` ve `Hayvan.sesCikar` yazmıştır, çünkü denetimi referans tipine göre yapmıştır. `Kopek` ya da `Kedi` adı bayt kodunda geçmez. Hangi gövdenin çalışacağına `invokevirtual` komutu karar verir. Bu komut çalıştığında JVM şu yolu izler: referans → nesne → nesnenin sınıfı → o sınıfın tablosundaki ilgili yuva → gövde. Arama tek bir dizi erişimidir; hiyerarşide yukarı yürümek gerekmez.

Bütün çağrılar `invokevirtual` ile yapılmaz. Yapıcılar ve `super.m()` çağrıları `invokespecial`, `static` metotlar `invokestatic` komutuyla çağrılır. Bu komutlarda çalışacak gövde derleme zamanında bellidir ve tabloya bakılmaz. `private` örnek metotları Java 11'den beri `invokevirtual` ile çağrılır. Ama geçersiz kılınamadıkları için tek adayları vardır ve JVM onu doğrudan seçer.

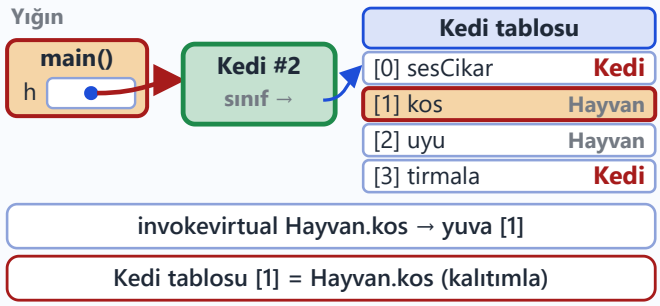
```
Hayvan[] d = {new Kopek(), new Kedi()};
for (Hayvan h : d) {
    h.kos();
    h.sesCikar();
}
```

Çağrı	Bayt kodu	Nesne	Tablodaki yuva	Çıktı
`h.kos()`	`Hayvan.kos`, yuva 1	`Kopek` #1	`Kopek.kos`	`Köpek koşuyor`
`h.sesCikar()`	`Hayvan.sesCikar`, yuva 0	`Kopek` #1	`Kopek.sesCikar`	`Hav hav!`
`h.kos()`	`Hayvan.kos`, yuva 1	`Kedi` #2	`Hayvan.kos` (kalıtımla)	`Hayvan koşuyor`
`h.sesCikar()`	`Hayvan.sesCikar`, yuva 0	`Kedi` #2	`Kedi.sesCikar`	`Miyav!`

Tablo 4.4.7: Metot tablosuyla çağrının izlemesi. Bayt kodu her çağrıda aynıdır; yalnız nesnenin tablosu değişir.



Şekil 4.4.13: İlk `h.kos()` çağrısı. Nesne bir `Kopek`'tir; onun tablosunun 1. yuvasında `Kopek.kos` vardır.



Şekil 4.4.14: Aynı bayt kodu, başka nesne. `Kedi`, `kos`'u geçersiz kılmadığı için tablosunun 1. yuvası `Hayvan.kos`'u gösterir.

Programın dört satırlık çıktısı tablonun son sütunudur. Dört çağrının hiçbirinde `if` ya da tip sınaması yapılmamıştır; seçimi tablolar yapmıştır. Şekil 4.4.13 ile Şekil 4.4.14 karşılaştırıldığında farkın yalnız nesnenin bağlı olduğu tabloda olduğu görülür. Yol, yuva numarası ve bayt kodu aynıdır.

Bu yapı dinamik bağlamanın maliyetinin de küçük olduğunu gösterir. Bir sanal çağrı, düz bir çağrıya göre yalnız bir iki bellek okuması fazladır. Modern JVM'ler çoğu durumda bunu da ortadan kaldırır. Bir çağrı noktasında hep aynı sınıftan nesneler görülüyorsa çalışma sırasında derleyen bileşen (JIT) çağrıyı doğrudan o gövdeye bağlayabilir. Bu yüzden çok biçimlilikten başarım kaygısıyla kaçınmak gerekmez.

4.4.4.5 Neler dinamik bağlanmaz?

Dinamik bağlama yalnız geçersiz kılınabilen örnek metotları için geçerlidir. Alanlar, `static` metotlar, `private` ve `final` metotlar statik bağlanır: hangisinin kullanılacağına derleyici, referansın bildirilen tipine bakarak karar verir.

```
class Hayvan {
    String tur = "hayvan";

    static void bilgi() {
        System.out.println("Hayvan.bilgi()");
    }

    void tanit() {
        System.out.println("Ben bir " + tur);
    }
}

class Kopek extends Hayvan {
    String tur = "köpek"; // gizler, ezmez

    static void bilgi() {
        System.out.println("Kopek.bilgi()");
    }

    @Override
    void tanit() {
        System.out.println("Ben bir " + tur);
    }
}
```

Bu iki sınıfta iki farklı durum vardır. `Kopek`, `tur` adında yeni bir alan tanımlar. Bu alan `Hayvan`'ın `tur` alanını ezmez, **gizler**: bir `Kopek` nesnesinde iki ayrı `tur` alanı bulunur. `Kopek` ayrıca aynı imzalı bir `static bilgi()` metodu tanımlar. `static` metotlar da geçersiz kılınmaz, gizlenir. `tanit()` ise sıradan bir örnek metodudur ve gerçekten geçersiz kılınmıştır.

Tanım 4.4.12

Alt sınıfta üst sınıftakiyle aynı adlı bir alan ya da aynı imzalı bir `static` metot tanımlanmasına **gizleme** (*hiding*) denir. Gizlenen üye silinmez; nesnede ya da sınıfta varlığını sürdürür. Hangisinin kullanılacağı referansın bildirilen tipine göre derleme zamanında belirlenir.

```
Hayvan h = new Kopek();
System.out.println(h.tur);
h.bilgi();
h.tanit();
```

```
hayvan
Hayvan.bilgi()
Ben bir köpek
```

Üye	Neye göre seçilir?	Bağlama
Örnek metodu	Nesne tipi	Dinamik
Alan (değişken)	Referans tipi	Statik
`static` metot	Referans tipi	Statik
`private` / `final` metot	Geçersiz kılınamaz; tek aday vardır	Statik

Tablo 4.4.8: Hangi üye neye göre seçilir?

`final` metot alt sınıfta geçersiz kılınamaz; denenirse derleyici `overridden method is final` der. `private` metot ise alt sınıfa hiç görünmez. Alt sınıf aynı adla bir metot yazarsa bu yeni ve ilgisiz bir metottur. Bu iki durumda da çağrı için tek aday vardır ve seçim derleme zamanında bellidir.

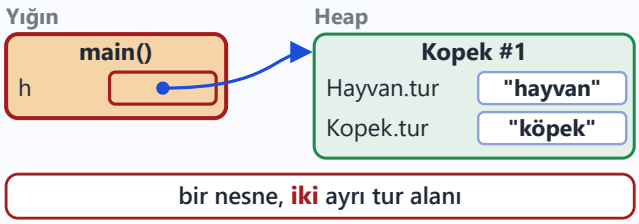
`static` metot bir nesneye değil, sınıfa aittir. Java, `h.bilgi()` gibi bir referans üzerinden çağrıya izin verir, ama bu yazım yanıltıcıdır. Derleyici çağrıyı sessizce `Hayvan.bilgi()` 'ye çevirir; referansın gösterdiği nesneye hiç bakılmaz. Referans `null` olsa bile çağrı çalışır ve `NullPointerException` çıkmaz. Bu yüzden `static` metot her zaman sınıf adıyla çağrılır: `Kopek.bilgi()`. Alt sınıfta üst sınıftakiyle aynı adlı alan tanımlamaktan da kaçınılır.

4.4.4.6 Kod izleme: alan gizleme ve `static`

Aynı fikir, değer döndüren metotlarla yazılmış kısa bir programda adım adım izlenebilir.

```
class Hayvan {
    String tur = "hayvan";
    static String bilgi() { return "Hayvan.bilgi"; }
    String tanit() { return "Ben bir " + tur; }
}
class Kopek extends Hayvan {
    String tur = "köpek";            // gizler
    static String bilgi() { return "Kopek.bilgi"; }
    String tanit() { return "Ben bir " + tur; }
}
Hayvan h = new Kopek();   // main
System.out.println(h.tur);
System.out.println(h.bilgi());
System.out.println(h.tanit());
System.out.println(((Kopek) h).tur);
```

```
hayvan
Hayvan.bilgi
Ben bir köpek
köpek
```

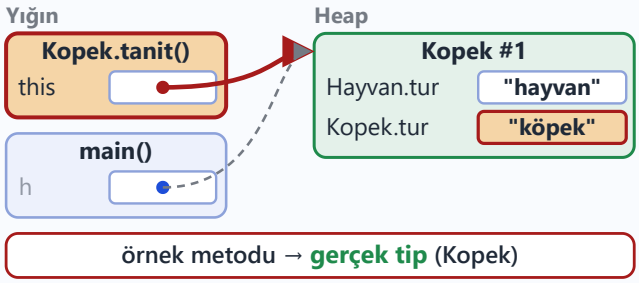


Şekil 4.4.15: Bir nesne, iki ayrı `tur` alanı. `Kopek`'teki `tur`, `Hayvan`'dakini ezmez; ikisi de nesnede yer alır.

İfade	Üye türü	Seçim ölçütü	Seçilen	Çıktı
`h.tur`	Alan	Bildirilen tip `Hayvan`	`Hayvan.tur`	`hayvan`
`h.bilgi()`	`static` metot	Bildirilen tip `Hayvan`	`Hayvan.bilgi()`	`Hayvan.bilgi`
`h.tanit()`	Örnek metodu	Gerçek tip `Kopek`	`Kopek.tanit()`; içinde `tur` = `Kopek.tur`	`Ben bir köpek`
`((Kopek)h).tur`	Alan	Bildirilen tip `Kopek` (dönüşümle)	`Kopek.tur`	`köpek`

Tablo 4.4.9: Alan gizleme programının izlemesi. `h`'nin bildirilen tipi `Hayvan`, gösterdiği nesne `Kopek`'tir.

Üçüncü satır ayrıca dikkat ister. `h.tanit()` dinamik bağlanır ve `Kopek.tanit()` çalışır. Bu metodun içindeki `tur` ifadesi de bir alan erişimidir ve o da statik bağlanır. Ama bu kez bildirilen tip, metodun yazıldığı sınıftır. `Kopek` sınıfında yazılan kodda `tur`, `this.tur` anlamına gelir ve `this`'in tipi `Kopek`'tir. Bu yüzden `Kopek.tur` okunur ve `Ben bir köpek` yazılır.



Şekil 4.4.16: `h.tanit()` çağrısı. Örnek metodu gerçek tipe göre seçilir; `Kopek.tanit()` içindeki `this`, `Kopek` tipindedir ve `Kopek.tur`'u okur.

Karşılaştırma için `tanit()` metodunun `Kopek`'te geçersiz kılınmadığı düşünülsün. O zaman `h.tanit()`, `Hayvan`'daki gövdeyi çalıştırır. Bu gövde `Hayvan` sınıfında yazıldığı için içindeki `tur`, `Hayvan.tur`'dur ve çıktı `Ben bir hayvan` olur. Nesne yine bir köpektir. Alan gizlemenin kafa karıştırıcı yanı budur: aynı nesnede, aynı adla iki değer vardır ve hangisinin okunacağı kodun nerede yazıldığına bağlıdır.

Son satırda dönüşüm, ifadenin bildirilen tipini `Kopek` yapar. Aynı nesnenin diğer alanı okunur: `köpek`. Nesne değişmemiştir; değişen yalnız derleyicinin baktığı tiptir.

Sık yapılan hata

Alanların da geçersiz kılındığını sanmak. `h.tur` her zaman referans tipine göre okunur. Alt sınıfa göre değişen bir bilgi gerekiyorsa alan `private` yapılır ve bilgiyi döndüren bir metot (getter) yazılır. Metot geçersiz kılınabilir; alan kılınamaz.

4.4.4.7 Yapıcıdan geçersiz kılınabilir metot çağırmak

Dinamik bağlama nesne kurulurken de çalışır. Bu, beklenmedik bir tuzak doğurur: üst sınıfın yapıcısı, alt sınıfın henüz hazırlanmamış alanlarını kullanan bir metodu çalıştırabilir.

```
class Bilesen {
    Bilesen() {
        System.out.println("Bilesen kuruluyor");
        ciz();
    }
    void ciz() { System.out.println("boş bileşen"); }
}
class Etiket extends Bilesen {
    private String metin = "Merhaba";
    Etiket() { System.out.println("Etiket kuruldu"); }
    @Override void ciz() {
        System.out.println("Etiket: " + metin);
    }
}
Etiket e = new Etiket();   // main
e.ciz();
```

```
Bilesen kuruluyor
Etiket: null
Etiket kuruldu
Etiket: Merhaba
```

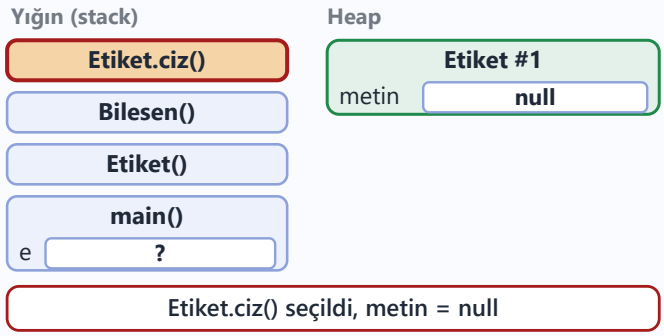
Çıktının ikinci satırı şaşırtıcıdır. `metin` alanına tanımında `"Merhaba"` değeri verilmiştir, ama `Etiket: null` yazılır. Nedeni, nesne kurulurken işlerin yapıldığı sıradır.

Adım	Çalışan kod	`metin`	Çıktı
1	`new` nesneye yer ayırır; alanlar varsayılan değeri alır	`null`	—
2	`Etiket()` 'in ilk işi gizli `super()` çağrısıdır	`null`	—
3	`Bilesen()` gövdesi, ilk satır	`null`	`Bilesen kuruluyor`
4	`ciz()` = `this.ciz()` ; nesne bir `Etiket` → `Etiket.ciz()`	`null`	`Etiket: null`
5	`super()` döndü; alt sınıfın alan ilk değerleri çalışır	`"Merhaba"`	—
6	`Etiket()` gövdesi	`"Merhaba"`	`Etiket kuruldu`

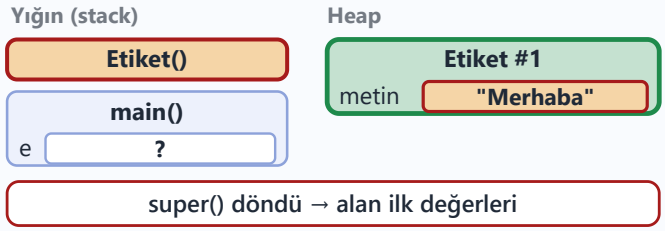
Adım	Çalışan kod	`metin`	Çıktı
7	`main` : `e.ciz()`	`"Merhaba"`	`Etiket: Merhaba`

Tablo 4.4.10: `new Etiket()` çağrısının izlemesi.

Kritik nokta 4. ve 5. adımların sırasıdır. Java'da bir alt sınıf yapıcısı önce üst sınıf yapıcısını çalıştırır. Alt sınıfın alanlarına tanımlarında verilen ilk değerler ise üst sınıf yapıcısı döndükten sonra atanır. `Bilesen()`'deki `ciz()` çağrısı dinamik bağlanır; nesnenin sınıfı `Etiket` olduğu için geçersiz kılınmış sürüm seçilir. O anda `metin` hâlâ varsayılan değeri olan `null`'dadır. Alt sınıfın metodu yarı kurulmuş bir nesne üzerinde çalışır.



Şekil 4.4.17: 4. adım. Yığında `Etiket()`, onun üstünde `Bilesen()` ve en üstte `Etiket.ciz()` vardır. Heap'teki nesnenin `metin` alanı henüz `null`'dır.



Şekil 4.4.18: 5. adım. `super()` döndükten sonra alan ilk değeri çalışır ve `metin`'e `"Merhaba"` atanır. `ciz()` ise çoktan çağrılmıştır.

Bu örnekte metot alanı yalnız yazdırdığı için zarar küçüktür. Metot alanı kullansaydı, örneğin `metin.length()` çağırsaydı, program çökerdi:

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()"
because "this.metin" is null
        at Etiket.ciz(Ana.java:21)
        at Bilesen.<init>(Ana.java:16)
        at Etiket.<init>(Ana.java:19)
```

JDK 15'ten beri `NullPointerException` iletisi varsayılan olarak hangi ifadenin `null` olduğunu da söyler: `"this.metin" is null`. Yığın izindeki (*stack trace*) satırların sırası da durumu özetler. En üstte hatanın çıktığı `Etiket.ciz`, altında onu çağıran `Bilesen` yapıcısı, onun altında da `Etiket` yapıcısı vardır. `<init>`, bayt kodunda yapıcının adıdır. Zincir, `Etiket`'in yapıcısından başlayıp yine `Etiket`'in bir metoduna dönen bir döngü çizer.

Tanım 4.4.13

Yapıcı kuralı: Bir yapıcıdan yalnız `private`, `final` ya da `static` metotlar çağrılmalıdır. Bu metotlar geçersiz kılınamaz; bu yüzden alt sınıfın henüz kurulmamış durumuna erişemezler.

Sık yapılan hata

Üst sınıf yapıcısında "şablon" niyetiyle geçersiz kılınabilir bir metot çağırmak (`ciz()`, `baslat()`, `yukle()` gibi). Alt sınıf bu metodu geçersiz kılarsa metot, alt sınıfın alanları atanmadan çalışır. Sonuç `null`, `0` ya da `false` değerleriyle yanlış çıktı veya `NullPointerException`'dır. Böyle bir hazırlık işi gerekiyorsa nesne kurulduktan sonra ayrıca çağrılır.

4.4.5 Aşırı yükleme

Aşırı yükleme (*method overloading*), aynı adlı ama parametre listesi farklı birden çok metot tanımlamaktır. Metotlar konusundan tanıdık bir araçtır: `Math.abs`'ın `int`, `long`, `float` ve `double` sürümleri vardır; `println`'in on sürümü vardır. Kalıtım ve dinamik bağlamayla birlikte kullanıldığında ise aşırı yüklemenin bir özelliği önem kazanır: seçim derleme zamanında yapılır.

4.4.5.1 Aşırı yükleme derleme zamanında seçilir

```
static void yazdir(Hayvan h) {
    System.out.println("yazdir(Hayvan)");
}

static void yazdir(Kopek k) {
    System.out.println("yazdir(Kopek)");
}

public static void main(String[] args) {
    Kopek kopek = new Kopek();
    Hayvan hayvan = kopek;
    yazdir(kopek);
    yazdir(hayvan);
    yazdir((Kopek) hayvan);
}
```

```
yazdir(Kopek)
yazdir(Hayvan)
yazdir(Kopek)
```

Programın üç çağrısı da aynı nesneyle yapılır. Yine de ikinci çağrı farklı bir sürüm seçer. İlk çağrıda argüman `kopek`'tir ve bildirilen tipi `Kopek`'tir; iki sürüm de uyar, ama `yazdir(Kopek)` daha özeldir. İkinci çağrıda argüman `hayvan`'dır. Nesne bir köpek olsa da `hayvan`'ın bildirilen tipi `Hayvan`'dır ve derleyici yalnız bunu bilir. `Hayvan` tipindeki bir argüman `yazdir(Kopek)`'e uymaz; tek aday `yazdir(Hayvan)`'dır. Üçüncü çağrıda açık dönüşüm ifadenin bildirilen tipini `Kopek` yapar ve yine `yazdir(Kopek)` seçilir.

Tanım 4.4.14

Aşırı yüklenmiş sürümler arasındaki seçimi derleyici, argümanların **bildirilen tiplerine** göre yapar. Seçilen imza bayt koduna yazılır ve program çalışırken değişmez. Argümanların gösterdiği nesnelerin gerçek sınıfı bu seçimi etkilemez.

Bu kural, geçersiz kılmanın kuralının tam tersidir. Geçersiz kılmada nesnenin gerçek tipi belirleyicidir; aşırı yüklemede argümanın bildirilen tipi. Tablo 4.4.11 iki mekanizmayı yan yana koyar.

	Aşırı yükleme	**Geçersiz kılma**
Nerede?	Aynı sınıf (ya da sınıf ve alt sınıfı)	Alt sınıf
İmza	Farklı parametre listesi	Aynı imza
Seçimi yapan	Derleyici	JVM
Ölçüt	Argümanın bildirilen tipi	Alıcı nesnenin gerçek tipi

	Aşırı yükleme	Geçersiz kılma
Zaman	Derleme zamanı	Çalışma zamanı

Tablo 4.4.11: Aşırı yükleme ile geçersiz kılmanın karşılaştırması.

Tablodaki "alıcı nesne" ifadesi ayrıca açıklanmalıdır. `a.m(b)` çağrısında `a` alıcıdır: metodun hangi nesne üzerinde çağrıldığını belirtir. `b` ise argümandır. Java'da dinamik bağlama yalnız alıcının tipine bakar. Argümanların tipine bakan çalışma zamanı seçimi yoktur; argümanlar için her şey derleme zamanında biter. Bu yüzden Java'nın dinamik bağlamasına tek yönlü seçim (*single dispatch*) denir.

Bir ayrıntı da dönüş tipiyle ilgilidir. Metot imzası yalnız ad ve parametre tiplerinden oluşur; dönüş tipi imzaya dahil değildir. Bu yüzden yalnız dönüş tipi farklı iki metot aşırı yüklenemez. `static int g(int x)` ve `static long g(int x)` aynı sınıfta yazılırsa derleyici `method g(int) is already defined` der.

4.4.5.2 En özel sürüm ve belirsizlik

Bir argüman birden çok sürüme uyabilir. Bu durumda derleyici en özel olanı seçer.

Tanım 4.4.15

İki aday metottan birinin her parametre tipi, ötekinin karşılık gelen parametre tipinin alt tipiyse (ya da aynısıysa), o metot daha **özeldir**. Derleyici uyan adaylar arasından en özel olanı seçer. Tek bir en özel aday yoksa çağrı **belirsizdir** ve derleme hatası verir.

```
static void goster(Object o) { System.out.println("goster(Object)"); }
static void goster(Hayvan h) { System.out.println("goster(Hayvan)"); }
static void goster(String s) { System.out.println("goster(String)"); }
// main içinde:
goster(new Kedi());
goster("a");
Object o = "a";
goster(o);
goster((Hayvan) null);
```

```
goster(Hayvan)
goster(String)
goster(Object)
goster(Hayvan)
```

Dört çağrıyı sırayla inceleyelim. `goster(new Kedi())`: kedi hem `Hayvan`'a hem `Object`'e uyar, `String`'e uymaz. `Hayvan`, `Object`'in alt tipidir, dolayısıyla daha özeldir: `goster(Hayvan)`. `goster("a")`: argüman `String` ve `Object` sürümlerine uyar; `String` daha özeldir. `goster(o)`:

`o`'nun gösterdiği nesne bir dizgi olsa da bildirilen tipi `Object`'tir. `Object` tipindeki bir argüman yalnız `goster(Object)`'e uyar.

Dördüncü çağrı özel bir değer kullanır. `null` her referans tipine uyar; bu yüzden `goster(null)` yazıldığında üç sürüm de aday olur. `Object` en genel olanıdır ve elenir. `Hayvan` ile `String` arasında ise kalıtım ilişkisi yoktur; hiçbiri ötekinden özel değildir. Derleyici seçim yapamaz:

```
error: reference to goster is ambiguous
  both method goster(Hayvan) in Ana and
  method goster(String) in Ana match
```

Çözüm, ifadenin bildirilen tipini açık dönüşümle belirlemektir: `goster((Hayvan) null)`. `(Hayvan) null` ifadesinin değeri yine `null`'dır, ama tipi `Hayvan`'dır. Artık `String` sürümü uymaz ve `goster(Hayvan)` seçilir. Çıktıdaki dördüncü satır budur.

Uyan sürümler tek bir kalıtım zinciri üzerindeyse belirsizlik olmaz. `goster(Object)`, `goster(Hayvan)` ve `goster(Kedi)` varken `goster(null)` derlenir ve en alttakini, `goster(Kedi)`'yi seçer. Çünkü `Kedi`, öteki iki tipin de alt tipidir. Belirsizlik ancak birbirinin alt tipi olmayan iki aday aynı anda uyduğunda çıkar.

> **Sık yapılan hata**
>
> İlişkisiz iki aşırı yükleme varken `null` geçirmek: `goster(null)` → `reference to goster is ambiguous`. Hangi sürümün istendiği açık dönüşümle belirtilir: `goster((String) null)` ya da `goster((Hayvan) null)`.

4.4.5.3 Kod izleme: derleyici mi, JVM mi?

En öğretici durum, iki mekanizmanın aynı çağrıda birlikte çalıştığı durumdur. Aşağıdaki programda `yazdir` aşırı yüklenmiştir; her sürümün içinde de geçersiz kılınmış `sesCikar()` çağrılır.

```
static void yazdir(Hayvan h) {
    System.out.print("yazdir(Hayvan): ");
    h.sesCikar();
}
static void yazdir(Kopek k) {
    System.out.print("yazdir(Kopek): ");
    k.sesCikar();
}
// main içinde:
Hayvan hayvan = new Kopek();
Hayvan diger = new Kedi();
yazdir(hayvan);
yazdir(diger);
yazdir((Kopek) hayvan);
```

```
yazdir(Hayvan): Hav hav!
yazdir(Hayvan): Miyav!
yazdir(Kopek): Hav hav!
```

Çağrı	Argümanın bildirilen tipi	Derleyicinin seçimi	Nesne	JVM'in seçimi	Çıktı
`yazdir(hayvan)`	`Hayvan`	`yazdir(Hayvan)`	`Kopek`	`Kopek.sesCikar()`	`yazdir(Hayvan): Hav hav!`
`yazdir(diger)`	`Hayvan`	`yazdir(Hayvan)`	`Kedi`	`Kedi.sesCikar()`	`yazdir(Hayvan): Miyav!`
`yazdir((Kopek) hayvan)`	`Kopek`	`yazdir(Kopek)`	`Kopek`	`Kopek.sesCikar()`	`yazdir(Kopek): Hav hav!`

Tablo 4.4.12: Her çağrıda iki karar. Derleyici `yazdir`'ın sürümünü, JVM `sesCikar()`'ın sürümünü seçer.



Şekil 4.4.19: İlk çağrıda iki karar. Derleyici bildirilen tipe (`Hayvan`) bakıp `yazdir(Hayvan)`'ı seçmiştir; metodun içinde JVM nesneye (`Kopek`) bakıp `Kopek.sesCikar()`'ı seçer.

İlk iki çağrıda değişkenlerin bildirilen tipi aynıdır, gösterdikleri nesneler farklıdır. Derleyici iki çağrıda da aynı sürümü, `yazdir(Hayvan)`'ı seçer. Metodun içindeki `h.sesCikar()` ise geçersiz kılınmış bir metottur ve JVM her seferinde nesneye bakar: önce `Hav hav!`, sonra `Miyav!`. Üçüncü çağrıda açık

dönüşüm ifadenin bildirilen tipini değiştirir ve derleyici `yazdir(Kopek)`'i seçer.

İzlemenin özeti tek cümledir: aşırı yükleme seçimi bildirilen tiple derlemede, geçersiz kılma seçimi gerçek tiple çalışırken yapılır. Bir çağrının ne yazdıracağını bulmak için iki soru sırayla sorulur. Önce "derleyici hangi imzayı seçer?", sonra "o imzanın hangi sınıftaki gövdesi çalışır?".

4.4.5.4 Aşırı yükleme seçiminin üç aşaması

Argüman ilkel bir değer olduğunda seçim bir adım daha karmaşıklaşır. Bir `int`, `long`'a genişletilebilir; `Integer`'a kutulanabilir; `int...` parametresine de verilebilir. Derleyici bu yolları aynı anda değil, sırayla dener.

Tanım 4.4.16

Derleyici aşırı yüklenmiş sürümleri **üç aşamada** arar:

1. **Kutulamasız aşama.** Yalnız tam eşleşme, ilkel genişletme (`int` → `long`) ve referans genişletmesi (`String` → `Object`) denenir.
2. **Kutulama aşaması.** 1. aşamada aday yoksa otomatik kutulama ve kutudan çıkarma da denenir: `double` → `Double` → `Object`.
3. **Değişken sayıda parametre aşaması.** Hâlâ aday yoksa `int...` gibi değişken sayıda parametreli sürümler devreye girer.

Bir aşamada en az bir aday bulunursa sonraki aşamalara geçilmez. O aşamanın adayları arasından en özel olan seçilir.

Kuralın en çok şaşırtan sonucu şudur: `f(long)` ve `f(Integer)` varken `f(5)` çağrısı `f(long)`'u seçer. `Integer`, `int`'in kendi sarmalayıcı sınıfıdır ve ilk bakışta daha yakın görünür. Ama ona ulaşmak için kutulama gerekir; `long`'a ise genişletmeyle, 1. aşamada ulaşılır. Genişletmenin kutulamadan önce gelmesinin nedeni tarihseldir. Kutulama Java 5'te eklendi. O zamana kadar `f(5)`, `f(long)`'u çağırıyordu. Yeni kural eski programların davranışını değiştirmesin diye kutulama ikinci aşamaya kondu.

Daraltma (`long` → `int`, `double` → `long`) hiçbir aşamada denenmez. Üç aşamada da aday bulunmazsa derleyici `no suitable method found` der. Örneğin yalnız `yazdir(Hayvan)` ve `yazdir(Kopek)` varken `yazdir(true)` yazılırsa ileti `no suitable method found for yazdir(boolean)` olur.

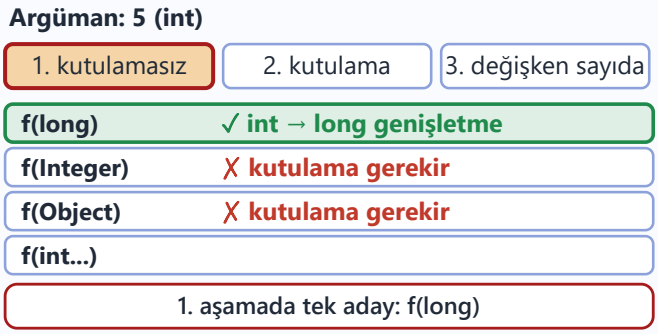
Bu sıra derleme zamanında uygulanır ve seçilen imza bayt koduna yazılır. Seçilen imzanın hangi sınıftaki gövdesinin çalışacağı ise yine çalışma zamanında belirlenir. Üç aşama yalnız "hangi imza?" sorusunu cevaplar.

```
static void f(long x)    { p("f(long)"); }
static void f(Integer x) { p("f(Integer)"); }
static void f(Object x)  { p("f(Object)"); }
static void f(int... x)  { p("f(int...)"); }
// main içinde (p: System.out.println):
byte b = 1;
f(5);
f(b);
f(Integer.valueOf(5));
f("x");
f(2.5);
f();
f(1, 2);
```

```
f(long)
f(long)
f(Integer)
f(Object)
f(Object)
f(int...)
f(int...)
```

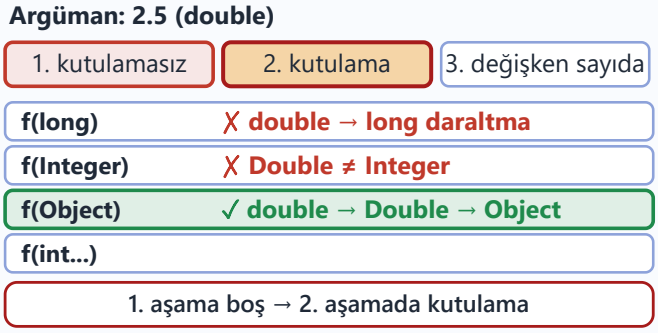
Çağrı	Argüman tipi	1. aşama adayları	2. aşama	3. aşama	Seçilen
`f(5)`	`int`	`f(long)` (genişletme)	—	—	`f(long)`
`f(b)`	`byte`	`f(long)` (genişletme)	—	—	`f(long)`
`f(Integer.valueOf(5))`	`Integer`	`f(Integer)`, `f(Object)`	—	—	`f(Integer)` (daha özel)
`f("x")`	`String`	`f(Object)`	—	—	`f(Object)`
`f(2.5)`	`double`	yok	`f(Object)` (`Double`'a kutulama)	—	`f(Object)`
`f()`	—	yok	yok	`f(int...)`	`f(int...)`
`f(1, 2)`	`int`, `int`	yok	yok	`f(int...)`	`f(int...)`

Tablo 4.4.13: Yedi çağrının aşama aşama çözümü.



Şekil 4.4.20: `f(5)` için aşamalar. 1. aşamada yalnız genişletmeyle `long` uyar; `Integer` ve `Object` kutulama ister ve bu aşamada denenmez.

Tablonun birkaç satırı ayrıca açıklanmalıdır. `f(b)`'de argüman `byte`'tır ve o da `long`'a genişler. Kutulansaydı bile `Byte` olurdu, `Integer` olmazdı; `Byte` ile `Integer` arasında hiçbir ilişki yoktur. `f(Integer.valueOf(5))`'te argüman zaten bir nesnedir. 1. aşamada iki aday vardır: tam eşleşmeyle `f(Integer)` ve referans genişletmesiyle `f(Object)`. `Integer`, `Object`'ten özel olduğu için seçilir. `f(long)` ise kutudan çıkarma gerektirir ve bu aşamada denenmez.



Şekil 4.4.21: `f(2.5)` için aşamalar. `double`, `long`'a daraltılamaz; 1. aşama boş kalır. 2. aşamada `Double`'a kutulanır ve `Object`'e uyar.

`f(2.5)` satırı daraltmanın hiç denenmediğini gösterir. `double`'dan `long`'a geçmek bilgi kaybettirir; derleyici bunu kendiliğinden yapmaz. 1. aşama boş kalınca 2. aşamaya geçilir. `2.5`, `Double`'a kutulanır. `Double`, `Integer`'a uymaz, ama `Object`'e uyar. Son iki çağrıda sıfır ya da iki argüman vardır. Bu sayıları yalnız `int...` parametresi karşılar ve ona en son, 3. aşamada başvurulur.

Örnek 4.4.4

Aynı dört sürüm (`f(long)`, `f(Integer)`, `f(Object)`, `f(int...)`) için `short s = 3; f(s);`, `f('A');` ve `f(Long.valueOf(4));` hangi sürümleri seçer?

`f(s)`: `short`, `long`'a genişler; 1. aşamada tek aday `f(long)`'dur.

`f('A')`: `char` da bir tam sayı tipidir ve `long`'a genişler: `f(long)`. Karakterin kodu, 65, `long` olarak verilir.

`f(Long.valueOf(4))`: argüman bir `Long` nesnesidir. `f(long)` kutudan çıkarma ister ve 1. aşamada denenmez. `Long`, `Integer`'a uymaz. 1. aşamada referans genişletmesiyle tek aday `f(Object)`'tir.

Sık yapılan hata

Kutulamanın önce geldiğini sanmak. `f(long)` ve `f(Integer)` varken `f(5)`, `f(Integer)`'ı değil `f(long)`'u çağırır: genişletme 1. aşamada, kutulama 2. aşamadadır.

4.4.6 Kalıtım ve çok biçimlilik

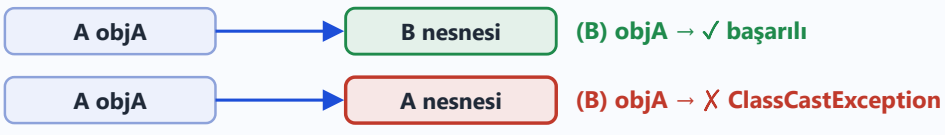
Çok biçimliliğin asıl değeri, programın tasarımında görülür. Çok biçimli kod, tip sınayan koda göre daha kısa ve yeni sınıflara karşı daha dayanıklıdır. Örneklerin çoğu bir şekil hiyerarşisi üzerine kuruludur.

4.4.6.1 Kalıtım ile çok biçimliliğin ilişkisi

İki kavramın ilişkisi tek yönlüdür. Çalışma zamanı çok biçimliliği için kalıtım gereklidir: bir sınıf hiyerarşisi olmadan geçersiz kılma da, dinamik bağlama da olmaz. Kalıtımı kullanmak için ise çok biçimlilik gerekli değildir. Bir sınıf yalnız kodu yeniden kullanmak için de başka bir sınıftan türetilebilir. Kalıtım hiyerarşiyi kurar; çok biçimlilik o hiyerarşiden yararlanır.

`B` sınıfı `A`'nın alt sınıfıysa `B`'nin nesnesi `A`'nın nesnesi gibi davranabilir; tersi doğru değildir. `A` referansını `B` referansına atamak için daraltma dönüşümü gerekir ve dönüşüm parantezle yazılır.

```
A objA = new B(); // B, A gibi davranır
B objB = (B) objA; // geçerli, ancak
                   // dikkatli kullanılmalı
```



Şekil 4.4.22: `(B) objA` dönüşümünün iki olası sonucu. `objA` bir `B` nesnesini gösteriyorsa dönüşüm başarılı olur; düz bir `A` nesnesini gösteriyorsa `ClassCastException` fırlatılır.

Son satır derlenir. Çalışırken `objA` gerçekten bir `B` nesnesini göstermiyorsa `ClassCastException` fırlatılır. Koddaki "dikkatli kullanılmalı" yorumu bu riski anlatır.

Java'da çok biçimliliğin tek kaynağı sınıf kalıtımı değildir. Arayüzler (*interface*) de aynı mekanizmayı sağlar. Bir arayüzü uygulayan (`implements`) her sınıfın nesnesi, o arayüz tipindeki bir referansa atanabilir. Bir sınıf yalnız bir üst sınıftan türeyebilir, ama birçok arayüzü uygulayabilir. `Hayvan h` yerine `SesCikarabilen s` gibi bir arayüz tipiyle de aynı dinamik bağlama çalışır.

4.4.6.2 Şekil hiyerarşisi

Bir çizim programında kareler, dikdörtgenler ve çemberler bulunsun. Her şeklin bir adı vardır; her şeklin alanı ve çevresi hesaplanabilir; her şekil çizilebilir. Ama alan ve çevre her şekilde farklı formülle bulunur. Bu ortak yapı soyut bir üst sınıfta toplanır.

```
abstract class Sekil {
    private String isim;

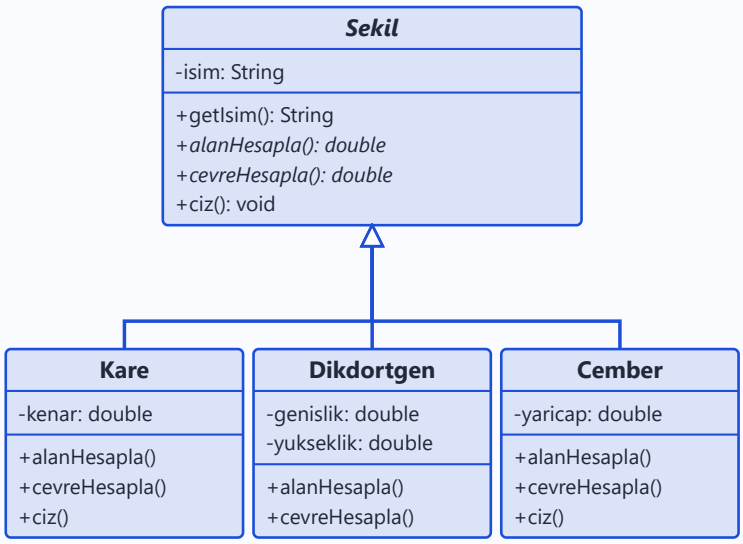
    public Sekil(String isim) {
        this.isim = isim;
    }

    public String getIsim() {
        return isim;
    }

    public abstract double alanHesapla();

    public abstract double cevreHesapla();

    public void ciz() {
        System.out.println(isim + " çiziliyor.");
    }
}
```



Şekil 4.4.23: Soyut `Sekil` sınıfı ve alt sınıfları `Kare`, `Dikdortgen`, `Cember`. Eğik yazılan ad ve metotlar soyuttur. `Dikdortgen`, `ciz()`'i geçersiz kılmaz.

`Sekil` sınıfının başındaki `abstract` sözcüğü sınıfın soyut olduğunu belirtir. Soyut bir sınıftan doğrudan nesne oluşturulamaz; `new Sekil("x")` yazılırsa derleyici `Sekil is abstract; cannot be instantiated` der. `alanHesapla()` ve `cevreHesapla()` da soyut metotlardır: gövdeleri yoktur ve her somut alt sınıf onları yazmak zorundadır. `ciz()` ise gövdeli, sıradan bir metottur. Alt sınıflar onu olduğu gibi kullanabilir ya da geçersiz kılabilir. `isim` alanı `private`'tır; alt sınıflar ona yapıcı ve `getIsim()` üzerinden ulaşır.

Soyut sınıf çok biçimlilik için ideal bir üst sınıftır. “Bir şeklin alanı” sorusunun genel bir cevabı yoktur; cevabı ancak somut şekil verebilir. Soyut metot bu durumu dile getirir: “her şeklin alanı vardır, ama nasıl hesaplanacağını alt sınıf söyler.”

```
class Kare extends Sekil {
    private double kenar;
    public Kare(double kenar) {
        super("Kare");
        this.kenar = kenar;
    }
    @Override
    public double alanHesapla() {
        return kenar * kenar;
    }
    @Override
    public double cevreHesapla() {
        return 4 * kenar;
    }
    @Override
    public void ciz() {
        System.out.println("Kare çiziliyor: kenar " + kenar);
    }
}
```

`Kare`'nin yapıcısı `super("Kare")` ile ismi üst sınıfa verir ve kenar uzunluğunu kendi alanında tutar. Üç metodu da geçersiz kılar. `Kare` soyut metotları yazmasaydı `abstract` olarak bildirilmedikçe derlenmezdi.

```
class Dikdortgen extends Sekil {
    private double genislik;
    private double yukseklik;
    public Dikdortgen(double genislik, double yukseklik) {
        super("Dikdörtgen");
        this.genislik = genislik;
        this.yukseklik = yukseklik;
    }
    @Override
    public double alanHesapla() {
        return genislik * yukseklik;
    }
    @Override
    public double cevreHesapla() {
        return 2 * (genislik + yukseklik);
    }
    // ciz() ezilmedi → Sekil'inki kullanılır
}
```

`Dikdortgen` alanı ve çevreyi geçersiz kılar, `ciz()`'i ise geçersiz kılmaz. Bir dikdörtgen çizildiğinde `Sekil.ciz()` çalışır ve ismi kullanarak `Dikdörtgen çiziliyor.` yazar. Kodun sonundaki yorum bunu hatırlatır.

```
class Cember extends Sekil {
    private double yaricap;
    public Cember(double yaricap) {
        super("Çember");
        this.yaricap = yaricap;
    }
    @Override
    public double alanHesapla() {
        return Math.PI * yaricap * yaricap;
    }
    @Override
    public double cevreHesapla() {
        return 2 * Math.PI * yaricap;
    }
    @Override
    public void ciz() {
        System.out.println("Çember çiziliyor: yarıçap " + yaricap);
    }
}
```

`Cember`, alanı ve çevreyi `Math.PI` sabitiyle hesaplar ve `ciz()`'i kendi bilgisiyle, yarıçapı da yazarak geçersiz kılar.

4.4.6.3 Kod izleme: şekiller dizisi

```
Sekil[] sekiller = new Sekil[3];
sekiller[0] = new Kare(5);
sekiller[1] = new Dikdortgen(4, 6);
sekiller[2] = new Cember(7);

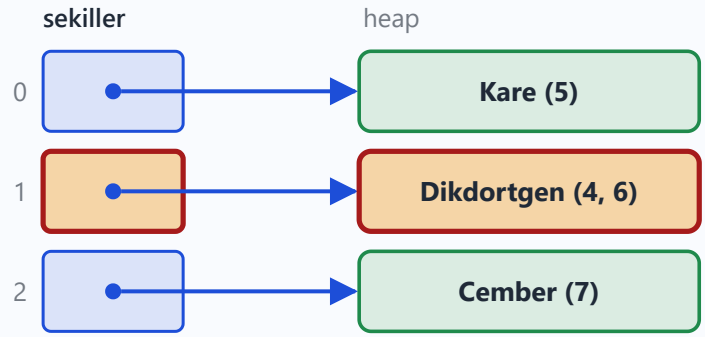
for (Sekil sekil : sekiller) {
    System.out.println(sekil.getIsim()
        + " alanı: " + sekil.alanHesapla());
    sekil.ciz();
}
```

```
Kare alanı: 25.0
Kare çiziliyor: kenar 5.0
Dikdörtgen alanı: 24.0
Dikdörtgen çiziliyor.
Çember alanı: 153.93804002589985
Çember çiziliyor: yarıçap 7.0
```

Dizi `Sekil` tipindedir; öğeler üç farklı alt sınıfın nesnelerini gösterir. Döngü her turda aynı iki çağrıyı yapar: `sekil.alanHesapla()` ve `sekil.ciz()`. `getIsim()` ise hiçbir alt sınıfta geçersiz kılınmamıştır; her turda `Sekil`'in sürümü çalışır ve yapıcıda verilen ismi döndürür.

Tur	Nesne	`alanHesapla()`	Alan	`ciz()`
1	`Kare(5)`	`Kare` : 5 × 5	25.0	`Kare.ciz()`
2	`Dikdortgen(4, 6)`	`Dikdortgen` : 4 × 6	24.0	`Sekil.ciz()` (geçersiz kılınmamış)
3	`Cember(7)`	`Cember` : π × 7 × 7	153.93804002589985	`Cember.ciz()`

Tablo 4.4.14: Şekiller dizisi döngüsünün izlemesi.



Şekil 4.4.24: İkinci tur. `sekiller[1]` bir `Dikdortgen`'i gösterir. `Dikdortgen`, `ciz()`'i geçersiz kılmadığı için arama yukarı çıkar ve `Sekil.ciz()` çalışır.

Çemberin alanı 49π'dir. `double` aritmetiği bu değeri 153.93804002589985 olarak hesaplar ve `println` bütün anlamlı basamakları yazar. Daha okunur bir çıktı isteniyorsa `printf("%.2f", ...)` kullanılır.

4.4.6.4 Neden çok biçimlilik?

Çok biçimliliğin değeri, onsuz yazılmış kodla karşılaştırınca görülür. Aşağıdaki metot bir şeklin alanını tip sınayarak bulur.

```
static double alanKotu(Sekil s) {
    double alan;
    if (s instanceof Kare) {
        alan = ((Kare) s).alanHesapla();
    } else if (s instanceof Dikdortgen) {
        alan = ((Dikdortgen) s).alanHesapla();
    } else if (s instanceof Cember) {
        alan = ((Cember) s).alanHesapla();
    } else {
        alan = 0; // yeni tür unutuldu!
    }
    return alan;
}
```

Metot doğru çalışır, ama kırılgandır. Hiyerarşiye yeni bir şekil eklendiğinde bu metot da değişmelidir. Bir `DikUcgen extends Sekil` sınıfı yazılıp metot güncellenmezse `alanKotu(new DikUcgen(3, 4))` hiçbir hata vermeden `0.0` döndürür. Programdaki her `instanceof` zinciri, yeni sınıf eklendiğinde unutulabilecek bir yer demektir. Üstelik metottaki dönüşümler gereksizdir: `alanHesapla()` zaten `Sekil`'de tanımlıdır ve dönüşüm yapılmadan da çağrılabilir.

```
static double toplamAlan(Sekil[] dizi) {
    double toplam = 0;
    for (Sekil sekil : dizi) {
        // her nesne kendi alanını
        toplam += sekil.alanHesapla();
    }
    return toplam;
}
```

Çok biçimli sürümde hiçbir tip sınaması yoktur. Her nesne kendi alanını kendisi hesaplar. Yeni bir `DikUcgen extends Sekil` sınıfı eklemek yeter; `toplamAlan` hiç değişmez. Örneğin `{Kare(5), Dikdortgen(4, 6), DikUcgen(3, 4)}` dizisi için sonuç 25 + 24 + 6 = **55.0**'dır. Kod kısalır ve `ClassCastException` riski ortadan kalkar.

> **Tanım 4.4.17**
>
> **Açık/kapalı ilkesi** (*open/closed principle*): Bir yazılım birimi genişletmeye açık, değiştirmeye kapalı olmalıdır. Yeni davranış, var olan kodu değiştirerek değil, yeni kod (örneğin yeni bir alt sınıf) ekleyerek kazandırılır.

`toplamAlan` bu ilkeye uyar: yeni şekillerle genişler, ama kendisi değişmez. `alanKotu` ise ilkeyi çiğner. Büyük programlarda fark çok daha belirgindir. Bir şekil düzenleyicide alan, çevre, çizim, kaydetme, yazdırma gibi onlarca işlem olabilir. Tip sınayan tasarımda yeni bir şekil bu işlemlerin hepsine dokunmayı gerektirir; çok biçimli tasarımda yalnız yeni sınıf yazılır.

Örnek 4.4.5

Şekil hiyerarşisine `DikUcgen(a, b)` eklensin: alan = a·b/2, çevre = a + b + hipotenüs. `ciz()` geçersiz kılınmasın ve yapıcı `super("Dik üçgen")` çağırsın. `DikUcgen(3, 4)` için alan, çevre ve `ciz()` çıktısı nedir?

Alan: 3 · 4 / 2 = **6.0**. Hipotenüs: $\sqrt{(3^2 + 4^2)} = \sqrt{25} = 5$, çevre: 3 + 4 + 5 = **12.0**. `ciz()` geçersiz kılınmadığı için `Sekil.ciz()` çalışır: `Dik üçgen çiziliyor.`

Şekil döngüsünde hiçbir şey değişmez; yalnız diziye yeni nesne eklenir. Çevre için kodda `a + b + Math.sqrt(a * a + b * b)` yazılır.

4.4.6.5 Çok biçimli parametre ve dönüş değeri

Çok biçimlilik yalnız dizilerde değil, metot parametrelerinde ve dönüş değerlerinde de kullanılır.

```
class Veteriner {
    void muayene(Hayvan h) {     // her alt sınıf
        System.out.print("Muayene: ");
        h.sesCikar();
    }
    static Hayvan sahiplen(String tur) { // gizli
        if (tur.equals("kedi")) return new Kedi();
        return new Kopek();
    }
}
// main içinde:
Veteriner v = new Veteriner();
v.muayene(new Kedi());
Hayvan h = Veteriner.sahiplen("köpek");
v.muayene(h);
System.out.println(h.getClass().getSimpleName());
```

```
Muayene: Miyav!
Muayene: Hav hav!
Kopek
```

`muayene` metodunun parametresi `Hayvan` tipindedir. Bu yüzden metot, bugün var olan ve ileride yazılacak her alt sınıfın nesnesini kabul eder. İlk çağrıda bir kedi, ikinci çağrıda bir köpek muayene edilir; metodun içindeki `h.sesCikar()` her seferinde nesnenin kendi sürümünü çalıştırır. Hiyerarşiye yeni bir `Kus` sınıfı eklendiğinde `muayene` değişmez.

`sahiplen` metodunun dönüş tipi de `Hayvan`'dır. Metot, hangi alt sınıfı ürettiğini çağırandan gizler. Çağıran yalnız `Hayvan` sözleşmesine güvenir: dönen nesne ses çıkarabilir, koşabilir, uyuyabilir. `sahiplen` ileride başka bir alt sınıf döndürmeye başlasa kullanan kod etkilenmez. Son satırdaki

`getSimpleName()`, nesnenin gerçek sınıfının paket adı olmadan yazılmış adını verir: `Kopek`.

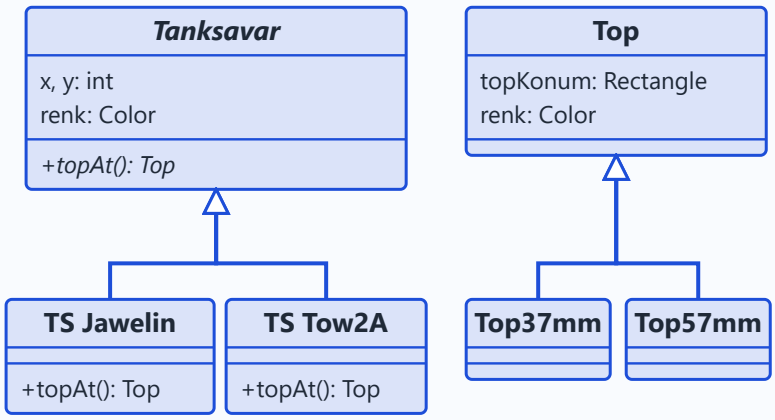
Tanım 4.4.18

Fabrika metodu (*factory method*), nesneyi üretip üst sınıf ya da arayüz tipinde döndüren metottur. Çağıran, hangi somut sınıfın üretildiğini bilmek zorunda kalmaz.

Bu örnekten genel bir ilke çıkar: parametre ve değişken tipleri, işi görmeye yeten en genel tip olarak seçilir. `muayene` yalnız `sesCikar()` çağırdığı için `Hayvan` parametresi yeterlidir; `Kopek` parametresi gereksiz yere kediyi dışarıda bırakırdı. Java kütüphanesinde de aynı ilke görülür: değişkenler `ArrayList<String>` yerine çoğunlukla `List<String>` tipinde tanımlanır.

4.4.6.6 Tanksavar oyunu

Çok biçimliliğin bir oyundaki kullanımı daha gerçekçi bir örnektir. Swing ile yazılmış bir tanksavar oyununda ekranın üstünden tanklar düşer; oyuncu ekranın altındaki tanksavarla onlara top atar. Skor 10'un katı olunca tanksavarın türü değişir ve farklı bir top atmaya başlar.



Şekil 4.4.25: Tanksavar oyununun iki hiyerarşisi. Soyut `Tanksavar` sınıfının `topAt()` metodunu iki alt sınıf geçersiz kılar; `Top` sınıfının da `Top37mm` ve `Top57mm` alt sınıfları vardır.

Şekildeki "TS Jawelin" `TanksavarJawelin`, "TS Tow2A" ise `TanksavarTow2A` sınıfıdır. `Tank` sınıfının da `TankT90MS` ve `TankLeopard2A4` alt sınıfları vardır. Oyun döngüsü ise yalnız `Tanksavar` ve `Tank` tiplerini bilir.

```
Tanksavar tanksavar = new TanksavarJawelin(x, y);
...
// hangi top? nesneye bağlı (dinamik bağlama)
toplar.add(tanksavar.topAt());
...
if (tanksavar instanceof TanksavarJawelin) {
    tanksavar = new TanksavarTow2A(tanksavar.x, tanksavar.y);
}
```

`toplar.add(tanksavar.topAt())` satırı hangi topun üretileceğini bilmez. `topAt()` soyut bir metottur; çağrı dinamik bağlanır ve o anda tanksavar hangi sınıftansa onun sürümü çalışır. Dönüş tipi `Top`'tur; dönen nesne `Top37mm` ya da `Top57mm` olabilir. `toplar` listesi de `Top` tipinde nesneler tutar ve hepsini aynı döngüyle hareket ettirip çizer.

Koddaki `instanceof` sınaması ise başka bir iş görür: tanksavarı yükseltmek. Sınama yalnız yükseltme anında bir kez yapılır; her top atışında yapılmaz. Yeni tanksavar nesnesi eskisinin konumunu (`x`, `y`) alır ve `tanksavar` değişkeni artık onu gösterir. Değişkenin bildirilen tipi değişmemiştir; değişen yalnız gösterdiği nesnedir. Bu andan sonra aynı `topAt()` çağrısı başka bir gövdeyi çalıştırır.

```
Top t1 = tanksavar.topAt(); // Jawelin
// → Top37mm, yükseklik 30
tanksavar = new TanksavarTow2A(100, 540);
Top t2 = tanksavar.topAt();
// → Top57mm, yükseklik 40
```

Oyundan bağımsız bir sınama çıktısı bunu doğrular. Aynı `topAt()` çağrısı, tanksavar değişince farklı bir top üretir: Jawelin yüksekliği 30 olan bir `Top37mm`, Tow2A yüksekliği 40 olan bir `Top57mm` atar. Oyun kodunda topun türüne göre hiçbir `if` gerekmez. Yeni bir tanksavar türü eklemek için yalnız yeni bir alt sınıf ve yeni bir top sınıfı yazılır.

4.4.6.7 `equals` ve kalıtım: simetri tuzağı

`Object`'in metotları da dinamik bağlanır. `Object o = k1; System.out.println(o);` yazıldığında `println`, `o.toString()`'i çağırır ve nesnenin kendi `toString`'i çalışır. `o.equals(x)` de nesnenin kendi `equals`'ını çalıştırır. Bu, koleksiyonların her sınıfla doğru çalışmasını sağlar. `ArrayList.contains`, `HashSet.add` gibi metotlar yalnız `Object` tipini bilir, ama öğelerin kendi `equals` ve `hashCode` metotlarını çağırır.

`equals` metodunun uyması gereken kurallar vardır. Bunlardan biri **simetridir**: `x.equals(y)` doğruysa `y.equals(x)` de doğru olmalıdır. Kalıtım işin içine girince bu kural kolayca bozulur.

```
class Nokta {
    final int x, y;
    Nokta(int x, int y) { this.x = x; this.y = y; }
    @Override public boolean equals(Object o) {
        if (!(o instanceof Nokta)) return false;
        Nokta n = (Nokta) o;
        return x == n.x && y == n.y;
    }
    @Override public int hashCode() { return 31 * x + y; }
}
class RenkliNokta extends Nokta {
    final String renk;
    RenkliNokta(int x, int y, String renk) {
        super(x, y);
        this.renk = renk;
    }
    @Override public boolean equals(Object o) {
        if (!(o instanceof RenkliNokta)) return false;
        RenkliNokta r = (RenkliNokta) o;
        return super.equals(r) && renk.equals(r.renk);
    }
}
```

`Nokta`, iki noktayı koordinatları eşitse eşit sayar. `RenkliNokta` ise buna renk koşulunu ekler ve karşı tarafın da bir `RenkliNokta` olmasını ister. İki sınıf ayrı ayrı bakıldığında makuldür. Birlikte kullanıldıklarında sorun çıkar:

```
Nokta p = new Nokta(1, 2);
RenkliNokta k =
    new RenkliNokta(1, 2, "kırmızı");
System.out.println(p.equals(k)
    + " " + k.equals(p));
```

```
true false
```

Hangi `equals`'ın çalışacağını alıcı nesne belirler. `p.equals(k)` çağrısında alıcı `p`'dir ve `Nokta`'nın metodu çalışır. `k` bir `RenkliNokta`, dolayısıyla bir `Nokta`'dır; koordinatlar eşittir ve sonuç `true`'dur. `k.equals(p)` çağrısında ise alıcı `k`'dir ve `RenkliNokta`'nın metodu çalışır. `p` bir `RenkliNokta` değildir; sonuç `false`'tur. Simetri bozulmuştur.

Bu bozukluk koleksiyonlarda tutarsız davranışa yol açar. `k`'yi içeren bir listede `contains(p)` `true` verir, çünkü liste `p.equals(k)`'yi çağırır. `p`'yi içeren bir listede `contains(k)` ise `false` verir. Bir `HashSet`'e önce `p`, sonra `k` eklenirse kümede 2 öğe olur. Önce `k`, sonra `p` eklenirse 1 öğe olur. Kümenin içeriği, öğelerin eklenme sırasına bağlı hâle gelmiştir.

İşlem	İçeride çağrılan	Sonuç
`[k].contains(p)`	`p.equals(k)`	`true`
`[p].contains(k)`	`k.equals(p)`	`false`
`HashSet` : önce `p` , sonra `k`	`k.equals(p)`	2 öğe
`HashSet` : önce `k` , sonra `p`	`p.equals(k)`	1 öğe

Tablo 4.4.15: Simetri bozukluğunun koleksiyonlardaki sonuçları. `p` = `Nokta(1, 2)` , `k` = `RenkliNokta(1, 2, "kırmızı")` .

Sorunun iki bilinen çözümü vardır. Birincisi, iki sınıfta da `instanceof` yerine `getClass()` karşılaştırması kullanmaktır. Böylece yalnız tam aynı sınıftan nesneler eşit sayılır ve `p.equals(k)` de `false` olur:

```
// Nokta'da:
@Override public boolean equals(Object o) {
    if (o == null || getClass() != o.getClass()) return false;
    Nokta n = (Nokta) o;
    return x == n.x && y == n.y;
}
// RenkliNokta'da:
@Override public boolean equals(Object o) {
    if (!super.equals(o)) return false;   // sınıf ve koordinat
    return renk.equals(((RenkliNokta) o).renk);
}
```

Bu çözümle `p.equals(k)` ve `k.equals(p)` ikisi de `false` verir; iki aynı renkli nokta ise eşit çıkar. Bedeli şudur: `Nokta` 'nın, hiçbir alan eklemeyen sıradan bir alt sınıfı bile artık `Nokta` 'ya eşit sayılmaz.

İkinci çözüm kalıtım yerine oluşumdur (*composition*). `RenkliNokta` , `Nokta` 'dan türemez; bir `Nokta` alanı tutar ve ayrıca bir renk taşır. İki sınıf arasında is-a ilişkisi kalmadığı için birbirleriyle karşılaştırılmaları da anlamını yitirir. Bir renkli noktanın konumu gerekirse `konum()` gibi bir metotla `Nokta` olarak alınır.

```
class RenkliNokta {              // extends yok
    private final Nokta konum;
    private final String renk;
    RenkliNokta(int x, int y, String renk) {
        this.konum = new Nokta(x, y);
        this.renk = renk;
    }
    Nokta konum() { return konum; }
    @Override public boolean equals(Object o) {
        if (!(o instanceof RenkliNokta)) return false;
        RenkliNokta r = (RenkliNokta) o;
        return konum.equals(r.konum) && renk.equals(r.renk);
    }
    @Override public int hashCode() { return 31 * konum.hashCode() + renk.hashCode(); }
}
```

Sık yapılan hata

`equals`'ı alt sınıfta tek yönlü genişletmek. Alt sınıf `equals`'a yeni bir koşul eklerken üst sınıf eski kuralı korursa `p.equals(k)` `true`, `k.equals(p)` `false` olur. Simetri bozulur ve koleksiyonlar tutarsızlaşır.

4.4.7 Çözümlü problemler

Aşağıdaki problemler, notta anlatılan kuralları birlikte kullanmayı gerektirir. Her problemde önce kod verilir, ardından çözüm adım adım açıklanır. Bütün çıktılar JDK 21 ile çalıştırılarak doğrulanmıştır.

Örnek 4.4.6

Problem 1: Çıktı nedir? `cagir()` yalnız `A`'da tanımlıdır ve içinden `yaz()`'ı çağırır. `B` ise yalnız `yaz()`'ı geçersiz kılar.

```
class A {
    void yaz() {
        System.out.println("A.yaz");
    }

    void cagir() {
        yaz(); // this.yaz()
    }
}

class B extends A {
    @Override
    void yaz() {
        System.out.println("B.yaz");
    }
}

// main içinde:
A a = new B();
a.cagir();
new A().cagir();
```

`a.cagir()` : `a` 'nın gösterdiği nesne bir `B` 'dir. `B` 'de `cagir()` yoktur; arama yukarı çıkar ve `A.cagir()` çalışır. Bu metodun içindeki `yaz()` aslında `this.yaz()` 'dır. `this` , metodun çağrıldığı nesnedir, yani bir `B` nesnesi. `this.yaz()` da dinamik bağlanır ve `B.yaz()` çalışır: `B.yaz` .

`new A().cagir()` : nesne bir `A` 'dır; `A.cagir()` içindeki `this.yaz()` , `A.yaz()` 'ı çalıştırır: `A.yaz` .

```
B.yaz
A.yaz
```

Problemin dersi şudur: üst sınıfın kodu, alt sınıfın geçersiz kıldığı metodu farkında olmadan çağırabilir. `A` 'yı yazan programcı `cagir()` 'ın `A.yaz()` 'ı çağıracağını düşünmüş olabilir. Ama `this` üzerinden yapılan her çağrı dinamik bağlanır.

Örnek 4.4.7

Problem 2: Derlenir mi, çalışır mı? Her satır için derleme hatası, çalışma zamanı hatası ya da sorunsuz (çıktısıyla) kararı verilecek.

```
Hayvan h = new Kedi();
Kedi k1 = h;              // (1)
Kedi k2 = (Kedi) h;       // (2)
Kopek k3 = (Kopek) h;     // (3)
h.tirmala();              // (4)
((Kedi) h).tirmala();     // (5)
Object o = new Kopek();
o.sesCikar();             // (6)
((Hayvan) o).sesCikar();  // (7)
```

#	Sonuç	Neden
1	Derleme hatası	`Hayvan` → `Kedi` için açık dönüşüm gerekir: `incompatible types: Hayvan cannot be converted to Kedi`.
2	Sorunsuz	Nesne gerçekten bir `Kedi`'dir.
3	Çalışma zamanı hatası	Derlenir, ama nesne kedidir: `ClassCastException`.
4	Derleme hatası	`Hayvan`'da `tirmala()` yoktur: `cannot find symbol`.
5	Sorunsuz	`Kedi tırmalıyor` yazar.
6	Derleme hatası	`Object`'te `sesCikar()` yoktur.
7	Sorunsuz	Dinamik bağlama: `Hav hav!` yazar.

Çözümün yöntemi her satırda aynıdır. Önce derleyicinin sorusu sorulur: bildirilen tiplere göre atama ya da çağrı geçerli mi? Geçerliyse JVM'in sorusu sorulur: nesneye bakıldığında dönüşüm başarılı mı, hangi sürüm çalışır? 1. ve 4. satırda derleyici `h`'nin `Hayvan` olduğunu bilir, kedi olduğunu bilmez. 7. satırda dönüşüm aşağı doğrudur (Object → Hayvan) ve nesne köpektir; `Kopek.sesCikar()` çalışır.

Örnek 4.4.8

Problem 3: Aşırı yükleme ve geçersiz kılma. `Ust`'te iki aşırı yüklenmiş `f` vardır; `Alt` yalnız `f(Object)`'i geçersiz kılar. İki çağrı ne yazdırır?

```
class Ust {
    void f(Object o) {
        System.out.println("Ust.f(Object)");
    }

    void f(String s) {
        System.out.println("Ust.f(String)");
    }
}

class Alt extends Ust {
    @Override
    void f(Object o) {
        System.out.println("Alt.f(Object)");
    }
}

// main içinde:
Ust u = new Alt();
Object o = "merhaba";
u.f(o);
u.f("merhaba");
```

`u.f(o)` : önce derleyici sürümü seçer. `u` 'nun bildirilen tipi `Ust` 'tür; adaylar `f(Object)` ve `f(String)` 'dir. `o` 'nun bildirilen tipi `Object` 'tir; yalnız `f(Object)` uyar. Gösterdiği nesnenin bir dizgi olması bu seçimi değiştirmez. Çalışırken alıcı nesne bir `Alt` 'tır ve `f(Object)` 'i geçersiz kılmıştır: `Alt.f(Object)` .

`u.f("merhaba")` : argüman `String` 'dir; iki sürüm de uyar, `f(String)` daha özeldir. `Alt` bu sürümü geçersiz kılmadığı için `Ust.f(String)` çalışır.

```
Alt.f(Object)
Ust.f(String)
```

Sıra her zaman aynıdır: önce aşırı yükleme (derleme zamanı, bildirilen tip), sonra geçersiz kılma (çalışma zamanı, nesne tipi).

Örnek 4.4.9

Problem 4: Kurulum, aşama, dizi. Aşağıdaki sınıflar için `new Alt();` ve ardından `new Alt().goster();` ne yazdırır?

```
class Ust {
    Ust() { goster(); }
    void goster() { System.out.println("Ust"); }
}
class Alt extends Ust {
    private int[] dizi = {1, 2, 3};
    @Override void goster() {
        System.out.println(dizi == null
            ? "dizi yok" : "uzunluk " + dizi.length);
    }
}
```

Her `new Alt()` sırasında önce `Ust()` çalışır ve `goster()`'i çağırır. Çağrı dinamik bağlanır; `Alt.goster()` çalışır. O anda `dizi` alanına ilk değeri henüz atanmamıştır ve `null`'dır: `dizi yok`. İkinci satırda yeni bir nesne oluşturulur ve aynı şey yeniden olur: yine `dizi yok`. Ardından kurulmuş nesne üzerinde `goster()` çağrılır; artık `dizi` hazırdır: `uzunluk 3`. Çıktı üç satırdır: `dizi yok`, `dizi yok`, `uzunluk 3`.

Aynı problemin ikinci sorusu: `Object[] d = new String[1]; d[0] = 1;` derlenir mi, çalışır mı? Derlenir: `Object[]` referansına `String[]` atanabilir ve `1`, `Integer`'a kutulanıp `Object` olarak yazılabilir. Çalışırken ise dizi gerçekte `String[]`'dir: `ArrayStoreException: java.lang.Integer`.

Örnek 4.4.10

Problem 5: Maaşlar. `Calisan` sınıfının `maas()` metodu temel maaşı döndürür. `Mudur` bunu temel × 1,5 ile, `Stajyer` sabit 5000 ile geçersiz kılar. `Calisan[]` = {Ayşe 20000, Müdür Mehmet 30000, Stajyer Can} için toplam maaş nedir?

```
class Mudur extends Calisan {
    Mudur(String ad, double temelMaas) { super(ad, temelMaas); }
    @Override double maas() { return temelMaas * 1.5; }
}
class Stajyer extends Calisan {
    Stajyer(String ad) { super(ad, 0); }
    @Override double maas() { return 5000; }
}
// main içinde:
Calisan[] cs = { new Calisan("Ayşe", 20000), new Mudur("Mehmet", 30000),
                 new Stajyer("Can") };
double toplam = 0;
for (Calisan c : cs) toplam += c.maas();
System.out.println(toplam);   // 70000.0
```

Döngü yalnız `Calisan` tipini bilir; her turda `c.maas()` nesnenin kendi sürümünü çalıştırır. Ayşe düz bir çalışandır: 20000. Mehmet müdürdür: 30000 × 1,5 = 45000. Can stajyerdir: 5000. Toplam 20000 + 45000 + 5000 = **70000.0**'dır. Sonuç `double` olduğu için ondalık noktayla yazılır.

Örnek 4.4.11

Problem 6: X, Y, Z. `X`'te `ad()` `"X"` döndürür, `yaz()` `println(ad())` yapar. `Y extends X` yalnız `ad()`'ı `"Y"` döndürecek biçimde geçersiz kılar. `Z extends Y` yalnız `yaz()`'ı `println("Z:" + ad())` olarak geçersiz kılar. `{new X(), new Y(), new Z()}` dizisinde her öğe için `yaz()` çağrılırsa çıktı nedir?

`X`: `X.yaz()` → `X.ad()`: `X`. `Y`: `Y`'de `yaz()` yoktur, `X.yaz()` çalışır; ama içindeki `ad()`, `this` bir `Y` olduğu için `Y.ad()`'dır: `Y`. `Z`: `Z.yaz()` çalışır; `Z`'de `ad()` yoktur, arama `Y`'de biter: `Z:Y`.

```
X
Y
Z:Y
```

Örnek 4.4.12

Problem 7: `instanceof` zincirini kaldırmak. `instanceof` zincirli döngü (`Kopek` → havla, `Kedi` → tırmala, diğer → `"Özel davranış yok"`) hiç `instanceof` kullanmadan nasıl yazılır?

`Hayvan`'a `void ozelDavranis()` metodu eklenir ve `"Özel davranış yok"` yazar. `Kopek` onu `havla()`'yı çağıracak, `Kedi` de `tirmala()`'yı çağıracak biçimde geçersiz kılar. Döngü tek satıra iner:

```
// Hayvan'da:
void ozelDavranis() { System.out.println("Özel davranış yok"); }
// Kopek'te:
@Override void ozelDavranis() { havla(); }
// Kedi'de:
@Override void ozelDavranis() { tirmala(); }
// döngü:
for (Hayvan x : hayvanlar) {
    x.ozelDavranis();
}
```

Çıktı aynı üç satırdır. Yeni bir alt sınıf eklendiğinde döngü değişmez; yeni sınıf isterse `ozelDavranis()` 'i geçersiz kılar, istemezse varsayılan davranışı alır.

Örnek 4.4.13

Problem 8: `Object` referansıyla çağrı. `Object o = new Kopek();` ardından `o.sesCikar();` neden derlenmez? `yazdir(Hayvan)` ve `yazdir(Kopek)` varken `yazdir(o)` derlenir mi?

`Object` 'te `sesCikar()` yoktur: `cannot find symbol … location: variable o of type Object` . Düzeltme: `((Hayvan) o).sesCikar();` . Nesne köpek olduğu için `Hav hav!` yazar.

`yazdir(o)` da derlenmez. `o` 'nun bildirilen tipi `Object` 'tir ve iki sürüm de `Object` kabul etmez: `no suitable method found for yazdir(Object)` . `yazdir((Kopek) o)` ile derlenir ve `yazdir(Kopek)` seçilir.

4.4.8 Sık yapılan hatalar

Çok biçimlilikle ilgili hatalar iki gruba ayrılır. Derleme hataları, derleyicinin yalnız bildirilen tipe bakmasından doğar ve iletileri açıktır. Çalışma zamanı ve mantık hataları ise derleyiciden geçer; ya istisnayla ya da sessizce yanlış sonuçla ortaya çıkar. Aşağıdaki iki tablo, notun gövdesinde ayrı ayrı işlenen hataları bir arada verir. İletiler JDK 21'in javac ve java araçlarıyla alınmıştır.

Kod	javac iletisi	Neden / çözüm
`Kopek k = new Hayvan();`	`incompatible types: Hayvan cannot be converted to Kopek`	Her hayvan köpek değildir; ters atama yapılamaz.
`Kopek k = hayvan;`	`incompatible types: Hayvan cannot be converted to Kopek`	Nesne köpek olsa bile açık dönüşüm gerekir: `(Kopek) hayvan` .
`hayvan.havla();`	`cannot find symbol … variable hayvan of type Hayvan`	Referans tipinde metot yok; önce aşağı dönüşüm gerekir.
`(Kopek) h.havla();`	`not a statement`	Parantez eksik: `((Kopek) h).havla();`
`(Kedi) kopek`	`incompatible types: Kopek cannot be converted to Kedi`	Kardeş sınıflar arasında dönüşüm yoktur.
`@Override void sescikar()`	`method does not override or implement a method from a supertype`	Ad yanlış yazılmıştır (büyük/küçük harf).
`void sesCikar()` (üstte `public`)	`attempting to assign weaker access privileges; was public`	Geçersiz kılan metot erişimi daraltamaz.
`goster(null)` (ilişkisiz iki sürüm)	`reference to goster is ambiguous`	Açık dönüşümle seçilir: `goster((Hayvan) null)` .

Kod	javac iletisi	Neden / çözüm
`List<Hayvan> l = new ArrayList<Kopek>();`	`incompatible types: ArrayList<Kopek> cannot be converted to List<Hayvan>`	Jenerikler kovaryant değildir.

Tablo 4.4.16: Çok biçimlilikle ilgili sık derleme hataları.

Hata	Sonuç	Düzeltme
Denetimsiz aşağı dönüşüm: `(Kopek) h`, oysa `h` bir kedi olabilir	`ClassCastException`	Önce `instanceof` ile sınanır.
"Dönüşüm nesneyi değiştirir" sanmak	`((Hayvan) kopek).sesCikar()` yine `Hav hav!` yazar	Dönüşüm yalnız bakılan tipi değiştirir.
Alanların da geçersiz kılındığını sanmak	`h.tur` referans tipine göre okunur: `hayvan`	Alan `private` yapılır, getter metodu geçersiz kılınır.
Aşırı yüklemeyi geçersiz kılma sanmak: `equals(Kisi k)`	Koleksiyonlar bu metodu çağırmaz; `Object.equals` çalışır	İmza `equals(Object)` olur; `@Override` hatayı yakalar.
`==` ile nesne karşılaştırmak	İçerik değil referans karşılaştırılır	Geçersiz kılınmış `equals` kullanılır.
Her yerde `instanceof` zinciri	Yeni alt sınıf eklendikçe sessizce yanlış sonuç	Davranış geçersiz kılınan bir metoda taşınır.
Yapıcıdan geçersiz kılınabilir metot çağırmak	`null` çıktısı ya da `NullPointerException`	Yapıcıdan yalnız `private`, `final`, `static` metot çağrılır.
Dizi kovaryansına güvenmek: `Hayvan[] d = new Kopek[2]; d[1] = new Kedi();`	`ArrayStoreException: Kedi`	Dizi üst sınıf tipinde oluşturulur.
Kutulamanın önce geldiğini sanmak	`f(5)` → `f(long)`, `f(Integer)` değil	Genişletme 1. aşamadadır.
`equals`'ı alt sınıfta tek yönlü genişletmek	`p.equals(k)` `true`, `k.equals(p)` `false`	`getClass()` karşılaştırması ya da oluşum.

Tablo 4.4.17: Çok biçimlilikle ilgili sık çalışma zamanı ve mantık hataları.

Tablodaki `equals(Kisi k)` hatası ayrıca açıklanmalıdır. `Object`'teki metodun imzası `equals(Object)`'tir. Alt sınıfta `equals(Kisi k)` yazmak onu geçersiz kılmaz; aynı adla yeni bir aşırı yükleme ekler. `Kisi` tipinde iki değişkenle yapılan `a.equals(b)` çağrısında yeni metot seçilir ve doğru çalışıyor gibi görünür. Ama `ArrayList.contains` ya da `HashSet`, argümanı `Object` tipinde verir. Derleyici o kodda yalnız `equals(Object)`'i görür; `Kisi` onu geçersiz kılmadığı için `Object`'in referans karşılaştırması çalışır. Metodun başına `@Override` yazılsaydı derleyici `method does not override or implement a method from a supertype` der ve hata baştan yakalanırdı.

4.4.9 Özet

- Çok biçimlilik, aynı ifadenin farklı nesnelerde farklı davranışa yol açmasıdır. Kod üst sınıf tipiyle bir kez yazılır; her nesne çağrıya kendi sürümüyle cevap verir.
- Derleme zamanı çok biçimliliği aşırı yüklemeyle, çalışma zamanı çok biçimliliği geçersiz kılma ve dinamik bağlamayla sağlanır. İlkini derleyici, ikincisini JVM seçer.

- Her referans değişkeninin bir bildirilen tipi, bir de gösterdiği nesnenin gerçek tipi vardır. Hangi metotların çağrılabileceğine ve hangi atamaların geçerli olduğuna bildirilen tip; hangi sürümün çalışacağına gerçek tip karar verir.
- Referans kuralı: üst sınıf referansı alt sınıf nesnesini gösterebilir; tersi derlenmez. `Hayvan h = new Kopek();` geçerli, `Kopek k = new Hayvan();` hatalıdır.
- Yukarı dönüşüm otomatik ve her zaman güvenlidir. Aşağı dönüşüm açıkça yazılır; nesne uygun değilse çalışma zamanında `ClassCastException` fırlatılır. Kardeş sınıflar arasındaki dönüşüm hiç derlenmez.
- Nokta işleci dönüşümden önce uygulanır: doğru yazım `((Kopek) h).havla()`'dır.
- `instanceof`, dönüşümden önce nesnenin tipini sınar. Java 16'dan beri örüntü eşleme sınama ile dönüşümü birleştirir: `if (x instanceof Kopek kk)`.
- Diziler kovaryanttır; yanlış tipte öğe yazmak `ArrayStoreException` verir. Jenerikler kovaryant değildir ve aynı hatayı derleme zamanında engeller.
- Metot araması nesnenin gerçek sınıfından başlar ve yukarı çıkar. JVM bunu sınıf başına tutulan metot tablolarıyla, tek bir dizi erişimiyle yapar; bayt kodunda yalnız referans tipinin metodu yazılıdır (`invokevirtual`).
- Alanlar, `static`, `private` ve `final` metotlar dinamik bağlanmaz. Alt sınıftaki aynı adlı alan ya da `static` metot üst sınıftakini gizler, geçersiz kılmaz.
- Yapıcıdan çağrılan geçersiz kılınabilir metot, alt sınıfın alanları atanmadan çalışır. Yapıcıdan yalnız `private`, `final` ya da `static` metotlar çağrılır.
- Aşırı yükleme seçimi argümanların bildirilen tipine göre derleme zamanında yapılır. Birden çok sürüm uyarsa en özel olan seçilir; tek bir en özel sürüm yoksa çağrı belirsizdir.
- Aşırı yükleme üç aşamada çözülür: kutulamasız (genişletme), kutulama, değişken sayıda parametre. Bir aşamada aday bulunursa sonrakilere geçilmez; daraltma hiç denenmez.
- Çok biçimli kod genişletmeye açık, değiştirmeye kapalıdır: yeni bir alt sınıf eklemek için var olan döngüler ve metotlar değişmez. Parametre ve değişken tipleri işi görmeye yeten en genel tip olarak seçilir.
- `equals` alt sınıfta tek yönlü genişletilirse simetri bozulur. Çözüm `getClass()` karşılaştırması ya da kalıtım yerine oluşumdur.

4.4.10 Kavram listesi

çok biçimlilik *polymorphism* · derleme zamanı / çalışma zamanı *compile time / run time* · bildirilen tip / gerçek tip *declared type / actual type* · referans *reference* · üst sınıf / alt sınıf *superclass / subclass* · is-a / has-a ilişkisi *is-a / has-a relationship* · Object sınıfı *Object class* · yukarı dönüşüm / aşağı dönüşüm *upcasting / downcasting* · açık dönüşüm *cast (explicit conversion)* · genişletme / daraltma dönüşümü *widening / narrowing conversion* · dinamik bağlama *dynamic binding*

metot tablosu *virtual method table* · geçersiz kılma *method overriding* · aşırı yükleme *method overloading* · metot imzası *method signature* · otomatik kutulama / kutudan çıkarma *autoboxing / unboxing* · yapıcı *constructor* · yığın izi *stack trace* · jenerikler *generics* · fabrika metodu *factory method* · oluşum *composition* · soyut metot *abstract method* · gizleme *hiding* · dizi kovaryansı *array covariance* · örüntü eşleme *pattern matching* · açık/kapalı ilkesi *open/closed principle*

4.4.11 Kendini Dene

Sorularda örnek hiyerarşi kullanılır: `Kopek`, `sesCikar()` ve `kos()`'u; `Kedi` yalnız `sesCikar()`'ı geçersiz kılar.

1. `Hayvan h = new Kedi(); h.kos(); h.sesCikar();` ne yazdırır?

   Cevap

   `Hayvan koşuyor` ve `Miyav!`. `kos()` için arama `Kedi`'de başlar; `Kedi` bu metodu geçersiz kılmadığı için `Hayvan`'da biter. `sesCikar()` ise `Kedi`'de bulunur.

2. `Object o = new Kedi();` sonrasında `Hayvan h = (Hayvan) o;`, `Kedi c = (Kedi) o;`, `Kopek k = (Kopek) o;` ve `Kedi c2 = o;` satırlarının her biri için sonuç nedir?

   Cevap

   İlk üç satır derlenir; `Object`'ten her sınıfa aşağı dönüşüm yazılabilir. Çalışırken ilk ikisi başarılıdır, çünkü nesne bir kedidir ve kedi bir hayvandır. Üçüncüsü `ClassCastException` verir. Dördüncü satır hiç derlenmez: açık dönüşüm yoktur (`incompatible types: Object cannot be converted to Kedi`).

3. `class U { int x = 1; int getX() { return x; } }` ve `class V extends U { int x = 2; int getX() { return x; } }` verilsin. `U u = new V();` için `u.x + " " + u.getX() + " " + ((V) u).x` ne verir?

   Cevap

   `1 2 2`. `u.x` alan erişimidir ve bildirilen tipe (`U`) göre okunur: 1. `u.getX()` dinamik bağlanır; `V.getX()` içindeki `x`, `V`'nin alanıdır: 2. Dönüşümden sonra bildirilen tip `V`'dir: 2.

4. `static void g(Object o)` `"O"`, `static void g(String s)` `"S"` yazdırsın. `Object a = "x";`

için `g(a)`, `g((String) a)` ve `g(null)` ne yazdırır?

Cevap

`O`, `S`, `S`. `a`'nın bildirilen tipi `Object`'tir; nesnenin dizgi olması önemli değildir. Dönüşümle tip `String` olur. `null` iki sürüme de uyar; ama `String`, `Object`'in alt tipidir ve iki aday tek bir zincir üzerindedir. Daha özel olan `g(String)` seçilir; belirsizlik yoktur.

5. `static void h(double d)` ve `static void h(Integer i)` varken `h(5)`, `h(Integer.valueOf(5))` ve `h(5L)` hangi sürümü seçer?

Cevap

`h(5)`: 1. aşamada `int` → `double` genişletmesi uyar; `Integer` kutulama ister ve denenmez: `double`. `h(Integer.valueOf(5))`: 1. aşamada tam eşleşme `h(Integer)`'dır; `double`'a ulaşmak kutudan çıkarma ister: `Integer`. `h(5L)`: `long` → `double` genişler: `double`.

6. `class A { A() { m(); } void m() { System.out.println("A"); } }` ve `class B extends A { int n = 7; void m() { System.out.println("B " + n); } }` verilsin. `new B();` ne yazdırır?

Cevap

`B 0`. `A()` içindeki `m()` çağrısı dinamik bağlanır ve nesne bir `B` olduğu için `B.m()` çalışır. O anda `n`'ye ilk değeri henüz atanmamıştır; `int`'in varsayılan değeri 0'dır. `n = 7`, `A()` döndükten sonra çalışır.

7. `Object[] d = new Integer[2]; d[0] = 5; d[1] = "a";` satırlarından hangisi sorun çıkarır?

Cevap

Üç satır da derlenir. `d[0] = 5` çalışır: 5, `Integer`'a kutulanır ve dizinin gerçek öğe tipine uyar. `d[1] = "a"` ise çalışırken `ArrayStoreException: java.lang.String` verir; dizi gerçekte `Integer[]`'dir.

8. `class P { String ad() { return "P"; } String yaz() { return ad(); } }` ve `class Q extends P { String ad() { return "Q"; } String yaz() { return "[" + super.yaz() + "]"; } }` verilsin. `P p = new Q();` için `p.yaz()` ne döndürür?

Cevap

`[Q]`. `p.yaz()` dinamik bağlanır: `Q.yaz()`. `super.yaz()`, `P.yaz()`'ı doğrudan çağırır (`invokespecial`). Onun içindeki `ad()` ise `this.ad()`'dır ve `this` hâlâ bir `Q` nesnesidir: `Q.ad()` → `"Q"`. `super` yalnız o tek çağrının hedefini sabitler; içerideki çağrılar yine dinamik bağlanır.

4.4.12 İleri okuma

- Oracle, *The Java Tutorials*, “Polymorphism” ve “Overriding and Hiding Methods” sayfaları: dinamik bağlamayı ve alan ile `static` metot gizlemeyi kısa örneklerle anlatır. Java'nın resmî ve ücretsiz başlangıç kaynağıdır.
- J. Bloch, *Effective Java* (3. baskı), Madde 10 “Obey the general contract when overriding equals” ve Madde 19 “Design and document for inheritance or else prohibit it”: `equals` simetri tuzağını ve yapıcıdan geçersiz kılınabilir metot çağırmanın tehlikesini ayrıntılı tartışır.
- P. Deitel ve H. Deitel, *Java: How to Program* (11. baskı), Bölüm 10 “Object-Oriented Programming: Polymorphism and Interfaces”: bordro örneğiyle çok biçimliliği adım adım kurar; soyut sınıflara ve arayüzlere de geçer.
- J. Gosling ve diğerleri, *The Java Language Specification, Java SE 21 Edition*, 5.5 “Casting Contexts” ve 15.12 “Method Invocation Expressions”: dönüşümlerin ve aşırı yükleme seçiminin üç aşamasının kesin tanımıdır.

4.4.13 Kaynaklar

1. Oracle, *The Java Tutorials*: “Polymorphism”; “Overriding and Hiding Methods”; “Inheritance”.
2. J. Gosling ve diğerleri, *The Java Language Specification, Java SE 21 Edition*, Oracle, 2023 (5.5 Casting Contexts; 8.4.8 Inheritance, Overriding, and Hiding; 15.12 Method Invocation Expressions).
3. P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (Bölüm 10: Object-Oriented Programming: Polymorphism and Interfaces).
4. K. Sierra ve B. Bates, *Head First Java*, 3. baskı, O'Reilly, 2022 (Inheritance and Polymorphism).
5. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson, 2022 (Bölüm 5: Inheritance).
6. A. B. Downey ve C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019.
7. J. Bloch, *Effective Java*, 3. baskı, Addison-Wesley, 2018.

KISIM II · BÖLÜM 4.5

Soyut Sınıflar ve Arayüzler

Kalıtım ortak kodu üst sınıfta toplar, çok biçimlilik de aynı çağrının her nesnede kendi sınıfının metodunu çalıştırmasını sağlar. Bu iki aracın bir eksiği kalır: üst sınıf bazen bir metodun gövdesini yazamaz. Genel bir "şekil"in alanı hesaplanamaz; formülü yalnızca kare, çember ya da üçgen bilir. Java bu durum için iki araç sunar: **soyut sınıf** ve **arayüz**. İkisi de "ne yapılacağını" söyler, "nasıl yapılacağını" alt tiplere bırakır. Not önce soyut sınıfı ve soyut metodu, ardından arayüzü, arayüzün gövdeli metotlarını ve mühürlü tipleri işler. Tek metotlu arayüzlerin kısa yazımı olan lambda ifadeleri, tip parametreli sınıflar (jenerikler) ve koda üst veri ekleyen etiketler de notun konuları arasındadır. Son kesim, çoğu arayüzlere dayanan beş tasarım desenini (Tekil, Fabrika, Gözlemci, Strateji ve MVC) küçük konsol programlarıyla anlatır. Bütün programlar JDK 21 ile derlenip çalıştırılmıştır; çıktılar, izleme tablolarındaki son değerler ve derleyici iletileri bu çalıştırmalardan alınmıştır.

ÖĞRENME ÇIKTILARI

Soyut sınıf
`abstract` sınıf ve metot tanımlamak; soyut sınıftan neden nesne oluşturulamadığını ve somut alt sınıfın hangi metotları yazmak zorunda olduğunu açıklamak.

Şablon metot
Soyut sınıfın somut bir metodunun, alt sınıfta yazılan soyut metodu çağırdığını kodu adım adım izleyerek göstermek.

Arayüz
`interface` tanımlamak; `implements` ile bir ya da birden çok arayüzü uygulamak; `default`, `static` ve `private` arayüz metotlarını kullanmak.

Lambda
Fonksiyonel arayüzü tanımak; tek metotluk bir davranışı lambda ifadesiyle yazmak ve lambdanın ne zaman çalıştığını izlemek.

Tasarım kararı
Mühürlü tipleri tanımak; bir problem için arayüz mü soyut sınıf mı (ya da ikisi birden) gerektiğine karar vermek; jenerik tipleri ve etiketleri kullanmak.

Tasarım desenleri
Tekil, Fabrika, Gözlemci, Strateji ve MVC desenlerinin hangi sorunu çözdüğünü ve arayüze nasıl dayandığını açıklamak.

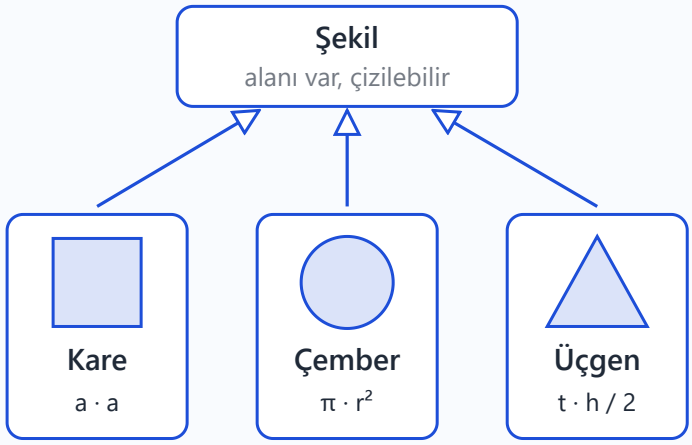
4.5.1 Soyutlama

Araba kullanan biri direksiyonu, vitesi ve pedalları bilir. Motorun içinde yakıtın nasıl yandığını, şanzımanın dişlilerini nasıl değiştirdiğini bilmek zorunda değildir. Arabanın sürücüye sunduğu yüz basittir; karmaşık ayrıntı bu yüzün arkasında kalır. Üstelik aynı yüz farklı arabalarda da aynıdır. Benzinli bir arabayı kullanabilen biri elektrikli bir arabayı da kullanabilir, çünkü ikisi de direksiyon ve pedal sözleşmesine uyar.

Tanım 4.5.1

Soyutlama (*abstraction*), karmaşık ayrıntıları gizleyip yalnızca temel özellikleri göstermektir. Programlamada soyutlama, bir işin "ne" yaptığını onu "nasıl" yaptığından ayırır.

Programda da aynı ayrım işe yarar. Bir çizim programı ekrandaki şekillerin toplam alanını hesaplamak istesin. Her şeklin bir alanı vardır; bu, "şekil" kavramının temel özelliğidir. Ama genel bir şeklin alanı hesaplanamaz. Kare için formül $a \cdot a$, çember için $\pi \cdot r^2$, üçgen için $t \cdot h \,/\, 2$'dir. Formülü yalnızca somut şekil bilir. Şekil 4.5.1 bu durumu çizer: yukarıda ortak kavram, aşağıda ayrıntı durur.



Şekil 4.5.1: "Şekil" bir kavramdır: alanı vardır ama hesaplanamaz. Kare, Çember ve Üçgen kavramı somutlaştırır ve her biri alanı kendi formülüyle hesaplar.

Toplam alanı hesaplayan kod yalnızca "her şeklin bir `alan()` metodu vardır" bilgisine dayanmalıdır. Bu kod kare mi çember mi olduğunu sormaz, her şekilden alanını ister. Yarın programa altıgen eklenirse toplama kodu değişmez; yalnızca yeni sınıf yazılır. Soyutlamanın kazancı budur: kullanan kod

ayrıntıya değil kavrama bağlanır.

Java'da soyutlamanın iki aracı vardır: **soyut sınıflar** (`abstract class`) ve **arayüzler** (`interface`). İkisi de ne yapılacağını söyler, nasıl yapılacağını alt sınıflara bırakır. Aralarındaki fark kısaca şudur. Soyut sınıf yakın akraba sınıfların ortak atasıdır; alanları, yapıcısı ve hazır metotları olabilir. Arayüz ise birbiriyle akraba olmayan sınıflara ortak bir yetenek kazandıran sözleşmedir.

4.5.2 Soyut sınıflar

Kalıtımda üst sınıf, alt sınıfların ortak alanlarını ve metotlarını taşır. `Sekil` sınıfı bütün şekillerin ortak `isim` alanını ve `tasi` metodunu yazabilir. `alan()` metodunda ise takılır. Gövdeye ne yazılacaktır? `return 0;` yazmak yanlıştır. Hiçbir şeklin alanı gerçekten 0 değildir. Bir alt sınıf metodu geçersiz kılmayı unutursa program sessizce yanlış sonuç üretir. Doğru çözüm, gövdeyi hiç yazmamak ve bunu derleyiciye açıkça söylemektir.

4.5.2.1 Soyut sınıf tanımlamak

Tanım 4.5.2

Soyut sınıf (*abstract class*), `abstract` anahtar sözcüğüyle tanımlanan ve nesnesi oluşturulamayan sınıftır. Soyut sınıf hem **soyut metotlar** (gövdesiz) hem de **somut metotlar** (*concrete*, gövdeli) içerebilir. Soyut metotların hepsini yazan alt sınıfa **somut sınıf** denir; nesnesi ancak böyle bir sınıftan oluşturulur.

```
public abstract class Sekil {
    // Soyut olmayan (somut) metot
    void tasi(int x, int y) {
        System.out.println("(" + x + ", " + y + ") noktasına taşındı");
    }
    // Soyut metot (alt sınıflarca uygulanmalı)
    abstract void ciz();
}
```

Bu `Sekil` sınıfında iki metot vardır. `tasi` somuttur: gövdesi hazırdır. Alt sınıflar onu olduğu gibi kalıtımla alır, isterse geçersiz kılar (*override*). Şekli bir noktaya taşımak her şekil için aynı işi gerektirdiği için gövdeyi üst sınıfta bir kez yazmak doğrudur. `ciz` ise soyuttur: yalnızca imzası vardır. Bir kareyi çizmekle bir çemberi çizmek farklı işlerdir; bu yüzden her somut alt sınıf kendi `ciz()` gövdesini yazmak zorundadır.

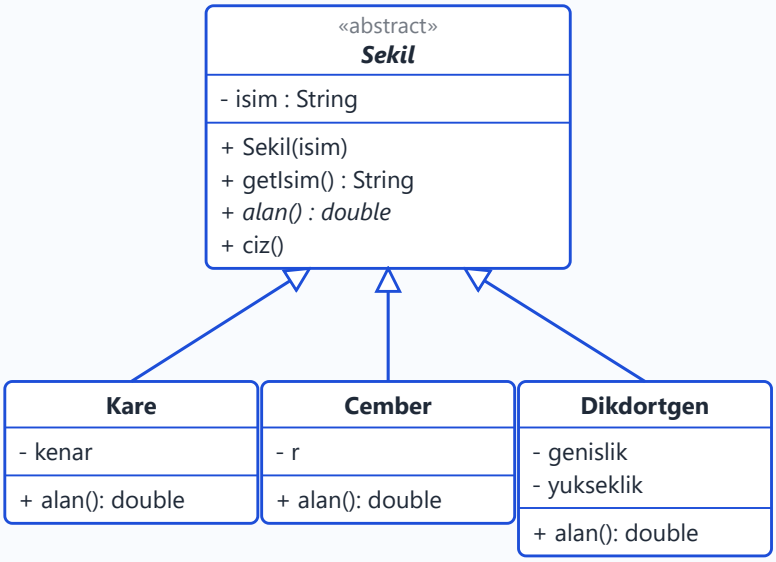
Soyut sınıf yarım kalmış bir plana benzer. Bazı odalar bitmiştir, bazılarının ise yalnızca yeri bellidir. Planı devralan mimar bitmiş odalara dokunmadan boş yerleri doldurur. Aşağıdaki küçük program sınıfı bir `Kare` ile tamamlar ve iki metodu da çağırır:

```
abstract class Sekil {
    // Soyut olmayan (somut) metot
    void tasi(int x, int y) {
        System.out.println("(" + x + ", " + y + ") noktasına taşındı");
    }
    // Soyut metot (alt sınıflarca uygulanmalı)
    abstract void ciz();
}
class Kare extends Sekil {
    void ciz() { System.out.println("Kare çiziliyor"); }
}
public class D03 {
    public static void main(String[] args) {
        Sekil s = new Kare();
        s.tasi(3, 4);
        s.ciz();
    }
}
```

```
(3, 4) noktasına taşındı
Kare çiziliyor
```

`Kare` yalnızca `ciz()`'i yazmıştır; `tasi`'yi `Sekil`'den alır. Değişkenin tipi soyut `Sekil`'dir ama gösterdiği nesne bir `Kare`'dir. `s.ciz()` çağrısında hangi gövdenin çalışacağı, çok biçimlilikte olduğu gibi, çalışma zamanında nesnenin sınıfına göre seçilir. Bu karara **dinamik bağlama** (*dynamic binding*) denir.

Soyut sınıfın öteki özellikleri sıradan bir sınıfınkiyle aynıdır. Alanları, yapıcıları, `static` üyeleri olabilir. Başka sınıflar onu `extends` ile genişletir. Şekil 4.5.2 dört metotlu bir `Sekil` sınıfını ve onu genişleten üç somut sınıfı çizer. Çizim UML (*Unified Modeling Language*, birleşik modelleme dili) gösterimindedir.



Şekil 4.5.2: UML'de soyut sınıfın adı ve soyut metotları eğik yazılır. `Sekil`'in `isim` alanı, yapıcısı ve `getIsim()` metodu somuttur; `alan()` ve `ciz()` soyuttur. `Kare`, `Cember` ve `Dikdortgen` kendi alanlarını taşır ve `alan()`'ı kendi formülleriyle yazar.

Şekildeki tasarımın getirdiği kazanç, ortak durumun (alanların) ve davranışın tek bir yerde toplanmasıdır. `isim` alanı ve onu okuyan `getIsim()` üç sınıfta ayrı ayrı yazılmaz. Alt sınıflar soyut metotları kendi ihtiyacına göre doldurur. Soyut sınıflar bu yüzden çoğu zaman kalıtım hiyerarşisinin tepesinde yer alır.

4.5.2.2 Soyut sınıftan nesne oluşturulamaz

Soyut sınıfın en belirgin kuralı, ondan `new` ile nesne oluşturulamamasıdır. Aşağıdaki üç satırdan ilki derlenmez, öteki ikisi derlenir:

```
Sekil s1 = new Sekil("?");     // HATA
Sekil s2 = new Kare(3);        // Doğru: değişkenin türü soyut olabilir
Sekil[] dizi = new Sekil[3];   // Doğru: dizi yalnızca referans tutar
```

```
SoyutNew.java:8: error: Sekil is abstract; cannot be instantiated
        Sekil s1 = new Sekil("?");
                   ^
1 error
```

Yasak keyfî değildir. `Sekil`'in `alan()` metodunun gövdesi yoktur. `new Sekil("?")` izinli olsaydı `s1.alan()` çağrısında çalıştırılacak kod bulunmazdı. Derleyici bu durumu baştan engeller: gövdesi eksik bir nesne hiç oluşmaz.

Soyut sınıfın adı ise referans tipi olarak serbestçe kullanılır. Değişken, parametre, dizi öğesi ve dönüş tipi `Sekil` olabilir. İkinci satırdaki `Sekil s2 = new Kare(3);` tam da soyutlamanın istediği yazımdır: değişkenin tipi kavram, nesnenin sınıfı ayrıntıdır. Üçüncü satır öğrencileri sık şaşırtır. `new Sekil[3]` bir `Sekil` nesnesi oluşturmaz; üç gözlü bir dizi oluşturur ve her göz `null` başlar. Dizi nesnesi referans tutar, soyut sınıfın nesnesini değil. Satırlar çalıştırılınca bu açıkça görülür:

```
abstract class Sekil {
    private String isim;
    Sekil(String isim) { this.isim = isim; }
    abstract double alan();
}
class Kare extends Sekil {
    private double kenar;
    Kare(double kenar) { super("Kare"); this.kenar = kenar; }
    double alan() { return kenar * kenar; }
}
public class D05 {
    public static void main(String[] args) {
        Sekil s2 = new Kare(3);        // Doğru: değişkenin türü soyut olabilir
        Sekil[] dizi = new Sekil[3];   // Doğru: dizi yalnızca referans tutar
        System.out.println(s2.alan());
        System.out.println(dizi.length + " " + dizi[0]);
    }
}
```

```
9.0
3 null
```

Bir sınıf hem `abstract` hem `final` olamaz. `final` sınıfın genişletilmesini yasaklar; `abstract` ise genişletilmeyi gerektirir, çünkü nesne ancak bir alt sınıftan oluşturulabilir. İki belirteç birbiriyle çelişir:

```
abstract final class Sekil { }
public class SoyutFinal { }
```

```
SoyutFinal.java:1: error: illegal combination of modifiers: abstract and final
abstract final class Sekil { }
                ^
1 error
```

4.5.2.3 Soyut metotlar ve kuralları

Tanım 4.5.3

Soyut metot (*abstract method*), gövdesi olmadan, yalnızca imzasıyla tanımlanan metottur. Süslü parantez içermez, noktalı virgülle biter. Somut her alt sınıf onu uygulamak, yani gövdesini yazmak zorundadır.

```
public abstract void ciz();
```

Soyut metodu olan sınıfın kendisi de soyut olmak zorundadır. Aşağıdaki iki dosyada `Sekil` soyut `ciz()` metodunu bildirir, `Cember` ise onu uygular:

```
public abstract class Sekil {
    // Soyut metot
    public abstract void ciz();
}

// Ayrı bir dosyada:
public class Cember extends Sekil {
    // ciz metodunu uygulamak zorunlu
    public void ciz() {
        System.out.println("Çember çiziliyor");
    }
}
```

```
Çember çiziliyor
```

`new Cember().ciz();` çağrısı `Cember`'in yazdığı gövdeyi çalıştırır. Soyut metodun uygulanması, sıradan bir metodun geçersiz kılınmasıyla aynı kurallara uyar. İmza (ad ve parametre listesi) birebir aynı olmalı, erişim daraltılmamalıdır. Üst sınıftaki metot `public` olduğu için `Cember`'deki de `public`'tir.

Soyut metotlarla ilgili kurallar kısa bir tabloya sığar. Tablo 4.5.1 doğru ve yanlış yazımları gerçek derleyici iletileriyle birlikte verir.

Yazılan	Sonuç
`abstract void ciz();` (soyut sınıfta)	Doğru
`abstract void ciz() { }`	`abstract methods cannot have a body`
`class Sekil { abstract void ciz(); }`	`Sekil is not abstract and does not override abstract method ciz() in Sekil`

Yazılan	Sonuç
`class Cember extends Sekil { }` (`ciz` yok)	`Cember is not abstract and does not override abstract method ciz() in Sekil`
`private abstract`, `static abstract`, `final abstract`	`illegal combination of modifiers: abstract and private` (`static` ve `final` için de aynı ileti)
`abstract class Cember extends Sekil { }`	Doğru: soyut alt sınıf gerçeklemeyi kendi alt sınıflarına bırakabilir
`abstract class A { void f() { } }`	Doğru: hiç soyut metodu olmayan soyut sınıf da yazılabilir; yine de `new A()` yapılamaz

Tablo 4.5.1: Soyut metot kuralları ve javac iletileri.

Tablonun üçüncü satırındaki ileti ilk bakışta tuhaf görünür: "`Sekil` soyut değil ve `Sekil`'deki soyut `ciz()` metodunu geçersiz kılmıyor." Derleyici, soyut metodu olan her sınıfa iki seçenek sunar: ya sınıf soyut olur ya da metodu yazar. Kendi bildirdiği metodu kendisinin yazmaması, aynı iletiyi doğurur. Dördüncü satır aynı kuralın alt sınıftaki karşılığıdır.

Beşinci satırdaki üç belirteç neden yasaktır? Üçü de metodun alt sınıfta geçersiz kılınmasını engeller. `private` metot alt sınıfa görünmez. `static` metot nesneye değil sınıfa bağlıdır ve dinamik bağlamaya girmez. `final` metot ise geçersiz kılınamaz. Soyut metot ise yalnızca geçersiz kılınmak için vardır. İletinin `static` sürümü şöyledir:

```
abstract class Sekil {
    static abstract void ciz();
}
public class StaticSoyut { }
```

```
StaticSoyut.java:2: error: illegal combination of modifiers: abstract and static
    static abstract void ciz();
                         ^
1 error
```

Son iki satır izin verilen ama az bilinen iki durumu gösterir. Soyut bir sınıfın alt sınıfı da soyut olabilir; o zaman metodu yazma yükümlülüğü bir alt kuşağa geçer. Hiç soyut metodu olmayan bir sınıf da `abstract` işaretlenebilir. Bu, "bu sınıftan doğrudan nesne oluşturulmasın, yalnızca genişletilsin" demenin yoludur. Aşağıdaki programda her iki durum da derlenip çalışır:

```
abstract class Sekil { abstract void ciz(); }
abstract class Cember extends Sekil { }
abstract class A { void f() { System.out.println("f"); } }
class Daire extends Cember { void ciz() { System.out.println("Daire çiziliyor"); } }
public class Main {
    public static void main(String[] args) {
        Sekil s = new Daire();
        s.ciz();
        A a = new A() { };   // adsız somut alt sınıf
        a.f();
    }
}
```

```
Daire çiziliyor
f
```

`Cember` soyut kaldığı için `ciz()`'i yazmamıştır; onu `Cember`'i genişleten somut `Daire` yazar. `new A() { }` ifadesi ise `A`'nın nesnesini değil, `A`'yı genişleten adsız bir alt sınıfın nesnesini oluşturur. Bu yazıma anonim sınıf denir.

Sık yapılan hata

Soyut metodun imzasını yanlış yazmak. `Sekil`'deki soyut metot `double alan()` iken alt sınıfta `double Alan()` ya da `double alan(int x)` yazılırsa bu yeni bir metottur. Soyut `alan()` hâlâ eksiktir ve derleyici sınıfı reddeder:

```
ImzaYanlis.java:2: error: Kare is not abstract and does not override abstract
method alan() in Sekil
class Kare extends Sekil {
^
1 error
```

İleti hatanın yerini değil, sonucunu gösterir; yazım hatası gözden kaçabilir. Metodun üstüne `@Override` yazılırsa derleyici asıl hatayı doğrudan söyler: `method does not override or implement a method from a supertype`.

4.5.2.4 Soyut sınıfın yapıcısı

Soyut sınıftan nesne oluşturulamıyorsa yapıcısı neye yarar? Cevap, alt sınıf nesnesinin kuruluşundadır. Bir `Kare` nesnesi oluşturulurken nesnenin `Sekil`'den gelen kısmı da kurulmalıdır: `isim` alanına değer verilmelidir. Bu işi `Sekil`'in yapıcısı yapar ve alt sınıf onu `super(...)` ile çağırır. Aşağıdaki programda `Sekil` yapıcısı bir de ileti yazdırır; böylece çağrının ne zaman yapıldığı çıktıda görülür.

```
abstract class Sekil {
    private String isim;
    Sekil(String isim) {
        this.isim = isim;
        System.out.println(isim + " kuruldu");
    }
    abstract double alan();
}
class Kare extends Sekil {
    private double kenar;
    Kare(double kenar) {
        super("Kare");
        this.kenar = kenar;
    }
    double alan() { return kenar * kenar; }
}
// main içinde:
Sekil s = new Kare(3);
System.out.println(s.alan());
```

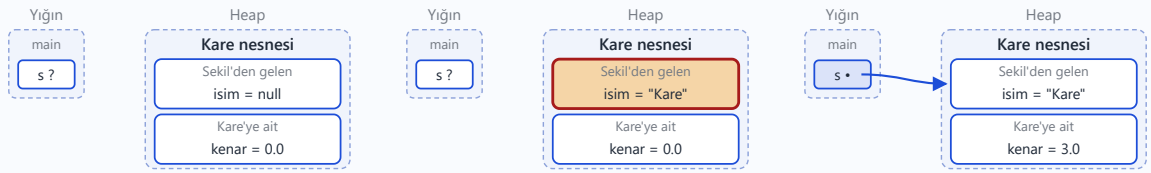
```
Kare kuruldu
9.0
```

Tablo 4.5.2 programın on adımını verir. Satır numaraları yukarıdaki kodun satırlarıdır. “Çağrı yığını” sütunu o anda yığında duran çerçeveleri soldan sağa, en alttakinden en üsttekine doğru sıralar.

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	18: `Sekil s = new Kare(3);`	main()	s=—	—
2	11: `Kare(double kenar) {`	main() → Kare(3)	kenar=3.0	—
3	12: `super("Kare");`	main() → Kare(3)	kenar=3.0	—
4	3: `Sekil(String isim) {`	main() → Kare(3) → Sekil("Kare")	isim="Kare"	—
5	4: `this.isim = isim;`	main() → Kare(3) → Sekil("Kare")	isim="Kare"	—
6	5: `System.out.println(isim + " kuruldu");`	main() → Kare(3) → Sekil("Kare")	isim="Kare"	`Kare kuruldu`
7	13: `this.kenar = kenar;`	main() → Kare(3)	kenar=3.0	—
8	18: `Sekil s = new Kare(3);`	main()	s=→ Kare	—
9	15: `double alan() { return kenar * kenar; }`	main() → Kare.alan()	kenar=3.0	—
10	19: `System.out.println(s.alan());`	main()	s=→ Kare	`9.0`

Tablo 4.5.2: `new Kare(3)` ifadesinin ve `s.alan()` çağrısının izlenmesi.

Birinci adımda `new Kare(3)` heap'te bir `Kare` nesnesi için yer ayırır. Nesnenin iki alanı vardır: `Sekil`'den gelen `isim` ve `Kare`'ye ait `kenar`. İkisi de varsayılan değerini alır: `null` ve `0.0`. İkinci adımda `Kare`'nin yapıcısı çalışmaya başlar. Argüman `int` tipindeki 3'tür, parametre ise `double`'dır; değer otomatik olarak 3.0'a genişletilir.



Şekil 4.5.3: Üç an. Solda `new` nesneyi ayırmış, alanlar varsayılan değerde. Ortada `Sekil` yapıcısı nesnenin `Sekil`'den gelen kısmını kurmuş: `isim = "Kare"`. Sağda `Kare` yapıcısı da bitmiş ve referans `s`'ye atanmış.

Üçüncü adımda `Kare` yapıcısının ilk satırı `super("Kare")` çalışır ve üst sınıfın yapıcısı çağrılır. Dördüncü adımda yığına `Sekil("Kare")` çerçevesi eklenir. Burada çok önemli bir ayrıntı vardır: ayrı bir `Sekil` nesnesi **oluşmaz**. Heap'te hâlâ tek bir nesne vardır. `Sekil` yapıcısı aynı nesnenin `Sekil`'den gelen kısmını kurar. Beşinci adımda `this.isim = isim` bu kısmın alanını "Kare" yapar, altıncı adımda ileti yazdırılır ve `Sekil`'in çerçevesi yığından çıkar.

Yedinci adımda denetim `Kare`'nin yapıcısına döner ve `this.kenar = kenar` çalışır. Nesne artık tamamen kurulmuştur. Sekizinci adımda `new` ifadesinin değeri olan referans `s`'ye atanır. Değişkenin tipi soyut `Sekil`'dir, gösterdiği nesne ise bir `Kare`'dir. Şekil 4.5.3'ün sağındaki çizim bu anı gösterir.

Dokuzuncu adım soyutlamanın asıl işini yapar. `s.alan()` derlenirken derleyici yalnızca `Sekil`'e bakar: `Sekil`'de `alan()` adlı, `double` döndüren bir metot vardır, bu yüzden çağrı geçerlidir. Çalışma zamanında ise JVM nesnenin gerçek sınıfına bakar ve `Kare.alan()`'ı çalıştırır: $3.0 \cdot 3.0 = 9.0$. Onuncu adımda bu değer yazdırılır.

Yapıcı zincirinin sırası kalıtımdaki sırayla aynıdır: önce üst sınıfın yapıcısı, sonra alt sınıfınki. Soyut sınıf bu sırayı değiştirmez; yalnızca zincirin en üst halkasından doğrudan nesne oluşturulmasını yasaklar. `Sekil`'in parametresiz yapıcısı olmadığı için `Kare`'nin yapıcısı `super("Kare")`'yi yazmak zorundadır. Yazmasaydı derleyici `super()` çağrısını kendisi ekler ve parametresiz yapıcıyı bulamadığı için hata verirdi.

4.5.2.5 Şablon metot

Soyut sınıfın somut bir metodu, aynı sınıfın soyut bir metodunu çağırabilir. İlk duyuşta bu çelişkili görünür: gövdesi olmayan bir metot nasıl çağrılır? Çağrı derlenir, çünkü metodun imzası bellidir. Çalışırken ise `this` her zaman somut bir alt sınıfın nesnesini gösterir ve o nesnenin sınıfında metodun gövdesi vardır. Aşağıdaki `Icecek` sınıfı bir içeceğin hazırlanma sırasını belirler:

```
abstract class Icecek {
    final void hazirla() {
        suyuKaynat();
        demle();
        System.out.println("Servis edildi");
    }
    void suyuKaynat() {
        System.out.println("Su kaynadı");
    }
    abstract void demle();
}
class Cay extends Icecek {
    void demle() {
        System.out.println("Çay demlendi");
    }
}
// main içinde:
Icecek i = new Cay();
i.hazirla();
```

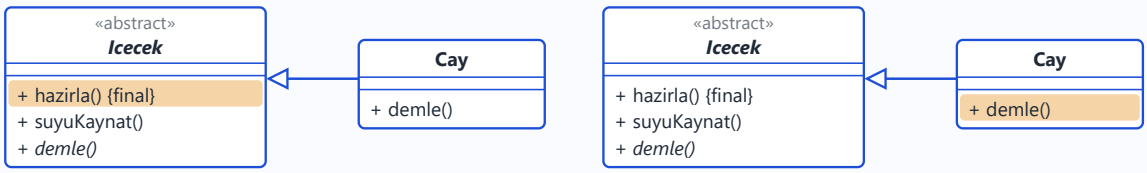
```
Su kaynadı
Çay demlendi
Servis edildi
```

`hazirla()` üç adımı sırayla çağırır: suyu kaynatmak, demlemek, servis etmek. Su kaynatmak her içecekte aynıdır; `suyuKaynat()` somut olarak `Icecek`'te yazılmıştır. Demleme içeceğe göre değişir; `demle()` soyuttur ve `Cay` onu kendi gövdesiyle doldurur. `hazirla()` ise `final`'dır: alt sınıflar onu geçersiz kılamaz, yani hazırlama sırasını bozamaz.

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	18: `Icecek i = new Cay();`	main()	i=→ Cay	—
2	2: `final void hazirla() {`	main() → hazirla()	this=→ Cay	—
3	3: `suyuKaynat();`	main() → hazirla()	this=→ Cay	—
4	8: `System.out.println("Su kaynadı");`	main() → hazirla() → suyuKaynat()	this=→ Cay	`Su kaynadı`
5	4: `demle();`	main() → hazirla()	this=→ Cay	—
6	14: `System.out.println("Çay demlendi");`	main() → hazirla() → Cay.demle()	this=→ Cay	`Çay demlendi`
7	5: `System.out.println("Servis edildi");`	main() → hazirla()	this=→ Cay	`Servis edildi`
8	19: `i.hazirla();`	main()	i=→ Cay	—

Tablo 4.5.3: `i.hazirla()` çağrısının izlenmesi. `this`, metodun çağrıldığı `Cay` nesnesini gösterir.

Tablo 4.5.3 çağrıyı adım adım verir. Birinci adımda değişkenin tipi soyut `Icecek`, nesnesi `Cay`'dir. `Cay` tek soyut metodu `demle()`'yi uyguladığı için somuttur ve nesnesi oluşturulabilir. İkinci adımda `i.hazirla()` çağrılır. `Cay`'de `hazirla()` yoktur; kalıtımla alınan `Icecek` sürümü çalışır. Üçüncü ve dördüncü adımda `suyuKaynat()`, yani `this.suyuKaynat()` çağrılır ve `Icecek`'teki somut gövde "Su kaynadı" yazar.



Şekil 4.5.4: Soldaki an: `hazirla()`'nın içinde `demle()` çağrısına gelinmiş; `Icecek`'te bu metodun gövdesi yoktur. Sağdaki an: `this` bir `Cay` nesnesi olduğu için çağrı `Cay.demle()`'ye gider.

Beşinci adım kritik andır. `hazirla()` `demle()`'yi çağırır. `Icecek`'te bu metodun gövdesi yoktur, ama `this` bir `Cay` nesnesidir. Dinamik bağlama çağrıyı `Cay.demle()`'ye yönlendirir ve altıncı adımda "Çay demlendi" yazılır (Şekil 4.5.4). Soyut sınıftaki kod, kendisi yazıldığında henüz var olmayan bir alt sınıfın kodunu çağırmıştır. Yedinci adımda servis iletisi yazdırılır.

Tanım 4.5.4

Şablon metot (*template method*), bir algoritmanın iskeletini üst sınıfta tanımlayan, değişen adımları ise alt sınıflara bırakan tasarımdır. İskeleti kuran metot genellikle `final`'dır; değişen adımlar soyut metotlardır.

Şablon metodun gücü yeni bir içecek eklenince görülür. Bir `Kahve` sınıfı yalnızca `demle()`'yi yazar; su kaynatma ve servis adımlarını, sıralarıyla birlikte, hazır alır:

```
abstract class Icecek {
    final void hazirla() {
        suyuKaynat();
        demle();
        System.out.println("Servis edildi");
    }
    void suyuKaynat() {
        System.out.println("Su kaynadı");
    }
    abstract void demle();
}
class Cay extends Icecek {
    void demle() {
        System.out.println("Çay demlendi");
    }
}
class Kahve extends Icecek {
    void demle() {
        System.out.println("Kahve süzüldü");
    }
}
public class Main {
    public static void main(String[] args) {
        Icecek[] siparis = { new Cay(), new Kahve() };
        for (Icecek i : siparis) {
            i.hazirla();
        }
    }
}
```

```
Su kaynadı
Çay demlendi
Servis edildi
Su kaynadı
Kahve süzüldü
Servis edildi
```

Döngü iki nesnede de aynı `hazirla()`'yı çağırır. İskelet değişmez; ortadaki satır her nesnede kendi sınıfının metodunu çalıştırır. Hazırlama sırası tek bir yerde yazılı olduğu için, örneğin servis öncesine bir "bardağı ısıt" adımı eklemek bütün içecekleri birlikte değiştirir.

Sık yapılan hata

`final` **şablon metodu alt sınıfta geçersiz kılmaya çalışmak.** Şablon metot iskeleti korumak için `final` yazılır. Alt sınıf onu yeniden yazmak isterse derleyici reddeder:

```
FinalHazirla.java:7: error: hazirla() in Cay cannot override hazirla() in Icecek
  void hazirla() { }
       ^
  overridden method is final
1 error
```

Sıranın bir kısmı gerçekten değişebilir olmalıysa şablon bozulmaz. İskelete gövdeli ama geçersiz kılınabilen bir "kanca" metot eklenir (örneğin `boolean sut() { return false; }`).

4.5.3 Arayüzler

Soyut sınıf, akraba sınıfların ortak atası olarak iyi çalışır. Kare de çember de birer şekildir. Ama bazı ortak noktalar akrabalıktan doğmaz. Bir kuş, bir uçak ve bir insansız hava aracı uçabilir; üçünün ortak bir atası yoktur ve olmamalıdır. Bir ördek hem uçar hem yüzer; tek bir üst sınıfı olduğu için iki ayrı soyut sınıftan aynı anda türeyemez. Java bu ihtiyaç için ikinci bir soyutlama aracı sunar: arayüz.

4.5.3.1 Arayüz bir sözleşmedir

Tanım 4.5.5

Arayüz (*interface*), sınıfların uygulaması gereken metotları belirleyen bir tiptir. `interface` anahtar sözcüğüyle tanımlanır. Bir sınıf `implements` ile arayüzü **uygular** (*implement*), yani arayüzdeki soyut metotların gövdesini yazar.

Arayüz bir **sözleşmedir** (*contract*). Onu uygulayan her sınıfın belirli metotları sunacağını garanti eder. İki yazılım parçası arasındaki buluşma noktasıdır: kullanan taraf yalnızca arayüzü bilir, sözleşmenin arkasındaki sınıfı tanımaz. Şekil 4.5.5 bu ilişkiyi çizer.



Şekil 4.5.5: Kullanan kod yalnızca `GeometrikSekil` arayüzünü ve onun `ciz()` metodunu bilir. `Kare` ve `Cember` aynı sözleşmeyi uygular (kesikli oklar).

Gündelik bir benzetme arayüzü iyi anlatır: priz ve fiş. Priz bir sözleşmedir. Hangi cihaz takılırsa takılsın aynı gerilimi verir. Ütü de bilgisayar da prizin standardına uyar. Şebeke cihazın içini bilmez, cihaz da şebekenin nasıl kurulduğunu bilmez. Yeni bir cihaz üretmek için şebekeyi değiştirmek gerekmez;

standarda uymak yeter.

Arayüzün kuralları sınıfınkinden farklıdır:

- Arayüzdeki metotlar, `public` yazılsın ya da yazılmasın `public`'tir.
- Bir metot `static`, `default` ya da `private` değilse kendiliğinden `abstract`'tır.
- Alan olarak yalnızca `public static final` sabitler bulunabilir. Örnek alanı (*instance field*) ve yapıcı yoktur.

Soyut sınıf ile arayüz arasındaki farkı iki ilişki adıyla özetlemek mümkündür. Soyut sınıf "bir türüdür" (*is-a*) ilişkisi kurar: kare bir şekildir. Arayüz ise "yapabilir" (*can-do*) yeteneği tanımlar: ördek uçabilir, ördek yüzebilir.

4.5.3.2 Arayüz tanımlamak

Aşağıda solda programcının yazdığı arayüz, sağda derleyicinin onu nasıl gördüğü verilmiştir. Derleyici, yazılmayan belirteçleri kendisi ekler:

```
public interface GeometrikSekil {
    // Soyut metot (public abstract)
    void ciz();
    // Sabit (public static final)
    int KENAR_SAYISI = 4;
}
```

```
public interface GeometrikSekil {
    public abstract void ciz();
    public static final int KENAR_SAYISI = 4;
}
```

`void ciz();` satırı `public abstract void ciz();` olarak okunur. `int KENAR_SAYISI = 4;` satırı ise `public static final` bir sabittir. Sabitlere arayüzün adıyla erişilir: `GeometrikSekil.KENAR_SAYISI` 4 verir. Sabit adları geleneksel olarak `BÜYÜK_HARF` ile yazılır. Bu örtük belirteçlerin üç sonucu vardır, üçünü de derleyici denetler.

Birincisi, arayüzdeki bir sabite değer vermek zorunludur. Değer verilmemiş bir `final` alan sonradan da değer alamaz, çünkü arayüzün yapıcısı yoktur:

```
interface GeometrikSekil {
    int deger;
}
public class SabitDegersiz { }
```

```
SabitDegersiz.java:2: error: = expected
    int deger;
             ^
1 error
```

İkincisi, sabit değiştirilemez. `final` bir alana ikinci kez değer atanamaz:

```
interface GeometrikSekil {
    void ciz();
    int KENAR_SAYISI = 4;
}
public class SabitAta {
    public static void main(String[] args) {
        GeometrikSekil.KENAR_SAYISI = 5;
    }
}
```

```
SabitAta.java:7: error: cannot assign a value to static final variable KENAR_SAYISI
        GeometrikSekil.KENAR_SAYISI = 5;
                      ^
1 error
```

Üçüncüsü, arayüzden de `new` ile nesne oluşturulamaz. İleti soyut sınıftakiyle aynıdır, çünkü arayüz de örtük olarak soyuttur:

```
ArayuzNew.java:6: error: GeometrikSekil is abstract; cannot be instantiated
        GeometrikSekil g = new GeometrikSekil();
                           ^
1 error
```

Arayüzün örnek alanı olmaması, durum (*state*) tutamaması demektir. Bir arayüz, "bir sayaç değeri saklar" diyemez; yalnızca "`deger()` diye bir metot sunar" diyebilir. Değeri saklamak onu uygulayan sınıfın işidir. Arayüzde `int deger = 0;` yazılırsa bu bir sayaç değil, değişmez bir sabittir; onu artırmaya çalışan bir metot derlenmez:

```
interface Sayac {
    int deger = 0;
    default void arttir() { deger++; }
}
public class ArayuzAlan { }
```

```
ArayuzAlan.java:3: error: cannot assign a value to static final variable deger
    default void arttir() { deger++; }
                            ^
1 error
```

4.5.3.3 Arayüzü uygulamak

Bir sınıf bir arayüzü `implements` anahtar sözcüğüyle uygular. Sınıf, arayüzdeki bütün soyut metotları yazmalı ya da kendisi `abstract` olmalıdır. Aşağıdaki `Kare`, `GeometrikSekil`'in tek soyut metodu olan `ciz()`'i yazar:

```
public class Kare implements GeometrikSekil {
    // ciz metodunu uygulamak zorunlu
    public void ciz() {
        System.out.println("Kare çiziliyor");
    }
}
// main içinde:
GeometrikSekil g = new Kare();
g.ciz();
System.out.println(GeometrikSekil.KENAR_SAYISI);
```

```
Kare çiziliyor
4
```

`GeometrikSekil g = new Kare();` satırında değişkenin tipi arayüzdür. Arayüz tipindeki bir değişken, o arayüzü uygulayan her sınıfın nesnesini gösterebilir. `g.ciz()` çağrısı derlenirken derleyici arayüze bakar; çalışırken `Kare`'nin metodu çalışır. Son satır sabite arayüz adıyla erişir ve 4 yazdırır.

Uygulayan metot `public` olmalıdır. Arayüzdeki metot örtük olarak `public`'tir ve geçersiz kılan bir metot erişimi daraltamaz. Sınıfta yalnızca `void ciz()` yazılırsa metot paket erişimli olur ve derleyici bunu erişimi zayıflatmak olarak görür:

```
interface GeometrikSekil {
    void ciz();
}
class Kare implements GeometrikSekil {
    void ciz() { System.out.println("Kare çiziliyor"); }
}
public class ArayuzZayif { }
```

```
ArayuzZayif.java:5: error: ciz() in Kare cannot implement ciz() in GeometrikSekil
    void ciz() { System.out.println("Kare çiziliyor"); }
         ^
  attempting to assign weaker access privileges; was public
1 error
```

Metot hiç yazılmazsa ileti soyut sınıftakinin aynısıdır; tek fark, eksik metodun hangi tipte bildirildiğini söyleyen son kelimedir:

```
ArayuzEksik.java:4: error: Kare is not abstract and does not override abstract method
ciz() in GeometrikSekil
class Kare implements GeometrikSekil { }
^
1 error
```

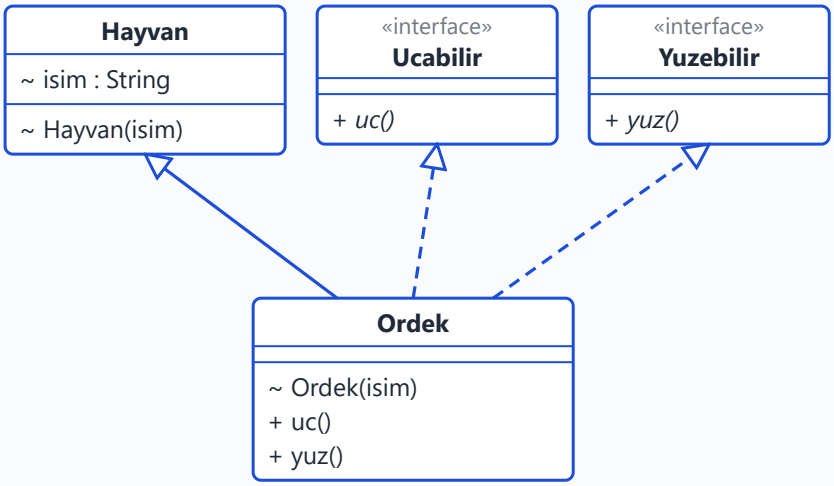
Sık yapılan hata

Arayüz metodunu `public` yazmayı unutmak. Arayüzde `public` yazılmadığı için sınıfta da yazılmayabileceği düşünülür. Oysa arayüzdeki belirteç örtüktür, sınıftaki değildir. Sınıfta belirteç yazılmazsa metot paket erişimli olur ve `attempting to assign weaker access privileges; was public` iletisi gelir. Arayüz metodu sınıfta her zaman `public` yazılır.

4.5.3.4 Bir sınıf, birden çok arayüz

Java'da bir sınıf en çok bir üst sınıfı genişletebilir. Buna karşılık sıfır ya da daha çok arayüzü uygulayabilir: `implements A, B, …`. İkisi birlikte yazılacaksa önce `extends`, sonra `implements` gelir. Aşağıdaki `Ordek` sınıfı `Hayvan`'dan türer ve iki yeteneği birden üstlenir:

```
class Hayvan {
    String isim;
    Hayvan(String isim) { this.isim = isim; }
}
interface Ucabilir { void uc(); }
interface Yuzebilir { void yuz(); }
class Ordek extends Hayvan implements Ucabilir, Yuzebilir {
    Ordek(String isim) { super(isim); }
    public void uc() { System.out.println(isim + " uçuyor"); }
    public void yuz() { System.out.println(isim + " yüzüyor"); }
}
```



Şekil 4.5.6: `Ordek` sınıfı `Hayvan`'ı genişletir (düz çizgi, içi boş üçgen) ve `Ucabilir` ile `Yuzebilir` arayüzlerini uygular (kesikli çizgi).

`Ordek`'in yapıcısı `super(isim)` ile `Hayvan`'ın yapıcısını çağırır ve `isim` alanını ondan alır. `uc()` ve `yuz()` metotları iki arayüzün sözleşmesini yerine getirir; ikisi de `public`'tir. Şekil 4.5.6 bu üç ilişkiyi UML'nin iki çizgi türüyle gösterir: kalıtım düz, arayüzü uygulama kesikli çizilir.

İki üst sınıf yazmak ise söz dizimi hatasıdır. Derleyici `extends`'ten sonra tek bir ad bekler; virgülü görünce sınıf gövdesinin başlamasını bekler:

```
class A { }
class B { }
class C extends A, B { }
public class IkiExtends { }
```

```
IkiExtends.java:3: error: '{' expected
class C extends A, B { }
                 ^
1 error
```

Bir `Ordek` nesnesi artık üç farklı tipten görülebilir: `Ordek`, `Ucabilir` ve `Yuzebilir`. Aşağıdaki program tek bir nesneyi üç farklı referansla tutar:

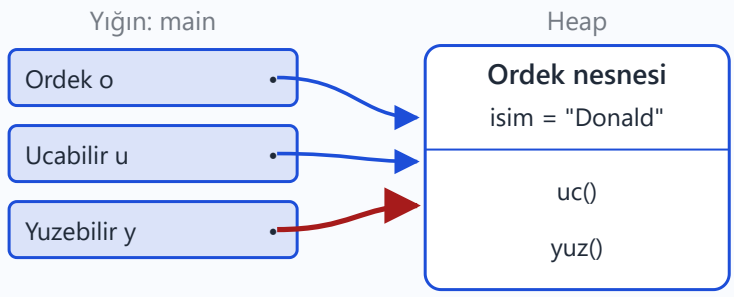
```
Ordek o = new Ordek("Donald");
Ucabilir u = o;
Yuzebilir y = o;
u.uc();
y.yuz();
// u.yuz();   derleme hatası!
System.out.println(o.isim);
System.out.println(u == o);
```

```
Donald uçuyor
Donald yüzüyor
Donald
true
```

Adım	Satır	Değişkenler	Yeni çıktı
1	1: `Ordek o = new Ordek("Donald");`	o=→ Ordek	—
2	2: `Ucabilir u = o;`	o=→ Ordek, u=→ Ordek	—
3	3: `Yuzebilir y = o;`	o=→ Ordek, u=→ Ordek, y=→ Ordek	—
4	4: `u.uc();`	o=→ Ordek, u=→ Ordek, y=→ Ordek	`Donald uçuyor`
5	5: `y.yuz();`	o=→ Ordek, u=→ Ordek, y=→ Ordek	`Donald yüzüyor`
6	6: `// u.yuz(); derleme hatası!`	o=→ Ordek, u=→ Ordek, y=→ Ordek	—
7	7: `System.out.println(o.isim);`	o=→ Ordek, u=→ Ordek, y=→ Ordek	`Donald`
8	8: `System.out.println(u == o);`	o=→ Ordek, u=→ Ordek, y=→ Ordek	`true`

Tablo 4.5.4: Aynı `Ordek` nesnesinin üç referansla kullanılması. `→ Ordek` yazımı, değişkenin heap'teki tek `Ordek` nesnesini gösterdiğini belirtir.

Tablo 4.5.4'ün ilk üç adımı yalnızca atama yapar. Birinci adımda heap'te tek bir `Ordek` nesnesi oluşur ve `o` onu gösterir. İkinci adımdaki `Ucabilir u = o;` nesneyi kopyalamaz; `u` aynı nesneyi gösterir. Ördek bir `Ucabilir` olduğu için bu dönüşüm otomatiktir, açık dönüşüm (*cast*) gerekmez. Üçüncü adımda `y` da aynı nesneyi gösterir. Artık tek nesneye üç farklı tipten bakılır (Şekil 4.5.7).



Şekil 4.5.7: Yığındaki `o`, `u` ve `y` değişkenlerinin tipleri farklıdır, ama üçü de heap'teki aynı `Ordek` nesnesini gösterir.

Dördüncü adımda `u.uc()` çağrılır. `u`'nun tipi `Ucabilir`'dir ve bu tipten yalnızca `uc()` çağrılabilir. Çalışan, nesnenin kendi `uc()` metodudur: "Donald uçuyor". Beşinci adımda `y.yuz()` çalışır; `Yuzebilir` tipi `yuz()`'ü bilir. Altıncı adım yorum satırındaki çağrıyı tartışır. `u.yuz()` yazılsaydı derleyici reddederdi:

```
RefTip.java:10: error: cannot find symbol
        u.yuz();
         ^
  symbol:   method yuz()
  location: variable u of type Ucabilir
1 error
```

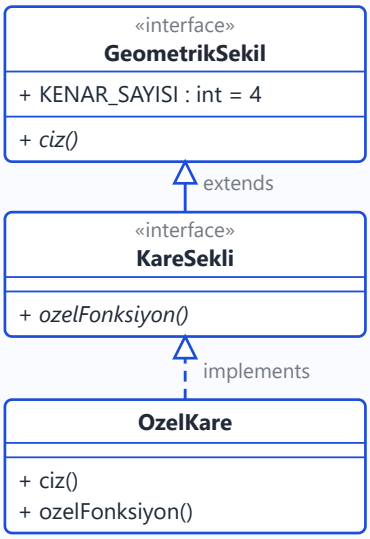
Nesnenin `yuz()` metodu vardır, ama derleyici nesneye değil referansın tipine bakar. `Ucabilir` sözleşmesinde `yuz()` yoktur. Hangi metotların çağrılabileceğini referansın tipi belirler; hangi gövdenin çalışacağını nesnenin sınıfı belirler. Bu iki kural çok biçimliliğin özüdür. Yedinci adımda `o.isim` okunur; `o`'nun tipi `Ordek` olduğu için alan dahil her üyeye erişilebilir. Son adımda `u == o` iki referansı karşılaştırır. İkisi aynı nesneyi gösterdiği için sonuç `true`'dur.

4.5.3.5 Arayüzden kalıtım

Bir arayüz başka bir arayüzü `extends` ile genişletebilir. Arayüzler arasında çoklu kalıtım serbesttir: `interface C extends A, B` yazılabilir. Genişleten arayüz, üst arayüzün bütün metotlarını devralır ve yenilerini ekler. Aşağıdaki `KareSekli`, `GeometrikSekil`'in `ciz()` metoduna bir de `ozelFonksiyon()` ekler:

```
interface GeometrikSekil {
    void ciz();
    int KENAR_SAYISI = 4;
}
interface KareSekli extends GeometrikSekil {
    // Ek metotlar veya sabitler eklenebilir
    void ozelFonksiyon();
}
class OzelKare implements KareSekli {
    public void ciz() {
        System.out.println("Özel kare çiziliyor");
    }
    public void ozelFonksiyon() {
        System.out.println("Köşegen: " + Math.sqrt(2));
    }
}
public class Main {
    public static void main(String[] args) {
        KareSekli k = new OzelKare();
        k.ciz();
        k.ozelFonksiyon();
        GeometrikSekil g = k;   // üst arayüze atanabilir
        System.out.println(g instanceof KareSekli);
    }
}
```

```
Özel kare çiziliyor
Köşegen: 1.4142135623730951
true
```



Şekil 4.5.8: `KareSekli` arayüzü `GeometrikSekil`'i genişletir; `OzelKare` sınıfı `KareSekli`'ni uygular ve iki

metodu da yazar.

`OzelKare` yalnızca `KareSekli`'ni uyguladığını bildirir, ama sözleşme iki metodu da kapsar. Biri eksik kalsaydı sınıf derlenmezdi. `OzelKare` nesnesi dolaylı olarak bir `GeometrikSekil`'dir; bu yüzden `GeometrikSekil g = k;` dönüşümsüz derlenir (Şekil 4.5.8). Programın son satırı bir `instanceof` sınamasıdır ve `true` yazar: üst arayüz tipindeki referans hâlâ aynı nesneyi gösterir. Köşegen satırındaki 1.4142135623730951, kenarı 1 olan bir karenin köşegeni olan √2'nin `double` gösterimidir.

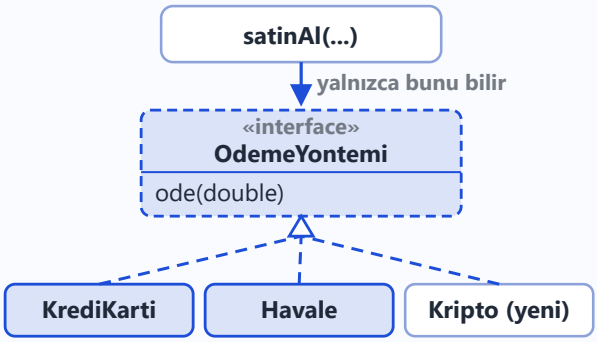
4.5.3.6 Arayüz tipiyle programlama

Arayüzün asıl gücü, kullanan kodu somut sınıflardan bağımsız kılmasıdır. Bunun yolu, değişken, parametre ve dönüş tiplerini somut sınıf yerine arayüz olarak yazmaktır. Bir alışveriş sitesinin ödeme adımını düşünelim. Ödeme kartla ya da havaleyle yapılabilir; yarın kripto para da eklenebilir.

```
interface OdemeYontemi { void ode(double tutar); }
class KrediKarti implements OdemeYontemi {
    public void ode(double tutar) {
        System.out.println("Kart: " + tutar + " TL");
    }
}
class Havale implements OdemeYontemi {
    public void ode(double tutar) {
        System.out.println("Havale: " + (tutar + 5) + " TL");
    }
}
static void satinAl(OdemeYontemi y, double toplam) {
    y.ode(toplam);
}
satinAl(new KrediKarti(), 250.0);   // main
satinAl(new Havale(), 120.0);
```

```
Kart: 250.0 TL
Havale: 125.0 TL
```

`satinAl(OdemeYontemi y, double toplam)` metodu ödemenin kartla mı havaleyle mi yapıldığını bilmez. Yalnızca `y.ode(toplam)` der; gerisini nesne yapar. Şekil 4.5.9 bu bağımlılığı çizer: `satinAl`'dan çıkan tek ok arayüze gider.



Şekil 4.5.9: `satinAl` yalnızca `OdemeYontemi` arayüzünü bilir. `KrediKarti`, `Havale` ve sonradan eklenen `Kripto` sözleşmeyi uygular.

Adım	Satır	Değişkenler	Yeni çıktı

1	15: `satinAl(new KrediKarti(), 250.0); // main`
	12: `static void satinAl(OdemeYontemi y, double toplam) {`
		y=→KrediKarti #1, toplam=250.0	—

2	13: `y.ode(toplam);` 3: `public void ode(double tutar) {` 4: `System.out.println("Kart: " + tutar + " TL");`	this=→KrediKarti #1, tutar=250.0	`Kart: 250.0 TL`
3	15: `satinAl(new KrediKarti(), 250.0); // main`	—	—
4	16: `satinAl(new Havale(), 120.0);` 12: `static void satinAl(OdemeYontemi y, double toplam) {`	y=→Havale #2, toplam=120.0	—
5	13: `y.ode(toplam);` 8: `public void ode(double tutar) {` 9: `System.out.println("Havale: " + (tutar + 5) + " TL");`	this=→Havale #2, tutar=120.0	`Havale: 125.0 TL`
6	12: `static void satinAl(OdemeYontemi y, double toplam) {` 13: `y.ode(toplam);`	—	—

Tablo 4.5.5: `satinAl` metodunun iki farklı ödeme nesnesiyle izlenmesi. `#1` ve `#2` heap'teki nesnelerin sıra numarasıdır.

Tablo 4.5.5'in birinci adımında 15. satır `satinAl`'ı yeni bir `KrediKarti` nesnesiyle çağırır. Parametre `y`'nin tipi arayüzdür; gelen nesne bir `KrediKarti`'dır. İkinci adımda 13. satırdaki `y.ode(toplam)` dinamik olarak bağlanır. Nesne kart olduğu için `KrediKarti.ode()` çalışır ve "Kart: 250.0 TL" yazılır. Üçüncü adımda çağrı biter. Kart nesnesini artık gösteren bir referans kalmamıştır; çöp toplayıcı onu silebilir.

Dördüncü adımda aynı `satinAl` bu kez bir `Havale` nesnesiyle çağrılır. Beşinci adımda aynı 13. satır bu sefer `Havale.ode()`'yi çalıştırır. Havale 5 TL masraf ekler: 120.0 + 5 = 125.0. Altıncı adımın vardığı sonuç önemlidir: `satinAl` iki farklı sınıfla çalıştı ama hiçbirinin adını bilmiyor. Yeni bir ödeme türü eklendiğinde de değişmeyecek.

Bu bağımsızlığa **gevşek bağlılık** (*loose coupling*) denir. İki somut yararı vardır. Birincisi, yeni bir ödeme türü yalnızca arayüzü uygulayan bir sınıf yazarak eklenir; `satinAl` ve onu çağıran kodlar hiç değişmez. İkincisi, kod kolay denenir. Gerçek banka yerine arayüzü uygulayan sahte bir sınıf verilir; bu sınıf para çekmez, yalnızca kendisine ne kadar ödeme geldiğini kaydeder:

```
interface OdemeYontemi { void ode(double tutar); }
class Kripto implements OdemeYontemi {
    public void ode(double tutar) {
        System.out.println("Kripto: " + tutar / 2000 + " birim");
    }
}
class SahteOdeme implements OdemeYontemi {
    double alinan = 0;
    public void ode(double tutar) { alinan += tutar; }
}
public class Main {
    static void satinAl(OdemeYontemi y, double toplam) {
        y.ode(toplam);
    }
    public static void main(String[] args) {
        satinAl(new Kripto(), 500.0);
        SahteOdeme sahte = new SahteOdeme();
        satinAl(sahte, 99.5);
        satinAl(sahte, 0.5);
        System.out.println(sahte.alinan);
    }
}
```

```
Kripto: 0.25 birim
100.0
```

`Kripto`, sonradan eklenen bir ödeme türüdür; 500 TL'yi 2000 TL'lik bir birim fiyatıyla 0.25 birime çevirir. `SahteOdeme` ise iki ödemenin toplamını tutar: 99.5 + 0.5 = 100.0. `satinAl`'ın doğru tutarı doğru nesneye ilettiği, gerçek bir bankaya bağlanmadan böylece sınanmış olur. Java'nın kendi kütüphanesi de aynı düzende yazılmıştır: `List<String> l = new ArrayList<>();` satırında değişkenin tipi `List` arayüzüdür. Program ileride `ArrayList` yerine `LinkedList` kullanmak isterse yalnızca bu satır değişir.

4.5.4 Lambda ifadeleri

`OdemeYontemi` gibi arayüzlerin çoğunda tek bir soyut metot bulunur. Böyle bir arayüzü uygulamak için ayrı bir sınıf yazmak, ona ad vermek ve gövdeyi bu sınıfın içine yerleştirmek gerekir. Sınıf yalnızca tek bir yerde, tek bir kez kullanılacaksa bu tören gereksiz uzundur. Java 8 ile gelen lambda ifadeleri, tek metotluk bir davranışı doğrudan kullanılacağı yerde yazmayı sağlar.

4.5.4.1 Adsız fonksiyon ve söz dizimi

Tanım 4.5.6

Lambda ifadesi (*lambda expression*), parametre listesi, ok işareti (`->`) ve gövdeden oluşan adsız bir fonksiyondur: `(ParametreListesi) -> İfade` . Adsız (anonim) fonksiyon, ad yerine doğrudan kodun kendisini temsil eder.



Şekil 4.5.10: Lambda ifadesinin üç parçası: parantez içinde parametre listesi, ayırıcı ok ve gövde olan ifade. İfadenin değeri, lambdanın sonucudur.

Şekil 4.5.10'daki `(int x, int y) -> x + y` ifadesi iki tam sayı alır ve toplamlarını döndürür. Aynı işi yapan sıradan metot şudur:

```
static int toplama(int x, int y) {
    return x + y;
}
```

İki yazım arasındaki farklar lambdanın kısa yazımını açıklar. Lambdanın adı yoktur; dönüş tipi yazılmaz, ifadenin tipinden çıkarılır; tek ifadeli gövdede `return` ve süslü parantez de yazılmaz. Söz diziminin birkaç biçimi vardır:

- Parametre tipleri çoğu zaman yazılmaz, derleyici onları bağlamdan çıkarır: `(x, y) -> x + y` .
- Tek parametrede parantez de gerekmez: `s -> s * 2` .
- Parametresiz lambda boş parantezle yazılır: `() -> 4` .
- Gövde birden çok deyimden oluşuyorsa süslü parantez ve `return` gerekir: `(a, b) -> { int s = a + b; return s; }` .

Aşağıdaki program bu biçimlerin hepsini aynı toplamla dener. Her lambda bir arayüz tipindeki değişkene atanmıştır. Lambdanın tipi her zaman böyle bir arayüzdür.

```
import java.util.function.IntBinaryOperator;
public class Main {
    static int toplama(int x, int y) {
        return x + y;
    }
    public static void main(String[] args) {
        IntBinaryOperator t1 = (int x, int y) -> x + y;
        IntBinaryOperator t2 = (x, y) -> x + y;
        IntBinaryOperator t3 = (a, b) -> { int s = a + b; return s; };
        java.util.function.IntUnaryOperator iki = s -> s * 2;
        System.out.println(toplama(5, 3) + " " + t1.applyAsInt(5, 3) + " " +
t2.applyAsInt(5, 3) + " "
                + t3.applyAsInt(5, 3) + " " + iki.applyAsInt(21));
    }
}
```

```
8 8 8 8 42
```

Dört toplama da 8 verir; tek parametreli `iki` lambdası 21'i ikiye katlar. Yazım değişse de davranış aynıdır.

4.5.4.2 Fonksiyonel arayüz

Java'da her değerin bir tipi vardır. Lambda ifadesinin tipi nedir? Java bu soruya yeni bir tip türü ekleyerek cevap verir: lambda, tek soyut metodu olan bir arayüzün nesnesidir.

Tanım 4.5.7

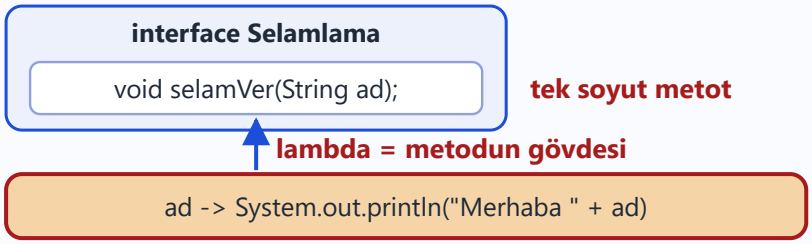
Fonksiyonel arayüz (*functional interface*), tek bir soyut metodu olan arayüzdür. Lambda ifadesi ancak böyle bir tipe atanabilir. Lambda o tek metodun gövdesini yazar; parametre ve dönüş tipleri metodun imzasından çıkarılır.

```
@FunctionalInterface
interface Selamlama {
    void selamVer(String ad);
}

Selamlama s = ad -> System.out.println("Merhaba " + ad);
s.selamVer("Ayşe");
```

```
Merhaba Ayşe
```

`Selamlama` arayüzünün tek soyut metodu `selamVer(String ad)`'dır. `ad -> System.out.println("Merhaba " + ad)` lambdası bu metodun gövdesidir. Derleyici `ad`'ın tipini metodun parametresinden `String` olarak çıkarır. Dönüş tipi `void`'dir; `println` çağrısı da bir değer üretmez. Şekil 4.5.11 eşleşmeyi gösterir.



Şekil 4.5.11: Fonksiyonel arayüzün tek soyut metodu, lambda ifadesiyle doldurulur. Lambda, metodun gövdesidir.

`@FunctionalInterface` etiketi isteğe bağlıdır. Yazılırsa derleyici arayüzün gerçekten fonksiyonel olduğunu denetler ve ikinci bir soyut metot eklenmesine izin vermez. Etiket yazılmasa da tek soyut metotlu her arayüz lambdaya hedef olabilir. Etiket, arayüzü ileride değiştirecek programcıya karşı bir güvencedir. Biri yanlışlıkla ikinci bir metot eklerse arayüzü kullanan bütün lambdalar bozulur. Etiket bu hatayı arayüzün kendisinde yakalar.

Lambda ile ilgili en önemli kural, onun ne zaman çalıştığıdır. Lambda tanımlandığında **çalışmaz**. Tanım yalnızca bir arayüz nesnesi oluşturur ve onu değişkene atar. Gövde, arayüzün metodu çağrıldığında, yani `s.selamVer(...)` satırında çalışır. Bunu görmek için programa numaralı iletiler ekleyelim:

```
@FunctionalInterface
interface Selamlama {
    void selamVer(String ad);
}
public class Main {
    public static void main(String[] args) {
        System.out.println("1: lambda tanımlanıyor");
        Selamlama s = ad -> System.out.println("3: Merhaba " + ad);
        System.out.println("2: henüz çağrılmadı");
        s.selamVer("Ayşe");
        s.selamVer("Ali");
    }
}
```

```
1: lambda tanımlanıyor
2: henüz çağrılmadı
3: Merhaba Ayşe
3: Merhaba Ali
```

Çıktıdaki sıra kuralı açıkça gösterir. Lambda 2. iletiden önce tanımlanmıştır ama gövdesi 2. iletiden sonra çalışır. Aynı lambda iki kez çağrıldığı için gövdesi de iki kez çalışır. Lambda bir değer gibi saklanan, taşınan ve istenince çağrılan bir davranıştır.

Lambdanın fonksiyonel bir arayüz dışında bir tipe atanması derleme hatasıdır. `Object` bir sınıftır, tek soyut metodu yoktur:

```
public class LambdaObject {
    public static void main(String[] args) {
        Object o = () -> System.out.println("?");
    }
}
```

```
LambdaObject.java:3: error: incompatible types: Object is not a functional interface
        Object o = () -> System.out.println("?");
                   ^
1 error
```

İki soyut metodu olan bir arayüz de lambdaya hedef olamaz. Derleyici lambdanın hangi metodun gövdesi olduğunu bilemez:

```
IkiSoyutLambda.java:7: error: incompatible types: Islem2 is not a functional interface
        Islem2 i = (a, b) -> a + b;
                   ^
    multiple non-overriding abstract methods found in interface Islem2
1 error
```

4.5.4.3 Hazır fonksiyonel arayüzler

Her lambda için yeni bir arayüz yazmak gerekmez. `java.util.function` paketinde sık kullanılan biçimler hazırdır. Tablo 4.5.6 en sık kullanılan dördünü verir.

Arayüz	Girdi	Çıktı	Metot	Örnek kullanım
`Consumer<T>`	bir değer	yok	`accept`	yazdırma, kaydetme
`Supplier<T>`	yok	bir değer	`get`	değer üretme
`Function<T, R>`	bir değer	bir değer	`apply`	dönüştürme

Arayüz	Girdi	Çıktı	Metot	Örnek kullanım
`Predicate<T>`	bir değer	`boolean`	`test`	süzme, koşul

Tablo 4.5.6: `java.util.function` paketinin dört temel arayüzü.

Arayüz adlarının yanındaki `<T>` ve `<T, R>`, tip parametreleridir. `Function<String, Integer>`, `String` alıp `Integer` döndüren bir fonksiyon demektir. Açılı parantezin içindeki tipler, girdinin ve çıktının tipidir.

```
Consumer<String> yaz = m -> System.out.println(m);
yaz.accept("Consumer çalıştı");
Supplier<Integer> zar = () -> 4;
System.out.println(zar.get());
Function<String, Integer> uzunluk = m -> m.length();
System.out.println(uzunluk.apply("Giresun"));
Predicate<Integer> pozitifMi = x -> x > 0;
System.out.println(pozitifMi.test(-7));
```

```
Consumer çalıştı
4
7
false
```

Program dört arayüzü birer kez kullanır. `Consumer` aldığı iletiyi yazar ve bir şey döndürmez. `Supplier` parametresizdir; her `get()` çağrısında 4 üretir. `Function` “Giresun” dizgisini uzunluğuna dönüştürür: 7. `Predicate` −7 için “pozitif mi?” sorusunu cevaplar: `false`. Her arayüzün metodunun adı farklıdır (`accept` , `get` , `apply` , `test`); lambda o adı bilmez, çağıran kod bilir.

İlkel tipler için ayrı arayüzler de vardır. `IntBinaryOperator` iki `int` alıp `int` döndürür, metodu `applyAsInt` 'tir. `IntUnaryOperator` tek `int` alır. Bu arayüzler kutulama ve kutudan çıkarma maliyetini ortadan kaldırır.

4.5.4.4 Lambda ne zaman çalışır?

Lambdanın bir değer gibi saklanıp sonradan çağrıldığını bir liste süzme örneğinde adım adım izleyelim. `ciftMi` lambdası bir sayının çift olup olmadığını söyler; döngü bu lambdayla bir listeyi süzer.

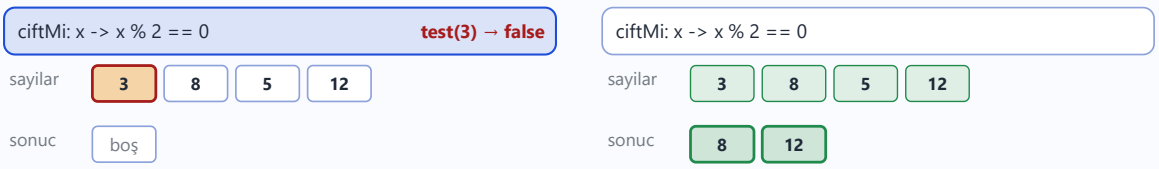
```
Predicate<Integer> ciftMi = x -> x % 2 == 0;
List<Integer> sayilar = Arrays.asList(3, 8, 5, 12);
List<Integer> sonuc = new ArrayList<>();
for (int s : sayilar) {
    if (ciftMi.test(s)) {
        sonuc.add(s);
    }
}
System.out.println(sonuc);
```

```
[8, 12]
```

Adım	Satır	Değişkenler	Yeni çıktı
1	1: `Predicate<Integer> ciftMi = x -> x % 2 == 0;`	sonuc=-	—
2	2: `List<Integer> sayilar = Arrays.asList(3, 8, 5, 12);`	sonuc=-	—
3	3: `List<Integer> sonuc = new ArrayList<>();`	sonuc=[]	—
4	4: `for (int s : sayilar) {`	s=3, sonuc=[]	—
5	5: `if (ciftMi.test(s)) {`	s=3, test=false, sonuc=[]	—
6	4: `for (int s : sayilar) {`	s=8, sonuc=[]	—
7	5: `if (ciftMi.test(s)) {`	s=8, test=true, sonuc=[]	—
8	6: `sonuc.add(s);`	s=8, test=true, sonuc=[8]	—
9	4: `for (int s : sayilar) {`	s=5, sonuc=[8]	—
10	5: `if (ciftMi.test(s)) {`	s=5, test=false, sonuc=[8]	—
11	4: `for (int s : sayilar) {`	s=12, sonuc=[8]	—
12	5: `if (ciftMi.test(s)) {`	s=12, test=true, sonuc=[8]	—
13	6: `sonuc.add(s);`	s=12, test=true, sonuc=[8, 12]	—
14	9: `System.out.println(sonuc);`	sonuc=[8, 12]	`[8, 12]`

Tablo 4.5.7: `Predicate` ile süzmenin izlenmesi. `test` sütunu, o adımda lambdanın döndürdüğü değerdir.

Tablo 4.5.7'nin birinci adımında 1. satır çalışır. Lambda bir `Predicate<Integer>` nesnesi olarak saklanır; gövdesi henüz çalışmamıştır. `ciftMi`, bu davranışı gösteren bir referanstır. İkinci adımda liste 3, 8, 5, 12 öğeleriyle kurulur, üçüncü adımda sonuç listesi boş başlar.



Şekil 4.5.12: Solda `ciftMi.test(3)` çağrısı lambdanın gövdesini çalıştırır ve `false` döner. Sağda döngü bitmiştir: sonuç listesinde 8 ve 12 vardır.

Dördüncü adımda döngü ilk öğeyi alır: `s = 3`. Beşinci adımda `ciftMi.test(3)` çağrılır ve lambdanın gövdesi ilk kez çalışır: `3 % 2 == 0` yanlıştır, sonuç `false`'tur (Şekil 4.5.12'nin solu). 3 listeye eklenmez. Altıncı ve yedinci adımlarda aynı iş 8 için yapılır: `8 % 2 == 0` doğrudur. Sekizinci adımda 8 sonuç listesine eklenir. 5 için koşul yanlış, 12 için doğrudur; on üçüncü adımda 12 de eklenir. Son adımda döngü biter ve `[8, 12]` yazdırılır.

Lambda dört kez çalışmıştır: her `test` çağrısında bir kez. Tanım anında hiç çalışmamıştır. Bu sayımı doğrulamak için lambdanın gövdesine bir sayaç eklenebilir:

```
import java.util.*;
import java.util.function.*;
public class Main {
    static int sayac = 0;
    public static void main(String[] args) {
        Predicate<Integer> ciftMi = x -> { sayac++; return x % 2 == 0; };
        System.out.println("tanımdan sonra: " + sayac);
        List<Integer> sayilar = Arrays.asList(3, 8, 5, 12);
        List<Integer> sonuc = new ArrayList<>();
        for (int s : sayilar) {
            if (ciftMi.test(s)) {
                sonuc.add(s);
            }
        }
        System.out.println(sonuc + " " + sayac);
    }
}
```

```
tanımdan sonra: 0
[8, 12] 4
```

Tanımdan hemen sonra sayaç 0'dır; döngüden sonra 4'tür. Sayacın yerel değişken değil `static` bir alan olduğuna dikkat edilmelidir. Bunun nedeni, lambdanın içinden yerel değişkenlerin değiştirilememesidir.

4.5.4.5 Akışla süzme ve dönüştürme

Lambdanın en sık kullanıldığı yer, koleksiyonların işlenmesidir. Java 8 ile gelen akış (*stream*) arayüzü, bir listeyi bir dizi işlemden geçirmeyi sağlar ve her işlem bir lambda alır. Aşağıdaki kod `import java.util.*;`, `java.util.function.IntBinaryOperator` ve `java.util.stream.Collectors` içe aktarılmış bir `main` içinde çalışır:

```
IntBinaryOperator toplama = (int x, int y) -> x + y;
System.out.println(toplama.applyAsInt(5, 3));

List<Integer> sayilar = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
List<Integer> ciftSayilar = sayilar.stream().
                    filter(s -> s % 2 == 0).
                    collect(Collectors.toList());
System.out.println(ciftSayilar);
System.out.println(sayilar.stream().map(s -> s * s).toList());
```

```
8
[2, 4, 6]
[1, 4, 9, 16, 25, 36, 49]
```

İlk iki satır toplama lambdasını `IntBinaryOperator` tipine atar ve `applyAsInt(5, 3)` ile çağırır: 8. Ardından 1'den 7'ye kadar sayıların listesi kurulur. `sayilar.stream()` listeden bir akış üretir. `filter(s -> s % 2 == 0)` yalnızca lambdanın `true` döndürdüğü öğeleri, yani çift sayıları tutar. `collect(Collectors.toList())` kalan öğeleri yeni bir listede toplar: `[2, 4, 6]`. Lambda burada açıkça yazılmış bir döngünün yerini almıştır; döngüyü `filter`'ın içinde Java kütüphanesi çalıştırır.

Son satırdaki `map(s -> s * s)` her öğeyi dönüştürür: her sayı karesiyle yer değiştirir. Sonuç `[1, 4, 9, 16, 25, 36, 49]`'dur. Kısaca `filter` öğe sayısını azaltır ama değerlere dokunmaz; `map` öğe sayısını korur ama değerleri değiştirir. Son satırdaki `.toList()` Java 16 ile gelmiştir. Daha eski sürümlerde aynı iş `collect(Collectors.toList())` ile yapılır. Burada akışın yalnızca lambda alan yüzü kullanılmıştır.

4.5.4.6 Lambda ve anonim sınıf

Lambdadan önce tek metotluk bir davranış **anonim sınıfla** (*anonymous class*) yazılırdı. Anonim sınıf, tanımlandığı yerde tek nesnesi oluşturulan adsız bir iç sınıftır. Aşağıdaki program aynı `Runnable` arayüzünü iki yolla uygular:

```
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Anonim sınıf");
    }
};
Runnable r2 =
        () -> System.out.println("Lambda");
r1.run();
r2.run();
```

```
Anonim sınıf
Lambda
```

`r1` için yazılan anonim sınıf altı satır tutar: `new Runnable()`, süslü parantez, `@Override`, tam metot imzası ve gövde. `r2` için yazılan lambda aynı işi tek ifadede yapar. Derleyici lambdayı `Runnable`'ın tek metodu `run()`'ın gövdesi olarak yorumlar. İki nesnenin `run()` çağrıları beklenen iletileri yazar.

Lambdanın kısalığının bir bedeli vardır; üç sınırı bilinmelidir. Birincisi, lambda yalnızca fonksiyonel bir arayüz tipine atanabilir. Anonim sınıf ise birden çok metodu olan bir arayüzü ya da bir sınıfı da genişletebilir.

İkincisi, lambdanın kullandığı yerel değişkenler `final` ya da **fiilen final** (*effectively final*) olmalıdır. Fiilen final, `final` yazılmamış ama ilk değerinden sonra hiç değer atanmamış değişken demektir. Aşağıdaki lambda `kdv` yerel değişkenini okur ve 150 TL'lik bir fiyata %20 KDV ekler: $150 + 150 \cdot 20 / 100 = 180$.

```
int kdv = 20;
Function<Integer, Integer> vergili =
        fiyat -> fiyat + fiyat * kdv / 100;
System.out.println(vergili.apply(150)); // 180
kdv = 18;  // eklenirse: derleme hatası
```

Son satır yorumda kalırsa program derlenir ve 180 yazar. O satır eklenirse `kdv` artık fiilen final değildir ve derleyici lambdanın bulunduğu satırı işaret eder:

```
LambdaFinal.java:6: error: local variables referenced from a lambda expression must be
final or effectively final
                fiyat -> fiyat + fiyat * kdv / 100;
                                         ^
1 error
```

Kuralın nedeni lambdanın sonradan çalışabilmesidir. Lambda, tanımlandığı metottan çok sonra, hatta o metot bittikten sonra çağrılabilir. Bu yüzden yerel değişkenin kendisini değil, değerinin bir kopyasını taşır. Değişken sonradan değişseydi lambdanın gördüğü değer ile metodun gördüğü değer birbirinden ayrılırdı. Java bu belirsizliği baştan yasaklar. Alanlar bu kurala girmez; lambda bir alanı okuyabilir ve değiştirebilir. Sayaç örneğinde sayacın `static` alan olarak yazılmasının nedeni budur.

Üçüncüsü, `this` anahtar sözcüğünün anlamı farklıdır. Lambda içindeki `this` çevreleyen nesneyi, yani lambdayı yazan metodun nesnesini gösterir. Anonim sınıftaki `this` ise anonim nesnenin kendisini gösterir:

```
public class Main {
    String ad = "dış nesne";
    void dene() {
        Runnable lam = () -> System.out.println("lambda: " + this.ad);
        Runnable ano = new Runnable() {
            String ad = "anonim nesne";
            public void run() { System.out.println("anonim: " + this.ad); }
        };
        lam.run();
        ano.run();
    }
    public static void main(String[] args) {
        new Main().dene();
    }
}
```

```
lambda: dış nesne
anonim: anonim nesne
```

İki gövde de `this.ad` okur. Lambdanın kendine ait bir nesnesi olmadığı için `this` dıştaki `Main` nesnesidir. Anonim sınıf ise kendi `ad` alanı olan ayrı bir nesnedir.

4.5.4.7 Kullanım alanları

Lambda ifadeleri üç tür işte kodu belirgin biçimde kısaltır:

- **Koleksiyon işlemleri.** Liste süzme, sıralama, eşleştirme ve her öğe için bir iş yapma (`filter`, `sort`, `map`, `forEach`).
- **İş parçacığı yönetimi.** Paralel yürütülecek bir işin gövdesini kısaca vermek.
- **Okunabilir kod.** Tek yerde kullanılacak küçük bir işlem için ayrı bir metot ya da sınıf tanımlamak yerine işlemi kullanıldığı yerde yazmak.

```
List<String> isimler = new ArrayList<>(List.of("Zeynep", "Ali", "Mehmet"));
isimler.sort((a, b) -> a.length() - b.length());
System.out.println(isimler);
isimler.forEach(isim -> System.out.println("Merhaba " + isim));
Thread is = new Thread(() -> System.out.println("İş parçacığı çalıştı"));
is.start();
```

```
[Ali, Zeynep, Mehmet]
Merhaba Ali
Merhaba Zeynep
Merhaba Mehmet
İş parçacığı çalıştı
```

`isimler.sort(...)` çağrısı sıralama ölçütünü bir lambda olarak alır. `(a, b) -> a.length() - b.length()` iki ismin uzunluk farkını döndürür. Fark negatifse `a` önce gelir, pozitifse `b` önce gelir, sıfırsa ikisi eşit sayılır. “Ali” üç harfli olduğu için başa geçer. “Zeynep” ve “Mehmet” altışar harflidir; Java'nın liste sıralaması eşit öğelerin ilk sırasını korur, bu yüzden “Zeynep” “Mehmet”'ten önce kalır. Böyle sıralamaya kararlı sıralama denir.

`forEach` listenin her öğesi için verilen lambdayı çağırır ve üç selam yazar. Son iki satır bir iş parçacığı oluşturur. `Thread`'in yapıcısı bir `Runnable` bekler; `() -> ...` parametresiz bir lambdadır ve `run()`'ın gövdesi olur. `is.start()` bu gövdeyi ayrı bir iş parçacığında başlatır. Çalıştırılan programa `is.join()` satırı eklenmiştir; `main`'in iş parçacığının bitmesini beklemesini sağlar. Bu programda son ileti zaten en son yazılır, çünkü ondan önce başka iş kalmamıştır.

Sıralama ölçütünü değiştirmek yalnızca lambdayı değiştirmek demektir. Aşağıdaki program aynı listeyi üç ölçütle sıralar. `Comparator.comparing(String::length)` “uzunluğa göre” demenin hazır yoludur; `String::length` yazımı, yalnızca bir metodu çağıran `s -> s.length()` lambdasının kısaltmasıdır (metot referansı).

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        List<String> isimler = new ArrayList<>(List.of("Zeynep", "Ali", "Mehmet"));
        isimler.sort(Comparator.comparing(String::length));
        System.out.println(isimler);
        isimler.sort((a, b) -> b.length() - a.length());
        System.out.println(isimler);

isimler.sort(Comparator.comparing(String::length).thenComparing(Comparator.naturalOrder()))
;
        System.out.println(isimler);
    }
}
```

```
[Ali, Zeynep, Mehmet]
[Zeynep, Mehmet, Ali]
[Ali, Mehmet, Zeynep]
```

İlk satır kısa olandan uzuna, ikinci satır uzundan kısaya sıralar; ikincide de eşit uzunluktakiler ilk sıralarını korur. Üçüncü satır uzunluk eşitse alfabetik sıraya bakar ve "Mehmet"'i "Zeynep"'in önüne alır.

4.5.5 Arayüzün gövdeli metotları

Java 8'e kadar bir arayüzde yalnızca soyut metotlar ve sabitler bulunabiliyordu. Bu katılık büyük bir sorun doğurdu. Java'nın kendi `Collection` ve `List` arayüzlerine lambda ile çalışan yeni metotlar (`forEach`, `removeIf`, `stream`) eklenmek istendi. Arayüze yeni bir soyut metot eklemek, onu uygulayan bütün sınıfları bozar: dünyadaki her liste sınıfı birden derlenmez hâle gelirdi. Çözüm, arayüzde gövdeli metot yazmaya izin vermek oldu. Bugün bir arayüzde üç tür gövdeli metot bulunabilir: varsayılan (`default`), `static` ve `private`.

4.5.5.1 Varsayılan metotlar

Tanım 4.5.8

Varsayılan metot (*default method*), arayüzde `default` anahtar sözcüğüyle ve gövdesiyle yazılan metottur (Java 8+). Arayüzü uygulayan sınıf onu hazır olarak kalıtımla alır; isterse geçersiz kılar.

Aşağıdaki `Collection` arayüzü üç soyut metot ve bir varsayılan metot içerir. Bu `Collection` bizim yazdığımız küçük bir arayüzdür; `java.util.Collection` ile karıştırılmamalıdır.

```
interface Collection {
    void add(Object o);
    void remove(Object o);
    int size();
    default boolean isEmpty() {
        return size() == 0;
    }
}
```

```
class Sepet implements Collection {
    private int adet = 0;
    public void add(Object o) { adet++; }
    public void remove(Object o) { adet--; }
    public int size() { return adet; }
    // isEmpty() yazılmadı: default sürüm kullanılır
}
// main içinde:
Sepet sepet = new Sepet();
System.out.println(sepet.isEmpty());
sepet.add("elma");
sepet.add("armut");
System.out.println(sepet.size() + " " + sepet.isEmpty());
```

```
true
2 false
```

`isEmpty()` soyut `size()`'ı çağırır. Bir koleksiyonun boş olup olmadığı, eleman sayısının sıfır olup olmadığıdır; bu bilgi her koleksiyon için aynı yoldan hesaplanır. Hangi `size()`'ın çalışacağı ise nesneye göre belirlenir. `Sepet` yalnızca üç soyut metodu yazmıştır; `isEmpty()`'yi hazır alır. Boş sepette `true`, iki öğe eklendikten sonra `2 false` yazılır.

Varsayılan metodun çalışma biçimi şablon metoda benzer: arayüzdeki gövde, henüz yazılmamış bir sınıfın metodunu çağırır. Aşağıdaki kısa sürümü adım adım izleyelim.

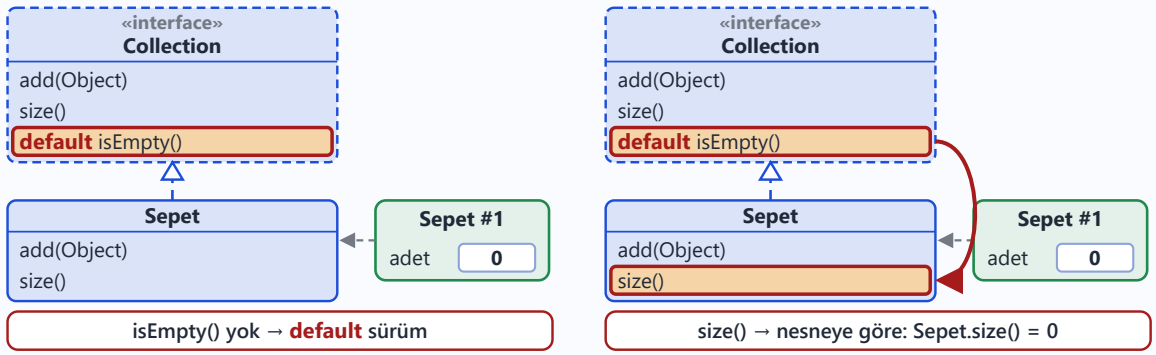
```
interface Collection {
    void add(Object o);
    int size();
    default boolean isEmpty() {
        return size() == 0;
    }
}
class Sepet implements Collection {
    private int adet = 0;
    public void add(Object o) { adet++; }
    public int size() { return adet; }
}
Collection c = new Sepet();      // main
System.out.println(c.isEmpty());
c.add("elma");
System.out.println(c.isEmpty());
```

```
true
false
```

Adım	Satır	Değişkenler	Yeni çıktı
1	13: `Collection c = new Sepet(); // main`	c=→Sepet #1, adet=0	—
2	14: `System.out.println(c.isEmpty());` 4: `default boolean isEmpty() {`	c=→Sepet #1	—
3	5: `return size() == 0;` 11: `public int size() { return adet; }`	size()=0	—
4	5: `return size() == 0;`	size()=0, sonuç=true	`true`
5	15: `c.add("elma");` 10: `public void add(Object o) { adet++; }`	adet=1	—
6	16: `System.out.println(c.isEmpty());` 5: `return size() == 0;` 11: `public int size() { return adet; }`	size()=1, sonuç=false	`false`

Tablo 4.5.8: Varsayılan `isEmpty()` metodunun iki çağrısının izlenmesi.

Tablo 4.5.8'in birinci adımında bir `Sepet` nesnesi oluşur ve arayüz tipindeki `c`'ye atanır. `Sepet` yalnızca iki soyut metodu, `add` ve `size`'ı yazmıştır. İkinci adımda `c.isEmpty()` çağrılır. JVM metodu önce nesnenin sınıfında arar; `Sepet`'te `isEmpty()` yoktur. Arama arayüze çıkar ve oradaki varsayılan gövde çalışır.



Şekil 4.5.13: Solda `Sepet`'te `isEmpty()` bulunmadığı için arayüzdeki varsayılan sürüm seçilir. Sağda varsayılan gövdenin çağırdığı `size()` dinamik olarak `Sepet.size()`'a bağlanır.

Üçüncü adımda varsayılan gövde `size()`'ı çağırır. Bu çağrı `this.size()` demektir ve `this` bir `Sepet`'tir; hangi gövdenin çalışacağını nesne belirler: `Sepet.size()` 0 döndürür (Şekil 4.5.13). Dördüncü adımda `0 == 0` karşılaştırması `true` verir ve yazdırılır. Beşinci adımda `c.add("elma")` `Sepet`'in sürümünü çalıştırır ve `adet` 1 olur. Altıncı adımda aynı varsayılan gövde bu kez 1 görür ve `false` döndürür.

Örneğin dersi şudur: arayüz yazıldıktan sonra eklenen bir varsayılan metot, eski sınıfları bozmadan onlara da gelir. `Sepet` yazılırken `isEmpty()` hiç bilinmiyor olsaydı bile, arayüze varsayılan metot olarak eklendiği gün `Sepet` nesneleri onu kullanmaya başlardı. Java'nın `forEach` ve `removeIf` metotları bütün liste sınıflarına bu yolla eklenmiştir.

Varsayılan metotla ilgili bir öncelik kuralı daha vardır: **sınıf kazanır**. Bir sınıf aynı imzalı bir metodu hem üst sınıfından hem bir arayüzün varsayılan metodundan alıyorsa üst sınıftaki sürüm seçilir:

```
class Makine {
    public void calis() { System.out.println("Makine çalışıyor"); }
}
interface Yazici {
    default void calis() { System.out.println("Yazdırılıyor"); }
}
class Fotokopi extends Makine implements Yazici { }
public class Main {
    public static void main(String[] args) {
        Yazici y = new Fotokopi();
        y.calis();
    }
}
```

```
Makine çalışıyor
```

`Fotokopi` `calis()`'ı yazmamıştır. Metodu hem `Makine`'den hem `Yazici`'nın varsayılan gövdesinden alabilir. Kural sınıf hiyerarşisini öne koyar ve "Makine çalışıyor" yazılır. Varsayılan metot, sınıfta zaten bulunan davranışı ezmez; yalnızca eksik olanı tamamlar.

4.5.5.2 static metotlar

Arayüzde `static` metot da yazılabilir (Java 8+). Bu metotlar nesneye bağlı değildir; arayüzle ilgili yardımcı işlevleri bir arada tutar. Yalnızca arayüzün adıyla çağrılır.

```
interface Geometry {
    static double circleArea(double r) { return Math.PI * r * r; }
    static double squareArea(double s) { return s * s; }
    static double perimeter(double s) { return s * 4.0; }
    static double boxVolume(double w, double h, double d) {
        return w * h * d;
    }
}
// main içinde:
System.out.println(Geometry.circleArea(2));
System.out.println(Geometry.squareArea(3));
System.out.println(Geometry.perimeter(2.5));
System.out.println(Geometry.boxVolume(2, 3, 4));
```

```
12.566370614359172
9.0
10.0
24.0
```

`Geometry` arayüzü dört geometrik hesabı toplar. Yarıçapı 2 olan dairenin alanı π · 2 · 2 = 12.566370614359172'dir. Kenarı 3 olan karenin alanı 9.0, kenarı 2.5 olan karenin çevresi 10.0'dır. 2 × 3 × 4 boyutlu kutunun hacmi ise 24.0'dır. Çağrılar `Geometry.circleArea(2)` biçimindedir.

Arayüzün `static` metotları onu uygulayan sınıfa kalıtımla geçmez. Bu, sınıflardaki `static` metotlardan farklıdır; bir sınıfın `static` metodu alt sınıfın adıyla da çağrılabilir. `Daire implements Geometry` olsa bile `Daire.squareArea(2)` derlenmez:

```
interface Geometry {
    static double squareArea(double s) { return s * s; }
}
class Daire implements Geometry { }
public class StaticKalitim {
    public static void main(String[] args) {
        System.out.println(Daire.squareArea(2));
    }
}
```

```
StaticKalitim.java:7: error: cannot find symbol
        System.out.println(Daire.squareArea(2));
                                 ^
  symbol:   method squareArea(int)
  location: class Daire
1 error
```

Kuralın nedeni çoklu kalıtımdır. Bir sınıf birçok arayüzü uygulayabilir; iki arayüzde aynı adlı `static` metot bulunsaydı `Daire.squareArea` hangisini gösterecekti? Java metodu her zaman arayüz adıyla çağırtarak bu soruyu ortadan kaldırır.

Dairenin alanında kullanılan π sabiti `Math` sınıfındadır ve tam adıyla yazılmalıdır. Yalnızca `PI` yazılırsa derleyici onu arayüzde arar ve bulamaz:

```
PIYalniz.java:2: error: cannot find symbol
    static double circleArea(double r) { return PI * r * r; }
                                                ^
  symbol:   variable PI
  location: interface Geometry
1 error
```

4.5.5.3 Default çakışması

Bir sınıf iki arayüzü uyguladığında ve iki arayüz aynı imzalı birer varsayılan metot getirdiğinde sınıf hangisini alacağını bilemez. Java bu durumda tahmin yürütmez; sınıfı metodu kendisi yazmaya, yani geçersiz kılmaya zorlar. İstenen arayüzün sürümü `ArayuzAdi.super.metot()` yazımıyla çağrılabilir.

```
interface Selamlayan {
    default void selam() { System.out.println("Merhaba"); }
}
interface Karsilayan {
    default void selam() { System.out.println("Hoş geldiniz"); }
}
class Gorevli implements Selamlayan, Karsilayan {
    public void selam() {       // geçersiz kılmak zorunlu
        Selamlayan.super.selam();
        Karsilayan.super.selam();
    }
}
```

```
Merhaba
Hoş geldiniz
```

`Gorevli` iki selamı da kullanmak istemiştir. `Selamlayan.super.selam()` "Merhaba"yı, `Karsilayan.super.selam()` "Hoş geldiniz"i yazar. `super`'in önündeki arayüz adı, hangi üst tipin varsayılan gövdesinin istendiğini söyler. `Gorevli`'de `selam()` hiç yazılmasaydı sınıf derlenmezdi:

```
DefaultCakisma.java:7: error: types Selamlayan and Karsilayan are incompatible;
class Gorevli implements Selamlayan, Karsilayan { }
^
  class Gorevli inherits unrelated defaults for selam() from types Selamlayan and
Karsilayan
1 error
```

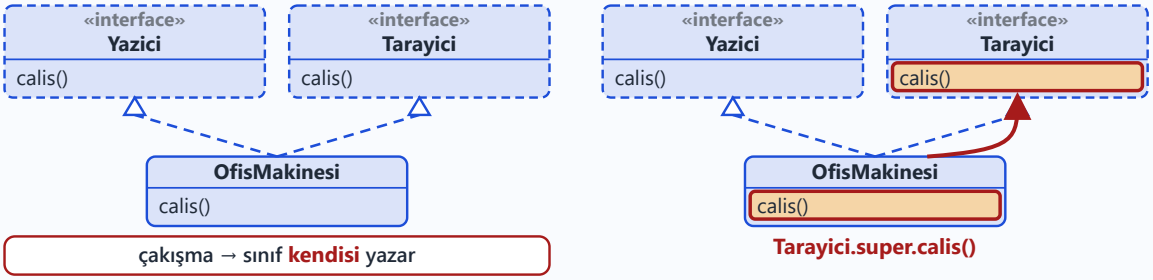
Aşağıdaki ofis makinesi örneği aynı durumu adım adım gösterir. Makine hem yazıcı hem tarayıcıdır ve iki arayüz de `calis()` için varsayılan gövde getirir.

```
interface Yazici {
    default void calis() { System.out.println("Yazdırılıyor"); }
}
interface Tarayici {
    default void calis() { System.out.println("Taranıyor"); }
}
class OfisMakinesi implements Yazici, Tarayici {
    public void calis() {
        Tarayici.super.calis();
        Yazici.super.calis();
        System.out.println("Kopya hazır");
    }
}
Yazici y = new OfisMakinesi();   // main
y.calis();
```

```
Taranıyor
Yazdırılıyor
Kopya hazır
```

Adım	Satır	Değişkenler	Yeni çıktı
1	14: `Yazici y = new OfisMakinesi(); // main`	y=→OfisMakinesi #1 (tip: Yazici)	—
2	15: `y.calis();` 8: `public void calis() {`	this=→OfisMakinesi #1	—
3	9: `Tarayici.super.calis();` 5: `default void calis() { System.out.println("Taranıyor"); }`	this=→OfisMakinesi #1	`Taranıyor`
4	10: `Yazici.super.calis();` 2: `default void calis() { System.out.println("Yazdırılıyor"); }`	this=→OfisMakinesi #1	`Yazdırılıyor`
5	11: `System.out.println("Kopya hazır");`	this=→OfisMakinesi #1	`Kopya hazır`
6	15: `y.calis();`	y=→OfisMakinesi #1	—

Tablo 4.5.9: Çakışan iki varsayılan metodu birleştiren `OfisMakinesi.calis()`'ın izlenmesi.



Şekil 4.5.14: Solda iki arayüz aynı varsayılan metodu getirir; çakışmayı sınıfın kendisi çözmek zorundadır. Sağda `Tarayici.super.calis()` adı verilen arayüzün gövdesini çağırır.

Tablo 4.5.9'un birinci adımında `OfisMakinesi` nesnesi oluşur ve `Yazici` tipindeki `y`'ye atanır. İkinci adımda `y.calis()` çağrılır. Referansın tipi `Yazici`'dır ama nesne `OfisMakinesi`'dir; sınıfın kendi `calis()` metodu çalışır. Referansın `Yazici` olması, `Yazici`'nın varsayılan gövdesinin seçileceği anlamına gelmez; sınıfın yazdığı sürüm her zaman önce gelir.

Üçüncü adımda `Tarayici.super.calis()` `Tarayici`'nın varsayılan gövdesini çağırır ve "Taranıyor" yazılır. Dördüncü adımda sıra `Yazici`'nınkine gelir: "Yazdırılıyor". Hangi sürümlerin hangi sırayla kullanılacağına sınıf karar verir. Beşinci adımda sınıf kendi eklemesini yapar: "Kopya hazır". Sonuç, iki yeteneği birleştiren yeni bir davranıştır. Altıncı adımda çağrı biter.

Sık yapılan hata

Arayüzün varsayılan metodunu `super.metot()` ile çağırmaya çalışmak. Yalın `super` üst *sınıfı* gösterir; arayüzün gövdesine ulaşmak için arayüzün adı yazılır: `Tarayici.super.calis()`. Bu yazım yalnızca sınıfın doğrudan uyguladığı arayüzler için geçerlidir.

4.5.5.4 private metotlar

Varsayılan metotlar çoğaldıkça aralarında ortak kod birikir. Bu kodu ayrı bir metoda taşımak gerekir, ama metot `default` olursa arayüzü uygulayan her sınıfa açılır. Java 9 bu sorunu `private` arayüz metotlarıyla çözer.

```
interface Yazdirilabilir {
    String icerik();
    default void yazdir() {
        System.out.println(baslik() + ": " + icerik());
    }
    private String baslik() {   // Java 9+
        return "Belge";
    }
}
class Rapor implements Yazdirilabilir {
    public String icerik() { return "Vize sonuçları"; }
}
```

```
Belge: Vize sonuçları
```

`private` metotlar, varsayılan ve `static` metotların ortak yardımcı kodunu dışarı açmadan paylaşır. Gövdeleri vardır ve yalnızca arayüzün içinden çağrılabilir. Akış şöyledir: `yazdir()` (varsayılan) önce `baslik()`'ı (`private`), sonra `icerik()`'i (`Rapor`'un uyguladığı soyut metot) çağırır ve ikisini birleştirir (Şekil 4.5.15).



Şekil 4.5.15: Varsayılan `yazdir()` metodu, yalnızca arayüzün içinden görülen `baslik()`'ı ve `Rapor`'un yazdığı `icerik()`'i çağırır; sonuç "Belge: Vize sonuçları"dır.

Uygulayan sınıf `private` metodu göremez. `Rapor` içinden `baslik()` çağrılırsa derleyici metodu `Rapor`'da arar ve bulamaz:

```
PrivateCagri.java:11: error: cannot find symbol
    public String icerik() { return baslik(); }
                                    ^
  symbol:   method baslik()
  location: class Rapor
1 error
```

4.5.5.5 Arayüz üyelerinin özeti

Arayüzde bulunabilecek bütün üyeler, örtük belirteçleri ve hangi Java sürümüyle geldikleri Tablo 4.5.10'da toplanmıştır.

Üye	Örnek	Örtük belirteçler	Gövde	Java
Soyut metot	`void ciz();`	`public abstract`	yok	1.0
Sabit	`int KENAR_SAYISI = 4;`	`public static final`	—	1.0
Varsayılan metot	`default boolean isEmpty() {…}`	`public`	var	8
`static` metot	`static double squareArea(…) {…}`	`public`	var	8
Özel metot	`private String baslik() {…}`	yalnızca arayüz içinde	var	9
Örnek alanı, yapıcı	—	İzin verilmez: arayüzün nesnesi yoktur, durum tutamaz		

Tablo 4.5.10: Arayüz üyeleri.

Uygulayan sınıf için kural kısadır: arayüzdeki bütün soyut metotlar `public` olarak yazılır. Varsayılan metotları yazmak isteğe bağlıdır; `static` ve `private` metotlar ise zaten kalıtımla geçmez. Tablonun son satırındaki yasak da derleyicide görülür. Arayüzün içine bir yapıcı yazılırsa derleyici onu yapıcı olarak tanımaz; dönüş tipi olmayan bir metot gibi okur ve tip adı bekler:

```
interface Sayac {
    Sayac() { }
}
public class ArayuzYapici { }
```

```
ArayuzYapici.java:2: error: <identifier> expected
    Sayac() { }
         ^
1 error
```

4.5.6 Mühürlü tipler

Bir sınıf ya da arayüz yazıldığında, varsayılan olarak herkes onu genişletebilir ya da uygulayabilir. Bu açıklık çoğu zaman yararlıdır, ama bazen tasarımın gerçeğini yansıtmaz. Bir trafik ışığı yalnızca kırmızı, sarı ya da yeşil olabilir; bir ödeme yalnızca kart, havale ya da nakit olabilir. Böyle bir hiyerarşide yeni bir alt tipin habersizce eklenmesi, onu ele alan bütün kodu eksik bırakır. Java 17 ile gelen mühürleme, alt tiplerin listesini kapatmayı sağlar.

Tanım 4.5.9

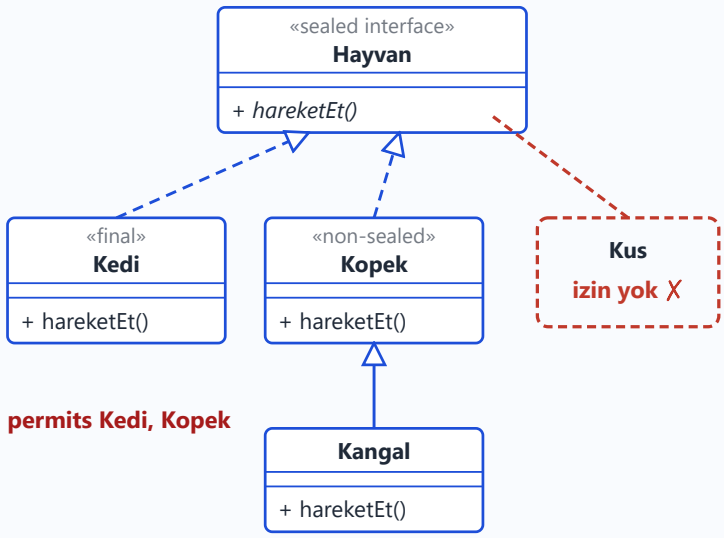
Mühürlü sınıf ya da arayüz (*sealed class / interface*), `sealed` anahtar sözcüğüyle tanımlanan tiptir (Java 17+). Bu tipi yalnızca `permits` listesinde adı geçen alt tipler genişletebilir ya da uygulayabilir.

İzin verilen her alt tip, hiyerarşinin kendisinden sonra nasıl devam edeceğini seçmek zorundadır. Üç seçenek vardır:

- `final` : hiyerarşi burada biter; bu tipten türetme yapılamaz.
- `sealed` : tip yeniden kısıtlanır; kendi `permits` listesini verir.
- `non-sealed` : kısıt kaldırılır; herkes bu tipi genişletebilir.

Mühürlemenin asıl yararı, derleyicinin bütün alt tipleri bilmesidir. Böyle bir hiyerarşiye **kapalı hiyerarşi** denir. Aşağıdaki örnekte `Hayvan` arayüzü yalnızca `Kedi` ve `Kopek` 'e izin verir:

```
public sealed interface Hayvan permits Kedi, Kopek {
    void hareketEt();
}
final class Kedi implements Hayvan {
    public void hareketEt() {
        System.out.println("Kedi sessizce yürüyor");
    }
}
non-sealed class Kopek implements Hayvan {
    public void hareketEt() {
        System.out.println("Köpek koşuyor");
    }
}
class Kangal extends Kopek {   // serbest
    public void hareketEt() {
        System.out.println("Kangal sürüyü koruyor");
    }
}
```



Şekil 4.5.16: Mühürlü `Hayvan` arayüzü yalnızca `Kedi` (`final`) ve `Kopek` (`non-sealed`) tiplerine izin verir. `Kangal` , `Kopek` 'i serbestçe genişletir; `Kus` 'a izin yoktur.

`Kedi` `final` olduğu için hiyerarşinin o kolu kapanmıştır. `Kopek` `non-sealed` 'dir; bu yüzden `Kangal` hiçbir izin gerekmeden `Kopek` 'i genişletir (Şekil 4.5.16). `Kangal` dolaylı olarak bir `Hayvan` 'dır, ama `Hayvan` 'ın doğrudan alt tipi değildir; bu yüzden `permits` listesinde yer alması gerekmez.

Mühürlü tipin derleyiciye kazandırdığı bilgi, `switch` ile birlikte kullanılınca görülür. Java 21'de `switch` bir nesnenin tipine göre dallanabilir ve derleyici bütün alt tiplerin ele alınıp alınmadığını denetler:

```
// Java 21: switch tüm alt türleri kapsar,
// default dalı gerekmez
static String tur(Hayvan h) {
    return switch (h) {
        case Kedi k -> "kedi";
        case Kopek k -> "köpek";
    };
}
// main içinde:
Hayvan[] hayvanlar =
    { new Kedi(), new Kopek(), new Kangal() };
for (Hayvan h : hayvanlar) {
    h.hareketEt();
    System.out.println(tur(h));
}
```

```
Kedi sessizce yürüyor
kedi
Köpek koşuyor
köpek
Kangal sürüyü koruyor
köpek
```

`tur` metodundaki `switch`'in `default` dalı yoktur. Derleyici `Hayvan`'ın yalnızca `Kedi` ve `Kopek` olabileceğini bilir; bu yüzden iki dalın her durumu kapsadığını kanıtlayabilir. `case Kedi k` dalı, nesne bir `Kedi` ise onu `k` adıyla yakalar. Döngüdeki üçüncü nesne bir `Kangal`'dır; `hareketEt()` kendi sürümünü çalıştırır, `switch` ise onu bir `Kopek` olarak tanır ve "köpek" döndürür.

Bu denetimin değeri, hiyerarşi değiştiğinde ortaya çıkar. `permits` listesine yeni bir alt tip eklenirse onu ele almayan her `switch` derlenmez hâle gelir. Aşağıdaki `switch` yalnızca `Kedi`'yi ele alır:

```
sealed interface Hayvan permits Kedi, Kopek { }
final class Kedi implements Hayvan { }
final class Kopek implements Hayvan { }
public class SwitchEksik {
    static String tur(Hayvan h) {
        return switch (h) {
            case Kedi k -> "kedi";
        };
    }
}
```

```
SwitchEksik.java:6: error: the switch expression does not cover all possible input values
        return switch (h) {
               ^
1 error
```

Bir `default` dalı yazılsaydı bu yardım kaybolurdu: yeni tip sessizce `default`'a düşerdi. Mühürlü hiyerarşide `default` yazmamak bu yüzden bilinçli bir seçimdir.

Mühürleme kurallarının ihlali derleme hatasıdır. İzin listesinde olmayan bir sınıf mühürlü tipi uygulayamaz:

```
sealed interface Hayvan permits Kedi, Kopek { }
final class Kedi implements Hayvan { }
final class Kopek implements Hayvan { }
final class Kus implements Hayvan { }
public class MuhurIzinsiz { }
```

```
MuhurIzinsiz.java:4: error: class is not allowed to extend sealed class: Hayvan (as it is
not listed in its 'permits' clause)
final class Kus implements Hayvan { }
      ^
1 error
```

İletideki "sealed class" ifadesi arayüz için de kullanılır. İzinli bir alt tip üç belirteçten birini seçmezse de derleyici durur:

```
sealed interface Hayvan permits Kedi, Kopek { }
class Kedi implements Hayvan { }
final class Kopek implements Hayvan { }
public class MuhurBelirtecsiz { }
```

```
MuhurBelirtecsiz.java:2: error: sealed, non-sealed or final modifiers expected
class Kedi implements Hayvan { }
^
1 error
```

Bu hatanın üç düzeltmesi, üç farklı tasarım kararıdır. `final class Kedi` hiyerarşiyi orada bitirir. `sealed class Kedi permits Tekir` yeniden kısıtlar; o zaman `Tekir` de bir belirteç seçmelidir. `non-sealed class Kedi` ise `Kedi`'yi herkese açar.

4.5.7 Arayüz mü, soyut sınıf mı?

Soyut sınıf ve arayüz birbirine benzer: ikisinden de nesne oluşturulamaz, ikisi de referans tipi olarak kullanılır, ikisi de alt tiplere metot yazdırır. Java 8'den sonra arayüzler de gövdeli metot taşıdığı için fark daha da incelmiştir. Tablo 4.5.11 iki aracı yan yana koyar.

Özellik	Arayüz	Soyut sınıf
Tanım / kullanım	`interface` · `implements`	`abstract class` · `extends`
Metotlar	Soyut; Java 8+ varsayılan ve `static`, Java 9+ `private`	Hem soyut hem somut metotlar
Alanlar	Yalnızca `public static final` sabitler; örnek alanı olamaz	Her türlü alan (örnek alanları dahil)
Yapıcı	Yok	Olabilir; alt sınıf `super(…)` ile çağırır
Erişim	Üyeler örtük `public` (`private` yardımcılar hariç)	`public`, `protected`, `private`, paket
Kalıtım	Bir sınıf çok sayıda arayüz uygular; arayüz birden çok arayüzü genişletir	Bir sınıf yalnızca tek bir sınıftan türetilir
Amaç	İlgisiz sınıflara ortak yetenek ("yapabilir")	Ortak alanlı yakın sınıfları gruplamak ("bir türüdür")

Özellik	Arayüz	Soyut sınıf
Nesne	İkisinden de `new` ile nesne oluşturulamaz; ikisi de referans tipi olarak kullanılır	

Tablo 4.5.11: Arayüz ve soyut sınıfın karşılaştırması.

Tablodaki en belirleyici satırlar "Alanlar" ve "Kalıtım" satırlarıdır. Soyut sınıf durum taşır ama tek ebeveynlidir. Arayüz durum taşımaz ama bir sınıf istediği kadar arayüz alabilir. Seçim çoğu zaman bu iki ölçüte göre yapılır.

Soyut sınıf şu durumlarda uygundur:

- Alt sınıflar yakın akrabadır: `Kare` ve `Cember` birer `Sekil`'dir.
- Ortak alanlar ve yapıcı gerekir (`isim` gibi).
- Ortak kodun bir kısmı hazırdır (şablon metot gibi).

Arayüz şu durumlarda uygundur:

- Aynı yetenek ilgisiz sınıflara verilecektir: `Kus`, `Ucak` ve `Drone`'un hepsi `Ucabilir`'dir.
- Bir sınıfa birden çok yetenek kazandırılacaktır.
- Kullanan kod ile uygulayan kod arasında bir sözleşme kurulacaktır. Böyle bir sözleşmeye API (*application programming interface*, uygulama programlama arayüzü) denir.

Çoğu zaman ikisi birlikte kullanılır:

```
abstract class Hayvan implements Sesli { … }
class Ordek extends Hayvan implements Ucabilir, Yuzebilir { … }
```

Bu yazımda `Hayvan` soyut sınıfı akrabalığı ve ortak durumu taşır; `Sesli`, `Ucabilir` ve `Yuzebilir` arayüzleri yetenekleri ekler. `Ordek` hem bir `Hayvan`'dır hem uçabilir hem yüzebilir.

Java'nın sınıflarda çoklu kalıtıma izin vermemesinin nedeni, iki üst sınıfın aynı alanı ya da metodu farklı tanımlaması durumunda hangisinin geçerli olacağının belirsizleşmesidir. Bu soruna, iki üst sınıfın ortak bir atadan geldiği çizimin biçimi yüzünden **elmas problemi** denir. Arayüzler durum taşımadığı için sorunun büyük kısmı ortadan kalkar. Kalan tek belirsizlik iki varsayılan metodun çakışmasıdır ve onu da derleyici sınıfa çözdürür.

Bir örnekle karar sürecini izleyelim. Plakası ve yapıcısı olan, soyut `int tekerSayisi()` metodu bulunan bir `Tasit` ile `KAPASITE = 100` sabiti ve `void sarjEt(int yuzde)` metodu olan bir `Sarj` tasarlanacak. `Tasit`'in alanı ve yapıcısı vardır; bu yüzden soyut sınıf olmalıdır. Şarj olma yeteneği ise elektrikli arabaya da, telefona da, matkaba da verilebilir; bu yüzden `Sarj` bir arayüzdür. Elektrikli araba ikisini birlikte kullanır:

```
abstract class Tasit {
  private final String plaka;
  Tasit(String plaka) { this.plaka = plaka; }
  abstract int tekerSayisi();
  String getPlaka() { return plaka; }
}
interface Sarj {
  int KAPASITE = 100;
  void sarjEt(int yuzde);
}
class ElektrikliAraba extends Tasit implements Sarj {
  private int doluluk = 20;
  ElektrikliAraba(String plaka) { super(plaka); }
  int tekerSayisi() { return 4; }
  public void sarjEt(int yuzde) {
    doluluk = Math.min(KAPASITE, doluluk + yuzde);
  }
  int getDoluluk() { return doluluk; }
}
public class Main {
  public static void main(String[] args) {
    ElektrikliAraba e = new ElektrikliAraba("28 ABC 61");
    e.sarjEt(50);
    System.out.println(e.getDoluluk());
    e.sarjEt(50);
    System.out.println(e.getDoluluk());
    Tasit t = e; Sarj s = e;
    System.out.println(t.tekerSayisi() + " " + (s == t));
  }
}
```

```
70
100
4 true
```

Doluluk %20'den başlar. İlk `sarjEt(50)` onu 70 yapar. İkinci çağrıda 70 + 50 = 120 hesaplanır ama `Math.min(KAPASITE, …)` sonucu kapasiteyle sınırlar: 100. `KAPASITE` sabiti arayüzden kalıtımla gelir ve sınıfın içinde adıyla kullanılır. Sınıf bildiriminde önce `extends`, sonra `implements` yazılmıştır ve `sarjEt` `public`'tir. Nesne hem `Tasit` hem `Sarj` tipindeki referanslara atanabilir; iki referans aynı nesneyi gösterdiği için `s == t` `true` verir.

4.5.8 Jenerikler

Arayüzler ve soyut sınıflar, bir kodun farklı *sınıflarla* çalışmasını sağlar. Jenerikler aynı fikri bir adım öteye taşır: bir sınıfın ya da metodun farklı *tiplerle* çalışmasını, üstelik tip güvenliğini kaybetmeden sağlar. `List<String>` ve `Function<String, Integer>` yazımlarındaki açılı parantezler jeneriklerin söz dizimidir.

4.5.8.1 Neden jenerikler?

Her tipte değer saklayabilen bir kutu sınıfı yazmak isteyelim. Jenerikler olmadan akla gelen tek yol, her şeyin üst sınıfı olan `Object`'i kullanmaktır:

```
class KutuObj {
    private Object icerik;
    void koy(Object o) { icerik = o; }
    Object al() { return icerik; }
}
KutuObj k = new KutuObj();
k.koy("Merhaba");
String s = (String) k.al(); // dönüşüm şart
k.koy(42);                  // itiraz yok
String t = (String) k.al(); // çalışırken hata
```

```
Exception in thread "main" java.lang.ClassCastException: class java.lang.Integer cannot be
cast to class java.lang.String (java.lang.Integer and java.lang.String are in module
java.base of loader 'bootstrap')
        at Main.main(Main.java:12)
```

Bu tasarımın iki kusuru vardır. Birincisi, kutudan değer okurken her seferinde açık dönüşüm gerekir: `(String) k.al()`. Derleyici kutunun içinde ne olduğunu bilmez; programcının sözüne güvenir. İkincisi, yanlış tipte bir değer konulduğunda kimse itiraz etmez. `k.koy(42)` derlenir, çünkü 42 kutulanıp bir `Integer` nesnesi olur ve her nesne bir `Object`'tir. Hata ancak program çalışırken, 42'yi `String`'e dönüştürmeye çalışan satırda ortaya çıkar: `ClassCastException`. Hatanın yapıldığı yer ile fark edildiği yer birbirinden uzaktır.

```
class Kutu<T> {
    private T icerik;
    void koy(T x) { icerik = x; }
    T al() { return icerik; }
}
Kutu<String> k = new Kutu<>();
k.koy("Merhaba");
String s = k.al();          // dönüşüm yok
k.koy(42);                  // derlenmez
```

```
JenerikKoy.java:10: error: incompatible types: int cannot be converted to String
        k.koy(42);
              ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
1 error
```

Jenerik sürümde kutunun tipi bir parametredir: `Kutu<T>`. `Kutu<String>` yazıldığında `T`'nin yerine `String` geçer. `koy` artık yalnızca `String` kabul eder, `al` da `String` döndürür. Okurken dönüşüm gerekmez. Yanlış tipteki `k.koy(42)` ise program çalışmadan, derlenirken reddedilir.

Jeneriklerin iki kazancı böylece özetlenir. **Tip güvenliği**: yanlış tipteki değer çalışırken değil, derlerken yakalanır. **Yeniden kullanım**: tek bir gerçekleme her tip için çalışır ve okurken açık dönüşüm gerekmez.

4.5.8.2 Jenerik sınıf ve jenerik metot

Tanım 4.5.10

Jenerikler (*generics*), sınıf ve metotların bir ya da birkaç **tip parametresiyle** yazılmasıdır. Tip parametresi (`T`), sınıf ya da metot kullanılırken verilen **tip argümanıyla** (`String`, `Integer` ...) yer değiştiren bir yer tutucudur.

```
class Kutu<T> {                    // T: tip parametresi
    private T icerik;
    void koy(T x) { icerik = x; }
    T al() { return icerik; }
}
static <T> void yazdir(T oge) {    // jenerik metot
    System.out.println("Öğe: " + oge);
}
// main içinde:
Kutu<String> k1 = new Kutu<>();
k1.koy("Merhaba");
Kutu<Integer> k2 = new Kutu<>();
k2.koy(42);
int n = k2.al() + 1;               // kutudan çıkarma
System.out.println(k1.al() + " " + n);
yazdir(3.14);
yazdir('A');
```

```
Merhaba 43
Öğe: 3.14
Öğe: A
```

`Kutu<String> k1` ve `Kutu<Integer> k2` aynı sınıfın iki farklı kullanımıdır. `k2.al()` bir `Integer` döndürür; `+ 1` işleminde bu değer kutudan çıkarılır ve sonuç 43 olur. `new Kutu<>()` yazımındaki boş açılı parantez çiftine **elmas** (*diamond*) denir; derleyici tipi sol taraftaki bildirimden çıkarır.

`yazdir` bir **jenerik metottur**. Tip parametresini dönüş tipinden önce bildirir: `static <T> void yazdir(T oge)`. Çağrılırken tip yazılmaz; derleyici onu argümandan çıkarır. `yazdir(3.14)`'te `T` `Double`, `yazdir('A')`'da `Character`'dır.

Tip argümanı bir referans tipi olmalıdır. `Kutu<int>` derlenmez; ilkel tipler için sarmalayıcı sınıflar kullanılır (`Kutu<Integer>`):

```
JenerikInt.java:6: error: unexpected type
        Kutu<int> k = new Kutu<>();
             ^
  required: reference
  found:    int
1 error
```

Tip parametresi adları geleneksel olarak tek büyük harftir: `T` tip, `E` öğe (*element*), `K` anahtar (*key*), `V` değer (*value*), `R` sonuç (*result*). `Function<T, R>` adındaki iki harf, girdi tipini ve sonuç tipini anlatır.

4.5.8.3 Sınırlı tip parametresi

Yalın bir `T` hakkında derleyici hiçbir şey bilmez; `T` tipindeki bir değer üzerinde ancak `Object`'in metotları çağrılabilir. Bir sayının iki katını hesaplayan jenerik bir metot yazmak istersek `T`'nin bir sayı olduğunu söylememiz gerekir.

> **Tanım 4.5.11**
>
> **Sınırlı tip parametresi** (*bounded type parameter*), `<T extends Ust>` biçiminde yazılan tip parametresidir. `T` yalnızca `Ust` ya da onun bir alt tipi olabilir. `Ust`'e **üst sınır** denir. Karşılığında metot gövdesi `Ust`'ün metotlarını kullanabilir.

```
static <T extends Number> double ikiKati(T x) {
    return x.doubleValue() * 2;   // Number'ın metodu
}
// main içinde:
System.out.println(ikiKati(21));    // T = Integer
System.out.println(ikiKati(1.5));   // T = Double
System.out.println(ikiKati("on")); // derlenmez
```

```
42.0
3.0
```

`<T extends Number>` sınırı sayesinde `x.doubleValue()` çağrısı derlenir; bu metot `Number` sınıfındadır ve `Integer`, `Double` gibi bütün alt sınıflarda vardır. 21 için `T = Integer`, 1.5 için `T = Double` çıkarılır ve sonuçlar 42.0 ile 3.0'dır. Üçüncü satırda `"on"` bir `String`'dir ve `Number`'ın alt tipi değildir; derleyici çağrıyı reddeder:

```
IkiKatiOn.java:6: error: method ikiKati in class IkiKatiOn cannot be applied to given
types;
        System.out.println(ikiKati("on"));
                           ^
  required: T
  found:    String
  reason: inference variable T has incompatible bounds
    upper bounds: Number
    lower bounds: String
  where T is a type-variable:
    T extends Number declared in method <T>ikiKati(T)
1 error
```

İletinin son kısmı derleyicinin akıl yürütmesini gösterir: `T` için üst sınır `Number`, argümandan gelen alt sınır ise `String`'dir. İkisini birden sağlayan bir tip yoktur.

Arayüz de sınır olabilir; sınır bir arayüz olsa da `implements` değil `extends` yazılır. Aşağıdaki metot her karşılaştırılabilir tipte (`Integer`, `String` …) en büyük öğeyi bulur. `T extends Comparable<T>` sınırı, `T` tipindeki her değerin başka bir `T` ile `compareTo` metoduyla karşılaştırılabileceğini garanti eder.

```
static <T extends Comparable<T>> T enBuyuk(T[] dizi) {
    T eb = dizi[0];
    for (int i = 1; i < dizi.length; i++) {
        if (dizi[i].compareTo(eb) > 0) {
            eb = dizi[i];
        }
    }
    return eb;
}
// main içinde:
Integer[] sayilar = {7, 12, 5};
String[] adlar = {"elma", "kiraz", "armut"};
System.out.println(enBuyuk(sayilar));
System.out.println(enBuyuk(adlar));
```

```
12
kiraz
1 -1 6 -10
```

`compareTo` sözleşmesi şöyledir: `a.compareTo(b)` negatifse `a` küçüktür, sıfırsa eşittir, pozitifse büyüktür. Çıktının üçüncü satırı, izlemede kullanılan dört `compareTo` değerini doğrulamak için programa eklenmiştir.

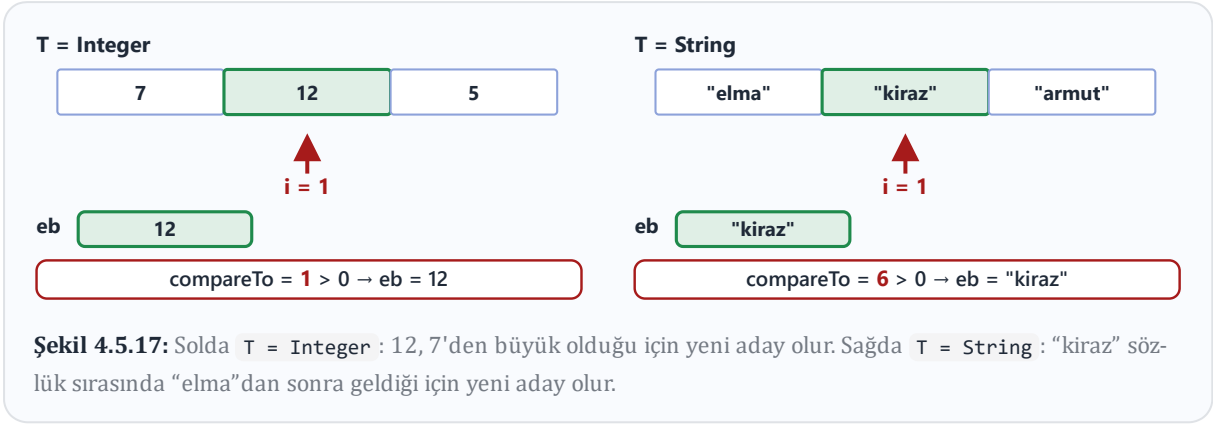
Adım	Satır	Değişkenler	Yeni çıktı
1	11: `Integer[] sayilar = {7, 12, 5};` 12: `String[] adlar = {"elma", "kiraz", "armut"};`	sayilar=[7, 12, 5], adlar=[elma, kiraz, armut]	—
2	13: `System.out.println(enBuyuk(sayilar));` 1: `static <T extends Comparable<T>> T enBuyuk(T[] dizi) {` 2: `T eb = dizi[0];`	T=Integer, eb=7	—
3	3: `for (int i = 1; i < dizi.length; i++) {` 4: `if (dizi[i].compareTo(eb) > 0) {` 5: `eb = dizi[i];`	T=Integer, i=1, compareTo=1, eb=12	—
4	3: `for (int i = 1; i < dizi.length; i++) {` 4: `if (dizi[i].compareTo(eb) > 0) {`	T=Integer, i=2, compareTo=-1, eb=12	—
5	8: `return eb;` 13: `System.out.println(enBuyuk(sayilar));`	T=Integer, eb=12	`12`
6	14: `System.out.println(enBuyuk(adlar));` 2: `T eb = dizi[0];`	T=String, eb="elma"	—

Adım	Satır	Değişkenler	Yeni çıktı
7	3: `for (int i = 1; i < dizi.length; i++) {` 4: `if (dizi[i].compareTo(eb) > 0) {` 5: `eb = dizi[i];`	T=String, i=1, compareTo=6, eb="kiraz"	—
8	3: `for (int i = 1; i < dizi.length; i++) {` 4: `if (dizi[i].compareTo(eb) > 0) {`	T=String, i=2, compareTo=-10, eb="kiraz"	—
9	8: `return eb;` 14: `System.out.println(enBuyuk(adlar));`	T=String, eb="kiraz"	`kiraz`

Tablo 4.5.12: `enBuyuk` metodunun iki tiple izlenmesi.

Tablo 4.5.12'nin birinci adımında iki dizi kurulur. İkisi de uygundur: `Integer` ve `String`, `Comparable`'ı kendi tipleriyle uygular. Dizideki sayılar otomatik olarak kutulanır. İkinci adımda derleyici `T`'yi `Integer` olarak çıkarmıştır. İlk öğe en büyük aday olarak alınır: `eb = 7`.

Üçüncü adımda `12.compareTo(7)` pozitiftir (1); 12 daha büyüktür ve `eb = 12` olur. Dördüncü adımda `5.compareTo(12)` negatiftir (−1); `eb` değişmez. Beşinci adımda döngü biter ve 12 döner. Dönüş tipi `T`, yani `Integer`'dır; dönüşüm gerekmez.



Şekil 4.5.17: Solda `T = Integer`: 12, 7'den büyük olduğu için yeni aday olur. Sağda `T = String`: "kiraz" sözlük sırasında "elma"dan sonra geldiği için yeni aday olur.

Altıncı adımda aynı metot bu kez `T = String` ile çağrılır. `compareTo` artık dizgileri sözlük sırasıyla karşılaştırır: iki dizginin ilk farklı karakterlerinin kod farkını döndürür. Yedinci adımda "kiraz" ile "elma"nın ilk harfleri farklıdır: 'k' − 'e' = 107 − 101 = 6. Sonuç pozitiftir ve `eb = "kiraz"` olur (Şekil 4.5.17). Sekizinci adımda "armut" ile "kiraz" karşılaştırılır: 'a' − 'k' = 97 − 107 = −10; `eb` değişmez. Son adımda "kiraz" döner.

Tek bir gerçekleme iki farklı tipte doğru çalışmıştır. Üst sınır sayesinde `compareTo` çağrısı derleme zamanında güvence altındadır: `compareTo`'su olmayan bir tipin dizisiyle `enBuyuk` çağrılırsa program derlenmez. Sözlük sırasının harf kodlarına dayandığı unutulmamalıdır. Büyük harflerin kodları küçük harflerinkinden küçüktür; "Zeytin", "armut"tan önce gelir.

4.5.8.4 Joker tipler ve değişmezlik

`Integer`, `Number`'ın alt tipidir; bir `Integer` her yerde `Number` yerine kullanılabilir. O hâlde `List<Integer>` da `List<Number>`'ın alt tipi midir? İlk akla gelen cevap "evet"tir; Java'nın cevabı ise "hayır"dır:

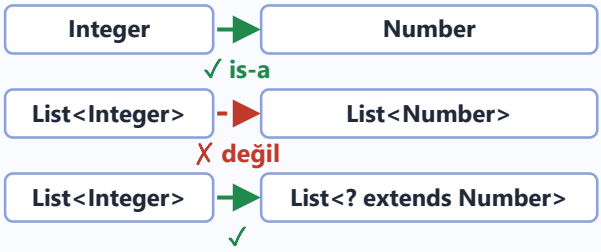
```
import java.util.*;
public class Degismezlik {
    public static void main(String[] args) {
        List<Integer> tamlar = Arrays.asList(1, 2, 3);
        List<Number> x = tamlar;
    }
}
```

```
Degismezlik.java:5: error: incompatible types: List<Integer> cannot be converted to
List<Number>
        List<Number> x = tamlar;
                         ^
1 error
```

Tanım 4.5.12

Değişmezlik (*invariance*): `A`, `B`'nin alt tipi olsa bile `List<A>`, `List<B>`'nin alt tipi değildir. Jenerik tipler, tip argümanları arasındaki alt tip ilişkisini kendiliğinden devralmaz.

Kuralın nedeni yazma işlemidir. `List<Integer>` bir `List<Number>` değişkenine atanabilseydi, o değişken üzerinden listeye 2.5 eklenebilirdi; çünkü 2.5 bir `Number`'dır. Ama liste aslında tam sayı listesidir ve içine bir `Double` girmiş olurdu. Listeden okuyan ve `Integer` bekleyen kod çalışırken çökerdi. Java bu kapıyı derleme zamanında kapatır (Şekil 4.5.18).



Şekil 4.5.18: `Integer`, `Number`'ın alt tipidir; `List<Integer>` ise `List<Number>`'ın alt tipi değildir. Bağlantı joker tiple kurulur: `List<Integer>`, `List<? extends Number>`'a atanabilir.

Değişmezlik güvenlidir ama esnekliği azaltır. Bir sayı listesindeki değerleri toplayan bir metot `List<Number>` parametresi alırsa ona `List<Integer>` verilemez. Bu sorunu **joker tipler** (*wildcards*) çözer:

- `?` : bilinmeyen herhangi bir tip.
- `? extends T` : `T` ya da onun bir alt tipi. Böyle bir liste değer *üretir*; ondan `T` olarak okunur.
- `? super T` : `T` ya da onun bir üst tipi. Böyle bir liste değer *tüketir*; içine `T` yazılır.

```
static double topla(List<? extends Number> liste) {
    double t = 0;
    for (Number n : liste) t += n.doubleValue();  // okuma
    return t;
}
static void doldur(List<? super Integer> liste) {
    liste.add(1);                                 // yazma
    liste.add(2);
}
// main içinde:
List<Integer> tamlar = Arrays.asList(1, 2, 3);
List<Number> sayilar = new ArrayList<>();
doldur(sayilar);
sayilar.add(2.5);
System.out.println(topla(tamlar) + " " + sayilar);
// List<Number> x = tamlar;   derlenmez
```

```
6.0 [1, 2, 2.5]
5.5
```

`topla` listeden yalnızca okur; parametresi `List<? extends Number>` 'dır. Bu yüzden `List<Integer>` da `List<Double>` da `List<Number>` da verilebilir. Okunan her öğe en azından bir `Number` 'dır ve `doubleValue()` çağrılabilir. 1 + 2 + 3 = 6.0 bulunur. `doldur` ise listeye yalnızca yazar; parametresi `List<? super Integer>` 'dır. Hangi liste gelirse gelsin bir `Integer` alabilir. Ona verilen `List<Number>` 'a 1 ve 2 eklenir; programın kendisi de 2.5 ekler ve liste `[1, 2, 2.5]` olur. Çalıştırılan programa eklenen son satır, aynı `topla` 'nın bu karışık listeyle de çalıştığını gösterir: 1 + 2 + 2.5 = 5.5.

`? extends` ile işaretlenmiş bir listeye yazmaya çalışmak derleme hatasıdır. Derleyici listenin gerçek tipini bilmez; belki bir `List<Double>` 'dır ve içine 1 konamaz:

```
import java.util.*;
public class ExtendsYaz {
    static void ekle(List<? extends Number> liste) {
        liste.add(1);
    }
}
```

```
ExtendsYaz.java:4: error: incompatible types: int cannot be converted to CAP#1
        liste.add(1);
                  ^
  where CAP#1 is a fresh type-variable:
    CAP#1 extends Number from capture of ? extends Number
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
1 error
```

İletideki `CAP#1`, derleyicinin bilinmeyen tip için uydurduğu addır. Hangi jokerin seçileceğini hatırlatan kısa bir kural vardır: PECS (*Producer Extends, Consumer Super*). Değer üreten (okunan) yapı `extends`, değer tüketen (yazılan) yapı `super` ile yazılır.

4.5.8.5 Tip silme

Jenerik tip bilgisi çoğunlukla derleyici içindir. Derleyici tip argümanlarını denetler, gereken dönüşümleri ekler ve sonra bu bilgiyi siler. Üretilen kodda `T`'nin yerinde `Object` (ya da sınırlı parametrede sınır tipi) kalır.

Tanım 4.5.13

Tip silme (*type erasure*), jenerik tip argümanlarının derleme sırasında denetlendikten sonra üretilen bayt kodundan çıkarılmasıdır. Çalışma zamanında `Kutu<String>` ile `Kutu<Integer>` aynı sınıftır.



Şekil 4.5.19: Kaynak kodda `Kutu<String>` ve dönüşümsüz okuma vardır. javac tipleri denetler ve siler; bayt kodda `Kutu` ile derleyicinin eklediği `(String)` dönüşümü kalır.

```
class Kutu<T> {
    private T icerik;
    // T yeni() { return new T(); }   derlenmez
}
// main içinde:
Kutu<String> a = new Kutu<>();
Kutu<Integer> b = new Kutu<>();
System.out.println(a.getClass() == b.getClass());
System.out.println(a.getClass().getName());
```

```
true
Kutu
```

İki kutunun `getClass()` sonucu aynı nesnedir: `true`. Sınıfın adı da yalnızca `Kutu`'dur; tip argümanı adın parçası değildir. Şekil 4.5.19 derleyicinin yaptığı dönüşümü gösterir: programcının yazmadığı `(String)` dönüşümünü bayt koda derleyici ekler. Tip denetimi geçtiği için bu dönüşüm hiçbir zaman başarısız olmaz.

Tip silmenin bir sonucu, `T`'nin çalışma zamanında bilinmemesidir. Bu yüzden `new T()` yazılamaz; JVM hangi sınıfın nesnesini oluşturacağını bilemez:

```
NewT.java:3: error: unexpected type
    T yeni() { return new T(); }
                        ^
  required: class
  found:    type parameter T
  where T is a type-variable:
    T extends Object declared in class Kutu
1 error
```

Aynı nedenle `new T[10]` ile jenerik dizi de oluşturulamaz. İleti bu kez farklıdır:

```
NewTDizi.java:2: error: generic array creation
    T[] dizi = new T[10];
               ^
1 error
```

4.5.9 Etiketler

Koda yalnızca derleyicinin ya da bir aracın okuyacağı bilgiler eklemek bazen işe yarar: “Bu metot üst tipteki bir metodu geçersiz kılıyor”, “Bu sınıf artık kullanılmamalı”, “Bu metot bir testtir.” Java bu tür bilgiler için etiketleri kullanır.

Tanım 4.5.14

Etiket (*annotation*), sınıf, metot, alan ya da parametreye `@` ile eklenen üst veridir. Etiket kendi başına davranış değiştirmez. Onu derleyici, araçlar ya da çalışan program (yansıma ile) okur ve ona göre davranır.

Etiket	Kim okur?	Ne yapar?
`@Override`	derleyici	Üst tipte böyle bir metot yoksa hata verir
`@Deprecated`	derleyici	Eskimiş öğe; onu kullanan kod uyarı alır
`@SuppressWarnings("unchecked")`	derleyici	Belirtilen uyarıyı susturur
`@FunctionalInterface`	derleyici	Arayüzde yalnızca bir soyut metot olduğunu denetler
`@Test` (JUnit)	test aracı	Test metotlarını bulur ve çalıştırır

Tablo 4.5.13: Sık kullanılan etiketler.

Tablodaki etiketlerin dördünü derleyici okur. `@Override` bu notta birkaç kez işe yaradı: imzası yanlış yazılan bir metot, etiket sayesinde hatanın gerçek nedeniyle birlikte yakalanır. `@FunctionalInterface` 'in denetimi ise şöyle görünür:

```
@FunctionalInterface
interface Islem {
    int uygula(int a, int b);
    int ters(int a);   // 2. soyut metot
}
```

```
FonkIki.java:1: error: Unexpected @FunctionalInterface annotation
@FunctionalInterface
^
  Islem is not a functional interface
    multiple non-overriding abstract methods found in interface Islem
1 error
```

Etiket yazılmasaydı arayüz sorunsuz derlenirdi; hata ancak biri bu arayüze bir lambda atamaya çalıştığında, başka bir dosyada ortaya çıkardı. Etiket, niyeti koda yazar ve derleyiciye denetletir.

Programcı kendi etiketini de tanımlayabilir. Etiket `@interface` ile tanımlanır; öğeleri metot gibi yazılır ve `default` ile varsayılan değer alabilir. Aşağıdaki `@Gorev` etiketi bir metodun yazarını ve önceliğini kaydeder. Program, çalışırken kendi sınıfını inceleyip etiketli metotları bulur ve çağırır:

```
@Retention(RetentionPolicy.RUNTIME)  // çalışırken okunur
@Target(ElementType.METHOD)          // yalnızca metotlara
@interface Gorev {
    String yazar();
    int oncelik() default 1;
}
class Isler {
    @Gorev(yazar = "Ayşe", oncelik = 3)
    public void yedekle() { System.out.println("Yedek alındı"); }
    public void temizle() { System.out.println("Temizlendi"); }
}
// main içinde (yansıma, java.lang.reflect.Method):
Isler isler = new Isler();
for (Method m : Isler.class.getDeclaredMethods()) {
    Gorev g = m.getAnnotation(Gorev.class);
    if (g != null) {
        System.out.println(m.getName() + ", "
            + g.yazar() + ", " + g.oncelik());
        m.invoke(isler);
    }
}
```

```
yedekle, Ayşe, 3
Yedek alındı
```

Etiketin üstündeki iki etiket, etiketin kendisi hakkında bilgi verir. `@Target(ElementType.METHOD)` etiketin yalnızca metotlara yazılabileceğini söyler. `@Retention(RetentionPolicy.RUNTIME)` ise etiketin çalışma zamanına kadar saklanacağını belirtir. Döngü `Isler` sınıfının bütün metotları üzerinde gezinir. `getAnnotation(Gorev.class)` metotta `@Gorev` yoksa `null` döndürür. Yalnızca `yedekle` etiketlidir; onun adı, yazarı ve önceliği yazdırılır, sonra `m.invoke(isler)` metodu çağırır ve "Yedek alındı" yazılır. `temizle` etiketsiz olduğu için hiç çağrılmaz.

`@Retention`	**Etiket nerede kalır?**
`SOURCE`	Yalnızca kaynak kodda (`@Override` gibi); derleyici okuduktan sonra atılır
`CLASS`	`.class` dosyasında; yansıma göremez (varsayılan değer)
`RUNTIME`	Çalışırken yansımayla okunur

Tablo 4.5.14: `@Retention` değerleri.

Tablo 4.5.14'teki farkı denemek kolaydır. Aynı program etiketin `@Retention` değeri `CLASS` yapılarak çalıştırılırsa etiket `.class` dosyasında durur ama yansıma onu göremez:

```
import java.lang.annotation.*;
import java.lang.reflect.Method;
@Retention(RetentionPolicy.CLASS)
@Target(ElementType.METHOD)
@interface Gorev {
    String yazar();
    int oncelik() default 1;
}
class Isler {
    @Gorev(yazar = "Ayşe", oncelik = 3)
    public void yedekle() { System.out.println("Yedek alındı"); }
}
public class Main {
    public static void main(String[] args) throws Exception {
        int bulunan = 0;
        for (Method m : Isler.class.getDeclaredMethods()) {
            if (m.getAnnotation(Gorev.class) != null) bulunan++;
        }
        System.out.println("bulunan etiket: " + bulunan);
    }
}
```

```
bulunan etiket: 0
```

Tanım 4.5.15

Yansıma (*reflection*), programın kendi sınıflarını, metotlarını, alanlarını ve etiketlerini çalışırken incelemesi ve kullanmasıdır. `java.lang.reflect` paketindeki `Method`, `Field` gibi sınıflarla yapılır.

Yansıma güçlü bir araçtır; test araçları ve büyük uygulama çerçeveleri etiketli metotları bu yolla bulur. Bedeli de vardır: `m.invoke(...)` gibi çağrılar derleyicinin denetiminden geçmez, yanlış bir çağrı ancak çalışırken fark edilir, kod da daha yavaş çalışır. Bu yüzden yansıma sıradan uygulama kodunda değil, gerçekten gerektiği yerlerde kullanılır.

4.5.10 Tasarım desenleri

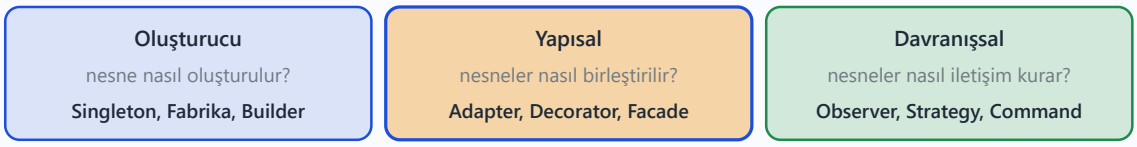
Arayüzler, soyut sınıflar ve lambdalar birer dil aracıdır. Bu araçların hangi problemde nasıl bir araya getirileceği ise ayrı bir bilgidir. Deneyimli programcılar aynı tasarım sorunlarıyla tekrar tekrar karşılaşmış ve işe yarayan çözümlere ad vermişlerdir. Aşağıdaki beş desenin hepsi küçük konsol programlarıyla anlatılmıştır; dördü doğrudan bir arayüze dayanır.

4.5.10.1 Desen nedir?

Tanım 4.5.16

Tasarım deseni (*design pattern*), sık karşılaşılan bir tasarım sorununa denenmiş ve yeniden kullanılabilir bir çözüm kalıbıdır. Desen kopyalanacak hazır bir kod değildir; sorunu, çözümü ve çözümün ödünleşimlerini adlandıran ortak bir sözlüktür.

Desenlerin ortak amacı sorumlulukları ayırmak ve sınıflar arasındaki bağımlılığı azaltmaktır. Bağımlılığı az olan bir parça, ötekilere dokunmadan değiştirilebilir ve tek başına denenebilir. Desenler geleneksel olarak üç gruba ayrılır (Şekil 4.5.20). **Oluşturucu** desenler nesnenin nasıl oluşturulacağıyla, **yapısal** desenler nesnelerin nasıl birleştirileceğiyle ilgilenir. **Davranışsal** desenler ise nesnelerin iş birliğini düzenler.



Şekil 4.5.20: Tasarım desenlerinin üç grubu ve her gruptan örnekler. Tekil ve Fabrika oluşturucu, Gözlemci ve Strateji davranışsal desenlerdir; MVC ise bir mimari kalıptır.

Desen ancak gerçek bir sorunu çözüyorsa yararlıdır. Gereksiz yere eklenen desen programı karmaşıklaştırır. Desenler, "bileşenler arasında az bağımlılık" ve "kalıtım yerine birleşim" gibi daha genel tasarım ilkelerinin somut uygulamalarıdır. Birleşim (*composition*), bir sınıfın davranışını kalıtımla devralmak yerine başka bir nesneyi alan olarak tutup işi ona yaptırmasıdır.

4.5.10.2 Tekil (Singleton)

Bazı sınıfların programda tek bir nesnesi olmalıdır. Uygulamanın ayarları, bir günlük dosyası ya da bir donanım sürücüsü iki kopya hâlinde tutulursa kopyalar birbirinden farklılaşabilir. Tekil deseni sınıfın yalnızca bir nesnesi olmasını ve o nesneye tek bir yerden erişilmesini sağlar.

```
public final class Ayarlar {
  private static final Ayarlar TEK = new Ayarlar();
  private Ayarlar() {
    System.out.println("Ayarlar oluşturuldu");
  }
  public static Ayarlar getInstance() { return TEK; }
}
// main:
System.out.println("main başladı");
Ayarlar a = Ayarlar.getInstance();
Ayarlar b = Ayarlar.getInstance();
System.out.println(a == b);
```

```
main başladı
Ayarlar oluşturuldu
true
```

Desenin üç parçası vardır. Birincisi, yapıcı `private`'tır; sınıfın dışından `new Ayarlar()` yazılamaz:

```
TekilNew.java:8: error: Ayarlar() has private access in Ayarlar
    Ayarlar a = new Ayarlar();
                ^
1 error
```

İkincisi, tek nesne sınıfın içinde `private static final` bir alanda tutulur. Üçüncüsü, bu nesneye `public static` bir metotla (`getInstance()`) erişilir. İki çağrı aynı nesneyi döndürdüğü için `a == b` `true` verir.

Çıktının sırası da öğreticidir: “main başladı” iletisi “Ayarlar oluşturuldu”dan önce yazılmıştır. JVM `TEK` alanını, `Ayarlar` sınıfı ilk kullanıldığında, yani ilk `getInstance()` çağrısında bir kez ilklendirir. JVM sınıf ilklendirmesini yalnızca bir kez yaptığı için bu yöntem birden çok iş parçacığı olduğunda da güvenlidir.

Tekil desenin ciddi bir sakıncası vardır: gizli bir küresel değişken gibi davranır. `Ayarlar.getInstance()`'ı içeriden çağıran her sınıf, o tek nesneye görünmez biçimde bağlanır. Bu sınıfları denerken gerçek ayarların yerine başka bir nesne koymak zorlaşır. Çoğu zaman daha iyi bir yol, nesneyi ona ihtiyaç duyan sınıfın yapıcısına parametre olarak vermektir. Bağımlılığın dışarıdan verilmesine **bağımlılık enjeksiyonu** (*dependency injection*) denir.

4.5.10.3 Fabrika

Bir çizim programı, kullanıcının menüden seçtiği şekli oluşturacak olsun. Seçim bir dizgi olarak gelir: “kare”, “daire” … Her yerde `if` zinciriyle hangi sınıfın oluşturulacağına karar vermek, oluşturma kodunu programa dağıtır. Fabrika deseni nesne oluşturmayı tek bir yerde toplar.

```
interface Sekil { double alan(); }
class Kare implements Sekil { … }   // alan: a * a
class Daire implements Sekil { … }  // alan: π r²

class SekilFabrikasi {
  static Sekil olustur(String tur, double olcu) {
    switch (tur.toLowerCase(Locale.ROOT)) {
      case "kare":  return new Kare(olcu);
      case "daire": return new Daire(olcu);
      default:      return null;  // bilinmeyen tür
    }
  }
}
Sekil s = SekilFabrikasi.olustur("Kare", 3);
System.out.println(s.alan());   // 9.0
```

Tanım 4.5.17

Fabrika metodu (*factory method*), nesneyi üretip üst sınıf ya da arayüz tipinde döndüren metottur. Çağıran kod hangi somut sınıfın oluşturulduğunu bilmez.

`SekilFabrikasi.olustur` metodu türü ve ölçüyü alır, uygun nesneyi oluşturur ve onu `Sekil` arayüzü tipinde döndürür. Çağıran kod `Kare`'yi değil, yalnızca `Sekil`'i bilir. Kodda `…` ile kısaltılan sınıf gövdeleri, çalıştırılan programda tamamlanmıştır:

```
import java.util.Locale;
interface Sekil { double alan(); }
class Kare implements Sekil {
  private final double a;
  Kare(double a) { this.a = a; }
  public double alan() { return a * a; }
}
class Daire implements Sekil {
  private final double r;
  Daire(double r) { this.r = r; }
  public double alan() { return Math.PI * r * r; }
}
class SekilFabrikasi {
  static Sekil olustur(String tur, double olcu) {
    switch (tur.toLowerCase(Locale.ROOT)) {
      case "kare":  return new Kare(olcu);
      case "daire": return new Daire(olcu);
      default:      return null;  // bilinmeyen tür
    }
  }
}
public class Main {
  public static void main(String[] args) {
    Sekil s = SekilFabrikasi.olustur("Kare", 3);
    System.out.println(s.alan());   // 9.0
    Sekil u = SekilFabrikasi.olustur("üçgen", 2);
    System.out.println(u == null);   // true
    System.out.println(SekilFabrikasi.olustur("DAIRE", 1).alan());
    System.out.println("DAIRE".toLowerCase(Locale.forLanguageTag("tr-TR")) + " " +
"DAIRE".toLowerCase(Locale.ROOT));
  }
}
```

```
9.0
true
3.141592653589793
daıre daire
```

“Kare” ve 3 için alan 9.0'dır. Yeni bir şekil eklemek yalnızca fabrikayı ve yeni sınıfı değiştirir; şekil isteyen kodlar aynı kalır. Bilinmeyen bir tür için fabrika `null` döndürür ve çağıran kod sonucu denetler: “üçgen” için `u == null` `true` verir.

`toLowerCase(Locale.ROOT)` ayrıntısı Türkçe kullanıcılar için önemlidir. `toLowerCase()` parametresiz çağrılırsa bilgisayarın yerel ayarına göre çalışır. Türkçe yerel ayarda büyük “I” harfinin küçüğü noktasız “ı”dır; “DAIRE” “daıre” olur ve `"daire"` dalıyla eşleşmez. Programın son satırı bu farkı gös-

terir: Türkçe kurallarla “daıre”, dilden bağımsız `Locale.ROOT` ile “daire” elde edilir. Fabrika `Locale.ROOT` kullandığı için “DAIRE” isteği doğru şekli üretir ve alanı π · 1 · 1 = 3.141592653589793 yazdırılır.

Fabrika her yerde gerekmez. Zaten basit oluşturulan nesneler için ayrı bir fabrika yazmak gereksiz karmaşıklıktır; `new Nokta(3, 4)` kadar açık bir ifadeyi bir metodun arkasına saklamak okuru yalnızca yorar.

4.5.10.4 Gözlemci

Bir hava istasyonu sıcaklığı ölçüyor. Yeni bir ölçüm geldiğinde ekran onu göstermeli, bir uyarı modülü sıcaklık 30 dereceyi aşınca uyarmalı, bir kayıtçı ölçümü saklamalıdır. İstasyon bu üç parçayı doğrudan çağırırsa her yeni parça istasyonun kodunu değiştirir. Gözlemci deseni bu bağımlılığı tersine çevirir.

Tanım 4.5.18

Gözlemci deseni (*observer*), bir **konu** nesnesi ile ona kayıtlı **gözlemciler** arasında bire çok ilişki kurar. Konunun durumu değişince kayıtlı bütün gözlemcilere haber verilir. Konu, gözlemcilerin somut sınıfını bilmez; yalnızca ortak arayüzü çağırır.

```
interface Gozlemci { void guncelle(int derece); }

class Istasyon {
  private final List<Gozlemci> liste = new ArrayList<>();
  void ekle(Gozlemci g) { liste.add(g); }
  void setSicaklik(int d) {
    for (Gozlemci g : liste) g.guncelle(d);
  }
}
```



Şekil 4.5.21: İstasyon (konu), sıcaklık değişince kayıtlı her gözlemcinin `guncelle` metodunu çağırır; gözlemcilerin ne yaptığını bilmez.

`Gozlemci` tek metotlu bir arayüzdür. `Istasyon` gözlemcileri bir listede tutar. `ekle` yeni bir gözlemciyi kaydeder, `setSicaklik` ise listede gezinip her gözlemcinin `guncelle` metodunu çağırır. Arayüzün tek metodu olduğu için gözlemciler lambdayla da yazılabilir. Aşağıdaki programda iki lamb-

da gözlemci kaydedilir:

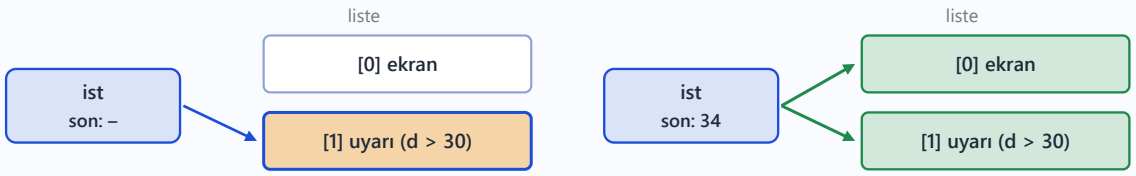
```
interface Gozlemci { void guncelle(int derece); }
class Istasyon {
  private final List<Gozlemci> liste = new ArrayList<>();
  void ekle(Gozlemci g) { liste.add(g); }
  void setSicaklik(int d) {
    for (Gozlemci g : liste) g.guncelle(d);
  }
}
// main:
Istasyon ist = new Istasyon();
ist.ekle(d -> System.out.println("Ekran: " + d + " °C"));
ist.ekle(d -> { if (d > 30) System.out.println("Uyarı: sıcak!"); });
ist.setSicaklik(25);
ist.setSicaklik(34);
```

```
Ekran: 25 °C
Ekran: 34 °C
Uyarı: sıcak!
```

Adım	Satır	Değişkenler	Yeni çıktı
1	10: `Istasyon ist = new Istasyon();`	ist=Istasyon, liste=[]	—
2	11: `ist.ekle(d -> System.out.println("Ekran: " + d + " °C"));` 4: `void ekle(Gozlemci g) { liste.add(g); }`	liste=[ekran]	—
3	12: `ist.ekle(d -> { if (d > 30) System.out.println("Uyarı: sıcak!"); });` 4: `void ekle(Gozlemci g) { liste.add(g); }`	liste=[ekran, uyarı]	—
4	13: `ist.setSicaklik(25);` 6: `for (Gozlemci g : liste) g.guncelle(d);`	d=25	—
5	6: `for (Gozlemci g : liste) g.guncelle(d);` 11: `ist.ekle(d -> System.out.println("Ekran: " + d + " °C"));`	d=25, g=ekran	`Ekran: 25 °C`
6	6: `for (Gozlemci g : liste) g.guncelle(d);` 12: `ist.ekle(d -> { if (d > 30) System.out.println("Uyarı: sıcak!"); });`	d=25, g=uyarı	—
7	14: `ist.setSicaklik(34);` 6: `for (Gozlemci g : liste) g.guncelle(d);`	d=34	—
8	6: `for (Gozlemci g : liste) g.guncelle(d);` 11: `ist.ekle(d -> System.out.println("Ekran: " + d + " °C"));`	d=34, g=ekran	`Ekran: 34 °C`
9	6: `for (Gozlemci g : liste) g.guncelle(d);` 12: `ist.ekle(d -> { if (d > 30) System.out.println("Uyarı: sıcak!"); });`	d=34, g=uyarı	`Uyarı: sıcak!`

Tablo 4.5.15: Gözlemci deseninin izlenmesi. “ekran” ve “uyarı”, kaydedilen iki lambda gözlemcinin adıdır.

Tablo 4.5.15'in birinci adımında konu nesnesi oluşur; gözlemci listesi boştur. İkinci adımda ilk gözlemci eklenir. Bu bir lambdadır ve `Gozlemci` arayüzünün tek metodunu uygular. Lambda listeye eklenir ama **çalışmaz**. Üçüncü adımda ikinci gözlemci eklenir; bu gözlemci yalnızca 30 derecenin üstünde konuşur.



Şekil 4.5.22: Solda iki gözlemci listeye kaydedilmiş, ama henüz hiçbiri çalışmamış. Sağda `setSicaklik(34)` listedeki ikinci gözlemciye ulaşmış; koşul doğru olduğu için uyarı yazılıyor.

Dördüncü adımda `setSicaklik(25)` çağrılır ve döngü listede baştan gezinir. Beşinci adımda ekran gözlemcisinin `guncelle(25)`'i çağrılır ve "Ekran: 25 °C" yazılır. Altıncı adımda uyarı gözlemcisi de haber alır, ama 25 > 30 yanlış olduğu için bir şey yazmaz. Yedinci adımda yeni ölçüm gelir: 34. Sekizinci adımda ekran yeni değeri yazar. Dokuzuncu adımda koşul bu kez doğrudur ve "Uyarı: sıcak!" yazılır (Şekil 4.5.22).

İstasyon, hiçbir gözlemcinin ne yaptığını bilmemiştir. Yeni bir gözlemci eklemek istasyonun kodunu değiştirmez. Gözlemci lambda olmak zorunda da değildir; arayüzü uygulayan sıradan bir sınıf da olabilir. Aşağıdaki kayıtçı her ölçümü bir listede saklar:

```
import java.util.*;
interface Gozlemci { void guncelle(int derece); }
class Istasyon {
  private final List<Gozlemci> liste = new ArrayList<>();
  void ekle(Gozlemci g) { liste.add(g); }
  void setSicaklik(int d) {
    for (Gozlemci g : liste) g.guncelle(d);
  }
}
class Kayitci implements Gozlemci {
  private final List<Integer> olcumler = new ArrayList<>();
  public void guncelle(int derece) { olcumler.add(derece); }
  List<Integer> getOlcumler() { return olcumler; }
}
public class Main {
  public static void main(String[] args) {
    Istasyon ist = new Istasyon();
    Kayitci k = new Kayitci();
    ist.ekle(k);
    ist.setSicaklik(25);
    ist.setSicaklik(34);
    ist.setSicaklik(18);
    System.out.println(k.getOlcumler());
  }
}
```

```
[25, 34, 18]
```

Kayıtçı istasyona bir kez kaydedilmiş ve üç ölçümün hepsini almıştır. Bu örnek, gözlemcinin konudan bağımsız olarak denenebildiğini de gösterir: kayıtçının listesi, istasyonun gerçekten her ölçümü ilettiğinin kanıtıdır. Gözlemci deseni, grafik arayüzlerdeki düğme tıklamalarından tablo programlarındaki hücre güncellemelerine kadar, “bir şey olunca haber ver” gereken her yerde kullanılır.

4.5.10.5 Strateji

Bir mağazanın kasası ödenecek tutarı hesaplıyor. Öğrenciye %10 indirim, kampanya döneminde 200 TL üstüne 50 TL indirim uygulanıyor; yarın başka bir kural gelebilir. Kasa bu kuralları `if` zinciriyle kendisi bilirse her yeni kural kasanın kodunu değiştirir.

Tanım 4.5.19

Strateji deseni (*strategy*), birbirinin yerine geçebilen algoritmaları ortak bir arayüzün arkasına koyar. Algoritmayı kullanan nesne, işi kendisine verilen strateji nesnesine devreder; strateji değişince kodu değişmeden davranışı değişir.

```
interface Indirim { double uygula(double tutar); }

class OgrenciIndirimi implements Indirim {
  public double uygula(double tutar) { return tutar * 0.90; }
}
class Kasa {
  private final Indirim strateji;
  Kasa(Indirim strateji) { this.strateji = strateji; }
  double odenecek(double tutar) {
    return strateji.uygula(tutar);
  }
}
Kasa k = new Kasa(new OgrenciIndirimi());
System.out.println(k.odenecek(250));   // 225.0
```

```
225.0
```

`Kasa` indirimi kendisi hesaplamaz. Yapıcıda verilen `Indirim` nesnesini bir alanda tutar ve `odenecek` çağrıldığında işi ona devreder. `OgrenciIndirimi` bu arayüzün bir gerçeklemesidir: 250 · 0.90 = 225.0. Arayüzün tek metodu olduğu için strateji bir lambda da olabilir. Aşağıdaki program aynı `Kasa` sınıfını üç farklı stratejiyle kullanır:

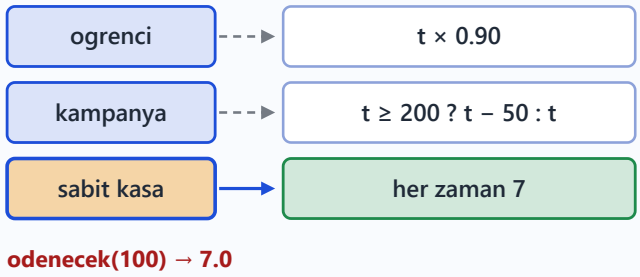
```
interface Indirim { double uygula(double tutar); }
class Kasa {
  private final Indirim strateji;
  Kasa(Indirim strateji) { this.strateji = strateji; }
  double odenecek(double tutar) { return strateji.uygula(tutar); }
}
// main:
Kasa ogrenci = new Kasa(t -> t * 0.90);
Kasa kampanya = new Kasa(t -> t >= 200 ? t - 50 : t);
System.out.println(ogrenci.odenecek(250));
System.out.println(kampanya.odenecek(250));
System.out.println(kampanya.odenecek(120));
// sabit strateji: Kasa tek başına denenir
System.out.println(new Kasa(t -> 7).odenecek(100));
```

```
225.0
200.0
120.0
7.0
```

Adım	Satır	Değişkenler	Yeni çıktı
1	8: `Kasa ogrenci = new Kasa(t -> t * 0.90);`	ogrenci=Kasa(%10)	—
2	9: `Kasa kampanya = new Kasa(t -> t >= 200 ? t - 50 : t);`	kampanya=Kasa(200 ve üstü −50)	—
3	10: `System.out.println(ogrenci.odenecek(250));` 5: `double odenecek(double tutar) { return strateji.uygula(tutar); }`	tutar=250.0	`225.0`
4	11: `System.out.println(kampanya.odenecek(250));` 5: `double odenecek(double tutar) { return strateji.uygula(tutar); }`	tutar=250.0	`200.0`
5	12: `System.out.println(kampanya.odenecek(120));` 5: `double odenecek(double tutar) { return strateji.uygula(tutar); }`	tutar=120.0	`120.0`
6	14: `System.out.println(new Kasa(t -> 7).odenecek(100));`	tutar=100.0	`7.0`

Tablo 4.5.16: Strateji deseninin izlenmesi.

Tablo 4.5.16'nın birinci adımında ilk kasaya %10 indirim lambdası takılır. İkinci adımda aynı `Kasa` sınıfı başka bir stratejiyle kurulur: tutar 200 TL ya da üstündeyse 50 TL düşülür. Üçüncü adımda `ogrenci.odenecek(250)` işi stratejiye devreder: 250 × 0.90 = 225.0. Dördüncü adımda kampanya stratejisi çalışır: 250 ≥ 200 olduğu için 250 − 50 = 200.0. Beşinci adımda 120 < 200'dür ve indirim yoktur: 120.0. Kasanın kodu hiç değişmeden davranışı değişmiştir.



Şekil 4.5.23: Üç kasa ve takılı stratejileri: `t × 0.90`, `t ≥ 200 ? t − 50 : t` ve tutardan bağımsız olarak 7 döndüren sabit strateji.

Altıncı adımın amacı farklıdır. Gerçek bir indirim kuralı yerine, tutar ne olursa olsun 7 döndüren sabit bir strateji takılır (Şekil 4.5.23). Sonuç 7.0'dır; `t -> 7` lambdasının döndürdüğü `int` değer, metodun dönüş tipi `double` olduğu için genişletilmiştir. Bu satır indirim hesabını değil, yalnızca `Kasa`'nın kendisini dener: `Kasa` işi stratejiye doğru biçimde devrediyor mu? Sabit strateji takıldığında sonuç yalnızca stratejiden gelebilir; 7.0 görülüyorsa devretme doğrudur. Bir parçayı, bağımlı olduğu parçanın yerine basit bir sahte nesne koyarak tek başına denemek, stratejinin dışarıdan verilmesiyle mümkün olmuştur.

Java kütüphanesinde stratejinin en bilinen örneği sıralamadır. `isimler.sort((a, b) -> a.length() - b.length())` çağrısındaki lambda bir `Comparator` stratejisidir; sıralama algoritması aynı kalır, karşılaştırma ölçütü değişir.

4.5.10.6 MVC

Kullanıcıyla etkileşen programlarda üç farklı iş iç içe geçme eğilimindedir: verinin kendisi ve kuralları, verinin ekranda gösterilmesi ve kullanıcının girdilerine tepki verilmesi. Bu üç işi tek sınıfta toplamak, ekranda yapılan küçük bir değişikliğin veri kurallarını bozmasına yol açar.

Tanım 4.5.20

MVC (*Model–View–Controller*, Model–Görünüm–Denetleyici), programı üç sorumluluğa ayıran mimari kalıptır. **Model** veriyi ve kuralları tutar, **görünüm** yalnızca gösterir, **denetleyici** girdiyi alır ve modeli ile görünümü yönetir.



Şekil 4.5.24: MVC: denetleyici kullanıcı girdisini alır, modeli günceller ya da okur, sonucu görünüme verir. Model ekrandan habersizdir.

Sorumlulukları tek tek ele alalım. **Model** (`Ogrenci`, `getAd()`) veriyi ve kuralları içerir. Ekrandan habersizdir; ekran olmadan da denenebilir. **Görünüm** (`OgrenciGorunum`) yalnızca gösterir; `goster(ad)` metodu konsola yazar. **Denetleyici** (`OgrenciDenetleyici`) girdiyi alır ve iki tarafı yönetir; `guncelle()` modeli okur, sonucu görünüme verir.

```
Ogrenci m = new Ogrenci("Ayşe");
OgrenciGorunum g = new OgrenciGorunum();
OgrenciDenetleyici d =
    new OgrenciDenetleyici(m, g);
d.guncelle();  // g.goster(m.getAd())
```

Bu kısa kesitte kullanılan üç sınıfın tam hâli aşağıdaki programdadır:

```
class Ogrenci {
  private final String ad;
  Ogrenci(String ad) { this.ad = ad; }
  String getAd() { return ad; }
}
class OgrenciGorunum {
  void goster(String ad) { System.out.println("Öğrenci: " + ad); }
}
class OgrenciDenetleyici {
  private final Ogrenci model;
  private final OgrenciGorunum gorunum;
  OgrenciDenetleyici(Ogrenci model, OgrenciGorunum gorunum) {
    this.model = model;
    this.gorunum = gorunum;
  }
  void guncelle() { gorunum.goster(model.getAd()); }
}
public class Main {
  public static void main(String[] args) {
    Ogrenci m = new Ogrenci("Ayşe");
    OgrenciGorunum g = new OgrenciGorunum();
    OgrenciDenetleyici d =
        new OgrenciDenetleyici(m, g);
    d.guncelle();  // g.goster(m.getAd())
  }
}
```

```
Öğrenci: Ayşe
```

Denetleyici, model ve görünüm nesnelerini yapıcısında alır. Bu da bir bağımlılık enjeksiyonudur: denetleyici hangi model ve görünümle çalışacağını kendisi oluşturmaz, dışarıdan alır. Görünümü değiştirmek, örneğin adı konsol yerine bir dosyaya yazmak, yalnızca `OgrenciGorunum`'u etkiler; model hiç değişmez. Uygulama çerçeveleri MVC'yi farklı biçimlerde uygular. Tek bir "doğru" sınıf yapısı yoktur; ortak olan, üç sorumluluğun birbirinden ayrılmasıdır.

4.5.10.7 Desen ne zaman kullanılmaz?

Desen öğrenmenin bir yan etkisi, desenleri gereksiz yerlerde kullanma isteğidir. Dört yaygın yanlış vardır:

- **Tekil'i küresel değişken yapmak.** Her yerden erişilen ve değiştirilebilen durum, programın parçalarını birbirine görünmez biçimde bağlar.
- **Gereksiz fabrika.** `new Nokta(3, 4)` kadar basit bir oluşturma için ayrı bir sınıf yazmak.
- **Gereksiz gözlemci.** Tek bir doğrudan metot çağrısı yeterken bir olay sistemi kurmak; programın akışını izlemek zorlaşır.

- **Deseni ad olsun diye kullanmak.** Önce sorun, sonra desen gelir. Bir desen eklenmeden önce iki soru sorulur: "Hangi değişikliği kolaylaştırıyor? Hangi bağımlılığı azaltıyor?"

Strateji ve bağımlılığın dışarıdan verilmesi, gerçek bir parçanın yerine basit bir sahte nesne koymayı sağlar. Kasaya sabit strateji takmak ve ödeme metoduna sahte ödeme nesnesi vermek bunun örnekleridir. Tersinden bakılırsa tek başına denenmesi zor bir sınıf, çoğu zaman tasarımının gözden geçirilmesi gerektiğini gösterir.

4.5.11 Örnek problemler ve alıştırmalar

4.5.11.1 Örnek problemler

Aşağıdaki problemler, soyut sınıf ve arayüz kurallarının bir arada uygulanmasını ister. Her problemin çözümü, kuralı gerekçesiyle birlikte verir; çıktılar ve derleyici iletileri çalıştırılarak doğrulanmıştır.

Örnek 4.5.1

Aşağıdaki (a)–(e) parçalarının her biri için: parça derlenir mi, derlenmiyorsa hata nedir? Derlenen parçalardan hangisinin nesnesi `new` ile oluşturulabilir?

```
// (a)
abstract class A {
    void f() { System.out.println("f"); }
}
// (b)
class B { abstract void g(); }
// (c)
interface I { int X = 1; }
...  I.X++;
// (d)
interface J { void h(); }
class C implements J { void h() { } }
// (e)
abstract class D { abstract void k(); }
abstract class E extends D { }
```

(a) derlenir. `A`'nın hiç soyut metodu yoktur ama soyut işaretlenmiştir; bu geçerlidir. **(b)** derlenmez. `B`'nin soyut bir metodu vardır, kendisi ise soyut değildir:

```
P57b.java:1: error: B is not abstract and does not override abstract method g() in B
class B { abstract void g(); }
^
1 error
```

(c) derlenmez. `X` arayüzde tanımlandığı için örtük olarak `public static final`'dır; `I.X++` ona yeni bir değer atamaya çalışır:

```
P57c.java:3: error: cannot assign a value to static final variable X
    void f() { I.X++; }
               ^
1 error
```

(d) derlenmez. `J`'deki `h()` örtük olarak `public`'tir; `C`'deki `h()` ise paket erişimlidir ve erişimi daraltır:

```
P57d.java:2: error: h() in C cannot implement h() in J
class C implements J { void h() { } }
                            ^
  attempting to assign weaker access privileges; was public
1 error
```

(e) derlenir. `E` de soyut olduğu için `k()`'yı yazmak zorunda değildir; yükümlülük `E`'nin somut alt sınıflarına geçer. Sorunun ikinci kısmının cevabı "hiçbiri"dir. Derlenen iki parçadaki `A` ve `E` soyuttur; `new A()` ve `new E()` da derleme hatasıdır:

```
P57new.java:8: error: A is abstract; cannot be instantiated
        A a = new A();
              ^
P57new.java:9: error: E is abstract; cannot be instantiated
        E e = new E();
              ^
2 errors
```

Örnek 4.5.2

Aşağıdaki program ne yazdırır? `Balik`, `ses()`'i yazmadığı hâlde neden derlenir? `Kedi`'deki `ses()`'ten `public` silinirse ne olur? `Sesli s = new Balik(); s.ad();` derlenir mi?

```
interface Sesli {
    default String ses() { return "..."; }
}
abstract class Hayvan implements Sesli {
    Hayvan() { System.out.print("H "); }
    abstract String ad();
}
class Kedi extends Hayvan {
    Kedi() { System.out.print("K "); }
    String ad() { return "kedi"; }
    public String ses() { return "miyav"; }
}
class Balik extends Hayvan {
    String ad() { return "balık"; }
}
// main içinde:
Hayvan[] d = { new Kedi(), new Balik() };
System.out.println();
for (Hayvan h : d)
    System.out.println(h.ad() + ": " + h.ses());
```

```
H K H
kedi: miyav
balık: ...
```

Dizi kurulurken iki nesne oluşturulur. `new Kedi()`'de önce üst sınıfın yapıcısı çalışır ve “H ” yazar, sonra `Kedi`'nin yapıcısı “K ” yazar. `new Balik()`'te `Balik`'in yazılmış bir yapıcısı yoktur; derleyicinin eklediği varsayılan yapıcı `super()`'i çağırır ve yalnızca “H ” yazılır. Böylece ilk satır “H K H ” olur; `println()` satırı bitirir. Döngüde `ad()` her nesnede kendi sınıfının sürümünü çalıştırır. `Kedi` `ses()`'i geçersiz kılmıştır: “miyav”. `Balik` kılmamıştır; arayüzün varsayılan gövdesi çalışır: “...”.

`Balik`'in derlenmesinin nedeni, `ses()`'in soyut değil varsayılan bir metot olmasıdır. `Balik` onu kalıtımla alır. Somut olmak için tek soyut metot olan `ad()`'ı yazması yeterlidir. `Kedi`'deki `ses()`'ten `public` silinirse metot arayüzdeki `public` sürümü daha dar erişimle geçersiz kılmaya çalışır:

```
P58public.java:9: error: ses() in Kedi cannot implement ses() in Sesli
    String ses() { return "miyav"; }
           ^
  attempting to assign weaker access privileges; was public
1 error
```

Son soru referans tipinin rolünü sınar. `s`'nin tipi `Sesli`'dir ve `Sesli` yalnızca `ses()`'i bilir; nesnenin `ad()` metodu olsa da çağrı derlenmez:

```
P58yuz.java:13: error: cannot find symbol
        s.ad();
         ^
  symbol:   method ad()
  location: variable s of type Sesli
1 error
```

Örnek 4.5.3

Soyut sınıfın yapıcısı bir `static` sayacı artırıyor ve somut `ozet()` metodu soyut `alan()`'ı çağırıyor. Program ne yazar?

```
abstract class Sekil {
  static int adet = 0;
  String ad;
  Sekil(String ad) { this.ad = ad; adet++; }
  abstract double alan();
  String ozet() { return ad + ":" + alan(); }
}
class Kare extends Sekil {
  double k;
  Kare(double k) { super("Kare"); this.k = k; }
  double alan() { return k * k; }
}
class Daire extends Sekil {
  double r;
  Daire(double r) { super("Daire"); this.r = r; }
  double alan() { return 3 * r * r; }
}
public class Main {
  public static void main(String[] args) {
    Sekil[] d = { new Kare(2), new Daire(1) };
    for (Sekil s : d) System.out.println(s.ozet());
    System.out.println(Sekil.adet);
  }
}
```

```
Kare:4.0
Daire:3.0
2
```

Dizinin öğe tipi soyut `Sekil`'dir, gösterdiği nesneler somuttur. `new Kare(2)`, `super("Kare")` ile `Sekil`'in yapıcısını çağırır; ayrı bir `Sekil` nesnesi oluşmaz, aynı nesnenin `Sekil` kısmı kurulur ve `adet` 1 olur. `new Daire(1)` aynı yoldan geçer ve `adet` 2 olur. Döngüde `ozet()` `Sekil`'den gelir;

içindeki `alan()` çağrısı `this`'in sınıfına bağlanır. `Kare` için 2 · 2 = 4.0, `Daire` için 3 · 1 · 1 = 3.0 bulunur. Son satır, iki yapıcı çağrısını sayan `static` sayacı yazar: 2.

Örnek 4.5.4

Birbiriyle ilgisi olmayan üç sınıf aynı `Fiyatli` arayüzünü uyguluyor. Döngü sonunda toplam ve en pahalı ürünün fiyatı nedir?

```
interface Fiyatli { double fiyat(); }
class Kitap implements Fiyatli {
  public double fiyat() { return 40; }
}
class Kalem implements Fiyatli {
  int adet;
  Kalem(int adet) { this.adet = adet; }
  public double fiyat() { return 7.5 * adet; }
}
class Bilet implements Fiyatli {
  public double fiyat() { return 25; }
}
public class Main {
  public static void main(String[] args) {
    Fiyatli[] sepet = { new Kitap(), new Kalem(4),
                        new Bilet() };
    double toplam = 0;
    Fiyatli enPahali = sepet[0];
    for (Fiyatli f : sepet) {
      toplam += f.fiyat();
      if (f.fiyat() > enPahali.fiyat()) enPahali = f;
    }
    System.out.println(toplam + " " + enPahali.fiyat());
  }
}
```

```
95.0 40.0
```

Dizinin öğe tipi bir arayüzdür. `Kitap`, `Kalem` ve `Bilet`'in ortak noktası yalnızca `Fiyatli` sözleşmesidir. `enPahali` ilk öğeyle, yani kitapla başlar. Birinci turda `f` kitabı gösterir: fiyat 40.0, toplam 40.0; 40 > 40 yanlış olduğu için en pahalı değişmez. İkinci turda `f` kalemi gösterir: 7.5 × 4 = 30.0, toplam 70.0; 30 > 40 yanlış. Üçüncü turda bilet: 25.0, toplam 95.0; 25 > 40 yanlış. Çıktı “95.0 40.0”dır. Aynı `f.fiyat()` çağrısı her turda başka bir sınıfın metodunu çalıştırmıştır. Yeni bir `Fiyatli` sınıfı eklemek için döngü değişmez.

Örnek 4.5.5

`ikiKez()` arayüzde bir kez yazılmış bir varsayılan metottur. İki nesnede kaç artış yapar? Program ne yazar?

```
interface Sayac {
  int deger();
  void arttir();
  default boolean sifirMi() { return deger() == 0; }
  default void ikiKez() { arttir(); arttir(); }
}
class Basit implements Sayac {
  private int d;
  public int deger() { return d; }
  public void arttir() { d++; }
}
class Ciftli extends Basit {
  @Override public void arttir() {
    super.arttir(); super.arttir();
  }
}
public class Main {
  public static void main(String[] args) {
    Sayac a = new Basit();
    Sayac b = new Ciftli();
    a.ikiKez();
    b.ikiKez();
    System.out.println(a.deger() + " " + b.deger()
        + " " + b.sifirMi());
  }
}
```

```
2 4 false
```

`a` bir `Basit`'tir. `a.ikiKez()`'de varsayılan gövde `arttir()`'ı iki kez çağırır; bu çağrı `this.arttir()`'dir ve `Basit`'in sürümüne bağlanır: `d` önce 1, sonra 2 olur. `b` bir `Ciftli`'dir. Aynı varsayılan gövde çalışır, ama `this` bu kez bir `Ciftli`'dir ve geçersiz kılınmış `arttir()` her çağrıda `super.arttir()`'ı iki kez çağırır: `d` 2, sonra 4 olur. `Ciftli`'nin `d` alanı `Basit`'ten gelir; `private` olduğu için yalnızca `Basit`'in metotlarıyla değişir. Son olarak `sifirMi()` de varsayılan bir metottur; `deger()` 4'tür ve sonuç `false`'tur. Çıktı "2 4 false"dur. Tek bir varsayılan metot, onu çağıran sınıfın geçersiz kıldığı metotlar sayesinde iki nesnede farklı davranmıştır.

Örnek 4.5.6

Gerekli içe aktarmalar yapılmıştır. Program ne yazdırır?

```
import java.util.*;
import java.util.function.*;
import java.util.stream.Collectors;
public class Main {
  public static void main(String[] args) {
    IntBinaryOperator f = (a, b) -> a * b + 1;
    IntUnaryOperator g = x -> x * x;
    int t = f.applyAsInt(3, 4) + g.applyAsInt(5);
    System.out.println(t);
    List<Integer> l =
            Arrays.asList(5, 8, 3, 12, 7);
    System.out.println(l.stream()
            .filter(s -> s > 5)
            .map(s -> s / 2)
            .collect(Collectors.toList()));
  }
}
```

```
38
[4, 6, 3]
```

`f.applyAsInt(3, 4)` lambdayı `a = 3`, `b = 4` ile çalıştırır: 3 · 4 + 1 = 13. `g.applyAsInt(5)` 5 · 5 = 25 verir. İki değer de `int`'tir ve toplamları 38'dir. Akışta `filter(s -> s > 5)` 5'ten büyük öğeleri, sıralarını koruyarak tutar: 8, 12, 7. `map(s -> s / 2)` her birini dönüştürür. Bölme tam sayı bölmesidir: 8 / 2 = 4, 12 / 2 = 6, 7 / 2 = 3. Sonuç `[4, 6, 3]`'tür.

4.5.11.2 Alıştırmalar

Aşağıdaki alıştırmalar kod yazmayı gerektirir. Her birinin ardından olası bir çözüm ve sonucu verilmiştir.

Alıştırma 1. Soyut bir `Calisan` sınıfı: `isim` alanı, yapıcı, soyut `double maas()` ve adı ile maaşı yazdıran `bilgi()`. Alt sınıflar: aylık 30000 alan `TamZamanli` ve 120 saat × 200 TL alan `SaatUcretli`. Bir `Calisan[]` dizisinde `bilgi()` çağrılır. Aynı programda iki küçük soru daha cevaplanır: soyut `double alan()` ve `"Alan: " + alan()` döndüren varsayılan `ozet()` metodu olan `Sekil2B` arayüzünde `Ucgen(6, 4)` için `ozet()` ne döndürür? `interface Hesap { static int topla(int a, int b) … }` ve `class Basit implements Hesap {}` için `Hesap.topla(3, 4)` ve `Basit.topla(3, 4)` ne verir?

```
abstract class Calisan {
    private final String isim;
    Calisan(String isim) { this.isim = isim; }
    abstract double maas();
    void bilgi() { System.out.println(isim + ": " + maas()); }
}
class TamZamanli extends Calisan {
    TamZamanli(String isim) { super(isim); }
    double maas() { return 30000; }
}
class SaatUcretli extends Calisan {
    private final int saat;
    private final double ucret;
    SaatUcretli(String isim, int saat, double ucret) { super(isim); this.saat = saat;
this.ucret = ucret; }
    double maas() { return saat * ucret; }
}
interface Sekil2B {
    double alan();
    default String ozet() { return "Alan: " + alan(); }
}
class Ucgen implements Sekil2B {
    private final double taban, yukseklik;
    Ucgen(double taban, double yukseklik) { this.taban = taban; this.yukseklik = yukseklik;
}
    public double alan() { return taban * yukseklik / 2; }
}
interface Hesap { static int topla(int a, int b) { return a + b; } }
class Basit implements Hesap { }
public class Main {
    public static void main(String[] args) {
        Calisan[] calisanlar = { new TamZamanli("Ayşe"), new SaatUcretli("Ali", 120, 200)
};
        for (Calisan c : calisanlar) c.bilgi();
        System.out.println(new Ucgen(6, 4).ozet());
        System.out.println(Hesap.topla(3, 4));
    }
}
```

```
Ayşe: 30000.0
Ali: 24000.0
Alan: 12.0
7
```

`bilgi()` somuttur ve `Calisan`'da bir kez yazılmıştır; içindeki `maas()` her nesnede kendi sınıfının formülünü çalıştırır: 30000.0 ve 120 · 200 = 24000.0. Üçgenin alanı 6 · 4 / 2 = 12.0'dır ve varsayılan `ozet()` "Alan: 12.0" döndürür. `Hesap.topla(3, 4)` 7 verir. `Basit.topla(3, 4)` ise derlenmez, çünkü arayüzün `static` metotları kalıtımla geçmez:

```
BasitTopla.java:5: error: cannot find symbol
        System.out.println(Basit.topla(3, 4));
                           ^
  symbol:   method topla(int,int)
  location: class Basit
1 error
```

Alıştırma 2. `sealed abstract class Sekil permits Kare, Cember` yazılır; `Kare` `final`, `Cember` `non-sealed`'dir. `Elips extends Cember` ve `Ucgen extends Sekil` derlenir mi? Aynı programda bir de "hem hayvan hem yüzebilen balık" tasarlanır.

```
sealed abstract class Sekil permits Kare, Cember { abstract double alan(); }
final class Kare extends Sekil {
    double a = 2;
    double alan() { return a * a; }
}
non-sealed class Cember extends Sekil {
    double r = 1;
    double alan() { return 3 * r * r; }
}
class Elips extends Cember { }
abstract class Hayvan { }
interface Yuzebilir { void yuz(); }
class Balik extends Hayvan implements Yuzebilir {
    public void yuz() { System.out.println("Balık yüzüyor"); }
}
public class Main {
    static String ad(Sekil s) {
        return switch (s) {
            case Kare k -> "kare";
            case Cember c -> "çember";
        };
    }
    public static void main(String[] args) {
        Sekil[] dizi = { new Kare(), new Cember(), new Elips() };
        for (Sekil s : dizi) System.out.println(ad(s) + " " + s.alan());
        Yuzebilir y = new Balik();
        y.yuz();
        Hayvan h = (Hayvan) y;
        System.out.println(h instanceof Yuzebilir);
    }
}
```

```
kare 4.0
çember 3.0
çember 3.0
Balık yüzüyor
true
```

`Elips` derlenir: `Cember` `non-sealed` olduğu için onu herkes genişletebilir. `switch` bir `Elips`'i `Cember` olarak tanır ve "çember" der; alanı da `Cember`'den gelir. `Ucgen` ise izin listesinde yoktur:

```
MuhurSinif.java:4: error: class is not allowed to extend sealed class: Sekil (as it is not
listed in its 'permits' clause)
class Ucgen extends Sekil { }
^
1 error
```

`Kare` `final` olduğu için onu genişletmek de mümkün değildir; ileti bu kez mühürlemeden değil `final`'dan söz eder:

```
MuhurFinalAlt.java:4: error: cannot inherit from final Kare
class BuyukKare extends Kare { }
                        ^
1 error
```

Balık tasarımında hem akrabalık hem yetenek vardır. `Hayvan` soyut sınıftır, yüzme yeteneği `Yuzebilir` arayüzüdür: `class Balik extends Hayvan implements Yuzebilir`. Aynı nesne `Yuzebilir` tipinde tutulup `Hayvan`'a dönüştürülebilir; ikisi de aynı nesnenin farklı tipleridir.

Alıştırma 3. `class Kare implements GeometrikSekil` içinde `public void Ciz() { … }` yazılmıştır. Ne olur? Metodun üstüne `@Override` eklenirse ne değişir?

`Ciz` ile `ciz` farklı adlardır; Java büyük ve küçük harfi ayırt eder. `Ciz()` yeni bir metottur ve `ciz()` hâlâ eksiktir: `Kare is not abstract and does not override abstract method ciz() in GeometrikSekil`. `@Override` eklenince derleyici ikinci bir ileti verir ve hatanın asıl yerini gösterir:

```
OverrideYok.java:4: error: Kare is not abstract and does not override abstract method
ciz() in GeometrikSekil
class Kare implements GeometrikSekil {
^
OverrideYok.java:5: error: method does not override or implement a method from a supertype
    @Override
    ^
2 errors
```

4.5.12 Sık yapılan hatalar

Soyut sınıf ve arayüz hatalarının neredeyse hepsini derleyici yakalar. İletiler İngilizcedir ve kalıplaşmıştır; bir kez tanınınca hatanın yeri hemen bulunur. Aşağıdaki kutular en sık görülenleri iletileriyle birlikte verir.

Sık yapılan hata

Soyut sınıftan nesne oluşturmak. `new Sekil("?")` → `Sekil is abstract; cannot be instantiated` . Nesne somut bir alt sınıftan oluşturulur; değişkenin tipi soyut kalabilir: `Sekil s = new Kare(3);` . Aynı ileti arayüzden nesne oluşturulmaya çalışıldığında da gelir.

Sık yapılan hata

Soyut metodu yazmayı unutmak. `Cember is not abstract and does not override abstract method ciz() in Sekil` . Ya metot yazılır ya da sınıf `abstract` yapılır. Arayüz uygulanırken ileti aynıdır; yalnızca son kelime arayüzün adıdır.

Sık yapılan hata

Soyut metoda gövde yazmak ya da sınıfı soyut yapmamak. `abstract void ciz() { }` → `abstract methods cannot have a body` . Metot ya soyuttur ya da gövdelidir. `class Sekil { abstract void ciz(); }` → `Sekil is not abstract and does not override abstract method ciz() in Sekil` .

Sık yapılan hata

Çelişen belirteçler. `abstract final` , `private abstract` , `static abstract` → `illegal combination of modifiers` . Üç belirteç de geçersiz kılmayı engeller; soyut metot ve soyut sınıf ise alt sınıf için vardır.

Sık yapılan hata

İki sınıftan türetmek. `class C extends A, B` → `'{' expected` . Bir sınıfın tek `extends` 'i, çok sayıda `implements` 'i olabilir. İletinin hatayı açıklamaması, onu tanımayı zorlaştırır.

Sık yapılan hata

Referansın tipinde olmayan metodu çağırmak. `Ucabilir u = new Ordek(); u.yuz();` → `cannot find symbol`. Hangi metotların çağrılabileceğini nesne değil, referansın tipi belirler. Metot, onu bilen tipte bir referansla çağrılır (`Yuzebilir` ya da `Ordek`).

Sık yapılan hata

Çakışan varsayılan metotları çözmemek. `class Gorevli inherits unrelated defaults for selam() from types Selamlayan and Karsilayan`. Metot sınıfta geçersiz kılınır; istenen arayüzün sürümü `ArayuzAdi.super.selam()` ile çağrılabilir.

Sık yapılan hata

Mühürleme kurallarını unutmak. İzinsiz alt tip: `class is not allowed to extend sealed class: Hayvan`. İzinli alt tipte belirteç yok: `sealed, non-sealed or final modifiers expected`.

Sık yapılan hata

Lambdanın kullandığı yerel değişkeni değiştirmek. `local variables referenced from a lambda expression must be final or effectively final`. Lambda yerel değişkenin değerinin bir kopyasını taşır; değişken sonradan değiştirilemez. Değişmesi gereken bir değer alanda tutulur.

Sık yapılan hata

Jenerik tiplerin alt tip ilişkisini devraldığını sanmak. `List<Number> x = tamlar;` (`tamlar` bir `List<Integer>`) → `incompatible types: List<Integer> cannot be converted to List<Number>`. Okuyan metot parametresini `List<? extends Number>`, yazan metot `List<? super Integer>` olarak yazar.

4.5.13 Özet

- **Soyutlama**, ayrıntıyı gizleyip “ne yapılır”ı tanımlamak, “nasıl”ı alt tiplere bırakmaktır. Java'da iki aracı vardır: soyut sınıf ve arayüz.

- **Soyut sınıf** `abstract class` ile tanımlanır; nesnesi oluşturulamaz. Alan, yapıcı, soyut ve somut metot içerebilir. Yapıcısı, alt sınıf nesnesinin kendisinden gelen kısmını kurar ve `super(...)` ile çağrılır.
- **Soyut metot** gövdesizdir, `;` ile biter. Somut alt sınıf onu uygulamak zorundadır. `private`, `static` ve `final` ile birlikte yazılamaz.
- **Şablon metot**, algoritmanın iskeletini üst sınıfta (`final` bir metotta) tanımlar; değişen adımlar alt sınıflarda yazılan soyut metotlardır.
- **Arayüz** bir sözleşmedir. Metotları örtük `public abstract`, alanları `public static final`'dır; örnek alanı ve yapıcısı yoktur. Bir sınıf tek `extends`, çok sayıda `implements` yazabilir; uygulayan metot `public` olmalıdır.
- Hangi metotların çağrılabileceğini **referansın tipi**, hangi gövdenin çalışacağını **nesnenin sınıfı** belirler. Arayüz tipiyle programlamak gevşek bağlılık sağlar.
- **Lambda ifadesi** `(parametreler) -> ifade` biçiminde yazılan adsız fonksiyondur ve bir **fonksiyonel arayüze** (tek soyut metotlu arayüz) atanır (Java 8+). Tanımlandığında değil, arayüzün metodu çağrıldığında çalışır. Kullandığı yerel değişkenler fiilen final olmalıdır.
- Arayüzde gövdeli metotlar: **varsayılan** (Java 8, kalıtımla gelir), `static` (Java 8, yalnızca arayüz adıyla çağrılır) ve `private` (Java 9, yalnızca arayüz içinde). Çakışan varsayılan metotları sınıf geçersiz kılar; üst sınıftaki metot varsayılan metoda üstün gelir.
- **Mühürlü tip** (`sealed`, Java 17+) alt tiplerini `permits` ile sınırlar; her alt tip `final`, `sealed` ya da `non-sealed`'dir. Kapalı hiyerarşide `switch` `default` dalı olmadan her durumu kapsayabilir.
- Soyut sınıf yakın akraba sınıfları ortak durum ve kodla gruplar ("bir türüdür"); arayüz ilgisiz sınıflara yetenek kazandırır ("yapabilir"). Çoğu zaman ikisi birlikte kullanılır.
- **Jenerikler** sınıf ve metotları tip parametresiyle yazar (`Kutu<T>`, `<T extends Ust>`). Tip hataları derlemede yakalanır. Jenerik tipler değişmezdir; esneklik joker tiplerle (`? extends`, `? super`) sağlanır. Tip bilgisi derlemeden sonra silinir.
- **Etiketler** `@Override` gibi üst verilerdir; `RUNTIME` etiketleri yansımayla okunur.
- **Tasarım desenleri** denenmiş çözüm kalıplarıdır: Tekil (tek nesne), Fabrika (oluşturmayı toplamak), Gözlemci (değişikliği kayıtlı nesnelere bildirmek), Strateji (değiştirilebilir algoritma) ve MVC (veri, gösterim ve denetimi ayırmak). Desen ancak gerçek bir sorunu çözüyorsa kullanılır.

4.5.14 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

soyutlama *abstraction* · soyut sınıf *abstract class* · soyut metot *abstract method* · somut sınıf *concrete class* · şablon metot *template method* · yapıcı zinciri *constructor chaining* · arayüz *interface* · uygulamak *implement* · çoklu kalıtım *multiple inheritance* · dinamik bağlama *dynamic binding* · bildirilen tip / gerçek tip *declared type / actual type* · gevşek bağlılık *loose coupling*

lambda ifadesi *lambda expression* · fonksiyonel arayüz *functional interface* · anonim sınıf *anonymous class* · fiilen final *effectively final* · varsayılan metot *default method* · mühürlü sınıf / arayüz *sealed class / interface* · jenerikler *generics* · sınırlı tip parametresi *bounded type parameter* · joker tip *wildcard* · tip silme *type erasure* · etiket *annotation* · yansıma *reflection* · tasarım deseni *design pattern* · fabrika metodu *factory method* · bağımlılık enjeksiyonu *dependency injection* · MVC *Model–View–Controller*

4.5.15 Kendini Dene

1. Soyut `Belge` sınıfının yapıcısı `"B "`, alt sınıflar `Mektup` ve `Kitap`'ın yapıcıları sırasıyla `"M "` ve `"K "` yazdırıyor. `Belge`'nin `final String etiket() { return ad + "(" + sayfa() + ")"; }` metodu soyut `sayfa()`'yı çağırıyor; `Mektup` 1, `Kitap(int bolum)` `bolum * 20` döndürüyor. `Belge[] b = { new Mektup(), new Kitap(3) };` satırı ne yazdırır? Ardından her öğenin `etiket()`'i nedir?

   Cevap

   Her nesnede önce soyut üst sınıfın yapıcısı, sonra alt sınıfınki çalışır: `B M B K`. Ayrı bir `Belge` nesnesi oluşmaz. `etiket()` `Belge`'de bir kez yazılmıştır ama `sayfa()` her nesnede kendi sınıfına bağlanır: `mektup(1)` ve $3 \cdot 20 = 60$ ile `kitap(60)`.

2. `Sayici` arayüzünde soyut `int say()` ve `default int ikiKati() { return 2 * say(); }`; `Olcer` arayüzünde `default int ikiKati() { return 100; }` var. `Ikili implements Sayici, Olcer` sınıfı `say()`'da 7 döndürüyor ve `ikiKati()`'yı `Sayici.super.ikiKati() + Olcer.super.ikiKati()` olarak yazıyor. `new Ikili().ikiKati()` kaçtır? `Ikili` `ikiKati()`'yı yazmasaydı ne olurdu?

   Cevap

   `Sayici`'nın varsayılan gövdesi `this.say()`'ı çağırır ve $2 \cdot 7 = 14$ bulur; `Olcer`'inki 100 döndürür. Sonuç 114'tür. Metot yazılmasaydı iki arayüz aynı imzalı iki varsayılan gövde getirdiği için sınıf derlenmezdi: `inherits unrelated defaults for ikiKati()`.

3. `Function<Integer, Integer> f = x -> x + 1;` ve `Function<Integer, Integer> g = x -> x * 3;` tanımlarından sonra `f.apply(g.apply(4))` ve `g.apply(f.apply(4))` kaçtır? Tanım satırları çalıştığında lambdalardan herhangi biri çalışmış olur mu?

   Cevap

   İçteki çağrı önce çalışır. `g.apply(4)` = `12`, `f.apply(12)` = `13`. `f.apply(4)` = `5`, `g.apply(5)` = `15`. Sonuç `13 15`'tir. Tanım satırları yalnızca iki arayüz nesnesi oluşturur; gövdeler ancak `apply` çağrıldığında çalışır.

4. `List<String> l = new ArrayList<>(List.of("ayva", "elma", "kivi", "nar", "armut"));` listesine `l.removeIf(s -> s.length() == 4);` uygulanıyor. Liste ne olur? `removeIf`'in parametresi hangi fonksiyonel arayüzdür?

   Cevap

   Dört harfli "ayva", "elma" ve "kivi" silinir; kalan liste `[nar, armut]`'tur ve kalanların sırası korunur. `removeIf` bir `Predicate` alır: her öğe için `boolean` döndüren `test` metodu. `removeIf`, Java 8'de `Collection` arayüzüne varsayılan metot olarak eklenmiştir.

5. Bu bölümdeki `enBuyuk` jenerik metodu `Integer[] d = {3, 9, 4}` ve `String[] a = {"Zeytin", "armut", "Elma"}` ile çağrılıyor. Sonuçlar nedir? İkinci sonuç neden ilk bakışta şaşırtıcıdır?

   Cevap

   Sayılarda 9 bulunur. Dizgilerde `compareTo` harf kodlarını karşılaştırır ve büyük harflerin kodları (Z = 90, E = 69) küçük harflerinkinden (a = 97) küçüktür. Bu yüzden sözlük sırasında en sonda gelen, yani "en büyük" dizgi `armut`'tur. Sonuç `9 armut`'tur.

6. Mühürlü `Hayvan` arayüzü yalnızca `Kedi` ve `Kopek`'e izin veriyor ve `tur` metodundaki `switch`'in iki dalı var, `default` dalı yok. Bir gün `permits` listesine `Kus` eklenip `final class Kus implements Hayvan` yazılıyor. Ne olur? `switch`'te baştan bir `default` dalı olsaydı ne olurdu?

   Cevap

   `tur` metodu derlenmez hâle gelir: `the switch expression does not cover all possible input values`. Derleyici, yeni tipin ele alınmadığı her yeri gösterir. `default` dalı olsaydı program derlenir ve `Kus` sessizce `default`'a düşerdi; eksik dal ancak çalışırken, belki hiç fark edilmezdi.

7. `Makine` sınıfında `public void calis()` "Makine çalışıyor" yazıyor. `Yazici` arayüzünde `default void calis()` "Yazdırılıyor" yazıyor. `class Fotokopi extends Makine implements Yazici { }` için `Yazici y = new Fotokopi(); y.calis();` ne yazdırır? Sınıf neden derlenir?

   Cevap

   "Makine çalışıyor" yazılır. Bir sınıf aynı imzalı metodu hem üst sınıftan hem bir arayüzün varsayılan gövdesinden alıyorsa sınıf hiyerarşisindeki metot kazanır. Bu durumda çakışma yoktur, bu yüzden sınıf derlenir. Referansın `Yazici` tipinde olması sonucu değiştirmez.

8. `Predicate<Integer> p = x -> { sayac.add(x); return x > 2; };` (`sayac` boş bir `ArrayList<Integer>`) tanımlanıyor. `List.of(1, 5, 2, 8)` üzerinde gezinen bir döngü `p.test(x)` doğruysa `n`'yi artırıyor. Döngüden sonra `n` ve `sayac` nedir? `sayac` bir `int` yerel

değişkeni olsaydı ve lambda onu `sayac++` ile artırsaydı ne olurdu?

Cevap

Lambda her `test` çağrısında bir kez çalışır ve gördüğü sayıyı listeye ekler. 5 ve 8 koşulu sağlar: `n = 2`, `sayac = [1, 5, 2, 8]`. Listeye öğe eklemek `sayac` değişkenine yeni bir değer atamak değildir; değişken hep aynı listeyi gösterir ve fiilen final kalır. Bir `int` yerel değişkenini `sayac++` ile artırmak ise ona yeni değer atamaktır ve derlenmez: `local variables referenced from a lambda expression must be final or effectively final`.

4.5.16 İleri okuma

- Oracle, *The Java Tutorials*, "Interfaces and Inheritance", "Lambda Expressions" ve "Generics" dersleri: soyut sınıf, arayüz, varsayılan metot, lambda ve jenerikleri kısa, çalıştırılabilir örneklerle işler; çevrim içi ve ücretsizdir.
- J. Bloch, *Effective Java* (3. baskı, Addison-Wesley, 2018): 20. madde ("Prefer interfaces to abstract classes") arayüz ile soyut sınıf seçimini, 31. madde joker tiplerin PECS kuralını, 42. madde ise lambdaları anonim sınıflara tercih etmenin gerekçelerini tartışır.
- E. Freeman ve E. Robson, *Head First Design Patterns* (2. baskı, O'Reilly, 2020): Strateji, Gözlemci, Fabrika ve Tekil desenlerini Java örnekleri ve bol çizimle, yeni başlayanlara göre anlatır.
- E. Gamma, R. Helm, R. Johnson ve J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, 1994): tasarım desenleri sözlüğünü kuran temel kitap; her desenin amacını, yapısını ve ödünleşimlerini tanımlar.

4.5.17 Kaynaklar

- Oracle, *The Java Tutorials*: "Abstract Methods and Classes", "Interfaces", "Default Methods", "Lambda Expressions", "Generics", "Annotations".
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21 Edition (8.1.1.1 abstract Classes, 8.1.1.2 sealed, non-sealed, and final Classes, Bölüm 9: Interfaces, 15.27 Lambda Expressions).
- Oracle, JEP 409: Sealed Classes (Java 17) ve JEP 441: Pattern Matching for switch (Java 21).
- P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (Bölüm 10: Object-Oriented Programming: Polymorphism and Interfaces).
- K. Sierra ve B. Bates, *Head First Java*, 3. baskı, O'Reilly (Bölüm 8: Interfaces and Abstract Classes).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson (Bölüm 5–6).
- E. Gamma, R. Helm, R. Johnson ve J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
- J. Bloch, *Effective Java*, 3. baskı, Addison-Wesley, 2018 (20., 31. ve 42. maddeler).
- E. Freeman ve E. Robson, *Head First Design Patterns*, 2. baskı, O'Reilly, 2020.

KISIM II · BÖLÜM 5

Koleksiyonlar

Gerçek programlar tek tek değişkenlerle değil, veri yığınlarıyla çalışır. Bir sınıfın öğrenci listesi, bir sözlüğün kelimeleri, bir yazıcının bekleyen işleri hep birden çok öğeden oluşur. Dizi bu işin ilk aracıdır, ama boyutu baştan sabittir ve ekleme, silme, arama gibi işleri programcıya bırakır. Java bu ihtiyaç için hazır bir kütüphane sunar: Koleksiyon Çerçevesi. Not bu kütüphaneyi dört aile hâlinde ele alır. Listeler öğeleri sırayla tutar, kuyruklar işlenmeyi bekleyen öğeleri düzenler, kümeler tekrarı önler, eşlemeler bir anahtarı bir değere bağlar. Her ailenin hangi iç yapıyla çalıştığı, hangi işlemin ne kadar sürdüğü ve hangi durumda seçileceği adım adım izlenir. Ardından yineleyici, `Collections` sınıfının hazır algoritmaları ve koleksiyonları işlem hattı gibi işleyen Akış API'si gelir. Bütün programlar JDK 21 ile derlenip çalıştırılmıştır; çıktılar, izleme tablolarındaki değerler ve derleyici iletileri bu çalıştırmalardan alınmıştır.

Bu bölümün kodu: `github.com/sercankulcu/object-oriented-programming-java/Bolum05/src`

ÖĞRENME ÇIKTILARI

Çerçeveyi tanımak
Koleksiyon Çerçevesinin arayüz, gerçekleme ve algoritma unsurlarını ayırmak; arayüz ve sınıf hiyerarşisini okumak; değişkeni arayüz tipinde tanımlamanın kazancını açıklamak.

Jenerik tiplerle çalışmak
`List<String>` gibi jenerik tiplerle tip güvenli kod yazmak; ilkel tipler için sarmalayıcı sınıfları ve otomatik kutulamayı doğru kullanmak.

Listeleri seçmek
`ArrayList`, `LinkedList`, `Vector` ve `Stack` sınıflarını kullanmak; iç yapılarını çizip işlem maliyetlerini karşılaştırmak.

Kuyrukları kullanmak
`Queue`, `PriorityQueue` ve `ArrayDeque` ile ilk giren ilk çıkar, öncelik ve son giren ilk çıkar davranışlarını kurmak; ikili yığının nasıl çalıştığını izlemek.

Küme ve eşleme seçmek
`HashSet`, `TreeSet`, `HashMap`, `TreeMap` ve bağlantılı sürümlerini sıra ve hız ihtiyacına göre seçmek; kendi sınıfları için `equals` ve `hashCode` yazmak.

Gezinmek, sıralamak, akış kurmak
Yineleyiciyle güvenle gezinip silmek; `Collections` ile sıralamak, karıştırmak ve aramak; `Comparable` ve `Comparator` yazmak; `java.util.stream` ile süzme, dönüştürme ve toplama hatları kurmak.

5.1 Koleksiyon Çerçevesi

Dizi, aynı tipten değerleri art arda tutan en temel yapıdır. Hızlıdır ve indisle her öğeye anında ulaşır. Ama iki büyük eksiği vardır. Birincisi boyutudur: `new int[3]` ile oluşturulan dizi ömrü boyunca üç öğe tutar. İkincisi hazır işlem eksikliğidir: bir öğenin dizide olup olmadığını sormak, ortadan bir öğe silmek ya da diziyi sıralamak için her seferinde döngü yazmak gerekir. Bu kesim, bu iki eksiği kapatan kütüphaneyi ve onun temel fikirlerini tanıtır.

5.1.1 Neden koleksiyonlar?

Tanım 5.1

Koleksiyon (*collection*), birden çok öğeyi tek bir nesnede saklayan ve yöneten veri yapısıdır. **Java Koleksiyon Çerçevesi** (*Java Collections Framework*) ise `java.util` paketindeki hazır arayüzlerin, sınıfların ve algoritmaların bütünüdür.

Koleksiyonun diziden ilk farkı boyutun dinamik olmasıdır. Öğe eklendikçe koleksiyon büyür, öğe silindikçe küçülür. İkinci fark hazır metotlardır: `add` ekler, `remove` siler, `contains` bir öğenin var olup olmadığını söyler, `size` öğe sayısını verir. Bu metotlar yıllardır milyonlarca programda kullanılmış ve sınanmıştır. Aynı işi elle yazan bir programcının hata yapma olasılığı, kütüphaneyi kullananınkinden çok daha yüksektir.

Farkı somut görmek için dört sayı saklamayı deneyelim. Dizi üç hücreyle oluşturulmuşsa dördüncü değer sığmaz:

```
int[] dizi = new int[3];      // boyut sabit: 3
dizi[0] = 10;
dizi[1] = 20;
dizi[2] = 30;
// dizi[3] = 40;  → ArrayIndexOutOfBoundsException
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds
for length 3
```

Yorum satırı olarak gösterilen `dizi[3] = 40;` çalıştırıldığında Java `ArrayIndexOutOfBoundsException` fırlatır. İleti sorunu açıkça söyler: uzunluğu 3 olan dizide 3 numaralı indis yoktur. Geçerli indisler 0, 1 ve 2'dir. Aynı dört değer bir listeye ise sorunsuz girer:

```
List<Integer> liste = new ArrayList<>();
liste.add(10);
liste.add(20);
liste.add(30);
liste.add(40);       // sorun yok: büyür
System.out.println(liste);
System.out.println("boyut: " + liste.size());
```

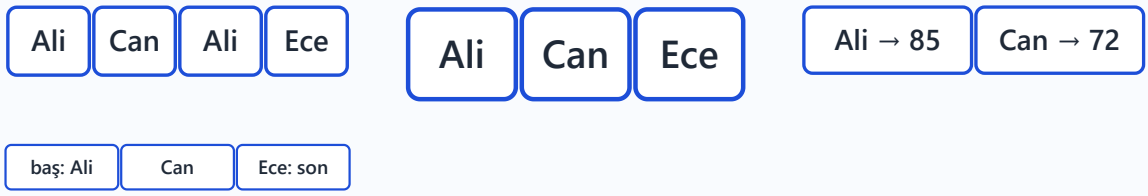
```
[10, 20, 30, 40]
boyut: 4
```

Burada `List<Integer>` değişkenin tipidir: "tam sayılardan oluşan bir liste". `new ArrayList<>()` ise gerçek nesneyi oluşturur. Başlangıçta liste boştur ve her `add` çağrısı değeri sona ekler. `println` listeyi köşeli parantez içinde, öğeleri virgülle ayırarak yazar; bu biçimi listenin kendi `toString` metodu üretir. `size()` o anda listede kaç öğe bulunduğunu verir.

Sunumdaki benzetme bu farkı iyi anlatır. Dizi, koltuk sayısı sabit bir otobüse benzer: koltuklar dolunca yeni yolcu binemez. `ArrayList` ise yolcu arttıkça büyüyen bir otobüstür. Büyümenin nasıl olduğu, yani otobüsün içeride nasıl genişlediği, listeler kesiminde ayrıntılı izlenir.

5.1.2 Dört temel tür

Çerçevedeki onlarca sınıf dört temel türde toplanır. Şekil 5.1 her türü küçük bir örnekle gösterir.



Şekil 5.1: Dört temel koleksiyon türü. Liste (Ali, Can, Ali, Ece) sıralıdır ve tekrara izin verir. Küme (Ali, Can, Ece) her öğeden yalnız bir tane tutar. Eşleme anahtarları değerlere bağlar (Ali → 85, Can → 72). Kuyrukta baştaki öğe ilk çıkar.

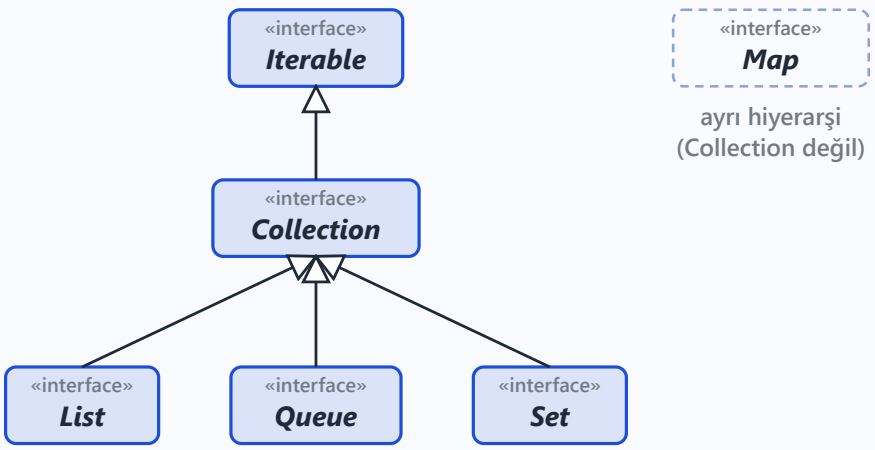
- **Liste** (*list*): öğeler belirli bir sırayla tutulur ve her birinin bir indisi vardır. Aynı öğe birden çok kez bulunabilir. Örnekler `ArrayList` ve `LinkedList`'tir.
- **Küme** (*set*): yalnızca tekil öğeler bulunur. Var olan bir öğe ikinci kez eklenmez. Örnekler `HashSet` ve `TreeSet`'tir.

- **Eşleme** (*map*): anahtar → değer çiftleri tutulur. Her anahtar bir kez bulunur, değerler tekrar edebilir. Örnekler `HashMap` ve `TreeMap`'tir.
- **Kuyruk** (*queue*): işlenmeyi bekleyen öğeler tutulur. Genellikle ilk giren ilk çıkar (*First In, First Out*, FIFO). Örnekler `LinkedList` ve `PriorityQueue`'dur.

Çerçeve bu türlerle birlikte bazı ortak hedefler gözetir. Temel işlemler (ekleme, çıkarma, güncelleme, arama) her türde benzer adlarla yapılır ve hepsinde for-each döngüsüyle gezinilebilir. API sade ve tutarlıdır; bir türü öğrenen öteki türleri kolayca kullanır. Her sınıf belirli bir veri hacmine ve erişim biçimine göre ayarlanmıştır. Jenerik tipler sayesinde yanlış tipteki öğe derleme zamanında yakalanır. Sıralama, arama ve karıştırma gibi algoritmalar hazırdır ve hatalı kullanım bir istisnayla bildirilir. Sonuçta daha az kod yazılır ve yazılan kod daha güvenilir olur.

5.1.3 Arayüz ve sınıf hiyerarşisi

Çerçevenin iskeleti arayüzlerden oluşur. Arayüz, bir tipte hangi metotların bulunacağını söyler, ama bu metotların gövdesini yazmaz. Şekil 5.2 temel arayüzleri ve aralarındaki genişletme ilişkisini gösterir.



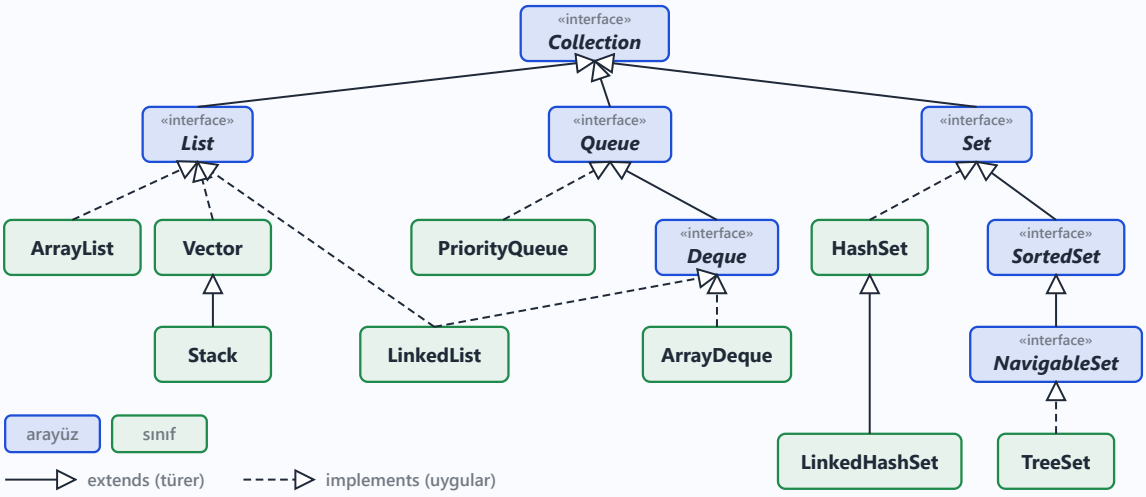
Şekil 5.2: Koleksiyon arayüzlerinin hiyerarşisi. `Collection`, `Iterable`'ı genişletir; `List`, `Queue` ve `Set` de `Collection`'ı genişletir. `Map` çerçevenin parçasıdır, ama `Collection`'dan türemez. İçi boş üçgen uçlu oklar UML gösterimidir ve genişletilen üst arayüzü gösterir.

Hiyerarşinin en üstünde `Iterable` durur. Bu arayüzün tek zorunlu metodu `iterator()`'dır. Bir tip `Iterable` ise for-each döngüsüyle gezilebilir. `Collection`, bütün koleksiyonların ortak metotlarını toplar: `add`, `remove`, `size`, `contains` ve ötekiler. `List`, `Queue` ve `Set` bu ortak metotları devralır ve kendi metotlarını ekler. Örneğin indisle erişim (`get(i)`) yalnız `List`'te vardır.

`Map` ise ayrı bir daldır. Bir eşlemenin öğesi tek bir değer değil, anahtar-değer çiftidir. `add(e)` gibi tek öğe alan bir metot eşlemeye uymaz; eşlemede ekleme `put(anahtar, değer)` ile yapılır. Bu yüzden `Map` `Collection`'dan türemez ve `Iterable` da değildir. Bunun önemli bir sonucu vardır: bir eşleme

doğrudan for-each döngüsüne verilemez. Eşlemenin anahtarları, değerleri ya da çiftleri ayrı görünümler olarak alınıp onlar gezilir.

Arayüzlerin altında onları uygulayan sınıflar bulunur. Şekil 5.3 en sık kullanılan sınıfları ve hangi arayüzü uyguladıklarını toplu olarak verir.



Şekil 5.3: Koleksiyon sınıflarının hiyerarşisi. Mavi kutular arayüzdür; yalnızca davranışı tanımlar. Yeşil kutular `new` ile nesnesi oluşturulabilen sınıflardır. `LinkedList` hem `List` hem `Deque` arayüzünü uygular.

Şekilde dikkat çeken nokta, bir sınıfın birden çok arayüzü uygulayabilmesidir. `LinkedList` hem bir listedir (indisle erişim) hem de çift uçlu bir kuyruktur (baştan ve sondan ekleme, çıkarma). Aynı nesne, hangi arayüz tipindeki değişkenle kullanılırsa o arayüzün metotlarını sunar. Arayüz tipinden nesne ise oluşturulamaz: `new List<>()` yazılırsa derleyici durur.

```
ListNew.java:4: error: List is abstract; cannot be instantiated
        List<Integer> a = new List<>();
                          ^
1 error
```

İleti, `List` 'in soyut (*abstract*) olduğunu söyler. Soyut bir tipin metot gövdeleri yoktur; nesneyi her zaman somut bir sınıf (`ArrayList` , `LinkedList`) verir.

5.1.4 Çerçevenin üç unsuru

Koleksiyon Çerçevesi üç unsurdan oluşur ve her birinin ayrı bir görevi vardır.

Arayüz (*interface*)
: Bir veri yapısının nasıl davranması gerektiğini tanımlar, yani *ne* yapılacağını söyler. `List` sıralı bir listeyi temsil eder: öğeler bir sıradadır, indisle erişilir, tekrar olabilir.

Gerçekleme (*implementation*)

Arayüzü uygulayan somut sınıftır, yani işin *nasıl* yapıldığını belirler. `ArrayList`, `List` arayüzünü dinamik bir diziyle uygular; `LinkedList` aynı arayüzü bağlı düğümlerle uygular.

Algoritma (*algorithm*)

Sıralama, arama, karıştırma gibi işleri yapan hazır metotlardır. Çoğu `Collections` sınıfında `static` metot olarak bulunur. `Collections.sort()` bir listeyi sıralar.

Üç unsur kısa bir kodda bir arada görülür:

```
List<String> liste = new ArrayList<>();  // arayüz ← sınıf
liste.add("Mavi");
liste.add("Kırmızı");
Collections.sort(liste);                  // algoritma
System.out.println(liste);
```

```
[Kırmızı, Mavi]
```

İlk satırda solda arayüz (`List<String>`), sağda gerçekleme (`ArrayList`) vardır. `Collections.sort` algoritmadır ve listeyi doğal sırasına, yani dizgiler için alfabetik sıraya dizer. “Kırmızı” K ile, “Mavi” M ile başladığı için K önce gelir. Algoritma listenin hangi sınıftan olduğunu bilmek zorunda değildir; `List` arayüzünün sunduğu metotlarla çalışır. Aynı `sort` çağrısı bir `LinkedList` üzerinde de çalışır.

Bu düzen modüler bir tasarımdır. Her bileşenin belirli bir rolü ve sorumluluğu vardır. Farklı gerçeklemeler ve algoritmalar aynı arayüz üzerinden birlikte kullanılabilir. Yeni bir liste sınıfı yazan biri, `List` arayüzünü uyguladığı anda bütün hazır algoritmalardan yararlanır.

5.1.5 Arayüze göre programlama

Arayüz bir davranışı tanımlar; sınıflar bu davranışı kendi metotlarıyla uygular. Her sınıf farklı bir depolama biçimi kullanır: `ArrayList` dinamik dizi, `LinkedList` çift yönlü bağlı liste, `HashMap` anahtar-değer tablosu. Her biri belirli bir veri yapısının gereksinimlerini karşılar. Programcı için iyi alışkanlık, değişkeni ve parametreyi arayüz tipinde tanımlamaktır.

```
public class ListeYazdir {
    // Parametre türü arayüz: her List ile çalışır
    static void yazdir(List<String> liste) {
        for (String s : liste) {
            System.out.print(s + " ");
        }
        System.out.println("(" + liste.size() + " öğe)");
    }

    public static void main(String[] args) {
        List<String> a = new ArrayList<>();
        a.add("Elma");
        a.add("Armut");
        List<String> b = new LinkedList<>(a);  // aynı öğeler
        b.add("Kiraz");
        yazdir(a);
        yazdir(b);
    }
}
```

```
Elma Armut (2 öğe)
Elma Armut Kiraz (3 öğe)
```

`yazdir` metodunun parametresi `List<String>` tipindedir. Metot, kendisine verilen listenin `ArrayList` mi, `LinkedList` mi olduğunu bilmez ve bilmesi de gerekmez. İçerideki for-each döngüsü ve `size()` çağrısı her listede vardır. `main`'de `a` bir `ArrayList`, `b` ise bir `LinkedList`'tir. `new LinkedList<>(a)` yazımı, `a`'nın öğeleriyle doldurulmuş yeni bir liste oluşturur; bu yüzden `b` önce Elma ve Armut'u içerir, sonra Kiraz eklenir. İki çağrı aynı metoda gider ve iki farklı sınıfın nesnesini aynı biçimde yazar.

Bu yaklaşımın asıl kazancı değişiklik anında görülür. Bir gün listenin başına çok sık ekleme yapıldığı ve `ArrayList`'in yavaş kaldığı anlaşılırsa, yalnız `new` satırı değiştirilir. Değişkenin tipi zaten `List` olduğu için kodun geri kalanı aynı kalır. Değişken `ArrayList` tipinde tanımlanmış olsaydı, o tipi kullanan her metot imzasının da değişmesi gerekirdi. Joshua Bloch'un *Effective Java* kitabı bu ilkeyi "nesnelere arayüzleriyle başvurun" diye özetler.

Sık yapılan hata

Arayüz tipindeki değişkenle sınıfa özgü metodu çağırmak. Değişken `List<String> l = new LinkedList<>();` diye tanımlanmışsa `l.addFirst("x")` derlenmez, çünkü `addFirst` `List`'te değil, `Deque`'te tanımlıdır. Derleyici değişkenin tipine bakar, nesnenin gerçek sınıfına bakmaz. Uçlarda çalışmak gerekiyorsa değişken `Deque<String>` tipinde tanımlanır.

5.1.6 Jenerik tipler ve tip güvenliği

Tanım 5.2

Jenerik tip (*generic type*), bir sınıfın ya da arayüzün bir tip parametresiyle yazılmasıdır. `List<E>`'deki `E` öğe tipini temsil eder; `List<String>` yazıldığında `E`'nin yerine `String` gelir ve tip "String listesi" olur. **Tip güvenliği** (*type safety*), yanlış tipteki değerin derleme zamanında yakalanmasıdır.

```
List<String> isimler = new ArrayList<>();
isimler.add("Ali");
// isimler.add(42);          // derleme hatası
String ilk = isimler.get(0);  // tür dönüşümü gerekmez

List<Integer> notlar = new ArrayList<>();  // List<int> olmaz
notlar.add(85);               // kutulama: int → Integer
int n = notlar.get(0);        // kutudan çıkarma → int
System.out.println(ilk + " " + n);
```

```
Ali 85
```

Açılı parantezler (`<>`) içine öğe tipi yazılır. `isimler` yalnız `String` kabul eder. Yorum satırındaki `isimler.add(42)` açılırsa program derlenmez:

```
YanlisTip.java:5: error: incompatible types: int cannot be converted to String
        isimler.add(42);
                    ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
1 error
```

Hata çalışma anına kalmaz; program hiç çalışmadan önce yakalanır. İkinci kazanç `get` tarafındadır. `isimler.get(0)`'ın sonucu doğrudan `String`'dir; tip dönüşümü (*cast*) yazmak gerekmez.

Jenerikler Java 5 ile gelmiştir. Önceki "ham" (*raw*) kullanımda koleksiyon her öğeyi `Object` olarak tutar ve yanlış tip ancak çalışma anında, okunan öğe dönüştürülürken `ClassCastException` olarak ortaya çıkar. Jenerik tip aynı yanlışı derleme anında yakalar.

Sağ taraftaki boş `<>` işaretine **elmas** (*diamond*) işleci denir ve Java 7'den beri kullanılır. Derleyici tipi soldaki bildirimden çıkarır: `List<String> isimler = new ArrayList<>();` ile `new ArrayList<String>()` aynı anlamdadır. Elmas yalnız `new` tarafında yazılır; `ArrayList<> a = …` derlenmez.

Koleksiyonlar yalnızca nesne tutar

Tip parametresi bir referans tipi olmalıdır; `int`, `double` gibi ilkel tipler buraya yazılamaz. `List<int>` denemesinin iletisi şudur:

```
IlkelTip.java:4: error: unexpected type
        List<int> notlar = new ArrayList<>();
             ^
  required: reference
  found:    int
1 error
```

İleti "referans tipi bekleniyordu, `int` bulundu" der. İlkel tipler için her birine karşılık gelen bir **sarmalayıcı sınıf** (*wrapper class*) kullanılır. Tablo 5.1 en sık kullanılanları verir.

İlkel tip	Sarmalayıcı sınıf
`int`	`Integer`
`double`	`Double`
`char`	`Character`
`boolean`	`Boolean`
`long`	`Long`

Tablo 5.1: İlkel tipler ve sarmalayıcı sınıfları.

İlkel değer ile sarmalayıcı nesne arasındaki dönüşüm kendiliğinden olur. `notlar.add(85)` satırında `85` bir `int`'tir, ama liste `Integer` bekler. Derleyici bu çağrıyı `notlar.add(Integer.valueOf(85))`'e çevirir; buna **otomatik kutulama** (*autoboxing*) denir. `int n = notlar.get(0)` satırında ters yönde bir dönüşüm olur: `Integer` nesnesinin içindeki değer `intValue()` ile alınır. Buna **kutudan çıkarma** (*unboxing*) denir.

Kutudan çıkarmanın bir tuzağı vardır. Sarmalayıcı referans `null` ise içinden çıkarılacak bir değer yoktur. Bir eşlemede bulunmayan anahtarın değeri `null` döner ve bu değer `int`'e atanınca program çöker:

```
import java.util.*;
public class BosKutu {
    public static void main(String[] args) {
        Map<String, Integer> m = new HashMap<>();
        int n = m.get("yok");
    }
}
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.lang.Integer.intValue()" because the return value of "java.util.Map.get(Object)" is
null
```

İleti hatanın kaynağını tam gösterir: `Map.get`'in döndürdüğü değer `null`'dır, dolayısıyla `intValue()` çağrılamaz. Bu satırda gözle görülen bir metot çağrısı yoktur; `intValue()` çağrısını derleyici kendisi eklemiştir. Böyle hatalar bu yüzden ilk bakışta şaşırtır.

İlk örnekler: liste ve eşleme

İki temel yapıyı yan yana görmek için bir isim listesi ve bir telefon rehberi kuralım:

```
// İsim listesi (ArrayList)
List<String> isimListesi = new ArrayList<>();
isimListesi.add("Ahmet");
isimListesi.add("Ayşe");
System.out.println(isimListesi);

// Telefon rehberi (HashMap)
Map<String, String> telefonRehberi = new HashMap<>();
telefonRehberi.put("Ahmet", "555-1234");
telefonRehberi.put("Ayşe", "555-9876");
System.out.println(telefonRehberi.get("Ahmet"));
```

```
[Ahmet, Ayşe]
555-1234
```

`ArrayList` öğeleri eklenme sırasıyla tutar; `add` her seferinde sona ekler. `HashMap` ise bir anahtarı (isim) bir değere (numara) eşler. Eşlemeye `put(anahtar, değer)` ile ekleriz, `get(anahtar)` ile değeri buluruz. `telefonRehberi.get("Ahmet")` rehberi baştan sona taramaz; anahtardan hesaplanan bir konuma doğrudan gider. Bu hızın nasıl sağlandığı kümeler ve eşlemeler kesimlerinde açıklanır.

5.1.7 Collection arayüzünün metotları

`Collection` arayüzü bütün listelerin, kümelerin ve kuyrukların ortak metotlarını tanımlar. Tablo 5.2 bu metotları, görevlerini ve dönüş tiplerini verir.

Metot	Görevi	Döndürür
`add(E e)`	öğeyi koleksiyona ekler	`boolean` : eklendiyse `true`
`remove(Object o)`	öğeyi koleksiyondan çıkarır	`boolean` : çıkarıldıysa `true`
`size()`	öğe sayısını verir	`int`
`isEmpty()`	koleksiyon boş mu?	`boolean`

Metot	Görevi	Döndürür
`contains(Object o)`	öğe koleksiyonda var mı?	`boolean`
`containsAll(Collection<?> c)`	`c` 'nin bütün öğeleri bu koleksiyonda var mı?	`boolean`
`clear()`	bütün öğeleri çıkarır	—
`toArray()`	koleksiyonu diziye dönüştürür	`Object[]`
`iterator()`	gezinmek için bir yineleyici verir	`Iterator<E>`
`forEach(Consumer<? super E> a)`	her öğeye bir işlem uygular (Java 8+)	—

Tablo 5.2: `Collection<E>` arayüzünün temel metotları. `E` öğe tipidir; `?` joker (*wildcard*) "herhangi bir tip" anlamına gelir.

`add` 'in `boolean` döndürmesi ilk bakışta gereksiz görünebilir. Bir listeye ekleme her zaman başarılı olur ve `add` hep `true` döndürür. Ama kümede durum farklıdır: öğe zaten varsa eklenmez ve `add` `false` döndürür. Ortak arayüz her iki durumu da karşılayacak biçimde tasarlanmıştır. `remove` ve `contains` parametresini `Object` olarak alır; öğelerin eşitliğini `equals` metoduyla sınar.

Aşağıdaki örnekte değişkenin tipi bilinçli olarak `Collection` seçilmiştir:

```
Collection<String> renkler = new ArrayList<>();
renkler.add("kırmızı");
renkler.add("yeşil");
renkler.add("mavi");
System.out.println(renkler.size());
System.out.println(renkler.contains("yeşil"));
renkler.remove("yeşil");
System.out.println(renkler);
Collection<String> aranan = Arrays.asList("mavi", "kırmızı");
System.out.println(renkler.containsAll(aranan));
Object[] dizi = renkler.toArray();
System.out.println(dizi.length);
renkler.clear();
System.out.println(renkler.isEmpty());
```

```
3
true
[kırmızı, mavi]
true
2
true
```

Program satır satır şöyle ilerler. Üç renk eklenince `size()` 3 olur. `contains("yeşil")` `true` döndürür. `remove("yeşil")` öğeyi değerine göre bulur ve siler; liste `[kırmızı, mavi]` olur. `Arrays.asList("mavi", "kırmızı")` iki öğeli bir liste oluşturur. `containsAll` bu listenin her öğesi-

nin `renkler`'de bulunup bulunmadığını sorar; sıranın önemi yoktur ve sonuç `true`'dur. `toArray()` öğeleri yeni bir diziye kopyalar; dizinin uzunluğu 2'dir. Bu dizi koleksiyondan bağımsızdır: dizide yapılan değişiklik koleksiyonu etkilemez. Son olarak `clear()` her şeyi siler ve `isEmpty()` `true` döner.

Değişken `Collection` tipinde olduğu için yalnızca ortak metotlar kullanılabilir. Nesne gerçekte bir `ArrayList` olsa da `renkler.get(0)` derlenmez:

```
CollGet.java:6: error: cannot find symbol
        String s = renkler.get(0);
                           ^
  symbol:   method get(int)
  location: variable renkler of type Collection<String>
1 error
```

İleti, `Collection<String>` tipindeki değişkende `get(int)` adlı bir metot bulunmadığını söyler. Bu durum arayüze göre programlama kuralının öteki yüzüdür: değişkenin tipi, hangi metotların çağrılabileceğini belirler. `Collection`'ın metotlarının hepsi ise bütün `List`, `Set` ve `Queue` nesnelerinde vardır.

5.2 Listeler

Liste, koleksiyonların en çok kullanılanıdır. Programcıların büyük kısmı diziden sonra ilk olarak `ArrayList`'i öğrenir ve çoğu iş için ondan başka bir şeye ihtiyaç duymaz. Bu kesim önce `List` arayüzünün sözleşmesini verir. Ardından iki temel gerçeklemeyi, `ArrayList` ve `LinkedList`'i iç yapılarıyla birlikte inceler. Kesimin sonunda Java'nın ilk sürümlerinden kalan `Vector` ve `Stack` ile listede gezinmenin üç yolu yer alır.

5.2.1 List arayüzü

Tanım 5.3

Liste (*list*), öğeleri eklenme sırasına göre tutan, her öğeye bir indisle erişim sağlayan ve aynı öğenin birden çok kez bulunmasına izin veren koleksiyondur. İlk öğenin indisi 0, son öğeninki `size() - 1`'dir.

Tanımdaki “sıra” sözcüğü iki farklı anlama gelebilir ve bu ikisi sık karıştırılır. Listedeki öğeler **sıralıdır** (*ordered*): her öğenin belirli bir konumu vardır ve öğeler o konumda kalır. Ama liste kendiliğinden **sıralanmış** (*sorted*) değildir. 30, 10, 20 sırasıyla eklenen sayılar listede yine 30, 10, 20 olarak durur. Küçükten büyüğe dizmek için ayrıca sıralama algoritması çağrılır.

`List` arayüzü `Collection`'dan gelen metotlara indisle çalışan metotlar ekler. Tablo 5.3 en sık kullanılanları verir.

Metot	Görevi
`add(E e)`	sona ekler
`add(int i, E e)`	`i` konumuna ekler; sonraki öğeler bir sağa kayar
`get(int i)`	`i`'deki öğeyi verir
`set(int i, E e)`	`i`'deki öğeyi değiştirir, eskisini döndürür
`remove(int i)`	`i`'deki öğeyi çıkarır ve döndürür
`remove(Object o)`	`o`'ya eşit ilk öğeyi çıkarır
`indexOf(Object o)`	`o`'nun ilk konumu; yoksa −1
`size()`	öğe sayısı

Tablo 5.3: `List<E>` arayüzünün temel metotları.

Arayüzün üç bilinen gerçeklemesi vardır. `ArrayList` dinamik bir dizi kullanır ve indisle erişimde çok hızlıdır. `LinkedList` çift yönlü bağlı liste kullanır; listenin uçlarında ekleme ve çıkarma hızlıdır. `Vector` ise eski ve senkronize bir sürümdür; yeni kodda yerine `ArrayList` tercih edilir.

Sık yapılan hata

`remove(int)` ile `remove(Object)`'u karıştırmak. `List<Integer>`'da iki metot da aynı adı taşır. `liste.remove(1)` çağrısında argüman bir `int`'tir ve derleyici kutulama yapmadan uyan `remove(int index)`'i seçer: indis 1'deki öğe silinir. Değeri 1 olan öğeyi silmek için argüman nesne olarak verilir: `liste.remove(Integer.valueOf(1))`. Örnekler sık yapılan hatalar kesiminde çalıştırılmıştır.

5.2.2 ArrayList ve iç dizisi

Tanım 5.4

Dinamik dizi (*dynamic array*), öğeleri içerideki sıradan bir dizide tutan ve bu dizi dolunca daha büyük bir diziye kopyalanarak büyüyen yapıdır. `ArrayList` `List` arayüzünü dinamik diziyle uygular; `AbstractList` sınıfından türetilmiştir.

```
ArrayList<String> meyveler = new ArrayList<>();
meyveler.add("Elma");
meyveler.add("Armut");
System.out.println(meyveler.get(1));  // indeksle
```

```
Armut
```

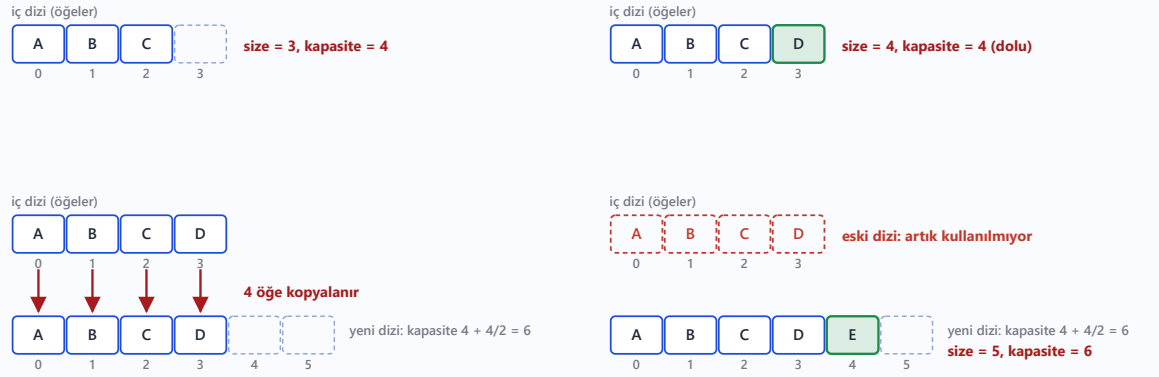
Bu örnekte değişken bilerek `ArrayList` tipinde tanımlanmıştır; sunum iki yazımı da gösterir. `meyveler.get(1)`, indis 1'deki öğeyi, yani eklenen ikinci meyveyi verir. İndisler 0'dan başladığı için "Elma" 0'da, "Armut" 1'dedir.

`ArrayList`'in hızı iç dizisinden gelir. `get(i)` çağrısı, dizinin `i` numaralı hücresine doğrudan gider. Dizideki her hücre aynı boyuttadır ve bellekte art arda durur. Bu yüzden `i` numaralı hücrenin adresi tek bir çarpma ve toplamayla bulunur. Liste on öğeli de olsa on milyon öğeli de olsa bu iş aynı sürer. `set(i, e)` de aynı nedenle hızlıdır.

Maliyetleri anlatırken büyük O gösterimi kullanılır. **O(1)**, sürenin öğe sayısından bağımsız, yani sabit olduğunu söyler. **O(n)**, sürenin öğe sayısı `n` ile orantılı büyüdüğünü söyler: öğe sayısı iki katına çıkınca süre de yaklaşık iki katına çıkar. `get` ve `set` O(1)'dir. Sona ekleme ortalamada O(1)'dir. Ortaya ekleme ve ortadan çıkarma ise öğeleri kaydırdığı için O(n)'dir.

ArrayList nasıl büyür?

İç dizinin uzunluğuna **kapasite** (*capacity*) denir. Kapasite, listenin `size()` değeriyle aynı şey değildir. `size()` dolu hücre sayısıdır; kapasite ise toplam hücre sayısıdır. Liste sona eklemeyi boş hücre kaldığı sürece doğrudan yapar. Dizi dolunca daha büyük bir dizi ayrılır ve eski öğeler oraya kopyalanır. Şekil 5.4 bu süreci kapasitesi 4 olan bir liste üzerinde gösterir.



Şekil 5.4: `ArrayList`'in büyümesi. Sol üstte `new ArrayList<>(4)` ile oluşturulmuş listenin 4 hücreli iç dizisinde A, B, C vardır. Sağ üstte `add("D")` boş hücreye yazar ve hiçbir öğe taşınmaz. Sol altta `add("E")` gelince dizi doludur: 6 hücreli yeni bir dizi ayrılır ve eski 4 öğe tek tek kopyalanır. Sağ altta E yeni dizinin 4 numaralı hücresine yazılır; eski dizi çöp toplayıcıya bırakılır.

Yeni dizinin boyu eski kapasitenin yaklaşık 1,5 katıdır: `4 + 4/2 = 6`. Bu kuralı kendimiz gözlemleyebiliriz. `ArrayList`'in iç dizisi `elementData` adlı özel bir alandadır. Normal programlar bu alana erişmez, ama ölçüm için yansıma (*reflection*) ile okunabilir. Aşağıdaki çıktı, notun ölçüm programının JDK 21'deki sonucudur:

```
size 3 kap 4
size 4 kap 4
size 5 kap 6
bos kap 0: 1->10 11->15 16->22 23->33 34->49 50->73 74->109
clear sonrası kap 109
```

İlk üç satır şekildeki adımlarla birebir aynıdır: kapasite 4 iken üç ve dört öğe sığar, beşinci öğede kapasite 6 olur. Dördüncü satır parametresiz `new ArrayList<>()` 'in davranışını gösterir. Boş liste başlangıçta hiç hücre ayırmaz (kapasite 0). İlk `add` ile 10 hücrelik dizi ayrılır; sunumdaki "varsayılan kapasite 10" bu anlamdadır. Sonraki büyümeler 15, 22, 33, 49, 73 ve 109 hücreye çıkar. Her biri önceki-nin 1,5 katıdır, küsurat atılır. Son satır önemli bir ayrıntıyı gösterir: `clear()` öğeleri siler ama iç diziyi küçültmez. Silme, kapasiteyi azaltmaz.

Amortize O(1)

Büyüme anında `add` çağrısı bütün öğeleri kopyalar; bu tek çağrı O(n) sürer. Buna rağmen sona eklemenin "ortalamada O(1)" olduğu söylenir. Bunun nedeni kopyalamanın seyrek olmasıdır. Kapasite her seferinde 1,5 katına çıktığı için iki büyüme arasındaki ekleme sayısı da giderek artar. Toplam kopyalama sayısını `n` eklemeye bölünce sabit bir sayı çıkar. Notun sayım programı bunu doğrudan hesaplar:

```
10 ekleme: 0 kopya, oran 0,00
100 ekleme: 202 kopya, oran 2,02
1000 ekleme: 2456 kopya, oran 2,46
1000000 ekleme: 2430972 kopya, oran 2,43
```

Bir milyon ekleme boyunca yaklaşık 2,4 milyon öğe kopyalanır. Ekleme başına ortalama 2,43 kopya düşer ve bu oran `n` büyüdükçe artmaz. Ekleme sayısı ne olursa olsun her ekleme ortalamada sabit miktarda iş yapar. Bu tür ortalamaya **amortize** (*amortized*) maliyet denir. Tek bir çağrı bazen pahalıdır, ama uzun bir dizi çağrıda pahalı adımların payı sabit kalır.

5.2.3 ArrayList metotlarını izlemek

Liste metotlarının birlikte nasıl çalıştığını görmek için beş isimle bir liste kurup üzerinde değişiklikler yapalım:

```
ArrayList<String> isimler = new ArrayList<>();
isimler.add("Ali");
isimler.add("Banu");
isimler.add("Cem");
isimler.add("Deniz");
isimler.add("Ece");
System.out.println(isimler);
isimler.set(3, "Can");       // 3. indeksi değiştir
System.out.println(isimler);
isimler.remove(4);           // 4. indeksi sil
System.out.println(isimler);
isimler.add(2, "Fatma");     // 2. indekse ekle
System.out.println(isimler);
int indeks = isimler.indexOf("Banu");
System.out.println("indeks: " + indeks);
boolean varMi = isimler.contains("Ece");
System.out.println("Ece var mı: " + varMi);
System.out.println("boyut: " + isimler.size());
```

```
[Ali, Banu, Cem, Deniz, Ece]
[Ali, Banu, Cem, Can, Ece]
[Ali, Banu, Cem, Can]
[Ali, Banu, Fatma, Cem, Can]
indeks: 1
Ece var mı: false
boyut: 5
```

Tablo 5.4 her adımda listenin içeriğini ve yeni çıktıyı verir. Satır numaraları yukarıdaki kodun satırlarıdır.

Adım	Satır	Liste	Değişkenler	Yeni çıktı
1	1: `ArrayList<String> isimler = new ArrayList<>();`	boş	—	—
2	2: `isimler.add("Ali");`	Ali	—	—
3	3: `isimler.add("Banu");` 4: `isimler.add("Cem");` 5: `isimler.add("Deniz");` 6: `isimler.add("Ece");`	Ali, Banu, Cem, Deniz, Ece	—	—
4	7: `System.out.println(isimler);`	Ali, Banu, Cem, Deniz, Ece	—	`[Ali, Banu, Cem, Deniz, Ece]`
5	8: `isimler.set(3, "Can"); // 3. indeksi değiştir` 9: `System.out.println(isimler);`	Ali, Banu, Cem, Can, Ece	—	`[Ali, Banu, Cem, Can, Ece]`
6	10: `isimler.remove(4); // 4. indeksi sil` 11: `System.out.println(isimler);`	Ali, Banu, Cem, Can	—	`[Ali, Banu, Cem, Can]`

Adım	Satır	Liste	Değişkenler	Yeni çıktı
7	12: `isimler.add(2, "Fatma"); // 2. indekse ekle` 13: `System.out.println(isimler);`	Ali, Banu, Fatma, Cem, Can	—	`[Ali, Banu, Fatma, Cem, Can]`
8	14: `int indeks = isimler.indexOf("Banu");` 15: `System.out.println("indeks: " + indeks);`	Ali, Banu, Fatma, Cem, Can	indeks=1	`indeks: 1`
9	16: `boolean varMi = isimler.contains("Ece");` 17: `System.out.println("Ece var mı: " + varMi);`	Ali, Banu, Fatma, Cem, Can	indeks=1, varMi=false	`Ece var mı: false`
10	18: `System.out.println("boyut: " + isimler.size());`	Ali, Banu, Fatma, Cem, Can	indeks=1, varMi=false	`boyut: 5`

Tablo 5.4: `ArrayList` metotlarının izlenmesi. Liste sütunu adımın sonundaki öğeleri, indis sırasıyla verir.

İlk adımda boş bir liste oluşur; `size()` 0'dır. `add("Ali")` ilk öğeyi 0 numaralı indise koyar. Dört `add` daha, öğeleri 1, 2, 3 ve 4 numaralı indislere yerleştirir. Liste yazdırılınca öğeler eklenme sırasıyla görünür. Şekil 5.5 izlemenin üç önemli anını gösterir.



Şekil 5.5: Üç değişiklik anı. Solda `set(3, "Can")` : Deniz'in yerine Can yazılır, boyut 5 kalır. Ortada `remove(4)` : son öğe Ece silinir, kaydırma gerekmez. Sağda `add(2, "Fatma")` : Cem ve Can birer sağa kayar, Fatma 2 numaralı indise girer.

`set(3, "Can")` boyutu değiştirmez; yalnızca 3 numaralı hücrenin içeriği değişir. Metot eski değeri ("Deniz") döndürür, ama bu örnekte dönüş değeri kullanılmamıştır. `remove(4)` son öğeyi siler. Son öğenin arkasında başka öğe olmadığı için hiçbir şey kaymaz ve işlem O(1) sürer. `add(2, "Fatma")` ise listenin ortasına ekler. Önce 2 ve 3 numaralı indislerdeki Cem ve Can birer hücre sağa taşınır, sonra boşalan 2 numaralı hücreye Fatma yazılır.

Son üç satır arama yapar. `indexOf("Banu")` listeyi baştan tarar ve ilk eşleşmenin indisini döndürür: 1. `contains("Ece")` de aynı taramayı yapar, ama Ece daha önce silindiği için sonuç `false`'tur. Son boyut 5'tir: 5 ekleme ve 1 araya ekleme yapılmış, 1 öğe silinmiştir. İki arama metodu da en kötü durumda bütün listede gezindiği için O(n)'dir. Bir listede sık sık "var mı?" sorusu soruluyorsa kümeye geçmek doğru seçimdir.

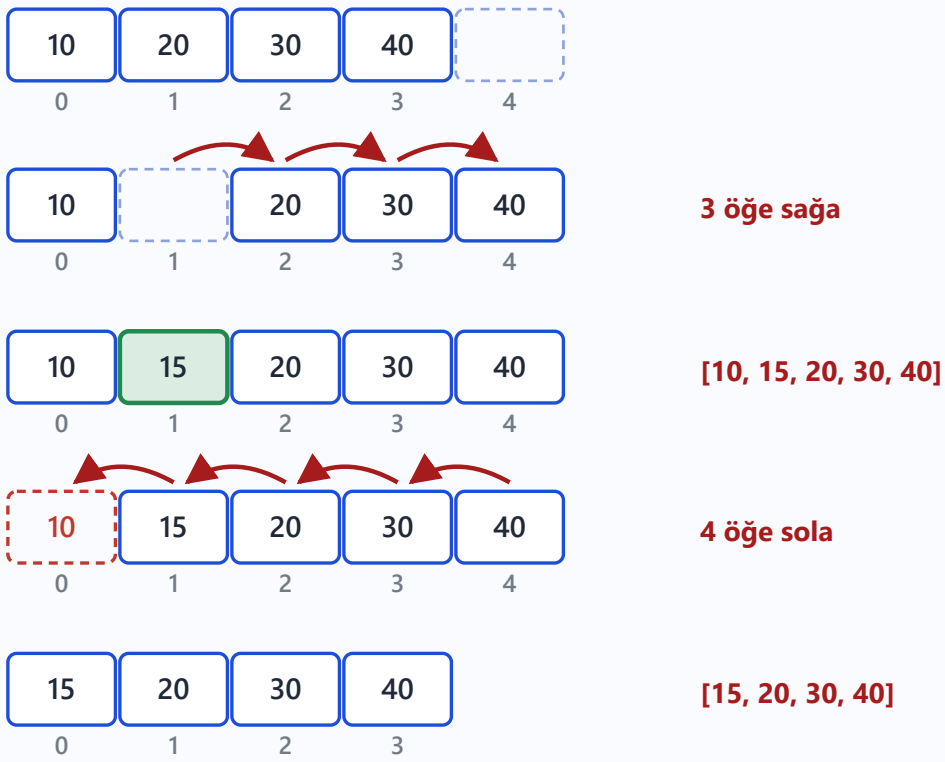
5.2.4 Ortaya ekleme ve silme

Kaydırmanın maliyetini daha yakından görmek için dört sayılık bir listeye araya bir öğe ekleyip baştan bir öğe silelim:

```
List<Integer> liste2 = new ArrayList<>(Arrays.asList(10, 20, 30, 40));
liste2.add(1, 15);      // araya ekle: 20, 30, 40 sağa kayar
System.out.println(liste2);
liste2.remove(0);       // baştan sil: hepsi sola kayar
System.out.println(liste2);
```

```
[10, 15, 20, 30, 40]
[15, 20, 30, 40]
```

`new ArrayList<>(Arrays.asList(10, 20, 30, 40))` yazımı, verilen değerlerle doldurulmuş değiştirilebilir bir liste oluşturur. Bu kalıp kısa örneklerde sık kullanılır. Şekil 5.6 iki işlemi adım adım gösterir.



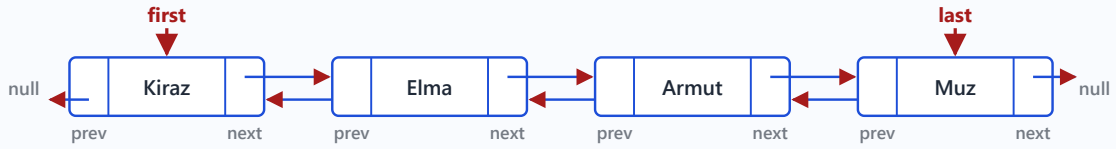
Şekil 5.6: Ortaya ekleme ve baştan silme. Başlangıçta liste `[10, 20, 30, 40]` 'tır. `add(1, 15)` için önce 1 numaralı indisten sonraki öğeler sondan başlayarak birer sağa kayar, sonra 15 boşalan hücreye yazılır. `remove(0)` 10'u siler; arkasındaki dört öğe birer sola kayar. Sonuç `[15, 20, 30, 40]` 'tır.

Sağa kaydırma sondan başlar. Önce 40, sonra 30, sonra 20 bir hücre sağa taşınır. Baştan başlansaydı 20, 30'un üzerine yazılır ve 30 kaybolurdu. Dört öğeli listede `add(1, 15)` üç öğe taşır. Genel olarak `i` konumuna ekleme `n - i` öğe taşır. En kötü durum başa eklemedir; bütün `n` öğe yer değiştirir. Silmede de durum aynıdır: `remove(0)` arkadaki bütün öğeleri birer sola çeker.

5.2.5 LinkedList

Tanım 5.5

Çift yönlü bağlı liste (*doubly linked list*), öğeleri **düğümlerde** (*node*) saklayan yapıdır. Her düğüm bir değer ile önceki (`prev`) ve sonraki (`next`) düğümün referansını tutar. `LinkedList` bu yapıyı kullanır; `AbstractSequentialList` sınıfından türetilmiştir ve hem `List` hem `Deque` arayüzünü uygular.



Şekil 5.7: Çift yönlü bağlı liste: Kiraz, Elma, Armut, Muz. `first` ilk düğümü, `last` son düğümü gösterir. Her düğümün `next` oku sonraki düğüme, `prev` oku önceki düğüme gider; uçlardaki oklar `null`'dır.

`LinkedList`'te kapasite diye bir kavram yoktur. Her ekleme yeni bir düğüm oluşturur ve onu komşularına bağlar. Hiçbir öğe kaydırılmaz. Listenin uçlarında ekleme ve çıkarma bu yüzden O(1)'dir: `first` ve `last` referansları uçları hazır tutar ve yalnızca birkaç referans değişir.

Bedel indisle erişimde ödenir. Düğümler bellekte art arda durmaz; her biri heap'in herhangi bir yerinde olabilir. 3 numaralı öğeyi bulmak için baştan başlayıp `next` bağlarını tek tek izlemek gerekir. `get(i)` bu yüzden O(n)'dir. Java'nın gerçeklemesi küçük bir iyileştirme yapar: indis listenin ilk yarısındaysa baştan, ikinci yarısındaysa sondan yürür. Bu, yürüyüşü en fazla `n/2` adıma indirir, ama büyüme hızı yine O(n)'dir.

Sunumdaki benzetme bu dengeyi iyi anlatır. Bağlı liste, el ele tutuşmuş bir sıra insana benzer. Araya biri girecekse yalnızca iki kişi el değiştirir. Ama 3. kişiyi bulmak için baştan saymak gerekir.

```
LinkedList<String> meyveler = new LinkedList<>();
meyveler.add("Elma");
meyveler.add("Armut");
meyveler.addFirst("Kiraz");   // başa ekle
meyveler.addLast("Muz");      // sona ekle
System.out.println(meyveler);
System.out.println(meyveler.getFirst());
System.out.println(meyveler.getLast());
meyveler.removeFirst();
System.out.println(meyveler);
```

```
[Kiraz, Elma, Armut, Muz]
Kiraz
Muz
[Elma, Armut, Muz]
```

Bu örnekteki `addFirst`, `addLast`, `getFirst`, `getLast`, `removeFirst` ve `removeLast` metotları `Deque` arayüzünden gelir. `add`, `get` ve `remove(i)` ise `List`'ten gelir. Değişken `LinkedList` tipinde tanımlandığı için her iki arayüzün metotları da kullanılabilir.

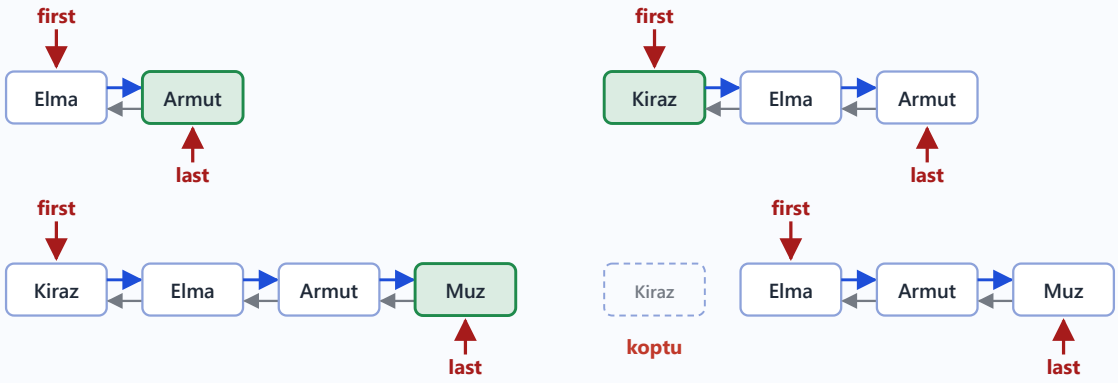
Düğümleri adım adım izlemek

Tablo 5.5 aynı programı satır satır izler. Düğümler sütunu, listenin `first`'ten başlayıp `next` bağlarıyla gezildiğinde verdiği sırayı gösterir.

Adım	Satır	Düğümler (baştan sona)	Yeni çıktı
1	1: `LinkedList<String> meyveler = new LinkedList<>();`	first = last = null	—
2	2: `meyveler.add("Elma");`	[Elma]	—
3	3: `meyveler.add("Armut");`	[Elma, Armut]	—
4	4: `meyveler.addFirst("Kiraz");`	[Kiraz, Elma, Armut]	—
5	5: `meyveler.addLast("Muz");`	[Kiraz, Elma, Armut, Muz]	—
6	6: `System.out.println(meyveler);`	[Kiraz, Elma, Armut, Muz]	`[Kiraz, Elma, Armut, Muz]`
7	7: `System.out.println(meyveler.getFirst());`	[Kiraz, Elma, Armut, Muz]	`Kiraz`
8	8: `System.out.println(meyveler.getLast());`	[Kiraz, Elma, Armut, Muz]	`Muz`
9	9: `meyveler.removeFirst();`	[Elma, Armut, Muz]	—
10	10: `System.out.println(meyveler);`	[Elma, Armut, Muz]	`[Elma, Armut, Muz]`

Tablo 5.5: `LinkedList` işlemlerinin izlenmesi.

Başlangıçta liste boştur: `first` ve `last` ikisi de `null`'dır. “Elma” için ilk düğüm oluşur; tek düğüm hem ilk hem son olduğu için iki referans da onu gösterir. “Armut” sona bağlanırken üç referans değişir. Elma'nın `next`'i Armut'u, Armut'un `prev`'i Elma'yı gösterir; `last` Armut'a geçer. Şekil 5.8 bu bağlama işini ve `removeFirst`'ün etkisini gösterir.



Şekil 5.8: Bağlı listede dört an. Sol üstte Armut sona bağlanır. Sağ üstte `addFirst("Kiraz")` yeni düğümü başa ekler; Elma'nın `prev`'i ve `first` değişir. Sol altta `addLast("Muz")`. Sağ altta `removeFirst()`: `first` Elma'ya geçer, Elma'nın `prev`'i `null` olur ve Kiraz düğümü listeden kopar.

`addFirst("Kiraz")` çağrısında yeni düğümün `next`'i Elma'yı gösterir, Elma'nın `prev`'i Kiraz'ı gösterir ve `first` Kiraz'a geçer. Bir dizide başa ekleme bütün öğeleri kaydırırdı. Burada yalnızca üç referans değişti. `addLast("Muz")` aynı işi sonda yapar. `println` listeyi `first`'ten başlayıp `next` bağlarını izleyerek yazar.

`getFirst()` ve `getLast()` doğrudan `first` ve `last`'in gösterdiği düğümün değerini döndürür; ikisi de O(1)'dir. `removeFirst()` `first`'ü bir sonraki düğüme taşır ve o düğümün `prev`'ini `null` yapar. Kiraz düğümüne artık hiçbir referans kalmadığı için çöp toplayıcı onu sonra temizler.

Boş bir `LinkedList`'te `peekFirst()` `null` döndürür, `getFirst()` ise `NoSuchElementException` fırlatır.

5.2.6 ArrayList mi, LinkedList mi?

İki sınıf aynı `List` arayüzünü uygular, ama işlemlerin maliyeti çok farklıdır. Tablo 5.6 temel işlemleri karşılaştırır.

İşlem	`ArrayList`	`LinkedList`
`get(i)`, `set(i, e)`	O(1)	O(n): düğüm düğüm yürür
Sona ekleme `add(e)`	O(1) ortalama (arada bir kopyalama)	O(1)
Başa ekleme / baştan silme	O(n): bütün öğeler kayar	O(1)
Ortaya ekleme / ortadan silme	O(n): kaydırma	O(n): konumu bulma + O(1) bağlama
Bellek kullanımı	az: yalnızca dizi	fazla: her düğümde 2 ek referans

Tablo 5.6: `ArrayList` ile `LinkedList` işlemlerinin maliyeti.

Tablonun dördüncü satırı sık yanlış anlaşılır. “Bağlı listede ortaya ekleme O(1)'dir” denir, ama bu yalnızca doğru düğüm elde hazırsa geçerlidir. `add(i, e)` çağrısı önce `i` numaralı düğümü bulmak için yürür; bu yürüyüş O(n)'dir. Bağlama adımı ise O(1)'dir. Sonuçta ortaya ekleme iki yapıda da O(n)'dir.

Aradaki fark, `ArrayList`'in öğeleri kaydırması, `LinkedList`'in ise düğümleri saymasıdır. Bellekte art arda duran bir diziyi kaydırmak, dağınık düğümlerde yürümekten çoğu zaman daha hızlıdır.

Farkın büyüklüğünü görmek için iki uç durumu ölçelim. Ölçüm programı her iki listenin başına 100.000 öğe ekler, sonra 100.000 kez `get(i)` çağırır:

```
başa ekleme ms: ArrayList 539, LinkedList 1 | get(i) ms: ArrayList 0, LinkedList 5161
```

Süreler makineye ve çalıştırmaya göre değişir; bu sayılar notun yazıldığı bilgisayardaki bir çalıştırmadan alınmıştır. Oranlar ise kararlıdır. Başa eklemede `LinkedList` yüzlerce kat hızlıdır, çünkü `ArrayList` her eklemede bütün öğeleri kaydırır ve toplam iş $O(n^2)$'ye çıkar. İndisle okumada durum tersine döner: `ArrayList` bir milisaniyenin altında biter, `LinkedList` ise her `get` için düğüm düğüm yürüdüğü için saniyeler harcar.

Çoğu durumda doğru seçim `ArrayList`'tir. Programlar listeyi çoğunlukla sona ekleyerek doldurur ve indisle ya da for-each ile okur. Bu iki işte `ArrayList` en az `LinkedList` kadar hızlıdır ve daha az bellek kullanır. Ekleme ve çıkarma sık ve özellikle listenin uçlarında yapılıyorsa `LinkedList` düşünülebilir. Kuyruk ve yığın ihtiyacı için ise genellikle `ArrayDeque` daha iyi bir seçimdir.

> **Sık yapılan hata**
>
> **`LinkedList`'i indisli `for` döngüsüyle gezmek.** `for (int i = 0; i < l.size(); i++) l.get(i)` döngüsünde her `get(i)` baştan ya da sondan yürür. `n` öğeli listede toplam iş $O(n^2)$ olur ve ölçümde görüldüğü gibi büyük listelerde saniyeler sürer. Bağlı liste for-each ya da yineleyiciyle gezilir; bu yollar her adımda bir sonraki düğüme geçer ve toplam iş $O(n)$'dir.

5.2.7 Vector ve Stack

`Vector` ve `Stack` Java 1.0'dan kalmadır; Koleksiyon Çerçevesi Java 1.2 ile geldiğinde bu iki sınıf `List` arayüzüne uyarlanmıştır. Eski kodlarda ve bazı ders kitaplarında hâlâ görülürler. Bu yüzden nasıl çalıştıklarını bilmek gerekir, ama yeni kodda yerlerine daha uygun sınıflar kullanılır.

Vector

`Vector`, `List` arayüzünü uygulayan genişletilebilir bir dinamik dizidir. Belirtilen kapasiteyle, belirtilmezse 10 hücreyle başlar. Dolunca kapasitesi iki katına çıkar; `ArrayList`'in 1,5 katı büyümesinden farkı budur. Asıl ayrım ise metotlarının **senkronize** (*synchronized*) olmasıdır. Her metot çağrılırken nesne kilitlenir. Böylece birden çok iş parçacığı aynı listeye aynı anda güvenle erişebilir. Ama kilit tek iş parçacıklı programlarda da her çağrıda alınıp bırakılır ve bu boş yere zaman harcatır.

```
Vector<Integer> v = new Vector<>();   // kapasite: 10
for (int i = 1; i <= 11; i++) {
    v.addElement(i * 10);
}
System.out.println(v.size() + " / " + v.capacity());
System.out.println(v.elementAt(0));
v.removeElement(110);    // değeri 110 olan öğe
System.out.println(v.size() + " / " + v.capacity());
```

```
11 / 20
10
10 / 20
```

10, 20, ..., 110 değerleri eklenir. On birinci öğe geldiğinde 10 hücrelik dizi dolmuştur ve kapasite iki katına, 20'ye çıkar. `elementAt(0)` ilk öğeyi verir. `removeElement(110)` değeri 110 olan öğeyi siler; boyut 10'a iner, ama kapasite 20'de kalır. Silme kapasiteyi küçültmez. Tablo 5.7 `Vector`'a özgü eski adlı metotları verir. Bu metotların hepsinin `List`'te bir karşılığı vardır: `addElement` yerine `add`, `elementAt` yerine `get`.

Metot	Görevi
`addElement(E obj)`	öğe ekler
`elementAt(int i)`	`i`'deki öğeyi verir
`removeElement(Object obj)`	öğeyi çıkarır
`size()`	öğe sayısı
`capacity()`	mevcut kapasite

Tablo 5.7: `Vector`'a özgü metotlar.

Stack

Tanım 5.6

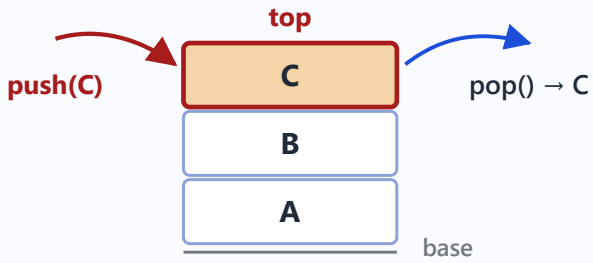
Yığın (*stack*), son giren ilk çıkar (*Last In, First Out*, LIFO) ilkesiyle çalışan yapıdır. En son eklenen öğe en üstte (*top*), en eski eklenen en altta (*base*) durur. Ekleme ve çıkarma yalnızca üstten yapılır.

Yığın günlük hayatta üst üste konmuş tabaklara benzer: tabak hep en üste konur ve en üstten alınır. Programlamada yığının pek çok kullanımı vardır. Metin düzenleyicideki geri alma (*undo*) son yapılan işi ilk geri alır. Bir ifadedeki parantezlerin dengeli olup olmadığı yığınla denetlenir. Derinlik öncelikli arama ziyaret edilecek düğümleri yığında tutar. Metot çağrıları da çağrı yığınında saklanır.

Java'nın `Stack` sınıfı `Vector`'dan türetilmiştir. Tablo 5.8 metotlarını, Şekil 5.9 temel işleyişini verir.

Metot	Görevi
`push(E item)`	en üste ekler
`pop()`	en üsttekini çıkarır ve döndürür
`peek()`	en üsttekini döndürür, çıkarmaz
`empty()`	yığın boş mu?
`search(Object o)`	üstten 1'den başlayan konum; yoksa −1

Tablo 5.8: `Stack` sınıfının metotları.



Şekil 5.9: Yığın. `push(C)` C'yi en üste koyar; `pop()` en üstteki C'yi çıkarır ve döndürür. A en altta (taban) durur.

```
Stack<String> yigin = new Stack<>();
yigin.push("A");
yigin.push("B");
yigin.push("C");
System.out.println(yigin);
System.out.println(yigin.peek());
System.out.println(yigin.pop());
System.out.println(yigin.search("A"));
System.out.println(yigin.empty());
```

```
[A, B, C]
C
C
2
false
```

Tablo 5.9 programı adım adım izler. Yığın sütunu öğeleri alttan üste doğru verir.

Adım	Satır	Değişkenler	Yeni çıktı
1	1: `Stack<String> yigin = new Stack<>();`	yigin=[]	—
2	2: `yigin.push("A");`	yigin=[A]	—
3	3: `yigin.push("B");`	yigin=[A, B]	—

Adım	Satır	Değişkenler	Yeni çıktı
4	4: `yigin.push("C");`	yigin=[A, B, C]	—
5	5: `System.out.println(yigin);`	yigin=[A, B, C]	`[A, B, C]`
6	6: `System.out.println(yigin.peek());`	yigin=[A, B, C]	`C`
7	7: `System.out.println(yigin.pop());`	yigin=[A, B]	`C`
8	8: `System.out.println(yigin.search("A"));`	yigin=[A, B]	`2`
9	9: `System.out.println(yigin.empty());`	yigin=[A, B]	`false`

Tablo 5.9: `Stack` işlemlerinin izlenmesi.

Üç `push`'tan sonra A en altta, C en üsttedir. `println` yığını alttan üste yazar: `[A, B, C]`. Bu sıra ilk bakışta şaşırtabilir, çünkü "üst" sağ uçtadır. `Stack` bir `Vector` olduğu için öğeleri iç dizide eklenme sırasıyla tutar ve en üst, dizinin son hücresidir. `peek()` C'yi döndürür ama çıkarmaz. `pop()` C'yi çıkarır ve döndürür; son giren ilk çıkmıştır. `search("A")` konumu üstten 1'den başlayarak sayar: B 1, A 2'dir. Bu sayım indislerden farklıdır, 0'dan değil 1'den başlar. Son satırda yığında iki öğe vardır ve `empty()` `false` döndürür.

`push`, `pop` ve `peek` iç dizinin sonunda çalıştığı için üçü de O(1)'dir (`push` amortize). Boş bir yığında `pop()` ya da `peek()` çağrılırsa `EmptyStackException` fırlatılır:

```
Exception in thread "main" java.util.EmptyStackException
```

Modern kodda `Stack` yerine `Deque` arayüzünü uygulayan `ArrayDeque` (ya da `LinkedList`) önerilir. `Stack`'in iki sorunu vardır. Birincisi `Vector`'dan gelen gereksiz senkronizasyondur. İkincisi kalıtımın getirdiği fazla metotlardır. `Stack` bir `Vector` olduğu için `add(0, x)` ya da `remove(1)` gibi çağrılar da yapılabilir. Bunlar yığının "yalnızca üstten erişim" kuralını bozar.

5.2.8 Listede gezinme

Bir listenin bütün öğelerini sırayla işlemek için üç yaygın yol vardır:

```
List<String> dersler = new ArrayList<>(
        Arrays.asList("Fizik", "Java", "Devre"));
// 1) İndeksli for
for (int i = 0; i < dersler.size(); i++) {
    System.out.print(dersler.get(i) + " ");
}
System.out.println();
// 2) Gelişmiş for (for-each)
for (String d : dersler) {
    System.out.print(d + " ");
}
System.out.println();
// 3) forEach + lambda (Java 8+)
dersler.forEach(d -> System.out.print(d + " "));
System.out.println();
```

```
Fizik Java Devre
Fizik Java Devre
Fizik Java Devre
```

Birinci yol indisli `for` döngüsüdür. Döngü değişkeni `i` 0'dan `size() - 1`'e kadar gider ve her öğe `get(i)` ile alınır. Bu yol, indisin kendisine ihtiyaç olduğunda kullanılır; örneğin "3. sırada Java var" diye yazmak için. Yalnızca listelerde çalışır, çünkü kümelerde ve kuyruklarda `get(i)` yoktur.

İkinci yol gelişmiş `for`, yani for-each döngüsüdür. En okunur yazımdır ve `Iterable` olan her koleksiyonda çalışır. Derleyici bu döngüyü arka planda bir yineleyiciye çevirir. Yineleyici kesiminde bu dönüşüm ayrıntılı izlenir.

Üçüncü yol `forEach` metodu ve bir lambda ifadesidir (Java 8+). `d -> System.out.print(d + " ")` ifadesi "her `d` öğesi için şunu yap" anlamına gelir. Kısa bir yazımdır, ama döngü gövdesinden dışarıdaki yerel değişkenleri değiştiremez. Lambda içinden bir sayaç artırmak istenirse derleyici durur:

```
LambdaDis.java:6: error: local variables referenced from a lambda expression must be final
or effectively final
        l.forEach(x -> toplam += x);
                       ^
1 error
```

Lambda, dışarıdaki yerel değişkenleri yalnızca okuyabilir; o değişkenler `final` ya da fiilen `final` (bir kez atanıp hiç değişmeyen) olmalıdır. Toplam gibi birikimli değerler için ya for-each kullanılır ya da akışların `sum`, `reduce` gibi işlemleri.

Üç yolun çıktısı aynıdır. Gezinirken listeden öğe silmek gerekiyorsa üçü de uygun değildir. Bu iş için yineleyicinin kendi `remove` metodu kullanılır.

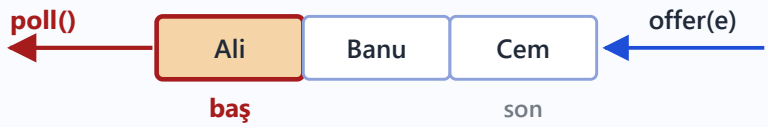
5.3 Kuyruklar

Bir kuyruk, işlenmeyi bekleyen öğeleri tutar. Yazıcıya gönderilen belgeler, bir sunucuya gelen istekler, bir oyunda sırası gelecek hamleler hep bir kuyrukta bekler. Kuyruğun asıl sorusu "sıradaki kim?" sorusudur. Bu sorunun yanıtı yapıya göre değişir: sıradan bir kuyrukta ilk gelen, öncelikli kuyrukta en acil olan, yığında ise en son gelen sıradadır. Bu kesim bu üç davranışı sırasıyla `Queue`, `PriorityQueue` ve `Deque` üzerinden anlatır.

5.3.1 Queue arayüzü

> **Tanım 5.7**
>
> **Kuyruk** (*queue*), ilk giren ilk çıkar (FIFO) ilkesiyle çalışan yapıdır. Yeni öğe kuyruğun sonuna girer; işlenecek öğe kuyruğun başından alınır. Öğeler kuyruğa eklendikleri sırayla çıkar.

Benzetme olarak market kasası düşünülebilir. Sıraya en sondan girilir, ödemeyi en öndeki yapar. Kimse sıranın ortasına giremez ve ortadan çıkamaz. Şekil 5.10 bu düzeni gösterir.



Şekil 5.10: Kuyruk. `offer(e)` sona ekler, `poll()` baştakini (Ali) çıkarır.

`Queue` arayüzünün üç bilinen gerçeklemesi vardır. `LinkedList` çift yönlü bağlı listeyle, `ArrayDeque` dinamik diziyle çalışır; ikisi de gerçek FIFO kuyruğudur. `PriorityQueue` ise öğeleri önceliğe göre çıkarır; bu yüzden FIFO değildir. Kuyruğun tipik kullanımları yazıcı kuyruğu, görev sırası ve genişlik öncelikli aramadır.

Her işlemin iki biçimi

`Queue` arayüzü her temel işlemi iki biçimde sunar. Bir biçim başarısızlıkta istisna fırlatır, öteki özel bir değer döndürür. Tablo 5.10 bu ikiliği verir.

İşlem	Hata fırlatır	Özel değer döner
Ekle (sona)	`add(e)`	`offer(e)` → `false`
Çıkar (baştan)	`remove()`	`poll()` → `null`
Bak (baştakine)	`element()`	`peek()` → `null`

Tablo 5.10: `Queue` metotlarının iki biçimi.

Kuyruk boşken `remove()` ve `element()` `NoSuchElementException` fırlatır; `poll()` ve `peek()` ise `null` döndürür. Ekleme tarafında fark ancak kapasitesi sınırlı kuyruklarda ortaya çıkar. Dolu kuyrukta `add` istisna fırlatır, `offer` ise `false` döndürür. `LinkedList` ve `ArrayDeque` sınırsız büyüdüğü için bu ikisinde ekleme hep başarılı olur. Boş kuyruktaki davranış farkı aşağıda görülür:

```
import java.util.*;
public class BosKuyruk {
    public static void main(String[] args) {
        Queue<String> k = new LinkedList<>();
        System.out.println(k.poll() + " " + k.peek());
        try { k.element(); } catch (RuntimeException e) { System.out.println("element: " +
e); }
        k.remove();
    }
}
```

```
null null
element: java.util.NoSuchElementException
Exception in thread "main" java.util.NoSuchElementException
```

Kuyruğu izlemek

```
Queue<String> kuyruk = new LinkedList<>();
kuyruk.offer("Ali");
kuyruk.offer("Banu");
kuyruk.offer("Cem");
System.out.println(kuyruk);
System.out.println(kuyruk.peek());
System.out.println(kuyruk.poll());
System.out.println(kuyruk.poll());
System.out.println(kuyruk.size());
System.out.println(kuyruk.poll());
System.out.println(kuyruk.poll());
```

```
[Ali, Banu, Cem]
Ali
Ali
Banu
1
Cem
null
```

Değişken `Queue` tipinde, nesne ise `LinkedList`'tir. Bu sayede yalnızca kuyruk metotları kullanılabilir; `kuyruk.get(1)` gibi kuyruk düzenini bozacak bir çağrı derlenmez. Tablo 5.11 her adımda kuyruğun içeriğini verir.

Adım	Satır	Kuyruk (baştan sona)	Yeni çıktı
1	1: `Queue<String> kuyruk = new LinkedList<>();`	boş	—
2	2: `kuyruk.offer("Ali");`	Ali	—
3	3: `kuyruk.offer("Banu");`	Ali, Banu	—
4	4: `kuyruk.offer("Cem");`	Ali, Banu, Cem	—
5	5: `System.out.println(kuyruk);`	Ali, Banu, Cem	`[Ali, Banu, Cem]`
6	6: `System.out.println(kuyruk.peek());`	Ali, Banu, Cem	`Ali`
7	7: `System.out.println(kuyruk.poll());`	Banu, Cem	`Ali`
8	8: `System.out.println(kuyruk.poll());`	Cem	`Banu`
9	9: `System.out.println(kuyruk.size());`	Cem	`1`
10	10: `System.out.println(kuyruk.poll());`	boş	`Cem`
11	11: `System.out.println(kuyruk.poll());`	boş	`null`

Tablo 5.11: `Queue` işlemlerinin izlenmesi. Kuyruk sütunu öğeleri baştan sona verir.

Üç `offer` Ali, Banu ve Cem'i sırayla sona ekler. `peek()` baştakini (Ali) döndürür, ama kuyruğu değiştirmez. İlk `poll()` Ali'yi çıkarır ve döndürür; ilk giren ilk çıkmıştır. İkinci `poll()` Banu'yu çıkarır. Bu noktada `size()` 1'dir. Üçüncü `poll()` Cem'i çıkarır ve kuyruk boşalır. Dördüncü `poll()` hata vermez, `null` döndürür. Kuyruğun başı ve sonu sabit zamanda bulunduğu için `offer`, `poll` ve `peek` O(1)'dir.

5.3.2 PriorityQueue ve ikili yığın

> **Tanım 5.8**
>
> **Öncelikli kuyruk** (*priority queue*), çıkan öğenin her zaman en öncelikli öğe olduğu kuyruktur; eklenme sırası belirleyici değildir. `PriorityQueue`'da varsayılan öncelik doğal sıradır: en küçük öğe önce çıkar.

Öncelikli kuyruğun tipik kullanımları görev çizelgeleme, acil servis sırası ve en kısa yol algoritmalarıdır. Acil serviste hastalar geliş sırasına göre değil, durumlarının ağırlığına göre çağrılır. İşletim sistemi de öncelikli işleri öne alır. Öğelerin karşılaştırılabilir olması gerekir. Ya öğe sınıfı `Comparable` arayüzünü uygular ya da kuyruğa bir `Comparator` verilir. Ters sıra, yani en büyüğün önce çıkması için de bir karşılaştırıcı verilir. Tablo 5.12 temel metotları listeler.

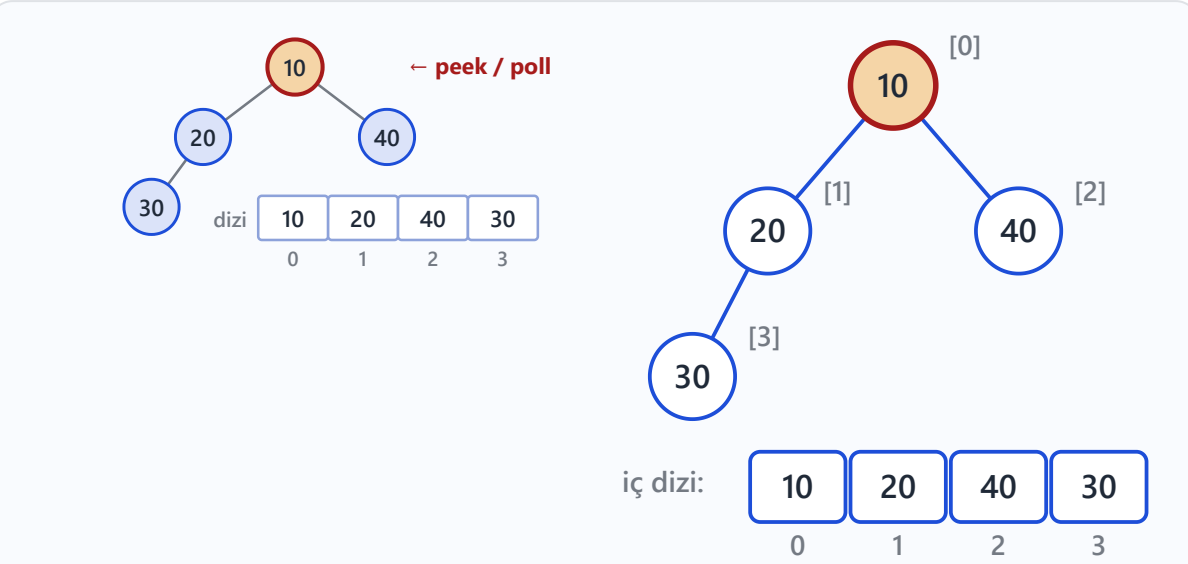
Metot	Görevi	Maliyet
`add(E e)`	öğe ekler	O(log n)

Metot	Görevi	Maliyet
`poll()`	en öncelikliyi çıkarır ve döndürür	O(log n)
`peek()`	en öncelikliyi döndürür, çıkarmaz	O(1)
`remove(Object o)`	belirtilen öğeyi çıkarır	O(n)
`size()`	öğe sayısı	O(1)

Tablo 5.12: `PriorityQueue` metotları ve maliyetleri.

`PriorityQueue` içte bir **ikili yığın** (*binary heap*) kullanır. Bu "yığın" sözcüğü LIFO yığınıyla aynı adı taşır, ama başka bir yapıdır. İkili yığın, bir dizide saklanan tam bir ikili ağaçtır. Ağacın kuralı şudur: her ebeveyn çocuklarından küçük ya da onlara eşittir. Bu kural yalnızca ebeveyn ile çocuk arasındadır; kardeşler arasında bir sıra yoktur. Kuralın sonucu olarak en küçük öğe her zaman köktedir.

Ağaç bir dizide şöyle tutulur. Kök indis 0'dadır. İndis `i`'deki düğümün çocukları `2i + 1` ve `2i + 2`'de, ebeveyni `(i - 1) / 2`'dedir. Böylece hiçbir referans tutmadan ağaçta aşağı ve yukarı gidilebilir. Şekil 5.11 dört öğeli bir yığını hem ağaç hem dizi olarak gösterir.



Şekil 5.11: İkili yığın. Solda kök en küçük öğedir; `peek` ve `poll` kökü kullanır. Sağda 30, 10, 40, 20 eklendikten sonraki iç düzen: dizi `[10, 20, 40, 30]`. Ağaçta 10 köktür; 20 ve 40 onun çocukları, 30 ise 20'nin çocuğudur.

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.add(30);
pq.add(10);
pq.add(40);
pq.add(20);
System.out.println(pq.peek());   // en küçük
System.out.println(pq);          // iç düzen (yığın)
while (!pq.isEmpty()) {
    System.out.print(pq.poll() + " ");
}
System.out.println();
```

```
10
[10, 20, 40, 30]
10 20 30 40
```

Çıktının ikinci satırı sık sık şaşırtır. `println(pq)` öğeleri sıralı yazmaz; iç diziyi olduğu gibi yazar. İç dizi yalnızca “her ebeveyn çocuklarından küçük” kuralına uyar: 10 kendi çocukları 20 ve 40'tan, 20 de kendi çocuğu 30'dan küçüktür. Ama 40 ile 30 arasında bir sıra yoktur. Sıralı sonuç için öğeler `poll` ile tek tek çıkarılır; çıkış sırası her zaman küçükten büyüğedir.

Yığın kuralını adım adım izlemek

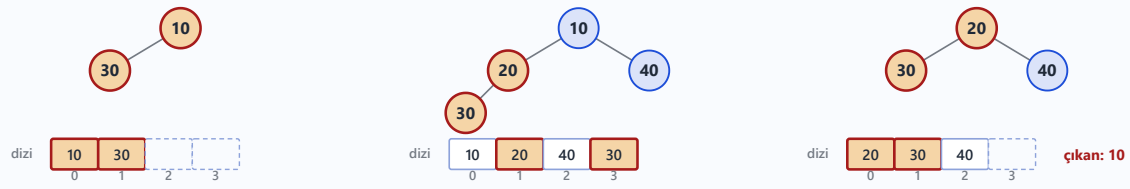
Her ekleme ve çıkarmanın içeride ne yaptığını Tablo 5.13 gösterir. İç dizi sütunu her adımdan sonraki diziyi verir; değerler aynı programın her adımda iç diziyi yazdıran bir sürümüyle doğrulanmıştır.

Adım	Satır	İç dizi	Değişkenler	Yeni çıktı
1	1: `PriorityQueue<Integer> pq = new PriorityQueue<>();`	[]	—	—
2	2: `pq.add(30);`	[30]	—	—
3	3: `pq.add(10);`	[10, 30]	—	—
4	4: `pq.add(40);`	[10, 30, 40]	—	—
5	5: `pq.add(20);`	[10, 20, 40, 30]	—	—
6	6: `System.out.println(pq.peek());`	[10, 20, 40, 30]	—	`10`
7	7: `System.out.println(pq);`	[10, 20, 40, 30]	—	`[10, 20, 40, 30]`
8	9: `System.out.print(pq.poll() + " ");`	[20, 30, 40]	x=10	`10`
9	9: `System.out.print(pq.poll() + " ");`	[30, 40]	x=20	`20`
10	9: `System.out.print(pq.poll() + " ");`	[40]	x=30	`30`
11	9: `System.out.print(pq.poll() + " ");`	[]	x=40	`40`

Adım	Satır	İç dizi	Değişkenler	Yeni çıktı
12	11: `System.out.println();`	[]	—	—

Tablo 5.13: `PriorityQueue`'nun iç dizisinin izlenmesi.

Ekleme her zaman dizinin sonuna yapılır, sonra yeni öğe gerektiği kadar **yukarı çıkar**. 30 boş diziye girer ve kök olur. 10 indis 1'e eklenir. Ebeveyni indis 0'daki 30'dur ve 10 ondan küçüktür; kural bozulduğu için ikisi yer değiştirir. Kök artık 10'dur. 40 indis 2'ye girer; ebeveyni olan 10'dan büyük olduğu için yerinde kalır. 20 indis 3'e girer. Ebeveyni indis 1'deki 30'dur. 20 daha küçük olduğu için yer değiştirirler. 20 şimdi indis 1'dedir; ebeveyni olan 10'dan büyük olduğu için durur. Şekil 5.12 bu adımları ağaç üzerinde gösterir.



Şekil 5.12: İkili yığında üç an. Solda 10 eklenmiş ve 30 ile yer değiştirerek köke çıkmıştır. Ortada 20 eklenmiş, 30 ile yer değiştirmiş ve 10'un altında durmuştur. Sağda ilk `poll()`: kök 10 çıkar, son öğe 30 köke gelir ve küçük çocuğu 20 ile yer değiştirip aşağı iner.

`peek()` yalnızca köke bakar ve O(1)'dir. Çıkarma ise şöyle yapılır. Kök çıkarılır, dizinin son öğesi köke taşınır ve gerektiği kadar **aşağı iner**. İlk `poll()`'da 10 çıkar, son öğe 30 köke gelir. 30'un çocukları 20 ve 40'tır; küçük olan 20 ile yer değiştirir. Dizi `[20, 30, 40]` olur. İkinci `poll()`'da 20 çıkar, 40 köke gelir ve küçük çocuğu 30 ile yer değiştirir: `[30, 40]`. Sonra 30 ve 40 sırayla çıkar ve kuyruk boşalır. Çıkış sırası 10 20 30 40'tır.

Yukarı çıkma ve aşağı inme yalnızca bir dalı izler. `n` öğeli tam ikili ağacın yüksekliği yaklaşık $\log_2 n$'dir. Bu yüzden `add` ve `poll` O(log n)'dir. Bir milyon öğeli bir yığında bile her işlem en çok 20 kadar karşılaştırma yapar. `remove(Object o)` ise önce öğeyi bulmak için diziyi tarar ve O(n)'dir.

Sık yapılan hata

Öncelikli kuyruğu for-each ile gezip sıralı sonuç beklemek. `for (int x : pq)` ve `println(pq)` iç diziyi dizi sırasıyla verir; sonuç sıralı değildir. Öğeleri öncelik sırasıyla almak için `while (!pq.isEmpty()) pq.poll()` kalıbı kullanılır. Bu kalıp kuyruğu boşaltır; kuyruk sonra yeniden gerekiyorsa önce bir kopyası alınır.

5.3.3 Önceliği belirlemek

Sayılar ve dizgiler zaten karşılaştırılabilir; doğal sıraları bellidir. Kendi sınıflarımızın nesneleri ise ancak bir sıra tanımlanırsa öncelikli kuyruğa girebilir. Bunun birinci yolu sınıfın `Comparable` arayüzünü uygulamasıdır:

```
class Gorev implements Comparable<Gorev> {
    String ad;
    int oncelik;          // küçük = acil

    Gorev(String ad, int oncelik) {
        this.ad = ad;
        this.oncelik = oncelik;
    }

    @Override
    public int compareTo(Gorev diger) {
        return Integer.compare(oncelik, diger.oncelik);
    }

    @Override
    public String toString() {
        return ad + "(" + oncelik + ")";
    }
}
```

```
PriorityQueue<Gorev> gorevler =
        new PriorityQueue<>();
gorevler.add(new Gorev("Rapor", 3));
gorevler.add(new Gorev("Arıza", 1));
gorevler.add(new Gorev("Toplantı", 2));
while (!gorevler.isEmpty()) {
    System.out.println(gorevler.poll());
}
```

```
Arıza(1)
Toplantı(2)
Rapor(3)
```

`Gorev` sınıfı `Comparable<Gorev>`'u uygular ve tek bir metot yazar: `compareTo`. Bu metot iki görevi karşılaştırır. `this` önce gelecekse negatif, eşitse 0, sonra gelecekse pozitif bir sayı döndürür. `Integer.compare(oncelik, diger.oncelik)` tam olarak bunu yapar. Burada küçük öncelik numarası acil anlamına gelir. Arıza (1) en önce, Toplantı (2) sonra, Rapor (3) en son çıkar. Eklenme sırasının hiçbir önemi yoktur.

Doğal sıranın tersi gerekiyorsa sınıfı değiştirmeye gerek yoktur. Kuyruğun yapıcısına bir karşılaştırıcı verilir. `Collections.reverseOrder()` doğal sırayı tersine çeviren hazır bir karşılaştırıcıdır:

```
PriorityQueue<Integer> buyukOnce =
        new PriorityQueue<>(
        Collections.reverseOrder());
buyukOnce.add(30);
buyukOnce.add(10);
buyukOnce.add(40);
System.out.println(buyukOnce.poll());
```

```
40
```

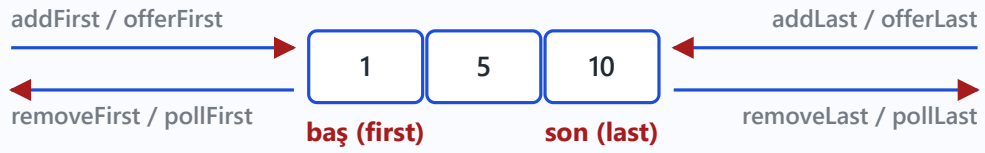
Bu kuyrukta en büyük öğe köktedir ve önce çıkar. Karşılaştırıcılar algoritmalar kesiminde ayrıntılı işlenir.

Öğe sınıfı ne `Comparable`'ı uyguluyor ne de kuyruğa karşılaştırıcı verilmişse program derlenir, ama ilk eklemede `ClassCastException` fırlatılır.

5.3.4 Deque ve ArrayDeque

Tanım 5.9

Çift uçlu kuyruk (*Double-Ended Queue*, `Deque`, "dek" okunur), hem baştan hem sondan öğe eklenebilen ve çıkarılabilen kuyruktur. Kuyruğun (FIFO) ve yığının (LIFO) işlevlerini tek yapıda birleştirir.



Şekil 5.13: Çift uçlu kuyruk. Baş tarafta `addFirst` / `offerFirst` ekler, `removeFirst` / `pollFirst` çıkarır; son tarafta `addLast` / `offerLast` ve `removeLast` / `pollLast` aynı işleri yapar.

`Deque` arayüzünün iki gerçeklemesi vardır: `LinkedList` (çift yönlü bağlı liste) ve `ArrayDeque` (dinamik dizi). Her iki uç için aynı üç işlem bulunur: ekle, çıkar ve bak. Tablo 5.14 bu metotları toplar. `Queue`'da olduğu gibi her işlemin istisna fırlatan ve özel değer döndüren iki biçimi vardır. `remove…` ve `get…` boş deque'de istisna fırlatır; `poll…` ve `peek…` `null` döndürür.

İşlem	Baş uçta	Son uçta
Ekle	`addFirst(e)` (`offerFirst`)	`addLast(e)` (`offerLast`)
Çıkar ve döndür	`removeFirst()` (`pollFirst`)	`removeLast()` (`pollLast`)
Döndür, çıkarma	`getFirst()` (`peekFirst`)	`getLast()` (`peekLast`)

Tablo 5.14: `Deque` metotları. Parantez içindeki biçimler boş deque'de istisna yerine `null` döndürür.

`Deque` ayrıca yığın ve kuyruk arayüzlerinin tanıdık adlarını da taşır. Yığın işlemleri baş uçta çalışır: `push` = `addFirst`, `pop` = `removeFirst`, `peek` = `peekFirst`. Kuyruk işlemlerinde `offer` sona ekler (`offerLast`), `poll` baştan alır (`pollFirst`). Bu eşleşmeler akılda tutulursa bir deque'in içeriği kolayca izlenir.

ArrayDeque

`ArrayDeque`, çift uçlu kuyruğu **dairesel** bir diziyle uygular. Dizi iki indis tutar: baş (`head`) ve son (`tail`). Başa ekleme baş indisini bir geri alır, sona ekleme son indisini bir ileri alır. İndis dizinin bir ucundan taşarsa öbür uca sarar. Böylece her iki uçta da öğe kaydırmadan O(1) ekleme ve çıkarma yapılır. Dizi dolunca daha büyük bir diziye kopyalanır, tıpkı `ArrayList`'teki gibi. Şekil 5.14 bu düzeni gösterir.



Şekil 5.14: `ArrayDeque`'in dairesel dizisi. Baş indisi (head) 6'da, son öğe indis 0'dadır. Baştan sona okuma 6, 7 ve sonra başa sararak 0 numaralı hücreyi izler: deque 1, 5, 10'dur.

`ArrayDeque` çoğu durumda hem `Stack`'ten hem `LinkedList`'ten hızlı ve hafiftir. `Stack` gibi her çağrıda kilit almaz. `LinkedList` gibi her öğe için ayrı bir düğüm nesnesi oluşturmaz. Bu nedenle Java belgeleri yığın ve kuyruk ihtiyacı için `ArrayDeque`'i önerir. Tek kısıtı `null` öğe kabul etmemesidir. `poll` boş deque'de `null` döndürdüğü için `null` öğe bu dönüş değeriyle karışırdı; sınıf bu yüzden `null`'ı baştan reddeder:

```
import java.util.*;
public class NullDeque {
    public static void main(String[] args) {
        Deque<String> d = new ArrayDeque<>();
        d.offer(null);
    }
}
```

```
Exception in thread "main" java.lang.NullPointerException
```

Yığın olarak kullanmak için değişken `Deque` tipinde tanımlanır:

```
Deque<String> yigin = new ArrayDeque<>();
yigin.push("A");     // başa ekler
yigin.push("B");
System.out.println(yigin.pop());
```

```
B
```

A ve B başa itilir; B en üsttedir ve `pop()` onu döndürür. `Stack` alttan üste yazar; `ArrayDeque` ise baştan sona, yani üstten alta yazar: aynı A, B, C itmelerinden sonra biri `[A, B, C]`, öteki `[C, B, A]` verir.

ArrayDeque'i izlemek

```
ArrayDeque<Integer> sayilar = new ArrayDeque<>();
sayilar.addFirst(5);
sayilar.addLast(10);
sayilar.addFirst(1);
System.out.println(sayilar);
int ilkSayi = sayilar.pollFirst();
int sonSayi = sayilar.pollLast();
System.out.println(ilkSayi + " " + sonSayi);
System.out.println(sayilar.peekFirst());
```

```
[1, 5, 10]
1 10
5
```

Tablo 5.15 programı adım adım izler.

Adım	Satır	Deque (baştan sona)	Değişkenler	Yeni çıktı
1	1: `ArrayDeque<Integer> sayilar = new ArrayDeque<>();`	boş	—	—
2	2: `sayilar.addFirst(5);`	5	—	—
3	3: `sayilar.addLast(10);`	5, 10	—	—
4	4: `sayilar.addFirst(1);`	1, 5, 10	—	—

Adım	Satır	Deque (baştan sona)	Değişkenler	Yeni çıktı
5	5: `System.out.println(sayilar);`	1, 5, 10	—	`[1, 5, 10]`
6	6: `int ilkSayi = sayilar.pollFirst();`	5, 10	ilkSayi=1	—
7	7: `int sonSayi = sayilar.pollLast();`	5	ilkSayi=1, sonSayi=10	—
8	8: `System.out.println(ilkSayi + " " + sonSayi);`	5	ilkSayi=1, sonSayi=10	`1 10`
9	9: `System.out.println(sayilar.peekFirst());`	5	ilkSayi=1, sonSayi=10	`5`

Tablo 5.15: `ArrayDeque` işlemlerinin izlenmesi. Deque sütunu öğeleri baştan sona verir.

`addFirst(5)` tek öğeyi ekler; 5 hem baştır hem son. `addLast(10)` 10'u sona ekler. `addFirst(1)` 1'i başa ekler; 5 ve 10 yerlerinde kalır, hiçbir öğe kaymaz. Deque baştan sona `[1, 5, 10]` olarak yazılır. `pollFirst()` baştaki 1'i çıkarır. Metot bir `Integer` döndürür; `int ilkSayi`'ye atanırken otomatik kutudan çıkarma yapılır. `pollLast()` sondaki 10'u çıkarır. İki değer yazılır: `1 10`. Son olarak `peekFirst()` baştaki 5'i döndürür ama çıkarmaz.

Bu programın ince bir tuzağı vardır. `pollFirst()` boş deque'de `null` döndürür. Sonuç doğrudan bir `int`'e atanırsa kutudan çıkarma `null` üzerinde yapılır ve program çöker:

```
import java.util.*;
public class BosDeque {
    public static void main(String[] args) {
        ArrayDeque<Integer> d = new ArrayDeque<>();
        int x = d.pollFirst();
    }
}
```

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.lang.Integer.intValue()" because the return value of
"java.util.ArrayDeque.pollFirst()" is null
```

"Özel değer döndüren" biçimin güvenliği burada kaybolur. `poll` ailesi kullanılırken dönüş değeri önce `Integer` olarak alınıp `null` denetimi yapılır ya da çağrıdan önce `isEmpty()` sorulur.

5.4 Kümeler

Bir öğrencinin aldığı derslerin kodları, bir metindeki farklı kelimeler, bir ağdaki ziyaret edilmiş düğümler: bunların ortak noktası tekrarın anlamsız olmasıdır. Aynı ders iki kez alınmaz, aynı kelime farklı kelime sayısını artırmaz. Küme bu tür veriler için tasarlanmıştır. Kümenin asıl gücü "bu öğe var

mı?" sorusunu çok hızlı yanıtlamasıdır. Bu kesim kümenin sözleşmesini, hash tablosunun işleyişini, kendi sınıflarımızın kümede doğru davranması için gereken `equals` ve `hashCode` kurallarını ve sıralı kümeyi anlatır.

5.4.1 Set arayüzü

Tanım 5.10

Küme (*set*), tekrar eden öğeye izin vermeyen koleksiyondur. Matematikteki küme kavramına karşılık gelir: her öğeden yalnızca bir tane bulunur. Var olan bir öğe eklenmek istendiğinde `add` öğeyi eklemez ve `false` döndürür.

`Set` arayüzü genel olarak bir sıra garantisi vermez; gezinme sırası gerçeklemeye bağlıdır. Üç gerçekleme üç farklı sıra sunar. `HashSet` sırasızdır. `LinkedHashSet` eklenme sırasını korur. `TreeSet` öğeleri sıralı tutar; sıra doğal sıra ya da verilen bir `Comparator`'dır. `Set`, `Collection`'a yeni metot eklemez; aynı metotları "tekrar yok" kuralıyla uygular.

```
Set<String> meyveler = new HashSet<>();
System.out.println(meyveler.add("Elma"));
System.out.println(meyveler.add("Armut"));
System.out.println(meyveler.add("Elma"));
System.out.println(meyveler.size());
System.out.println(meyveler.contains("Armut"));
meyveler.remove("Armut");
System.out.println(meyveler);
```

```
true
true
false
2
true
[Elma]
```

İlk iki `add` yeni öğe eklediği için `true` döndürür. Üçüncü `add("Elma")` var olan bir öğeyi eklemeye çalışır; küme değişmez ve sonuç `false`'tur. Bu yüzden `size()` 3 değil, 2'dir. `contains("Armut")` `true` döndürür. Armut silindikten sonra kümede yalnızca Elma kalır. `add`'in dönüş değeri pratikte de işe yarar: "bu kelimeyi daha önce gördük mü?" sorusu ayrı bir `contains` çağrısı yapmadan, tek bir `add` ile yanıtlanır.

5.4.2 HashSet ve hash tablosu

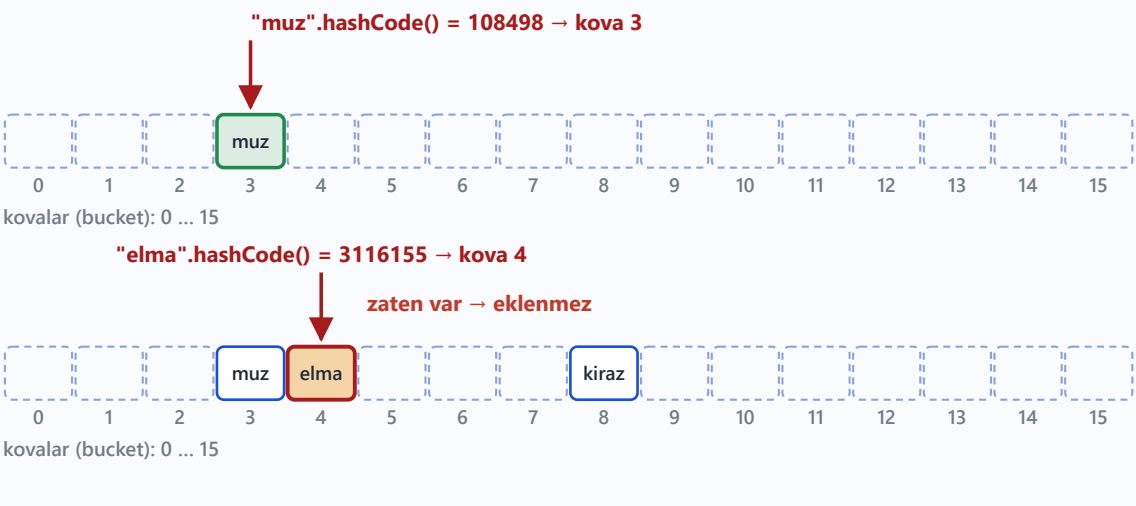
> **Tanım 5.11**
>
> **Hash tablosu** (*hash table*), öğeleri **kova** (*bucket*) adı verilen hücrelere dağıtan yapıdır. Her öğenin `hashCode()` metodu bir tam sayı üretir; bu sayıdan bir kova numarası hesaplanır ve öğe doğrudan o kovaya gider. Ekleme ve arama ortalamada O(1) sürer.

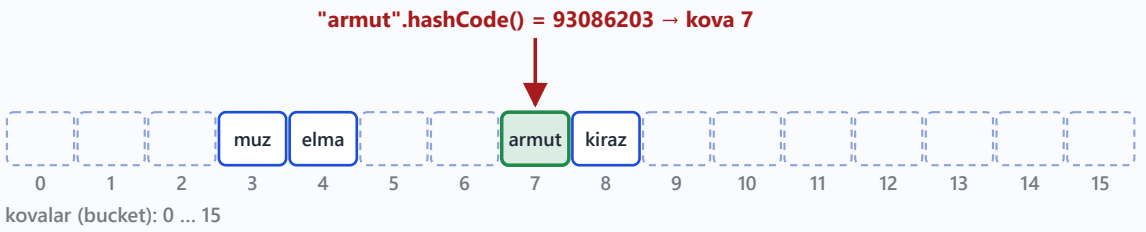
`HashSet` tekil öğeleri sırasız saklar. `AbstractSet`'ten türetilmiştir ve `Set` arayüzünü uygular. İçeride aslında bir `HashMap` kullanır: kümenin öğeleri eşlemenin anahtarlarıdır. Bir öğeyi aramak için listedeki gibi baştan sona taramak gerekmez. Öğenin kovası hesaplanır ve yalnız o kovaya bakılır.

Kova numarası iki adımda bulunur. Önce öğenin `hashCode()` değeri alınır. Java, bu değerin üst 16 bitini alt 16 bitiyle karıştırır: `h ^ (h >>> 16)`. Sonra sonucun son bitleri kova numarası olur. 16 kovalı bir tabloda son 4 bit kullanılır: `(h ^ (h >>> 16)) & 15`. Notun ölçüm programı dört meyvenin kovasını bu formülle hesaplar:

```
muz hashCode=108498 kova=3
elma hashCode=3116155 kova=4
kiraz hashCode=102057485 kova=8
armut hashCode=93086203 kova=7
Zeynep hashCode=-1621354421 kova=7
Ahmet hashCode=63235125 kova=1
Mert hashCode=2394618 kova=14
Elma hashCode=2162843 kova=10
Armut hashCode=63533531 kova=2
```

İlk dört satır sunumdaki kova numaralarını doğrular: muz 3, elma 4, kiraz 8, armut 7. Aynı formül eşlemeler kesimindeki rehber örneğinde de kullanılacaktır. Şekil 5.15 bu öğelerin tabloya yerleşmesini gösterir.





Şekil 5.15: 16 kovalı bir `HashSet`. Üstte `add("muz")`: hash değerinden kova 3 hesaplanır. Ortada ikinci `add("elma")`: kova 4'te `equals` ile eşit bir öğe zaten vardır, öğe eklenmez ve `add` `false` döner. Altta `add("armut")` kova 7'ye yerleşir; gezinme kovaların sırasıyla yapılır: `[muz, elma, armut, kiraz]`.

İkinci `add("elma")` çağrısı kümenin "tekrar yok" kuralının nasıl uygulandığını gösterir. Öğenin kovası hesaplanır: 4. O kovadaki öğelerle `equals` ile karşılaştırılır. Eşit bir öğe bulunduğu için ekleme yapılmaz. Bütün küme taranmaz; yalnızca tek bir kovaya bakılır.

Gezinme sırası kovaların sırasıdır. Tablo 0'dan 15'e taranır ve dolu kovalardaki öğeler sırayla verilir. Bu yüzden `HashSet`'in sırası eklenme sırasıyla ilgisizdir. Kova sayısı değişirse sıra da değişebilir. Bu sıraya hiçbir programda güvenilmemelidir.

İki farklı öğenin kova numarası aynı çıkabilir; buna **çakışma** (*collision*) denir. Örneğin yukarıdaki çıktıda armut ile Zeynep'in kovası 7'dir. Çakışan öğeler aynı kovada küçük bir zincir oluşturur ve arama o zincirde `equals` ile yapılır. Java 8'den beri, bir kovadaki zincir çok uzarsa (8 öğeden fazla ve tablo yeterince büyükse) zincir dengeli bir ağaca çevrilir. Tablodaki öğe sayısı kova sayısının %75'ini (yük faktörü 0,75) aşınca tablo iki katına büyür ve öğeler yeni kovalara dağıtılır. Böylece zincirler kısa kalır ve ortalama maliyet O(1) olur.

5.4.3 LinkedHashSet ve üç kümenin sırası

`LinkedHashSet`, `HashSet`'ten türer ve öğeleri ek olarak bir bağlı listeyle birbirine bağlar. Hash tablosu hızlı erişimi ve tekilliği sağlar, bağlı liste ise eklenme sırasını korur. Hem hız hem sıra önemliyse `LinkedHashSet` seçilir. Sıra önemli değilse ve hız kritikse `HashSet` yeterlidir. Temel metotları aynıdır: `add`, `remove`, `contains`, `size`, `clear`.

```
LinkedHashSet<Integer> sayilar =
        new LinkedHashSet<>();
sayilar.add(5);
sayilar.add(10);
sayilar.add(5);         // tekrar: eklenmez
sayilar.add(1);
System.out.println(sayilar);
```

```
[5, 10, 1]
```

İkinci `add(5)` tekrar olduğu için eklenmez. Sonraki 1 listenin sonuna bağlanır. Çıktı eklenme sırasıdır: önce 5, sonra 10, sonra 1. Aynı kelimeleri üç kümeye birden ekleyince fark açıkça görülür:

```
String[] kelimeler = {"muz", "elma", "kiraz", "elma", "armut"};
Set<String> h = new HashSet<>();
Set<String> l = new LinkedHashSet<>();
Set<String> t = new TreeSet<>();
for (String k : kelimeler) {
    h.add(k);
    l.add(k);
    t.add(k);
}
System.out.println("HashSet:       " + h);
System.out.println("LinkedHashSet: " + l);
System.out.println("TreeSet:       " + t);
```

```
HashSet:       [muz, elma, armut, kiraz]
LinkedHashSet: [muz, elma, kiraz, armut]
TreeSet:       [armut, elma, kiraz, muz]
```

Beş kelimeden biri tekrardır (ikinci “elma”); üç kümede de dört öğe vardır. `HashSet` kovaların sırasını verir: muz 3, elma 4, armut 7, kiraz 8. `LinkedHashSet` eklenme sırasını korur. `TreeSet` alfabetik sıra verir. Tablo 5.16 üç sınıfı karşılaştırır.

Sınıf	Gezinme sırası	Temel yapı	`add` / `contains`
`HashSet`	garanti yok (kova sırası)	hash tablosu	O(1) ortalama
`LinkedHashSet`	eklenme sırası	hash tablosu + bağlı liste	O(1) ortalama
`TreeSet`	sıralı (küçükten büyüğe)	kırmızı-siyah ağaç	O(log n)

Tablo 5.16: Üç küme sınıfının karşılaştırması.

5.4.4 Kendi sınıflarımız: equals ve hashCode

Dizgi ve sayı kümeleri beklendiği gibi çalışır, çünkü `String` ve `Integer` sınıfları `equals` ve `hashCode` metotlarını değere göre yazmıştır. Kendi sınıflarımızda bu metotlar yazılmazsa `Object`'ten devralınan sürümler kullanılır. `Object`'in `equals`'ı iki referansın aynı nesneyi gösterip göstermediğine bakar. Alan değerleri aynı olan iki ayrı nesne bu yüzden “farklı” sayılır.

```
class Nokta {
    int x, y;
    Nokta(int x, int y) {
        this.x = x;
        this.y = y;
    }
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof Nokta)) return false;
        Nokta n = (Nokta) o;
        return x == n.x && y == n.y;
    }
    @Override
    public int hashCode() {
        return Objects.hash(x, y);
    }
}
```

```
Set<NoktaA> s1 = new HashSet<>();
s1.add(new NoktaA(1, 2));
s1.add(new NoktaA(1, 2));
System.out.println(s1.size());

Set<Nokta> s2 = new HashSet<>();
s2.add(new Nokta(1, 2));
s2.add(new Nokta(1, 2));
System.out.println(s2.size());
```

```
2
1
```

`NoktaA` aynı iki alanlı sınıftır, ama `equals` ve `hashCode` yazılmamıştır. `new NoktaA(1, 2)` iki kez çalışınca bellekte iki ayrı nesne oluşur. `Object`'in `equals`'ı bunları farklı sayar ve küme boyutu 2 olur. `Nokta` sınıfında ise iki metot da geçersiz kılınmıştır. `equals` alanları karşılaştırır. `hashCode`, `Objects.hash(x, y)` ile aynı alanlardan bir sayı üretir. Aynı nokta kümeye bir kez girer ve boyut 1 olur.

Tanım 5.12

`equals` – `hashCode` **sözleşmesi**: `a.equals(b)` doğruysa `a.hashCode() == b.hashCode()` da doğru olmalıdır. Tersi gerekmez: hash kodu eşit olan iki nesne farklı olabilir (çakışma). Bu yüzden `equals`'ı geçersiz kılan her sınıf `hashCode`'u da geçersiz kılar ve ikisi aynı alanlardan hesaplanır.

Kuralın gerekçesi hash tablosunun çalışma biçimindedir. Önce kova `hashCode` ile bulunur, sonra o kovadaki öğeler `equals` ile karşılaştırılır. Eşit iki nesnenin hash kodları farklıysa farklı kovalara giderler. `equals` hiç çağrılmaz ve ikisi de kümeye girer. Yalnızca `equals`'ı yazıp `hashCode`'u unutan bir sınıf tam olarak bu hatayı yapar:

```
// yalnız equals yazılmış, hashCode unutulmuş
class NoktaE {
    int x, y;
    NoktaE(int x, int y) { this.x = x; this.y = y; }
    @Override
    public boolean equals(Object o) {
        if (!(o instanceof NoktaE)) return false;
        NoktaE n = (NoktaE) o;
        return x == n.x && y == n.y;
    }
}
```

```
import java.util.*;
public class YalnizEquals {
    public static void main(String[] args) {
        NoktaE a = new NoktaE(1, 2), b = new NoktaE(1, 2);
        Set<NoktaE> s = new HashSet<>();
        s.add(a);
        s.add(b);
        System.out.println(a.equals(b) + " " + s.size() + " " + s.contains(new NoktaE(1,
2)));
        List<NoktaE> l = new ArrayList<>();
        l.add(a);
        System.out.println(l.contains(new NoktaE(1, 2)));
    }
}
```

```
true 2 false
true
```

İlk satır çelişkiyi gösterir: `a.equals(b)` `true`'dur, ama küme iki öğe tutar. Yeni bir `NoktaE(1, 2)` ile yapılan `contains` da `false` döner, çünkü yeni nesnenin hash kodu (dolayısıyla kovası) başkadır. İkinci satırda aynı nesne bir listede aranır ve bulunur. `ArrayList.contains` yalnızca `equals` kullanır; hash koduna bakmaz. Aynı sınıf listede doğru, kümede yanlış çalışır. Derleyici bu durumu fark eder ve `-Xlint` seçeneğiyle derlenince uyarı verir: `Class NoktaE overrides equals, but neither it nor any superclass overrides hashCode method`.

`Objects.hash(x, y)`'nin ürettiği sayı basit bir formüldür: `31 · (31 · 1 + x) + y`. (1, 2) noktası için 31 · 32 + 2 = 994 çıkar. Bu sayı bir sonraki örnekte kullanılacaktır.

5.4.5 Kümedeki nesneyi değiştirmek

`equals` ve `hashCode` doğru yazılmış olsa da bir tuzak daha vardır. Kümeye girmiş bir nesnenin, hash kodunu belirleyen alanı sonradan değiştirilirse ne olur? Aşağıdaki program bunu dener:

```
Set<Nokta> kume = new HashSet<>();
Nokta p = new Nokta(1, 2);
kume.add(p);
System.out.println(kume.contains(p));
p.x = 5;              // alan değişti
System.out.println(kume.contains(p));
System.out.println(kume.contains(new Nokta(1, 2)));
kume.add(p);
System.out.println(kume.size());
```

```
true
false
false
2
```

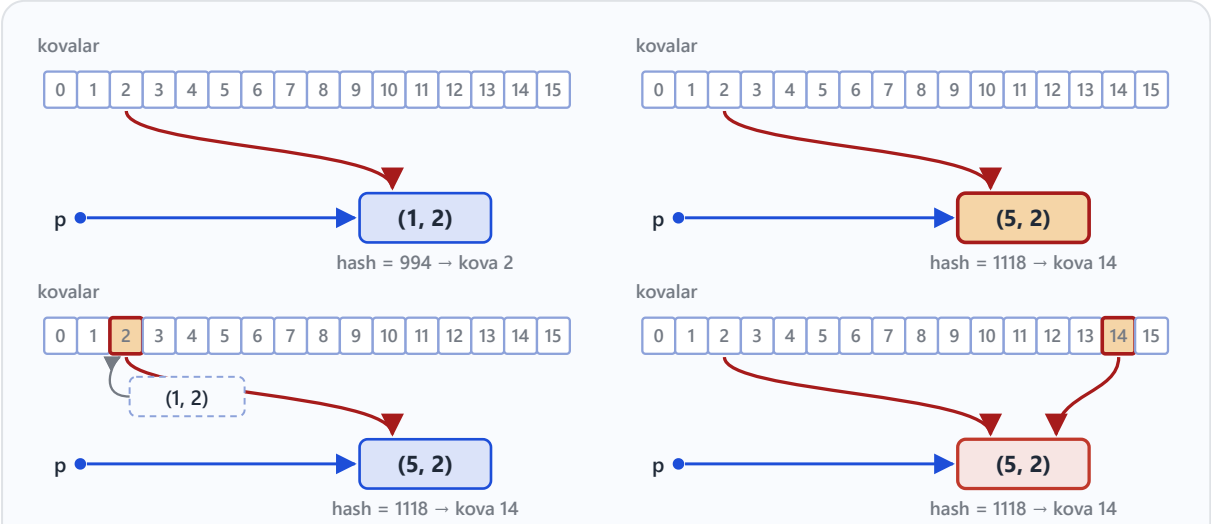
Tablo 5.17 programı adım adım izler. Değişkenler sütununda `hash`, `p`'nin o anki hash kodu, `kova` ise bu koddan hesaplanan kova numarasıdır.

Adım	Satır	Değişkenler	Yeni çıktı
1	1: `Set<Nokta> kume = new HashSet<>();`	—	—
2	2: `Nokta p = new Nokta(1, 2);`	p=(1, 2), hash=994, kova=2	—
3	3: `kume.add(p);`	p=(1, 2), hash=994, kova=2	—
4	4: `System.out.println(kume.contains(p));`	p=(1, 2), hash=994, kova=2	`true`
5	5: `p.x = 5; // alan değişti`	p=(5, 2), hash=1118, kova=14	—
6	6: `System.out.println(kume.contains(p));`	p=(5, 2), hash=1118, kova=14	`false`
7	7: `System.out.println(kume.contains(new Nokta(1, 2)));`	p=(5, 2), hash=1118, kova=14, yeni=(1, 2)	`false`
8	8: `kume.add(p);`	p=(5, 2), hash=1118, kova=14	—
9	9: `System.out.println(kume.size());`	p=(5, 2), hash=1118, kova=14	`2`

Tablo 5.17: Kümedeki nesnenin alanı değişince olanlar. Kova numarası, hash kodunun son 4 bitidir (16 kovalı tablo).

Başlangıçta `Objects.hash(1, 2)` 994'tür. 994'ün ikili gösterimi `1111100010`'dur ve son 4 biti `0010`, yani 2'dir. `add(p)` nesneyi 2 numaralı kovaya koyar. `contains(p)` yine 2 numaralı kovaya bakar, `equals` doğru döner ve sonuç `true` olur.

Sonra `p.x = 5` ataması yapılır. Nesnenin hash kodu artık `31 · 36 + 2 = 1118`'dir ve kovası 14'tür. Ama nesne hâlâ 2 numaralı kovada durur; küme, alanın değiştiğinden haberdar değildir. Şekil 5.16 bu durumu ve sonucunu gösterir.



Şekil 5.16: Kayıp öğe. Sol üstte `p` (1, 2) kova 2'ye yerleşir. Sağ üstte `p.x = 5` sonrası hash 1118, kova 14 olur; nesne kova 2'de kalır. Sol altta `new Nokta(1, 2)` kova 2'ye bakar, ama oradaki nesne artık (5, 2)'dir ve `equals` yanlış döner. Sağ altta `add(p)` kova 14'ü boş bulur ve aynı nesneyi ikinci kez ekler.

İkinci `contains(p)` kova 14'e bakar. Orası boştur ve sonuç `false`'tur. Nesne kümededir, ama bulunamaz. Üçüncü sorguda yeni bir `Nokta(1, 2)` aranır. Bu nesnenin kovası 2'dir ve orada gerçekten bir nesne vardır. Ama o nesne artık (5, 2)'dir; `equals` yanlış döner ve sonuç yine `false`'tur. Son olarak `add(p)` kova 14'ü boş bulur ve aynı nesneyi ikinci kez ekler. `size()` 2 olur. Küme aynı nesneyi iki kez içerir; "tekrar yok" kuralı bozulmuştur.

Bu hatanın ders çıkarılacak yanı, hiçbir istisna fırlatılmamasıdır. Program çalışmaya devam eder ve yanlış sonuç üretir. Kural basittir: hash'li bir koleksiyondaki (`HashSet`, `HashMap` anahtarı) nesnenin `equals` ve `hashCode`'a giren alanları değiştirilmez. Anahtar olarak `String`, `Integer` gibi değiştirilemez nesneler ya da alanları `final` olan sınıflar seçilir. Bir alanın değişmesi gerekiyorsa nesne önce kümeden çıkarılır, değiştirilir ve sonra yeniden eklenir.

5.4.6 SortedSet ve TreeSet

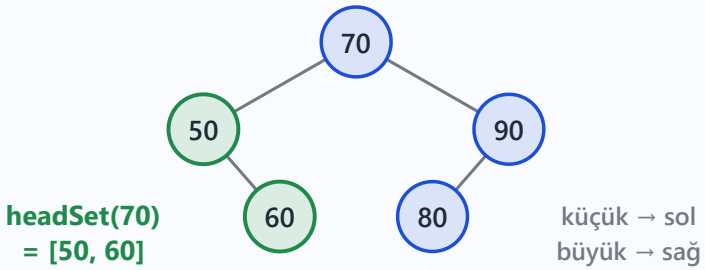
`SortedSet`, öğeleri sıralı saklayan küme arayüzüdür. Sıra doğal sıraya (`Comparable`) ya da verilen bir `Comparator`'a göredir. Arayüz, sıralı olmanın getirdiği yeni sorgular sunar: en küçük ve en büyük öğe, belirli bir değerden küçükler, belirli bir aralıktakiler. `TreeSet` `AbstractSet`'ten türetilir ve `NavigableSet` (dolayısıyla `SortedSet`) arayüzünü uygular. İçte bir **kırmızı-siyah ağaç** (*red-black tree*) kullanır.

Kırmızı-siyah ağaç kendini dengeleyen bir ikili arama ağacıdır. Her düğümün solundaki öğeler ondan küçük, sağındakiler ondan büyüktür. Ekleme ve silme sırasında ağaç, yüksekliği log n düzeyinde kalacak biçimde düğümleri döndürür. Bu yüzden ekleme, silme ve arama O(log n)'dir. `HashSet`'in

O(1)'inden yavaştır, ama karşılığında sıra verir. Tablo 5.18 `TreeSet`'in sıralı sorgularını, Şekil 5.17 ağacı verir.

Metot	Görevi
`first()` / `last()`	en küçük / en büyük öğe
`headSet(to)`	`to`'dan küçükler (`to` hariç)
`tailSet(from)`	`from` ve büyükler (`from` dahil)
`subSet(from, to)`	`from` dahil, `to` hariç
`comparator()`	sıralayan `Comparator`; doğal sırada `null`
`add`, `remove`	ekleme, çıkarma

Tablo 5.18: `TreeSet`'in sıralı küme metotları.



Şekil 5.17: `TreeSet`'in içindeki ağaç (50, 60, 70, 80, 90). Kök 70'tir; küçükler solda, büyükler sağdadır. Yeşil düğümler `headSet(70)` = `[50, 60]` aralığını gösterir.

```
TreeSet<Integer> notlar = new TreeSet<>();
notlar.add(70);
notlar.add(50);
notlar.add(90);
notlar.add(60);
notlar.add(80);
System.out.println(notlar);
System.out.println(notlar.first() + " " + notlar.last());
System.out.println(notlar.headSet(70));     // 70 hariç
System.out.println(notlar.tailSet(70));     // 70 dahil
System.out.println(notlar.subSet(60, 90));  // [60, 90)
System.out.println(notlar.comparator());    // doğal sıra
```

```
TreeSet<String> meyveler = new TreeSet<>();
meyveler.add("Elma");
meyveler.add("Armut");
System.out.println(meyveler);
```

```
[50, 60, 70, 80, 90]
50 90
[50, 60]
[70, 80, 90]
[60, 70, 80]
null
[Armut, Elma]
```

Notlar 70, 50, 90, 60, 80 sırasıyla eklenir, ama küme her zaman sıralı yazılır: `[50, 60, 70, 80, 90]`. `first()` 50, `last()` 90'dır. Aralık metotlarında sınırların dahil olup olmadığı önemlidir. Kural "başlangıç dahil, bitiş hariç"tir; bu, Java'nın `substring` gibi öteki aralık metotlarıyla da uyumludur. `headSet(70)` 70'i içermez: `[50, 60]`. `tailSet(70)` 70'i içerir: `[70, 80, 90]`. `subSet(60, 90)` matematikteki [60, 90) aralığıdır: `[60, 70, 80]`. Doğal sıra kullanıldığı için `comparator()` `null` döndürür. Dizgiler de karakter kodlarına göre sıralanır: "Armut", "Elma"dan önce gelir.

5.4.7 Küme işlemleri

Matematikteki birleşim, kesişim ve fark işlemlerinin her birinin `Collection`'da bir karşılığı vardır. Tablo 5.19 eşleşmeyi verir.

İşlem	Metot
birleşim ∪	`addAll`
kesişim ∩	`retainAll`
fark −	`removeAll`
alt küme ⊆	`containsAll`

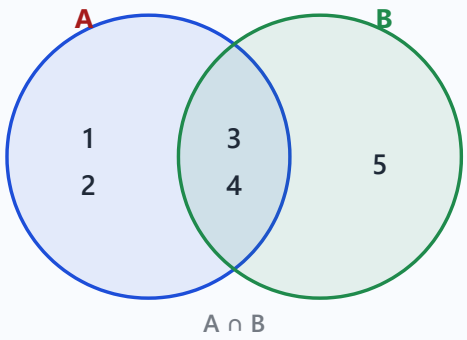
Tablo 5.19: Küme işlemleri ve Java metotları.

```
Set<Integer> a = new TreeSet<>(Arrays.asList(1, 2, 3, 4));
Set<Integer> b = new TreeSet<>(Arrays.asList(3, 4, 5));

Set<Integer> birlesim = new TreeSet<>(a);
birlesim.addAll(b);
Set<Integer> kesisim = new TreeSet<>(a);
kesisim.retainAll(b);
Set<Integer> fark = new TreeSet<>(a);
fark.removeAll(b);

System.out.println("A ∪ B = " + birlesim);
System.out.println("A ∩ B = " + kesisim);
System.out.println("A − B = " + fark);
System.out.println(a.containsAll(kesisim));
```

```
A ∪ B = [1, 2, 3, 4, 5]
A ∩ B = [3, 4]
A − B = [1, 2]
true
```



Şekil 5.18: Venn şeması. A = {1, 2, 3, 4}, B = {3, 4, 5}; kesişim {3, 4}'tür.

Bu metotlar çağrıldıkları kümeyi değiştirir. `a.addAll(b)` yazılsaydı `a`'nın kendisi birleşime dönüşür ve sonraki işlemlerde asıl `a` kaybolurdu. Bu yüzden her işlemden önce `new TreeSet<>(a)` ile bir kopya alınır. Birleşim kopyaya `b`'nin öğelerini ekler; tekrar eden 3 ve 4 ikinci kez eklenmez. Kesişim, kopyada yalnızca `b`'de de bulunan öğeleri bırakır. Fark, kopyadan `b`'nin öğelerini çıkarır. Son satır `kesisim`'in `a`'nın alt kümesi olduğunu doğrular. `TreeSet` kullanıldığı için sonuçlar sıralı yazılır. Bu işlemler, parametre olarak verilen koleksiyonun her öğesi için bir `contains` ya da `add` yapar; maliyetleri öğe sayısı ile bu işlemin maliyetinin çarpımıdır.

5.5 Eşlemeler

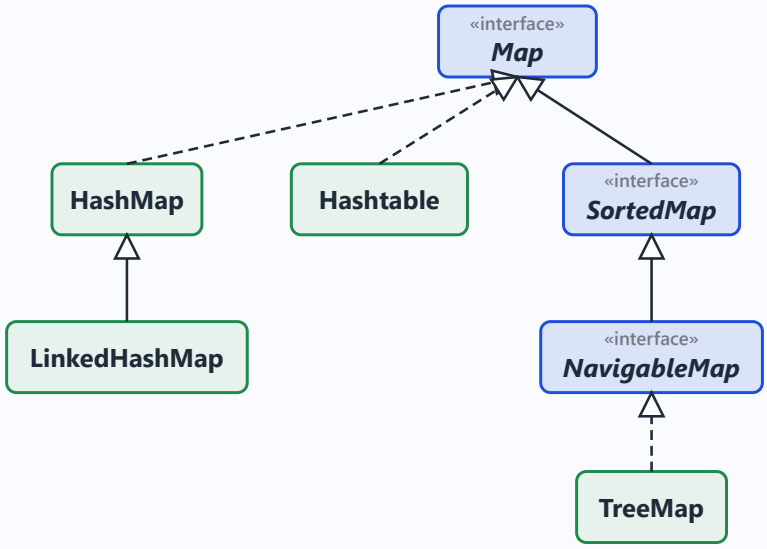
Bir öğrenci numarasından öğrenciye, bir ürün adından fiyata, bir kelimeden geçiş sayısına ulaşmak programlamanın en sık işlerindendir. Bu işlerde veri çiftler hâlinde gelir: bir anahtar ve ona bağlı bir değer. Eşleme bu çiftleri saklar ve anahtardan değere hızla gider. Bu kesimde eşleme arayüzü, üç temel gerçeklemesi ve eski `Hashtable` sınıfı ele alınır.

5.5.1 Map arayüzü ve metotları

Tanım 5.13

Eşleme (*map*), anahtar-değer (*key-value*) çiftlerini saklayan yapıdır. Her anahtar bir değere eşlenir. Anahtarlar tekildir; değerler tekrar edebilir.

Eşlemenin en bilinen benzetmesi sözlüktür. Kelime (anahtar) ile anlamı (değer) bulunur ve aynı kelime sözlükte iki kez yer almaz. İki farklı kelimenin anlamı ise aynı olabilir. Eşleme anahtar üzerinden hızlı erişim sağlar ve çiftler üzerinde işlem yapmaya olanak tanır. Şekil 5.19 eşleme sınıflarının hiyerarşisini gösterir.



Şekil 5.19: `Map` hiyerarşisi. `HashMap` ve `Hashtable` `Map`'i uygular; `LinkedHashMap` `HashMap`'ten türer. `TreeMap`, `SortedMap`'i genişleten `NavigableMap`'i uygular.

Üç temel gerçekleme, kümelerdeki üçlünün eşlemedeki karşılığıdır. `HashMap` sıra garantisi vermez. `LinkedHashMap` eklenme sırasını korur. `TreeMap` anahtarları sıralı tutar; sıra doğal sıra ya da verilen `Comparator`'dır. Tablo 5.20 arayüzün metotlarını verir. `K` anahtar tipini, `V` değer tipini gösterir. Örneğin `Map<String, Integer>` isimden nota bir eşlemedir.

Metot	Görevi
`put(K key, V value)`	anahtarı değerle eşler; anahtar varsa değeri değiştirir ve eskisini döndürür
`get(Object key)`	anahtarın değerini döndürür; anahtar yoksa `null`
`remove(Object key)`	anahtarı ve değerini çıkarır
`containsKey(Object key)`	anahtar eşlemede var mı?
`containsValue(Object v)`	değer eşlemede var mı?
`keySet()`	bütün anahtarları bir `Set` olarak döndürür
`values()`	bütün değerleri bir `Collection` olarak döndürür
`entrySet()`	bütün anahtar-değer çiftlerini bir `Set<Map.Entry<K, V>>` olarak döndürür
`size()`, `isEmpty()`	çift sayısı, boş mu?
`getOrDefault(key, v)`	anahtar yoksa `v`'yi döndürür (Java 8+)

Tablo 5.20: `Map<K, V>` arayüzünün temel metotları.

5.5.2 HashMap

`HashMap`, anahtar-değer çiftlerini bir hash tablosunda saklar. `AbstractMap`'ten türetilmiştir ve `Map`'i uygular. Çiftin kovası anahtarın hash kodundan hesaplanır; değerin hash kodu hiç kullanılmaz. Bu yüzden `get` ve `put` ortalamada O(1)'dir. Öğeler sırasız depolanır.

```
HashMap<String, Double> urunFiyatlari = new HashMap<>();
urunFiyatlari.put("Telefon", 2000.0);
urunFiyatlari.put("Kulaklık", 350.0);
urunFiyatlari.put("Kılıf", 120.0);
Double eski = urunFiyatlari.put("Telefon", 1800.0);
System.out.println(eski);
System.out.println(urunFiyatlari.get("Telefon"));
System.out.println(urunFiyatlari.get("Tablet"));
System.out.println(urunFiyatlari.containsKey("Kılıf"));
urunFiyatlari.remove("Kılıf");
System.out.println(urunFiyatlari.size());
```

```
urunFiyatlari.put(null, 0.0);         // tek null anahtar
urunFiyatlari.put("Kalem", null);     // null değer
System.out.println(urunFiyatlari.size());
```

```
2000.0
1800.0
null
true
2
4
null true 0.0
```

Program şöyle ilerler. Üç ürün eklenir. Dördüncü `put("Telefon", 1800.0)` yeni bir çift eklemez; var olan anahtarın değerini değiştirir. Anahtarlar tekil olduğu için böyle olur. `put` eski değeri döndürür: 2000.0. Bu dönüş değeri bir anahtarın daha önce var olup olmadığını öğrenmenin de yoludur; yeni anahtarda `put` `null` döndürür. `get("Telefon")` yeni değeri verir. `get("Tablet")` bulunmayan bir anahtar için `null` döndürür, istisna fırlatmaz. `containsKey("Kılıf")` `true` döner. Kılıf silindikten sonra eşlemede iki çift kalır.

`HashMap` bir `null` anahtara ve birden çok `null` değere izin verir. Ek satırlar bir `null` anahtar ve bir `null` değer ekler; boyut 4 olur. Notun programına eklenmiş son satır bu iznin doğurduğu belirsizliği gösterir. `get("Kalem")` `null` döndürür, ama bu "anahtar yok" anlamına gelmez; `containsKey("Kalem")` `true`'dur. `null` değer saklayan bir eşlemede "yok" ile "değeri null" ancak `containsKey` ile ayırt edilir. `getOrDefault("Tablet", 0.0)` ise anahtar yoksa verilen varsayılanı döndürür: 0.0.

Sık yapılan hata

`get`**'in sonucunu doğrudan ilkel tipe atamak.** `int n = sayac.get(k);` anahtar yoksa `null` 'ı kutudan çıkarmaya çalışır ve `NullPointerException` fırlatır (iletisi jenerik tipler kesiminde verilmiştir). Anahtarın bulunmayabileceği yerde `getOrDefault(k, 0)` kullanılır ya da önce `containsKey` sorulur.

5.5.3 Eşleme üzerinde gezinme

`Map` `Iterable` değildir; doğrudan for-each döngüsüne verilemez. Gezinme, eşlemenin sunduğu üç görünümden biri üzerinden yapılır:

```
Map<String, Integer> notlar = new TreeMap<>();
notlar.put("Cem", 72);
notlar.put("Ayşe", 95);
notlar.put("Banu", 60);

for (String isim : notlar.keySet()) {
    System.out.print(isim + " ");
}
System.out.println();
for (Integer not : notlar.values()) {
    System.out.print(not + " ");
}
System.out.println();
for (Map.Entry<String, Integer> g : notlar.entrySet()) {
    System.out.println(g.getKey() + " → " + g.getValue());
}
```

```
Ayşe Banu Cem
95 60 72
Ayşe → 95
Banu → 60
Cem → 72
```

Burada nesne bir `TreeMap` 'tir; anahtarlar alfabetik sırayla gelir. `keySet()` yalnızca anahtarları verir: Ayşe Banu Cem. `values()` yalnızca değerleri, anahtarların sırasıyla verir: 95 60 72. `entrySet()` her çifti bir `Map.Entry` nesnesi olarak verir. `getKey()` anahtarı, `getValue()` değeri okur.

Hem anahtar hem değer gerekiyorsa en verimli yol `entrySet()` 'tir. `keySet()` üzerinde gezip her anahtar için `get` çağırmak da aynı sonucu verir, ama her adımda anahtarı yeniden arar. `HashMap` 'te bu ek arama ortalama O(1)'dir; `TreeMap` 'te ise her `get` O(log n)'dir ve toplam iş O(n log n)'e çıkar. `entrySet` ise çiftleri zaten elinde tutar ve gezinme O(n)'dir.

Eşlemenin kendisini for-each'e vermek derleme hatasıdır:

```
MapForEach.java:5: error: for-each not applicable to expression type
        for (String s : m) {
                        ^
  required: array or java.lang.Iterable
  found:    Map<String,Integer>
1 error
```

İleti, for-each'in yalnızca dizilerde ve `java.lang.Iterable` tiplerinde kullanılabildiğini, ama bir `Map<String,Integer>` bulunduğunu söyler.

5.5.4 Kelime sayma

Eşlemenin klasik kullanımı bir metindeki kelimeleri saymaktır. Her kelime bir anahtardır, değer ise o kelimenin kaç kez geçtiğidir:

```
String metin = "elma armut elma kiraz armut elma";
Map<String, Integer> sayac = new TreeMap<>();
for (String k : metin.split(" ")) {
    if (sayac.containsKey(k)) {
        sayac.put(k, sayac.get(k) + 1);
    } else {
        sayac.put(k, 1);
    }
}
System.out.println(sayac);
```

```
{armut=2, elma=3, kiraz=1}
```

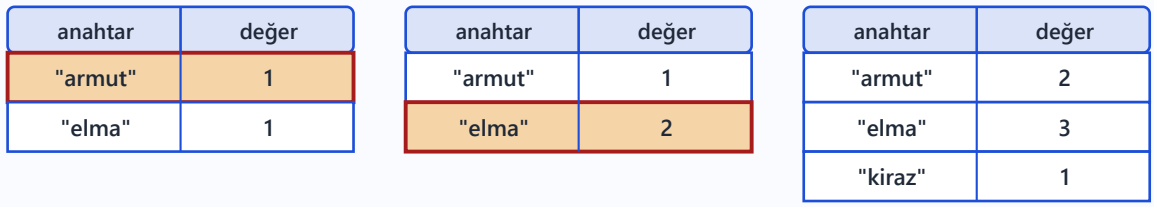
`metin.split(" ")` metni boşluklardan böler ve altı kelimelik bir dizi verir. Döngü her kelime için iki durumdan birini işler. Kelime daha önce görülmüşse eski sayı alınır, bir artırılır ve geri yazılır. İlk kez görülüyorsa 1 değeriyle eklenir. Tablo 5.21 her kelimeden sonraki durumu verir.

Adım	Satır	sayac	Değişkenler	Yeni çıktı
1	1: `String metin = "elma armut elma kiraz armut elma";` 2: `Map<String, Integer> sayac = new TreeMap<>();`	boş	—	—
2	3: `for (String k : metin.split(" ")) {` 4: `if (sayac.containsKey(k)) {` 7: `sayac.put(k, 1);`	elma=1	k="elma"	—
3	3: `for (String k : metin.split(" ")) {` 4: `if (sayac.containsKey(k)) {` 7: `sayac.put(k, 1);`	armut=1, elma=1	k="armut"	—

Adım	Satır	sayac	Değişkenler	Yeni çıktı
4	3: `for (String k : metin.split(" ")) {` 4: `if (sayac.containsKey(k)) {` 5: `sayac.put(k, sayac.get(k) + 1);`	armut=1, elma=2	k="elma"	—
5	3: `for (String k : metin.split(" ")) {` 4: `if (sayac.containsKey(k)) {` 7: `sayac.put(k, 1);`	armut=1, elma=2, kiraz=1	k="kiraz"	—
6	3: `for (String k : metin.split(" ")) {` 4: `if (sayac.containsKey(k)) {` 5: `sayac.put(k, sayac.get(k) + 1);`	armut=2, elma=2, kiraz=1	k="armut"	—
7	3: `for (String k : metin.split(" ")) {` 4: `if (sayac.containsKey(k)) {` 5: `sayac.put(k, sayac.get(k) + 1);`	armut=2, elma=3, kiraz=1	k="elma"	—
8	10: `System.out.println(sayac);`	armut=2, elma=3, kiraz=1	k="elma"	`{armut=2, elma=3, kiraz=1}`

Tablo 5.21: Kelime sayma programının izlenmesi. `sayac` sütunu adımın sonundaki eşlemedir.

“elma” ve “armut” ilk kez görüldüğünde 1 ile eklenir. Üçüncü kelime yine “elma”dır. `containsKey` doğru döner; eski değer 1 alınır ve 2 yazılır. “kiraz” 1 ile eklenir. “armut” 2'ye, son “elma” 3'e çıkar. Döngü bittiğinde `TreeMap` anahtarları alfabetik sırayla yazar. Şekil 5.20 eşlemenin üç anını gösterir.



Şekil 5.20: Kelime sayma eşlemesi. Solda iki kelimeden sonra: armut=1, elma=1. Ortada ikinci “elma”dan sonra elma=2. Sağda döngünün sonunda: armut=2, elma=3, kiraz=1.

Döngü gövdesi her kelime için en çok üç eşleme işlemi yapar: `containsKey`, `get` ve `put`. `getOrDefault` ile aynı iş tek satıra iner: `sayac.put(k, sayac.getOrDefault(k, 0) + 1)`. Java 8 ile gelen `merge` metodu daha da kısadır: `sayac.merge(k, 1, Integer::sum)`. Bu çağrı anahtar yoksa 1 yazar, varsa eski değerle 1'i toplar. Üç yazım aynı sonucu verir:

```
{armut=2, elma=3, kiraz=1} {armut=2, elma=3, kiraz=1}
{elma=3, kiraz=1, armut=2}
```

Çıktının ikinci satırı aynı sayaçların bir `HashMap`'e kopyalanmış hâlidir. Sıra artık alfabetik değildir: elma, kiraz, armut. Sayılar aynıdır, yalnızca gezinme sırası değişmiştir. Kelime sayımında alfabetik çıktı isteniyorsa `TreeMap`, en hızlı sayım isteniyorsa `HashMap` seçilir. `n` kelimelik bir metinde `HashMap` ile sayım ortalama O(n), `TreeMap` ile O(n log n) sürer.

5.5.5 LinkedHashMap ve Hashtable

LinkedHashMap

`LinkedHashMap`, `HashMap`'ten türeyen ve çiftleri ek olarak bağlı listeyle birbirine bağlayan eşlemedir. Bu bağlı liste sayesinde eklenme sırası korunur. Hız ve sıra birlikte önemliyse tercih edilir; sıra önemli değilse `HashMap` yeterlidir. Temel metotları `Map` ile aynıdır: `put`, `remove`, `get`, `containsKey`, `size`. Aynı rehberi üç eşlemeye koyunca sıra farkı görülür:

```
Map<String, String> rehber = new LinkedHashMap<>();
rehber.put("Zeynep", "555-1111");
rehber.put("Ahmet", "555-1234");
rehber.put("Mert", "555-2222");
Map<String, String> h = new HashMap<>(rehber);
Map<String, String> t = new TreeMap<>(rehber);
System.out.println("LinkedHashMap: " + rehber);
System.out.println("HashMap:       " + h);
System.out.println("TreeMap:       " + t);
```

```
LinkedHashMap: {Zeynep=555-1111, Ahmet=555-1234, Mert=555-2222}
HashMap:       {Ahmet=555-1234, Mert=555-2222, Zeynep=555-1111}
TreeMap:       {Ahmet=555-1234, Mert=555-2222, Zeynep=555-1111}
```

`new HashMap<>(rehber)` ve `new TreeMap<>(rehber)` yazımları, verilen eşlemenin bütün çiftlerini kopyalayan yapıcılardır. `LinkedHashMap` eklenme sırasını korur: Zeynep, Ahmet, Mert. `TreeMap` anahtarları alfabetik sıraya dizer. `HashMap`'in sırası ise kova sırasıdır ve buradaki alfabetik görünüm tesadüftür. Kopyalayan yapıcı, kova sayısını kaynak eşlemenin boyuna göre seçer: üç çift için 4 kova. Dört kovada kova numarası hash değerinin son 2 bitidir. Kümeler kesimindeki ölçüm programının hash değerlerinden Ahmet 1, Mert 2, Zeynep 3 numaralı kovaya düşer. Aynı üç isim 16 kovalı bir tabloya konsaydı kovaları 1, 14 ve 7 olurdu ve sıra Ahmet, Zeynep, Mert'e dönerdi. Tablonun 4 kovalı olduğu, iç diziyi yansımayla okuyan bir ölçüm programıyla doğrulanmıştır. Sonuç açıktır: `HashMap`'in sırası kova sayısına bağlıdır ve programlar bu sıraya hiçbir zaman güvenmez.

Hashtable

`Hashtable`, anahtar-değer çiftlerini saklayan, senkronize ve eski bir eşleme sınıfıdır (Java 1.0). `HashMap`'ten farkı, her metodunun kilitli olmasıdır. Kilitleme yüzünden tek iş parçacıklı programlarda `HashMap`'ten yavaştır. Çok iş parçacıklı programlarda ise `java.util.concurrent` paketindeki `ConcurrentHashMap` daha iyi bir seçimdir; tablonun tamamını kilitlemek yerine daha ince bir kilitleme yapar. Bugün `Hashtable` çoğunlukla eski kodlarda görülür.

`Hashtable`'ın ikinci farkı `null` kabul etmemesidir: anahtarlar da değerler de `null` olamaz.

```
Hashtable<String, String> telefonRehberi = new Hashtable<>();
telefonRehberi.put("Ahmet", "555-1234");
System.out.println(telefonRehberi.get("Ahmet"));
System.out.println(telefonRehberi.containsKey("Ayşe"));
// telefonRehberi.put(null, "x");  → NullPointerException
```

```
555-1234
false
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"Object.hashCode()" because "key" is null
```

Notun programında sunumdaki yorum satırı açılmıştır. İlk iki çıktı normaldir; "Ayşe" anahtarı yoktur. `put(null, "x")` ise `NullPointerException` fırlatır. İleti hatanın nedenini söyler: tablo kovayı bulmak için anahtarın `hashCode()` metodunu çağırmak ister, ama anahtar `null`'dır. `null` değer de reddedilir; `t.put("Ayşe", null)` da aynı istisnayı, bu kez iletisiz fırlatır.

5.5.6 SortedMap ve TreeMap

`SortedMap`, anahtarları sıralı saklayan eşleme arayüzüdür; her anahtar bir kez bulunur. Alt eşleme ve üst eşleme gibi aralık işlemleri sunar. `TreeMap` `AbstractMap`'ten türetilir ve `NavigableMap` arayüzünü uygular. Anahtarlar doğal sıraya ya da verilen `Comparator`'a göre dizilir. İçte `TreeSet`'teki gibi bir kırmızı-siyah ağaç vardır; `get`, `put`, `remove` O(log n)'dir. Tablo 5.22 sıralı eşleme metotlarını verir. Adları ve sınır kuralları `TreeSet`'in metotlarıyla aynıdır.

Metot	Görevi
`firstKey()` / `lastKey()`	en küçük / en büyük anahtar
`headMap(to)`	`to`'dan küçük anahtarlar (hariç)
`tailMap(from)`	`from` ve büyükleri (dahil)
`subMap(from, to)`	`from` dahil, `to` hariç
`comparator()`	sıralayan `Comparator` (doğal sırada `null`)
`put`, `get`, `remove`	`Map` ile aynı

Tablo 5.22: `TreeMap`'in sıralı eşleme metotları.

```
TreeMap<String, Double> urunFiyatlari = new TreeMap<>();
urunFiyatlari.put("Telefon", 2000.0);
urunFiyatlari.put("Kalem", 15.0);
urunFiyatlari.put("Defter", 40.0);
urunFiyatlari.put("Silgi", 5.0);
System.out.println(urunFiyatlari);
System.out.println(urunFiyatlari.firstKey());
System.out.println(urunFiyatlari.lastKey());
System.out.println(urunFiyatlari.headMap("Kalem"));
System.out.println(urunFiyatlari.tailMap("Kalem"));
System.out.println(urunFiyatlari.subMap("Defter", "Silgi"));
```

```
{Defter=40.0, Kalem=15.0, Silgi=5.0, Telefon=2000.0}
Defter
Telefon
{Defter=40.0}
{Kalem=15.0, Silgi=5.0, Telefon=2000.0}
{Defter=40.0, Kalem=15.0}
```

Ürünler hangi sırayla eklenirse eklensin, eşleme anahtarların alfabetik sırasıyla yazılır: Defter, Kalem, Silgi, Telefon. En küçük anahtar Defter, en büyüğü Telefon'dur. `headMap("Kalem")` Kalem'den küçük anahtarları verir ve Kalem'i içermez: yalnız Defter. `tailMap("Kalem")` Kalem'i ve ondan büyükleri içerir. `subMap("Defter", "Silgi")` Defter'i içerir, Silgi'yi içermez. Aralık sorguları sıralanmış anahtarlar sayesinde O(log n) zamanda başlar; `HashMap`'te aynı sorgu için bütün anahtarları taramak gerekirdi.

5.5.7 Eşleme sınıflarını karşılaştırmak

	`HashMap`	`LinkedHashMap`	`TreeMap`	`Hashtable`
Gezinme sırası	garanti yok	eklenme sırası	anahtar sıralı	garanti yok
`get` / `put`	O(1) ortalama	O(1) ortalama	O(log n)	O(1) ortalama, kilitli
`null` anahtar	bir tane	bir tane	hayır	hayır
`null` değer	evet	evet	evet	hayır
Senkronize	hayır	hayır	hayır	evet (eski)
Ne zaman?	varsayılan seçim	sıra korunmalı	sıralı anahtar, aralık sorgusu	eski kod

Tablo 5.23: Dört eşleme sınıfının karşılaştırması.

Tablodaki mantık kümelerde de geçerlidir. `HashSet`, `LinkedHashSet` ve `TreeSet` içte sırasıyla `HashMap`, `LinkedHashMap` ve `TreeMap` kullanır; kümenin öğeleri eşlemenin anahtarlarıdır. Bu yüzden bir kümenin sırası, hızı ve `null` kuralı karşılık geldiği eşlemeninkiyle aynıdır.

5.6 Yineleyici

Liste indisle, küme hash tablosuyla, `TreeSet` ağaçla, `ArrayDeque` dairesel diziyle çalışır. Bu kadar farklı yapının öğelerini tek bir döngü biçimiyle gezmek ancak ortak bir arayla mümkündür. Bu ara yineleyicidir. Her koleksiyon kendi iç yapısını bilen bir yineleyici üretir; kullanıcı ise yalnızca "sıradaki öğe var mı?" ve "sıradakini ver" diye sorar.

5.6.1 Iterator arayüzü

Tanım 5.14

Yineleyici (*iterator*), bir koleksiyonun öğelerini sırayla veren nesnedir. Koleksiyonun iç yapısını bilmeden her öğeye erişmeyi sağlar ve gezinirken güvenle silme yapılabilen tek yoldur. `Iterable` olan her koleksiyonun `iterator()` metodu bir yineleyici döndürür.

Metot	Görevi
`hasNext()`	bir sonraki öğe var mı?
`next()`	sonraki öğeyi döndürür, imleci ilerletir
`remove()`	son `next()` ile alınan öğeyi koleksiyondan çıkarır

Tablo 5.24: `Iterator<E>` arayüzünün metotları.

Yineleyiciyi anlamanın en kolay yolu bir **imleç** düşünmektir. İmleç öğelerin üzerinde değil, aralarında durur. Başlangıçta ilk öğenin önündedir. `next()` imleci bir öğenin üzerinden atlatır ve atladığı öğeyi döndürür. `hasNext()` imlecin önünde öğe kalıp kalmadığını söyler. Şekil 5.21 bu düzeni gösterir.



Şekil 5.21: Yineleyicinin imleci öğelerin arasında durur. `next()` 3'ü döndürüp imleci 3 ile 8'in arasına taşımıştır; `hasNext()` `true`'dur.

```
ArrayList<Integer> liste = new ArrayList<>();
liste.add(3);
liste.add(8);
liste.add(5);
Iterator<Integer> iterator = liste.iterator();
while (iterator.hasNext()) {
    Integer sayi = iterator.next();
    // öğe burada işlenir
    System.out.println(sayi);
}
```

```
3
8
5
```

Yineleyici genellikle `while (it.hasNext())` döngüsüyle kullanılır. Tablo 5.25 döngüyü adım adım izler.

Adım	Satır	Liste ve imleç	Değişkenler	Yeni çıktı
1	1: `ArrayList<Integer> liste = new ArrayList<>();` 2: `liste.add(3);` 3: `liste.add(8);` 4: `liste.add(5);`	liste 3, 8, 5	—	—
2	5: `Iterator<Integer> iterator = liste.iterator();`	liste 3, 8, 5; imleç 0. öğeden önce	—	—
3	6: `while (iterator.hasNext()) {` 7: `Integer sayi = iterator.next();`	liste 3, 8, 5; imleç 1. öğeden önce	sayi=3	—
4	9: `System.out.println(sayi);`	liste 3, 8, 5; imleç 1. öğeden önce	sayi=3	`3`
5	6: `while (iterator.hasNext()) {` 7: `Integer sayi = iterator.next();`	liste 3, 8, 5; imleç 2. öğeden önce	sayi=8	—
6	9: `System.out.println(sayi);`	liste 3, 8, 5; imleç 2. öğeden önce	sayi=8	`8`
7	6: `while (iterator.hasNext()) {` 7: `Integer sayi = iterator.next();`	liste 3, 8, 5; imleç sonda	sayi=5	—
8	9: `System.out.println(sayi);`	liste 3, 8, 5; imleç sonda	sayi=5	`5`
9	6: `while (iterator.hasNext()) {`	liste 3, 8, 5; imleç sonda	sayi=5	—

Tablo 5.25: Yineleyiciyle gezinmenin izlenmesi. İmleç konumu, imlecin önündeki öğenin indisiyle verilir; indis 3, listenin sonu demektir.

`liste.iterator()` imleci ilk öğenin önüne koyar. Döngünün her turunda `hasNext()` `true` döndürür, `next()` sıradaki öğeyi verir ve imleci ilerletir. Üç turdan sonra imleç sondadır. `hasNext()` `false` döner ve döngü biter. Öğe kalmamışken `next()` çağrılırsa `NoSuchElementException` fırlatılır.

For-each döngüsü arka planda tam olarak bu yineleyici döngüsüne çevrilir. `for (Integer sayi : liste) { … }` yazıldığında derleyici bir yineleyici oluşturur ve `hasNext` / `next` çağrılarını kendisi ekler. Java Dil Belirtimi bu dönüşümü kesin olarak tanımlar. For-each'in yalnızca `Iterable` tiplerde çalışmasının nedeni de budur.

5.6.2 Gezinirken silme

Bir listeden belirli bir koşulu sağlayan öğeleri silmek sık bir iştir. En doğal görünen yol, for-each içinde listenin kendi `remove` metodunu çağırmaktır. Bu yol yanlıştır:

```
List<Integer> sayilar = new ArrayList<>(Arrays.asList(4, 7, 10, 13, 16));
for (Integer s : sayilar) {
    if (s % 2 == 0) {
        sayilar.remove(s);
    }
}
```

```
Exception in thread "main" java.util.ConcurrentModificationException
```

İlk çift sayı 4 silinir, ama hemen ardından program `ConcurrentModificationException` ile durur. Doğru yol, yineleyiciyi açıkça yazmak ve onun kendi `remove()` metodunu çağırmaktır:

```
List<Integer> sayilar = new ArrayList<>();
Collections.addAll(sayilar, 4, 7, 10, 13, 16);
Iterator<Integer> it = sayilar.iterator();
while (it.hasNext()) {
    if (it.next() % 2 == 0) {
        it.remove();         // güvenli silme
    }
}
System.out.println(sayilar);
```

`it.next()` öğeyi alır ve çift olup olmadığı denetlenir. Çiftse `it.remove()` son alınan öğeyi listeden çıkarır. Yineleyici silmeyi kendisi yaptığı için iç durumunu da günceller ve gezinme bozulmadan sürer. Java 8 ile aynı işin kısa bir yolu gelmiştir: `removeIf`. Koşul bir lambda ifadesiyle verilir ve metot döngüyü kendisi kurar:

```
List<Integer> s2 = new ArrayList<>();
Collections.addAll(s2, 4, 7, 10, 13, 16);
s2.removeIf(s -> s % 2 == 0);
System.out.println(s2);
```

```
[7, 13]
[7, 13]
```

İki yolun çıktısı da aynıdır: yalnız tek sayılar kalır. `ArrayList.removeIf` önce silinecek öğeleri işaretler, sonra kalanları tek geçişte sola kaydırır. Bu yüzden toplam iş O(n)'dir. Yineleyiciyle tek tek silmek ise her silmede arkadaki öğeleri kaydırır ve en kötü durumda $O(n^2)$'ye çıkar. Büyük listelerde `removeIf` hem daha kısa hem daha hızlıdır.

`it.remove()` her `next()`'ten sonra en çok bir kez çağrılabilir; aksi hâlde `IllegalStateException` fırlatılır.

Sık yapılan hata

İndisli döngüde ileri giderken silmek. `for (int i = 0; i < a.size(); i++) if (a.get(i) % 2 == 0) a.remove(i);` döngüsü istisna fırlatmaz, ama yanlış sonuç verebilir. Bir öğe silinince arkasındaki öğe onun indisine kayar ve `i++` bu öğeyi atlar. `[4, 10, 7, 13, 16]` listesinde 4 silinir, 10 indis 0'a kayar ve hiç denetlenmez. Notun programında sonuç `[10, 7, 13]` olmuştur. Aynı döngü sondan başa doğru kurulursa kayma denetlenmemiş öğeleri etkilemez ve sonuç doğru çıkar: `[7, 13]`.

5.6.3 ConcurrentModificationException nereden gelir?

For-each içindeki silmenin neden istisna fırlattığını anlamak için yineleyicinin içine bakmak gerekir. `ArrayList`'in bir `modCount` alanı vardır. Bu alan listenin yapısal değişiklik sayacıdır: her ekleme ve silme onu bir artırır. Yineleyici oluşturulduğu anda bu sayacın değerini `expectedModCount` adıyla saklar ve bir de `cursor` (imleç) tutar. Her `next()` çağrısı önce iki sayacı karşılaştırır. Farklıysa liste yineleyicinin bilgisi dışında değiştirilmiştir ve gezinmeye devam etmek güvenli değildir. Yineleyici bu durumda `ConcurrentModificationException` fırlatır. Bu davranışa "hızlı başarısız olma" (*fail-fast*) denir.

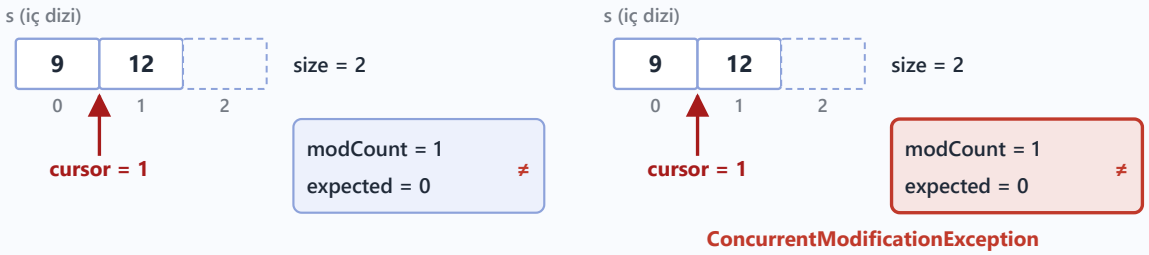
```
List<Integer> s = new ArrayList<>();
Collections.addAll(s, 6, 9, 12);
for (Integer x : s) {      // gizli Iterator
    if (x % 2 == 0) {
        s.remove(x);
    }
}
System.out.println(s);
```

Tablo 5.26 programı sayaçlarla birlikte izler.

Adım	Satır	Değişkenler	Yeni çıktı
1	1: `List<Integer> s = new ArrayList<>();` 2: `Collections.addAll(s, 6, 9, 12);`	s=[6, 9, 12], modCount=0	—
2	3: `for (Integer x : s) { // gizli Iterator`	s=[6, 9, 12], modCount=0	—
3	3: `for (Integer x : s) { // gizli Iterator`	s=[6, 9, 12], modCount=0, hasNext=true	—
4	3: `for (Integer x : s) { // gizli Iterator`	s=[6, 9, 12], modCount=0, x=6	—
5	4: `if (x % 2 == 0) {`	s=[6, 9, 12], modCount=0, x=6	—
6	5: `s.remove(x);`	s=[9, 12], modCount=1, x=6	—
7	3: `for (Integer x : s) { // gizli Iterator`	s=[9, 12], modCount=1, hasNext=true	—
8	3: `for (Integer x : s) { // gizli Iterator`	s=[9, 12], modCount=1	`Exception in thread "main" java.util.ConcurrentModificationException`

Tablo 5.26: for-each içinde silmenin izlenmesi.

Liste üç öğelidir ve `modCount` 0'dır. For-each gizlice `s.iterator()` çağırır: `cursor = 0`, `expectedModCount = 0`. İlk `hasNext()`, `cursor` (0) ile `size` (3) farklı olduğu için `true` döner. `next()` önce sayaçları karşılaştırır (0 = 0), 6'yı döndürür ve `cursor`'ı 1 yapar. 6 çifttir ve `s.remove(x)` çağrılır. Bu, listenin kendi metodudur: 9 ve 12 sola kayar, `size` 2'ye iner, `modCount` 1'e çıkar. Yineleyicinin bundan haberi yoktur. Şekil 5.22 bu anı ve sonrasını gösterir.



Şekil 5.22: Solda `s.remove(6)` sonrası: liste `[9, 12]`, `modCount = 1`, yineleyicinin `expected` değeri hâlâ 0 ve `cursor = 1`. Sağda sonraki `next()`: sayaçlar uyuşmaz ve `ConcurrentModificationException` fırlatılır.

Sonraki `hasNext()`, `cursor` (1) ile `size` (2) farklı olduğu için `true` döner; döngü sürer. Ama `next()` sayaçları karşılaştırınca `modCount` 1, `expectedModCount` 0'dır. İstisna fırlatılır ve `println`'e hiç ulaşılmaz.

```
Exception in thread "main" java.util.ConcurrentModificationException
```

İzleme bir şeyi daha gösterir: istisna, silme anında değil bir sonraki `next()`'te ortaya çıkar. Silmeden sonra `hasNext()` `false` dönerse `next()` hiç çağrılmaz ve istisna da oluşmaz. Liste `{6, 9}` olsaydı 6 silindikten sonra `cursor` 1, `size` 1 olurdu. `hasNext()` `false` döner ve döngü sessizce biterdi:

```
[9]
```

Program hatasız biter, ama 9 hiç denetlenmemiştir. Yanlış kod bazen çalışıyormuş gibi görünür. Bu yüzden "bende çalıştı" demek yeterli değildir. Gezinirken silme her zaman yineleyicinin `remove`'u ya da `removeIf` ile yapılır. İstisnanın adındaki "concurrent" (eş zamanlı) sözcüğü yanıltıcı olabilir. Bu hata çok iş parçacıklı programa özgü değildir; tek iş parçacıklı bir programda da, burada olduğu gibi, ortaya çıkar.

5.7 Algoritmalar

Koleksiyon Çerçevesinin üçüncü unsuru hazır algoritmalardır. Bunların çoğu `java.util.Collections` sınıfında toplanmıştır. Bu kesim sıralama, sıra tanımlama (`Comparable` ve `Comparator`), karıştırma, arama ve kompozisyon algoritmalarını sırasıyla anlatır.

5.7.1 Collections sınıfı

İki ad sık karıştırılır. `Collection` bir arayüzdür ve bütün koleksiyonların ortak metotlarını tanımlar. `Collections` (sonunda "s") ise koleksiyonlar üzerinde çalışan hazır `static` metotları toplayan bir yardımcı sınıftır. Bu sınıftan nesne oluşturulmaz; metotları `Collections.sort(liste)` gibi sınıf adıyla çağrılır.

Bu algoritmalar hızlı ve test edilmiştir; aynı işi elle yazmak hem zaman alır hem hata riski taşır. Dört grupta toplanırlar:

- **Sıralama**: `sort`, `reverse`, `reverseOrder`.
- **Karıştırma**: `shuffle`.
- **Arama**: `binarySearch`, `max`, `min`, `frequency`.
- **Kompozisyon**: `copy`, `fill`, `swap`, `reverse`.

5.7.2 Sıralama

```
List<String> isimler = new ArrayList<>(
        Arrays.asList("Cem", "Ayşe", "Banu", "Ali"));
Collections.sort(isimler);
System.out.println(isimler);
Collections.sort(isimler, Collections.reverseOrder());
System.out.println(isimler);
Collections.reverse(isimler);
System.out.println(isimler);
```

```
[Ali, Ayşe, Banu, Cem]
[Cem, Banu, Ayşe, Ali]
[Ali, Ayşe, Banu, Cem]
```

`Collections.sort(isimler)` listeyi doğal sıraya göre, yani alfabetik olarak sıralar. İkinci çağrı bir karşılaştırıcı alır: `Collections.reverseOrder()` doğal sırayı tersine çevirir ve liste Z'den A'ya dizilir. `Collections.reverse` ise hiçbir karşılaştırma yapmaz; listenin o anki sırasını ters çevirir. Azalan sıradaki liste ters çevrilince yeniden artan sıra elde edilir. Tablo 5.27 üç metodu özetler.

Metot	Görevi
`sort(list)`	doğal sıraya göre; öğeler `Comparable` olmalı
`sort(list, c)`	verilen karşılaştırıcıya (`Comparator`) göre
`reverse(list)`	listeyi ters sıraya çevirir

Tablo 5.27: Sıralama metotları.

`String`, `Integer`, `Double` gibi sınıflar zaten `Comparable`'dır. Java 8'den beri listenin kendi `sort` metodu da vardır: `liste.sort(null)` doğal sıraya göre sıralar, `liste.sort(c)` karşılaştırıcıya göre. Öğeleri karşılaştırılamayan bir liste `Collections.sort`'a verilirse hata derleme zamanında yakalanır:

```
OgrenciSort.java:9: error: no suitable method found for sort(List<Ogr>)
        Collections.sort(l);
                   ^
    method Collections.<T#1>sort(List<T#1>) is not applicable
      (inference variable T#1 has incompatible bounds
        equality constraints: Ogr
        upper bounds: Comparable<? super T#1>)
    method Collections.<T#2>sort(List<T#2>,Comparator<? super T#2>) is not applicable
      (cannot infer type-variable(s) T#2
        (actual and formal argument lists differ in length))
  where T#1,T#2 are type-variables:
    T#1 extends Comparable<? super T#1> declared in method <T#1>sort(List<T#1>)
    T#2 extends Object declared in method <T#2>sort(List<T#2>,Comparator<? super T#2>)
1 error
```

İleti uzundur ama özü tek satırdadır: `Ogr` tipi `Comparable<? super T>` üst sınırını sağlamaz, yani karşılaştırılabilir değildir. İkinci olarak önerilen `sort(List, Comparator)` biçimi de argüman sayısı tutmadığı için uymaz. Öncelikli kuyruktaki çalışma anı hatasından farklı olarak burada derleyici sorunu önceden görür; `sort`'un imzası tip parametresini sınırlamıştır.

`Collections.sort` nesne listelerinde TimSort algoritmasını kullanır. TimSort birleştirme sıralamasının uyarlanmış bir türüdür. En kötü durumda O(n log n) sürer ve kısmen sıralı verilerde çok daha hızlıdır. Ayrıca **kararlıdır** (*stable*): eşit sayılan öğelerin kendi arasındaki sırası korunur. İlkel dizileri sıralayan `Arrays.sort(int[])` ise Dual-Pivot Quicksort kullanır; ilkel değerlerde kararlılığın anlamı olmadığı için bu seçim yapılmıştır. Notun programı "Cem:B, Ali:A, Ece:B, Banu:A, Can:A" listesini yalnızca harf notuna göre sıralamıştır. Sonuç `[Ali:A, Banu:A, Can:A, Cem:B, Ece:B]`'dir: aynı notu alanlar listedeki eski sıralarını korur. Bu sayede alfabetik bir liste nota göre sıralanınca aynı notu alanlar alfabetik kalır.

5.7.3 Comparable: doğal sıra

Tanım 5.15

Doğal sıra (*natural ordering*), bir sınıfın `Comparable<T>` arayüzünü uygulayıp `compareTo` metodunu yazarak tanımladığı varsayılan sıralamadır. `a.compareTo(b)` negatifse `a` önce, 0 ise ikisi eşit, pozitifse `b` önce gelir.

```
class Ogrenci implements Comparable<Ogrenci> {
    String isim;
    int puan;

    Ogrenci(String isim, int puan) {
        this.isim = isim;
        this.puan = puan;
    }

    // puana göre artan sıra
    @Override
    public int compareTo(Ogrenci diger) {
        return Integer.compare(puan, diger.puan);
    }

    @Override
    public String toString() {
        return isim + ":" + puan;
    }
}
```

```
List<Ogrenci> sinif = new ArrayList<>();
sinif.add(new Ogrenci("Cem", 72));
sinif.add(new Ogrenci("Ayşe", 95));
sinif.add(new Ogrenci("Banu", 60));

Collections.sort(sinif);
System.out.println(sinif);
```

```
[Banu:60, Cem:72, Ayşe:95]
[Ayşe:95, Banu:60, Cem:72]
[Banu:60, Cem:72, Ayşe:95]
[Ayşe:95, Cem:72, Banu:60]
```

`Ogrenci` sınıfı `Comparable<Ogrenci>`'yi uygular ve puana göre artan sırayı tanımlar. `Collections.sort(sinif)` bu sırayı kullanır: Banu (60), Cem (72), Ayşe (95). Programın sonraki satırları karşılaştırıcı örnekleridir ve bir sonraki alt kesimde açıklanır.

`compareTo`'nun gövdesinde `Integer.compare(puan, diger.puan)` kullanılmıştır. Aynı iş `return puan - diger.puan;` ile de yapılabilir gibi görünür: fark negatifse `this` küçüktür. Puanlar küçük sayılar olduğu sürece bu doğru çalışır. Ama çok büyük ve zıt işaretli iki sayının farkı `int` aralığını aşar ve işaret ters döner:

```
public class Tasma {
    public static void main(String[] args) {
        int a = 2_000_000_000, b = -2_000_000_000;
        System.out.println((a - b) + " " + Integer.compare(a, b));
    }
}
```

```
-294967296 1
```

2 milyar −2 milyardan büyüktür ve fark pozitif olmalıdır. Ama 4 milyar `int`'e sığmaz ve sonuç negatif çıkar; sıralama yanlış olur. `Integer.compare` çıkarma yapmaz, yalnızca karşılaştırır ve doğru sonucu (1) verir. Bu yüzden çıkarma yerine her zaman `Integer.compare` (ya da `Double.compare`) kullanılır.

İyi bir `compareTo`'nun uyması gereken birkaç kural vardır. `a`, `b`'den önceyse `b`, `a`'dan sonra olmalıdır. `a` `b`'den, `b` de `c`'den önceyse `a` `c`'den önce olmalıdır. Ayrıca `compareTo`'nun 0 döndürdüğü nesnelerin `equals`'ının da doğru dönmesi önerilir. Bu son kural `TreeSet` ve `TreeMap` için önemlidir: bu yapılar eşitliği `equals` ile değil `compareTo` ile sınar. Puanı aynı iki farklı öğrenci bir `TreeSet`'e konursa ikincisi "tekrar" sayılır ve eklenmez.

5.7.4 Comparator: başka bir sıra

Tanım 5.16

Karşılaştırıcı (*comparator*), sınıfın dışında yazılan ayrı bir sıralama ölçütüdür. `Comparator<T>` arayüzünü uygular ve `compare(a, b)` metoduyla iki nesneyi karşılaştırır. Bir sınıfın tek bir doğal sırası, ama istenildiği kadar karşılaştırıcısı olabilir.

```
Collections.sort(sinif, new Comparator<Ogrenci>() {
    @Override
    public int compare(Ogrenci a, Ogrenci b) {
        return a.isim.compareTo(b.isim);   // isme göre
    }
});
System.out.println(sinif);

// Java 8+: aynı işin kısa yazımı
sinif.sort(Comparator.comparing(o -> o.puan));
System.out.println(sinif);
```

Bu kod, önceki programın devamıdır ve çıktının ikinci ve üçüncü satırlarını üretir. Klasik yazımda karşılaştırıcı bir **anonim sınıfla** (*anonymous class*) yazılır: `new Comparator<Ogrenci>() { … }` adı olmayan, yerinde tanımlanan ve tek nesnesi oluşturulan bir sınıftır. Bu karşılaştırıcı isimleri `String`'in `compareTo`'suyla karşılaştırır ve liste alfabetik olur: Ayşe, Banu, Cem.

Java 8 ile aynı iş tek satıra iner. `Comparator.comparing(o -> o.puan)`, her öğrenciden puanını çıkaran bir fonksiyon alır ve bu değerlere göre karşılaştıran bir karşılaştırıcı üretir. Liste yeniden puan sırasına döner. Ters sıra için `.reversed()` eklenir. Notun programındaki son satır bunu yapar: `Comparator.comparing((Ogrenci o) -> o.puan).reversed()` ile liste en yüksek puandan başlar. Lambdanın parametre tipinin bu yazımda açıkça verilmesi gerekir; `.reversed()` zinciri derleyicinin tipi soldan çıkarmasını engeller.

5.7.5 Karıştırma

Karıştırma, koleksiyon öğelerini rastgele bir sıraya sokmaktır. Kart dağıtma, kura çekme ve sınavda soru sırasını rastgele yapma tipik kullanımlardır. `Collections.shuffle(liste)` Fisher–Yates (Knuth) algoritmasını kullanır. Algoritma listenin sonundan başlar. Her adımda, henüz yerleşmemiş öğeler arasından rastgele birini seçer ve o anki konumdaki öğeyle yer değiştirir. `n` öğeli listede `n − 1` yer değiştirme yapılır ve iş O(n)'dir. Algoritmanın değerli yanı, her olası sıralamanın eşit olasılıkla ortaya çıkmasıdır (düzgün dağılım).

```
List<String> ogrenciler = new ArrayList<>(
        Arrays.asList("Ali", "Banu", "Cem", "Deniz"));
Collections.shuffle(ogrenciler);
System.out.println(ogrenciler);
```

Bu programın çıktısı her çalıştırmada farklıdır; sunumdaki `[Cem, Ali, Deniz, Banu]` olası 24 sıralamadan yalnızca biridir. Test yazarken ya da bir sonucu tekrar üretmek gerektiğinde aynı karışım istenir. Bunun için ikinci argüman olarak sabit tohumlu bir rastgele sayı üreteci verilir: `Collections.shuffle(liste, new Random(42))`. Notun programı aynı listeyi iki kez 42 tohumuyla karıştırır ve iki sonuç birebir aynıdır:

```
[Deniz, Banu, Ali, Cem]
[Deniz, Banu, Ali, Cem]
```

Loto örneği

Karıştırmanın güzel bir uygulaması tekrarsız rastgele seçimdir. 1'den 49'a kadar sayılar bir listeye konur, liste karıştırılır ve ilk 6 öğe alınır:

```
public class Loto {
    // Loto çekilişi yapan metot
    public static Integer[] lotoYap(int toplamSayi, int cekilecekSayi) {
        ArrayList<Integer> sayilar = new ArrayList<Integer>();
        for (int i = 0; i < toplamSayi; i++) {
            sayilar.add(i + 1);          // 1 … toplamSayi
        }
        Collections.shuffle(sayilar);    // karıştır
        Integer[] cekilenSayilar = new Integer[cekilecekSayi];
        for (int i = 0; i < cekilecekSayi; i++) {
            cekilenSayilar[i] = sayilar.get(i);
        }
        return cekilenSayilar;
    }

    public static void main(String[] args) {
        Integer[] sonuc = lotoYap(49, 6);   // 49'dan 6
        for (Integer item : sonuc) {
            System.out.print(item + " ");
        }
    }
}
```

Bir çalıştırmanın çıktısı şudur (her çalıştırmada farklıdır):

```
21 25 16 3 33 46
```

Karıştırma sayıları yalnızca yer değiştirdiği için listede her sayı tam bir kez bulunur. İlk altı öğe de bu yüzden tekrarsızdır. Notun sınama programı `lotoYap(49, 6)`'yı 10.000 kez çağırıp her sonucun 6 farklı ve 1–49 aralığında sayı içerdiğini denetlemiştir; hata sayısı 0'dır. Maliyet, seçilen sayı adedinden bağımsız olarak bütün listeyi kurup karıştırmanın maliyetidir: `toplamSayi` kadar, yani O(n).

5.7.6 Arama

Bir koleksiyonda öğe aramanın iki temel yolu vardır.

Doğrusal arama (*linear search*)
: Öğeleri baştan sona tek tek kontrol eder. Koleksiyonun sıralı olması gerekmez. Küçük koleksiyonlarda etkilidir. Maliyeti O(n)'dir. `indexOf` ve `contains` doğrusal arama yapar.

İkili arama (*binary search*)
: Sıralı koleksiyonda ortadaki öğeye bakar. Aranan değer küçükse sol yarıda, büyükse sağ yarıda aramaya devam eder. Her adımda aralık yarıya iner. Maliyeti O(log n)'dir. Koleksiyon sıralı olmalıdır.

```
List<Integer> sayilar = Arrays.asList(1, 3, 5, 7, 9, 11);
System.out.println(Collections.binarySearch(sayilar, 7));
System.out.println(Collections.binarySearch(sayilar, 6));
System.out.println(sayilar.indexOf(7));    // doğrusal arama
```

```
3
-4
3
-1 -7
2
```

İlk üç satır sunumdaki çıktıdır. `binarySearch(sayilar, 7)` 7'yi indis 3'te bulur. Arama şu adımlarla ilerler. Aralık 0–5'tir; ortadaki indis 2'deki 5, 7'den küçüktür ve arama sağ yarıya geçer. Aralık 3–5'tir; ortadaki indis 4'teki 9, 7'den büyüktür ve sol yarıya geçilir. Aralık 3–3'tür ve indis 3'teki 7 bulunur. Üç karşılaştırma yapılmıştır. Notun sayım programı, karşılaştırıcıya bir sayaç ekleyerek bu sayıyı doğrular:

```
7: sonuç 3, 3 karşılaştırma
6: sonuç -4, 3 karşılaştırma
1 milyon öğe, denenen en çok: 20
```

6 listede yoktur. Arama bitince 6'nın girmesi gereken yer, yani **ekleme noktası** 3'tür: 5'ten sonra, 7'den önce. Bulunamayan değer için dönüş `-(ekleme noktası) - 1`'dir: −(3) − 1 = −4. Bu biçim iki bilgiyi birden taşır. Sonuç negatifse öğe yoktur; `-(sonuç) - 1` ise öğenin sırayı bozmadan ekleneceği yeri verir. “Neden yalnızca −3 değil?” sorusunun yanıtı 0'dır: ekleme noktası 0 olsaydı −0 = 0 çıkar ve “indis 0'da bulundu” ile karışırdı. Notun programının dördüncü satırı uç durumları gösterir. 0 aranınca ekleme noktası 0'dır ve dönüş −1'dir. 12 aranınca ekleme noktası 6'dır (listenin sonu) ve dönüş −7'dir.

Liste sıralı değilse sonuç tanımsızdır. Programın son satırı 7'yi sırasız bir listede arar ve 2 döndürür. 7 gerçekten de indis 2'dedir, ama bu doğru sonuç tesadüftür. Başka bir değer aransa yanlış bir indis ya da “yok” dönebilirdi. `binarySearch` listenin sıralı olup olmadığını denetlemez; bu denetim O(n) süreceği için yapılmaz.

İki aramanın farkı büyük verilerde çarpıcıdır. Bir milyon öğede doğrusal arama bir milyona kadar karşılaştırma yapar. İkili arama ise en fazla 20 karşılaştırma yapar, çünkü $2^{20} \approx 1$ milyondur. Sayım programının son satırı, bir milyon öğeli listede denenen değerler arasında en çok 20 karşılaştırma yapıldığını gösterir. Sık arama yapılan bir liste bir kez sıralanır ve sonra ikili aramayla kullanılır. Sıralama O(n log n) sürer, ama sonraki her arama O(log n)'dir.

5.7.7 Kompozisyon algoritmaları

Kompozisyon algoritmaları bir listenin öğelerini kopyalar, değiştirir, yerlerini değiştirir ya da sayar. Tablo 5.28 bunları toplar.

Metot	Görevi
`copy(hedef, kaynak)`	kaynağın öğelerini hedefin başına kopyalar; `hedef.size() ≥ kaynak.size()` olmalı
`fill(liste, d)`	bütün öğeleri `d` yapar
`reverse(liste)`	öğelerin sırasını tersine çevirir
`swap(liste, i, j)`	`i` ve `j` konumlarını değiştirir
`max` / `min`	en büyük / en küçük öğe
`frequency(c, o)`	`o`'nun kaç kez geçtiği

Tablo 5.28: Kompozisyon algoritmaları.

```
List<Integer> n = Arrays.asList(4, 9, 2, 9, 5);
System.out.println(Collections.max(n));
System.out.println(Collections.min(n));
System.out.println(Collections.frequency(n, 9));
```

```
9
2
2
```

`max` ve `min` listenin bütün öğelerine bakar ve doğal sıraya göre en büyüğü ve en küçüğü verir: 9 ve 2. `frequency(n, 9)` 9'un kaç kez geçtiğini sayar: 2. Üçü de listede bir kez gezinir ve O(n)'dir. Listenin sıralı olması gerekmez.

```
List<String> isimler =
        Arrays.asList("Ahmet", "Ayşe", "Cem");
List<String> hedef = new ArrayList<>(
        Arrays.asList("X", "Y", "Z", "W"));
Collections.copy(hedef, isimler);
System.out.println(hedef);
Collections.reverse(hedef);
System.out.println(hedef);
Collections.swap(hedef, 0, 2);
System.out.println(hedef);
Collections.fill(hedef, "Java");
System.out.println(hedef);
```

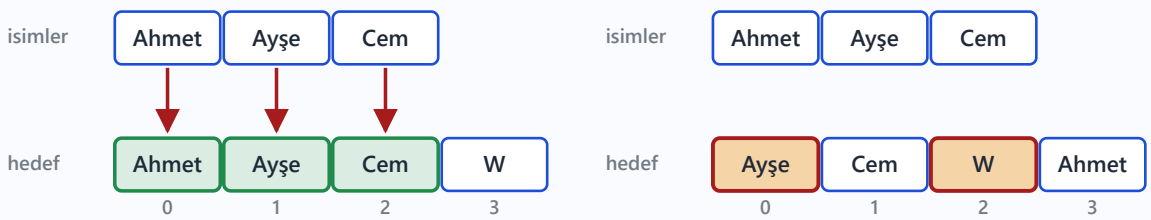
```
[Ahmet, Ayşe, Cem, W]
[W, Cem, Ayşe, Ahmet]
[Ayşe, Cem, W, Ahmet]
[Java, Java, Java, Java]
```

Tablo 5.29 dört işlemi adım adım verir.

Adım	Satır	Listeler	Yeni çıktı
1	1: `List<String> isimler =` 2: `Arrays.asList("Ahmet", "Ayşe", "Cem");` 3: `List<String> hedef = new ArrayList<>(` 4: `Arrays.asList("X", "Y", "Z", "W"));`	Ahmet, Ayşe, Cem; hedef: X, Y, Z, W	—
2	5: `Collections.copy(hedef, isimler);` 6: `System.out.println(hedef);`	Ahmet, Ayşe, Cem; hedef: Ahmet, Ayşe, Cem, W	`[Ahmet, Ayşe, Cem, W]`
3	7: `Collections.reverse(hedef);` 8: `System.out.println(hedef);`	Ahmet, Ayşe, Cem; hedef: W, Cem, Ayşe, Ahmet	`[W, Cem, Ayşe, Ahmet]`
4	9: `Collections.swap(hedef, 0, 2);` 10: `System.out.println(hedef);`	Ahmet, Ayşe, Cem; hedef: Ayşe, Cem, W, Ahmet	`[Ayşe, Cem, W, Ahmet]`
5	11: `Collections.fill(hedef, "Java");` 12: `System.out.println(hedef);`	Ahmet, Ayşe, Cem; hedef: Java, Java, Java, Java	`[Java, Java, Java, Java]`

Tablo 5.29: Kompozisyon işlemlerinin izlenmesi.

`isimler`, `Arrays.asList` ile oluşturulmuş sabit boyutlu bir listedir. `hedef` ise dört öğeli değiştirilebilir bir `ArrayList`'tir. `copy` kaynağın üç öğesini hedefin 0, 1 ve 2 numaralı konumlarına yazar. Hedefin dördüncü öğesi W yerinde kalır; `copy` hedefin boyunu değiştirmez. `reverse` sırayı tersine çevirir: `[W, Cem, Ayşe, Ahmet]`. `swap(hedef, 0, 2)` 0 ve 2 numaralı konumları değiştirir: W ile Ayşe yer değiştirir. `fill` bütün öğeleri "Java" yapar; boyut yine 4'tür. Şekil 5.23 iki ara durumu gösterir.



Şekil 5.23: Solda `copy` sonrası: hedef `[Ahmet, Ayşe, Cem, W]`. Sağda `reverse` ve `swap(hedef, 0, 2)` sonrası: `[Ayşe, Cem, W, Ahmet]`.

`copy`'nin hedef koşulu sık hata yaptırır. Hedef kaynaktan kısaysa istisna fırlatılır. Asıl tuzak, kapasitesi yeterli ama boş bir listedir. `new ArrayList<>(10)` 10 hücre ayırır, ama boyu 0'dır; `copy` kapasiteye değil boya bakar:

```
import java.util.*;
public class BosHedef {
    public static void main(String[] args) {
        List<String> kaynak = Arrays.asList("a", "b", "c");
        List<String> hedef = new ArrayList<>(10);
        System.out.println(hedef.size());
        Collections.copy(hedef, kaynak);
    }
}
```

```
0
Exception in thread "main" java.lang.IndexOutOfBoundsException: Source does not fit in
dest
```

Bir listenin bağımsız kopyası gerekiyorsa `copy` yerine kopyalayan yapıcı kullanılır: `new ArrayList<>(kaynak)`. Bu yol hedefin boyunu dert etmez ve tek satırdır.

5.8 Akış API'si

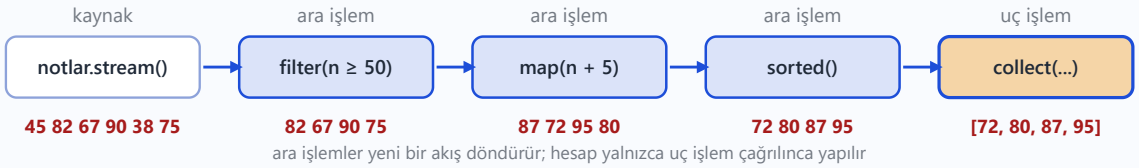
Önceki kesimlerdeki işlerin çoğu aynı kalıbı izler: bir koleksiyonu gez, bazı öğeleri seç, onları dönüştür, sonucu yeni bir koleksiyonda topla. Bu kalıp her seferinde bir döngü, bir koşul ve bir ara liste yazdırır. Java 8 ile gelen Akış API'si aynı işi bir işlem hattı olarak yazmayı sağlar.

5.8.1 Akış ve işlem hattı

Tanım 5.17

Akış (*stream*), `java.util.stream` paketinde bir veri dizisini **işlem hattı** (*pipeline*) olarak işleyen nesnedir. Hat bir kaynak, sıfır ya da daha çok ara işlem ve bir uç işlemden oluşur. Programcı "ne yapılacağını" söyler; döngüyü kütüphane kurar.

Buradaki akış, `InputStream` gibi giriş/çıkış (G/Ç) akışlarından farklı bir kavramdır; yalnızca adları aynıdır. Bir akış veri saklamaz. Bir kaynağın (liste, dizi, dosya satırları) üzerinden geçer ve kaynağı değiştirmez. Şekil 5.24 bir hattı gösterir.



Şekil 5.24: Akış hattı. Kaynak `notlar.stream()`; ara işlemler `filter(n ≥ 50)`, `map(n + 5)` ve `sorted()`; uç işlem `collect(...)`. Her kutunun altında o aşamadan geçen değerler yazılıdır. Ara işlemler yeni bir akış döndürür; hat yalnızca uç işlem çağrılınca çalışır.

```
List<Integer> notlar = Arrays.asList(45, 82, 67, 90, 38, 75);
List<Integer> gecenler = notlar.stream()
    .filter(n -> n >= 50)
    .map(n -> n + 5)
    .sorted()
    .collect(Collectors.toList());
System.out.println(gecenler);
```

```
[72, 80, 87, 95]
[45, 82, 67, 90, 38, 75]
[72, 80, 87, 95]
[72, 80, 87, 95]
```

Hat dört adımdan oluşur. `filter` 50 ve üstündeki notları geçirir: 82, 67, 90, 75. `map` her birine 5 ekler: 87, 72, 95, 80. `sorted` bunları sıralar. `collect(Collectors.toList())` sonucu yeni bir listede toplar. Notun programındaki ikinci satır kaynağın değişmediğini, üçüncü satır Java 16'daki kısa yazımı (`.toList()`), dördüncü satır ise aynı işin döngüyle yazılmış hâlini gösterir. Döngü sürümü bir ara liste, bir koşul ve ayrı bir sıralama çağrısı ister. Akış sürümü her adımı adıyla söyler. Gereken içe aktarmalar `java.util.*` ve `java.util.stream.Collectors`'tır.

`Collectors.toList()` ile `.toList()` arasında küçük bir fark vardır. İlki değiştirilebilir bir liste döndürür (pratikte bir `ArrayList`). İkincisi değiştirilemez bir liste döndürür; ona `add` çağrılırsa `UnsupportedOperationException` fırlatılır.

5.8.2 Ara ve uç işlemler

Ara işlem	Görevi	Uç işlem	Sonuç
`filter(p)`	koşulu sağlayanları geçirir	`collect(...)`	liste, küme, eşleme…
`map(f)`	her öğeyi dönüştürür	`forEach(f)`	her öğe için bir eylem
`flatMap(f)`	iç içe akışları düzleştirir	`count()`	öğe sayısı (`long`)
`sorted()`	sıralar (bütün öğeleri bekler)	`reduce(...)`	öğeleri tek değere indirger

Ara işlem	Görevi	Uç işlem	Sonuç
`distinct()`	tekrarları atar (`equals`)	`findFirst()`	ilk öğe (`Optional`)
`limit(n)` / `skip(n)`	ilk `n` öğeyi alır / atlar	`anyMatch` / `allMatch` / `noneMatch`	`boolean`
`peek(f)`	öğeleri izler (hata ayıklama)	`min` / `max`	`Optional`

Tablo 5.30: Ara işlemler ve uç işlemler.

Ara işlem yeni bir akış döndürür ve zincir sürer. Uç işlem akış dışında bir sonuç (ya da hiçbir şey) döndürür ve hattı çalıştırır. Uç işlemi olmayan bir hat hiçbir şey yapmaz. Tablodaki işlemlerin birkaçı notun programında denenmiştir:

```
import java.util.*;
import java.util.stream.*;
public class Ana68 {
    public static void main(String[] args) {
        List<List<Integer>> ic = Arrays.asList(Arrays.asList(1, 2), Arrays.asList(3),
Arrays.asList(4, 5));
        System.out.println(ic.stream().flatMap(List::stream).collect(Collectors.toList()));
        System.out.println(Stream.of(3, 1, 3, 2,
1).distinct().collect(Collectors.toList()));
        System.out.println(Stream.of(1, 2, 3, 4, 5,
6).skip(2).limit(3).collect(Collectors.toList()));
        System.out.println(Stream.of(4, 8, 15).count());
        System.out.println(Stream.of(4, 8, 15).anyMatch(x -> x > 10) + " " + Stream.of(4,
8, 15).allMatch(x -> x > 3)
                + " " + Stream.of(4, 8, 15).noneMatch(x -> x > 20));
        System.out.println(Stream.of(4, 8, 15).max(Integer::compare) + " " + Stream.
<Integer>empty().findFirst());
        Stream.of("a", "b").peek(s -> System.out.print("peek " + s + "; ")).forEach(s ->
System.out.print("forEach " + s + "; "));
        System.out.println();
        long n = Stream.of(1, 2, 3).peek(x -> System.out.print("görülmez ")).count();
        System.out.println("count " + n);
    }
}
```

```
[1, 2, 3, 4, 5]
[3, 1, 2]
[3, 4, 5]
3
true true true
Optional[15] Optional.empty
peek a; forEach a; peek b; forEach b;
count 3
```

`flatMap` üç iç listeyi tek bir sayı akışına açar. `distinct` tekrar eden 3 ve 1'i atar ve ilk görülme sırasını korur. `skip(2).limit(3)` ilk iki öğeyi atlar ve sonraki üçünü alır. `max` sonucu `Optional` içinde verir; boş akışta `findFirst` `Optional.empty` döndürür. `peek` satırı öğelerin hattan tek tek geçtiğini gösterir: "peek a" hemen ardından "forEach a" gelir. Son satır ise bir uyarıdır. JDK 21, `count()` 'un sonucunu kaynağın boyundan doğrudan bildiği için hattı hiç çalıştırmaz ve `peek` 'teki yazı görülmez. `peek` yalnızca hata ayıklama içindir; asıl iş onun içine konmaz.

5.8.3 Tembel değerlendirme ve tek kullanım

Tanım 5.18

Tembel değerlendirme (*lazy evaluation*): ara işlemler çağrıldıklarında hiçbir öğeyi işlemez, yalnızca bir tarif kaydeder. Öğeler, uç işlem çağrılınca ve tek tek hattan geçerek işlenir.

```
List<String> adlar =
    Arrays.asList("Arda", "Ece", "Ayşe", "Can");
Stream<String> hat = adlar.stream()
    .filter(s -> s.startsWith("A"))
    .map(String::toUpperCase);
// Henüz hiçbir öğe işlenmedi.
List<String> sonuc =
    hat.collect(Collectors.toList());
System.out.println(sonuc);  // [ARDA, AYŞE]
hat.count();   // IllegalStateException
```

```
[ARDA, AYŞE]
Exception in thread "main" java.lang.IllegalStateException: stream has already been
operated upon or closed
```

`hat` değişkeni bir `Stream<String>` 'dir. İlk üç satır çalıştığında henüz hiçbir ad süzülmemiş, hiçbir harf büyütülmemiştir. `collect` çağrılınca hat çalışır ve sonuç `[ARDA, AYŞE]` olur. `String::toUpperCase` bir **metot referansıdır**; `s -> s.toUpperCase()` lambdasının kısa yazımıdır. Son satır ikinci kuralı gösterir: akış **tek kullanımlıktır**. Uç işlemden sonra aynı akış yeniden kullanılırsa `IllegalStateException` fırlatılır. Gerekirse kaynaktan yeni bir akış alınır: `adlar.stream()` .

Tembelliğin iki önemli sonucu vardır. Birincisi, öğeler hattan *tek tek* geçer. İlk öğe `filter` 'dan ve `map` 'ten geçer, ancak sonra ikinci öğe gelir. İkincisi, `findFirst` ve `limit` gibi işlemler sonucu bulunca durur ve kalan öğeler hiç işlenmez. Bu, sonsuz bir kaynakla bile çalışmayı mümkün kılar: `Stream.iterate(1, x -> x * 2).filter(x -> x > 100).findFirst()` 1, 2, 4, ... diye sonsuza giden bir akıştan 128'i bulur ve durur. `sorted` ise bu kuralın istisnasıdır: sıralayabilmek için önce bütün öğeleri toplaması gerekir.

```
import java.util.*;
import java.util.stream.*;
public class SortedBekler {
    public static void main(String[] args) {
        List<Integer> l = Stream.of(3, 1, 2)
            .peek(x -> System.out.println("önce " + x))
            .sorted()
            .peek(x -> System.out.println("sonra " + x))
            .collect(Collectors.toList());
        System.out.println(l);
    }
}
```

```
önce 3
önce 1
önce 2
sonra 1
sonra 2
sonra 3
[1, 2, 3]
```

Üç “önce” satırı “sonra” satırlarından önce gelir; `sorted` hatta bir bekleme noktasıdır; bütün öğeler gelene kadar hiçbir öğeyi ileri geçirmez.

5.8.4 Akış hattını adım adım izlemek

Tembelliği gözle görmek için `filter` ve `map`'e birer yazdırma satırı ekleyelim:

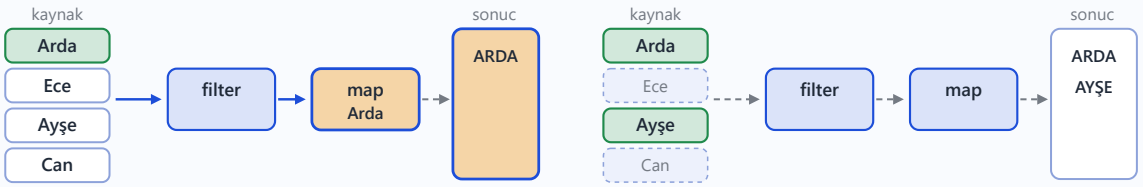
```
List<String> adlar = Arrays.asList("Arda", "Ece", "Ayşe", "Can");
Stream<String> hat = adlar.stream()
    .filter(s -> { System.out.println("filter " + s);
                   return s.startsWith("A"); })
    .map(s -> { System.out.println("map " + s);
                return s.toUpperCase(); });
System.out.println("hat hazır");
List<String> sonuc = hat.collect(Collectors.toList());
System.out.println(sonuc);
```

```
hat hazır
filter Arda
map Arda
filter Ece
filter Ayşe
map Ayşe
filter Can
[ARDA, AYŞE]
```

Adım	Satır	Değişkenler	Yeni çıktı
1	1: `List<String> adlar = Arrays.asList("Arda", "Ece", "Ayşe", "Can");` 2: `Stream<String> hat = adlar.stream()` 3: `.filter(s -> { System.out.println("filter " + s);` 4: `return s.startsWith("A"); })` 5: `.map(s -> { System.out.println("map " + s);` 6: `return s.toUpperCase(); });`	adlar=[Arda, Ece, Ayşe, Can], hat=Stream	—
2	7: `System.out.println("hat hazır");`	hat=Stream	`hat hazır`
3	8: `List<String> sonuc = hat.collect(Collectors.toList());` 3: `.filter(s -> { System.out.println("filter " + s);` 4: `return s.startsWith("A"); })`	s="Arda"	`filter Arda`
4	8: `List<String> sonuc = hat.collect(Collectors.toList());` 5: `.map(s -> { System.out.println("map " + s);` 6: `return s.toUpperCase(); });`	s="Arda"	`map Arda`
5	8: `List<String> sonuc = hat.collect(Collectors.toList());` 3: `.filter(s -> { System.out.println("filter " + s);` 4: `return s.startsWith("A"); })`	s="Ece"	`filter Ece`
6	8: `List<String> sonuc = hat.collect(Collectors.toList());` 3: `.filter(s -> { System.out.println("filter " + s);` 4: `return s.startsWith("A"); })`	s="Ayşe"	`filter Ayşe`
7	8: `List<String> sonuc = hat.collect(Collectors.toList());` 5: `.map(s -> { System.out.println("map " + s);` 6: `return s.toUpperCase(); });`	s="Ayşe"	`map Ayşe`
8	8: `List<String> sonuc = hat.collect(Collectors.toList());` 3: `.filter(s -> { System.out.println("filter " + s);` 4: `return s.startsWith("A"); })`	s="Can"	`filter Can`
9	8: `List<String> sonuc = hat.collect(Collectors.toList());`	sonuc=[ARDA, AYŞE]	—
10	9: `System.out.println(sonuc);`	sonuc=[ARDA, AYŞE]	`[ARDA, AYŞE]`

Tablo 5.31: Tembel akış hattının izlenmesi.

Hat kurulurken hiçbir lambda çalışmaz ve çıktı boştur. “hat hazır” satırı bu yüzden bütün işlemlerden önce yazılır. `collect` hattı çalıştırınca ilk öğe Arda `filter`'a girer, “A” ile başladığı için geçer ve hemen `map`'e gider. İkinci öğeye ancak o zaman sıra gelir. Ece koşulu sağlamaz ve `map`'e hiç ulaşmaz. Ayşe geçer ve büyütülür. Can elenir. Sonuçta `filter` 4 kez, `map` yalnızca 2 kez çalışmıştır. Şekil 5.25 iki anı gösterir.



Şekil 5.25: Solda Arda `filter`'ı geçip `map`'e ulaşmış ve ARDA sonuca eklenmiştir; öteki öğeler henüz kaynakta bekler. Sağda hat bitmiştir: sonuçta ARDA ve AYŞE vardır.

Aynı hatta `collect` yerine `findFirst` kullanılırsa hat ilk sonucu bulunca durur:

```
filter Arda
map Arda
ARDA
```

Yalnızca Arda işlenmiştir; Ece, Ayşe ve Can için hiçbir lambda çağrılmamıştır. Döngüyle yazılmış bir koddaki `break`'in karşılığı budur.

5.8.5 reduce, Collectors ve ilkel akışlar

```
List<String> meyveler = Arrays.asList(
    "elma", "kivi", "muz", "erik", "incir");
int toplam = Arrays.asList(4, 8, 15, 16, 23)
    .stream().reduce(0, Integer::sum);
Map<Integer, List<String>> boy = meyveler
    .stream().collect(
        Collectors.groupingBy(String::length));
String hepsi = meyveler.stream()
    .collect(Collectors.joining(", "));
int t = IntStream.rangeClosed(1, 100).sum();
Optional<String> z = meyveler.stream()
    .filter(m -> m.startsWith("z")).findFirst();
System.out.println(z.orElse("yok"));
```

```
yok
toplam = 66
boy = {3=[muz], 4=[elma, kivi, erik], 5=[incir]}
hepsi = elma, kivi, muz, erik, incir
t = 5050
OptionalDouble[50.5] OptionalDouble.empty
{3=1, 4=3, 5=1}
20
ilk e: elma
Optional[27]
```

`reduce(0, Integer::sum)` öğeleri tek bir değere indirger. Başlangıç değeri 0'dır ve her öğe bir önceki sonuca eklenir: 0 + 4 + 8 + 15 + 16 + 23 = 66. `Collectors.groupingBy(String::length)` meyveleri uzunluklarına göre gruplar ve bir `Map<Integer, List<String>>` üretir. `Collectors.joining(", ")` dizgileri araya ayıraç koyarak birleştirir. `IntStream.rangeClosed(1, 100)` 1'den 100'e kadar `int` değerlerinden oluşan bir akıştır. İlkel akışlar (`IntStream`, `LongStream`, `DoubleStream`) kutulama yapmaz ve `sum`, `average` gibi hazır uç işlemler sunar; toplam 5050'dir.

`findFirst`'ün dönüş tipi `Optional<String>`'dir. `Optional`, "sonuç olmayabilir" durumunu `null` yerine açıkça ifade eden bir kaptır. "z" ile başlayan meyve yoktur; `orElse("yok")` boş kap için verilen varsayılanı döndürür. `ifPresent` ise yalnızca değer varsa bir iş yapar. Ek satırlar üç şeyi daha gösterir. `average` bir `OptionalDouble` döndürür, çünkü boş akışta ortalama tanımsızdır. `groupingBy`'a ikinci toplayıcı olarak `counting()` verilebilir. `mapToInt` nesne akışından ilkel akışa geçer. Boş bir `Optional` üzerinde `get()` çağırmak `NoSuchElementException: No value present` fırlatır. Bu yüzden `get` yerine `orElse` ya da `ifPresent` tercih edilir.

5.8.6 Yan etkiler ve paralel akışlar

Akışlar "ne yapılacağını" tarif eder; hattaki lambdaların dış dünyayı değiştirmemesi beklenir. Dış bir değişkeni ya da koleksiyonu değiştiren lambdaya **yan etkili** denir. Sonuç `map`, `collect`, `reduce` ile üretilmelidir. Bu kural paralel akışlarda zorunlu hâle gelir. `parallel()` ya da `parallelStream()`, işi ortak `ForkJoinPool`'un iş parçacıklarına böler.

```
List<Integer> liste = new ArrayList<>();
IntStream.range(0, 10000).parallel()
    .forEach(liste::add);          // YANLIŞ: paylaşılan liste

List<Integer> dogru = IntStream.range(0, 10000).parallel()
    .boxed()
    .collect(Collectors.toList()); // DOĞRU: 10000 öğe
```

İlk hatta birden çok iş parçacığı aynı `ArrayList`'e aynı anda `add` yapar. `ArrayList` iş parçacığı güvenli değildir. İki iş parçacığı aynı hücreye yazabilir ya da biri iç diziyi büyütürken öteki eski diziye yazabilir. Sonuç çalıştırmaya göre değişir. Notun ölçüm programı ilk hattı 12 çekirdekli bir makinede

200 kez çalıştırmıştır:

```
200 deneme: AIOOBE 58, eksik 91 (en az 638, en çok 10000), tam 51, diğer 0; çekirdek 12
200 deneme: AIOOBE 51, eksik 145 (en az 969, en çok 10000), tam 4, diğer 0; çekirdek 12
```

Üç sonuç da görülmüştür: `ArrayIndexOutOfBoundsException`, 10.000'den az öğe ve bazen tesadüfen doğru boyut. Doğru sonucun ara sıra çıkması hatayı daha da tehlikeli yapar; bir sınama bunu yakalamayabilir. İkinci hat her çalıştırmada 10.000 öğe ve doğru sırayı verir. `collect` her iş parçacığına kendi ara listesini verir ve bunları sonda güvenle birleştirir.

```
boyut 6805
doğru 10000 0 9999
```

Paralel akış her zaman hızlı değildir. İşi bölmek, iş parçacıklarına dağıtmak ve sonuçları birleştirmek bir maliyettir. Küçük veride, ucuz işlemlerde, G/Ç bekleyen işlerde ve sıraya bağlı işlemlerde (`sorted`, `findFirst`) paralel akış yavaşlatabilir. Paralelliğe geçmeden önce ölçülür. Akışlarla ilgili sık hatalar şunlardır: akışı ikinci kez kullanmak, asıl işi `peek` içinde yapmak, `forEach` ile dış koleksiyon doldurmak, paralel akışı kendiliğinden hızlı saymak.

5.9 Doğru koleksiyonu seçmek

5.9.1 Seçim tablosu

Koleksiyon seçimi iki soruyla başlar: hangi işlem en sık yapılacak ve sıra önemli mi? Tablo 5.32 tipik ihtiyaçları karşılayan sınıfları verir.

İhtiyaç	Seçim	Neden
Sıralı liste, indisle erişim	`ArrayList`	hızlı `get(i)`; varsayılan liste
Başa/sona sık ekleme-çıkarma, kuyruk, yığın	`ArrayDeque`	iki uç O(1); `Stack` ve `LinkedList`'ten hızlı
En öncelikli öğeyi sürekli almak	`PriorityQueue`	`poll` en küçüğü (ya da ölçüte göre en önceliklisini) verir
Tekrarsız öğeler, hızlı "var mı?"	`HashSet`	`contains` O(1) ortalama
Tekrarsız + eklenme sırası	`LinkedHashSet`	sıra korunur
Tekrarsız + sıralı, aralık sorgusu	`TreeSet`	`first`, `headSet`, `subSet`
Anahtardan değere hızlı erişim	`HashMap`	`get` O(1) ortalama; varsayılan eşleme
Anahtarlar sıralı	`TreeMap`	`firstKey`, `headMap`, `subMap`

Tablo 5.32: İhtiyaca göre koleksiyon seçimi.

Değişken arayüz tipinde tanımlanırsa (`List` , `Set` , `Map` , `Queue` , `Deque`) ihtiyaç değiştiğinde yalnızca `new` satırı değişir. Önce doğru arayüz, sonra gerçekleme seçilir.

5.9.2 Değiştirilemez koleksiyonlar

Bazı koleksiyonların oluşturulduktan sonra hiç değişmemesi istenir: haftanın günleri, sabit renk listesi, bir yapılandırma tablosu. Java 9 ile gelen fabrika metotları bu tür koleksiyonları tek satırda üretir:

```
List<String> r = List.of("kırmızı", "yeşil");
Set<Integer> asal = Set.of(2, 3, 5);
Map<String, Integer> puan =
        Map.of("Ali", 90, "Banu", 85);
System.out.println(r);
System.out.println(puan.get("Banu"));
r.add("mavi");
```

```
[kırmızı, yeşil]
85
Exception in thread "main" java.lang.UnsupportedOperationException
```

`List.of` , `Set.of` ve `Map.of` **değiştirilemez** (*immutable*) koleksiyonlar döndürür. Okuma serbesttir; `add` , `remove` , `put` ise `UnsupportedOperationException` fırlatır. Bu koleksiyonlar `null` öğe kabul etmez. `Set.of(1, 1)` gibi tekrar içeren bir çağrı da `IllegalArgumentException: duplicate element: 1` fırlatır.

Java 1.2'den beri bulunan ikinci yol `Collections.unmodifiableList` 'tir. Bu metot yeni bir koleksiyon değil, bir **salt okunur görünüm** döndürür:

```
List<String> liste = new ArrayList<>();
liste.add("a");
List<String> salt =
        Collections.unmodifiableList(liste);
liste.add("b");
System.out.println(salt);
salt.add("c");
```

```
[a, b]
[a, b, x] [a, b]
Exception in thread "main" java.lang.UnsupportedOperationException
```

`salt` görünümü üzerinden değişiklik yapılamaz, ama alttaki `liste` değişirse görünüm de değişir: "b" eklenince `salt` `[a, b]` olur. Notun programında ayrıca `List.copyOf(liste)` (Java 10+) ile bağımsız bir kopya alınmıştır. Sonra listeye "x" eklenir; görünüm bunu gösterir, kopya ise `[a, b]` olarak kalır. Sınıf içindeki bir listeyi dışarıya "yalnızca okunsun" diye vermek için görünüm, gerçekten sabit bir liste için `List.of` ya da `List.copyOf` kullanılır. Değiştirilemez koleksiyonlar hash anahtarları ve sabitler için de güvenli seçimlerdir.

5.10 Örnek problemler

Örnek 5.1

Aşağıdaki program ne yazdırır? `fibonacci(10)` dönerken listede kaç öğe vardır ve son üçü nedir? `fibonacci(1)` çağrısında döngü kaç kez döner? Bu iş için liste gerçekten gerekli mi?

```
public class Fibonacci {
    public static int fibonacci(int sayi) {
        List<Integer> liste = new ArrayList<>();
        liste.add(0);
        liste.add(1);
        for (int i = 2; i < sayi + 1; i++) {
            liste.add(liste.get(i - 2) + liste.get(i - 1));
        }
        return liste.get(sayi);
    }
    public static void main(String[] args) {
        int sonuc = fibonacci(10);
        System.out.println(sonuc);
    }
}
```

Liste 0 ve 1 ile başlar. Döngü `i = 2` 'den `i = 10` 'a kadar dokuz kez döner ve her turda son iki öğenin toplamını ekler. `fibonacci(10)` dönerken listede 11 öğe vardır (indis 0…10): `[0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55]`. Son üçü 21, 34, 55'tir ve program 55 yazar. `fibonacci(1)` 'de `i = 2` için `2 < 2` yanlıştır ve döngü hiç dönmez; `liste.get(1)` = 1 döner. Liste gerçekten gerekli değildir: yalnızca son iki değeri tutan iki değişken yeterlidir ve bellek O(n)'den O(1)'e iner. Notun programı ek çağrılarla bir sınırı da gösterir:

```
import java.util.*;
public class Fibonacci {
    public static int fibonacci(int sayi) {
        List<Integer> liste = new ArrayList<>();
        liste.add(0);
        liste.add(1);
        for (int i = 2; i < sayi + 1; i++) {
            liste.add(liste.get(i - 2) + liste.get(i - 1));
        }
        return liste.get(sayi);
    }
    public static void main(String[] args) {
        int sonuc = fibonacci(10);
        System.out.println(sonuc);
        System.out.println(fibonacci(1) + " " + fibonacci(0) + " " + fibonacci(46));
        System.out.println(fibonacci(47));
    }
}
```

```
55
1 0 1836311903
-1323752223
```

`fibonacci(46)` = 1.836.311.903 `int`'e sığan son Fibonacci sayısıdır. 47'nci terim `int` sınırını aşar ve taşma yüzünden negatif bir sayı çıkar. Koleksiyon içindeki `Integer`'lar da ilkel `int`'in sınırlarına bağlıdır; daha büyük değerler için `Long` ya da `BigInteger` kullanılır.

Örnek 5.2

Aşağıdaki `tekrarSiz` metodu ne döndürür? `LinkedHashSet` yerine `HashSet` ya da `TreeSet` kullanılsaydı ne değişirdi?

```
public class TekrarSiz {
    static List<String> tekrarSiz(List<String> giris) {
        Set<String> gorulen = new LinkedHashSet<>(giris);
        return new ArrayList<>(gorulen);
    }

    public static void main(String[] args) {
        List<String> l = Arrays.asList("b", "a", "b", "c", "a");
        System.out.println(tekrarSiz(l));
    }
}
```

```
[b, a, c]
[a, b, c] [a, b, c]
```

`new LinkedHashSet<>(giris)` listedeki öğeleri sırayla kümeye ekler; tekrarlar atılır, ilk görülme sırası korunur. Sonuç `[b, a, c]`'dir. `TreeSet` alfabetik sıra verirdi: `[a, b, c]`. `HashSet` ise sıra garanti etmez. Programın ikinci satırında `HashSet`'in sonucu da `[a, b, c]` çıkmıştır; tek harfli dizgilerin hash kodları ardışık sayılar olduğu için kovalar alfabetik sıraya denk gelmiştir. Bu bir tesadüftür ve güvenilmez. Metodun maliyeti O(n)'dir: her öğe için bir ortalama O(1) ekleme.

Örnek 5.3

Parantez dengesi: her açılan parantez doğru türde ve doğru sırada kapanmalıdır. `dengeliMi("(a[b]c)")`, `dengeliMi("(a[b)c]")` ve `dengeliMi("((")` ne döndürür? Neden yığın kullanılmıştır?

```
static boolean dengeliMi(String ifade) {
    Deque<Character> yigin = new ArrayDeque<>();
    for (char c : ifade.toCharArray()) {
        if (c == '(' || c == '[') {
            yigin.push(c);
        } else if (c == ')' || c == ']') {
            if (yigin.isEmpty()) return false;
            char ac = yigin.pop();
            if (c == ')' && ac != '(') return false;
            if (c == ']' && ac != '[') return false;
        }
    }
    return yigin.isEmpty();
}
```

Açılan parantez yığına itilir. Kapanan parantez gelince yığının üstündeki açılan parantez çıkarılır ve türü karşılaştırılır. Yığın boşsa ya da tür uymuyorsa ifade dengesiz demektir. Sonda yığın boş değilse kapanmamış parantez kalmıştır. "(a[b]c)"'de `]` gelince üstte `[`, `)` gelince üstte `(` vardır ve sonda yığın boştur: `true`. "(a[b)c]"'de `)` gelince üstte `[` vardır: `false`. "(("'de sonda yığında iki parantez kalır: `false`. Notun programı üç ek ifadeyi de dener ("()", boş dizgi ve "[(x)](y)"):

```
true false false false true true
```

Yığın kullanılmasının nedeni, son açılan parantezin ilk kapanması gerekmesidir. Bu tam olarak LIFO davranışıdır. Yığın `Stack` değil `ArrayDeque` olarak seçilmiştir. Metot ifadeyi bir kez tarar ve her karakter için O(1) iş yapar; toplam maliyet O(n)'dir.

Örnek 5.4

İki kod parçasının çıktısı nedir? (`push` = `offerFirst` , `pop` = `removeFirst` .)

```
Deque<Integer> d = new ArrayDeque<>();
d.offerLast(1);
d.offerFirst(2);
d.offerLast(3);
d.push(4);
System.out.println(d);
System.out.println(d.pollLast() + d.pop());
System.out.println(d);
```

```
Map<String, Integer> m = new TreeMap<>();
m.put("b", 1);
m.put("a", 2);
m.put("b", 3);
Set<Integer> s = new TreeSet<>(m.values());
s.add(2);
System.out.println(m + " " + s + " " + m.size());
```

```
[4, 2, 1, 3]
7
[2, 1]
{a=2, b=3} [2, 3] 2
```

Deque şöyle değişir: 1 → `[1]` ; başa 2 → `[2, 1]` ; sona 3 → `[2, 1, 3]` ; `push(4)` başa → `[4, 2, 1, 3]` . `pollLast()` 3'ü, `pop()` 4'ü döndürür. İki `Integer` `+` ile toplanır, çünkü ifadede dizgi yoktur: 7. Kalan deque `[2, 1]` 'dir. İkinci parçada `"b"` ikinci `put` ile 3 olur; eşleme `{a=2, b=3}` 'tür. Değerlerden kurulan küme `[2, 3]` 'tür. `s.add(2)` tekrar olduğu için eklenmez ve eşlemenin boyutu 2'dir.

Örnek 5.5

Bir `String[]` dizisinde en sık geçen kelimeyi döndüren `enSikKelime` metodu bir `HashMap` ile nasıl yazılır?

```
static String enSikKelime(String[] kelimeler) {
    Map<String, Integer> sayac = new HashMap<>();
    for (String k : kelimeler) {
        sayac.put(k, sayac.getOrDefault(k, 0) + 1);
    }
    String enIyi = null;
    int enFazla = 0;
    for (Map.Entry<String, Integer> g : sayac.entrySet()) {
        if (g.getValue() > enFazla) {
            enFazla = g.getValue();
            enIyi = g.getKey();
        }
    }
    return enIyi;
}
```

```
su
null
```

İlk döngü kelimeleri `getOrDefault` ile sayar. İkinci döngü `entrySet` üzerinde gezer ve en büyük sayıyı tutan kelimeyi saklar. “su, çay, su, kahve, su, çay” için su (3) döner. Boş dizide hiçbir kelime yoktur ve metot `null` döndürür; çağıran bu durumu denetlemelidir. Eşit sayıda geçen iki kelime varsa hangisinin döneceği `HashMap` 'in gezinme sırasına bağlıdır, yani belirsizdir. Maliyet O(n)'dir.

Örnek 5.6

Her öğrencinin not listesini tutan bir `Map<String, List<Integer>>` kurulur; Ali: 80, 90 ve Banu: 70 için ortalamalar nasıl yazdırılır?

```
Map<String, List<Integer>> notlar = new TreeMap<>();
notlar.put("Ali", new ArrayList<>());
notlar.get("Ali").add(80);
notlar.get("Ali").add(90);
notlar.put("Banu", new ArrayList<>(Arrays.asList(70)));
for (Map.Entry<String, List<Integer>> g : notlar.entrySet()) {
    int toplam = 0;
    for (int n : g.getValue()) {
        toplam += n;
    }
    System.out.println(g.getKey() + " ort: " + toplam / g.getValue().size());
}
```

```
Ali ort: 85
Banu ort: 70
{Ali=[80, 90]}
```

Eşlemenin değeri bir listedir. `notlar.get("Ali")` o listeyi döndürür ve `.add(80)` doğrudan o listeye ekler; eşlemeye yeniden `put` yapmak gerekmez. Ortalama tam sayı bölmesiyle hesaplanır: 170 / 2 = 85. Notun programının son satırı Java 8'deki `computeIfAbsent` kalıbını gösterir. Bu metot anahtar yoksa yeni bir liste oluşturup ekler, varsa var olanı döndürür; "önce `put` et, sonra `get` et" iki adımı tek çağrıya iner.

Sunumdaki ikinci alıştırma, loto çekilişini `shuffle` kullanmadan bir `Random` ve bir `TreeSet` ile yazar:

```
Random r = new Random();
Set<Integer> cekilen = new TreeSet<>();
while (cekilen.size() < 6) {
    cekilen.add(r.nextInt(49) + 1);  // 1 … 49
}
System.out.println(cekilen);
```

```
[1, 32, 37, 40, 43, 45]
```

Küme tekrarı kendisi engeller. Aynı sayı ikinci kez gelirse `add` onu eklemez ve döngü bir tur daha döner. `TreeSet` sayıları sıralı tutar. Çıktı her çalıştırmada farklıdır. Bu yöntemde döngünün kaç kez döneceği rastgeledir, ama 49 sayıdan 6'sı seçilirken tekrar olasılığı küçük olduğu için birkaç fazla turla biter.

5.11 Sık yapılan hatalar

Bu kesim, koleksiyonlarla çalışırken en sık görülen hataları gerçek derleyici ve çalışma anı iletileriyle birlikte toplar.

Sık yapılan hata

İlkel tiple jenerik tanım. `List<int>` derlenmez; tip argümanı bir referans tipi olmalıdır. Doğrusu `List<Integer>` 'dır; `int` değerler otomatik kutulanır.

```
IlkelTip.java:4: error: unexpected type
        List<int> notlar = new ArrayList<>();
             ^
  required: reference
  found:    int
1 error
```

Sık yapılan hata

Yanlış tipte öğe eklemek. `List<String>`'e `add(42)` derleme zamanında reddedilir. Sayı dizgi olarak saklanacaksa `String.valueOf(42)` eklenir.

```
YanlisTip.java:5: error: incompatible types: int cannot be converted to String
        isimler.add(42);
                    ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
1 error
```

Sık yapılan hata

`remove(int)` ile `remove(Object)`'u karıştırmak. `List<Integer>`'da `remove(1)` indis 1'i siler. Değeri silmek için `Integer.valueOf(…)` verilir.

```
List<Integer> liste = new ArrayList<>();
Collections.addAll(liste, 10, 20, 30);
liste.remove(1);                  // indeks 1
System.out.println(liste);
liste.remove(Integer.valueOf(10)); // değer 10
System.out.println(liste);
```

Sık yapılan hata

`Integer`'ları `==` ile karşılaştırmak. `==` iki referansın aynı nesneyi gösterip göstermediğine bakar. Java −128 ile 127 arasındaki `Integer` nesnelerini önbellekte tutar ve `valueOf` bu aralıkta hep aynı nesneyi döndürür. Bu yüzden 100 ile yapılan deneme tesadüfen `true`, 1000 ile yapılan `false` verir. Değer karşılaştırması `equals` ile yapılır.

```
List<Integer> a = new ArrayList<>();
Collections.addAll(a, 1000, 1000);
System.out.println(a.get(0) == a.get(1));
System.out.println(a.get(0).equals(a.get(1)));
```

```
[10, 30]
[30]
false
true
true
```

Çıktının ilk iki satırı `remove` örneğine, sonraki iki satırı 1000 ile yapılan karşılaştırmaya aittir. Son satır notun programına eklenen 100 denemesidir: aynı karşılaştırma önbellek aralığında `true` verir.

Sık yapılan hata

`Arrays.asList` **listesine eklemek.** `Arrays.asList` diziye bağlı, sabit boyutlu bir liste döndürür. `set` çalışır, `add` ve `remove` çalışmaz. Değiştirilebilir liste için `new ArrayList<>(Arrays.asList(…))` yazılır.

```
[9, 2, 3]
Exception in thread "main" java.lang.UnsupportedOperationException
```

Sık yapılan hata

Var olmayan indis. Geçerli indisler 0 … `size() - 1` 'dir. Üç öğeli listede `get(3)` istisna fırlatır. Boş kuyrukta `remove()` da `NoSuchElementException` fırlatır; `poll()` ise `null` döndürür.

```
List<String> l = new ArrayList<>();
Collections.addAll(l, "a", "b", "c");
System.out.println(l.get(3));
```

```
Exception in thread "main" java.lang.IndexOutOfBoundsException: Index 3 out of bounds for
length 3
```

Sık yapılan hata

Türkçe adları `String`'in doğal sırasıyla sıralamak. `String` karakter kodlarıyla sıralanır: Ç, Ö, Ş, Ü, İ, ı gibi harfler "Z"den sonra gelir. Türkçe alfabe sırasını `java.text.Collator` verir.

```
List<String> adlar = new ArrayList<>();
Collections.addAll(adlar,
        "Zeynep", "Çağla", "Can", "Ömer");
Collections.sort(adlar);
System.out.println(adlar);
Locale tr = Locale.forLanguageTag("tr-TR");
Collections.sort(adlar, Collator.getInstance(tr));
System.out.println(adlar);
```

```
[Can, Zeynep, Çağla, Ömer]
[Can, Çağla, Ömer, Zeynep]
[Ilgaz, irmik, ufuk, üzüm, İzmir, ılık] [Ilgaz, ılık, irmik, İzmir, ufuk, üzüm]
```

İlk iki satır sunumdaki çıktıdır. Notun programına eklenen üçüncü satır sorunun boyutunu gösterir. Doğal sırada büyük "I" küçük harflerden önce, "İ" ve "ı" ise en sona düşer. Türkçe `Collator` ile sıra Ilgaz, ılık, irmik, İzmir, ufuk, üzüm olur. Bu sıralayıcı büyük-küçük harf farkını ikincil sayar; "ı" ile "i" ayrı harflerdir.

5.12 Özet

- **Koleksiyon Çerçevesi** üç unsurdan oluşur: arayüzler (ne yapılır), gerçeklemeler (nasıl yapılır) ve `Collections` sınıfındaki algoritmalar. `Iterable` → `Collection` → `List` / `Queue` / `Set` hiyerarşisinin yanında `Map` ayrı bir daldır ve `Iterable` değildir.
- Değişken ve parametre **arayüz tipinde** tanımlanır; böylece gerçekleme yalnız `new` satırında değişir.
- **Jenerik tipler** yanlış tipi derleme zamanında yakalar. Tip argümanı referans tipidir; ilkel değerler **otomatik kutulama** ve **kutudan çıkarma** ile sarmalayıcılara dönüşür. `null`'ı kutudan çıkarmak `NullPointerException` fırlatır.
- `ArrayList` dinamik dizidir: `get` / `set` O(1), sona ekleme amortize O(1), ortaya ekleme ve silme O(n). Kapasite dolunca yaklaşık 1,5 katına büyür; kapasite `size()` değildir.
- `LinkedList` çift yönlü bağlı listedir: uçlarda O(1), `get(i)` O(n). Çoğu durumda `ArrayList` daha iyi seçimdir.
- `Vector` ve `Stack` eski ve senkronizedir. Yığın ve kuyruk için `ArrayDeque` kullanılır; `push` / `pop` / `peek` baş uçta, `offer` sonda, `poll` baştadır.

- `Queue` işlemlerinin iki biçimi vardır: `add` / `remove` / `element` istisna fırlatır, `offer` / `poll` / `peek` `false` / `null` döndürür.
- `PriorityQueue` ikili yığındır: `add` / `poll` O(log n), `peek` O(1). `println(pq)` iç diziyi yazar ve sıralı değildir; öğeler `Comparable` olmalı ya da bir `Comparator` verilmelidir.
- `HashSet` / `HashMap` hash tablosudur: kova `hashCode`'dan hesaplanır, kovada `equals` ile karşılaştırılır; ortalama O(1), sıra yoktur. `LinkedHash…` eklenme sırasını korur, `Tree…` kırmızı-siyah ağaçla sıralı tutar ve O(log n)'dir.
- Eşit nesnelerin hash kodları eşit olmalıdır; `equals`'ı geçersiz kılan sınıf `hashCode`'u da geçersiz kılar. Hash'li koleksiyondaki nesnenin bu alanları değiştirilmez.
- `TreeSet` / `TreeMap` aralık sorguları sunar; kural "başlangıç dahil, bitiş hariç"tir. `HashMap`'te `put` eski değeri döndürür, olmayan anahtar için `get` `null` döndürür; `Hashtable` `null` kabul etmez.
- **Yineleyici** for-each'in arkasındaki mekanizmadır. Gezinirken silme `it.remove()` ya da `removeIf` ile yapılır; listenin kendi `remove`'u `modCount`'u değiştirir ve `ConcurrentModificationException`'a yol açar.
- `Collections.sort` kararlı TimSort'tur (O(n log n)). Doğal sıra `Comparable.compareTo` ile, başka sıralar `Comparator` ile tanımlanır; karşılaştırmada çıkarma değil `Integer.compare` kullanılır. `shuffle` Fisher–Yates'tir; `binarySearch` sıralı listede O(log n) çalışır ve bulamazsa `-(ekleme noktası) - 1` döndürür.
- **Akış** bir işlem hattıdır. Hatta bir kaynak, tembel ara işlemler (`filter`, `map`, `sorted`) ve hattı çalıştıran uç işlem (`collect`, `reduce`, `count`) bulunur. Öğeler hattan tek tek geçer; akış tek kullanımlıktır; lambdalar yan etkisiz olmalıdır, özellikle paralel akışlarda.
- `List.of` / `Set.of` / `Map.of` değiştirilemez koleksiyon, `Collections.unmodifiableList` salt okunur görünüm, `List.copyOf` bağımsız değiştirilemez kopya üretir.

5.13 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

koleksiyon *collection* · Koleksiyon Çerçevesi *Collections Framework* · jenerikler *generics* · tip güvenliği *type safety* · sarmalayıcı sınıf *wrapper class* · otomatik kutulama / kutudan çıkarma *autoboxing / unboxing* · arayüz / gerçekleme *interface / implementation* · liste *list* · dinamik dizi *dynamic array* · bağlı liste *linked list* · indis *index* · yığın *stack* · kuyruk *queue* · öncelikli kuyruk *priority queue* · çift uçlu kuyruk *deque* · küme *set* · hash / hash tablosu *hashing / hash table* · hash kodu *hash code* · eşleme *map* · yineleyici *iterator* · doğal sıra *natural ordering* · karşılaştırıcı *comparator* · anonim sınıf *anonymous class* · lambda ifadesi *lambda expression* · karıştırma *shuffle* · ikili arama *binary search* · değiştirilemez *immutable* · iş parçacığı güvenliği *thread safety*

5.14 Kendini Dene

1. `List<Integer> l = new ArrayList<>(Arrays.asList(5, 6, 7));` listesine sırayla `l.add(0, 4)`, `l.remove(Integer.valueOf(6))`, `l.remove(2)` ve `l.add(l.size(), 8)` uygulanıyor. Liste ve `l.indexOf(8)` nedir?

   Cevap

   `[4, 5, 6, 7]` → değer 6 silinir: `[4, 5, 7]` → indis 2 (7) silinir: `[4, 5]` → sona 8: `[4, 5, 8]`. `indexOf(8)` 2'dir.

2. Boş bir `Deque<String> d = new ArrayDeque<>()` için sırayla `push("x")`, `offer("y")`, `push("z")`, `addLast("w")` çağrılıyor. Sonra `String a = d.pop(); String b = d.pollLast();` çalışıyor. `d` ve `a + b` nedir?

   Cevap

   `[x]` → `[x, y]` → `[z, x, y]` → `[z, x, y, w]`. `pop` baştaki z'yi, `pollLast` sondaki w'yi alır. `d` = `[x, y]`, `a + b` = `"zw"`.

3. `new PriorityQueue<Integer>(Collections.reverseOrder())` kuyruğuna 7, 2, 9, 4 ekleniyor. İki `poll()`'un sonucu ve kalan boyut nedir?

   Cevap

   Ters sırada en büyük önce çıkar: 9, sonra 7. Kuyrukta 2 öğe (2 ve 4) kalır.

4. `"kelebek"` dizgisinin karakterleri bir `TreeMap<Character, Integer>`'te `merge(c, 1, Integer::sum)` ile sayılıyor. Eşleme nasıl yazdırılır?

   Cevap

   `{b=1, e=3, k=2, l=1}`. Harfler k-e-l-e-b-e-k'tir: e üç, k iki, l ve b birer kez geçer; `TreeMap` anahtarları sıralı yazar.

5. `TreeSet` 15, 5, 25, 10, 20 öğelerini içeriyor. `headSet(15)`, `subSet(10, 21)` ve `tailSet(26)` nedir?

   Cevap

   `[5, 10]` (15 hariç), `[10, 15, 20]` (10 dahil, 21 hariç) ve `[]` (26 ve üstünde öğe yok).

6. `Arrays.asList("ayva", "kiraz", "erik", "dut", "armut")` listesinde `stream().filter(x -> x.length() > 3).map(x -> x.substring(0, 2)).sorted().collect(Collectors.toList())` ne üretir? `mapToInt(String::length).sum()` kaçtır?

   Cevap

   "dut" elenir; kalanların ilk iki harfi ay, ki, er, ar'dır ve sıralanınca `[ar, ay, er, ki]` olur. Uzunluk toplamı 4 + 5 + 4 + 3 + 5 = 21'dir.

7. `[2, 3, 5, 8]` listesinde for-each içinde `if (x == 5) sayilar.remove(x);` çalışıyor. İstisna fırlatılır mı? Liste ne olur?

   Cevap

   Fırlatılmaz. 5 (indis 2) silinince `size` 3 olur; `cursor` zaten 3'tür ve `hasNext()` `false` döner. Döngü sessizce biter ve 8 hiç denetlenmez. Liste `[2, 3, 8]`'dir. Kod hatalıdır, ama bu veriyle bunu belli etmez.

8. `Collections.binarySearch(Arrays.asList(2, 4, 6, 8, 10), 9)` ne döndürür?

   Cevap

   9 yoktur; 8'den sonra, 10'dan önce, yani indis 4'e girmelidir. Dönüş −(4) − 1 = −5'tir.

9. Bir `Ogrenci` sınıfı yalnızca `equals`'ı geçersiz kılmıştır. Aynı alanlı iki nesne hem bir `ArrayList`'e hem bir `HashSet`'e ekleniyor ve ikinci nesneyle `contains` soruluyor. Sonuçlar neden farklıdır?

   Cevap

   `ArrayList.contains` yalnızca `equals` kullanır ve `true` döner. `HashSet` ise önce `hashCode` ile kovayı bulur. `Object`'ten gelen hash kodları farklı olduğu için iki nesne farklı kovalara düşer; küme iki öğe tutar ve `contains` `false` döner. Çözüm `hashCode`'u aynı alanlardan yazmaktır.

5.15 İleri okuma

- Oracle, *The Java Tutorials*, "Collections" ve "Aggregate Operations" bölümleri: arayüzleri, gerçeklemeleri, algoritmaları ve akış hatlarını resmî örneklerle adım adım anlatır.
- J. Bloch, *Effective Java* (3. baskı, Addison-Wesley, 2018), Madde 11, 14, 45–48 ve 64. `hashCode`, `Comparable`, akışların ölçülü ve yan etkisiz kullanımı ve nesnelere arayüzleriyle başvurma üzerine kısa, gerekçeli kurallar verir.
- M. Naftalin ve P. Wadler, *Java Generics and Collections* (O'Reilly): jeneriklerin ve koleksiyon sınıflarının iç yapısını ve tasarım gerekçelerini ayrıntılı inceler.

- C. S. Horstmann, *Core Java, Volume I: Fundamentals* (12. baskı, 2022), Bölüm 9: koleksiyonları hash tablosu, ağaç ve görünümlerin işleyişiyle birlikte anlatır.

5.16 Kaynaklar

- Oracle, *The Java Tutorials*: Collections (Interfaces, Implementations, Algorithms) ve Generics.
- Oracle, *Java SE 21 API Documentation*: `java.util` paketi (`Collection`, `List`, `Map`, `Collections` …) ve `java.util.stream`.
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21 Edition (4.5 Parameterized Types, 5.1.7 Boxing Conversion, 14.14.2 Enhanced for).
- P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (Bölüm 16: Generic Collections).
- K. Sierra ve B. Bates, *Head First Java*, 3. baskı, O'Reilly (Bölüm 11: Data Structures).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson (Bölüm 9: Collections).
- M. Naftalin ve P. Wadler, *Java Generics and Collections*, O'Reilly.
- J. Bloch, *Effective Java*, 3. baskı, Addison-Wesley (Madde 11: `equals` ile birlikte `hashCode`; Madde 64: nesnelere arayüzleriyle başvurmak).

KISIM III

Sağlam Program

Bölüm 6–7

KISIM III · BÖLÜM 6

İstisnalar

Bir program, yazıldığı gibi çalışsa bile dış dünyanın sürprizleriyle karşılaşır. Okunacak dosya silinmiş olabilir, kullanıcı sayı yerine harf girebilir, ağ bağlantısı kopabilir. Programın kendi içinde de hatalar olur: sıfıra bölme, dizinin dışına taşma, `null` bir referansın kullanılması. Java bu durumların hepsini tek bir düzenekle ele alır: istisna. Not önce istisnanın ne olduğunu ve yakalanmadığında programa ne olduğunu gösterir. Ardından `try`, `catch` ve `finally` bloklarını, kaynakların otomatik kapatılmasını ve iç içe blokları adım adım izler. İstisna sınıflarının hiyerarşisi, kontrol edilen ve kontrol edilmeyen istisnalar, `throw` ve `throws` anahtar sözcükleri notun merkezindedir. Yaygın istisna sınıfları tek tek örneklenir. Not özel istisna sınıfları yazmak, yığın izini okumak ve iyi uygulamalarla biter.

Bu bölümün kodu: `github.com/sercankulcu/object-oriented-programming-java/Bolum06/src`

ÖĞRENME ÇIKTILARI

Kavram
İstisnayı tanımlamak; çalışma zamanı sorunlarını `Exception` ve `Error` olarak ayırt etmek; dönüş kodu yerine istisna kullanmanın gerekçesini açıklamak.

Yakalama
`try`, `catch`, `finally` ve try-with-resources ile istisnaları ele almak; akışı adım adım izlemek.

Hiyerarşi
`Throwable`, `Error`, `Exception` ve `RuntimeException` arasındaki ilişkiyi ve `catch` sırasının önemini açıklamak.

Türler
Kontrol edilen ve kontrol edilmeyen istisnaları ayırmak; "yakala ya da bildir" kuralını uygulamak.

Fırlatma
`throw` ile istisna fırlatmak, `throws` ile bildirmek; istisnanın çağrı yığınındaki yayılımını izlemek.

Uygulama
Yaygın istisna sınıflarını tanımak, özel istisna sınıfı yazmak ve yığın izini okuyarak hatanın kaynağını bulmak.

6.1 İstisna kavramı

Programlama hataları üç gruba ayrılır. Derleme hatası, kodun dilin kurallarına uymamasıdır: noktalı virgül eksiktir ya da bir değişken tanımlanmamıştır. Derleyici bu hatayı bulur ve program hiç çalışmaz. Mantık hatası ise sessizdir: program çalışır, ama yanlış sonuç üretir. Ortalamayı hesaplarken toplamı yanlış sayıya bölmek buna örnektir. Üçüncü grup, program çalışırken ortaya çıkan sorunlardır. Kod derlenmiştir, ama belirli bir anda belirli bir değerle yürütülemez. Bu notun konusu bu üçüncü gruptur.

6.1.1 İstisna nedir?

> **Tanım 6.1**
>
> **İstisna** (*exception*), program çalışırken ortaya çıkan ve programın normal akışını bozan beklenmedik durumdur. Java'da istisna bir nesnedir. Bu nesne durumun tipini, açıklayıcı bir mesajı ve sorunun oluştuğu yeri taşır.

Gündelik örnekler hemen akla gelir. Açılmak istenen dosya bulunamaz. Bir tam sayı sıfıra bölünür. Üç öğeli bir dizinin beşinci öğesine erişilmek istenir. `null` değerli bir referans üzerinden metot çağrılır. Bu durumların hiçbiri derleyicinin önceden göreceği türden değildir. Dosyanın var olup olmadığı, paydanın sıfır olup olmadığı ancak program çalışırken belli olur.

İstisnayı iki taraf oluşturabilir. Birincisi Java sanal makinesidir (*Java Virtual Machine*, JVM). Sıfıra bölme ya da dizi sınırının aşılması gibi durumları JVM kendisi fark eder ve istisna nesnesini kendiliğinden üretir. İkincisi programcının kendisidir. Programcı, örneğin geçersiz bir argüman gördüğünde `throw` anahtar sözcüğüyle bilerek bir istisna oluşturur. Her iki durumda da sonuç aynıdır: ortada bir istisna nesnesi vardır ve normal akış durmuştur.

İstisna düzeneği programcıya olağan dışı durumlarla düzenli biçimde başa çıkma olanağı verir. Kabaca söylersek, Java bir istisnayla karşılaştığında şöyle der: “Pes ettim, şu an ne yapacağımı bilmiyorum. Bunu biri halletsin!” Sorunu çözecek kod, sorunun çıktığı yerde olmak zorunda değildir.

Bir lokantayı düşünelim. Garson mutfağa gider ve istenen yemeğin bittiğini öğrenir. Garson bu sorunu kendi başına çözemez; durumu yöneticiye bildirir. Yönetici ya sorunu çözer ve müşteriye başka bir yemek önerir ya da durumu bir üst kademeye iletir. İstisna da metotlar arasında böyle yol alır. Sorunu fark eden metot onu çözemiyorsa kendisini çağıran metoda bildirir. Bu zincir, sorunu çözebilecek bir metoda ulaşana kadar sürer.

6.1.2 Çalışma zamanı hataları: istisna ve Error

Program çalışırken ortaya çıkan sorunu JVM bir `Throwable` nesnesi olarak fırlatır. Bu sınıfın adı “fırlatılabilir” anlamına gelir. `Throwable` 'ın iki ana kolu vardır ve iki kol çok farklı durumları anlatır.

Kol	Ne anlatır?	Program ne yapabilir?	Örnek
İstisna (`Exception`)	Programın öngörebileceği, olağan dışı bir durum	Yakalayıp toparlanabilir	`10 / 0`, üç öğeli dizide `dizi[5]`, olmayan dosya
Hata (`Error`)	JVM ya da sistem düzeyinde ciddi bir sorun	Genellikle yakalanmaz; program sonlanır	`StackOverflowError`, `OutOfMemoryError`

Tablo 6.1: Çalışma zamanı sorunlarının iki ana kolu.

Tablodaki ayrımın ölçütü "program bu durumdan kurtulabilir mi?" sorusudur. Dosya bulunamadıysa program kullanıcıdan başka bir dosya adı isteyebilir. Kullanıcı sayı yerine harf girdiyse program uyarı verip yeniden sorabilir. Bunlar istisnadır. Bellek tükendiyse ya da çağrı yığını taştıysa programın elinde düzeltme yapacak kaynak kalmamıştır. Bunlar `Error` sınıfına girer.

İstisna düzeneği ve istisnanın bıraktığı yığın izi, çalışma zamanı hatalarının nedenini bulup düzeltmenin, yani hata ayıklamanın (*debugging*) en güçlü araçlarındandır. Derleme hatasını derleyici gösterir; mantık hatası iz bırakmaz. Çalışma zamanı hatasında ise JVM sorunun tipini, mesajını ve tam yerini yazdırır.

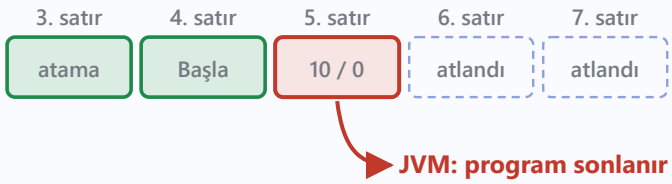
6.1.3 Yakalanmayan istisna

Önce hiçbir önlem alınmadığında ne olduğuna bakalım. Aşağıdaki program bir tam sayıyı sıfıra böler.

```
public class Cokus {
  public static void main(String[] args) {
    int pay = 10, payda = 0;
    System.out.println("Başla");
    int sonuc = pay / payda;
    System.out.println("Sonuç: " + sonuc);
    System.out.println("Bitti");
  }
}
```

```
Başla
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Cokus.main(Cokus.java:5)
```

Program önce `Başla` yazar. 5. satırda `pay / payda` ifadesi hesaplanırken payda sıfırdır. Tam sayı bölmesinde sıfıra bölmenin bir sonucu yoktur. JVM bir `ArithmeticException` nesnesi oluşturur ve onu fırlatır. Bu satırda istisnayı yakalayacak bir kod yoktur. `main` metodunu çağıran da JVM'in kendisidir. Bu yüzden istisna JVM'e kadar ulaşır ve program sonlanır. `Sonuç` ve `Bitti` satırları hiç yazılmaz.



Şekil 6.1: Yakalanmayan istisnada akış. 3. ve 4. satırlar çalışır; 5. satırda istisna oluşur ve JVM programı sonlandırır. 6. ve 7. satırlar atlanır.

JVM'in yazdırdığı ileti üç bilgi taşır. `Exception in thread "main"` kısmı, istisnanın `main` iş parçacığında yakalanmadığını söyler. Ardından istisnanın tam sınıf adı gelir: `java.lang.ArithmeticException`. İki noktadan sonraki `/ by zero` istisnanın mesajıdır. Son satır sorunun yerini verir: `Cokus` sınıfının `main` metodu, `Cokus.java` dosyasının 5. satırı. `at` ile başlayan bu satırlara yığın izi denir.

> **Sık yapılan hata**
>
> Programın "çöktüğünü" görünce derleme hatası sanmak. Bu program sorunsuz derlenir. Derleyici `payda` değişkeninin çalışma sırasında hangi değeri alacağını bilmez. Sorun yalnız çalıştırınca ortaya çıkar. İletideki `Exception in thread` ifadesi bunun çalışma zamanı sorunu olduğunu gösterir.

6.1.4 Ele almak ya da bildirmek

Bir metotta istisna oluşabiliyorsa metodun iki seçeneği vardır.

Birinci yol, istisnayı ele almaktır (*handle*). Metot, istisna oluşturabilecek kodu bir `try` bloğuna alır ve `catch` bloğunda durumu kendisi çözer. Örneğin ayar dosyasını okuyan bir metot düşünelim. Dosya bulunamazsa (`FileNotFoundException`) metot kullanıcıya bir mesaj gösterir ve varsayılan ayarlarla devam eder.

İkinci yol, istisnayı çağıranın sorunu yapmaktır (*declare*). Metot istisnayı ele almaz; imzasında `throws` anahtar sözcüğüyle bu istisnayı fırlatabileceğini bildirir. Böylece istisna, metodu çağıran kodun sorunu olur. Çağıran kod da aynı seçimle karşılaşır: ya istisnayı ele alır ya da bir üst metoda iletir.

Hangi yolun seçileceğine şu soru karar verir: istisnayı anlamlı biçimde çözebilecek yer neresidir? Dosya adını kullanıcıdan alan kod, "dosya yok" durumunda kullanıcıya yeniden sorabilir. Dosyayı satır satır okuyan alt düzey bir metot ise kullanıcıyla konuşmaz; dosya adının nereden geldiğini de bilmez. Bu metot için doğru seçim istisnayı bildirmek, kullanıcı arayüzü kodu için doğru seçim ise onu ele almaktır.

6.1.5 Dönüş kodu yerine istisna

İstisnalar, bir programın "bir şeyin ters gittiğini" bildirmesi için kullanılır. Ama "ters gitmek" özneldir. Bir dizide aranan ismin bulunamaması bir hata mıdır, yoksa olağan bir sonuç mu? İstisna düzeneği yaygınlaşmadan önce bu tür durumlar özel bir dönüş değeriyle bildirilirdi. Aşağıdaki metot, aranan ismi bulamazsa `-1` döndürür.

```
public static int indexOf(String[] isimler, String isim) {
  for (int i = 0; i < isimler.length; i++) {
    if (isimler[i].equals(isim)) {
      return i;
    }
  }
  return -1;
}
// main içinde
String[] isimler = {"Ali", "Ayşe", "Can"};
System.out.println(indexOf(isimler, "Can"));
int k = indexOf(isimler, "Ece");   // k = -1
System.out.println(isimler[k]);
```

```
2
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index -1 out of
bounds for length 3
    at DonusKodu.main(DonusKodu.java:14)
```

İlk çağrı `"Can"` ismini 2. indiste bulur ve `2` yazdırılır. İkinci çağrıda `"Ece"` dizide yoktur ve metot `-1` döndürür. Çağıran kod bu değeri denetlemeyi unutmuştur. `isimler[-1]` erişimi bir `ArrayIndexOutOfBoundsException` fırlatır. Asıl sorun aramada çıkmıştır, ama hata başka bir yerde, dizi erişiminde görünür. Hatanın kaynağıyla belirtisinin ayrı yerlerde olması hata ayıklamayı zorlaştırır.

Dönüş kodunun bir sorunu daha vardır: hata değeri olağan sonuçlarla karışabilir. `-1` burada geçersiz bir indistir, bu yüzden işe yarar. Ama sıcaklık ölçen bir metot için `-1` geçerli bir değerdir. Aynı metodun bir hatayı bildirmek için kullanabileceği boş bir değer kalmaz.

Aynı metot istisnayla şöyle yazılır.

```
public static int bul(String[] isimler, String isim) {
  for (int i = 0; i < isimler.length; i++) {
    if (isimler[i].equals(isim)) {
      return i;
    }
  }
  throw new IllegalArgumentException("Bulunamadı: " + isim);
}
// main içinde
try {
  System.out.println(bul(isimler, "Ece"));
} catch (IllegalArgumentException e) {
  System.out.println("Hata: " + e.getMessage());
}
```

```
Hata: Bulunamadı: Ece
```

Eşleşme bulunamazsa metot bir `IllegalArgumentException` fırlatır. Mesaj, aranan ismi de içerir: `Bulunamadı: Ece`. Artık çağıranın denetlemeyi unutması mümkün değildir. İstisna yakalanmazsa program o noktada durur ve sorunun yerini açıkça gösterir. Hata, normal sonuçla karışmaz: `bul` metodu bir sayı döndürdüyse bu sayı her zaman geçerli bir indistir.

Bu nedenle `-1` gibi özel dönüş kodlarından genellikle kaçınılır ve Java'nın istisna düzeneği tercih edilir. İstisnaların yararları üç maddede toplanır:

- Hata işleme açık ve tutarlıdır; hata değeri normal sonuçla karışmaz.
- Kod daha okunaklı olur, hata ayıklama kolaylaşır: hata doğduğu yerde görünür.
- Kodun güvenilirliği artar: istisnalar, denetlenmeyen dönüş kodları gibi sessizce gözden kaçmaz.

Sık yapılan hata

Her olumsuz sonucu istisnaya çevirmek. Bir arama metodunun “bulunamadı” sonucu, programın olağan akışının bir parçası olabilir. Java kütüphanesindeki `String.indexOf` bu yüzden hâlâ `-1` döndürür. İstisna, çağıranın beklemediği ve sıradan bir değerle anlatılamayan durumlar içindir. Karar, metodun sözleşmesine göre verilir ve belgelenir.

6.2 `try` ve `catch`

6.2.1 `try` ve `catch` deyimleri

Java'da hata yönetiminin temel yapısı `try` - `catch` deyimidir. Hata çıkarabilecek kod `try` bloğuna yazılır. Hata çıkarsa ne yapılacağı ise `catch` bloğuna yazılır.

> **Tanım 6.2**
>
> `try` **bloğu**, istisna oluşturabilecek deyimleri içerir. `catch` **bloğu**, `try` bloğunda belirtilen tipte bir istisna oluştuğunda çalışır ve onu ele alır; buna istisnayı *yakalamak* denir. `catch`'in parantezindeki `(ArithmeticException e)` bir parametre bildirimidir: `e`, fırlatılan istisna nesnesini gösteren bir referanstır.

```
try {
  // Potansiyel hata içeren kodlar
  int sonuc = 10 / 0; // Hata: ArithmeticException
} catch (ArithmeticException e) {
  // Hata durumuyla başa çıkma
  System.out.println("hata oluştu: " + e.getMessage());
}
```

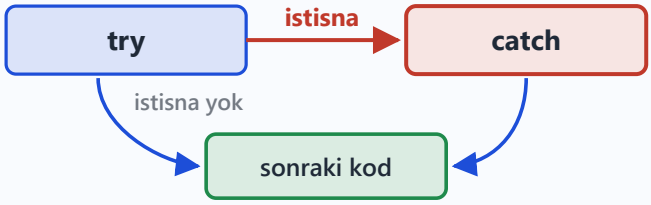
```
hata oluştu: / by zero
```

`10 / 0` ifadesi bir `ArithmeticException` fırlatır. Fırlatılan nesnenin tipi `catch`'te yazılan tiple eşleşir. JVM `e` parametresine bu nesneyi bağlar ve `catch` bloğunu çalıştırır. `e.getMessage()` nesnenin mesajını, `/ by zero` metnini döndürür.

Bu yapının akışı iki kuralla özetlenir:

- İstisna oluşursa `try` bloğunun kalan satırları atlanır ve denetim eşleşen `catch` bloğuna geçer.
- İstisna oluşmazsa `catch` bloğu hiç çalışmaz.

Her iki durumda da `catch` bloğundan sonra program normal akışına devam eder. İstisna yakalandığı için program çökmez.



Şekil 6.2: `try` - `catch` akışının iki yolu. İstisna yoksa `try` bitince doğrudan sonraki koda geçilir. İstisna varsa önce `catch` çalışır, sonra sonraki koda geçilir.

Burada kullanılan kod, istisnayı göstermek için yazılmış yapay bir örnektir. Değişmez `10 / 0` ifadesinde derleyici de sorunu görür ve bir uyarı verir (`division by zero`). Gerçek programlarda payda çoğunlukla bir değişkendir ve değeri ancak çalışırken belli olur.

6.2.2 Kod izleme: `try` - `catch` akışı

Yakalanmayan istisna örneğindeki program bu kez bir `try` - `catch` içine alınmıştır.

```
public class BolmeOrnegi {
  public static void main(String[] args) {
    int pay = 10, payda = 0;
    System.out.println("Başla");
    try {
      System.out.println("Bölme öncesi");
      int sonuc = pay / payda;
      System.out.println("Sonuç: " + sonuc);
    } catch (ArithmeticException e) {
      System.out.println("Hata: " + e.getMessage());
    }
    System.out.println("Bitti");
  }
}
```

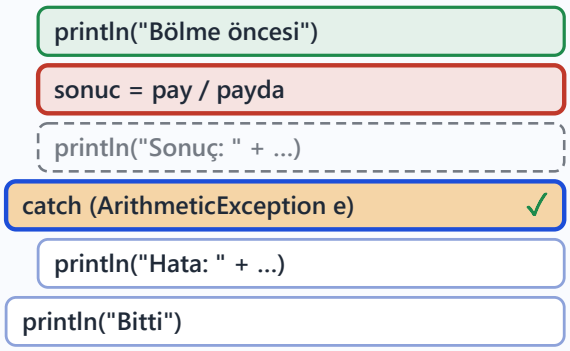
Program satır satır ilerler. Tablo 6.2, sunumdaki adım adım izlemenin her adımındaki satırı, değişkenleri ve o adımda eklenen çıktıyı verir.

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	3: `int pay = 10, payda = 0;`	main()	pay=10, payda=0	—
2	4: `System.out.println("Başla");`	main()	pay=10, payda=0	`Başla`
3	6: `System.out.println("Bölme öncesi");`	main()	pay=10, payda=0	`Bölme öncesi`
4	7: `int sonuc = pay / payda;`	main()	pay=10, payda=0	—
5	9: `} catch (ArithmeticException e) {`	main()	pay=10, payda=0, e=ArithmeticException	—

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
6	10: `System.out.println("Hata: " + e.getMessage());`	main()	pay=10, payda=0, e=ArithmeticException	`Hata: / by zero`
7	12: `System.out.println("Bitti");`	main()	pay=10, payda=0	`Bitti`

Tablo 6.2: `BolmeOrnegi` programının izlemesi.

İlk iki adımda istisnayla ilgili bir şey yoktur. Değişkenler oluşur ve `try` dışındaki ilk satır `Başla` yazar. 3. adımda `try` bloğuna girilir; ilk satır sorunsuz çalışır ve `Bölme öncesi` yazılır. 4. adım işin kalbidir. `pay / payda` hesaplanırken JVM bir `ArithmeticException` nesnesi oluşturur ve fırlatır. Atama yapılmaz; `sonuc` değişkeni hiç değer almaz.



Şekil 6.3: İzlemenin 5. adımı. `try`'ın ilk satırı başarıyla çalıştı (yeşil), ikinci satır istisna fırlattı (kırmızı), üçüncü satır atlandı (kesik çizgi). Tip eşleştiği için `catch` bloğu çalışıyor.

5. adımda JVM uygun bir `catch` arar. Fırlatılan nesne bir `ArithmeticException`'dır ve `catch`'te de bu tip yazılıdır. Tip eşleşir, `e` istisna nesnesini gösterir. `Sonuç` satırı atlanmıştır. 6. adımda `e.getMessage()` çağrılır ve `Hata: / by zero` yazılır. 7. adımda `catch` bloğu biter ve program normal akışına döner: `Bitti` yazılır.

İki programı karşılaştırmak öğreticidir. `try`-`catch` olmadan program 5. satırda sonlanıyor ve `Bitti` yazılmıyordu. Şimdi aynı istisna oluşuyor, ama program sonuna kadar çalışıyor. İstisna yok olmadı; yalnız ele alındı.

Sık yapılan hata

`try` içinde tanımlanan değişkeni bloğun dışında kullanmak. `int sonuc` yukarıdaki programda `try` bloğunda tanımlıdır; kapsamı bloğun kapanış parantezinde biter. `try`'dan sonra `System.out.println(sonuc)` yazılırsa derleyici `cannot find symbol` hatası verir. Değişken dışarıda kullanılacaksa `try`'dan önce tanımlanır ve bir ilk değer verilir.

6.2.3 İstisna nesnesinden bilgi almak

`catch` parametresi sıradan bir referanstır. İstisna nesnesi de sıradan bir nesnedir ve metotlarını `Throwable` sınıfından alır. Aşağıdaki kod bu metotların ne verdiğini gösterir.

```
try {
  int[] dizi = new int[3];
  dizi[5] = 7;
} catch (ArrayIndexOutOfBoundsException e) {
  System.out.println(e.getMessage());
  System.out.println(e);
  System.out.println(e.getClass().getSimpleName());
  e.printStackTrace();
}
```

```
Index 5 out of bounds for length 3
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
ArrayIndexOutOfBoundsException
java.lang.ArrayIndexOutOfBoundsException: Index 5 out of bounds for length 3
    at IstisnaNesnesi.main(IstisnaNesnesi.java:5)
```

Metot	Ne verir?
`getMessage()`	Yalnız açıklama mesajı; mesaj verilmemişse `null`
`toString()`	Tam sınıf adı, ": " ve mesaj; `println(e)` bunu kullanır
`getClass()`	İstisnanın gerçek sınıfı; `getSimpleName()` ile paketsiz adı
`printStackTrace()`	`toString()` ve yığın izi; `System.err`'e yazar
`getCause()`	Bu istisnaya yol açan başka istisna; yoksa `null`

Tablo 6.3: İstisna nesnesinden bilgi alan metotlar.

Çıktının ilk satırı `getMessage()`'dan gelir: `Index 5 out of bounds for length 3`. İkinci satırı `println(e)` yazar. `println`, nesnenin `toString()` metodunu çağırır ve sınıfın tam adıyla mesajı birleştirir. Üçüncü satır yalnız sınıfın kısa adıdır. Son iki satırı `printStackTrace()` üretir. Bu metot önce `toString()` sonucunu, ardından yığın izini yazar.

`printStackTrace()` standart çıktıya değil, hata çıktısına (`System.err`) yazar. İki akış aynı konsolda görünse de ayrıdır. Bu yüzden iki akışa birlikte yazılan satırların sırası konsolda karışabilir. Çoğu geliştirme ortamı (*IDE*) hata akışını kırmızı renkte gösterir.

`getCause()` zincirlenmiş istisnalar içindir. Bir metot, alt düzeyde yakaladığı bir istisnayı kendi tipine çevirip yeniden fırlatabilir. Bunu yaparken eski istisnayı "neden" olarak yeni nesneye ekler: `new RuntimeException("Ayarlar okunamadı", e)`. Böylece üst düzeydeki kod, sorunun asıl kaynağına `getCause()` ile ulaşabilir. Bu örnekte istisnanın bir nedeni yoktur ve metot `null` döndürür.

Sık yapılan hata

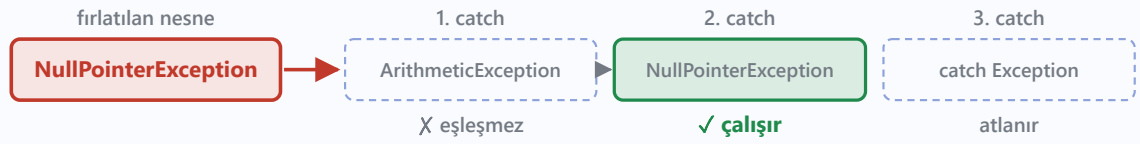
Kullanıcıya gösterilecek mesajda `getMessage()` 'ın her zaman dolu olduğunu varsaymak. `new Exception()` gibi mesajsız oluşturulan istisnalarda `getMessage()` `null` döndürür. Kayıt tutarken `e.toString()` ya da `println(e)` daha güvenlidir: en azından sınıf adı her zaman yazılır.

6.2.4 Birden çok `catch` bloğu ve sıra

Bir kod bloğunda birden çok tipte istisna oluşabilir. Kullanıcıdan alınan iki sayıyı bölen bir program düşünelim. Girdi sayı olmayabilir, payda sıfır olabilir, girdi eksik olabilir. Her hata türü için ayrı bir `catch` bloğu yazılır ve her blok o tipe özgü tepkiyi içerir.

```
try {
  // Potansiyel hata içeren kodlar
} catch (ArithmeticException e) {
  // ArithmeticException ile başa çıkma
} catch (NullPointerException e) {
  // NullPointerException ile başa çıkma
} catch (Exception e) {
  // Diğer hata türleri için genel durum
}
```

İstisna fırlatıldığında `catch` blokları yukarıdan aşağıya sırayla denenir. Eşleşen ilk blok çalışır, diğerleri atlanır. Burada "eşleşmek", fırlatılan nesnenin `catch` 'teki tipin kendisi ya da bir alt sınıfı olması demektir. Bu, `instanceof` işlecinin mantığıdır. Son sıradaki `catch (Exception e)` geri kalan bütün istisnaları yakalayan genel durumdur.



Şekil 6.4: Fırlatılan bir `NullPointerException` için `catch` bloklarının denenmesi. Birinci blok eşleşmez, ikinci blok çalışır, üçüncü blok hiç denenmez.

Sıra, `catch` tipleri arasında bir kalıtım ilişkisi olduğunda önem kazanır. Aşağıdaki kodda genel tip özel tipten önce yazılmıştır.

```
try {
  int sonuc = 10 / 0;
} catch (Exception e) {
  System.out.println("Genel hata");
} catch (ArithmeticException e) {
  System.out.println("Aritmetik hata");
}
```

```
SiraHatasi.java:7: error: exception ArithmeticException has already been caught
    } catch (ArithmeticException e) {
```

`ArithmeticException` bir `Exception`'dır. Fırlatılan her `ArithmeticException`'ı birinci blok zaten yakalar. İkinci blok hiçbir durumda çalışamaz. Java derleyicisi erişilemeyen bir `catch` bloğuna izin vermez ve hata verir. Kural basittir: alt sınıflar üst sınıflardan önce gelir. Doğru sıralama şöyledir.

```
} catch (ArithmeticException e) {
  System.out.println("Aritmetik hata");
} catch (Exception e) {
  System.out.println("Genel hata");
}
```

Kalıtım ilişkisi olmayan tiplerde sıra önemli değildir. `ArithmeticException` ile `NullPointerException` birbirinin alt sınıfı değildir. Bir nesne aynı anda ikisi birden olamaz; bu yüzden iki blok hangi sırayla yazılırsa yazılsın en fazla biri eşleşir.

Örnek 6.1

`NumberFormatException`, `IllegalArgumentException`'ın alt sınıfıdır. Önce `catch (IllegalArgumentException e)`, sonra `catch (NumberFormatException e)` yazılırsa ne olur?

Derleme hatası oluşur: `exception NumberFormatException has already been caught`. Her `NumberFormatException` bir `IllegalArgumentException` olduğu için birinci blok onu zaten yakalar. İki bloğun yeri değiştirilirse kod derlenir. Önce özel durum ("sayı değil"), sonra genel durum ("geçersiz argüman") ele alınır.

6.2.5 Kod izleme: hangi `catch` çalışır?

Aşağıdaki metot bir metin dizisinden iki sayı okur ve birini ötekine böler. Üç ayrı `catch` bloğu ve bir `finally` bloğu vardır. `finally` bloğu, istisna olsun olmasın her durumda çalışan bloktur; burada her çağrının sonunda bir satır yazar.

```
static void bol(String[] girdi) {
  try {
    int x = Integer.parseInt(girdi[0]);
    int y = Integer.parseInt(girdi[1]);
    System.out.println("Sonuç: " + (x / y));
  } catch (NumberFormatException e) {
    System.out.println("Sayı değil: " + e.getMessage());
  } catch (ArithmeticException e) {
    System.out.println("Sıfır: " + e.getMessage());
  } catch (ArrayIndexOutOfBoundsException e) {
    System.out.println("Eksik: " + e.getMessage());
  } finally {
    System.out.println("finally çalıştı");
  }
}
// main içinde
bol(new String[] {"10", "abc"});
bol(new String[] {"10", "0"});
bol(new String[] {"10"});
```

```
Sayı değil: For input string: "abc"
finally çalıştı
Sıfır: / by zero
finally çalıştı
Eksik: Index 1 out of bounds for length 1
finally çalıştı
```

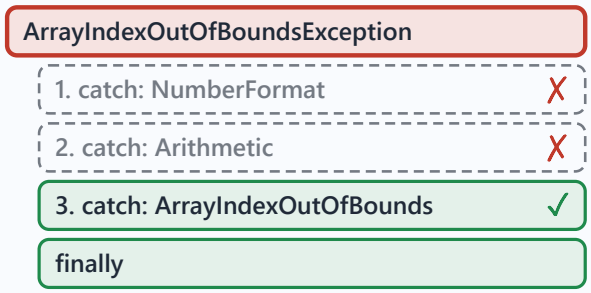
Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	4: `int y = Integer.parseInt(girdi[1]);`	main() → bol(girdi)	girdi=["10", "abc"], x=10, y=—	—
2	6: `} catch (NumberFormatException e) {` 7: `System.out.println("Sayı değil: " + e.getMessage());`	main() → bol(girdi)	girdi=["10", "abc"], x=10, y=—	`Sayı değil: For input string: "abc"`
3	12: `} finally {`	main() → bol(girdi)	girdi=["10", "abc"], x=10, y=—	`finally çalıştı`
4	5: `System.out.println("Sonuç: " + (x / y));`	main() → bol(girdi)	girdi=["10", "0"], x=10, y=0	—
5	8: `} catch (ArithmeticException e) {` 9: `System.out.println("Sıfır: " + e.getMessage());`	main() → bol(girdi)	girdi=["10", "0"], x=10, y=0	`Sıfır: / by zero`
6	4: `int y = Integer.parseInt(girdi[1]);`	main() → bol(girdi)	girdi=["10"], x=10, y=—	`finally çalıştı`

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
7	10: `} catch (ArrayIndexOutOfBoundsException e) {` 11: `System.out.println("Eksik: " + e.getMessage());` 12: `} finally {`	main() → bol(girdi)	girdi=["10"], x=10, y=—	`Eksik: Index 1 out of bounds for length 1` ⏎ `finally çalıştı`

Tablo 6.4: Üç çağrıda hangi `catch`'in çalıştığı.

Birinci çağrıda `girdi` dizisi `{"10", "abc"}`'dir. `x` 10 olur. `Integer.parseInt("abc")` ise bir `NumberFormatException` fırlatır. Bölme satırı atlanır. JVM `catch` bloklarını sırayla dener. İlk blok `NumberFormatException` içindir ve eşleşir. Diğer bloklara bakılmaz bile. Ardından `finally` çalışır.

İkinci çağrıda iki sayı da geçerlidir: `x = 10`, `y = 0`. Bölme `ArithmeticException` fırlatır. Birinci `catch` eşleşmez, ikinci `catch` eşleşir. Mesaj yine `/ by zero`'dur.



Şekil 6.5: Üçüncü çağrıda fırlatılan `ArrayIndexOutOfBoundsException`. İlk iki blok eşleşmez, üçüncüsü eşleşir; ardından `finally` çalışır.

Üçüncü çağrıda dizinin tek öğesi vardır. `girdi[1]` erişimi bir `ArrayIndexOutOfBoundsException` fırlatır. Mesaj, hangi indisin hangi uzunluktaki dizide geçersiz olduğunu söyler: `Index 1 out of bounds for length 1`. İlk iki `catch` eşleşmez, üçüncüsü eşleşir; sonra `finally` çalışır.

Üç çağrıda üç farklı istisna oluştu ve her birini ayrı bir blok karşıladı. İstisna oluşmasaydı ne olurdu? `bol(new String[] {"10", "4"})` çağrısında hiçbir `catch` çalışmaz. Metot `Sonuç: 2` ve `finally çalıştı` yazar. Tam sayı bölmesinde 10 / 4 = 2'dir.

6.2.6 Tek `catch`, birden çok tip

Bazen farklı istisnalara aynı tepki verilir. Her biri için ayrı ve özdeş bir `catch` yazmak kod tekrarına yol açar. Java 7'den beri tipler tek bir `catch`'te `|` işaretiyle birleştirilebilir. Bu yapıya çoklu yakalama (*multi-catch*) denir.

```
try {
  Object o = "Merhaba";
  Integer i = (Integer) o;
} catch (ClassCastException
    | NullPointerException e) {
  System.out.println("Tür veya null hatası: "
    + e.getClass().getSimpleName());
}
```

```
Tür veya null hatası: ClassCastException
```

`"Merhaba"` bir `String` nesnesidir; `Integer`'a dönüştürülemez. Fırlatılan `ClassCastException`, bloktaki iki tipten birine uyar ve blok çalışır. `e.getClass().getSimpleName()` gerçekte hangi istisnanın yakalandığını gösterir.

Çoklu yakalamanın iki kuralı vardır. Birincisi, bu blokta `e` örtük olarak `final` sayılır; ona yeni bir değer atanamaz. `e = null;` yazılırsa derleyici `multi-catch parameter e may not be assigned` hatası verir. Tek tipli bir `catch`'te aynı atama yasak değildir, ama iyi bir alışkanlık da değildir.

İkinci kural, alternatiflerin birbirinin alt sınıfı olamamasıdır.

```
} catch (FileNotFoundException
    | IOException e) {
```

```
Ana.java:9: error: Alternatives in a multi-catch statement cannot be related by
subclassing
    | IOException e) {
  Alternative FileNotFoundException is a subclass of alternative IOException
```

`FileNotFoundException`, `IOException`'ın alt sınıfıdır. `IOException` yazmak alt tipi zaten kapsar; ikisini birlikte yazmak anlamsızdır. Çözüm yalnız üst tipi yazmaktır: `catch (IOException e)`.

6.3 `finally` ve kaynak yönetimi

6.3.1 `finally` bloğu

Bir program dosya açar, veri okur ve dosyayı kapatır. Okuma sırasında bir istisna oluşursa ne olur? `try` bloğunun kalanı atlanır; kapatma satırı da `try`'ın içindeyse hiç çalışmaz. Dosya açık kalır. İşletim sistemi bir programın aynı anda açabileceği dosya sayısını sınırlar. Kapatılmayan dosyalar, ağ

bağlantıları ve veri tabanı bağlantıları birikir ve sonunda program yeni kaynak açamaz hâle gelir. Bu soruna kaynak sızıntısı denir.

Tanım 6.3

`finally` **bloğu** (*finally block*), `try` ve `catch` bloklarının ardından yazılan ve her durumda çalışan bloktur. İstisna oluşsun ya da oluşmasın, istisna yakalansın ya da yakalanmasın, `finally` çalışır.

```
try {
  // Potansiyel hata içeren kodlar
} catch (Exception e) {
  // Diğer hata türleri için genel durum
} finally {
  // Her durumda çalıştırılacak kodlar
}
```

`finally`'nin "her durumda" çalışması dört durumu kapsar:

- İstisna yoktur: `try` biter, `finally` çalışır.
- İstisna vardır ve bir `catch` yakalar: `catch` biter, `finally` çalışır.
- İstisna vardır ama hiçbir `catch` eşleşmez: `finally` çalışır, ardından istisna metottan dışarı çıkar.
- `try` ya da `catch` içinde `return` vardır: `finally`, metottan çıkmadan hemen önce çalışır.

Bu güvence, belirli adımların kesinlikle çalışmasını sağlar. Tipik kullanım temizlik işleridir: dosyaları, ağ ve veri tabanı bağlantılarını kapatmak, kilitleri bırakmak, geçici dosyaları silmek.

Bir `try`'dan sonra en az bir `catch` ya da bir `finally` gelmelidir. Yalnız `try { } finally { }` yazmak geçerlidir. Bu biçimde istisna yakalanmaz, ama temizlik yine yapılır. Tek başına bir `try` ise derleme hatasıdır: `'try' without 'catch', 'finally' or resource declarations`.

Kuralın asıl istisnası `System.exit()` çağrısıdır. Bu çağrı JVM'i hemen durdurur. `try` içinde `System.exit(0)` çalışırsa `finally` çalışmaz. JVM'in kendisinin çökmesi ya da bilgisayarın kapanması da elbette `finally`'yi engeller.

6.3.2 Kod izleme: `finally` ve `return`

`finally`'nin en şaşırtıcı yanı `return` ile ilişkisidir. Aşağıdaki metotta `try` da `catch` da bir değer döndürür.

```
static int bol(int a, int b) {
  try {
    System.out.println("try başladı");
    int s = a / b;
    System.out.println("try bitti");
    return s;
  } catch (ArithmeticException e) {
    System.out.println("catch: " + e.getMessage());
    return -1;
  } finally {
    System.out.println("finally");
  }
}
public static void main(String[] args) {
  System.out.println("Sonuç = " + bol(10, 2));
  System.out.println("Sonuç = " + bol(10, 0));
}
```

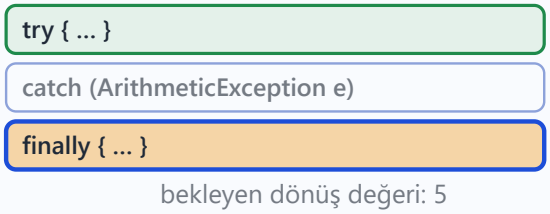
```
try başladı
try bitti
finally
Sonuç = 5
try başladı
catch: / by zero
finally
Sonuç = -1
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	15: `System.out.println("Sonuç = " + bol(10, 2));`	main()	—	—
2	3: `System.out.println("try başladı");`	main() → bol(10, 2)	a=10, b=2	`try başladı`
3	4: `int s = a / b;` 5: `System.out.println("try bitti");`	main() → bol(10, 2)	a=10, b=2, s=5	`try bitti`
4	6: `return s;`	main() → bol(10, 2)	a=10, b=2, s=5	—
5	11: `System.out.println("finally");`	main() → bol(10, 2)	a=10, b=2, s=5	`finally`
6	15: `System.out.println("Sonuç = " + bol(10, 2));`	main()	—	`Sonuç = 5`
7	16: `System.out.println("Sonuç = " + bol(10, 0));`	main()	—	—
8	3: `System.out.println("try başladı");`	main() → bol(10, 0)	a=10, b=0	`try başladı`

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
9	4: `int s = a / b;`	main() → bol(10, 0)	a=10, b=0	—
10	8: `System.out.println("catch: " + e.getMessage());`	main() → bol(10, 0)	a=10, b=0, e=ArithmeticException	`catch: / by zero`
11	9: `return -1;`	main() → bol(10, 0)	a=10, b=0, e=ArithmeticException	—
12	11: `System.out.println("finally");`	main() → bol(10, 0)	a=10, b=0	`finally`
13	16: `System.out.println("Sonuç = " + bol(10, 0));`	main()	—	`Sonuç = -1`

Tablo 6.5: `bol(10, 2)` ve `bol(10, 0)` çağrılarının izlemesi.

1. adımda `println`'ın argümanı hesaplanmalıdır. Bunun için önce `bol(10, 2)` çağrılır. `try` başlar, bölme başarılıdır ve `s = 5` olur. 4. adımda `return s` çalışır. Dönüş değeri olan 5 hazırlanır, ama metot hemen bitmez. Metottan çıkmadan önce `finally` bloğu çalışmalıdır. 5. adımda `finally` yazılır. `catch` ise atlanmıştır, çünkü istisna yoktur. Ancak 6. adımda bekleyen 5 döner ve `Sonuç = 5` yazılır.



Şekil 6.6: 5. adım. `try` bloğu `return s` ile bitti; dönüş değeri 5 bekletiliyor. Metot ancak `finally` çalıştıktan sonra döner.

İkinci çağrı `bol(10, 0)`'dır. `try başladı` yazılır. 9. adımda `10 / 0` istisna fırlatır. `try`'ın kalanı, yani `try bitti` satırı ve `return s` atlanır. `catch` eşleşir ve `catch: / by zero` yazar. 11. adımda `return -1` çalışır: değer hazırdır, ama yine önce `finally` çalışır. Sonunda `-1` döner. Sonuç açıktır: `finally` iki durumda da, dönüşten hemen önce çalıştı.

Örnek 6.2

Bir metotta `int s = 1;` tanımlıdır. `try` içinde `s = 2; return s;`, `finally` içinde `s = 3;` ve `System.out.print(s + " ");` yazılıdır. `System.out.println(g())` ne yazdırır?

`3 2`. `return s;` çalıştığında dönüş değeri hesaplanır ve saklanır: 2. Ardından `finally` çalışır, `s`'yi 3 yapar ve `3 ` yazar. Ama saklanmış dönüş değeri bir kopyadır; `s`'deki değişiklik onu etkilemez. Metot 2 döndürür. `finally` içinde ayrıca bir `return` olsaydı o değer dönerdi. Bu kullanım bekleyen değeri ve hatta bekleyen istisnayı sessizce siler; bu yüzden önerilmez.

Sık yapılan hata

`finally` içine `return` yazmak. `try` bir istisna fırlatmış olsa bile `finally`'deki `return` bu istisnayı yok eder ve metot sanki her şey yolundaymış gibi bir değer döndürür. Hata iz bırakmadan kaybolur. `finally` yalnız temizlik içindir; sonuç üretmez.

6.3.3 try-with-resources

`finally` ile kaynak kapatmak doğrudur, ama kod uzar. Kaynak değişkeni `try`'dan önce `null` olarak tanımlanır, `finally`'de `null` olup olmadığı denetlenir, `close()` çağrılır. `close()` de bir istisna fırlatabildiği için çoğu zaman bir `try` - `catch` daha gerekir. Java 7 bu işi kısaltan bir yapı getirdi.

Tanım 6.4

try-with-resources, `try` anahtar sözcüğünden sonraki parantezde kaynak açan `try` deyimidir. Parantezde açılan kaynak, blok bitince otomatik olarak kapatılır: JVM kaynağın `close()` metodunu çağırır. Kapatma, blok istisnayla bitse bile yapılır. Kaynak sınıfı `AutoCloseable` arayüzünü uygulamalıdır.

Sunumdaki örnek, açılışını ve kapanışını ekrana yazan küçük bir kaynak sınıfı kullanır.

```
class Kaynak implements AutoCloseable {
  private String ad;
  Kaynak(String ad) {
    this.ad = ad;
    System.out.println(ad + " açıldı");
  }
  void kullan(int x) {
    System.out.println(ad + " kullanılıyor");
    int s = 10 / x;
  }
  @Override
  public void close() {
    System.out.println(ad + " kapatıldı");
  }
}
```

```
try (Kaynak k = new Kaynak("dosya")) {
  k.kullan(0);
} catch (ArithmeticException e) {
  System.out.println("Hata: " + e.getMessage());
}
```

```
dosya açıldı
dosya kullanılıyor
dosya kapatıldı
Hata: / by zero
```

Çıktının sırası dikkat ister. Önce kaynak açılır ve kullanılır. `kullan(0)` içindeki bölme istisna fırlatır. Bu noktada `catch`'e geçilmeden önce kaynak kapatılır: `dosya kapatıldı`. Ancak ondan sonra `catch` çalışır. Yani kaynak, `catch`'ten önce kapanır.

Java kütüphanesinde kaynak işi gören sınıfların çoğu `AutoCloseable`'ı uygular: dosya akışları, `Scanner`, NIO kanalları, veri tabanı bağlantıları. Bu sınıflarla `finally` ve `null` denetimi yazmaya gerek kalmaz. try-with-resources, kaynak kapatmak için `finally` yazmanın modern ve güvenli yoludur.

`AutoCloseable`'ı uygulamayan bir sınıf parantezde kullanılamaz. `kapat()` adlı bir metodu olsa bile derleyici şu hatayı verir: `incompatible types: try-with-resources not applicable to variable type`. JVM'in çağıracağı metodun adı ve imzası arayüzle sabittir: `close()`.

Sık yapılan hata

`close()` metodunun bildirdiği istisnayı unutmak. `AutoCloseable.close()`, `throws Exception` ile bildirilmiştir. Bir sınıf `close()`'u bu bildirimi koruyarak yazarsa try-with-resources'ın kendisi kontrol edilen bir istisna fırlatabilir hâle gelir. Derleyici `unreported exception Exception` hatası verir ve örtük `close()` çağrısını gösterir. Sunumdaki `Kaynak` sınıfı `close()`'u `throws` olmadan yazdığı için bu sorun çıkmaz.

6.3.4 Kod izleme: iki kaynağın açılış ve kapanış sırası

Parantezde birden çok kaynak açılabilir; kaynaklar noktalı virgülle ayrılır.

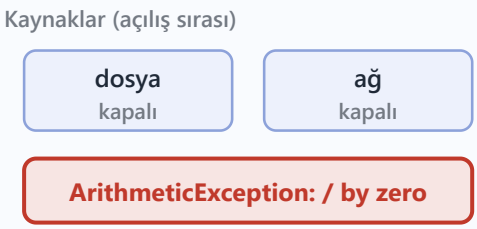
```
try (Kaynak a = new Kaynak("dosya");
     Kaynak b = new Kaynak("ağ")) {
  b.kullan(0);
  System.out.println("try sonu");
} catch (ArithmeticException e) {
  System.out.println("Hata: " + e.getMessage());
} finally {
  System.out.println("finally");
}
```

```
dosya açıldı
ağ açıldı
ağ kullanılıyor
ağ kapatıldı
dosya kapatıldı
Hata: / by zero
finally
```

Adım	Satır	Değişkenler	Yeni çıktı
1	1: `try (Kaynak a = new Kaynak("dosya");`	—	`dosya açıldı`
2	2: `Kaynak b = new Kaynak("ağ")) {`	—	`ağ açıldı`
3	3: `b.kullan(0);`	—	`ağ kullanılıyor`
4	3: `b.kullan(0);`	—	—
5	2: `Kaynak b = new Kaynak("ağ")) {`	—	`ağ kapatıldı`
6	1: `try (Kaynak a = new Kaynak("dosya");`	—	`dosya kapatıldı`
7	5: `} catch (ArithmeticException e) {` 6: `System.out.println("Hata: " + e.getMessage());`	—	`Hata: / by zero`
8	8: `System.out.println("finally");`	—	`finally`

Tablo 6.6: İki kaynaklı try-with-resources izlemesi.

Kaynaklar yazılış sırasıyla açılır: önce `dosya`, sonra `ağ`. 3. adımda `b.kullan(0)` önce `ağ kullanılıyor` yazar, sonra `10 / 0` istisnası fırlatılır. `try sonu` satırı atlanır. Ama denetim hemen `catch`'e geçmez; önce kaynaklar kapatılır.



Şekil 6.7: 6. adım. İki kaynak da istisnaya rağmen kapandı. İstisna nesnesi henüz yakalanmadı; sırada `catch` var.

Kapatma ters sırayla yapılır. Son açılan `b` önce kapanır, sonra `a` kapanır. Bunun nedeni kaynaklar arasındaki olası bağımlılıktır. İkinci kaynak çoğu zaman birinciyi kullanır; örneğin bir okuyucu bir dosya akışını sarar. Önce dıştaki sarmalayıcı kapatılır, sonra içteki akış. İki kaynak da kapandıktan sonra `catch` çalışır, en son `finally`. Sıra şöyle özetlenir: aç, kullan, ters sırayla kapat, `catch`, `finally`.

İstisna olmasaydı sıra nasıl olurdu? `b.kullan(5)` çağrısında bölme başarılıdır. `try sonu` yazılır, ardından kaynaklar yine ters sırayla kapanır ve `finally` çalışır:

```
dosya açıldı
ağ açıldı
ağ kullanılıyor
try sonu
ağ kapatıldı
dosya kapatıldı
finally
```

6.3.5 İç içe `try` blokları

Bir `try` bloğunun içinde başka bir `try` bloğu bulunabilir. Bu yapı, bir işin bir bölümüne özel önlem alırken bütün işe genel bir önlem almak için kullanılır.

```
try {
  // Dıştaki try bloğu
  try {
    // İçteki try bloğu
    // Potansiyel hata içeren kodlar
  } catch (Exception e) {
    // İçteki catch bloğu
  } finally {
    // İçteki finally bloğu
  }
} catch (Exception e) {
  // Dıştaki catch bloğu
} finally {
  // Dıştaki finally bloğu
}
```

İçteki `try`'da bir istisna oluşursa JVM şu sırayla ilerler:

1. İçteki `try`'ın `catch` blokları denenir. Uygun bir `catch` varsa o çalışır.
2. Uygun `catch` yoksa istisna dışa doğru ilerler. Dış `try`'ın `catch` blokları denenir ve ilk uygun olan çalışır.
3. Hiçbir `catch` eşleşmezse istisna metottan çıkar. Sonunda kimse yakalamazsa JVM'in varsayılan işleyicisi programı sonlandırır.

Her durumda, istisnanın geçtiği her düzeydeki `finally` bloğu çalışır.

6.3.6 Kod izleme: istisna dışa doğru ilerler

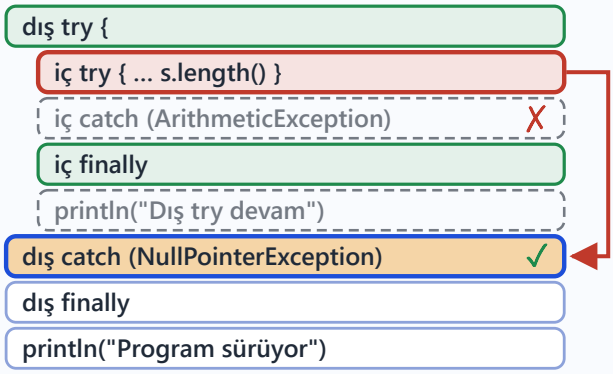
```
public static void main(String[] args) {
  try {
    System.out.println("Dış try");
    try {
      System.out.println("İç try");
      String s = null;
      System.out.println(s.length());
    } catch (ArithmeticException e) {
      System.out.println("İç catch");
    } finally {
      System.out.println("İç finally");
    }
    System.out.println("Dış try devam");
  } catch (NullPointerException e) {
    System.out.println("Dış catch: NPE");
  } finally {
    System.out.println("Dış finally");
  }
  System.out.println("Program sürüyor");
}
```

```
Dış try
İç try
İç finally
Dış catch: NPE
Dış finally
Program sürüyor
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	3: `System.out.println("Dış try");`	main()	—	`Dış try`
2	5: `System.out.println("İç try");` 6: `String s = null;`	main()	s=null	`İç try`
3	7: `System.out.println(s.length());`	main()	s=null	—
4	8: `} catch (ArithmeticException e) {`	main()	s=null	—
5	11: `System.out.println("İç finally");`	main()	s=null	`İç finally`
6	14: `} catch (NullPointerException e) {` 15: `System.out.println("Dış catch: NPE");`	main()	e=NullPointerException	`Dış catch: NPE`
7	17: `System.out.println("Dış finally");`	main()	—	`Dış finally`
8	19: `System.out.println("Program sürüyor");`	main()	—	`Program sürüyor`

Tablo 6.7: İç içe `try` programının izlemesi.

Dış `try` başlar, ardından iç `try`. `s` değişkeni `null` değerini alır. 3. adımda `s.length()` çağrılır. `null` bir referans üzerinden metot çağrılamaz; JVM bir `NullPointerException` fırlatır. İç `catch` yalnız `ArithmeticException` yakalar ve eşleşmez. İstisna iç düzeyden çıkmadan önce iç `finally` çalışır: `İç finally`.



Şekil 6.8: 6. adım. İstisna iç `try`'dan çıktı; iç `catch` eşleşmedi, iç `finally` çalıştı. `Dış try devam` satırı atlandı ve dış `catch` eşleşti.

6. adımda istisna dış düzeye geçmiştir. Dış `try`'da iç bloğun ardından gelen `Dış try devam` satırı atlanır. Dış `catch` `NullPointerException` içindir ve eşleşir. Ardından dış `finally` çalışır. İstisna ele alındığı için program normal akışına döner ve `Program sürüyor` yazılır.

Örnek 6.3

İç `try`'da `Object o = "12"; Integer i = (Integer) o;` çalışsın. İç `catch` `NumberFormatException`, dış `catch` `ClassCastException` yakalasın. Her düzeyde bir `finally` ve dış `try`'ın sonunda `dış try sonu` yazan bir satır bulunsun. Program ne yazdırır?

`iç finally`, `dış catch`, `dış finally` (ardından programın son satırı `son` yazar). `o` bir `String` nesnesidir. `"12"` bir sayıya benzese de dönüşüm metni sayıya çevirmez; `ClassCastException` fırlatır. İç `catch` bu tipi yakalamaz. İstisna dışa ilerlerken iç `finally` çalışır. `dış try sonu` atlanır, dış `catch` yakalar, dış `finally` çalışır ve program sürer.

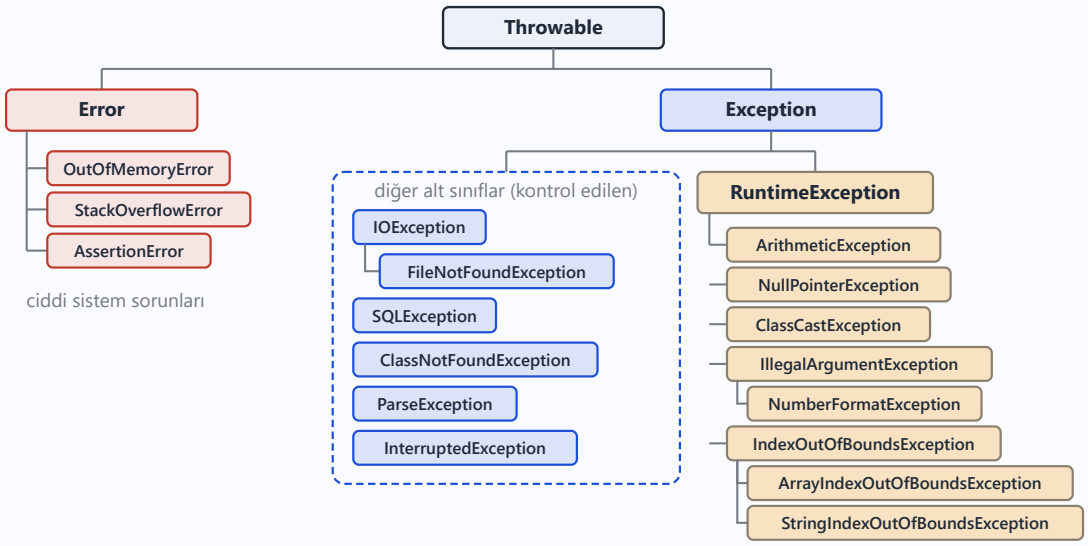
6.4 İstisna hiyerarşisi

6.4.1 `Throwable` ağacı

Java'da istisna sınıfları bir sınıf hiyerarşisi içindedir. Ağacın kökünde `Throwable` sınıfı bulunur. `Throwable`, fırlatılabilen her şeyin atasıdır. Yalnız `Throwable`'ın ve alt sınıflarının nesneleri `throw` ile fırlatılabilir ve `catch` ile yakalanabilir. `throw "hata";` gibi bir satır bu yüzden derlenmez:

`incompatible types: String cannot be converted to Throwable`.

`Throwable`'ın iki doğrudan alt sınıfı vardır: `Error` ve `Exception`. `Exception`'ın alt sınıflarından biri olan `RuntimeException` ise ayrı bir aile oluşturur. Şekil 6.9 bu yapıyı gösterir.



Şekil 6.9: İstisna sınıf ağacı. Mavi kutular kontrol edilen istisnalardır. Turuncu kutular `RuntimeException` ve alt sınıflarıdır; kırmızı kutular `Error` ve alt sınıflarıdır. Bu iki aile kontrol edilmeyen istisnaları oluşturur.

Hiyerarşinin pratik bir sonucu vardır: bir `catch` bloğu, yazılan tipi ve onun bütün alt sınıflarını yakalar. Bu, kalıtımın doğal sonucudur. Her `FileNotFoundException` aynı zamanda bir `IOException`'dır. Bu yüzden `catch (IOException e)` hem `FileNotFoundException`'ı hem de diğer giriş/çıkış hatalarını yakalar. Aynı aileden istisnalar tek bir üst tiple yakalanabilir.

Ağaçtaki birkaç ilişki ayrıca not edilmeye değer. `NumberFormatException`, `IllegalArgumentException`'ın alt sınıfıdır. `ArrayIndexOutOfBoundsException` ve `StringIndexOutOfBoundsException`, `IndexOutOfBoundsException`'ın alt sınıflarıdır. `StackOverflowError` ve `OutOfMemoryError` doğrudan `Error`'dan değil, onun alt sınıfı `VirtualMachineError`'dan türer. Şekil bu ara düzeyi sadelik için göstermez.

Java, yeni istisna sınıfları yazmayı da destekler. Uygulamaya özgü bir durum, örneğin "bakiye yetersiz", kendi sınıfıyla bu hiyerarşiye eklenir. Bu sınıflar da ağacın bir dalına bağlanır ve aynı kurallara uyar.

Örnek 6.4

`new FileReader("yok.txt")` bir `FileNotFoundException` fırlatıyor. `RuntimeException`, `Error`, `SQLException` ve `IOException` tiplerinden hangisi bu istisnayı yakalar?

Yalnız `IOException`. İstisnanın üst sınıf zinciri `FileNotFoundException` → `IOException` → `Exception` → `Throwable`'dır. `RuntimeException` ve `SQLException` `Exception`'ın ayrı dallarıdır. `Error` ise `Exception` ailesinden değildir. `catch (Exception e)` ya da `catch (Throwable t)` da yakalardı, ama seçenekler arasında yoktur.

6.4.2 `Error` ve `Exception`

Ağacın iki büyük kolu farklı soruları yanıtlar.

Tanım 6.5

`Error`, programın kurtarılamayacak ciddi bir sorunla karşılaştığını belirtir. Bu durumlarla başa çıkmak çoğunlukla mümkün değildir ve program genellikle sonlanır. `Exception` ise programın öngörebileceği ve çoğu zaman kurtarabileceği durumları temsil eder; `try`-`catch` ile ele alınması beklenen tip budur.

`Error` örnekleri aşırı durumlardır: program belleğinin tükenmesi (`OutOfMemoryError`), çağrı yığınının taşması (`StackOverflowError`), disk sürücüsünün "kaybolması". Bu durumlar programın bir satırından çok, ortamın durumundan kaynaklanır. Uygulama kodunda `Error` yakalanması önerilmez. Bellek tükenmişse `catch` bloğunun çalışacak belleği de kalmamış olabilir.

`Exception` örnekleri ise gündelik sorunlardır: dosya yok, ağ koptu, girdi geçersiz, referans `null`. Bunların bir bölümü programın dışından gelir; bir bölümü programcının hatasıdır. `Exception`'ın alt sınıfı `RuntimeException` çoğunlukla programcı hatalarını temsil eder. Kontrol edilen ve kontrol edilmeyen istisna ayrımı bu farka dayanır.

`Error` ve `Exception`'ın ikisi de `Throwable`'dan türer; bu yüzden ikisi de fırlatılabilir ve teknik olarak yakalanabilir. Ama "ele alınmalı" olan, `Exception` ailesidir.

Sık yapılan hata

"`catch (Exception e)` her şeyi yakalar" sanmak. `Exception`, `Error` ailesini kapsamaz. Sonsuz özyinelemeden doğan `StackOverflowError` bu blokla yakalanmaz ve program yığın iziyle sonlanır. Her şeyi yakalayan tip `Throwable`'dır; ama onu yakalamak da `Error`'ları gizlediği için önerilmez.

6.5 Kontrol edilen ve kontrol edilmeyen istisnalar

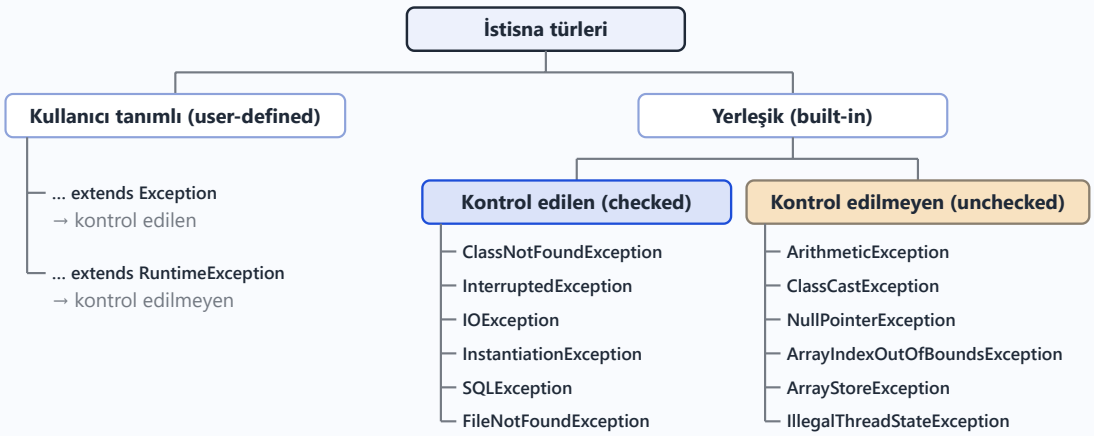
6.5.1 İki tür istisna

Java istisnaları derleyicinin tutumuna göre ikiye ayrılır.

Tanım 6.6

Kontrol edilen istisna (*checked exception*), derleme zamanında derleyicinin denetlediği istisnadır. Programcı bu istisnayı ya ele almak ya da bildirmek zorundadır. `Exception`'dan türeyip `RuntimeException`'dan türemeyen sınıflar bu gruptadır. **Kontrol edilmeyen istisna** (*unchecked exception*) ise derleme zamanında denetlenmez ve çalışma zamanında ortaya çıkar. `RuntimeException`, `Error` ve bunların alt sınıfları bu gruptadır.

Kontrol edilen istisnalar, ortaya çıkabileceği önceden bilinen durumları anlatır. Dosya okuyan her program, dosyanın bulunamaması olasılığıyla yüzleşmelidir. Derleyici bu olasılık için bir plan ister. Kontrol edilmeyen istisnalar ise çoğunlukla bir programlama hatasının belirtisidir. Derleyici her bölmede, her dizi erişiminde ve her metot çağrısında bir plan isteseydi kod okunmaz hâle gelirdi.



Şekil 6.10: İstisna türleri. Yerleşik istisnalar kontrol edilen ve kontrol edilmeyen olarak ikiye ayrılır. Kullanıcı tanımlı bir istisna `Exception`'ı genişletirse kontrol edilen, `RuntimeException`'ı genişletirse kontrol edilmeyen olur.

6.5.2 Kontrol edilen istisna: yakala ya da bildir

Kontrol edilen istisnalar için Java'nın kuralı "yakala ya da bildir" (*catch or declare*) kuralıdır. İstisna fırlatabilecek kod ya bir `try-catch` içinde ele alınır ya da metodun imzasında `throws` ile bildirilir. Bu kural derleme zamanında denetlenir; uyulmazsa program derlenmez.

Kuralın gerekçesi basittir. Var olmayan bir dosyayı okumaya çalışan program ne kadar doğru yazılırsa yazılsın, dosya o sırada silinmiş olabilir. Bu bir programcı hatası değildir; dış dünyanın bir gerçeğidir. Derleyici, programcının bu olasılığı unutmamasını sağlar.

```
public void dosyaOku() {
  FileReader okuyucu = new FileReader("veri.txt");
}
```

```
Ana.java:5: error: unreported exception FileNotFoundException; must be caught or declared
to be thrown
  FileReader okuyucu = new FileReader("veri.txt");
```

`FileReader`'ın yapıcısı, imzasında `throws FileNotFoundException` bildirir. `dosyaOku` metodu bu istisnayı ne yakalamış ne de bildirmiştir. Derleyici hatanın iki çözümünü iletinin kendisinde söyler: *must be caught* (yakalanmalı) ya da *declared to be thrown* (fırlatılacağı bildirilmeli). Bildirerek çözüm şöyledir:

```
public void dosyaOku() throws FileNotFoundException {
  // Dosya okuma kodu
  FileReader okuyucu = new FileReader("veri.txt");
}
```

Artık metot derlenir. Ama sorun ortadan kalkmamış, metodu çağıran koda geçmiştir. `dosyaOku()`'yu çağıran her kod da aynı kurala uymak zorundadır.

6.5.3 Kontrol edilen istisnayı ele almak

İkinci çözüm, istisnayı metodun içinde yakalamaktır.

```
public static void dosyaOku2() {
  try {
    FileReader okuyucu = new FileReader("veri.txt");
    // Dosya okuma kodu
  } catch (FileNotFoundException e) {
    // Kontrol edilen istisna burada ele alınıyor
    System.err.println("Dosya bulunamadı: " + e.getMessage());
    // Gerekirse istisna tekrar fırlatılabilir:
    // throw e;
  }
}
```

```
Dosya bulunamadı: veri.txt (The system cannot find the file specified)
```

Bu sürümde metot istisnayı kendisi yakalar; imzasında `throws` yoktur. Çağıran kodun hiçbir şey yapması gerekmez. Mesaj `System.err`'e, yani hata çıktısı akışına yazılır. Mesajın ikinci bölümü işletim sisteminden gelir. Yukarıdaki çıktı Windows'ta alınmıştır: `(The system cannot find the file specified)`. Linux'ta aynı durumda `(No such file or directory)` yazılır.

Yorum satırındaki `throw e;` başka bir olasılığı gösterir. Yakalanan istisna, örneğin bir kayıt tutulduktan sonra, yeniden fırlatılabilir. O zaman istisna yine metottan dışarı çıkar ve metot imzasında yine `throws FileNotFoundException` bildirilmelidir.

Sık yapılan hata

Fırlatılmayan bir kontrol edilen istisnayı yakalamaya çalışmak. Dosya işlemi içermeyen bir `try`'a `catch (FileNotFoundException e)` yazılırsa derleyici `exception FileNotFoundException is never thrown in body of corresponding try statement` hatası verir. Derleyici kontrol edilen istisnaların nereden gelebileceğini bilir. `catch (Exception e)` ise bu hatayı vermez, çünkü `Exception` kontrol edilmeyen `RuntimeException`'ları da kapsar.

6.5.4 Kontrol edilmeyen istisna

Kontrol edilmeyen istisnaların ele alınması ya da bildirilmesi zorunlu değildir. Bunlar genellikle mantık hatalarının ya da programcı hatalarının sonucudur: sıfıra bölme, `null` kullanımı, dizi sınırını aşma. `throws` ile belirtilmeleri ya da `try`-`catch` içine alınmaları gerekmez; derleyici sessiz kalır.

```
public void bolmeIslemi(int pay, int payda) {
  // Kontrol edilmeyen istisna: ArithmeticException
  int sonuc = pay / payda;
}
```

Bu metot derlenir ve `throws` gerektirmez. `payda` sıfır olursa çalışma zamanında `ArithmeticException` oluşur. Böyle bir hatanın doğru çözümü çoğu zaman yakalamak değil, önlemektir. Bölmeden önce payda, kullanmadan önce `null` denetlenir.

```
public void bolmeIslemi(int pay, int payda) {
  if (payda == 0) {   // önleme
    System.out.println("Payda sıfır olamaz");
    return;
  }
  int sonuc = pay / payda;
}
```

Önleme sürümünde istisna hiç oluşmaz. Denetim, sorunu kaynağında yakalar ve programcının durumu nasıl ele alacağına kendisinin karar vermesini sağlar. Burada metot bir mesaj yazıp erken döner. Gerçek bir programda çoğu zaman anlamlı bir mesajla `IllegalArgumentException` fırlatmak daha doğrudur, çünkü çağıranın bir hata yaptığını açıkça bildirir.

6.5.5 Üç ailenin karşılaştırması

	Kontrol edilen	**Kontrol edilmeyen**	`Error`
Üst sınıf	`Exception` (`RuntimeException` hariç)	`RuntimeException`	`Error`
Derleyici denetler mi?	Evet: ele al ya da bildir	Hayır	Hayır
Tipik neden	Programın dışındaki koşullar (dosya, ağ, veri tabanı)	Programcı hatası (`null`, sınır, bölme)	JVM ya da sistem sorunu
Ne yapılmalı?	Yakalayıp kurtarmak ya da `throws` ile iletmek	Kodu düzeltmek, girdiyi denetlemek	Genellikle yakalamamak; programı düzgün sonlandırmak
Örnekler	`IOException`, `SQLException`, `ClassNotFoundException`	`NullPointerException`, `ArithmeticException`	`OutOfMemoryError`, `StackOverflowError`

Tablo 6.8: Kontrol edilen istisnalar, kontrol edilmeyen istisnalar ve `Error`.

Örnek 6.5

`IOException`, `NumberFormatException`, `SQLException`, `NullPointerException`, `FileNotFoundException`, `ClassCastException` ve `StackOverflowError` sınıflarını kontrol edilen ve kontrol edilmeyen olarak ayıralım.

Her sınıf için üst sınıf zincirine bakılır. Zincirde `RuntimeException` ya da `Error` varsa sınıf kontrol edilmeyendir. Kontrol edilen: `IOException`, `SQLException`, `FileNotFoundException`. Kontrol edilmeyen: `NumberFormatException` (`IllegalArgumentException` üzerinden), `NullPointerException`, `ClassCastException` ve bir `Error` olan `StackOverflowError`.

Örnek 6.6

`main` içinde, `try-catch` ve `throws` olmadan yazılınca `Integer.parseInt("5")`, `Thread.sleep(100)`, `new ArrayList<Integer>(-1)` ve `Objects.requireNonNull(x)` satırlarından hangisi derlenmez?

`Thread.sleep(100)`. Bu metot kontrol edilen `InterruptedException`'ı fırlatabileceğini bildirir; derleyici `unreported exception InterruptedException` hatası verir. Diğerleri en fazla kontrol edilmeyen bir istisna fırlatır (`NumberFormatException`, `IllegalArgumentException`, `NullPointerException`). Derleyici bunları denetlemez. Üçüncü satır derlenir, ama çalışırken istisna verir.

6.6 İstisna fırlatmak

6.6.1 `throw` deyimi

Java çalışma zamanı beklenmedik bir durumla karşılaştığında bir istisna nesnesi oluşturur ve onu fırlatır. Amaç, durumu üst düzeydeki koda bildirmektir. Programcı da aynı işi `throw` anahtar sözcüğüyle yapabilir.

Tanım 6.7

`throw` **deyimi**, bir istisna nesnesini fırlatır: `throw ifade;`. İfadenin değeri bir `Throwable` nesnesi olmalıdır. Nesne çoğunlukla aynı satırda `new` ile oluşturulur ve yapıcısına bir mesaj verilir.

```
throw new Exception("Bu bir istisna durumu!");
```

İstisna, normal program akışını değiştirir. `throw`'dan sonraki satırlar çalışmaz; denetim uygun bir `catch`'e geçer. Fırlatılan istisna, çağıran kodda bir `try-catch` ile yakalanabilir. Hiçbir yerde ele alınmazsa JVM programı sonlandırır ve neyin yanlış gittiğini, nerede olduğunu bildirir.

```
try {
  throw new Exception("Bu bir istisna durumu!");
} catch (Exception e) {
  System.out.println(e);
}
```

```
java.lang.Exception: Bu bir istisna durumu!
```

Fırlatılabilecek pek çok tür vardır: `IOException`, `NullPointerException`, `IllegalArgumentException` ve programcının kendi yazdığı özel sınıflar. Yukarıdaki örnekte `println(e)`, nesnenin `toString()` sonucunu yazar: sınıfın tam adı ve yapıcıya verilen mesaj.

Sık yapılan hata

`throw`'dan sonra aynı bloğa kod yazmak. `throw` koşulsuz çalışıyorsa ondan sonraki satıra hiçbir zaman ulaşılamaz. Derleyici bunu fark eder ve `unreachable statement` hatası verir. `throw` çoğunlukla bir `if`'in içinde bulunur; o zaman sonraki satırlar koşul yanlışken çalışır.

6.6.2 Durumu en iyi anlatan tip

İstisna tipi seçilirken durumu en iyi temsil eden sınıf seçilir. Geçersiz bir argüman için `IllegalArgumentException`, nesnenin o anki durumunda yapılamayacak bir işlem için `IllegalStateException` uygundur. Var olan sınıflardan hiçbiri uymuyorsa programa özel bir istisna sınıfı yazılır.

```
class CustomException extends Exception {
  public CustomException(String mesaj) {
    super(mesaj);
  }
}
public static void kontrolEt(int deger)
    throws CustomException {
  if (deger < 0) {
    // Belirli bir durumu kontrol et ve istisna fırlat
    throw new CustomException("Değer negatif olamaz!");
  }
  // Değer pozitifse, normal işlem devam eder
  System.out.println("Değer uygun: " + deger);
}
// main içinde
try {
  kontrolEt(5);
  kontrolEt(-3);
  kontrolEt(8);
} catch (CustomException e) {
  System.out.println("Yakalandı: " + e.getMessage());
}
```

```
Değer uygun: 5
Yakalandı: Değer negatif olamaz!
```

`CustomException`, `Exception`'ın alt sınıfıdır. `RuntimeException`'dan türemediği için kontrol edilen bir istisnadır. Bu yüzden `kontrolEt` metodunun imzasında `throws CustomException` şarttır. Bu bildirim kaldırılırsa derleyici `unreported exception CustomException; must be caught or declared to be thrown` hatası verir.

Çıktıda `Değer uygun: 8` satırı yoktur. `kontrolEt(-3)` istisna fırlattığı anda `try`'ın kalanı atlanır; `kontrolEt(8)` hiç çağrılmaz.

6.6.3 Kod izleme: `throw`

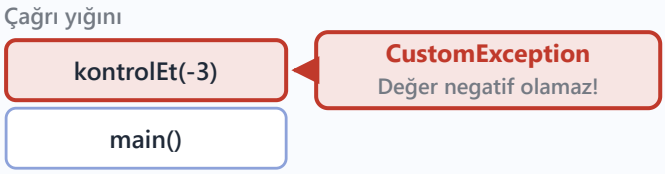
Aynı programı çağrı yığınıyla birlikte adım adım izleyelim. Çağrı yığını, çalışan metotların çerçevelerini tutan yığındır; her metot çağrısı yığına bir çerçeve ekler, metot bitince çerçeve kaldırılır.

```
static void kontrolEt(int deger) throws CustomException {
  if (deger < 0) {
    throw new CustomException("Değer negatif olamaz!");
  }
  System.out.println("Değer uygun: " + deger);
}
// main içinde
try {
  kontrolEt(5);
  kontrolEt(-3);
  kontrolEt(8);
} catch (CustomException e) {
  System.out.println("Yakalandı: " + e.getMessage());
}
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	9: `kontrolEt(5);`	main()	—	—
2	2: `if (deger < 0) {`	main() → kontrolEt(5)	deger=5	—
3	5: `System.out.println("Değer uygun: " + deger);`	main() → kontrolEt(5)	deger=5	`Değer uygun: 5`
4	10: `kontrolEt(-3);`	main()	—	—
5	2: `if (deger < 0) {`	main() → kontrolEt(-3)	deger=-3	—
6	3: `throw new CustomException("Değer negatif olamaz!");`	main() → kontrolEt(-3)	deger=-3	—
7	12: `} catch (CustomException e) {`	main()	e=CustomException	—
8	13: `System.out.println("Yakalandı: " + e.getMessage());`	main()	e=CustomException	`Yakalandı: Değer negatif olamaz!`
9	14: `}`	main()	—	—

Tablo 6.9: `throw` programının izlemesi.

1. adımda `main`, `try` içinde `kontrolEt(5)`'i çağırır ve yığına yeni bir çerçeve eklenir. `5 < 0` yanlıştır; `throw` satırı atlanır. `Değer uygun: 5` yazılır ve metot normal biçimde döner. 4. adımda sıradaki çağrı `kontrolEt(-3)`'tür. Bu kez koşul doğrudur.



Şekil 6.11: 6. adım. `kontrolEt(-3)` çerçevesinde bir `CustomException` nesnesi oluştu ve fırlatıldı. Metotta `catch` olmadığı için bu çerçeve atılacak.

6. adımda `new` ile istisna nesnesi oluşur ve `throw` onu fırlatır. Metodun 5. satırı çalışmaz. `kontrolEt`'te `catch` yoktur; bu yüzden çerçeve atılır ve istisna `main`'e geçer. 7. adımda `try`'ın kalanı, yani `kontrolEt(8)` atlanır ve uygun `catch` bulunur. 8. adımda `getMessage()`, yapıcıya verilen mesajı döndürür: `Yakalandı: Değer negatif olamaz!`. Son adımda `catch` biter ve program normal akışına döner.

6.6.4 `throw` ve `throws` farkı

İki anahtar sözcük yalnız bir harfle ayrılır, ama görevleri tamamen farklıdır.

	`throw`	`throws`
Görevi	Bir istisnayı fırlatır (şimdi)	Metodun fırlatabileceği istisnaları bildirir (ileride olabilir)
Yeri	Metot gövdesinde, bir deyim olarak	Metot imzasında, parametrelerden sonra
Ardından	Tek bir nesne: `throw new X("…");`	Bir ya da daha çok tip: `throws X, Y`
Örnek	`throw new ArithmeticException("Sıfıra bölme");`	`void bol(int x, int y) throws ArithmeticException`

Tablo 6.10: `throw` ile `throws`'un karşılaştırması.

`throw` bir eylemdir; `throws` ise bir uyarı etiketidir. Bir metot `throws` ile bildirdiği istisnayı hiç fırlatmayabilir; bildirim yalnız bir olasılığı duyurur. Kontrol edilmeyen istisnaların `throws` ile bildirilmesi zorunlu değildir, ama belgeleme amacıyla yazılabilir. Tablodaki `throws ArithmeticException` bu türden bir bildirimdir.

İki kelimeyi karıştırmak iki tipik derleme hatasına yol açar. Birincisinde `throws` gövdede kullanılmıştır:

```
if (x < 0) {
  throws new IllegalArgumentException("negatif");
}
```

```
Ana.java:6: error: illegal start of expression
  throws new IllegalArgumentException("negatif");
```

Gövdede `throws` diye bir deyim yoktur; derleyici satırı bir ifadenin başlangıcı olarak tanıyamaz. İkincisinde kontrol edilen bir istisna bildirilmeden fırlatılmıştır:

```
static void f(int x) {
  if (x < 0) throw new Exception("negatif");
}
```

```
Ana.java:5: error: unreported exception Exception; must be caught or declared to be thrown
  if (x < 0) throw new Exception("negatif");
```

`Exception`, `RuntimeException` olmadığı için kontrol edilen bir tiptir. İki çözüm vardır: imzaya `throws Exception` eklemek ya da `IllegalArgumentException` gibi kontrol edilmeyen ve durumu daha iyi anlatan bir tip fırlatmak. Negatif argüman bir programcı hatası olduğu için ikinci çözüm daha uygundur.

Sık yapılan hata

"`throw` yalnız `catch` bloğunda kullanılabilir" sanmak. `throw` gövdenin her yerinde geçerli bir deyimdir: `if` içinde, döngüde, `catch` 'te, hatta metodun ilk satırında. `catch` içinde kullanıldığında yakalanan istisnayı yeniden fırlatır.

6.6.5 İstisna fırlatan metotları çağırmak

Bir metot çağrılırken imzasında bildirilen istisnalar dikkate alınır ve uygun bir strateji seçilir. Sunumdaki dosya okuma metodu kontrol edilen bir istisna bildirir.

```
public void dosyaOku() throws FileNotFoundException {
  // Dosya okuma kodu
}
```

Birinci strateji, çağıranın istisnayı ele almasıdır. Çağrı uygun bir `try` - `catch` bloğunun içine yazılır. Çoğu durumda tercih edilen yol budur.

```
try {
  dosyaOku();
} catch (FileNotFoundException e) {
  // Dosya bulunamadı durumuyla başa çıkma kodu
  System.err.println("Dosya bulunamadı: " + e.getMessage());
}
```

İkinci strateji, çağıranın da istisnayı bildirmesidir. Çağıran metot imzasında `throws` ile aynı tipi bildirir ve sorun bir üst düzeye geçer.

```
public void raporHazirla() throws FileNotFoundException {
  dosyaOku();
}
```

İkinci yol sorunu yalnız ertelediği için yakalamaktan daha az tercih edilir. Yine de bazen doğru seçimdir: `raporHazirla` kullanıcıyla konuşmuyorsa dosyanın bulunamaması durumunda ne yapılacağına da karar veremez. Bildirimdeki tip, fırlatılan tipin kendisi ya da bir üst sınıfı olabilir. `throws IOException` yazmak da derlenir, çünkü her `FileNotFoundException` bir `IOException`'dır. Hiçbir strateji seçilmezse derleyici `dosyaOku();` satırı için `unreported exception FileNotFoundException` hatası verir.

`main` metodu bile `throws` yazabilir: `public static void main(String[] args) throws IOException`. O zaman istisna JVM'e kadar iletilir ve yakalanmayan istisna programı sonlandırır. Küçük deneme programlarında pratik bir kısayoldur; gerçek uygulamalarda kullanıcıya anlaşılır bir mesaj vermek daha doğrudur.

6.6.6 Kod izleme: istisnanın yayılımı

> **Tanım 6.8**
>
> **İstisna yayılımı** (*exception propagation*), yakalanmayan bir istisnanın çağrı yığınında çağıran metoda doğru ilerlemesidir. İstisnanın geçtiği her metot o anda sona erer ve çerçevesi yığından atılır.

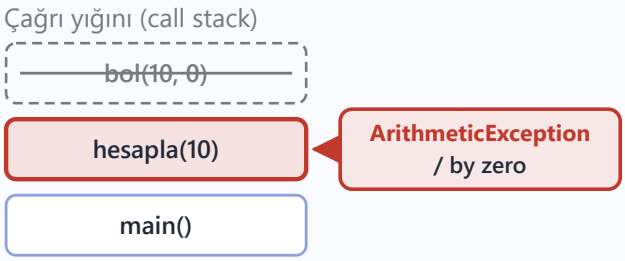
```
static int bol(int a, int b) {
  System.out.println("bol başladı");
  int s = a / b;
  System.out.println("bol bitti");
  return s;
}
static int hesapla(int x) {
  System.out.println("hesapla başladı");
  int r = bol(x, 0);
  System.out.println("hesapla bitti");
  return r;
}
public static void main(String[] args) {
  try {
    int sonuc = hesapla(10);
    System.out.println("Sonuç: " + sonuc);
  } catch (ArithmeticException e) {
    System.out.println("main: " + e.getMessage());
  }
  System.out.println("Program bitti");
}
```

```
hesapla başladı
bol başladı
main: / by zero
Program bitti
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	15: `int sonuc = hesapla(10);`	main()	—	—
2	8: `System.out.println("hesapla başladı");`	main() → hesapla(10)	x=10	`hesapla başladı`
3	9: `int r = bol(x, 0);`	main() → hesapla(10)	x=10	—
4	2: `System.out.println("bol başladı");`	main() → hesapla(10) → bol(10, 0)	a=10, b=0	`bol başladı`
5	3: `int s = a / b;`	main() → hesapla(10) → bol(10, 0)	a=10, b=0	—
6	9: `int r = bol(x, 0);`	main() → hesapla(10)	x=10	—
7	15: `int sonuc = hesapla(10);`	main()	—	—
8	17: `} catch (ArithmeticException e) {` 18: `System.out.println("main: " + e.getMessage());`	main()	e=ArithmeticException	`main: / by zero`
9	20: `System.out.println("Program bitti");`	main()	—	`Program bitti`

Tablo 6.11: İstisnanın `bol`'dan `main`'e yayılımı.

İlk dört adımda yığın büyür: `main`, `hesapla(10)`'u; `hesapla` da `bol(10, 0)`'ı çağırır. Yığının tepesinde artık `bol` vardır. 5. adımda `10 / 0` hesaplanır ve istisna `bol`'da doğar. `bol`'da `try` yoktur; bu yüzden metodun kalanı çalışmaz ve çerçevesi atılır. `bol bitti` yazılmaz.



Şekil 6.12: 6. adım. `bol`'un çerçevesi atıldı (üstü çizili). İstisna, çağıran metoda, `hesapla`'ya geçti. Burada da `try` olmadığı için bu çerçeve de atılacak.

6. adımda istisna çağıran yere, `hesapla`'ya geçer. Burada da `try` yoktur. `hesapla bitti` satırı atlanır ve `hesapla`'nın çerçevesi de atılır. 7. adımda istisna `main`'e ulaşır. Çağrı bir `try` içindedir; JVM uygun bir `catch` arar. `Sonuç` satırı atlanır. 8. adımda `catch` eşleşir; istisna yakalanmış ve yayılım durmuştur. Son adımda program normal biçimde biter. Kimse yakalamasaydı istisna JVM'e ulaşır ve program sonlanırdı.

Bu izlemenin ana fikri şudur: istisnanın doğduğu yerle yakalandığı yer farklı metotlar olabilir. `bol` ve `hesapla` sorunu çözmez, ona karışmaz da. Sorunla ilgilenen tek metot `main`'dir. Her metot yalnız kendi işini yapar; hata işleme kodu tek bir yerde toplanır.

Örnek 6.7

`c(x)`, `c x` yazar; `x > 2` ise `IllegalArgumentException("x=" + x)` fırlatır, değilse `c bitti` yazar. `b(x)`, bir `try` içinde `c(x)`'i, `c(x + 2)`'yi çağırır ve `b bitti` yazar; `finally`'de `b finally` yazar. `main`, `try` içinde `b(1)`'i çağırır ve `main devam` yazar; `catch` `main:` ile mesajı yazar; en sonda `son` yazılır. Çıktı nedir?

`c 1`, `c bitti`, `c 3`, `b finally`, `main: x=3`, `son`. İlk çağrıda `1 > 2` yanlıştır ve `c` normal biter. `c(3)`'te istisna doğar; `c bitti` yazılmaz. `b`'de `catch` yoktur ama `finally` vardır: istisna yoluna devam etmeden önce `b finally` yazılır. `b bitti` ve `main devam` atlanır.

6.6.7 JVM istisnayı nasıl ele alır?

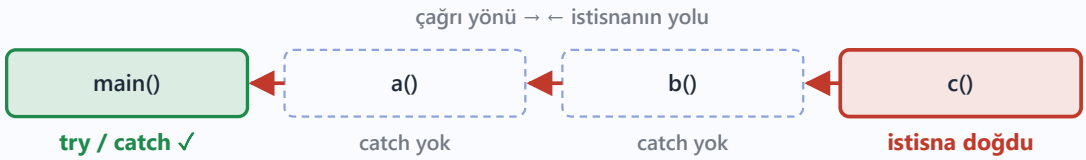
Yayılım izlemesinde görülen süreç her istisnada aynı adımlarla işler:

1. JVM ya da bir `throw` deyimi bir istisna nesnesi oluşturur. Nesne tipi, mesajı ve oluştuğu yerin yığın izini içerir.
2. İstisnanın oluştuğu metotta, onu saran bir `try` ve uyumlu bir `catch` aranır.

3. Bulunamazsa metot o anda sona erer; varsa `finally` bloğu çalışır. İstisna çağıran metoda geçer ve arama orada sürer.
4. Uygun bir `catch` bulunursa o çalışır; program `catch`'ten sonraki satırdan devam eder.
5. `main` de yakalamazsa JVM'in varsayılan işleyicisi yığın izini yazdırır ve programı, daha kesin söylemek gerekirse o iş parçacığını sonlandırır.

Tanım 6.9

Uygun `catch`'i bulmak için çağrı yığınında aşağı doğru ilerleyen ve yol üzerindeki çerçeveleri atan bu sürece **yığın çözülmesi** (*stack unwinding*) denir.



Şekil 6.13: Yığın çözülmesi. Çağrılar soldan sağa yapıldı: `main` → `a` → `b` → `c`. İstisna `c`'de doğar ve ters yönde, `catch` bulana kadar ilerler.

Yakalanan istisna nesnesi sıradan bir nesnedir. `catch` bloğu bittikten sonra ona başka bir referans tutulmuyorsa nesne artık kullanılmaz. Çöp toplayıcı (*garbage collector*) onu bir süre sonra bellekten temizler.

6.7 Çalışma zamanı istisnaları

6.7.1 `RuntimeException` ailesi

`RuntimeException` ve alt sınıfları, programcı hatası ya da mantık hatası sonucu oluşan istisnaları temsil eder. Derleme zamanında denetlenmezler ve ele alınmaları zorunlu değildir. Ama bu, onları görmezden gelmek anlamına gelmez. Programcı bu durumları bilinçli olarak önler ya da ele alır.

Sınıf	Ne zaman oluşur?	Örnek
`NullPointerException`	`null` bir referansın üyesi kullanılınca	`String s = null; s.length();`
`ArrayIndexOutOfBoundsException`	Dizi sınırları dışında indis	`int[] d = new int[3]; d[5] = 1;`
`ArithmeticException`	Tam sayıyı sıfıra bölme, kesin aritmetikte taşma	`10 / 0`
`IllegalArgumentException`	Metoda geçersiz argüman verilince	`new ArrayList<>(-1)`
`NumberFormatException`	Metin geçerli bir sayı değilken dönüştürme	`Integer.parseInt("abc")`

Sınıf	Ne zaman oluşur?	Örnek
`ClassCastException`	Nesne, dönüştürülmek istenen tipte değilken	`(Integer) nesne`

Tablo 6.12: Sık karşılaşılan `RuntimeException` sınıfları.

Sunumdaki "satır satır" örneklerde her satır *tek başına* denendiğinde ne olduğu verilir. Gerçek bir programda ilk istisna akışı keser ve sonraki satırlar çalışmaz. Notta verilen bütün iletiler JDK 21 ile her satır ayrı bir `try` içinde çalıştırılarak alınmıştır.

6.7.2 `NullPointerException`

`null` bir referans hiçbir nesneyi göstermez. Bu referans üzerinden metot çağırmak, bir alana erişmek ya da dizinin `length` değerini okumak `NullPointerException`'a (kısaca NPE) yol açar. Bu istisna en sık karşılaşılan hata kaynaklarındandır.

```
String str = null;
int length = str.length();
```

```
Exception in thread "main" java.lang.NullPointerException:
  Cannot invoke "String.length()" because "str" is null
```

Java 14'ten önce bu ileti yalnız `java.lang.NullPointerException` idi ve hangi referansın `null` olduğunu söylemezdi. Aynı satırda birkaç referans kullanılıyorsa (`a.b().c()`) suçluyu bulmak zordu. Java 14'ten beri açıklayıcı ileti hangi işlemin neden yapılamadığını söyler. Değişken adı ise yalnız sınıf hata ayıklama bilgisiyle derlendiğinde görünür (`javac -g`; IDE'ler bunu varsayılan olarak yapar). Düz `javac` ile derlenen aynı program adı bilemez ve şunu yazar:

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()"
because "<local1>" is null
```

NPE'yi önlemenin birkaç yolu vardır:

- `if (str != null)` ile açık denetim.
- `Objects.requireNonNull(x, mesaj)`: `x` `null` ise hemen, anlamlı bir mesajla NPE fırlatır.
- `Objects.requireNonNullElse(x, varsayilan)` (Java 9+): `x` `null` ise varsayılan değeri döndürür.
- `Optional` sınıfı (Java 8+): "değer olmayabilir" durumunu bir tiple açıkça temsil eder.
- Kütüphaneler: örneğin Apache Commons Lang'in `StringUtils` sınıfı `null`'a dayanıklı metin işlemleri sunar.

Aşağıdaki kod bu yolları bir arada gösterir.

```
1  String str = null;
2  int length = str.length();
3  Objects.requireNonNull(str, "Str null olamaz");
4  Optional<String> optionalStr = Optional.ofNullable(str);
5  length = optionalStr.map(String::length).orElse(0);
6  String defaultValue = Objects.requireNonNullElse(str, "0");
7  if (str != null) {
8    System.out.println(str.length());
9  }
10 System.out.println("Ali".equals(str));
```

Satır	Sonuç
2	`NullPointerException: Cannot invoke "String.length()" because "str" is null`
3	`NullPointerException: Str null olamaz`
5	`length = 0` : değer yoktur, `orElse(0)` devreye girer
6	`defaultValue = "0"`
7–9	Koşul yanlış; hiçbir şey yazılmaz, istisna yoktur
10	`false` : sabit metin öndedir, `null` argüman sorun değildir

Tablo 6.13: Satırların sonuçları (her satır tek başına denendiğinde; `str` her durumda `null`).

3. satır ile 2. satır aynı istisnayı fırlatır; o hâlde `requireNonNull` ne kazandırır? İki şey. Birincisi, mesajı programcı belirler. İkincisi ve daha önemlisi, hata kaynağında yakalanır. `requireNonNull` metot başlarında kullanılır. Bir yapıcı, aldığı parametre `null` ise nesneyi hiç oluşturmaz. Böylece bozuk bir nesne programın içinde dolaşıp çok sonra, ilgisiz bir yerde NPE vermez.

5. satır `Optional` 'ı kullanır. `Optional.ofNullable(str)` , `str` `null` ise boş bir kutu, değilse içinde `str` olan bir kutu üretir. `map(String::length)` kutu doluysa içindekinin uzunluğunu alır, boşsa hiçbir şey yapmaz. `orElse(0)` , kutu boşsa 0 verir. Sonuç bir NPE değil, 0'dır.

10. satır yaygın bir alışkanlığı gösterir. `"Ali".equals(str)` güvenlidir, çünkü metot sabit metin üzerinden çağrılır ve sabit metin hiçbir zaman `null` değildir. `str.equals("Ali")` ise `str` `null` olduğunda NPE fırlatır.

Sık yapılan hata

NPE'yi `catch (NullPointerException e)` ile yakalayıp geçmek. NPE neredeyse her zaman bir programlama hatasıdır: bir değişkene değer atanmamış ya da bir metot beklenmedik biçimde `null` döndürmüştür. Doğru çözüm `null` 'ın nereden geldiğini bulup orayı düzeltmektir. Yakalamak yalnız belirtiyi gizler.

6.7.3 Sınır aşımı istisnaları

Bir dizinin geçerli indisleri `0`'dan `length - 1`'e kadardır. Negatif indis de geçersizdir. Bu sınırların dışındaki bir indise erişmek `ArrayIndexOutOfBoundsException` fırlatır.

Bu sınıf, `IndexOutOfBoundsException` ailesinin bir üyesidir. Sınır aşımında hangi sınıfın fırlatılacağı, erişilen yapıya bağlıdır:

- dizi (`int[]`), `Vector` ve `Arrays.asList(...)` listesi → `ArrayIndexOutOfBoundsException`;
- `ArrayList.get` → `IndexOutOfBoundsException` (üst sınıfın kendisi);
- `String.charAt` → `StringIndexOutOfBoundsException`.

Hepsini birden yakalamak için ortak üst sınıf yazılır: `catch (IndexOutOfBoundsException e)`.

```
1  int[] numbers = {1, 2, 3};
2  int value = numbers[5];
3  ArrayList<String> list = new ArrayList<>();
4  String svalue = list.get(5);
5  Vector<Integer> vector = new Vector<>();
6  value = vector.get(10);
7  String[] names = {"Alice", "Bob", "Charlie"};
8  svalue = Arrays.asList(names).get(5);
9  String str = "Hello";
10 char ch = str.charAt(10);
```

Satır	İstisna	Mesaj
2	`ArrayIndexOutOfBoundsException`	`Index 5 out of bounds for length 3`
4	`IndexOutOfBoundsException`	`Index 5 out of bounds for length 0`
6	`ArrayIndexOutOfBoundsException`	`Array index out of range: 10`
8	`ArrayIndexOutOfBoundsException`	`Index 5 out of bounds for length 3`
10	`StringIndexOutOfBoundsException`	`Index 10 out of bounds for length 5`

Tablo 6.14: Sınır aşımında fırlatılan istisnalar (her satır tek başına denendiğinde).

4. satır özellikle öğreticidir. Liste `new ArrayList<>(20)` gibi bir kapasiteyle oluşturulsa bile boyutu 0'dır, çünkü henüz hiç öğe eklenmemiştir. `get` metodu indisi kapasiteye göre değil, boyuta göre denetler. 6. satırdaki `Vector`'ün mesajı farklı biçimdedir, çünkü bu eski sınıf mesajını kendisi oluşturur. 8. satırdaki liste, `names` dizisinin üzerine kurulmuş bir görünümdür; erişim doğrudan diziye yapılır ve dizinin istisnası fırlatılır.

Sınır aşımının en sık nedeni döngü koşulundaki tek bir karakterdir. Doğru döngü `length` ile sınırlanır:

```
int[] numbers = {1, 2, 3};
for (int i = 0; i < numbers.length; i++) {
  System.out.print(numbers[i] + " ");
}
```

```
1 2 3
```

Koşul `i <= numbers.length` yazılırsa döngü bir tur fazla döner. Son turda `numbers[3]`'e erişilir ve `Index 3 out of bounds for length 3` iletili istisna oluşur. Dizideki her öğe üzerinde gezinmek için en güvenli yazım for-each döngüsüdür; orada indis hiç yazılmaz.

6.7.4 `ArithmeticException`

`ArithmeticException`, matematiksel bir işlem sırasında oluşan hatayı temsil eder. En bilinen örneği tam sayıyı sıfıra bölmektir; hem `/` hem `%` işleci bu istisnayı fırlatır. Başka kaynakları da vardır:

- `Math` sınıfının "kesin" (*exact*) metotları taşmada fırlatır: `addExact`, `multiplyExact`, `toIntExact` ...
- `BigDecimal` yüksek duyarlıklı aritmetik sağlar. Sıfıra bölmede ve sonucu tam gösterilemeyen bölmede (1/3) fırlatır.

Bazı işlemler ise beklenenin aksine istisna fırlatmaz:

- `double` işlemleri istisna fırlatmaz: `10.0 / 0` → `Infinity`, `Math.sqrt(-1)` → `NaN`.
- `Random.nextInt(0)`, `ArithmeticException` değil, `IllegalArgumentException` fırlatır: sınır pozitif olmalıdır.
- Sıradan `int` toplaması taşınca istisna oluşmaz; değer başa sarar. `Integer.MAX_VALUE + 1` → `-2147483648`. Buna sessiz taşma denir. Taşmanın fark edilmesi gerekiyorsa `Math.addExact` kullanılır.

```
 1  int result = 10 / 0;
 2  double dresult = Math.sqrt(-1);
 3  System.out.println(dresult);
 4  System.out.println(10.0 / 0);
 5  int t = Math.addExact(Integer.MAX_VALUE, 1);
 6  Random random = new Random();
 7  int randomNumber = random.nextInt(0);
 8  BigDecimal num1 = new BigDecimal("10");
 9  BigDecimal num2 = BigDecimal.ZERO;
10  BigDecimal bresult = num1.divide(num2);
```

Satır	Sonuç
1	`ArithmeticException: / by zero`
3	`NaN` yazar, istisna yoktur
4	`Infinity` yazar, istisna yoktur
5	`ArithmeticException: integer overflow`
7	`IllegalArgumentException: bound must be positive`
10	`ArithmeticException: Division by zero`

Tablo 6.15: Aritmetik hatalar (her satır tek başına denendiğinde).

Kayan noktalı sayılarda sıfıra bölme, IEEE 754 standardının kurallarına göre özel değerler üretir. Pozitif bir sayının sıfıra bölümü `Infinity`, negatif bir sayınınki `-Infinity`, `0.0 / 0` ise `NaN`'dır (*Not a Number*, sayı değil). Bu değerler sessizce hesaplara karışır ve sonraki her işlemi bozar. Bu yüzden `double` sonuçları gerektiğinde ayrıca denetlenir: `Double.isNaN(x)`, `Double.isInfinite(x)`.

`BigDecimal` için 1/3 örneği de çalıştırılarak denendi. `BigDecimal.ONE.divide(new BigDecimal("3"))` şu istisnayı verir: `Non-terminating decimal expansion; no exact representable decimal result.` Sonsuz ondalık açılım tam olarak saklanamaz. Çözüm, bölmeye bir ölçek ve yuvarlama kuralı vermektir.

Örnek 6.8

`10 / 0`, `10 % 0`, `Math.toIntExact(3000000000L)` ve `10.0 / 0` ifadelerinden hangisi çalışırken istisna fırlatmaz?

`10.0 / 0`. Kayan noktalı bölme IEEE 754 kurallarına uyar ve `Infinity` üretir. Tam sayı bölmesi ve kalan işlemi sıfırla `ArithmeticException: / by zero` verir. 3.000.000.000 değeri `int` aralığına (en çok 2.147.483.647) sığmadığı için `toIntExact` `integer overflow` fırlatır.

6.7.5 `IllegalArgumentException`

`IllegalArgumentException`, bir metodun geçersiz bir argüman aldığını belirtir. Kontrol edilmeyen bir istisnadır. Bir metodun ilk işi argümanlarını denetleyip gerekiyorsa bu istisnayı fırlatmaktır. Böylece hata kaynağında durdurulur.

```
public void yasAyarla(int age) {
  if (age < 0) {
    throw new IllegalArgumentException("Yaş negatif olamaz");
  }
  this.age = age;
}
public void adAyarla(String name) {
  if (name.length() < 3) {
    throw new IllegalArgumentException("3 karakter olmalı");
  }
  this.name = name;
}
```

Java kütüphanesi de bu istisnayı yaygın biçimde kullanır:

- `Arrays.copyOfRange(d, 5, 2)` : başlangıç bitişten büyük;
- `new ArrayList<>(-1)` : negatif kapasite;
- `Thread` 'in `setPriority(11)` metodu: öncelik 1 ile 10 arasında olmalı;
- `new File(uri)` : adres `file:` biçiminde değil.

```
1  yasAyarla(-5);
2  adAyarla("Al");
3  int[] numbers = {1, 2, 3};
4  Arrays.copyOfRange(numbers, 5, 2);
5  List<Integer> liste = new ArrayList<>(-1);
6  File file = new File(URI.create("http://ornek.com/a.txt"));
7  Thread thread = new Thread(gorev);
8  thread.setPriority(11);
```

Satır	Fırlatılan istisna
1	`IllegalArgumentException: Yaş negatif olamaz`
2	`IllegalArgumentException: 3 karakter olmalı`
4	`IllegalArgumentException: 5 > 2`
5	`IllegalArgumentException: Illegal Capacity: -1`
6	`IllegalArgumentException: URI scheme is not "file"`
8	`IllegalArgumentException` (mesajsız)

Tablo 6.16: Geçersiz argüman örnekleri (her satır tek başına denendiğinde).

Her geçersiz durum bu istisnayı vermez. `new FileReader("invalid/file/path")` , kontrol edilen `FileNotFoundException` fırlatır. `new Thread(null)` ise hiç fırlatmaz: `null` görev, "yapılacak iş yok" anlamına gelir ve iş parçacığı başlatılınca hemen biter. `IllegalArgumentException` 'ın en tanınmış alt sınıfı `NumberFormatException` 'dır.

İyi bir mesaj, geçersiz değeri ve beklenen aralığı söyler: `"Yaş negatif olamaz: " + age`. 8. satırdaki mesajsız istisna hata ayıklamayı zorlaştırır. İletiyi okuyan programcı, önceliğin kaç verildiğini ve kaç olması gerektiğini bilemez.

> **Sık yapılan hata**
>
> Argüman denetiminde `null`'ı unutmak. `adAyarla(null)` çağrısında `name.length()` denetimin kendisi NPE fırlatır; programcının yazdığı anlamlı mesaj hiç görünmez. Denetim sırası önce `null`, sonra değerdir: `Objects.requireNonNull(name, "ad null olamaz")`, ardından uzunluk denetimi.

6.7.6 `NumberFormatException` ve güvenli dönüştürme

Bir metin sayıya dönüştürülürken metin geçerli bir sayıyı temsil etmiyorsa `NumberFormatException` oluşur. `Integer.parseInt`, `Double.parseDouble`, `Float.parseFloat`, `Short.parseShort`, `Long.parseLong`, `Byte.parseByte` ve bunların `valueOf` sürümleri bu istisnayı fırlatabilir. Değer tipin aralığı dışındaysa da aynı istisna oluşur: `Byte.parseByte("300")` şu iletiyi verir: `Value out of range. Value:"300" Radix:10`.

```
String strNumber = "abc";
int number = Integer.parseInt(strNumber);
strNumber = "123abc";
number = Integer.parseInt(strNumber);
String strDouble = "12.34abc";
double doubleValue = Double.parseDouble(strDouble);
String strShort = "45xyz";
short shortValue = Short.parseShort(strShort);
```

Çağrı	Mesaj
`Integer.parseInt("abc")`	`For input string: "abc"`
`Integer.parseInt("123abc")`	`For input string: "123abc"`
`Double.parseDouble("12.34abc")`	`For input string: "12.34abc"`
`Short.parseShort("45xyz")`	`For input string: "45xyz"`

Tablo 6.17: Dönüştürme çağrıları ve `NumberFormatException` mesajları.

`"123abc"` örneği dönüştürmenin “her şey ya da hiç” kuralını gösterir. Metnin başındaki 123 geçerli olsa da metin bir bütün olarak sayı değildir. `parseInt` yarım sonuç döndürmez. `NumberFormatException`, `IllegalArgumentException`'ın alt sınıfıdır. Kullanıcı girdisi her zaman bu riski taşır: girdi ya önceden denetlenir ya da istisna yakalanır.

Dönüştürme bir yardımcı metotta, tek bir yerde denetlenebilir. Geçersiz girdi için metot bir varsayılan değer döndürür.

```
static int guvenliOku(String metin, int varsayilan) {
  try {
    return Integer.parseInt(metin.trim());
  } catch (NumberFormatException e) {
    return varsayilan;
  }
}
// main içinde
System.out.println(guvenliOku(" 42 ", 0));
System.out.println(guvenliOku("kırk", 0));
System.out.println(guvenliOku("10.5", -1));
```

```
42
0
-1
```

`trim()`, baştaki ve sondaki boşlukları siler. Bu sayede `" 42 "` girdisi 42'ye dönüşür. `parseInt(" 42 ")` tek başına istisna fırlatırdı, çünkü boşluk bir rakam değildir. `"kırk"` bir sayı değildir; varsayılan 0 döner. `"10.5"` de bir `int` değildir; ondalık sayı için `Double.parseDouble` kullanılır.

Sunumdaki adım adım izleme aynı metodu üç yeni girdiyle çalıştırır.

```
static int guvenliOku(String metin, int varsayilan) {
  try {
    return Integer.parseInt(metin.trim());
  } catch (NumberFormatException e) {
    return varsayilan;
  }
}
System.out.println(guvenliOku(" 7 ", 0));
System.out.println(guvenliOku("yedi", 0));
System.out.println(guvenliOku("7.5", -1));
```

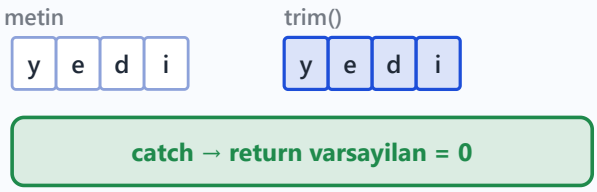
```
7
0
-1
```

Adım	Satır	Değişkenler	Yeni çıktı
1	8: `System.out.println(guvenliOku(" 7 ", 0));`	—	—
2	3: `return Integer.parseInt(metin.trim());`	metin=" 7 ", varsayilan=0	`7`

Adım	Satır	Değişkenler	Yeni çıktı
3	9: `System.out.println(guvenliOku("yedi", 0));`	—	—
4	3: `return Integer.parseInt(metin.trim());`	metin="yedi", varsayilan=0	—
5	4: `} catch (NumberFormatException e) {` 5: `return varsayilan;`	metin="yedi", varsayilan=0, e=NumberFormatException	`0`
6	10: `System.out.println(guvenliOku("7.5", -1));`	—	—
7	3: `return Integer.parseInt(metin.trim());`	metin="7.5", varsayilan=-1	—
8	4: `} catch (NumberFormatException e) {` 5: `return varsayilan;`	metin="7.5", varsayilan=-1, e=NumberFormatException	`-1`

Tablo 6.18: `guvenliOku` metodunun üç girdiyle izlemesi.

İlk çağrıda metin `" 7 "`'dir ve başta ve sonda birer boşluk vardır. `trim()` boşlukları atar, `"7"` sayıya çevrilir ve `try`'ın içinden 7 döner. İkinci çağrıda metin `"yedi"`'dir. `parseInt` rakam olmayan metinde istisna fırlatır ve `return` deyimi tamamlanamaz.



Şekil 6.14: İkinci çağrı. `trim()` sonucu yine `yedi`'dir; dönüşüm başarısız olur ve `catch` bloğu varsayılan değeri, 0'ı döndürür.

`catch` eşleşir ve varsayılan değer 0 döner. İstisna metodun dışına çıkmaz. Üçüncü çağrıda metin `"7.5"`, varsayılan değer −1'dir. Nokta bir rakam değildir; `"7.5"` geçerli bir `int` değildir. −1 döner. Çağıran kod, istisna yerine varsayılan değerle yoluna devam eder.

Bu yaklaşımın bir bedeli vardır. Varsayılan değer, geçerli bir sonuçla karışmayacak biçimde seçilmelidir. `guvenliOku("0", 0)` 0 döndürür, tıpkı `guvenliOku("yedi", 0)` gibi; `guvenliOku("-1", -1)` de −1 döndürür, tıpkı `guvenliOku("7.5", -1)` gibi. Varsayılan değer geçerli girdilerden ayırt edilemiyorsa istisnayı çağırana iletmek daha doğrudur. Bu, dönüş kodu tartışmasının küçük bir tekrarıdır.

6.7.7 `ClassCastException` ve `instanceof`

Bir nesne, dönüştürülmek istendiği tipten değilken açık dönüşüm (*cast*) yapılırsa `ClassCastException` oluşur. Belirleyici olan değişkenin tipi değil, nesnenin gerçek tipidir. `Object obj = "Merhaba";` satırından sonra `obj`'nin tipi `Object`'tir, ama gösterdiği nesne bir `String`'dir.

Ham (*raw*) tipli koleksiyonlar, yani tip parametresi yazılmamış `ArrayList`, `Map` ve `Set`, her türden nesneyi kabul eder. Yanlış tipte bir öğe alınıp dönüştürülürken dönüşüm başarısız olur. Sıralı koleksiyonlar (`TreeSet`, `TreeMap`) ise birbiriyle karşılaştırılamayan tipte öğe eklenirken de bu istisnayı fırlatır.

```
 1  Object obj = "Merhaba";
 2  Integer number = (Integer) obj;
 3  ArrayList list = new ArrayList();
 4  list.add("Merhaba");
 5  number = (Integer) list.get(0);
 6  Map map = new HashMap();
 7  map.put("anahtar", "Değer");
 8  Integer value = (Integer) map.get("anahtar");
 9  Set set = new HashSet();
10  set.add("Merhaba");
11  value = (Integer) set.iterator().next();
12  Object[] objArray = new String[5];
13  Integer[] intArray = (Integer[]) objArray;
```

2, 5, 8, 11 ve 13. satırların her biri `ClassCastException` fırlatır. Örneğin 2. satırın iletisi şudur:

```
class java.lang.String cannot be cast to class
java.lang.Integer
```

JDK 21'in tam iletisi bir parantez daha ekler: iki sınıfın da `java.base` modülünde ve aynı yükleyicide olduğunu söyler. 13. satır dizi tipleri arasındaki dönüşümdür. `objArray` gerçekte bir `String[]`'dir ve `Integer[]`'e dönüştürülemez; iletide diziler `[Ljava.lang.String;` biçiminde görünür.

Dönüşümden önce `instanceof` ile nesnenin tipi sınanabilir:

```
Object obj = "Merhaba";
if (obj instanceof Integer) {
  Integer sayi = (Integer) obj;
  System.out.println("Sayı: " + sayi);
} else {
  System.out.println("obj bir Integer değil");
}
```

```
obj bir Integer değil
```

Java 16'dan beri örüntü eşleme (*pattern matching*) sınama ile dönüşümü birleştirir. Koşul doğruysa `sayi` değişkeni zaten `Integer` tipindedir; ayrı bir dönüşüm satırı gerekmez.

```
if (obj instanceof Integer sayi) {
  System.out.println("Sayı: " + sayi);
}
```

Daha iyi çözüm, sorunu derleme zamanına taşımaktır. Jenerikler (*generics*) bir koleksiyonun hangi tipte öğe tuttuğunu derleyiciye söyler.

```
ArrayList<String> liste = new ArrayList<>();
liste.add("Merhaba");
Integer n = (Integer) liste.get(0);
```

```
Ana.java:8: error: incompatible types: String cannot be converted to Integer
Integer n = (Integer) liste.get(0);
```

Çalışma zamanı istisnası yerine bir derleme hatası alınır. Hata daha erken, program hiç çalıştırılmadan bulunur. Bu yüzden ham tipler yerine jenerik tipler tercih edilir.

6.8 Kontrol edilen istisna sınıfları

6.8.1 Sık karşılaşılan sınıflar

Kontrol edilen istisnalar, kodun hatayı ele almasını ve uygun biçimde kurtarmasını gerektiren durumları temsil eder. Bu durumlar çoğunlukla programın dışındaki koşullardan doğar: dosya sistemi, ağ, veri tabanı, başka bir iş parçacığı. Derleyici bu istisnaların ele alınmasını şart koşar.

Sınıf	Paket	Ne zaman?
`FileNotFoundException`	`java.io`	Açılmak istenen dosya yok ya da açılamıyor
`IOException`	`java.io`	Giriş/çıkış işlemlerinde genel hata (üst sınıf)
`SQLException`	`java.sql`	Veri tabanı erişiminde hata
`ParseException`	`java.text`	Metni bir biçime göre ayrıştırırken hata
`ClassNotFoundException`	`java.lang`	Adıyla yüklenmek istenen sınıf bulunamadı
`InterruptedException`	`java.lang`	Bekleyen iş parçacığı kesildi (`Thread.sleep`)
`NotSerializableException`	`java.io`	Serileştirilemeyen bir nesne yazılmak istendi

Tablo 6.19: Sık karşılaşılan kontrol edilen istisnalar.

Tablodaki sınıfların hepsi `Exception`'dan türer ve zincirlerinde `RuntimeException` yoktur. `ClassNotFoundException` araya bir düzey daha alır (`ReflectiveOperationException`); `NotSerializableException` ise `ObjectStreamException` üzerinden `IOException`'a bağlanır.

6.8.2 `FileNotFoundException` ve `IOException`

`FileNotFoundException`, bir dosya açılmak istendiğinde dosya bulunamazsa oluşur. Aynı istisna, verilen yol bir klasörse ya da dosyayı okuma izni yoksa da fırlatılır.

```
try {
  FileReader fileReader = new FileReader("dosya.txt");
} catch (FileNotFoundException e) {
  // Dosya bulunamadı durumuyla ilgili işlemler
  System.out.println(e);
}
```

```
java.io.FileNotFoundException: dosya.txt
  (The system cannot find the file specified)
```

Bu sınıf `IOException`'ın alt sınıfıdır; `catch (IOException e)` onu da yakalar. Göreli bir yol (`"dosya.txt"`), programın çalışma klasörüne göre aranır. Geliştirme ortamlarında çalışma klasörü genellikle proje klasörüdür, kaynak kodun bulunduğu klasör değil. Dosya "yanlış klasörde" olabilir. Bu istisnaya tipik tepki, kullanıcıya durumu bildirip yeni bir ad istemek ya da varsayılan değerlerle devam etmektir.

Aynı istisnanın klasör için verdiği ileti de denendi: `new FileReader(".")` Windows'ta `java.io.FileNotFoundException: . (Access is denied)` verir. Sınıfın adı "dosya bulunamadı" dese de istisna "dosya açılamadı" anlamında kullanılır.

`IOException`, giriş/çıkış işlemlerinde oluşan genel hatayı temsil eder ve `FileNotFoundException` gibi birçok alt sınıfı vardır. Aşağıdaki kod bir metin dosyasını satır satır okur.

```
try {
  BufferedReader reader =
    new BufferedReader(new FileReader(path));
  String line;
  while ((line = reader.readLine()) != null) {
    System.out.println(line);
  }
  reader.close();
} catch (IOException e) {
  // IOException durumuyla ilgili işlemler
  e.printStackTrace();
}
```

```
birinci satır
ikinci satır
```

Kod çalışır, ama bir kusuru vardır. `readLine()` bir istisna fırlatırsa denetim doğrudan `catch`'e geçer ve `reader.close()` satırı hiç çalışmaz. Dosya açık kalır: bu bir kaynak sızıntısıdır. Çözüm try-with-resources'tır:

```
try (BufferedReader okuyucu =
    new BufferedReader(new FileReader(path))) {
  String satir;
  while ((satir = okuyucu.readLine()) != null) {
    System.out.println(satir);
  }
} catch (IOException e) {
  System.out.println(e.getMessage());
}
```

Bu sürümde okuyucu, blok nasıl biterse bitsin kapatılır. Dosya yoksa `new FileReader(path)` istisna fırlatır; o durumda kapatılacak bir kaynak da oluşmamıştır ve `catch` doğrudan çalışır. `catch (IOException e)` hem açılma hatasını (`FileNotFoundException`) hem de okuma hatalarını yakalar. Var olmayan `yok.txt` ile denendiğinde blok, `yok.txt (The system cannot find the file specified)` mesajını yazar.

6.8.3 `SQLException`

`SQLException`, veri tabanı işlemleri sırasında bir hata oluştuğunda fırlatılır. Bağlantı kurulamayabilir, SQL söz dizimi yanlış olabilir, tablo bulunmayabilir ya da veri tabanı sürücüsü yüklenmemiş olabilir. İstisna nesnesi ayrıntı veren üç metot sunar: `getMessage()`, `getSQLState()` ve `getErrorCode()`.

```
try {
  Connection connection =
    DriverManager.getConnection("jdbc:mysql://localhost/okul");
  PreparedStatement preparedStatement =
    connection.prepareStatement("SELECT user_id FROM users");
  // Sorguyu çalıştırma ve sonuçları alıp işleme
  ResultSet resultSet = preparedStatement.executeQuery();
  while (resultSet.next()) {
    // Sonuçları işleme
    System.out.println("User ID: " + resultSet.getInt("user_id"));
  }
  // Bağlantıyı kapatma
  connection.close();
} catch (SQLException e) {
  // SQLException durumuyla ilgili işlemler
  System.out.println(e);
}
```

```
java.sql.SQLException: No suitable driver
found for jdbc:mysql://localhost/okul
```

Bu program MySQL sürücüsü sınıf yolunda yokken çalıştırılmıştır. `DriverManager`, `jdbc:mysql:` ile başlayan adresi tanıyan bir sürücü bulamaz. Bu durumda `getSQLState()` `08001` değerini, `getErrorCode()` ise 0 döndürür. `08` ile başlayan durum kodları bağlantı hatalarını anlatan standart kodlardır.

Koddaki `connection.close()` çağrısı, `IOException` örneğindeki kusuru taşır: sorgu sırasında bir istisna oluşursa bağlantı açık kalır. Bağlantı da bir kaynaktır ve try-with-resources ile kapatılır. `Connection`, `PreparedStatement` ve `ResultSet` üçü de `AutoCloseable`'dır ve aynı parantezde açılabilir.

6.8.4 `ParseException` ve `ClassNotFoundException`

`ParseException` (`java.text` paketi), bir metin belirli bir biçime göre ayrıştırılırken uyuşmazlık olursa fırlatılır.

```
try {
  SimpleDateFormat form =
    new SimpleDateFormat("dd/MM/yy");
  Date date = form.parse("01-01-2022");
} catch (ParseException e) {
  // Ayrıştırma hatasıyla ilgili işlemler
  System.out.println(e);
}
```

```
java.text.ParseException: Unparseable date: "01-01-2022"
```

Biçim `dd/MM/yy`'dir; gün ile ay arasında `/` beklenir. Metinde ise `-` vardır. Ayrıştırıcı ilk iki karakteri gün olarak okur ve üçüncü karakterde takılır. `e.getErrorOffset()` bu konumu verir: 2, yani üçüncü karakter.

`ClassNotFoundException`, bir sınıf adıyla yüklenmek istendiğinde (`Class.forName`) sınıf bulunamazsa fırlatılır.

```
try {
  Class.forName("java.util.Scanner");
  System.out.println("Scanner yüklendi");
  // Var olmayan bir sınıfı yüklemeye çalışma
  Class.forName("com.example.NonExistentClass");
} catch (ClassNotFoundException e) {
  System.out.println(e);
}
```

```
Scanner yüklendi
java.lang.ClassNotFoundException:
  com.example.NonExistentClass
```

`java.util.Scanner` standart kütüphanededir ve sorunsuz yüklenir. `com.example.NonExistentClass` ise sınıf yolunda yoktur. Sınıfı adıyla yüklemek, veri tabanı sürücüleri gibi derleme sırasında bilinmeyen sınıflarla çalışırken kullanılır. Ad bir metin olduğu için derleyici onu denetleyemez; sınıfın bulunup bulunmayacağı ancak çalışırken belli olur.

Bu istisna, Error ailesindeki `NoClassDefFoundError` ile karıştırılmamalıdır. `ClassNotFoundException`'da program bir sınıfı adıyla kendisi ister. `NoClassDefFoundError`'da ise derleme sırasında var olan bir sınıf, program çalışırken bulunamamıştır; bu bir kurulum sorunudur.

6.8.5 `NotSerializableException`

Tanım 6.10

Serileştirme (*serialization*), bir nesnenin bayt akışına dönüştürülmesidir. Bu akış bir dosyaya yazılabilir ya da ağ üzerinden gönderilebilir. Daha sonra akış yeniden nesneye dönüştürülür.

Java'da bir sınıfın nesnelerinin serileştirilebilmesi için sınıfın `Serializable` arayüzünü uygulaması gerekir. Bu arayüzün hiçbir metodu yoktur; yalnız bir izin işaretidir. Arayüzü uygulamayan bir sınıfın nesnesi yazılmak istenirse `NotSerializableException` oluşur.

```
class Nokta {               // Serializable değil
  int x = 3, y = 4;
}
// main içinde
try (ObjectOutputStream outputStream =
    new ObjectOutputStream(new ByteArrayOutputStream())) {
  // Nesneyi ObjectOutputStream ile serileştirmeye çalışma
  Nokta notSerializableObj = new Nokta();
  outputStream.writeObject(notSerializableObj);
} catch (NotSerializableException e) {
  System.out.println(e);
} catch (IOException e) {
  System.out.println("Diğer G/Ç hatası: " + e);
}
```

```
java.io.NotSerializableException: Nokta
```

`NotSerializableException`, `IOException`'ın dolaylı bir alt sınıfıdır. Bu yüzden onun `catch`'i `IOException`'ınkinden önce yazılır. Sıra ters olsaydı derleyici `has already been caught` hatası verirdi. Çözüm, sınıfa arayüzü eklemektir:

```
class Nokta implements Serializable {
  int x = 3, y = 4;
}
```

Bu düzeltmeyle nesne yazılır ve geri okunabilir. Deneme programı nesneyi bir bayt dizisine yazıp yeniden okudu ve `x = 3`, `y = 4` değerlerini geri aldı.

6.9 `Error` sınıfları

`Error` sınıfları, programın normal çalışmasını temelden etkileyen, sistem düzeyindeki sorunları temsil eder. Kontrol edilmeyen tiplerdir. Genellikle uygulama kodunun bir satırından değil, JVM'in ve çalışma ortamının durumundan kaynaklanırlar. JVM'in kendisi fırlatır. Bu hataların ele alınması ya da bildirilmesi önerilmez; programın düzgün biçimde sonlanması daha uygun bir yaklaşımdır. Yine de `Throwable` oldukları için teknik olarak yakalanabilirler.

Hata	Ne zaman?
`OutOfMemoryError`	Bellek (yığın bellek, *heap*) tükendi
`StackOverflowError`	Çağrı yığını taştı; genellikle sonsuz özyineleme
`AssertionError`	`assert` koşulu yanlış (yalnız `-ea` açıkken)

Hata	Ne zaman?
`NoClassDefFoundError`	Derlemede var olan sınıf çalışırken bulunamadı

Tablo 6.20: Sık karşılaşılan `Error` sınıfları.



Şekil 6.15: Sonsuz özyinelemede çağrı yığını dolar. Taban durum olmadığı için hiçbir çağrı geri dönmez; her çağrı yeni bir çerçeve ekler ve yığın sınırına ulaşılınca `StackOverflowError` fırlatılır.

Sunumdaki program üç hatayı da gösterir. Burada `Error`'ları yakalamanın tek amacı onları ekranda görmektir.

```
static void sonsuz() {
  sonsuz();          // durma koşulu yok
}
public static void main(String[] args) {
  try {
    sonsuz();
  } catch (StackOverflowError e) {
    System.out.println("SOE -> " + e);
  }
  try {
    long[] dizi = new long[Integer.MAX_VALUE];
  } catch (OutOfMemoryError e) {
    System.out.println("OOM -> " + e);
  }
  int yas = -5;
  assert yas >= 0 : "Yaş negatif: " + yas;
  System.out.println("assert geçti");
}
```

Program iki biçimde çalıştırılır. `java Ornek` ile çıktı şudur:

```
SOE -> java.lang.StackOverflowError
OOM -> java.lang.OutOfMemoryError: Requested array size exceeds VM limit
assert geçti
```

`java -ea Ornek` ile ise son satır değişir:

```
SOE -> java.lang.StackOverflowError
OOM -> java.lang.OutOfMemoryError: Requested array size exceeds VM limit
Exception in thread "main" java.lang.AssertionError: Yaş negatif: -5
    at Ornek.main(Ornek.java:17)
```

Birinci `try`'da `sonsuz()` metodu kendisini durmadan çağırır. Özyinelemede taban durum unutulursa her çağrı yeni bir çerçeve ekler ve yığın dolar. İkinci `try`'da yaklaşık iki milyar öğeli bir `long` dizisi istenir. Bu dizi 16 GB'tan büyüktür; ayrıca HotSpot JVM'inde bir dizinin olabilecek en büyük boyu `Integer.MAX_VALUE`'dan biraz küçüktür. JVM isteği `Requested array size exceeds VM limit` iletisiyle reddeder.

Üçüncü kısım `assert` deyimidir. `assert koşul : mesaj;` koşul yanlışsa mesajla bir `AssertionError` fırlatır. Ama doğrulamalar (`assert`) varsayılan olarak kapalıdır; JVM onları hiç çalıştırmaz. Bu yüzden ilk çalıştırmada `assert geçti` yazılır. `-ea` (*enable assertions*) seçeneğiyle açıldığında koşul sınanır, `yas` −5 olduğu için hata fırlatılır ve program sonlanır. `assert` geliştirme sırasında programcının varsayımlarını sınamak içindir; kullanıcı girdisini denetlemek için kullanılmaz, çünkü kapatılabilir.

Örnek 6.9

`static int f(int n) { return f(n + 1); }` metodu `main`'de `try` içinde çağrılıyor; `catch (Exception e) yakalandı` yazıyor; ardından `son` yazılıyor. Program çalıştırılınca ne olur?

Ne `yakalandı` ne de `son` yazılır. Her çağrı yığına yeni bir çerçeve ekler; yığın dolunca JVM `StackOverflowError` fırlatır. Bu bir `Error`'dur, `Exception` ailesinden değildir. `catch (Exception e)` onu yakalamaz; program uzun bir yığın iziyle sonlanır. Listede aynı `at` satırı yüzlerce kez tekrarlanır. Derleyici sonsuz özyinelemeyi denetlemez; kod sorunsuz derlenir.

6.10 Özel istisnalar

6.10.1 Özel istisna sınıfı yazmak

Java kütüphanesindeki istisnalar genel durumları anlatır. Bir bankacılık programındaki “bakiye yetersiz” ya da bir not sistemindeki “not aralık dışında” durumu için kütüphanede özel bir sınıf yoktur. Geliştirici, kendi programında ortaya çıkabilecek bu tür durumları yönetmek için özel istisna sınıfları yazabilir.

Tanım 6.11

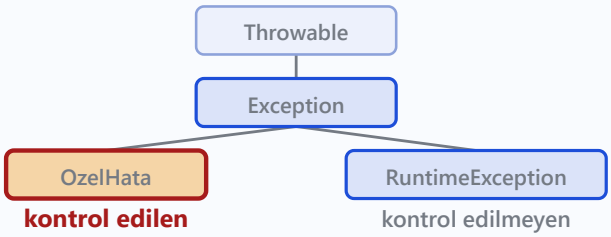
Özel istisna sınıfı (*custom exception*), `Exception` ya da `RuntimeException`'dan türetilen ve uygulamaya özgü bir durumu anlatan istisna sınıfıdır.

```
public class OzelHata extends Exception {
  // Constructor
  public OzelHata(String message) {
    super(message);
  }
}
```

Yapıcı, aldığı mesajı `super(message)` ile üst sınıfa iletir. Mesaj `Throwable`'ın içinde saklanır; böylece `getMessage()` ve `toString()` mesajı kendiliğinden gösterir. `super` çağrısı unutulursa `getMessage()` `null` döndürür.

Üst sınıf	Sonuç
`extends Exception`	Kontrol edilen: çağıran ele almak zorunda; kurtarılabilir iş kuralı ihlalleri
`extends RuntimeException`	Kontrol edilmeyen: hatalı kullanım, programcı hatası

Tablo 6.21: Özel istisnanın üst sınıfını seçmek.



Şekil 6.16: Özel istisna sınıflarının hiyerarşideki yeri. `OzelHata` `Exception`'ı genişlettiği için kontrol edilen bir istisnadır.

`Throwable`'ı genişleten her sınıfın nesnesi fırlatılabilir. Ancak genellikle `Exception` ya da `RuntimeException` genişletilir. `Error` JVM'in sorunları için ayrılmıştır. `Throwable`'ı doğrudan genişletmek ise sınıfı iki ailenin dışında bırakır; `catch (Exception e)` yazan kod onu yakalayamaz.

Özel istisna yazarken birkaç kural izlenir. Sınıfın adı durumu anlatır ve `…Hata` ya da `…Exception` ile biter: `YetersizBakiyeHatasi`, `GecersizNotHatasi`. İstisna sıradan bir sınıftır; ek alanlar ve metotlar içerebilir. Eksik tutar, hatalı değer ya da hatalı alanın adı gibi bilgiler alan olarak eklenebilir. Derleyici `-Xlint` ile çalıştırılırsa `serialVersionUID` alanının eksik olduğunu da söyler; `Throwable` serileştirilebilir bir sınıftır. Bu bir uyarıdır, hata değildir.

6.10.2 Özel istisnayı fırlatmak ve yakalamak

```
public class OzelHataTest {
  public static void main(String[] args) {
    try {
      fonksiyon();
    } catch (OzelHata e) {
      System.out.println("Yakalandı: " + e.getMessage());
    }
  }
  public static void fonksiyon() throws OzelHata {
    if (true) {
      throw new OzelHata("Özel hata oluştu.");
    }
  }
}
```

```
Yakalandı: Özel hata oluştu.
```

`OzelHata`, `Exception`'dan türediği için kontrol edilen bir istisnadır. `fonksiyon()`'un imzasında `throws OzelHata` zorunludur. `main`, çağrıyı bir `try` içine alarak "ele al" seçeneğini kullanır.

Koddaki `if (true)` yalnız gösterim içindir. `throw` burada metodun son deyimi olduğu için koşulsuz yazılması da derlenirdi. Ama koşulsuz bir `throw`'dan sonra herhangi bir kod gelseydi derleyici `unreachable statement` hatası verirdi. Gerçek programlarda koşul bir iş kuralını denetler: bakiye yeterli mi, not 0 ile 100 arasında mı, kullanıcı adı boş mu?

Örnek 6.10

Kontrol edilen bir `GecersizNotHatasi` sınıfı ve `int notGir(int not)` metodu yazalım. Not 0–100 dışındaysa istisna `"Not 0-100 arası olmalı: …"` mesajıyla fırlatılsın, geçerliyse not dönsün. 85, 120 ve 40 notları bir döngüde girilsin.

```
class GecersizNotHatasi extends Exception {
  GecersizNotHatasi(String m) { super(m); }
}
static int notGir(int not)
    throws GecersizNotHatasi {
  if (not < 0 || not > 100) {
    throw new GecersizNotHatasi(
        "Not 0-100 arası olmalı: " + not);
  }
  return not;
}
```

```
for (int g : new int[]{85, 120, 40}) {
  try {
    System.out.println("Kaydedildi: "
        + notGir(g));
  } catch (GecersizNotHatasi e) {
    System.out.println("Hata: "
        + e.getMessage());
  }
}
```

```
Kaydedildi: 85
Hata: Not 0-100 arası olmalı: 120
Kaydedildi: 40
```

Sınıf `Exception` 'ı genişlettiği için `throws` zorunludur. `try` döngünün içindedir: 120'nin hatası yalnız o turu etkiler ve 40'ın kaydedilmesini engellemez. `try` döngünün dışında olsaydı ilk hatada döngü de biterdi ve 40 hiç denenmezdi.

6.10.3 Bilgi taşıyan istisna: banka hesabı

Özel istisnanın asıl gücü, durumu anlatan bilgiyi taşıyabilmesidir. Aşağıdaki sınıf mesaja ek olarak eksik tutarı da saklar.

```
class YetersizBakiyeHatasi extends Exception {
  private double eksik;

  public YetersizBakiyeHatasi(String mesaj, double eksik) {
    super(mesaj);          // mesaj Throwable'da saklanır
    this.eksik = eksik;
  }

  public double getEksik() {
    return eksik;
  }
}
```

Bu sınıfı kullanan banka hesabı ve deneme kodu şöyledir:

```
class Hesap {
  private double bakiye;
  Hesap(double bakiye) { this.bakiye = bakiye; }
  public void paraCek(double miktar)
      throws YetersizBakiyeHatasi {
    if (miktar > bakiye) {
      throw new YetersizBakiyeHatasi(
        "Bakiye yetersiz", miktar - bakiye);
    }
    bakiye -= miktar;
    System.out.println("Kalan: " + bakiye);
  }
}
// main içinde
Hesap hesap = new Hesap(100);
try {
  hesap.paraCek(30);
  hesap.paraCek(90);
  hesap.paraCek(10);
} catch (YetersizBakiyeHatasi e) {
  System.out.println(e.getMessage()
    + ", eksik: " + e.getEksik());
}
```

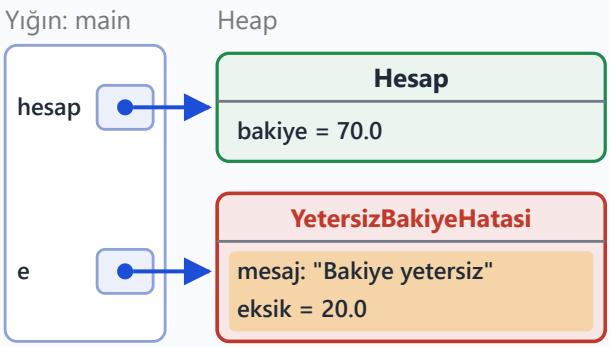
```
Kalan: 70.0
Bakiye yetersiz, eksik: 20.0
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	15: `Hesap hesap = new Hesap(100);`	main()	hesap=→Hesap	—
2	6: `if (miktar > bakiye) {`	main() → paraCek(30.0)	miktar=30.0, bakiye=100.0	—
3	10: `bakiye -= miktar;` 11: `System.out.println("Kalan: " + bakiye);`	main() → paraCek(30.0)	miktar=30.0, bakiye=70.0	`Kalan: 70.0`
4	6: `if (miktar > bakiye) {`	main() → paraCek(90.0)	miktar=90.0, bakiye=70.0	—
5	7: `throw new YetersizBakiyeHatasi(` 8: `"Bakiye yetersiz", miktar - bakiye);`	main() → paraCek(90.0)	miktar=90.0, bakiye=70.0	—
6	20: `} catch (YetersizBakiyeHatasi e) {`	main()	e=→YetersizBakiyeHatasi	—

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
7	21: `System.out.println(e.getMessage())` 22: `+ ", eksik: " + e.getEksik());`	main()	e=→YetersizBakiyeHatasi	`Bakiye yetersiz, eksik: 20.0`

Tablo 6.22: Banka hesabı örneğinin izlemesi.

1. adımda `Hesap` nesnesi yığın bellekte oluşur ve `hesap` onu gösterir; bakiye 100'dür. `paraCek(30)` çağrısında `30 > 100` yanlıştır; istisna yoktur. Bakiye 70 olur ve `Kalan: 70.0` yazılır. 4. adımda `paraCek(90)` çağrılır. Bu kez `90 > 70` doğrudur.



Şekil 6.17: 7. adım. Yığın bellekte iki nesne var: `Hesap` ve `YetersizBakiyeHatasi`. `catch` parametresi `e`, fırlatılan istisna nesnesini gösteriyor. Bakiye 70'te kaldı.

5. adımda yığın bellekte bir istisna nesnesi oluşur. Nesne iki bilgi taşır: `"Bakiye yetersiz"` mesajı ve `eksik = 90 - 70 = 20.0`. Nesne fırlatılır. `bakiye -= miktar` satırı çalışmadığı için bakiye değişmez. 6. adımda `paraCek`'in çerçevesi atılır ve `paraCek(10)` atlanır. `catch` eşleşir; `e` fırlatılan nesnenin aynısını gösterir. 7. adımda nesnenin bilgileri kullanılır: `Bakiye yetersiz, eksik: 20.0`.

Bu örnekte istisnanın da sıradan bir nesne olduğu açıkça görülür. `new` ile oluşturulur, alanları vardır, metotları çağrılır. Yakalayan kod bu bilgiyle anlamlı bir tepki verebilir. Örneğin kullanıcıya "20 TL eksik; yine de denemek ister misiniz?" diye sorabilir. `-1` gibi bir dönüş kodu bu kadar bilgi taşıyamazdı.

Bu sınıf için kontrol edilen tür seçilmiştir. Yetersiz bakiye öngörülebilir bir durumdur ve programcının hatası değildir. Para çekmek isteyen her kod bu olasılıkla başa çıkmalıdır; derleyici de bunu güvenceye alır.

Sık yapılan hata

Durumu denetlemeden önce durumu değiştirmek. `bakiye -= miktar;` satırı `if`'ten önce yazılsaydı istisna fırlatıldığında bakiye çoktan −20'ye düşmüş olurdu. Nesne bozuk bir durumda kalırdı. İyi bir metot önce bütün denetimleri yapar, istisna gerekiyorsa fırlatır ve ancak ondan

sonra nesnenin durumunu değiştirir.

6.11 Yığın izi ve iyi uygulamalar

6.11.1 Yığın izi nedir?

Tanım 6.12

Yığın izi (*stack trace*), istisnanın oluştuğu anda çağrı yığınında hangi metotların bulunduğunu ve istisnanın nerede oluştuğunu gösteren rapordur. İlk satır istisnanın tipini ve mesajını verir. Sonraki her `at` satırı bir çağrının sınıfını, metodunu, dosyasını ve satır numarasını verir.

Yakalanmayan bir istisnada JVM yığın izini hata akışına (`System.err`) yazdırır. `e.printStackTrace()` ile istenen her an da yazdırılabilir. Yığın izi, programcının hatayı anlamasına ve çözmesine yardım eden en önemli ipucudur.

```
Exception in thread "main" java.lang.NullPointerException
    at com.example.MyClass.myMethod(MyClass.java:10)
    at com.example.MyClass.main(MyClass.java:6)
```

Bu listede `main` metodu 6. satırda `myMethod`'u çağırmıştır. İstisna `myMethod`'un 10. satırında oluşmuştur. `com.example` paketin, `MyClass` sınıfın adıdır. Liste yukarıdan aşağıya "en son çağrılandan ilk çağrılana" doğru okunur.

Daha ayrıntılı bir örnek üzerinde okuma yöntemine bakalım.

```
 1  public class IzlemeOrnegi {
 2    static int uzunluk(String metin) {
 3      return metin.length();
 4    }
 5    static void yazdir(String ad) {
 6      System.out.println(ad + ": " + uzunluk(ad));
 7    }
 8    public static void main(String[] args) {
 9      yazdir("Ali");
10      yazdir(null);
11      System.out.println("Bitti");
12    }
13  }
```

```
Ali: 3
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()"
because "metin" is null
    at IzlemeOrnegi.uzunluk(IzlemeOrnegi.java:3)
    at IzlemeOrnegi.yazdir(IzlemeOrnegi.java:6)
    at IzlemeOrnegi.main(IzlemeOrnegi.java:10)
```

Yığın izi dört adımda okunur:

1. **İlk satır** tip ve mesajı verir. Burada `metin` değişkeni `null`'dır ve üzerinde `length()` çağrılamamıştır.
2. **İlk `at` satırı** istisnanın doğduğu yerdir: `uzunluk` metodu, 3. satır.
3. **Aşağı doğru** çağrı zinciri okunur: `main` (10) → `yazdir` (6) → `uzunluk` (3).
4. **Önce programın kendi satırlarına** bakılır. Kütüphane sınıflarının satırları (`java.` ile başlayanlar) çoğunlukla sorunun kaynağı değildir.

Asıl neden çoğu zaman listenin aşağısındadır. İstisna 3. satırda doğmuştur, ama 3. satırda bir yanlış yoktur: `uzunluk` metodu kendisine verilen metnin uzunluğunu doğru biçimde hesaplar. `null` değerini 10. satır göndermiştir. Düzeltilecek yer çoğu zaman istisnanın doğduğu yer değil, hatalı değeri üreten yerdir.

İletideki `"metin"` adı, sınıf hata ayıklama bilgisiyle (`javac -g`) derlendiğinde görünür. Düz `javac` ile derlenen aynı program adı bilemez ve şunu yazar:

```
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()"
because "<parameter1>" is null
```

Satır numaraları ise her iki durumda da yazılır, çünkü `javac` satır bilgisini varsayılan olarak sınıf dosyasına koyar.

6.11.2 Kod izleme: yığın izi nereden gelir?

Yığın izindeki satırlar bir anlık fotoğraftır: istisna nesnesi oluştuğu anda çağrı yığınının kaydını alır. Sunumdaki adım adım izleme bu fotoğrafın nasıl çekildiğini gösterir.

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	9: `yazdir("Ali");`	main:9	—	—

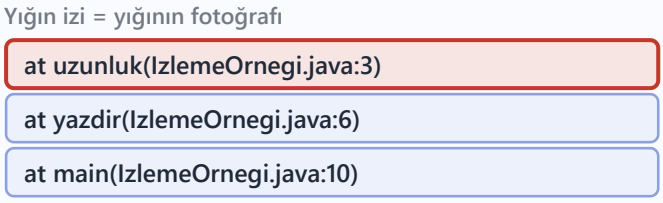
Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
2	6: `System.out.println(ad + ": " + uzunluk(ad));`	main:9 → yazdir:6	ad=Ali	—
3	3: `return metin.length();`	main:9 → yazdir:6 → uzunluk:3	metin=Ali	—
4	6: `System.out.println(ad + ": " + uzunluk(ad));`	main:9 → yazdir:6	ad=Ali	`Ali: 3`
5	10: `yazdir(null);`	main:10	—	—
6	6: `System.out.println(ad + ": " + uzunluk(ad));`	main:10 → yazdir:6	ad=null	—
7	3: `return metin.length();`	main:10 → yazdir:6 → uzunluk:3	metin=null	—
8	3: `return metin.length();`	main:10 → yazdir:6 → uzunluk:3	metin=null	—
9	10: `yazdir(null);`	—	—	`Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()" because "metin" is null ⏎ at IzlemeOrnegi.uzunluk(IzlemeOrnegi.java:3) ⏎ at IzlemeOrnegi.yazdir(IzlemeOrnegi.java:6) ⏎ at IzlemeOrnegi.main(IzlemeOrnegi.java:10)`

Tablo 6.23: `IzlemeOrnegi` programının izlemesi.

1. adımda `main`, 9. satırda `yazdir("Ali")`'yi çağırır. `yazdir` 6. satırda `uzunluk(ad)`'ı çağırır. Her çerçeve kendi metodunun hangi satırda olduğunu hatırlar. `"Ali".length()` 3 döndürür ve çerçeve silinir. `Ali: 3` yazılır ve `yazdir` biter.

5. adımda `main`, 10. satırda `yazdir(null)`'ı çağırır. `null` parametre olarak iletilir; bu noktada henüz bir sorun yoktur. Bir referansın `null` olması hata değildir; `null` üzerinden metot çağırmak hatadır. 7. adımda `null.length()` denenir ve NPE fırlatılır. O anda yığında üç çerçeve vardır:

`uzunluk:3`, `yazdir:6`, `main:10`.



Şekil 6.18: 8. adım. İstisna nesnesi doğduğu anda yığının fotoğrafını çeker. Her çerçeve bir `at` satırı olur; yığının tepesindeki çerçeve listenin ilk satırıdır.

8. adımda istisna nesnesi yığının kaydını alır. Her çerçeve bir `at` satırına dönüşür ve en üstteki çerçeve ilk sıraya yazılır. Listenin "ters" sırası buradan gelir: yığının tepesi listenin başıdır. Son adımda kimse istisnayı yakalamamıştır. Üç çerçeve de atılır; JVM yığın izini yazar ve programı sonlandırır. `Bitti` yazılmaz.

Örnek 6.11

Aşağıdaki program ne yazdırır? Yığın izine göre istisna hangi satırda doğar, `null` değeri nereden gelir?

```
1 public class Okul {
2   public static void main(String[] args) {
3     int[] a = {70, 80};
4     int[] b = null;
5     rapor("A", a);
6     rapor("B", b);
7     System.out.println("bitti");
8   }
9   static void rapor(String ad, int[] d) {
10    System.out.println(ad + ": " + ortalama(d));
11  }
12  static double ortalama(int[] notlar) {
13    int t = 0;
14    for (int n : notlar) {
15      t += n;
16    }
17    return (double) t / notlar.length;
18  }
19 }
```

```
A: 75.0
Exception in thread "main" java.lang.NullPointerException: Cannot read the array
length because "<local2>" is null
    at Okul.ortalama(Okul.java:14)
    at Okul.rapor(Okul.java:10)
    at Okul.main(Okul.java:6)
```

İlk `at` satırı istisnanın doğduğu yeri gösterir: `ortalama`, 14. satır. Bu satırdaki for-each döngüsü, `null` bir dizinin uzunluğunu okumaya çalışır. Mesaj da bunu söyler: `Cannot read the array length`. Değişken adı yerine `<local2>` görünür, çünkü program `-g` olmadan derlenmiştir. Zincir aşağıdan yukarı `main` (6) → `rapor` (10) → `ortalama` (14)'tür. `null`, 4. satırda `b`'ye atanmış ve 6. satırda gönderilmiştir. Kimse yakalamadığı için 7. satır çalışmaz.

6.11.3 İstisna yönetiminde iyi uygulamalar

İstisna düzeneğini doğru kullanmak, onu tanımak kadar önemlidir. Aşağıdaki sekiz kural, örneklerde görülen dersleri bir araya toplar.

Özel tipi yakalamak
: Mümkünse `catch (Exception e)` yerine beklenen tip yakalanır ve her türe uygun tepki verilir. Genel tip, beklenmeyen hataları da sessizce yutar.

İstisnayı yutmamak
: Boş bir `catch { }` hatayı gizler. İstisna en azından kayda geçirilir ya da kullanıcıya bildirilir.

Kaynakları kapatmak
: Dosya ve bağlantı gibi kaynaklar try-with-resources ya da `finally` ile kapatılır.

Önlenebilecekleri önlemek
: `null`, sınır ve sıfır denetimleri yapılır. `RuntimeException` yakalamak yerine kod düzeltilir.

Akış denetimi için kullanmamak
: Döngüden çıkmak gibi olağan işler istisnayla yapılmaz. İstisnalar istisnai durumlar içindir; ayrıca istisna nesnesi oluşturmak ve yığını kaydetmek sıradan bir `if`'ten çok daha pahalıdır.

Anlamlı mesaj
: Mesaj hatalı değeri ve beklentiyi söyler: `"Not 0-100 arası olmalı: 120"`.

Doğru yerde ele almak
: İstisna çözülebileceği yerde yakalanır; çözülemiyorsa `throws` ile iletilir.

Kayıt tutmak
: İstisnaları bir dosyaya kaydetmek (*logging*) hatanın izini sonradan sürmeyi kolaylaştırır. Kullanıcının ekranında kaybolan bir ileti, kayıt dosyasında kalır.

Sık yapılan hata

İstisnayı yutmak: `catch (Exception e) { }`. Bu kod derlenir ve çalışır, ama hata iz bırakmadan kaybolur. Program yanlış bir durumla çalışmayı sürdürür ve sorun çok sonra, ilgisiz bir yerde ortaya çıkar. O zaman asıl nedenin yığın izi çoktan yok olmuştur.

6.11.4 `final`, `finally` ve `finalize`

Üç adın yazılışı benzer, işleri tamamen farklıdır.

	Ne?	Görevi
`final`	Anahtar sözcük	Değişken: yeniden atanamaz · metot: geçersiz kılınamaz · sınıf: alt sınıfı olamaz
`finally`	Anahtar sözcük (blok)	`try`'dan sonra, istisna olsun olmasın her durumda çalışan temizlik bloğu
`finalize()`	`Object` sınıfının metodu	Çöp toplayıcı nesneyi silmeden önce çağırabilirdi. Ne zaman çağrılacağı belirsizdir; Java 9'dan beri kullanımdan kaldırılmış (*deprecated*) durumdadır.

Tablo 6.24: `final`, `finally` ve `finalize()`.

`finalize()`'ın sorunu güvenilmez olmasıdır. Çöp toplayıcının bir nesneyi ne zaman sileceği, hatta silip silmeyeceği belli değildir. Program biterken hiç çalışmamış olabilir. Java 18'den beri bu metot ileride tamamen kaldırılmak üzere işaretlenmiştir; onu geçersiz kılan bir sınıf derlenirken `-Xlint` şu uyarıyı verir: `finalize() in Object has been deprecated and marked for removal`. Kaynak temizliği için `finalize()` değil, try-with-resources ya da `finally` kullanılır.

6.12 Çözümlü problemler

Aşağıdaki problemler sunumun örnek problemleri ve alıştırmalarıdır. Her çözüm, gerekçesiyle birlikte adım adım verilir. Bütün çıktılar JDK 21 ile çalıştırılarak doğrulanmıştır.

Örnek 6.12 (Problem 1)

Aşağıdaki kod ne yazdırır?

```
static int f() {
  try {
    System.out.print("A");
    int x = 5 / 0;
    System.out.print("B");
    return 1;
  } catch (ArithmeticException e) {
    System.out.print("C");
    return 2;
  } finally {
    System.out.print("D");
  }
}
// main içinde
System.out.println(" " + f());
```

Çözüm. `main`, `println`'ın argümanını hesaplamak için önce `f()`'yi çağırır. `try` başlar ve `A` yazılır. `5 / 0` istisna fırlatır; `B` ve `return 1` atlanır. `catch` eşleşir, `C` yazar ve `return 2` ile 2'yi hazırlar. Metottan çıkmadan önce `finally` çalışır ve `D` yazar. Ardından 2 döner. `println`, `" " + 2` değerini yazar. Sonuç: `ACD 2`. `print` satır sonu eklemediği için dört parça aynı satırdadır.

Örnek 6.13 (Problem 2 ve 3)

```
try {
  String s = "12a";
  int n = Integer.parseInt(s);
  System.out.println("Sayı: " + n);
} catch (IllegalArgumentException e) {
  System.out.println("IAE");
} catch (Exception e) {
  System.out.println("Exception");
}
```

Kod ne yazdırır? İki `catch` bloğunun yeri değiştirilirse ne olur?

Çözüm. `"12a"` geçerli bir tam sayı değildir; `parseInt` bir `NumberFormatException` fırlatır. Birinci `catch` `IllegalArgumentException` içindir. `NumberFormatException` bu sınıfın alt sınıfı olduğu için birinci blok eşleşir: çıktı `IAE`'dir. İkinci blok denenmez.

Bloklar yer değiştirirse önce `catch (Exception e)` gelir. Bu blok `IllegalArgumentException`'ı da kapsar ve ikinci blok erişilemez olur. Kod derlenmez:

```
Ana.java:12: error: exception IllegalArgumentException has already been caught
} catch (IllegalArgumentException e) {
```

Örnek 6.14 (Problem 4)

```
static void a() {
  System.out.print("a1 ");
  b();
  System.out.print("a2 ");
}
static void b() {
  System.out.print("b1 ");
  throw new IllegalStateException("!");
}
// main içinde
try {
  a();
  System.out.print("m ");
} catch (IllegalStateException e) {
  System.out.print("yakalandı" + e.getMessage());
} finally {
  System.out.println(" son");
}
```

İstisna `b()` 'de doğuyor; `b()` 'de ve `a()` 'da `try` yok. Hangi `print` 'ler çalışır? `finally` ne zaman çalışır?

Çözüm. `main` `a()` 'yı çağırır: `a1` yazılır. `a`, `b()` 'yi çağırır: `b1` yazılır. `b` bir `IllegalStateException` fırlatır. `b` 'de `catch` yoktur; çerçeve atılır. İstisna `a` 'ya geçer; `a` 'da da `catch` yoktur ve `a2` atlanır. İstisna `main` 'e ulaşır; `try` 'ın kalanı (`m`) atlanır. `catch` eşleşir ve `yakalandı!` yazar; mesaj `"!"` olduğu için iki parça bitişiktir. En son `finally` `son` yazar ve satırı bitirir. Sonuç: `a1 b1 yakalandı! son` .

Örnek 6.15 (Problem 5)

```
int[] d = {4, 0, 2};
int toplam = 0;
for (int i = 0; i <= d.length; i++) {
  try {
    toplam += 12 / d[i];
  } catch (ArithmeticException e) {
    toplam += 100;
  } catch (ArrayIndexOutOfBoundsException e) {
    toplam += 1000;
  }
}
System.out.println(toplam);
```

`toplam` kaçtır?

Çözüm. Döngü koşulu `i <= d.length` 'tir; döngü 0, 1, 2 ve 3 için, yani dört tur döner. Her turun kendi `try` 'ı vardır.

- `i = 0` : `12 / 4 = 3` ; toplam 3.
- `i = 1` : `12 / 0` → `ArithmeticException` ; `+100` , toplam 103.
- `i = 2` : `12 / 2 = 6` ; toplam 109.
- `i = 3` : `d[3]` → `ArrayIndexOutOfBoundsException` ; `+1000` , toplam 1109.

Çıktı `1109` 'dur. `try` döngünün içinde olduğu için bir turdaki istisna döngüyü bitirmez. `i = 1` turunda `toplam += 12 / d[i]` ifadesinde bölme atamadan önce hesaplanır; istisna oluşunca toplama hiçbir şey eklenmez.

Örnek 6.16 (Problem 6)

```
try {
  throw new RuntimeException("X");
} finally {
  System.out.println("F");
}
```

Kod ne yazdırır?

Çözüm. `try` bir `RuntimeException` fırlatır. `catch` yoktur, ama `finally` vardır. İstisna metottan çıkmadan önce `finally` çalışır ve `F` yazar. Ardından istisna yoluna devam eder; `main` 'de de yakalanmadığı için program sonlanır:

```
F
Exception in thread "main" java.lang.RuntimeException: X
    at P6.main(P6.java:4)
```

Örnek 6.17 (Alıştırma: yaş okuma)

Metni yaşa çeviren `int yasOku(String metin)` metodunu yazalım. Metin sayı değilse `NumberFormatException` kendiliğinden oluşsun; sayı negatifse metot `IllegalArgumentException` fırlatsın. `"20"`, `"-3"` ve `"yirmi"` girdileri iki `catch` ile denensin. `catch` sırası ters yazılırsa ne olur?

Çözüm. Metot üç satırdır: `int yas = Integer.parseInt(metin);`, `if (yas < 0) throw new IllegalArgumentException("Yaş negatif: " + yas);` ve `return yas;`. Çağıran kodda önce `catch (NumberFormatException e)` ("Sayı değil"), sonra `catch (IllegalArgumentException e)` yazılır. Çıktı: `Yaş: 20`, `Yaş negatif: -3`, `Sayı değil: yirmi`.

Sıra ters yazılırsa `NumberFormatException` bloğuna hiçbir zaman ulaşılamaz, çünkü NFE bir `IllegalArgumentException`'dır. Derleyici şu hatayı verir: `exception NumberFormatException has already been caught`.

Örnek 6.18 (Alıştırma: kaynak ve boş dizi)

Bir `BufferedReader`'ı `finally` ile kapatan kod try-with-resources ile nasıl yazılır? `double ortalama(int[] notlar)` metodu boş dizi için ne yapmalıdır?

Çözüm. Okuyucu parantezde açılır: `try (BufferedReader r = new BufferedReader(new FileReader(yol))) { … } catch (IOException e) { … }`. `close()` otomatik çağrılır; `finally`, `null` denetimi ve iç içe `try` gerekmez.

Boş dizinin ortalaması tanımsızdır. 0 döndürmek yanlıştır, çünkü 0 geçerli bir ortalamayla karışır; bu, `-1` dönüş kodunun sorununun aynısıdır. Boş dizi geçersiz bir argümandır: `throw new IllegalArgumentException("Dizi boş");`. Kontrol edilmeyen bir tip seçilir, çünkü boş dizi göndermek çağıranın hatasıdır. Deneme programında `ortalama(new int[0])` bu istisnayı verdi, `{70, 80, 90}` dizisi ise 80.0 döndürdü.

6.13 Sık yapılan hatalar

İstisnalarla ilgili hatalar iki gruba ayrılır. Derleme hataları çoğunlukla kontrol edilen istisna kuralından ve `catch` sırasından doğar; iletileri açıktır. Mantık hataları ise derleyiciden geçer ve programın davranışını sessizce bozar. Aşağıdaki iki tablo, notun gövdesinde ayrı ayrı işlenen hataları bir arada

verir. Derleme iletileri JDK 21'in `javac` aracıyla alınmıştır.

Hata	javac iletisi	Neden / çözüm
Kontrol edilen istisnayı unutmak: `new FileReader(ad)`, `try` ya da `throws` yok	`unreported exception FileNotFoundException; must be caught or declared to be thrown`	Ya `try`-`catch` ya da `throws`.
Genel `catch`'i öne yazmak: önce `Exception`, sonra `ArithmeticException`	`exception ArithmeticException has already been caught`	Alt sınıf önce, üst sınıf sonra.
Fırlatılmayanı yakalamak: dosya işlemi olmayan `try`'a `catch (FileNotFoundException e)`	`exception FileNotFoundException is never thrown in body of corresponding try statement`	`catch` kaldırılır.
Yalnız `try` yazmak	`'try' without 'catch', 'finally' or resource declarations`	En az bir `catch` ya da `finally`.
`throw`'dan sonra aynı blokta kod	`unreachable statement`	Satır kaldırılır ya da `throw` bir koşula bağlanır.
`try` içindeki değişkeni dışarıda kullanmak	`cannot find symbol`	Değişken `try`'dan önce tanımlanır.
Metin fırlatmak: `throw "hata";`	`incompatible types: String cannot be converted to Throwable`	Bir istisna nesnesi fırlatılır.
Gövdede `throws`	`illegal start of expression`	Gövdede `throw`, imzada `throws`.
Çoklu yakalamada alt ve üst sınıf	`Alternatives in a multi-catch statement cannot be related by subclassing`	Yalnız üst tip yazılır.
`AutoCloseable` olmayan kaynak	`incompatible types: try-with-resources not applicable to variable type`	Sınıf `AutoCloseable`'ı uygular.

Tablo 6.25: İstisnalarla ilgili sık derleme hataları.

Hata	Sonuç	Düzeltme
İstisnayı yutmak: `catch (Exception e) { }`	Hata iz bırakmadan kaybolur	En azından kayıt tutulur ya da bildirilir
Dönüş kodunu denetlememek (`-1`)	Hata başka bir yerde, başka bir istisnayla ortaya çıkar	Dönüş kodu yerine istisna
`finally` içinde `return`	Bekleyen istisna ve dönüş değeri sessizce silinir	`finally` yalnız temizlik yapar
Kaynağı `try`'ın sonunda kapatmak	İstisnada kaynak açık kalır (sızıntı)	try-with-resources
`catch (Exception e)`'nin her şeyi yakaladığını sanmak	`StackOverflowError` gibi `Error`'lar yakalanmaz	`Error`'lar yakalanmaz; neden düzeltilir
NPE'yi yakalayıp geçmek	`null`'ın kaynağı gizlenir	`null`'ın nereden geldiği bulunur
Mesajsız istisna fırlatmak	İleti hatalı değeri söylemez	Değer ve beklenti mesaja yazılır
İstisnayı döngü denetimi için kullanmak	Okunması zor ve yavaş kod	Olağan durumlar `if` ile denetlenir

Hata	Sonuç	Düzeltme
Denetimden önce durumu değiştirmek	İstisnadan sonra nesne bozuk kalır	Önce denetim, sonra değişiklik

Tablo 6.26: İstisnalarla ilgili sık mantık hataları.

6.14 Özet

- İstisna, program çalışırken normal akışı bozan durumdur. Java'da istisna bir nesnedir; tipini, mesajını ve oluştuğu yeri taşır. JVM ya da programcı (`throw`) oluşturabilir.
- Çalışma zamanı sorunlarının iki kolu vardır: program toparlanabiliyorsa `Exception`, JVM ya da sistem düzeyinde ciddi bir sorun varsa `Error`.
- İstisna oluşabilen bir metot onu ya ele alır (`try`-`catch`) ya da çağırana bildirir (`throws`). Doğru yer, istisnayı anlamlı biçimde çözebilecek yerdir.
- Özel dönüş kodları (`-1`) hatayı normal sonuçla karıştırır ve denetlenmezse sessizce gözden kaçar. İstisna bu sorunları çözer.
- `try`'da istisna oluşursa bloğun kalanı atlanır ve eşleşen ilk `catch` çalışır. `catch`'ten sonra program normal akışına devam eder. İstisna yoksa `catch` çalışmaz.
- `catch` blokları yukarıdan aşağıya denenir. Bir blok, yazılan tipi ve bütün alt sınıflarını yakalar. Alt sınıf üst sınıftan önce yazılır; tersi `has already been caught` derleme hatasıdır.
- Java 7'den beri `catch (A | B e)` aynı tepkiyi tek blokta toplar. Alternatifler birbirinin alt sınıfı olamaz; `e` örtük olarak `final`'dır.
- `finally` her durumda çalışır; `try` ya da `catch` içindeki `return`'den bile önce. Dönüş değeri önce hesaplanıp saklanır. Yalnız `System.exit()` `finally`'yi engeller.
- try-with-resources, parantezde açılan `AutoCloseable` kaynakları otomatik kapatır. Kaynaklar yazılış sırasıyla açılır, ters sırayla ve `catch`'ten önce kapanır.
- İç içe `try`'da iç düzeyde yakalanmayan istisna dışa doğru ilerler. Geçtiği her düzeyin `finally`'si çalışır.
- Hiyerarşinin kökü `Throwable`'dır; iki alt sınıfı `Error` ve `Exception`'dır. `RuntimeException`, `Exception`'ın bir alt sınıfıdır.
- Kontrol edilen istisnalar (`Exception`'dan türeyip `RuntimeException`'dan türemeyenler) için derleyici "yakala ya da bildir" kuralını uygular. `RuntimeException` ve `Error` aileleri kontrol edilmez.
- `throw` gövdede bir nesne fırlatır; `throws` imzada fırlatılabilecek tipleri bildirir.
- Yakalanmayan istisna çağrı yığınında çağırana doğru yayılır; yol üzerindeki metotlar o anda sona erer. Bu sürece yığın çözülmesi denir. `main` de yakalamazsa JVM yığın izini yazar ve programı sonlandırır.
- Sık görülen çalışma zamanı istisnaları: `NullPointerException`, `ArrayIndexOutOfBoundsException`, `ArithmeticException`, `IllegalArgumentException`, `NumberFormatException`, `ClassCastException`. Bunlar çoğunlukla yakalanmaz, önlenir.

- `double` işlemleri istisna fırlatmaz (`Infinity`, `NaN`); `int` toplaması taşınca sessizce başa sarar. Taşma denetimi için `Math.addExact` kullanılır.
- Sık görülen kontrol edilen istisnalar: `IOException` ve alt sınıfı `FileNotFoundException`, `SQLException`, `ParseException`, `ClassNotFoundException`, `InterruptedException`.
- `Error`'lar (`OutOfMemoryError`, `StackOverflowError`, `AssertionError`) uygulama kodunda yakalanmaz. `assert` yalnız `-ea` ile açılır.
- Özel istisna sınıfı `Exception` (kontrol edilen) ya da `RuntimeException` (kontrol edilmeyen) genişletilerek yazılır; mesaj `super(mesaj)` ile iletilir. İstisna, ek alanlarla durumu anlatan bilgiyi taşıyabilir.
- Yığın izinin ilk satırı tip ve mesajdır; ilk `at` satırı istisnanın doğduğu yerdir. Aşağı doğru çağrı zinciri okunur. Asıl neden çoğu zaman hatalı değeri üreten satırdadır.
- İyi uygulamalar: özel tipi yakalamak, istisnayı yutmamak, kaynakları kapatmak ve önlenebileni önlemek. İstisna akış denetimi için kullanılmaz; mesaj anlamlı olur, istisna doğru yerde ele alınır ve kayda geçirilir.

6.15 Kavram listesi

istisna *exception* · çalışma zamanı hatası *runtime error* · mantık hatası *logic error* · Error *error (Error class)* · fırlatmak / yakalamak *throw / catch* · try-catch bloğu *try-catch block* · finally bloğu *finally block* · try-with-resources *try-with-resources* · kontrol edilen istisna *checked exception* · kontrol edilmeyen istisna *unchecked exception* · istisna yayılımı *exception propagation* · çağrı yığını *call stack* · yığın izi *stack trace* · özel istisna sınıfı *custom exception* · hata ayıklama *debugging* · doğrulama (assert) *assertion* · serileştirme *serialization* · çöp toplayıcı *garbage collector* · taşma *overflow* · açık dönüşüm (cast) *cast (explicit conversion)* · jenerikler *generics* · metot imzası *method signature*

yığın çözülmesi *stack unwinding* çoklu yakalama *multi-catch* kaynak sızıntısı *resource leak*

örüntü eşleme *pattern matching*

6.16 Kendini Dene

1. Dış `try` içinde `A` yazılıyor. İç `try` `throw new RuntimeException("x")` yapıyor; iç `catch` yalnız `IllegalStateException` yakalayıp `B` yazıyor; iç `finally` `C` yazıyor. İç bloktan sonra dış `try`'da `D` yazılıyor. Dış `catch (RuntimeException e)` `dış:` ile mesajı yazıyor. Bütün yazmalar

`print` ile yapılıyorsa çıktı nedir?

Cevap

`AC dış:x`. `RuntimeException`, `IllegalStateException`'ın üst sınıfıdır, alt sınıfı değil; iç `catch` eşleşmez ve `B` yazılmaz. İstisna dışarı çıkmadan önce iç `finally` `C` yazar. `D` atlanır ve dış `catch` yakalar.

2. `static int f() { try { return 1; } finally { return 2; } }` metodu ne döndürür? Bu yazım neden önerilmez?

Cevap

2. `try`'daki `return 1` değeri hazırlar, ama metottan çıkmadan önce `finally` çalışır ve kendi `return`'ü bekleyen değerin yerini alır. Aynı yazım, `try`'da fırlatılan bir istisnayı da sessizce yok ederdi. `javac -Xlint` bu yüzden `finally clause cannot complete normally` uyarısı verir.

3. `Integer.parseInt("12 ")` ve `Double.parseDouble(" 12 ")` çağrılarının sonuçları nedir?

Cevap

Birincisi `NumberFormatException: For input string: "12 "` fırlatır; `parseInt` sondaki boşluğu kabul etmez ve önce `trim()` gerekir. İkincisi `12.0` döndürür: `Double.parseDouble` baştaki ve sondaki boşlukları kendisi atar. İki benzer metodun farklı davranması, kullanıcı girdisini her zaman `trim()` ile temizlemenin bir gerekçesidir.

4. try-with-resources parantezinde `K a = new K("A"); K b = new K(null)` yazılı. `K`'nın yapıcısı ad `null` ise `IllegalArgumentException("ad yok")` fırlatıyor, değilse `… açıldı` yazıyor; `close()` `… kapandı` yazıyor. Blokta `iş` yazılıyor; `catch` `catch` ile mesajı yazıyor. Çıktı nedir?

Cevap

`A açıldı`, `A kapandı`, `catch ad yok`. İkinci kaynağın yapıcısı istisna fırlattığı için `b` hiç oluşmaz ve blok gövdesi çalışmaz. Başarıyla açılmış olan `a` yine kapatılır. Yalnız oluşturulabilen kaynaklar kapatılır.

5. `catch (IOException | Exception e)` derlenir mi? `static void a() throws IOException` ve `static void b() throws RuntimeException` metotları `main`'de `try` olmadan çağrılırsa hangi

çağrı hata verir?

Cevap

Çoklu yakalama derlenmez: `IOException`, `Exception`'ın alt sınıfıdır ve alternatifler kalıtımla ilişkili olamaz. `a()` çağrısı `unreported exception IOException` hatası verir. `b()` derlenir: `RuntimeException` kontrol edilmez; bildirimi yalnız belgeleme işi görür.

6. `Integer.MAX_VALUE * 2` ve `Math.multiplyExact(Integer.MAX_VALUE, 2)` ne verir?

Cevap

Birincisi istisnasız `-2` verir: sonuç `int` aralığını aşar ve değer başa sarar (sessiz taşma). İkincisi `ArithmeticException: integer overflow` fırlatır. Kesin metotlar taşmayı görünür kılar.

7. `class X extends Exception { X(String m) { } }` sınıfı için `new X("m").getMessage()` ne döndürür?

Cevap

`null`. Yapıcı mesajı `super(m)` ile üst sınıfa iletmemiştir. Derleyici örtük olarak parametresiz `super()`'ı çağırır ve `Throwable`'ın mesaj alanı boş kalır. Doğru yazım `X(String m) { super(m); }`'dir.

8. Bir yığın izi şöyledir: `java.lang.ArithmeticException: / by zero`, ardından `at Hesap.oran(Hesap.java:12)`, `at Hesap.rapor(Hesap.java:8)`, `at Hesap.main(Hesap.java:3)`. İstisna nerede doğmuştur? `main` hangi metodu hangi satırda çağırmıştır?

Cevap

İstisna ilk `at` satırında, `oran` metodunun 12. satırında doğmuştur. Liste aşağıdan yukarı okunursa `main` 3. satırda `rapor`'u, `rapor` da 8. satırda `oran`'ı çağırmıştır. Sıfır payda büyük olasılıkla `rapor` ya da `main` tarafında üretilmiştir; düzeltme için oradan geriye bakılır.

6.17 İleri okuma

- Oracle, *The Java Tutorials*, "Lesson: Exceptions": `try`, `catch`, `finally`, try-with-resources ve "Unchecked Exceptions — The Controversy" sayfalarıyla konunun resmî ve ücretsiz anlatımıdır.
- J. Bloch, *Effective Java* (3. baskı), Bölüm 10, Madde 69–77: istisnaların yalnız istisnai durumlar için kullanılmasını, kontrol edilen ve edilmeyen tip seçimini ve istisnaların yutulmamasını gerekçeleriyle tartışır.

- C. S. Horstmann, *Core Java, Volume I: Fundamentals* (12. baskı), Bölüm 7 “Exceptions, Assertions, and Logging”: istisnaları `assert` ve kayıt tutma ile birlikte, uygulamaya dönük biçimde işler.
- J. Gosling ve diğerleri, *The Java Language Specification, Java SE 21 Edition*, Bölüm 11 “Exceptions” ve 14.20 “The try statement”: kontrol edilen istisna kuralının ve `finally`'nin kesin tanımıdır.

6.18 Kaynaklar

1. Oracle, *The Java Tutorials*: Lesson: Exceptions (Catching and Handling Exceptions; The try-with-resources Statement; Unchecked Exceptions — The Controversy).
2. J. Gosling ve diğerleri, *The Java Language Specification, Java SE 21 Edition*, Bölüm 11: Exceptions; 14.20 The try statement.
3. Oracle, *Java SE 21 API Documentation*: `java.lang.Throwable`, `Exception`, `RuntimeException`, `Error`.
4. P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (Bölüm 11: Exception Handling: A Deeper Look).
5. K. Sierra, B. Bates ve T. Gee, *Head First Java*, 3. baskı, O'Reilly (Risky Behavior: exception handling).
6. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson (Bölüm 7: Exceptions, Assertions, and Logging).
7. J. Bloch, *Effective Java*, 3. baskı, Addison-Wesley (Bölüm 10: Exceptions, Madde 69–77).

KISIM III · BÖLÜM 7

Test ve Hata Ayıklama

Derlenen bir program doğru çalışan bir program demek değildir. Derleyici yalnız yazım kurallarını denetler; programın istenen işi yapıp yapmadığını bilemez. Bu sorunun cevabı testten gelir. Test, programın beklenen davranışı gösterip göstermediğini sistemli biçimde sınar. Bir test kaldığında da hata ayıklama başlar: arızanın nedeni bulunur ve düzeltilir. Not önce hata, kusur ve arıza kavramlarını ayırır, ardından birimden kabule kadar test düzeylerini tanıtır. Merkezde JUnit 5 ile otomatik birim testi vardır: doğrulama metotları, `@BeforeEach`, istisna testi, yaşam döngüsü ve parametreli testler. Sınır değerleri, regresyon testi ve test güdümlü geliştirme iyi test tasarımını tamamlar. İkinci yarıda hata ayıklayıcı, kesme noktaları, adım komutları ve `jdb` anlatılır. Not profil oluşturucu, kod kapsamı, statik analiz, özellik tabanlı test, Selenium ve bellek analiz araçlarıyla biter.

Bu bölümün kodu: `github.com/sercankulcu/object-oriented-programming-java/Bolum07/src`

ÖĞRENME ÇIKTILARI

Test türleri
Birim, modül, alt sistem, sistem ve kabul testini ayırt etmek; her birinin neyi doğruladığını açıklamak.

JUnit ile birim testi
`@Test`, `@BeforeEach` ve `assertEquals`, `assertThrows` gibi doğrulama metotlarıyla otomatik testler yazmak; başarısız testin iletisini okumak.

İyi test tasarımı
Sınır değerlerini seçmek, parametreli test yazmak; testleri belirleyici ve tekrarlanabilir kurmak; regresyon testinin değerini açıklamak.

Hata ayıklama
Kesme noktası koymak, programı adım adım (Step Into / Over / Out) yürütmek, değişkenleri izleyerek hatanın kaynağını bulmak.

Analiz araçları
Profil oluşturucu, kod kapsamı, statik analiz (SpotBugs), özellik tabanlı test, Selenium, Memory Analyzer ve VisualVM'in hangi soruyu cevapladığını bilmek.

Doğru araç, doğru soru
Bir sorun için test mi, hata ayıklayıcı mı, profil oluşturucu mu gerektiğine karar vermek.

7.1 Test ve hata ayıklamanın temelleri

Bir öğrenci ödev programını derler, birkaç sayı dener, sonuçlar doğru görünür ve programı teslim eder. Hoca ise programı negatif sayılarla, boş bir diziyle ya da çok büyük bir girdiyle dener; program çöker ya da yanlış sonuç verir. Öğrencinin denemesi de bir testtir, ama rastgele ve eksiktir. Bu notun konusu, bu denemeyi sistemli, tekrarlanabilir ve otomatik hâle getirmektir.

7.1.1 Neden test?

Yazılım dünyasında test üzerine pek çok söz dolaşır. Yazılım test uzmanı Michael Bolton, testin amacının hata avlamaktan çok sistemin beklendiği gibi çalıştığından emin olmak olduğunu vurgular. Programcılar arasındaki eski bir şaka, hatayı "henüz keşfedilmemiş bir özellik" diye anar. Kaliteyi bir eylem değil, alışkanlık olarak gören söz genellikle Aristoteles'e atfedilir; aslında Will Durant'ın Aristoteles yorumudur. Test etmeden yazılım yayımlamak da çoğu zaman Rus ruletine benzetilir.

Bu sözlerin en ünlüsü Edsger W. Dijkstra'ya aittir. Dijkstra'ya göre test, hataların varlığını gösterebilir, ama yokluğunu asla gösteremez. Bu cümle testin sınırını açıkça çizer. Bir test kümesi ne kadar büyük olursa olsun, yalnız denenen girdiler için bir şey söyler. Denenmemiş milyarlarca girdiden biri programı çökertebilir.

Tanım 7.1

Test (*testing*), programın beklenen davranışı gösterip göstermediğini sistemli biçimde sınamaktır. Programa seçilmiş girdiler verilir, üretilen sonuç beklenen sonuçla karşılaştırılır. Sonuçlar uyuşuyorsa test *geçer*, uyuşmuyorsa *kalır*.

Tanımdaki "beklenen sonuç" sözü önemlidir. Bir testin yazılabilmesi için önce programın ne yapması gerektiği bilinmelidir. "İki sayıyı toplayan metot 2 ve 3 için 5 döndürmeli" cümlesi bir beklentidir; test bu beklentiyi koda döker. Beklentisi belirsiz bir program test edilemez. Bu yüzden test yazmak, çoğu zaman programın gereksinimlerini netleştirmeye de yardım eder.

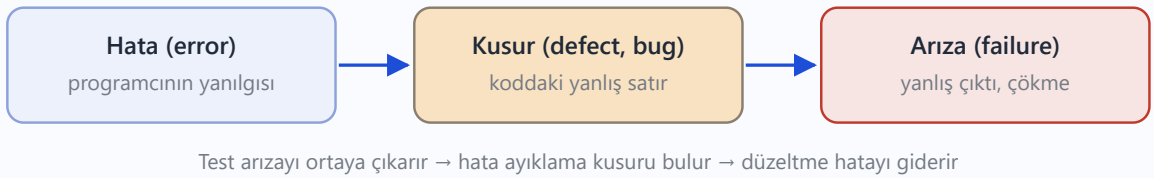
Hiçbir test kümesi hatasızlığı kanıtlamaz; ama iyi seçilmiş testler güveni büyük ölçüde artırır. Bir metot normal girdilerde, sınırlarda ve hatalı girdilerde beklenen davranışı gösteriyorsa, geri kalan girdilerde de doğru çalışma olasılığı yüksektir. Testin değeri, bu güveni ucuz ve hızlı biçimde üretmesindedir.

7.1.2 Hata, kusur, arıza

Gündelik dilde "hata" sözcüğü her şey için kullanılır: programcının yanılgısı da, koddaki yanlış satır da, ekranda görünen yanlış sonuç da hata diye anılır. Yazılım mühendisliği bu üç şeyi ayrı adlarla anar. Ayrım, test ile hata ayıklamanın neyi aradığını anlamayı kolaylaştırır.

Tanım 7.2

Hata (*error*), programcının zihnindeki yanılgıdır: yanlış bir varsayım, unutulmuş bir durum. **Kusur** (*defect, bug*), bu yanılgının koda yansıyan yanlış satırıdır. **Arıza** (*failure*), kusurlu kod çalışırken ortaya çıkan yanlış davranıştır: yanlış çıktı, çökme, donma.



Şekil 7.1: Hata zinciri. İnsan hatası koda kusur olarak girer; kusurlu satır çalışınca arıza görülür. Test arızayı ortaya çıkarır, hata ayıklama kusuru bulur, düzeltme hatayı giderir.

Şekil 7.1 zinciri soldan sağa gösterir. Zincirin her halkası farklı bir yerde yaşar. Hata programcının kafasındadır, kusur kaynak koddadır, arıza ise çalışan programın davranışındadır. Her kusur arızaya yol açmaz. Kusurlu satır hiç çalışmıyorsa ya da yalnız nadir girdilerde yanlış sonuç veriyorsa program uzun süre sorunsuz görünebilir.

Örnek 7.1

Bir programcı dizinin en büyük öğesini bulan bir metot yazar. En büyük değeri tutan değişkene başlangıç değeri olarak 0 verir: `int enb = 0;` . Bu programdaki üç halka şöyledir.

Hata: programcı "dizide en az bir pozitif sayı vardır" diye düşünmüştür. Sıcaklık ölçümlerinde bu varsayım kışın geçersizdir.

Kusur: koddaki `int enb = 0;` satırı. Tüm öğeler negatifse hiçbir öğe 0'ı aşamaz.

Arıza: sıcaklıklar `{-5, -2, -8}` iken ekranda `En yüksek: 0` görünür. 0 dizide yoktur bile.

Aynı metot `{3, 9, 4}` için doğru sonuç (9) verir. Kusur yalnız tüm öğeler negatif olduğunda arızaya dönüşür. Bu yüzden yalnız pozitif sayılarla yapılan bir deneme kusuru hiçbir zaman ortaya çıkarmaz.

7.1.3 Test ile hata ayıklamanın farkı

Test ile hata ayıklama sıkça karıştırılır. İkisi aynı amaca hizmet eder, ama farklı soruları cevaplar. Test "program çalışıyor mu?" diye sorar. Hata ayıklama ise bir arızanın varlığı bilindikten sonra "neden çalışmıyor?" sorusunu sorar.

> **Tanım 7.3**
>
> **Hata ayıklama** (*debugging*), bilinen bir arızanın nedenini, yani kusurlu satırı bulup düzeltme sürecidir. Bu işte kullanılan araca **hata ayıklayıcı** (*debugger*) denir.

	Test	Hata ayıklama
Amaç	Programın beklendiği gibi çalışıp çalışmadığını sınamak	Bilinen bir arızanın nedenini bulup düzeltmek
Soru	"Çalışıyor mu?"	"Neden çalışmıyor?"
Sonuç	Geçti / kaldı (✓ / ✗)	Hatalı satır ve düzeltme
Araçlar	JUnit, Selenium, kod kapsamı	Hata ayıklayıcı, yığın izi, `println`
Ne zaman	Sürekli, her değişiklikten sonra	Bir test kaldığında ya da kullanıcı hata bildirdiğinde

Tablo 7.1: Test ile hata ayıklamanın karşılaştırması.

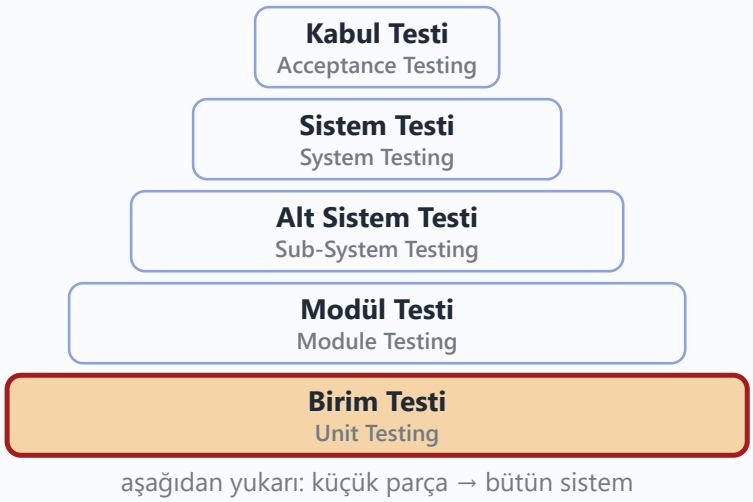
Tablo 7.1 iki etkinliğin birbirini nasıl izlediğini de gösterir. Test sürekli çalışır ve arızayı haber verir. Arıza görülünce hata ayıklama devreye girer ve kusuru bulur. Kusur düzeltildikten sonra yine test çalışır: düzeltmenin işe yaradığını ve başka bir şeyi bozmadığını doğrular. İyi bir yazılım süreci bu döngüyü defalarca tekrarlar.

Tablodaki "yığın izi", bir istisna yakalanmadığında JVM'in yazdırdığı çağrı listesidir (*stack trace*). Hangi metodun hangi satırda istisna fırlattığını ve o metodu kimin çağırdığını gösterir. Hata ayıklamada ilk bakılacak bilgi çoğunlukla budur.

7.2 Test türleri

Bir yazılım tek bir metottan değil, birbirine bağlı yüzlerce sınıftan oluşur. Bu yüzden test de tek bir adımda yapılmaz. Önce en küçük parçalar tek tek sınanır, sonra parçaların birlikte çalışması, en sonda da bütün sistem kullanıcının gözünden denetlenir. Her adım farklı bir test düzeyidir.

7.2.1 Test düzeyleri



Şekil 7.2: Test düzeyleri piramidi. Tabanda en çok sayıda, en küçük ve en hızlı testler (birim testi), tepede en az sayıda, en büyük ve en yavaş testler (kabul testi) bulunur.

Şekil 7.2 beş düzeyi bir piramit olarak gösterir. Piramidin biçimi bilinçlidir. Taban geniştir, çünkü birim testleri ucuzdur ve çok sayıda yazılır. Tepe dardır, çünkü kabul testleri pahalıdır, yavaştır ve az sayıda yapılır. Aşağıdan yukarı çıkıldıkça sınanan parça büyür.

Birim testi (*unit testing*) tek tek, bağımsız bileşenlere odaklanır: bir sınıf ya da bir metot. Her bileşenin tek başına doğru çalıştığından emin olunur. Örneğin bir `BankaHesabi` sınıfının `cek()` metodu, yetersiz bakiyede istisna fırlatıyor mu? Birim testi en küçük parçayı sınar; en çok sayıda yazılan ve en hızlı çalışan testtir.

Modül testi (*module testing*) birbiriyle ilgili bileşenlerden oluşan bir modülü sınar. Modül içindeki bileşenlerin uyumu ve birlikte işlevi doğrulanır. Örneğin `Hesap`, `Islem` ve `Dekont` sınıfları birlikte para transferini doğru yapıyor mu? Her sınıf tek başına doğru olsa bile, aralarındaki işbirliği yanlış olabilir.

Alt sistem testi (*sub-system testing*) alt sistemler arasındaki arayüz uyumsuzluklarını arar. Bileşenler arasındaki iletişimi ve veri akışını denetler. Örneğin hesap alt sistemi ile veri tabanı alt sistemi aynı tarih biçimini mi kullanıyor? Biri tarihi gün/ay/yıl, öteki yıl-ay-gün diye bekliyorsa iki alt sistem tek başına kusursuz çalışır, ama birlikte yanlış kayıt üretir.

Sistem testi (*system testing*) bütün sistemi bir bütün olarak sınar. Hem işlevsel gereksinimler (sistem ne yapıyor) hem de işlevsel olmayan gereksinimler (hız, güvenlik, kullanılabilirlik) denetlenir. “1000 eş zamanlı kullanıcıyla havale 2 saniyenin altında tamamlanıyor mu?” sorusu bir sistem testi sorusudur. Bu soruyu hiçbir birim testi cevaplayamaz.

Kabul testi (*acceptance testing*) sistemi benzetim verisi yerine gerçek verilerle sınar. Sistemin gerçek dünyadaki kullanılabilirliği ve başarımı doğrulanır. Kabul testi kullanıcılarla birlikte yapılır. Alfa testi geliştiricinin ortamında, beta testi kullanıcının kendi ortamında yürür. Sonunda tek bir soru cevaplanır: müşteri "bu sistemi kabul ediyorum" diyebiliyor mu?

7.2.2 Düzeyler bir bakışta

Düzey	Neyi test eder?	Kim yapar?	Veri
Birim	Tek sınıf / metot	Geliştirici	Küçük, elle seçilmiş
Modül	Birlikte çalışan birkaç sınıf	Geliştirici	Senaryo verisi
Alt sistem	Alt sistemler arası arayüzler	Geliştirici, test ekibi	Senaryo verisi
Sistem	Bütün sistem; hız, güvenlik	Test ekibi	Gerçeğe yakın
Kabul (alfa / beta)	Müşteri gereksinimleri	Kullanıcı ile birlikte	Gerçek veri

Tablo 7.2: Test düzeylerinin karşılaştırması.

Tablo 7.2 düzeylerin kimin elinde olduğunu da gösterir. Alt düzeyleri geliştirici kendisi yazar; üst düzeylerde ayrı bir test ekibi ve sonunda müşteri devreye girer. Veri de yukarı çıkıldıkça gerçeğe yaklaşır.

Düzeyler arasındaki en önemli fark maliyettir. Aşağı düzeydeki bir hata ucuz ve hızlı bulunur. Birim testi kaldığında kusurun yeri neredeyse bellidir: tek bir metot. Aynı hata kabul testinde ya da kullanıcının elinde ortaya çıkarsa önce hangi parçadan kaynaklandığı aranır, sonra düzeltilir, sonra bütün sistem yeniden denenir. Düzeltmenin maliyeti katlanarak büyür. Bu yüzden hataları olabildiğince aşağıda yakalamak hedeflenir.

Örnek 7.2

Aşağıdaki sorunlardan her birini en doğrudan hangi düzey yakalar?

(a) `faizHesapla()` metodu yüzdeyi 100'e bölmeyi unutuyor. Tek bir metodun mantık hatasıdır: *birim testi*.

(b) `Hesap` sınıfı tutarı kuruş cinsinden, `Dekont` sınıfı lira cinsinden bekliyor. İki sınıfın birlikte çalışmasıyla ilgilidir: *modül testi*.

(c) Ödeme alt sistemi tarihleri gün/ay/yıl, raporlama alt sistemi yıl-ay-gün biçiminde gönderiyor. Alt sistemlerin birleştiği arayüzdedir: *alt sistem testi*.

(d) Sistem 500 kullanıcıda yanıt vermez hâle geliyor. İşlevsel olmayan bir gereksinimdir: *sistem testi*.

(e) Müşteri, raporun kendi muhasebe düzenine uymadığını söylüyor. Gerçek kullanıcı gereksinimidir: *kabul testi*.

7.2.3 Kara kutu ve beyaz kutu testi

Testler düzeylerinin yanında, kodun içine bakıp bakmamalarına göre de ikiye ayrılır.

> **Tanım 7.4**
>
> **Kara kutu testi** (*black-box testing*), kodun içine bakmadan yalnız girdi ve beklenen çıktı ile yapılan testtir. Program dışarıdan kapalı bir kutu gibi görülür. **Beyaz kutu testi** (*white-box testing*) ise kodun yapısı bilinerek yapılır; amaç her dalı ve her döngüyü en az bir kez çalıştırmaktır.

Kabul ve sistem testleri çoğunlukla kara kutu testidir. Müşteri kodu okumaz; yalnız sistemin istediğini yapıp yapmadığına bakar. Birim testleri ve kod kapsamı ise beyaz kutu bakışını kullanır. Bir metodun içinde `if` varsa, iyi bir birim test kümesi o `if`'in hem doğru hem yanlış dalını çalıştırır. İki bakış birbirini tamamlar. Kara kutu testi gereksinimden yola çıkar ve "ne yapmalı?" sorusunu sorar. Beyaz kutu testi koddan yola çıkar ve "hangi yolu hiç denemedim?" sorusunu sorar.

7.3 Birim testi ve JUnit

Test düzeylerinin tabanı birim testidir ve programcının en sık yazdığı test de budur. Java'da birim testleri neredeyse her zaman JUnit çerçevesiyle yazılır. JUnit'in neden gerektiğini görmek için önce onsuz test etmenin nasıl bir şey olduğuna bakalım.

7.3.1 Elle test etmek

Test edilecek sınıf iki metotlu basit bir hesap makinesidir.

```
public class HesapMakinesi {
  public int topla(int a, int b) {
    return a + b;
  }

  public int carp(int a, int b) {
    return a * b;
  }
}
```

Programlamaya yeni başlayan biri bu sınıfı denemek için bir `main` metodu yazar, metotları çağırır ve sonuçları ekrana basar.

```
public class ElleTest {
  public static void main(String[] args) {
    HesapMakinesi hm = new HesapMakinesi();
    System.out.println("topla(2, 3) = " + hm.topla(2, 3));
    System.out.println("carp(2, -3) = " + hm.carp(2, -3));
    // Doğru mu? Ekrana bakıp kendimiz karar veriyoruz.
  }
}
```

```
topla(2, 3) = 5
carp(2, -3) = -6
```

Çıktı doğru görünüyor: 2 + 3 = 5 ve 2 × (−3) = −6. Ama “doğru görünüyor” kararını program değil, ekrana bakan insan verir. Koddaki yorum satırı da bunu itiraf eder. Bu yöntemin dört temel sorunu vardır.

- Sonucun doğru olduğuna insan gözü karar verir. Göz yorulur ve yanılır; 0,30000000000000004 ile 0,3 arasındaki farkı gözden kaçırmak çok kolaydır.
- Kod değiştiğinde her şeyin yeniden elle denenmesi gerekir. Bunu her seferinde kimse yapmaz.
- Test kodu `main` içinde kalır, iş bitince silinir. Harcanan emek kaybolur.
- Yüz metot için yüz çıktı satırı okunmalıdır. Büyük bir projede bu pratikte imkânsızdır.

Çözüm basittir: beklenen değer de kodun içine yazılır ve karşılaştırma bilgisayara bırakılır. Program “5 bekliyordum, 5 geldi” ya da “5 bekliyordum, 6 geldi” diye kendisi karar verir. Buna **otomatik birim testi** denir.

Tanım 7.5

Birim testi (*unit test*), bir metodu ya da sınıfı tek başına sınayan otomatik testtir. Test, metodu bilinen bir girdiyle çağırır, sonucu beklenen değerle karşılaştırır ve kararı kendisi verir: geçti ya da kaldı.

7.3.2 JUnit çerçevesi

JUnit, Java için en yaygın birim testi çerçevesidir (*framework*). Çerçeve, belirli bir işi kolaylaştıran ve programcının koduna kendisi çağrı yapan hazır bir kütüphanedir. Testler sıradan Java metotları olarak yazılır; JUnit onları bulur, çalıştırır ve sonuçlarını raporlar. Programcı `main` yazmaz, test metotlarını kimin hangi sırayla çağıracağıyla uğraşmaz.

Java'da bir *birim* genellikle bir sınıf anlamına gelir. Testler o sınıfın metotlarını tek tek sınar. JUnit temelde birim testi için tasarlanmıştır, ama esnektir. Birden fazla sınıfı içeren senaryolarda modül testi, sistem düzeyindeki etkileşimlerde sistem testi için de kullanılabilir. IntelliJ IDEA, Eclipse, NetBeans,

VS Code ve Maven / Gradle gibi derleme araçları JUnit testlerini doğrudan çalıştırır.

Sürüm	Test nasıl yazılır?
JUnit 3	`TestCase` sınıfından kalıtım, adı `test` ile başlayan metotlar
JUnit 4	`@Test` etiketi, `org.junit` paketi
JUnit 5	`@Test`, `org.junit.jupiter.api` paketi; lambda, parametreli test

Tablo 7.3: JUnit sürümlerinde test yazımı.

Tablo 7.3 JUnit'in gelişimini özetler. JUnit 3 testleri kalıtımla ve adlandırma kuralıyla tanırdı. JUnit 4 bunun yerine etiket (*annotation*) kullanmaya başladı. Etiket, `@` ile başlayan ve derleyiciye ya da bir araca bilgi veren bir nottur; `@Override` da bir etikettir. Bu notta JUnit 5 kullanılır. JUnit 5'in test yazılan kısmının adı *Jupiter*'dir; paket adlarındaki `jupiter` sözcüğü buradan gelir. Gradle kullanan bir projeye JUnit 5 şu satırla eklenir:

```
testImplementation("org.junit.jupiter:junit-jupiter:5.11.4")
```

Maven ve Gradle projelerinde dosyalar belirli bir düzende durur. Test sınıfları `src/test/java` klasörüne, test edilen sınıflar `src/main/java` klasörüne konur. Böylece test kodu programla birlikte derlenir ve çalıştırılır, ama son kullanıcıya giden pakete girmez.

7.3.3 İlk JUnit 5 testi

`HesapMakinesi` için elle yazılan denemeyi şimdi bir JUnit testine çevirelim.

```
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;

class HesapMakinesiTest {

  @Test
  void ikiSayiyiToplar() {
    HesapMakinesi hm = new HesapMakinesi();
    assertEquals(5, hm.topla(2, 3));
  }

  @Test
  void negatifSayiylaCarpar() {
    HesapMakinesi hm = new HesapMakinesi();
    assertEquals(-6, hm.carp(2, -3));
  }
}
```

Kodu satır satır inceleyelim. İlk satır `@Test` etiketini içe aktarır. İkinci satırdaki `import static`, `Assertions` sınıfının `static` metodu `assertEquals`'ı içe aktarır. Bu sayede her seferinde `Assertions.assertEquals(…)` yazmak gerekmez. Test sınıfının adı, test edilen sınıfın adına `Test` eklenerek verilir: `HesapMakinesiTest`. Bu bir kural değil, ama yaygın bir alışkanlıktır.

Sınıftaki her test metodu `@Test` ile işaretlenir. Bu etiket JUnit'e "bu metot bir testtir" der. Test metodu `void` ve parametresizdir. `main` gerekmez. JUnit 5'te test sınıfının ve metotlarının `public` olması da gerekmez; paket erişimi yeterlidir.

Her test metodu bir nesne oluşturur, metodu çağırır ve sonucu `assertEquals(beklenen, gerçek)` ile denetler. Parametrelerin sırası önemlidir: önce beklenen değer, sonra gerçek sonuç yazılır. Değerler eşitse metot sessizce döner ve test geçer (✓). Eşit değilse `AssertionFailedError` fırlatır ve test kalır (✗). Bu iki test çalıştırıldığında rapor şöyledir:

```
✓ ikiSayiyiToplar()
✓ negatifSayiylaCarpar()
2 test · 2 başarılı · 0 başarısız
```

Test metotlarının adları birer cümle gibidir: `ikiSayiyiToplar`, `negatifSayiylaCarpar`. Bu bilinçli bir seçimdir. Bir test kaldığında rapordaki ad, neyin bozulduğunu hemen söyler. `test1` gibi bir ad ise yalnız "bir şey bozuldu" der.

Sık yapılan hata

`import static` satırını unutmak. Bu satır olmadan `assertEquals` çağrısı derlenmez:

```
import org.junit.jupiter.api.Test;
class ImpTest {
  @Test void t() { int x = 5; assertEquals(5, x); }
}
```

```
ImpTest.java:3: error: cannot find symbol
  @Test void t() { int x = 5; assertEquals(5, x); }
                               ^
  symbol:   method assertEquals(int,int)
  location: class ImpTest
1 error
```

Derleyici `assertEquals` adlı bir metodu test sınıfında arar ve bulamaz. Çözüm, `import static org.junit.jupiter.api.Assertions.*;` satırını eklemektir.

7.3.4 Doğrulama metotları

Tanım 7.6

Doğrulama (*assertion*), beklenen ile gerçek sonucu karşılaştıran ya da bir varsayımı denetleyen ifadedir. JUnit 5'te doğrulama metotları `org.junit.jupiter.api.Assertions` sınıfının `static` metotlarıdır. Doğrulama başarısız olursa metot `AssertionFailedError` fırlatır ve test kalır.

`assertEquals` en sık kullanılan doğrulamadır, ama tek değildir. Tablo 7.4 en yaygın doğrulama metotlarını listeler. Tablonun son sütunundaki örneklerin hepsi geçer; her biri JUnit 5.11 ile çalıştırılarak denendi.

Metot	Ne zaman geçer?	Örnek (hepsi geçer)
`assertEquals(b, g)`	`b` ve `g` eşit (`equals`)	`assertEquals(5, 2 + 3)`
`assertEquals(b, g, delta)`	Ondalık sayılarda fark ≤ delta	`assertEquals(0.3, 0.1 + 0.2, 1e-9)`
`assertNotEquals(a, g)`	Eşit değil	`assertNotEquals(4, 2 + 3)`
`assertTrue` / `assertFalse`	Koşul doğru / yanlış	`assertFalse("abc".isEmpty())`
`assertNull` / `assertNotNull`	Değer `null` / `null` değil	`assertNotNull("x")`
`assertSame(a, b)`	Aynı nesne (`==`)	`assertSame(s, s)`
`assertArrayEquals`	Diziler öğe öğe eşit	`assertArrayEquals(new int[] {1, 2}, new int[] {1, 2})`
`assertThrows(T.class, kod)`	Kod o tipte istisna fırlatır	`assertThrows(ArithmeticException.class, () -> …)`
`assertAll(…)`	Tüm doğrulamalar (hepsi denenir)	`assertAll(() -> …, () -> …)`
`fail("mesaj")`	Asla; testi bilerek düşürür	`fail("henüz yazılmadı")`

Tablo 7.4: JUnit 5 doğrulama metotları.

Tablodaki her satırın arkasında bir ayrıntı vardır. `assertEquals` nesneler için `equals` metodunu kullanır; bu yüzden içerikleri aynı iki `String` eşit sayılır. `assertSame` ise `==` ile karşılaştırır ve yalnız aynı nesneyi gösteren iki referansta geçer. Diziler için `assertEquals` yanıltıcıdır. Dizilerin `equals` metodu `Object`'ten gelir ve yalnız referansları karşılaştırır. İçeriği aynı iki ayrı dizi bu yüzden eşit sayılmaz:

```
assertEquals(new int[] {1, 2}, new int[] {1, 2});   // kalır
```

```
expected: [I@4a22f9e2<[1, 2]> but was: [I@3c419631<[1, 2]>
```

İletideki `[I@4a22f9e2`, dizinin tipini (`[I`: `int` dizisi) ve kimlik kodunu gösterir; köşeli parantezlerin içindeki `[1, 2]` ise içeriğidir. JUnit, iki değerin metin hâli aynı olduğu için bu ayrıntıyı ekler. İçerik aynıdır, nesneler farklıdır. Dizileri öğe öğe karşılaştırmak için `assertArrayEquals` kullanılır. Kimlik kodları her çalıştırmada değişir.

Ondalık sayılar ayrı bir dikkat ister. 0,1 ve 0,2 ikili sistemde tam gösterilemez; toplamları 0,3 değil 0,30000000000000004 olur. Bu yüzden ondalık sayılarda üçüncü parametre olarak bir tolerans verilir. `assertEquals(0.3, 0.1 + 0.2, 1e-9)` çağrısı, iki değerin farkı 10^{-9}'dan küçük olduğu için geçer. Tolerans unutulursa test kalır:

```
expected: <0.3> but was: <0.30000000000000004>
```

Doğrulama metotlarının çoğu son parametre olarak isteğe bağlı bir ileti de alır. İleti, test kaldığında raporun başına eklenir ve neyin yanlış gittiğini açıklar. `assertEquals(4, 2 + 3, "toplam yanlış")` çağrısı şu iletiyle kalır:

```
toplam yanlış ==> expected: <4> but was: <5>
```

`assertAll` birden çok doğrulamayı bir araya toplar. Sıradan doğrulamalar art arda yazıldığında ilk başarısız doğrulama testi bitirir; sonrakiler hiç çalışmaz. `assertAll` ise bütün doğrulamaları çalıştırır ve başarısız olanların hepsini birlikte raporlar:

```
@Test
void ikiDogrulama() {
  assertAll(
    () -> assertEquals(4, 2 + 3),
    () -> assertEquals(7, 3 + 3));
}
```

```
org.opentest4j.MultipleFailuresError: Multiple Failures (2 failures)
        org.opentest4j.AssertionFailedError: expected: <4> but was: <5>
        org.opentest4j.AssertionFailedError: expected: <7> but was: <6>
```

İki doğrulama da kalmıştır ve rapor ikisini birden gösterir. Bir nesnenin birkaç alanını aynı anda denetlerken bu özellik zaman kazandırır: hangi alanların yanlış olduğu tek bir çalıştırmada görülür. `assertAll`'un parametreleri birer lambda ifadesidir. Lambda ifadesi, `() -> …` biçiminde yazılan adsız küçük bir fonksiyondur; burada "çalıştırılacak kod parçası" anlamına gelir.

Sık yapılan hata

Beklenen ile gerçeği ters yazmak. `assertEquals(hm.topla(2, 3), 6)` da derlenir ve çalışır, ama kalınca ileti ters okunur: `expected: <5> but was: <6>`. Rapor "metot 6 döndürdü" der gibi görünür; oysa metot 5 döndürmüştür. Yanlış iletiyle yapılan hata ayıklama yanlış yere gider. Kural: önce beklenen, sonra gerçek.

7.3.5 Test edilecek sınıf: `BankaHesabi`

Gerçekçi bir test örneği için biraz daha zengin bir sınıf gerekir. `BankaHesabi` sınıfı bir bakiye tutar; para yatırma ve çekme işlemlerinde girdiyi denetler.

```
public class BankaHesabi {
  private double bakiye;
  public BankaHesabi(double bakiye) {
    this.bakiye = bakiye;
  }
  public void yatir(double miktar) {
    if (miktar <= 0) {
      throw new IllegalArgumentException("Geçersiz miktar");
    }
    bakiye += miktar;
  }
  public void cek(double miktar) {
    if (miktar > bakiye) {
      throw new IllegalStateException("Yetersiz bakiye");
    }
    bakiye -= miktar;
  }
  public double getBakiye() {
    return bakiye;
  }
}
```

`yatir` metodu sıfır ya da negatif miktarı `IllegalArgumentException` ile reddeder. `cek` metodu bakiyeden büyük bir miktarı `IllegalStateException` ile reddeder. İki istisnanın seçimi de anlamlıdır. Negatif miktar geçersiz bir *argümandır*. Yetersiz bakiye ise argümanın değil, nesnenin o anki *durumunun* işleme izin vermemesidir.

Test yazmadan önce sınıfın hangi davranışlarının sınanacağı düşünülür. Tablo 7.5 başlangıç bakiyesi 100 olan bir hesap için bu davranışları listeler.

Çağrı	Beklenen	Tür
`yatir(50)`	bakiye 150	normal durum
`cek(30)`	bakiye 70	normal durum

Çağrı	Beklenen	Tür
`cek(500)`	`IllegalStateException`; bakiye 100 kalır	hatalı girdi
`yatir(-10)`	`IllegalArgumentException`	hatalı girdi
`yatir(0)`	`IllegalArgumentException`	sınır
`cek(100)`	bakiye 0, istisna yok	sınır

Tablo 7.5: `BankaHesabi` için sınanacak davranışlar (başlangıç bakiyesi 100).

Tablonun her satırı ayrı bir test metodu olur. Satırlar üç gruba ayrılır: normal durumlar, hatalı girdiler ve sınırlar. Yalnız normal durumları sınamak kolaydır ve çok yaygındır; ama hatalar çoğunlukla öteki iki gruptadır. `cek(100)` satırı buna iyi bir örnektir. Koşul `miktar > bakiye` yerine yanlışlıkla `miktar >= bakiye` yazılsaydı, bütün parayı çekmek istisna fırlatırdı. Bu kusuru yalnız tam sınırdaki girdi yakalar.

7.3.6 Bir testin yapısı: hazırla, çalıştır, doğrula

İyi yazılmış bir test metodu üç bölümden oluşur. Bu düzen İngilizce baş harfleriyle AAA (*Arrange, Act, Assert*) diye anılır.

```
class AaaTest {

  @Test
  void cekmeBakiyeyiAzaltir() {
    // Hazırla (Arrange)
    BankaHesabi hesap = new BankaHesabi(100);
    // Çalıştır (Act)
    hesap.cek(30);
    // Doğrula (Assert)
    assertEquals(70.0, hesap.getBakiye(), 0.001);
  }
}
```

Hazırla (*arrange*) bölümünde test için gereken nesne ve veriler kurulur: bakiyesi 100 olan bir hesap. **Çalıştır** (*act*) bölümünde test edilen metot bir kez çağrılır: `cek(30)`. **Doğrula** (*assert*) bölümünde sonuç beklenenle karşılaştırılır: bakiye 70 olmalıdır. Bakiye `double` olduğu için üçüncü parametre olarak 0.001 toleransı verilir.

AAA düzeni testi okunur kılar. Okuyan kişi testin neyi kurduğunu, neyi yaptığını ve neyi beklediğini üç bakışta görür. Düzenin bir sonucu da şudur: her test tek bir davranışı sınar. Bir testte birden çok "çalıştır" adımı varsa, o test aslında birkaç testtir ve bölünmelidir. Böylece test kaldığında hangi davranışın bozulduğu hemen anlaşılır.

7.3.7 Ortak hazırlık: `@BeforeEach`

`BankaHesabi` için yazılan testlerin hepsi aynı hazırlıkla başlar: bakiyesi 100 olan bir hesap. Bu satırı her teste ayrı ayrı yazmak hem tekrar hem de hata kaynağıdır. JUnit ortak hazırlık için `@BeforeEach` etiketini sunar.

```
class BankaHesabiTest {
  private BankaHesabi hesap;

  @BeforeEach
  void hazirla() {
    hesap = new BankaHesabi(100);
  }

  @Test
  void yatirmaBakiyeyiArtirir() {
    hesap.yatir(50);
    assertEquals(150.0, hesap.getBakiye(), 0.001);
  }

  @Test
  void fazlaCekmeIstisnaFirlatir() {
    assertThrows(IllegalStateException.class,
      () -> hesap.cek(500));
    assertEquals(100.0, hesap.getBakiye(), 0.001);
  }
}
```

`hesap` artık test sınıfının bir alanıdır. `@BeforeEach` ile işaretli `hazirla` metodu *her* testten önce çalışır ve bu alana yeni bir hesap atar. Böylece her test yeni ve temiz bir hesapla, bakiye 100 ile başlar.

Burada çok önemli bir ayrıntı vardır. JUnit her test metodu için test sınıfından *yeni bir nesne* oluşturur. `yatirmaBakiyeyiArtirir` testi bakiyeyi 150'ye çıkarır; ama bu değişiklik yalnız o testin nesnesindeki hesabı etkiler. `fazlaCekmeIstisnaFirlatir` testi kendi test nesnesinde, kendi hesabıyla çalışır. Testler birbirini etkilemez.

İkinci test `assertThrows` kullanır. Bu metot ikinci parametresindeki kodu çalıştırır ve o kodun birinci parametrede verilen tipte bir istisna fırlatmasını bekler. `() -> hesap.cek(500)` bir lambda ifadesidir: hemen çalışmaz, `assertThrows`'a "çalıştırılacak kod parçası" olarak verilir. `assertThrows` kodu kendi içinde çalıştırır ve istisnayı yakalar. İstisna yakalandığı için test devam eder ve ikinci doğrulama, bakiyenin değişmediğini denetler. Hatalı bir işlemin nesneyi bozmadığını görmek, istisnanın fırlatıldığını görmek kadar önemlidir.

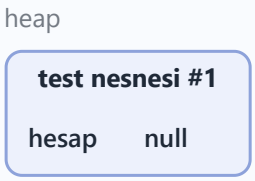
Bu iki test çalıştırıldığında ikisi de geçer:

```
✓ fazlaCekmeIstisnaFirlatir()
✓ yatirmaBakiyeyiArtirir()
2 test · 2 başarılı
```

Rapordaki sıra kaynak koddaki sıra değildir. JUnit 5, test metotlarını varsayılan olarak belirli ama açıkça belirtilmeyen bir algoritmayla sıralar. Bu sıra her çalıştırmada aynıdır, ama kod sırasıyla aynı olmak zorunda değildir.

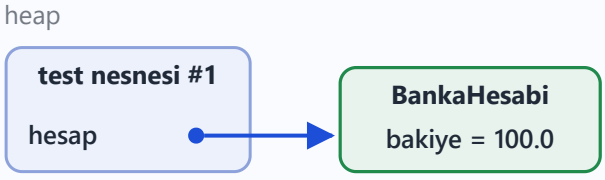
7.3.8 Kod izleme: JUnit testleri nasıl çalıştırır?

`BankaHesabiTest` sınıfının çalıştırılmasını adım adım izleyelim. Her adımda heap'teki nesnelere bakacağız: test nesnesi ve onun gösterdiği `BankaHesabi` nesnesi.



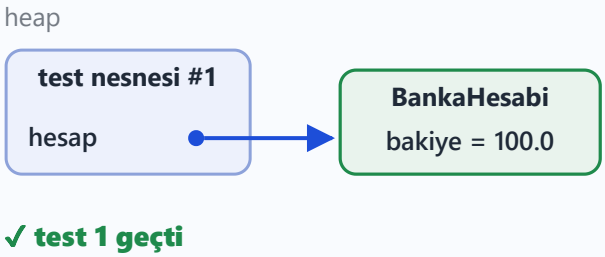
Şekil 7.3: Adım 1. JUnit sınıftaki `@Test` metotlarını bulur ve ilk test için yeni bir test nesnesi oluşturur. `hesap` alanı henüz `null`'dır.

İlk adımda JUnit, yansıma (*reflection*) denen düzenekle sınıfı inceler ve `@Test` ile işaretli metotları bulur. Ardından test sınıfının parametresiz yapıcısını çağırarak bir test nesnesi oluşturur. Bu nesnenin `hesap` alanı varsayılan değeri olan `null`'ı taşır (Şekil 7.3).



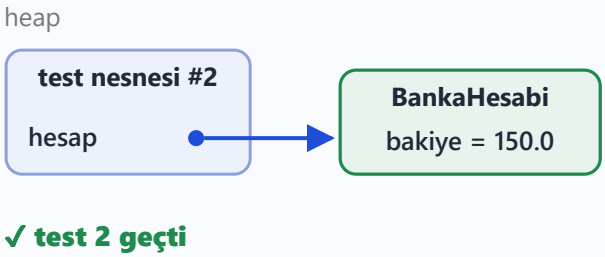
Şekil 7.4: Adım 2. Testten önce `@BeforeEach` çalışır: yeni bir hesap oluşur, bakiye 100.

İkinci adımda `hazirla` çalışır ve `hesap` alanı yeni bir `BankaHesabi` nesnesini gösterir. Sıra JUnit'tedir: önce `fazlaCekmeIstisnaFirlatir` çalışır. `assertThrows` lambdayı çağırır; `cek(500)`, 500 > 100 olduğu için `IllegalStateException` fırlatır. İstisna beklenen tiptedir, `assertThrows` geçer. Bakiye hâlâ 100'dür; ikinci doğrulama da geçer (Şekil 7.5).



Şekil 7.5: Adım 4. İstisna beklenen tipte; `assertThrows` geçer. Bakiye 100'de kalmıştır, ikinci doğrulama da geçer.

İkinci test için JUnit yepyeni bir test nesnesi (#2) oluşturur. Önceki test nesnesi ve onun hesabı artık kullanılmaz; çöp toplayıcı onları zamanı gelince temizler. `@BeforeEach` yine çalışır ve bu test de bakiyesi 100 olan bir hesapla başlar. `yatir(50)` çağrısıyla bakiye 150 olur. Doğrulamada 150,0 ile 150,0 karşılaştırılır; fark 0,001'den küçüktür ve test geçer (Şekil 7.6).



Şekil 7.6: Adım 8. İkinci test kendi test nesnesinde (#2), kendi hesabıyla çalıştı. İki test de geçti ve birbirinden bağımsız kaldı.

Adım	Olay	Test nesnesi	`hesap`	Bakiye
1	Test nesnesi oluşur	#1	`null`	—
2	`@BeforeEach`	#1	yeni hesap	100
3	`cek(500)`, istisna	#1	aynı	100
4	iki doğrulama geçer ✓	#1	aynı	100
5	Yeni test nesnesi	#2	`null`	—
6	`@BeforeEach`	#2	yeni hesap	100
7	`yatir(50)`	#2	aynı	150
8	doğrulama geçer ✓	#2	aynı	150

Tablo 7.6: `BankaHesabiTest` çalıştırılırken her adımdaki durum.

Tablo 7.6 izlemenin özetidir. Tablonun en önemli satırı 5. adımdır: JUnit ikinci test için önceki nesneyi yeniden kullanmaz. Bu tasarım, testlerin birbirinden bağımsız olmasını garanti eder. Bir testin bıraktığı durum bir sonrakine sızmaz. Bu garanti yalnız örnek alanları için geçerlidir; `static` alanlar sını-

fa aittir ve bütün test nesneleri aynı alanı paylaşır. Bir testin `static` bir sayacı artırması, sonraki testin gördüğü değeri de değiştirir.

7.3.9 İstisnaları test etmek

Hatalı girdinin reddedildiğini sınamak, doğru girdiyi sınamak kadar önemlidir. Bir metodun negatif miktarı kabul etmesi, yanlış sonuç vermesinden çok daha tehlikeli olabilir. `assertThrows` bu iş için yapılmıştır.

```
class BankaIstisnaTest {

  @Test
  void fazlaCekmeMesajiDogru() {
    BankaHesabi hesap = new BankaHesabi(100);
    IllegalStateException e = assertThrows(
      IllegalStateException.class, () -> hesap.cek(500));
    assertEquals("Yetersiz bakiye", e.getMessage());
  }

  @Test
  void negatifYatirmaReddedilir() {
    BankaHesabi hesap = new BankaHesabi(100);
    assertThrows(IllegalArgumentException.class,
      () -> hesap.yatir(-10));
  }

  @Test
  void yanlisBeklenti() {
    BankaHesabi hesap = new BankaHesabi(100);
    assertThrows(IllegalStateException.class,
      () -> hesap.cek(50));
  }
}
```

`assertThrows` yalnız istisnanın fırlatıldığını denetlemez; fırlatılan istisna nesnesini de döndürür. İlk test bu nesneyi `e` değişkenine alır ve iletisinin `"Yetersiz bakiye"` olduğunu ayrıca doğrular. İkinci test negatif yatırmanın reddedildiğini sınar. Üçüncü test ise bilerek yanlış yazılmıştır: 50 TL çekmek geçerli bir işlemdir ve istisna fırlatmaz. Sonuç şöyledir:

```
✓ fazlaCekmeMesajiDogru()
✓ negatifYatirmaReddedilir()
✗ yanlisBeklenti()
  Expected java.lang.IllegalStateException to be thrown, but nothing was thrown.
3 test · 2 başarılı · 1 başarısız
```

İleti açıktır: `IllegalStateException` beklendi, ama hiçbir istisna fırlatılmadı. `assertThrows` iki durumda kalır. Birincisi buradaki gibi hiç istisna fırlatılmamasıdır. İkincisi beklenenden farklı tipte bir istisna fırlatılmasıdır. Örneğin `assertThrows(IllegalStateException.class, () -> hesap.yatir(-10))` şu iletiyle kalır:

```
Unexpected exception type thrown, expected: <java.lang.IllegalStateException> but was:
<java.lang.IllegalArgumentException>
```

`assertThrows` alt tipleri de kabul eder. `Integer.parseInt("abc")` bir `NumberFormatException` fırlatır; bu sınıf `IllegalArgumentException`'ın alt sınıfıdır. Bu yüzden `assertThrows(IllegalArgumentException.class, () -> Integer.parseInt("abc"))` geçer. Tam tip isteniyorsa `assertThrowsExactly` kullanılır; bu metot aynı çağrıda kalır.

JUnit 4'te istisna testi farklı yazılırdı: `@Test(expected = IllegalStateException.class)`. Bu yazımda test metodunun herhangi bir satırı istisnayı fırlatırsa test geçer. `assertThrows` ise istisnanın tam olarak hangi çağrıdan beklendiğini belirtir ve istisna nesnesine erişim sağlar.

7.3.10 Test yaşam döngüsü

`@BeforeEach`, JUnit'in yaşam döngüsü etiketlerinden yalnız biridir. Dördü birlikte, testlerin çevresinde hangi kodun ne zaman çalışacağını belirler.

```
class YasamDongusuTest {

  @BeforeAll
  static void basla() { System.out.println("BeforeAll"); }

  @BeforeEach
  void hazirla() { System.out.println("  BeforeEach"); }

  @Test
  void testA() { System.out.println("    testA"); }

  @Test
  void testB() { System.out.println("    testB"); }

  @AfterEach
  void temizle() { System.out.println("  AfterEach"); }

  @AfterAll
  static void bitir() { System.out.println("AfterAll"); }
}
```

Bu sınıfın tek amacı, metotların çalışma sırasını ekrana yazdırmaktır. Çıktı şöyledir:

```
BeforeAll
  BeforeEach
    testA
  AfterEach
  BeforeEach
    testB
  AfterEach
AfterAll
```

Etiket	Ne zaman çalışır?	Tipik kullanım
`@BeforeAll`	Tüm testlerden önce bir kez (`static`)	Pahalı ortak kaynağı açmak
`@BeforeEach`	Her testten önce	Test nesnelerini yeniden kurmak
`@AfterEach`	Her testten sonra (test kalsa bile)	Geçici dosyayı silmek
`@AfterAll`	Tüm testlerden sonra bir kez (`static`)	Ortak kaynağı kapatmak

Tablo 7.7: JUnit 5 yaşam döngüsü etiketleri.

Çıktı Tablo 7.7'deki kuralları doğrular. `BeforeAll` ve `AfterAll` birer kez, en başta ve en sonda yazılır. `BeforeEach` ve `AfterEach` her testin çevresinde birer kez çalışır; iki test olduğu için ikişer kez görünür. `@AfterEach`'in test kalsa bile çalışması önemlidir: temizlik işi bu yüzden oraya konur. Testin ortasında fırlatılan bir istisna geçici dosyayı ortada bırakmaz.

`@BeforeAll` ve `@AfterAll` metotları `static` olmalıdır. Nedeni Tablo 7.6'da görülür: JUnit her test için ayrı bir nesne oluşturur. "Tüm testlerden önce bir kez" çalışacak metodu çağırmak için henüz hiçbir test nesnesi yoktur; metot sınıfa ait olmalıdır. Bu kural unutulursa JUnit sınıftaki hiçbir testi çalıştırmaz ve şu hatayı verir:

```
class StaticsizTest {
  @BeforeAll void basla() { }
  @Test void t() { }
}
```

```
org.junit.platform.commons.JUnitException: @BeforeAll method 'void StaticsizTest.basla()'
must be static unless the test class is annotated with @TestInstance(Lifecycle.PER_CLASS).
```

İletinin önerdiği `@TestInstance(Lifecycle.PER_CLASS)` etiketi, JUnit'e bütün testler için tek bir test nesnesi kullanmasını söyler. O zaman `@BeforeAll` örnek metodu olabilir; ama testlerin bağımsızlığı da zayıflar. Varsayılan davranış genellikle daha güvenlidir.

Testlerin kendi aralarındaki sırası belirlidir, ama kaynak koddaki sırayla aynı olmak zorunda değildir. Bu yüzden testler hiçbir zaman birbirine bağımlı yazılmaz. "Önce `testA` hesabı açar, sonra `testB` o hesaptan para çeker" gibi bir düzen, sıra değiştiğinde kırılır.

7.3.11 Kırmızıdan yeşile: başarısız test

Testin asıl değeri bir hata yakaladığında ortaya çıkar. Hata zinciri örneğindeki `enBuyuk` metodunun ilk sürümü şudur:

```
public class DiziIslem {
  public static int enBuyuk(int[] dizi) {
    int enb = 0;
    for (int i = 0; i < dizi.length; i++) {
      if (dizi[i] > enb) {
        enb = dizi[i];
      }
    }
    return enb;
  }
}
```

Kod derlenir ve ilk bakışta doğru görünür. Döngü her öğeyi `enb` ile karşılaştırır, daha büyüğünü bulunca `enb`'yi günceller. Bu metot için iki test yazalım: biri pozitif sayılardan, biri tamamen negatif sayılardan oluşan bir diziyle.

```
class DiziIslemTest {

  @Test
  void pozitifDizi() {
    int[] dizi = {3, 9, 4};
    assertEquals(9, DiziIslem.enBuyuk(dizi));
  }

  @Test
  void tumuNegatifDizi() {
    int[] dizi = {-5, -2, -8};
    assertEquals(-2, DiziIslem.enBuyuk(dizi));
  }
}
```

```
✓ pozitifDizi()
✗ tumuNegatifDizi()
  org.opentest4j.AssertionFailedError: expected: <-2> but was: <0>
2 test · 1 başarılı · 1 başarısız
```

Başarısız testin iletisi iki parçadan oluşur. `expected` beklenen değeri (−2), `but was` metodun gerçekte döndürdüğü değeri (0) gösterir. 0 dizide yoktur bile. Bu ipucu kusurun yerini neredeyse ele verir: 0 değeri dizinin içinden değil, başka bir yerden gelmiştir. Tek aday `enb = 0` başlangıcıdır. Tüm öğeler negatif olduğunda hiçbir öğe 0'ı aşamaz ve `enb` hiç güncellenmez.

Test dünyasında başarısız teste *kırmızı*, başarılı teste *yeşil* denir; IDE'ler sonuçları bu renklerle gösterir. Kusurun düzeltilmiş sürümü şudur:

```
public class DiziIslem {
  public static int enBuyuk(int[] dizi) {
    int enb = dizi[0];               // ilk eleman
    for (int i = 1; i < dizi.length; i++) {
      if (dizi[i] > enb) {
        enb = dizi[i];
      }
    }
    return enb;
  }
}
```

En büyük değer artık dizinin ilk öğesiyle başlatılır. Döngü de 1. indisten başlar, çünkü 0. öğe zaten `enb`'dedir. Aynı iki test yeniden çalıştırılır ve ikisi de geçer:

```
✓ pozitifDizi()
✓ tumuNegatifDizi()
2 test · 2 başarılı
```

`i`	`dizi[i]`	**İlk sürüm: koşul**	**İlk sürüm:** `enb`	**Düzeltilmiş: koşul**	**Düzeltilmiş:** `enb`
başlangıç	—	—	0	—	−5
0	−5	−5 > 0 yanlış	0	(döngü 1'den başlar)	−5
1	−2	−2 > 0 yanlış	0	−2 > −5 doğru	−2
2	−8	−8 > 0 yanlış	0	−8 > −2 yanlış	−2
dönüş			**0 ✗**		**−2 ✓**

Tablo 7.8: `{-5, -2, -8}` dizisi için iki sürümün izlemesi.

Tablo 7.8 iki sürümün farkını adım adım gösterir. İlk sürümde koşul hiçbir turda doğru olmaz. Düzeltilmiş sürümde ilk turda (`i = 1`) −2, −5'ten büyük çıkar ve `enb` güncellenir. Döngü ilk sürümde dizinin uzunluğu kadar, düzeltilmiş sürümde bir eksik döner; zaman maliyeti O(n), ek bellek O(1)'dir.

Bu testin bir değeri daha vardır: artık kalıcıdır. Kod bir gün yeniden değiştirilir ve aynı kusur geri gelirse, `tumuNegatifDizi` testi onu hemen yakalar. Düzeltmeden sonra aynı testlerin yeniden çalıştırılıp yeşile dönmesi, düzeltmenin işe yaradığının kanıtıdır.

Sık yapılan hata

Düzeltilmiş sürüm boş dizi için de bir soru doğurur. `dizi[0]` boş dizide `ArrayIndexOutOfBoundsException` fırlatır. İlk sürüm aynı girdi için sessizce 0 döndürürdü; bu da yanlış bir cevaptır, çünkü boş dizinin en büyük öğesi yoktur. Bir metodun boş girdide ne yapacağı açıkça kararlaştırılmalı ve bu karar da bir testle sınanmalıdır, örneğin `assertThrows(ArrayIndexOutOfBoundsException.class, () -> DiziIslem.enBuyuk(new int[0]))`.

7.4 İyi test tasarımı

JUnit testleri yazmayı kolaylaştırır, ama hangi testlerin yazılacağına karar vermez. Bir metodun girdi uzayı çoğu zaman çok büyüktür: `int` tipinde tek bir parametre bile dört milyardan fazla değer alabilir. Hepsini denemek imkânsızdır. İyi test tasarımı, az sayıda ama doğru seçilmiş girdiyle hataları yakalama sanatıdır.

7.4.1 Sınır değerleri ve denklik sınıfları

Örnek metot, 0–100 arasındaki bir puanı harf notuna çevirir.

```
public class Not {
  public static String harfNotu(int puan) {
    if (puan < 0 || puan > 100) {
      throw new IllegalArgumentException("Geçersiz puan: " + puan);
    }
    if (puan >= 90) return "AA";
    if (puan >= 80) return "BA";
    if (puan >= 70) return "BB";
    if (puan >= 60) return "CB";
    if (puan >= 50) return "CC";
    return "FF";
  }
}
```

Metot önce aralık dışı puanları reddeder, sonra yukarıdan aşağıya eşikleri dener. Hangi puanlarla test edilmeli? 101 farklı geçerli puan vardır; hepsini yazmak hem sıkıcıdır hem de gereksizdir. 61, 65 ve 68 puanları aynı yoldan geçer ve aynı sonucu (CB) verir. Biri doğruysa ötekilerin de doğru olması beklenir.

Tanım 7.7

Denklik sınıfı (*equivalence class*), program için aynı davranışa yol açan girdilerin kümesidir. Bir denklik sınıfından tek bir temsilci seçmek yeterlidir. **Sınır değer analizi** (*boundary value analysis*) ise her denklik sınıfının uç değerlerini ve geçersiz bölgenin ilk değerlerini test girdisi olarak seçer.



● test edilen değer: her aralığın iki ucu ve geçersiz bölgenin ilk değeri

Şekil 7.7: 0–100 puan doğrusu. FF, CC, CB, BB, BA ve AA aralıkları; işaretli noktalar test edilen sınır değerleridir. İki uçtaki ✗ bölgeleri geçersiz puanlardır.

Sınır değerlerinin neden bu kadar önemli olduğu bir gözlemle açıklanır: hatalar en çok sınırlarda olur. Programcılar `>` yerine `>=`, `< 100` yerine `<= 100` yazmayı sık sık karıştırır. Bu tür bir kusur, aralığın ortasındaki girdilerde hiç görünmez. 65 puanı `puan >= 60` ile de `puan > 60` ile de CB verir. Fark yalnız tam sınırda, 60'ta ortaya çıkar.

Şekil 7.7 seçilen değerleri gösterir. Her aralığın iki ucu sınanır: 49 ve 50, FF ile CC arasındaki sınırı iki yandan kuşatır; 89 ve 90, BA ile AA arasındakini. Geçersiz bölgenin ilk değerleri (−1 ve 101) de seçilir, çünkü aralık denetimindeki bir kusur ancak onlarla görülür. Geçerli bölgenin uçları (0 ve 100) da unutulmaz.

Örnek 7.3

`harfNotu` için eksiksiz bir sınır değer kümesi nasıl kurulur?

Denklik sınıfları: geçersiz alt (< 0), FF (0–49), CC (50–59), CB (60–69), BB (70–79), BA (80–89), AA (90–100), geçersiz üst (> 100). Toplam sekiz sınıf vardır.

Her geçerli sınıfın iki ucu alınır: 0, 49, 50, 59, 60, 69, 70, 79, 80, 89, 90, 100. Geçersiz bölgenin ilk değerleri eklenir: −1 ve 101. Toplam 14 girdi çıkar.

Bu küme 101 olası geçerli puanın yalnız 12'sini dener, ama her eşiği iki yandan sınar. Herhangi bir `>=` yanlışlıkla `>` yazılsaydı, o eşiğin alt ucu yanlış harfi verirdi ve test kalırdı.

7.4.2 Parametreli testler

Sınır değerleri seçildikten sonra her biri için ayrı bir test metodu yazmak gerekir mi? Her değer için neredeyse aynı bir metot yazmak hem uzun hem de kopyala-yapıştır hatalarına açıktır. JUnit 5 bunun yerine parametreli testi sunar.

```
class NotTest {

  @ParameterizedTest
  @CsvSource({"0, FF", "49, FF", "50, CC", "59, CC",
        "60, CB", "89, BA", "90, AA", "100, AA"})
  void sinirDegerleri(int puan, String beklenen) {
    assertEquals(beklenen, Not.harfNotu(puan));
  }

  @Test
  void aralikDisiPuanReddedilir() {
    assertThrows(IllegalArgumentException.class,
      () -> Not.harfNotu(-1));
    assertThrows(IllegalArgumentException.class,
      () -> Not.harfNotu(101));
  }
}
```

`@ParameterizedTest` etiketi, aynı test metodunun farklı verilerle tekrar tekrar çalışacağını belirtir. Veriler `@CsvSource` ile verilir. Her dizgi bir test çağrısıdır. Virgülle ayrılan değerler sırayla metodun parametrelerine dönüştürülür: `"50, CC"` dizgisinde `"50"` bir `int`'e çevrilip `puan`'a, `"CC"` ise `beklenen`'e verilir. Ad, CSV (*comma-separated values*, virgülle ayrılmış değerler) dosya biçiminden gelir.

Parametreli testler `junit-jupiter-params` modülündedir. Gradle'a eklenen `junit-jupiter` paketi bu modülü de içerir; ayrıca bir şey eklemek gerekmez. Sınıf çalıştırıldığında rapor her veri satırını ayrı bir test olarak gösterir:

```
sinirDegerleri(int, String)
  ✓ [1] 0, FF
  ✓ [2] 49, FF
  ✓ [3] 50, CC
  ✓ [4] 59, CC
  ✓ [5] 60, CB
  ✓ [6] 89, BA
  ✓ [7] 90, AA
  ✓ [8] 100, AA
✓ aralikDisiPuanReddedilir()
9 test · 9 başarılı
```

Raporda 9 test vardır: parametreli testin 8 çağrısı ve aralık dışı puanları sınayan sıradan test. Kalan bir satır olsaydı, köşeli parantez içindeki sıra numarası ve değerler hangi verinin sorun çıkardığını hemen gösterirdi. Aralık dışı test, `assertThrows`'u iki kez çağırır: biri −1, biri 101 için.

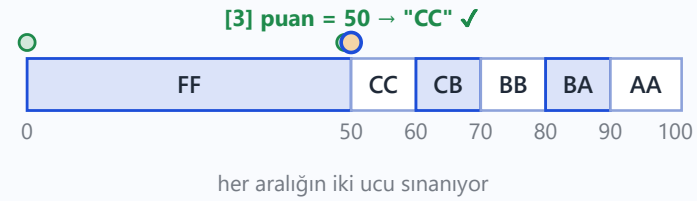
Örnek 7.3'teki eksiksiz kümeyle karşılaştırıldığında bu test birkaç sınırı atlar: 69/70 (CB/BB) ve 79/80 (BB/BA) çiftleri listede yoktur. BB aralığı hiç sınanmaz. `@CsvSource`'a dört dizgi daha eklemek bu boşluğu kapatır; test metodunun kendisi hiç değişmez. Parametreli testin asıl gücü budur: yeni bir durum eklemek tek satırlık iştir.

7.4.3 Kod izleme: parametreli test

Parametreli testin çalışmasını puan ekseni üzerinde izleyelim. JUnit önce `@CsvSource`'taki 8 dizgiyi okur; her biri ayrı bir test çağrısı olacaktır.



Şekil 7.8: Satır 1. `"0, FF"` virgülden bölünür; `"0"` `int`'e çevrilip `puan`'a, `"FF"` `beklenen`'e verilir. Sonuç `"FF"`, test geçer.



Şekil 7.9: Satır 3. 50, CC aralığının alt ucudur; 49 ile birlikte FF/CC sınırını iki yandan sınar.

Çağrı	`puan`	`beklenen`	İlk doğru koşul	Dönen	Rolü	Sonuç
[1]	0	FF	hiçbiri	FF	geçerli bölgenin ilk değeri	✓
[2]	49	FF	hiçbiri	FF	FF'nin üst ucu	✓
[3]	50	CC	`puan >= 50`	CC	CC'nin alt ucu	✓
[4]	59	CC	`puan >= 50`	CC	CC'nin üst ucu	✓
[5]	60	CB	`puan >= 60`	CB	CB'nin alt ucu	✓
[6]	89	BA	`puan >= 80`	BA	BA'nın üst ucu	✓
[7]	90	AA	`puan >= 90`	AA	AA'nın alt ucu	✓
[8]	100	AA	`puan >= 90`	AA	geçerli bölgenin son değeri	✓

Tablo 7.9: Parametreli testin sekiz çağrısı ve `harfNotu` içinde ilk doğru çıkan koşul.

Tablo 7.9 her çağrının metodun içinde hangi yoldan geçtiğini gösterir. 49 satırı ilginçtir: koşullardan hiçbiri doğru çıkmaz ve metot en alttaki `return "FF"`'ye ulaşır. Koşul `puan >= 50` yerine yanlışlıkla `puan > 48` yazılsaydı, 49 CC verirdi ve [2] kalırdı. 100 satırı ise `puan > 100` denetiminin doğru yazıldığını kanıtlar. Denetim yanlışlıkla `puan >= 100` olsaydı, 100 puan istisna fırlatırdı.

Örnek 7.4

`harfNotu` içinde `puan >= 50` yerine yanlışlıkla `puan > 50` yazılsın. Parametreli testin hangi satırı kalır?

49: hiçbir koşul doğru değil, FF. Beklenen FF; geçer.

50: `50 > 50` yanlış; metot FF döndürür. Beklenen CC; **kalır**: `expected: <CC> but was: <FF>`.

59: `59 > 50` doğru, CC. Beklenen CC; geçer.

Yalnız [3] kalır. 49 ve 59 bu kusuru göremez; hatayı yalnız tam sınırdaki değer yakalar.

7.4.4 İyi bir birim testinin özellikleri: FIRST

İyi bir birim testinin özellikleri, İngilizce baş harflerinden oluşan FIRST kısaltmasıyla özetlenir.

F · Hızlı (*fast*)
: Milisaniyeler içinde biter; binlercesi her değişiklikten sonra çalıştırılabilir. Birim testinde dosya, ağ ve veri tabanı kullanılmaz. Bir test kümesi beş dakika sürerse programcılar onu çalıştırmayı bırakır.

I · Bağımsız (*independent*)
: Başka bir teste, test sırasına ya da önceki testin bıraktığı duruma bağlı değildir. JUnit'in her test için yeni nesne oluşturması bu özelliği destekler.

R · Tekrarlanabilir (*repeatable*)
: Her makinede, her zaman aynı sonucu verir. Rastgelelik ve saat gibi değişken kaynaklar kontrol altına alınır.

S · Kendini doğrulayan (*self-validating*)
: Sonuç ✓ ya da ✗'dir. Çıktıyı bir insanın okuyup yorumlamasına gerek yoktur.

T · Zamanında (*timely*)
: Kodla birlikte, hatta koddan önce yazılır. Aylar sonraya bırakılan test genellikle hiç yazılmaz.

Bu beşine bir özellik daha eklenir: okunabilirlik. Açıklayıcı bir ad, AAA düzeni ve test başına tek davranış testi okunur kılar. İyi yazılmış bir test, kodun nasıl kullanılacağını gösteren bir belgedir. Yeni gelen bir programcı `BankaHesabiTest`'i okuyarak `cek` metodunun yetersiz bakiyede ne yaptığını öğrenir.

7.4.5 Birim testi mi, tümleştirme testi mi?

Gerçek sınıflar çoğu zaman tek başına çalışmaz. Bir sipariş sınıfı fiyatları veri tabanından okur, sonucu bir dosyaya yazar, e-posta gönderir. Böyle bir sınıfı gerçek veri tabanıyla birlikte sınayan test artık birim testi değildir.

	Birim testi	**Tümleştirme (entegrasyon) testi**
Kapsam	Tek sınıf / metot, yalıtılmış	Birlikte çalışan birkaç bileşen
Hız	Milisaniyeler	Saniyeler (veri tabanı, dosya, ağ)
Dış bağımlılık	Yerine basit bir sahte nesne (test dublörü) konur	Gerçek veri tabanı, dosya, servis
Bulduğu hata	Yerel mantık hatası; yeri hemen belli	Uyumsuz arayüz, yanlış yapılandırma

Tablo 7.10: Birim testi ile tümleştirme testinin karşılaştırması.

Tümleştirme testi (*integration test*), birlikte çalışan bileşenleri gerçek bağımlılıklarıyla sınar. Modül ve alt sistem testleri bu türdendir. Birim testinde ise dış bağımlılığın yerine onu taklit eden basit bir nesne konur. Bu nesneye **test dublörü** (*test double*) denir; adını, tehlikeli sahnelerde oyuncunun yerini alan dublörden alır. Örneğin fiyatı veri tabanından okuyan bir arayüzün yerine, her ürün için sabit fiyat döndüren küçük bir sınıf yazılır. Test hem hızlanır hem de veri tabanının o anki içeriğinden bağımsız olur.

Tablo 7.10 iki test türünün birbirinin yerini tutmadığını da gösterir. Birim testleri mantık hatalarını hızlı bulur ve yerlerini kesin olarak gösterir. Tümleştirme testleri ise parçaların birbirine doğru bağlandığını doğrular. Test piramidindeki gibi birim testleri çok, tümleştirme testleri daha az sayıda yazılır.

Test adları da davranışı anlatmalıdır. Aşağıdaki üç addan ilki hiçbir şey söylemez; öteki ikisi test kaldığında hangi kuralın bozulduğunu açıklar.

```
@Test void test1() { … }
@Test void altiUrundeIndirimVar() { … }
@Test void besUrundeIndirimYok() { … }
```

Birim testinde gerçeklemenin iç ayrıntısı değil, dışarıdan görünen davranış sınanır. Bir metodun içinde hangi yardımcı metodu çağırdığı değil, ne döndürdüğü önemlidir. İç ayrıntıya bağlı testler, davranış değişmese bile her yeniden düzenlemede kırılır. Birim testinde `Thread.sleep`, gerçek ağ ve dosya kullanılmaz; test hem yavaşlar hem de rastgele kalmaya başlar. Testler arasında `static` bir liste gibi değişen bir durum da paylaşılmaz.

7.5 Regresyon ve otomasyon

Bir testin değeri yazıldığı gün bitmez. Kod değiştikçe test yeniden çalışır ve eski davranışın korunup korunmadığını denetler. Testleri bu kadar değerli kılan, otomatik olmaları ve her değişiklikten sonra yeniden çalıştırılabilmeleridir.

7.5.1 Regresyon testi

Tanım 7.8

Regresyon testi (*regression testing*), yapılan değişikliklerden sonra önceden çalışan her şeyin hâlâ çalıştığını doğrulamaktır. Regresyon "geriye gidiş" demektir: düzelmiş bir şeyin yeniden bozulması. Yöntem basittir: daha önce yazılmış bütün testler her değişiklikten sonra yeniden çalıştırılır.



Şekil 7.10: Sürüm 1'de altı test geçiyor. Bir değişiklikten sonra sürüm 2'de `cek` testi kalıyor: değişiklik eski bir davranışı bozmuştur.

Şekil 7.10 tipik bir regresyonu gösterir. Programcı yeni bir özellik eklerken `cek` metoduna dokunmuş ve farkında olmadan onu bozmuştur. Testler olmasaydı bu kusur ancak bir müşteri parasını çekemediğinde fark edilirdi. Regresyon testi kusuru değişikliğin hemen ardından, değişikliği yapan kişinin aklı hâlâ o koddayken yakalar.

Regresyon testinin işe yaraması için testlerin iki özelliği olmalıdır.

- **Belirleyici** (*deterministic*): aynı koşullarda her seferinde aynı sonucu vermelidir. Bazen geçip bazen kalan teste *kararsız* (*flaky*) test denir. Kararsız test güveni yok eder: kaldığında "yine o test" denip geçilir ve gerçek bir hata da gözden kaçar.
- **Tekrarlanabilir** (*repeatable*): farklı zamanlarda, farklı makinelerde ve farklı durumlarda aynı biçimde tekrar edilebilmelidir. Yalnız programcının bilgisayarında geçen bir test, sunucuda kaldığında bir şey söylemez.

7.5.2 Rastgelelik içeren kodu belirleyici test etmek

Rastgele sayı üreten bir kod nasıl test edilir? Bir zar sınıfı her atışta 1–6 arasında farklı bir sayı döndürür; beklenen sonucu önceden yazmak imkânsız gibi görünür. Çözüm, rastgeleliğin kaynağını sınıfın dışına çıkarmaktır.

```
import java.util.Random;

public class Zar {
  private final Random rastgele;

  public Zar(Random rastgele) {      // dışarıdan verilir
    this.rastgele = rastgele;
  }

  public int at() {
    return rastgele.nextInt(6) + 1;
  }
}
```

`Zar` sınıfı kendi `Random` nesnesini oluşturmaz; onu yapıcısından alır. Bu tasarıma **bağımlılık enjeksiyonu** (*dependency injection*) denir: bir nesnenin ihtiyaç duyduğu başka bir nesne dışarıdan verilir. Program gerçek kullanımda `new Zar(new Random())` yazar ve her seferinde farklı sayılar alır. Test ise sabit bir tohumla oluşturulmuş bir `Random` verir.

`Random` sınıfı gerçekte rastgele değil, sözde rastgele (*pseudo-random*) sayılar üretir. Sayılar bir formülle, başlangıç değerinden türetilir. Bu başlangıç değerine **tohum** (*seed*) denir. Aynı tohumla oluşturulan iki `Random` nesnesi aynı sayı dizisini üretir. 42 tohumuyla yapılan beş atış her makinede ve her çalıştırmada şu sonucu verir:

```
3 4 1 3 1
```

```
class ZarTest {

  @Test
  void ayniTohumAyniSonuclar() {
    Zar z1 = new Zar(new Random(42));
    Zar z2 = new Zar(new Random(42));
    for (int i = 0; i < 100; i++) {
      assertEquals(z1.at(), z2.at());
    }
  }
}
```

Test aynı tohumla iki zar oluşturur ve 100 atış boyunca ikisinin hep aynı sayıyı verdiğini doğrular. Test her çalıştırmada aynı sonucu verir; yani belirleyicidir. Zar sınıfının içinde `new Random()` yazılmış olsaydı, testin hiçbir şekilde tohumu kontrol etme imkânı kalmazdı.

Belirleyicilik başka testler de yazmayı mümkün kılar. `assertEquals(3, new Zar(new Random(42)).at())` her zaman geçer, çünkü 42 tohumunun ilk atışı her zaman 3'tür. Aynı doğrulama tohumsuz `new Random()` ile yazılsaydı test ancak 1/6 olasılıkla geçerdi: tam bir kararsız test. Tohumsuz zarla yazılabilecek anlamlı bir test ise sonucun aralığını denetler: `int x = z.at(); assertTrue(x >= 1 && x <= 6);` türünden bir doğrulama her zarda doğrudur.

> **Sık yapılan hata**
>
> Saate bağlı kodu doğrudan test etmek. `LocalDate.now()` çağıran bir "doğum günü indirimi" metodu, testin hangi gün çalıştırıldığına göre farklı sonuç verir. Test yazıldığı gün geçer, ertesi gün kalır. Çözüm zar örneğindekiyle aynıdır: tarih ya da saat kaynağı dışarıdan verilir ve test sabit bir tarih kullanır.

7.5.3 Otomatik testler ve sürekli entegrasyon

Kod elle test edilmek yerine otomatik testlerle sınanmalıdır. Harcanan çaba aşağı yukarı aynıdır: elle deneme için de girdiler seçilir, çıktılar incelenir. Fark sonunda ortaya çıkar. Elle denemeden geriye hiçbir şey kalmaz; otomatik denemeden ise istenildiği zaman yeniden çalıştırılabilen bir test kalır. Bu test her derlemede, her değişiklikte tekrar çalışabilir.

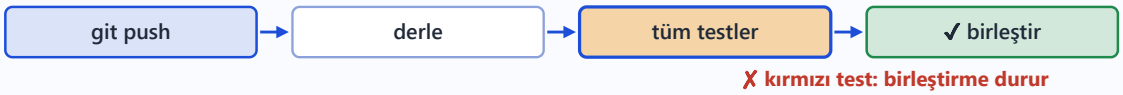
Yazılım bir kez doğru çalıştı diye bir daha test etmenin gereksiz olduğunu düşünmek yanlıştır. Kod değişir, kullanılan kütüphaneler yeni sürümlere geçer, programın çalıştığı ortam değişir. Dün doğru çalışan program bugün bozulabilir.

	Elle	Otomatik
İlk maliyet	Düşük	Biraz yüksek
Tekrar maliyeti	Her seferinde aynı	≈ 0
Hız	Dakikalar	Saniyeler
Güvenilirlik	İnsan hatası	Her seferinde aynı
Regresyon	Pratikte yapılmaz	Her değişiklikte
Belge değeri	Yok	Kullanım örneği

Tablo 7.11: Elle test ile otomatik testin karşılaştırması.

Tablo 7.11 iki yaklaşımı karşılaştırır. Otomatik testin ilk maliyeti biraz yüksektir: test kodu yazılmalıdır. Ama her tekrarın maliyeti neredeyse sıfırdır. Bir test 100 kez çalıştırılacaksa, otomatik olanı açık ara daha ucuzdur. Regresyon testi pratikte yalnız otomatik testlerle yapılabilir; hiçbir programcı her değişiklikten sonra yüz durumu elle denemez.

Otomasyonun son adımı testlerin programcıya hiç sorulmadan çalışmasıdır. **Sürekli entegrasyon** (*continuous integration*, CI) sistemleri bunu yapar. GitHub Actions ve Jenkins bu sistemlerin yaygın örnekleridir.



Şekil 7.11: Sürekli entegrasyon hattı. Her gönderimde kod sunucuda derlenir, testler çalışır, sonuç raporlanır. Kırmızı bir test birleştirmeyi durdurur.

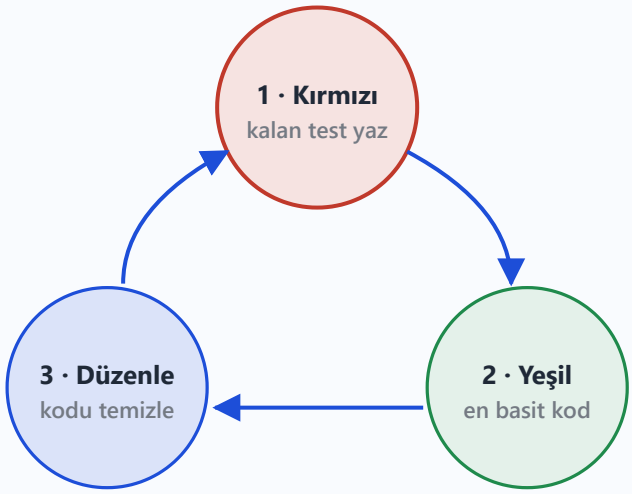
Şekil 7.11 tipik bir hattı gösterir. Programcı değişikliğini ortak depoya gönderir (`git push`). CI sunucusu kodu indirir, derler ve bütün testleri çalıştırır. Testlerden biri kalırsa değişiklik ana koda birleştirilmez ve programcıya haber verilir. Böylece hiçbir regresyon fark edilmeden ana koda giremez.

7.5.4 Test güdümlü geliştirme

Testler genellikle koddan sonra yazılır. Bazı programcılar sırayı tersine çevirir: önce test, sonra kod.

Tanım 7.9

Test güdümlü geliştirme (*test-driven development*, TDD), testi koddan önce yazan geliştirme yöntemidir. Yöntem kısa döngülerle ilerler; her döngü üç adımdan oluşur: kırmızı, yeşil, yeniden düzenleme.



Şekil 7.12: TDD döngüsü. 1: kalan bir test yazılır (kırmızı). 2: testi geçiren en basit kod yazılır (yeşil). 3: kod temizlenir, testler yeşil kalır (yeniden düzenleme).

Şekil 7.12 döngüyü gösterir. **Kırmızı** adımında henüz var olmayan bir davranış için test yazılır. Test kalır, hatta metot henüz yazılmadığı için derlenmez bile. **Yeşil** adımında testi geçirecek en basit kod yazılır. Bu adımda kodun güzel olması gerekmez; testin geçmesi yeterlidir. **Yeniden düzenleme** (*refactoring*) adımında kod temizlenir: tekrarlar kaldırılır, adlar düzeltilir. Testler yeşil kaldığı sürece davranış korunur.

Örnek 7.5

`harfNotu` metodu TDD ile nasıl geliştirilir?

Döngü 1, kırmızı: `assertEquals("CC", Not.harfNotu(50))` testi yazılır. `Not` sınıfı henüz yoktur; test derlenmez.

Döngü 1, yeşil: en basit kod yazılır: `return "CC";` . Test geçer. Kod açıkça eksiktir, ama elimizdeki tek test bu kadarını ister.

Döngü 2, kırmızı: `assertEquals("FF", Not.harfNotu(49))` eklenir. Metot hep CC döndürdüğü için yeni test kalır.

Döngü 2, yeşil: `if (puan >= 50) return "CC"; return "FF";` . İki test de geçer.

Her yeni not aralığı yeni bir döngüdür. Sonunda `harfNotu` 'nun her satırı bir testle doğmuş olur ve regresyon testleri kendiliğinden birikir.

TDD'nin en büyük kazancı, test edilmemiş kod bırakmamasıdır. Bir yan kazancı da tasarımı iyileştirmesidir. Test önce yazıldığı için programcı sınıfı önce bir kullanıcı gözüyle düşünür: metodun adı ne olmalı, hangi parametreleri almalı, hatalı girdide ne yapmalı? Test edilmesi zor bir sınıf genellikle kötü tasarlanmış bir sınıftır; TDD bunu kod yazılmadan fark ettirir.

7.6 Hata ayıklama

Test bir arızayı haber verir: `expected: <-2> but was: <0>` . Ama kusurun hangi satırda olduğunu her zaman söylemez. Küçük bir metotta kusur kodu okuyarak bulunabilir; yüzlerce satırlık bir programda ise program çalışırken içine bakmak gerekir. Bu işin aracı hata ayıklayıcıdır.

7.6.1 Hata ayıklayıcı

Tanım 7.10

Hata ayıklayıcı (*debugger*), çalışan bir programın durumunu incelemeyi sağlayan araçtır. Programı istenen satırda durdurur, satır satır ilerletir ve her an değişkenlerin değerlerini gösterir. Programın durdurulacağı satıra **kesme noktası** (*breakpoint*) denir.

Bir hata ayıklayıcıyla yapılabilecekler şunlardır:

- programı komut komut (satır satır) çalıştırmak;
- çalışan programın kaynak kodunu o anki satırla birlikte görmek;
- değişkenlerin değerlerini görüntülemek ve gerekirse değiştirmek;
- programa kesme noktaları eklemek ve programı bir kesme noktasına kadar çalıştırmak;
- çökmüş ya da durdurulmuş bir programın durumunu bir döküm dosyası üzerinden incelemek.

Son madde biraz açıklama ister. Program çöktüğünde ya da donduğunda o anki durum bir dosyaya yazılabilir. C dilinde bu dosyaya *core file* denir. Java'da iki tür döküm vardır. Heap dökümü (`.hprof` dosyası) heap'teki bütün nesneleri içerir. İş parçacığı dökümü ise her iş parçacığının o anda hangi satırda olduğunu gösterir. Döküm, programın o anki fotoğrafıdır; program yeniden çalıştırılmadan incelenebilir.



Şekil 7.13: Kesme noktasında duran program. Kırmızı nokta kesme noktasını, vurgulu satır programın o an durduğu yeri gösterir. Sağdaki panelde değişkenlerin anlık değerleri görünür: `i = 2`, `toplam = 1`.

Şekil 7.13 bir IDE'nin hata ayıklama görünümünü sadeleştirerek gösterir. Program `toplam += i;` satırında durmuştur. Satır henüz çalışmamıştır: döngünün ilk turunda `toplam` 1 olmuş, şimdi ikinci tur başlamıştır. Kesme noktasında duran program istenildiği kadar bekler. Programcı değişkenlere bakar, bir sonraki adıma geçer ya da programı devam ettirir.

IntelliJ IDEA, Eclipse, NetBeans ve VS Code'un hepsinde hata ayıklayıcı yerleşiktir. Kesme noktası satır numarasının yanına tıklanarak konur, program da *Debug* düğmesiyle başlatılır. Sıradan *Run* düğmesiyle başlatılan program kesme noktalarında durmaz.

Hata ayıklayıcının yaygın bir alternatifi `println` ile ayıklamadır: şüpheli yerlere `System.out.println("i = " + i)` gibi satırlar eklenir. Bu yöntem hızlıdır ve her yerde çalışır. Ama kodu kirletir, her değişiklikte yeniden derleme ister ve eklenen satırların sonradan silinmesi unutulur. Hata ayıklayıcı ise kodu hiç değiştirmez; hangi değişkenin izleneceğine program çalışırken karar verilir.

7.6.2 Kesme noktaları ve adım komutları

Program bir kesme noktasında durduktan sonra adım komutlarıyla ilerletilir. Komutların adları IDE'lerde İngilizcedir; kısayolları IDE'ye göre değişir.

Komut	Ne yapar?	IntelliJ	Eclipse
Kesme noktası (*breakpoint*)	Program bu satıra gelince, satır *çalışmadan önce* durur	Ctrl+F8	Ctrl+Shift+B
Step Over (üzerinden geç)	Satırı çalıştırır; metot çağrısı varsa içine girmeden tamamlar	F8	F6
Step Into (içine gir)	Satırdaki metot çağrısının içine girer	F7	F5
Step Out (dışarı çık)	Bulunulan metodu bitirir, çağıran metoda döner	Shift+F8	F7
Resume (devam)	Bir sonraki kesme noktasına kadar çalışır	F9	F8
Evaluate / Watch	Bir ifadenin (ör. `dizi[i] > enb`) değerini anlık hesaplar, sürekli izler	Alt+F8	Ctrl+Shift+I

Tablo 7.12: Hata ayıklayıcı komutları ve varsayılan kısayolları.

Tablo 7.12'deki komutların en çok karıştırılanları Step Over ile Step Into'dur. İkisi de bulunulan satırı çalıştırır. Fark, satırda bir metot çağrısı olduğunda ortaya çıkar. Step Into o metodun içine girer ve ilk satırında durur. Step Over ise metodu tek bir adımda çalıştırır ve bir sonraki satırda durur. Çağrılan metodun doğru çalıştığından emin olunuyorsa Step Over zaman kazandırır. Kusurun o metodun içinde olduğundan şüpheleniliyorsa Step Into kullanılır.

Kısayollar varsayılan tuş eşlemesine göredir; menüden de (Run / Debug) aynı komutlara ulaşılır. Tablodaki kısayolları ezberlemek gerekmez. Komutların ne yaptığını bilmek yeterlidir.

Sıradan kesme noktasının iki güçlü çeşidi vardır. **Koşullu kesme noktası** yalnız verilen koşul doğruysa durur. Bir döngü 10.000 kez dönüyorsa ve kusur yalnız `i` 999 iken görünüyorsa, kesme noktasına `i == 999` koşulu yazılır. Program 999 tur boyunca durmadan çalışır ve tam aranan anda durur. **İstisna kesme noktası** ise belirli bir istisna (ör. `NullPointerException`) fırlatıldığı anda durur. İstisnanın yakalandığı yerde değil, fırlatıldığı yerde durulduğu için o anki değişkenler henüz kaybolmamıştır.

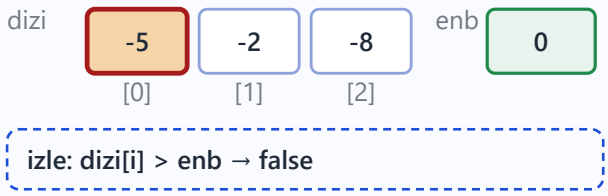
7.6.3 Kod izleme: hata ayıklayıcıyla kusuru bulmak

Hata zincirindeki arızaya bu kez hata ayıklayıcıyla yaklaşalım. Kodu okumadan, yalnız ekrandaki `En yüksek: 0` arızasından yola çıkarak kusuru bulmak istiyoruz.

```
public class HataAyikla {
  public static int enBuyuk(int[] dizi) {
    int enb = 0;
    for (int i = 0; i < dizi.length; i++) {
      if (dizi[i] > enb) {
        enb = dizi[i];
      }
    }
    return enb;
  }

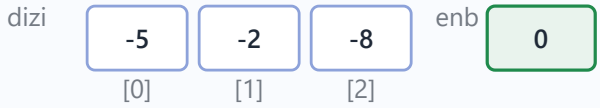
  public static void main(String[] args) {
    int[] sicaklik = {-5, -2, -8};
    int sonuc = enBuyuk(sicaklik);
    System.out.println("En yüksek: " + sonuc);
  }
}
```

Kesme noktası 5. satıra, `if (dizi[i] > enb)` koşuluna konur. `main`'in ilk satırına (13) da bir kesme noktası konur ve program Debug ile başlatılır. Program 13. satırda durur; bir Step Over ile 14. satırdaki `enBuyuk(sicaklik)` çağrısına gelinir. Step Into ile `enBuyuk`'un içine girilir. Yığına yeni bir çerçeve eklenir; bu çerçevedeki `dizi` parametresi, `main`'deki `sicaklik` dizisinin aynısını gösterir. Dizi kopyalanmamıştır, yalnız referansı aktarılmıştır. Resume ile program kesme noktasına kadar ilerler.



Şekil 7.14: İlk durma. `i = 0`, `dizi[0] = -5`, `enb = 0`. İzlenen ifade `dizi[i] > enb` yanlış (`false`) çıkar.

İzleme listesine `dizi[i] > enb` ifadesi eklenir. Hata ayıklayıcı bu ifadeyi her durmada yeniden hesaplar. İlk durmada ifade yanlıştır: −5, 0'dan büyük değildir (Şekil 7.14). Resume ile döngünün sonraki turlarına geçilir. `i = 1` için −2 > 0 yine yanlıştır; `enb` değişmez. `i = 2` için −8 > 0 da yanlıştır. Bu noktada şüphe belirginleşir: `enb` hiç güncellenmiyor.



Şekil 7.15: Döngü bitti. `enb` hâlâ 0'dır ve 0 döndürülür. Kusur bulundu: başlangıç değeri 0, tüm öğeler negatifse yanlış.

Adım	Komut	Satır	`i`	`dizi[i]`	`enb`	`dizi[i] > enb`
1	Debug	13 (`main`)	—	—	—	—
2	Step Over	14 (`main`)	—	—	—	—
3	Step Into	3	—	—	henüz yok	—
4	Resume	5	0	−5	0	`false`
5	Resume	5	1	−2	0	`false`
6	Resume	5	2	−8	0	`false`
7	Step Over (birkaç kez)	9 (`return`)	—	—	0	—

Tablo 7.13: Hata ayıklama oturumunun adımları.

Tablo 7.13 oturumu özetler. Üçüncü adımda yeni çerçeve açılır; `enb` henüz atanmamıştır. Son adımda döngü biter ve `return enb;` satırına gelinir; dönecek değer 0'dır. Kusurun yeri artık açıktır: `int enb = 0;` . Doğrusu `int enb = dizi[0];` 'dır. Program devam ettirildiğinde arıza ekranda görünür: `En yüksek: 0` .

Düzeltmeden sonra bu durum bir birim testi olarak eklenir. Hata ayıklayıcıyla bulunan her kusur, onu yakalayan bir testle kalıcı hâle getirilir. Böylece aynı kusur geri gelirse hata ayıklayıcıya bir daha gerek kalmaz; test onu hemen haber verir.

7.6.4 Step Into, Step Over, Step Out

Üç adım komutunun farkını iç içe metot çağrıları olan bir programda izleyelim.

```
public class AdimOrnek {
  static int kare(int x) {
    return x * x;
  }

  static int kareToplam(int a, int b) {
    int k1 = kare(a);
    int k2 = kare(b);
    return k1 + k2;
  }

  public static void main(String[] args) {
    int s = kareToplam(3, 4);
    System.out.println("s = " + s);
  }
}
```

Satır numaraları şöyledir: 1. satır sınıf başlığı, 3. satır `return x * x;`, 7. satır `int k1 = kare(a);`, 8. satır `int k2 = kare(b);`, 9. satır `return k1 + k2;`, 13. satır `int s = kareToplam(3, 4);`, 14. satır `println`, 15. satır `main`'in kapanış ayracıdır. Kesme noktası 13. satırdadır ve program orada durmuştur. Satır henüz çalışmamıştır.

Komut	Geçilen satır	Yığındaki çerçeveler	Görünen değişkenler	Açıklama
(kesme noktası)	13	`main`	—	satır henüz çalışmadı
Step Into	7	`main` → `kareToplam`	`a = 3`, `b = 4`	`kareToplam`'ın içine girildi
Step Over	8	`main` → `kareToplam`	`a`, `b`, `k1 = 9`	`kare(3)` çalıştı ama içine girilmedi
Step Into	3	`main` → `kareToplam` → `kare`	`x = 4`	bu kez `kare(4)`'ün içindeyiz
Step Out	8	`main` → `kareToplam`	`a`, `b`, `k1 = 9`	`kare` 16 döndürdü; atama tamamlanmak üzere
Step Over	9	`main` → `kareToplam`	`k1 = 9`, `k2 = 16`	atama tamamlandı
Step Out	13	`main`	—	`kareToplam` 25 döndürdü
Step Over	14	`main`	`s = 25`	atama tamamlandı
Step Over	15	`main`	`s = 25`	çıktı yazıldı: `s = 25`

Tablo 7.14: `AdimOrnek` programında adım komutlarının etkisi.

Tablo 7.14 üç komutun farkını açıkça gösterir. İlk Step Into, `kareToplam` çağrısının içine girer ve yığına yeni bir çerçeve ekler. Üçüncü sütundaki çerçeve zinciri, sözlükteki adıyla çağrı yığınıdır (*call stack*). Ardından gelen Step Over, 7. satırdaki `kare(3)` çağrısının içine girmez; çağrıyı bir bütün olarak çalıştırır ve 8. satıra geçer. `k1` artık 9'dur.

8. satırda bu kez Step Into seçilir ve `kare(4)` 'ün içine girilir. Yığında üç çerçeve vardır. `kare` çerçevesinde yalnız `x` görünür; `a`, `b` ve `k1` başka bir çerçevededir. Step Out, `kare` 'nin kalanını tek adımda çalıştırır ve çağıran satıra döner. Dikkat edilecek ince nokta şudur: Step Out'tan sonra hâlâ 8. satırdayız. Metot 16 döndürmüştür, ama bu değer henüz `k2` 'ye atanmamıştır. Atama ancak bir sonraki Step Over'da tamamlanır.

İkinci Step Out, `kareToplam` 'ı bitirir ve `main` 'deki 13. satıra döner; dönüş değeri 25'tir. Son iki Step Over önce atamayı (`s = 25`), sonra çıktıyı tamamlar. Bundan sonra Resume ile program sonlanır.

Örnek 7.6

Program 7. satırda (`int k1 = kare(a);`) duruyor ve `a = 3` . Art arda iki Step Over yapılırsa nerede olunur ve değişkenler ne olur?

Birinci Step Over `kare(3)` 'ü içine girmeden çalıştırır ve atamayı tamamlar: 8. satır, `k1 = 9` .

İkinci Step Over `kare(4)` 'ü çalıştırır: 9. satır, `k1 = 9` , `k2 = 16` .

Hiçbir zaman `kare` 'nin içine girilmez. Yığında hep iki çerçeve vardır: `main` ve `kareToplam` .

7.6.5 Komut satırında hata ayıklama: `jdb`

IDE'lerin hata ayıklayıcıları arka planda aynı düzeneği kullanır: JVM'in hata ayıklama arayüzü. JDK bu arayüzü komut satırından kullanan bir araçla gelir: `jdb` (*Java Debugger*). `jdb` , IDE'lerin düğmelerle yaptığını komutlarla yapar. IDE kurulu olmayan bir sunucuda ya da yalnız terminal erişimi olan bir makinede işe yarar.

```
public class Jdb {
  static int kare(int x) {
    int sonuc = x * x;
    return sonuc;
  }

  public static void main(String[] args) {
    int toplam = 0;
    for (int i = 1; i <= 3; i++) {
      toplam += kare(i);
    }
    System.out.println("Toplam: " + toplam);
  }
}
```

Program 1'den 3'e kadar sayıların karelerini toplar: 1 + 4 + 9 = 14. Hata ayıklayıcıyla çalıştırmak için önce `-g` seçeneğiyle derlenir, sonra `java` yerine `jdb` ile başlatılır:

```
javac -g Jdb.java
jdb Jdb
```

`-g` seçeneği yerel değişkenlerin adlarını da sınıf dosyasına yazar. Bu seçenek olmadan derlenen bir sınıfta `locals` komutu değişken adlarını gösteremez; çünkü sınıf dosyasında yalnız sıra numaraları vardır.

Komut	Anlamı
`stop at Sınıf:satır`	kesme noktası koyar
`clear Sınıf:satır`	kesme noktasını kaldırır
`run` / `cont`	başlatır / devam eder
`step` / `next` / `step up`	Step Into / Over / Out
`where`	çağrı yığınını gösterir
`locals`, `print x`	yerel değişkenleri, bir değeri gösterir

Tablo 7.15: Temel `jdb` komutları.

Tablo 7.15'teki komutlar IDE komutlarıyla birebir eşleşir. `step` Step Into'ya, `next` Step Over'a, `step up` Step Out'a, `cont` da Resume'a karşılık gelir.

`jdb` çalışan bir uygulamaya da bağlanabilir. Bunun için uygulama özel bir seçenekle başlatılır: `java -agentlib:jdwp=transport=dt_socket,server=y,suspend=y,address=5005 Uygulama`. Bu seçenek JVM'e hata ayıklama ajanını yükletir ve 5005 numaralı ağ kapısını dinletir. `suspend=y`, bir hata ayıklayıcı bağlanana kadar programın beklemesini sağlar. Ardından `jdb -attach 5005` komutu ya da bir IDE bu kapıya bağlanır. Uzak bir sunucudaki programda hata ayıklamanın yolu budur.

7.6.6 Bir `jdb` oturumu

Aşağıdaki oturum JDK 21 ile çalıştırılarak alınmıştır; `>` ve `main[1]` ile başlayan satırlar `jdb`'nin komut istemidir.

```
> stop at Jdb:4
Deferring breakpoint Jdb:4.
It will be set after the class is loaded.
> run
VM Started: Set deferred breakpoint Jdb:4
Breakpoint hit: "thread=main", Jdb.kare(), line=4 bci=4
main[1] where
  [1] Jdb.kare (Jdb.java:4)
  [2] Jdb.main (Jdb.java:10)
main[1] locals
Method arguments:
x = 1
Local variables:
sonuc = 1
```

`stop at Jdb:4` komutu 4. satıra, `return sonuc;`'a kesme noktası koyar. Sınıf henüz yüklenmediği için `jdb` kesme noktasını erteler; sınıf yüklenince yerleştirir. `run` programı başlatır. Program 4. satıra gelince durur. İletideki `bci=4`, metodun bayt kodunda kaçıncı baytta durulduğunu gösterir; kaynak düzeyinde çalışırken bu sayı önemli değildir.

`where` komutu çağrı yığınını gösterir: en üstte `kare` (4. satır), altında onu çağıran `main` (10. satır). `locals` ise o anki çerçevedeki değişkenleri listeler. `x = 1` metot argümanıdır, `sonuc = 1` yerel değişkendir. Bu ilk turdur: `i = 1` için `kare(1)` çağrılmıştır.

```
main[1] cont
Breakpoint hit: "thread=main", Jdb.kare(), line=4 bci=4
main[1] print sonuc
 sonuc = 4
main[1] clear Jdb:4
Removed: breakpoint Jdb:4
main[1] cont
> Toplam: 14
```

`cont` programı devam ettirir. Döngünün ikinci turunda `kare(2)` çağrılır ve program aynı kesme noktasında yeniden durur. `print sonuc` bu kez 4 gösterir. `clear Jdb:4` kesme noktasını kaldırır; son `cont` ile program bir daha durmadan sonuna kadar çalışır ve `Toplam: 14` yazar.

Komut	Durulan yer	`i` (`main`)	`x`	`sonuc`	`toplam`
`run`	`kare`, 4. satır	1	1	1	0
`cont`	`kare`, 4. satır	2	2	4	1
`clear` + `cont`	program biter	—	—	—	14

Tablo 7.16: `jdb` oturumunda programın durumu.

Tablo 7.16'daki `toplam` sütunu, kesme noktasında durulduğu anda `main`'deki değerdir. İkinci durmada `toplam` 1'dir, çünkü ilk turun sonucu eklenmiş, ikinci turunki henüz eklenmemiştir. Kesme noktası 3. satıra (`int sonuc = x * x;`) konsaydı, `locals` komutu `sonuc`'u göstermezdi: satır henüz çalışmadığı için değişken o anda yoktur.

7.6.7 Sistematik hata ayıklama

Hata ayıklayıcı güçlü bir araçtır, ama tek başına yöntem değildir. Rastgele kesme noktaları koyup kodu rastgele değiştirmek saatler sürebilir. Deneyimli programcılar beş adımlı bir yol izler.

1. **Yeniden üretmek.** Arızayı her seferinde ortaya çıkaran en küçük girdi bulunur, örneğin `{-5, -2, -8}`. Yeniden üretilemeyen bir hata ayıklanamaz.
2. **Test olarak yazmak.** Bu girdiyle kalan bir birim testi yazılır. Kırmızı test hatanın kanıtıdır ve düzeltmenin işe yaradığını da gösterecektir.
3. **Daraltmak.** Yığın izindeki ilk satırdan başlanır. Kesme noktasında değişkenler izlenir, sorunlu bölge ikiye bölünerek küçültülür. Aranan yer, bir değerin ilk kez yanlış olduğu satırdır.
4. **Hipotez kurmak ve sınamak.** "`enb` 0 ile başladığı için güncellenmiyor" gibi bir açıklama önerilir ve hata ayıklayıcıda doğrulanır. Tahminle kod değiştirmek yeni hatalar doğurur.
5. **Düzeltmek ve tüm testleri çalıştırmak.** Yeni test yeşile dönmeli, eskiler yeşil kalmalıdır; yani regresyon olmamalıdır. Test kalıcı olarak test kümesinde kalır.

Üçüncü adımdaki "ikiye bölme" fikri ikili aramaya benzer. 200 satırlık bir hesabın sonucu yanlışsa önce ortadaki, 100. satıra kesme noktası konur. Değerler orada doğruysa kusur ikinci yarıdadır; yanlışsa ilk yarıdadır. Her adımda aranan bölge yarıya iner. 200 satır en çok sekiz adımda tek satıra iner, çünkü $2^8 = 256 \geq 200$'dür.

Daraltmaya yardım eden bir başka araç Java'nın `assert` deyimidir. `assert`, programcının "bu noktada şu koşul her zaman doğrudur" varsayımını koda yazmasını sağlar. Koşul yanlışsa program `AssertionError` fırlatır ve sorun ortaya çıktığı yerde yakalanır.

```
static double ortalama(int[] d) {
  assert d.length > 0 : "boş dizi";
  int t = 0;
  for (int v : d) {
    t += v;
  }
  return (double) t / d.length;
}
```

```
public static void main(String[] a) {
  System.out.println(ortalama(new int[] {1, 2}));
  System.out.println(ortalama(new int[0]));
}
```

```
$ java As
1.5
NaN
$ java -ea As
1.5
Exception in thread "main" java.lang.AssertionError: boş dizi
        at As.ortalama(As.java:3)
        at As.main(As.java:12)
```

Çıktı önemli bir ayrıntıyı gösterir: `assert` deyimleri varsayılan olarak kapalıdır. Düz `java As` ile çalıştırılan program boş dizi için hiçbir uyarı vermez. 0 / 0 bölmesi `double`'da `NaN` (*not a number*, sayı değil) üretir; program sessizce yanlış bir değer yazar. `-ea` (*enable assertions*) seçeneğiyle çalıştırılınca varsayım denetlenir. Program boş diziyle çağrıldığı anda, 3. satırda `AssertionError: boş dizi` ile durur. Bu yüzden `assert` geliştirme ve test sırasında açılır, kullanıcıya giden programda ise kapalı kalır. Kullanıcı girdisini denetlemek için `assert` değil, istisna kullanılır; kapalı bir `assert` hiçbir şeyi engellemez.

7.7 Analiz araçları

Birim testi ve hata ayıklayıcı programcının en sık kullandığı iki araçtır, ama her soruyu cevaplamazlar. “Program neden yavaş?”, “Testlerim kodun ne kadarını deniyor?”, “Bellek neden doluyor?” gibi sorular için başka araçlar gerekir. Bu kesimdeki araçların her biri tek bir soruya odaklanır.

7.7.1 Profil oluşturucu

Tanım 7.11

Profil oluşturucu (*profiler*), bir programın zamanını ve belleğini nerede harcadığını ölçen araçtır. Hangi metodun kaç kez çağrıldığını, ne kadar sürdüğünü ve ne kadar bellek ayırdığını gösterir.

Profil oluşturucu, kimin kimi çağırdığını gösteren çağrı grafiğini (*call graph*) çeşitli biçimlerde sunar: düz liste, ağaç ya da alev grafiği (*flame graph*). Alev grafiğinde her metot bir kutudur; kutunun genişliği o metotta harcanan zamanla orantılıdır. Geniş kutular hemen göze çarpar. Ölçümün teknik ayrıntıları kullanılan derleyiciye ve JVM sürümüne göre değişir.

Profil oluşturmanın temel kuralı kısadır: önce ölç, sonra iyileştir. Bir programın zamanının çoğu genellikle küçük bir bölgede, bir *sıcak noktada* (*hot spot*) harcanır. Programcının tahmini bu noktayı çoğu zaman ıskalar. Tahminle yapılan optimizasyon kodu karmaşıklaştırır ve yanlış yeri hızlandırır.

Eski JVM'lerde yerleşik bir profil oluşturucu vardı ve `java -Xprof` seçeneğiyle açılırdı. Bu seçenek Java 10'da kaldırıldı. Java 21'de aynı seçenek yalnız bir uyarı üretir ve yok sayılır:

```
OpenJDK 64-Bit Server VM warning: Ignoring option -Xprof; support was removed in 10.0
```

Bugün Java'da başka araçlar kullanılır. JDK ile gelen **JDK Flight Recorder** (JFR) olayları kaydeder, **JDK Mission Control** bu kayıtları görselleştirir. Bunlara ücretsiz **VisualVM** ve IDE'lerin yerleşik profil oluşturucuları eklenir. JFR, programı çok az yavaşlatarak çalışma sırasında olayları kaydeder. Program JFR kaydıyla şöyle çalıştırılır ve kaydın özeti şöyle alınır:

```
java -XX:StartFlightRecording=filename=kayit.jfr Profil
jfr summary kayit.jfr
```

İlk komut programı çalıştırır ve olayları `kayit.jfr` dosyasına yazar. İkinci komut kayıttaki olay türlerini ve sayılarını listeler. Ayrıntılı inceleme genellikle bu dosya JDK Mission Control'de açılarak yapılır.

7.7.2 Sıcak noktayı ölçmek

Profil oluşturmanın değerini görmek için aynı işi iki farklı yolla yapan iki metot karşılaştırılır. İkisi de 0'dan n − 1'e kadar sayıları art arda yazarak bir metin oluşturur.

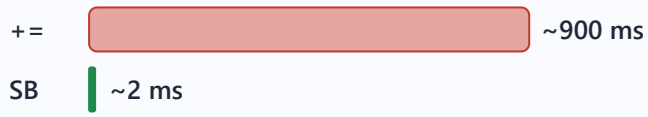
```
public class Profil {
  static String birlestirArti(int n) {
    String s = "";
    for (int i = 0; i < n; i++) {
      s += i;  // her turda yeni String
    }
    return s;
  }

  static String birlestirBuilder(int n) {
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < n; i++) {
      sb.append(i);
    }
    return sb.toString();
  }
  // main: yandaki kutuda
}
```

```
public static void main(String[] args) {
  long t0 = System.nanoTime();
  String a = birlestirArti(50_000);
  long t1 = System.nanoTime();
  String b = birlestirBuilder(50_000);
  long t2 = System.nanoTime();
  long ms1 = (t1 - t0) / 1_000_000;
  long ms2 = (t2 - t1) / 1_000_000;
  System.out.println("+= : " + ms1 + " ms");
  System.out.println("SB : " + ms2 + " ms");
  System.out.println(a.equals(b));
}
```

`main` metodu iki metodun süresini `System.nanoTime()` ile ölçer. Bu metot nanosaniye cinsinden bir zaman damgası döndürür; iki damganın farkı 1.000.000'a bölünerek milisaniyeye çevrilir. Son satır iki metnin aynı olduğunu doğrular: hız farkı sonucu değiştirmemelidir. Bir çalıştırmada çıktı şöyledir; sayılar makineye ve çalıştırmaya göre değişir:

```
+= : 873 ms
SB : 2 ms
true
```



Şekil 7.16: Çalışma süreleri: `+=` ile yaklaşık 900 milisaniye, `StringBuilder` ile yaklaşık 2 milisaniye.

Fark yüzlerce kattır (Şekil 7.16). Bu not hazırlanırken JDK 21 ile üç çalıştırmada `+=` sürümü 862–922 ms, `StringBuilder` sürümü 2–4 ms sürdü. Nedeni `String`'in değişmez (*immutable*) olmasıdır. `s += i` eski metni değiştiremez; her turda yeni bir `String` oluşturur. Eski metnin bütün karakterleri bu yeni nesneye kopyalanır. k. turda kopyalanan karakter sayısı aşağı yukarı k ile orantılıdır. Toplam iş $1 + 2 + \ldots + n \approx n^2/2$ ile büyür: $O(n^2)$. `StringBuilder` ise tek bir tampona ekler. Tampon dolunca daha büyüğüne taşınır, ama boyut her seferinde katlandığı için toplam kopyalama $O(n)$'de kalır.

Bu programda sıcak nokta açıktır. Gerçek bir programda ise ilk bakışta görünmez: `s += i` satırı yüzlerce satırın arasında masum bir döngünün içinde durur. Profil oluşturucu zamanın %99'unun bu satırda harcandığını gösterir ve iyileştirmenin nereye yapılacağını söyler.

7.7.3 Kod kapsamı

> **Tanım 7.12**
>
> **Kod kapsamı** (*code coverage*), testler çalışırken kodun hangi bölümlerinin çalıştırıldığını ölçer. Sonuç genellikle bir oran olarak verilir: çalışan satırların ya da dalların toplama oranı.

Kod kapsamı aracı, hiç çalıştırılmayan ya da kullanılmayan kod parçalarını ortaya çıkarır ve test edilmemiş bölgeleri bulmaya yardım eder. Bu yüzden yazılımın güvenilirliğini artırmada önemli bir araçtır. Java'da en yaygın araç Maven ve Gradle ile çalışan **JaCoCo**'dur. IntelliJ'deki "Run with Coverage" komutu ve Eclipse'teki **EclEmma** eklentisi de aynı işi yapar. Sonuç, kod satırları renklendirilerek gösterilir: çalışan satırlar yeşil, hiç çalışmayanlar kırmızı.

Ölçüt	Ne sayılır?
Satır (*line*)	En az bir kez çalışan satırlar
Dal (*branch*)	Her `if` / döngü koşulunun hem doğru hem yanlış sonucu
Metot	En az bir kez çağrılan metotlar

Tablo 7.17: Kod kapsamı ölçütleri.

Tablo 7.17'deki üç ölçütün en sıkısı dal kapsamıdır. Bir `if` satırı çalıştığında satır kapsamı onu "çalıştı" sayar. Dal kapsamı ise koşulun hem doğru hem yanlış çıktığını görmek ister. Koşul hep doğru çıkmışsa, yanlış dalda saklanan bir kusur hiç denenmemiştir.

Kapsamın sınırını iyi bilmek gerekir: %100 kapsam hatasız kod demek değildir. Kapsam yalnız satırın çalıştığını söyler, sonucun doğru olduğunu değil. Hiç doğrulama içermeyen, yalnız metodu çağıran bir test de kapsamı artırır. Kapsam raporu "burası hiç denenmedi" uyarısında çok değerlidir; "burası doğru" anlamında ise hiçbir şey söylemez.

7.7.4 Kod izleme: kod kapsamı

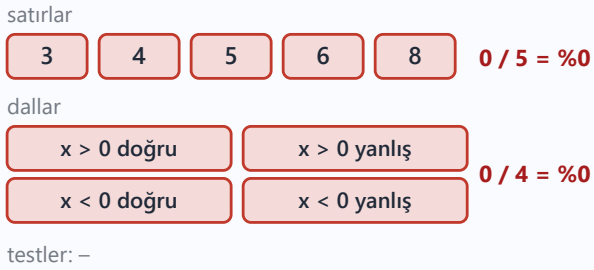
Örnek metot, bir sayının işaretini metin olarak döndürür.

```
public class Isaret {
  public static String isaret(int x) {
    if (x > 0) {
      return "pozitif";
    } else if (x < 0) {
      return "negatif";
    }
    return "sıfır";
  }
}
```

Metodun çalıştırılabilir satırları 3, 4, 5, 6 ve 8'dir. 1., 2. ve 7. satırlar bildirim ya da kapanış ayracıdır; kapsam araçları onları saymaz. İki koşul vardır: `x > 0` ve `x < 0`. Her koşulun iki dalı olduğu için toplam 4 dal sayılır. Metot için iki test yazılmıştır:

```
class IsaretTest {
  @Test
  void pozitif() {
    assertEquals("pozitif", Isaret.isaret(5));
  }

  @Test
  void negatif() {
    assertEquals("negatif", Isaret.isaret(-3));
  }
}
```



Şekil 7.17: Hiç test çalışmadı. Beş çalıştırılabilir satırın ve dört dalın hepsi kırmızı.

`isaret(5)` çağrısında 3. satır çalışır ve `x > 0` doğru çıkar; 4. satır `"pozitif"` döndürür. Test geçer. Kapsam: 5 satırdan 2'si, 4 daldan 1'i. `isaret(-3)` çağrısında 3. satırdaki koşul yanlış çıkar (yeni dal), 5. satırdaki `x < 0` doğru çıkar (yeni dal) ve 6. satır `"negatif"` döndürür. İki test de yeşildir.



Şekil 7.18: İki test sonrası rapor: satır kapsamı 4 / 5 = %80, dal kapsamı 3 / 4 = %75. 8. satır ve `x < 0`'ın yanlış dalı hiç denenmedi.

Test	Çalışan satırlar	Denenen dallar	Satır kapsamı	Dal kapsamı
(yok)	—	—	0 / 5	0 / 4

Test	Çalışan satırlar	Denenen dallar	Satır kapsamı	Dal kapsamı
`isaret(5)`	3, 4	`x > 0` doğru	2 / 5 = %40	1 / 4 = %25
+ `isaret(-3)`	3, 4, 5, 6	+ `x > 0` yanlış, `x < 0` doğru	4 / 5 = %80	3 / 4 = %75
+ `isaret(0)`	3, 4, 5, 6, 8	+ `x < 0` yanlış	5 / 5 = %100	4 / 4 = %100

Tablo 7.18: Her testten sonra satır ve dal kapsamı.

Tablo 7.18 eksik testi açıkça gösterir. İki yeşil teste rağmen 8. satır ve `x < 0` koşulunun yanlış dalı hiç denenmemiştir. Bu dala ancak iki koşulun da yanlış olduğu girdi, yani 0 ulaşır. Kapsam raporu böylece unutulan sınır değerini ortaya çıkarır: `isaret(0)`. Bu test eklendiğinde satır ve dal kapsamı %100 olur ve test metodun `"sıfır"` döndürdüğünü doğrular.

%100 kapsamda bile doğruluğu kapsam değil, `assertEquals` sınar. 8. satır yanlışlıkla `return "pozitif";` olsaydı, `isaret(0)`'ı çağırıp sonucuna bakmayan bir test kapsamı yine %100 yapardı.

7.7.5 Statik analiz: FindBugs ve SpotBugs

> **Tanım 7.13**
>
> **Statik analiz** (*static analysis*), kodu çalıştırmadan hata kalıplarını arayan incelemedir. "Statik" sözcüğü programın çalışmadığını anlatır; testler ve hata ayıklayıcı ise programı çalıştırır ve *dinamik* araçlardır.

FindBugs, Java kodunun statik analizi için geliştirilmiş bir araçtı. Programı çalıştırmadan derlenmiş kodu incelerdi ve bilinen hata kalıplarını arardı: `null` erişimi, yanlış karşılaştırma, kapatılmayan kaynak gibi. FindBugs artık geliştirilmiyor; devamı **SpotBugs**'tır. SpotBugs Maven, Gradle ve IDE eklentileriyle kullanılır. Benzer araçlar arasında PMD, Checkstyle ve SonarQube sayılabilir.

```
import java.util.Scanner;

public class SpotOrnek {
  public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    String cevap = sc.nextLine();
    // sonuç kullanılmıyor:
    cevap.trim();
    // String == ile karşılaştırılıyor:
    if (cevap == "evet") {
      System.out.println("Onaylandı");
    } else {
      System.out.println("Reddedildi");
    }
  }
}
```

Program kullanıcıdan bir cevap okur ve `"evet"` ise onay verir. Kullanıcı `evet` yazdığı hâlde çıktı `Reddedildi` olur. Programda iki kusur vardır ve derleyici ikisi için de sessiz kalır.

SpotBugs kalıbı	Uyarı
`ES_COMPARING_STRINGS_WITH_EQ`	`String` nesneleri `==` ile karşılaştırılıyor; doğrusu `equals`
`RV_RETURN_VALUE_IGNORED`	`trim()` yeni bir `String` döndürür; sonuç atılıyor

Tablo 7.19: SpotBugs'ın bu programda işaretlediği kalıplar.

Tablo 7.19'daki ilk kalıp, `==`'nin iki referansın aynı nesneyi gösterip göstermediğine baktığını hatırlatır. `nextLine()` klavyeden okunan metin için yeni bir `String` nesnesi oluşturur. Bu nesne, koddaki `"evet"` sabitiyle aynı nesne değildir; içerikleri aynı olsa bile `==` yanlış döner. Doğrusu `"evet".equals(cevap)`'tır. İkinci kalıp `String`'in değişmezliğiyle ilgilidir. `cevap.trim()`, `cevap`'ı değiştirmez; baştaki ve sondaki boşlukları silinmiş yeni bir metin döndürür. Sonuç bir değişkene atanmadığı için satır hiçbir iş yapmaz. Doğrusu `cevap = cevap.trim();`'dir.

Bu iki kusuru bir test de bulabilirdi, ama ancak doğru girdiyle yazılmışsa. Statik analiz ise girdiye ihtiyaç duymaz: kalıbı gördüğü her yerde uyarır. Bu yüzden testi tamamlar, onun yerini tutmaz. Statik analiz mantık hatalarını bilemez; "toplam yerine fark hesaplanıyor" gibi bir kusur hiçbir kalıba uymaz.

7.7.6 İlk statik analizci: derleyici

Her Java programcısının elinde zaten bir statik analizci vardır: `javac`. Derleyici kodu çalıştırmadan inceler ve derleme hatalarını bildirir. `-Xlint` seçeneğiyle derleme hatası olmayan, ama şüpheli görünen yapıları da bildirir.

```
public class Lint {
  public static void main(String[] args) {
    int gun = 2;
    switch (gun) {
      case 1:
        System.out.println("Pazartesi");
      case 2:
        System.out.println("Salı");
      case 3:
        System.out.println("Çarşamba");
        break;
    }
    int x = 10 / 0;
  }
}
```

Program derlenir ve çalışır, ama iki kusur içerir. `case 1` ile `case 2`'nin sonunda `break` unutulmuştur. 13. satırda da sabit bir sıfıra bölme vardır. Çalıştırıldığında çıktı şöyledir:

```
Salı
Çarşamba
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Lint.main(Lint.java:13)
```

`gun = 2` için `case 2`'den girilir. `break` olmadığı için akış `case 3`'e düşer (*fall-through*) ve `Çarşamba` da yazılır. Ardından 13. satırdaki bölme `ArithmeticException` fırlatır. Düz `javac Lint.java` bu programı hiç uyarı vermeden derler. `-Xlint:all` ile derlendiğinde ise iki kusur da çalıştırmadan görünür:

```
Lint.java:13: warning: [divzero] division by zero
    int x = 10 / 0;
                ^
Lint.java:7: warning: [fallthrough] possible fall-through into case
      case 2:
      ^
Lint.java:9: warning: [fallthrough] possible fall-through into case
      case 3:
      ^
3 warnings
```

`[divzero]` uyarısı sabit sıfıra bölmeyi, iki `[fallthrough]` uyarısı da bir önceki `case`'in sonunda `break` olmadığını bildirir. Uyarıların sırası derleyicinin iç aşamalarına göredir, satır sırasına göre değil. `case 1` için uyarı yoktur, çünkü ondan önce bir `case` yoktur. Uyarıları okumak, hatayı çalıştırmadan önce bulmanın en ucuz yoludur.

7.7.7 Özellik tabanlı test

Birim testlerinde girdileri programcı seçer. Ne kadar dikkatli seçerse seçsin, aklına gelmeyen bir durum olabilir. Özellik tabanlı test bu sorunu girdileri bilgisayara seçtirerek çözer.

> **Tanım 7.14**
>
> **Özellik tabanlı test** (*property-based testing*), rastgele üretilen çok sayıda girdiyle bir özelliğin her zaman sağlandığını sınayan testtir. Programcı tek tek örnek yerine bir **özellik** (*property*) yazar: “her girdi için doğru olmalı”. Özelliği bozan bir girdi bulunursa **karşı örnek** olarak raporlanır.

Yöntemin öncüsü **QuickCheck**'tir. QuickCheck testleri otomatik olarak üretir: girdileri programcı değil, araç rastgele seçer. Karşı örnek bulunduğunda araç onu mümkünse küçültür; böylece 50 öğelik bir dizi yerine 2 öğelik bir karşı örnek raporlanır. İlk QuickCheck Haskell dili için yazılmıştır. Birçok

dil için sürümleri vardır; Java'dakiler **jqwik** ve **junit-quickcheck**'tir. Aşağıdaki test aynı fikri düz JUnit ile uygular.

```
class OzellikTest {

  static String rastgeleMetin(Random r) {
    int uzunluk = r.nextInt(20);  // 0..19
    StringBuilder sb = new StringBuilder();
    for (int i = 0; i < uzunluk; i++) {
      sb.append((char) ('a' + r.nextInt(26)));
    }
    return sb.toString();
  }

  @Test
  void ikiKezTersCevirmekAyniMetniVerir() {
    Random r = new Random(42);    // sabit tohum
    for (int k = 0; k < 1000; k++) {
      String s = rastgeleMetin(r);
      String t = MetinIslem.ters(s);
      assertEquals(s, MetinIslem.ters(t));
    }
  }
}
```

Test edilen metot, bir metni ters çeviren basit bir yardımcıdır:

```
public class MetinIslem {
  public static String ters(String s) {
    return new StringBuilder(s).reverse().toString();
  }
}
```

Sınanan özellik şudur: bir metni iki kez ters çevirmek, metnin kendisini verir. `rastgeleMetin` 0–19 uzunlukta, küçük harflerden oluşan rastgele bir metin üretir. `r.nextInt(26)` 0–25 arasında bir sayı verir; `'a'`'ya eklenip `char`'a çevrilince `a`–`z` arasında bir harf olur. Test bu metinlerden 1000 tane üretir ve her biri için özelliği doğrular. Sabit tohum (42) testi belirleyici yapar: her çalıştırmada aynı 1000 metin denenir. Test çalıştırıldığında rapor tek satırdır: 1 test, 1000 girdi, başarılı.

Bu test elle seçilmiş örneklerden farklı bir şey dener. Boş metin (uzunluk 0), tek harfli metin, aynı harflerin tekrarı gibi durumlar rastgele üretimde kendiliğinden ortaya çıkar. Programcının bunları tek tek düşünmesi gerekmez.

7.7.8 Özellik testi ilk sürümdeki hatayı buluyor

Özellik testi, `enBuyuk` metodunun ilk, hatalı sürümüne de uygulanabilir. En büyük öğenin iki özelliği vardır: sonuç dizinin hiçbir öğesinden küçük değildir ve sonuç dizinin bir öğesidir.

```
class OzellikTest2 {

  @Test
  void enBuyukDizininEnBuyukElemanidir() {
    Random r = new Random(7);
    for (int k = 0; k < 1000; k++) {
      int[] dizi = new int[1 + r.nextInt(5)];
      for (int i = 0; i < dizi.length; i++) {
        dizi[i] = r.nextInt(201) - 100;     // -100..100
      }
      int enb = DiziIslem.enBuyuk(dizi);
      boolean dizideVar = false;
      for (int x : dizi) {
        assertTrue(enb >= x);               // özellik 1
        if (x == enb) dizideVar = true;
      }
      assertTrue(dizideVar,                   // özellik 2
        "Karşı örnek: " + Arrays.toString(dizi));
    }
  }
}
```

Test 7 tohumuyla 1000 rastgele dizi üretir. Her dizinin uzunluğu 1–5, öğeleri −100 ile 100 arasındadır: `r.nextInt(201)` 0–200 verir, 100 çıkarılınca aralık −100..100 olur. Her dizi için önce özellik 1 (`enb >= x`), sonra özellik 2 (`dizideVar`) denetlenir. İlk hatalı sürümle çalıştırıldığında sonuç şudur:

```
✗ enBuyukDizininEnBuyukElemanidir()
  Karşı örnek: [-60, -69, -39, -65, -13] ==> expected: <true> but was: <false>
1 test · 1 başarısız
```

Karşı örnek ikinci rastgele dizidir; ilk dizide negatif olmayan bir öğe vardır ve hatalı sürüm doğru sonucu verir. İkinci dizinin bütün öğeleri negatiftir; hatalı sürüm 0 döndürür. Özellik 1 hatalı sürümde de geçer: 0 bütün negatif sayılardan büyüktür. Özellik 2 ise hatayı yakalar: 0 dizide yoktur. `assertTrue` 'nun ikinci parametresi olan ileti, karşı örneği rapora yazar.

Düzeltilmiş sürümle aynı test çalıştırıldığında 1000 girdinin tamamı geçer. Buradan çıkan ders önemlidir: özellikler eksiksiz yazılmalıdır. Yalnız özellik 1 yazılsaydı test hatalı kodu da onaylardı. Zayıf bir özellik, yanlış bir güven duygusu verir.

Örnek 7.7

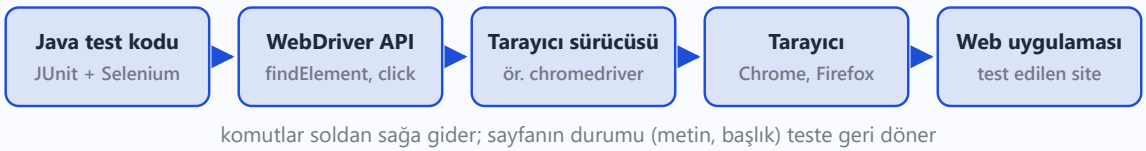
Bir sıralama metodu için "sonuç dizisinde her öğe bir sonrakinden büyük değildir" özelliği yazılsın. Bu özellik tek başına yeterli mi?

Yeterli değildir. Girdiyi hiç kullanmayıp hep boş dizi döndüren bir metot da bu özelliği sağlar. Hep sıfırlardan oluşan bir dizi döndüren metot da öyle. İkisi de sıralı bir dizi üretir, ama girdiyle ilgisi yoktur.

Eksik özellik şudur: sonuç, girdinin bir permütasyonu olmalıdır; aynı öğeler, aynı sayıda. İki özellik birlikte "sıralamak" kavramını tam olarak tanımlar.

7.7.9 Selenium

Selenium, web uygulamalarını test etmek için kullanılan bir çerçevedir. Tarayıcı tabanlı testleri kolaylaştırır: gerçek bir tarayıcıyı programla açar, bağlantıya tıklar, forma yazar, düğmeye basar ve sayfada beklenen metnin çıktığını doğrular. Java, Python, C# ve JavaScript gibi dillerle kullanılır. Java'da genellikle bir JUnit testinin içinden çağrılır.



Şekil 7.19: Selenium mimarisi. Java test kodu WebDriver arayüzünü çağırır; komutlar tarayıcı sürücüsü üzerinden gerçek tarayıcıya, oradan web uygulamasına ulaşır.

Şekil 7.19 komutların izlediği yolu gösterir. Test kodu WebDriver adlı ortak arayüzü kullanır. Her tarayıcının kendi sürücüsü vardır (Chrome için ChromeDriver, Firefox için GeckoDriver). Aynı test kodu sürücü değiştirilerek farklı tarayıcılarda çalıştırılabilir. Tipik bir senaryo şöyledir: giriş sayfası açılır, kullanıcı adı ve parola yazılır, "Giriş" düğmesine basılır ve başlıkta "Hoş geldiniz" metninin çıkıp çıkmadığına bakılır.

Selenium testleri kullanıcının gördüğü bütünü sınar; sistem ve kabul testleri düzeyindedir. Bu güçlerinin bedeli vardır. Birim testlerinden çok daha yavaştırlar: bir tarayıcı açmak bile saniyeler sürer. Ayrıca kırılgandırlar: bir düğmenin adının değişmesi bile testi bozabilir. Bu yüzden sayıları az tutulur; test piramidinin tepesinde dururlar.

7.7.10 Bellek sızıntısı

Java'da bellek, çöp toplayıcı sayesinde kendiliğinden yönetilir. Programcı `new` ile nesne oluşturur, ama onu hiçbir zaman açıkça silmez. Buna rağmen Java programları da bellek sızdırabilir.

Tanım 7.15

Çöp toplayıcı (*garbage collector*, GC), hiçbir referansın göstermediği nesnelerin belleğini geri alan JVM bileşenidir. **Bellek sızıntısı** (*memory leak*) ise artık gerekmeyen nesnelere hâlâ bir referans kaldığı için belleğin boşalmamasıdır.

Bu tanımın arkasındaki kilit kavram erişilebilirliktir. Çöp toplayıcı yalnız erişilemeyen nesneleri siler. Bir nesneye programın herhangi bir yerinden, örneğin bir `static` alandan bir referans zinciri ulaşıyorsa, o nesne program için gereksiz olsa bile silinmez.

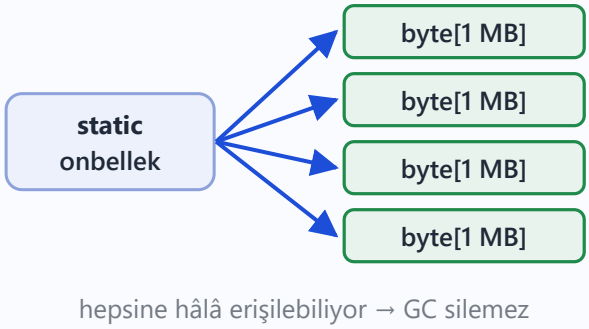
```
import java.util.ArrayList;
import java.util.List;

public class Sizinti {
  static List<byte[]> onbellek = new ArrayList<>();

  static void istekIsle(int no) {
    byte[] veri = new byte[1024 * 1024];  // 1 MB
    onbellek.add(veri);                   // hiç silinmiyor!
  }

  public static void main(String[] args) {
    for (int i = 1; ; i++) {
      istekIsle(i);
      if (i % 20 == 0) {
        System.out.println(i + " istek işlendi");
      }
    }
  }
}
```

Program sonsuz bir döngüde istekleri işler. Her istek 1 MB'lık bir dizi oluşturur ve onu `static` bir listeye ekler. Liste hiçbir zaman temizlenmez.



Şekil 7.20: `static` `onbellek` listesi her istekte eklenen 1 MB'lık dizileri tutar. Dizilerin hepsine hâlâ erişilebilir; çöp toplayıcı onları silemez.

Program `java -Xmx256m Sizinti` komutuyla çalıştırıldığında çıktı şöyledir; aradaki satırlar kısaltılmıştır:

```
20 istek işlendi
…
120 istek işlendi
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
    at Sizinti.istekIsle(Sizinti.java:8)
    at Sizinti.main(Sizinti.java:14)
```

`-Xmx256m` seçeneği heap'i en çok 256 MB ile sınırlar. Program 120 isteği işledikten bir süre sonra `OutOfMemoryError: Java heap space` ile çöker. Yığın izi sorunun yerini doğru gösterir: 8. satırdaki `new byte[1024 * 1024]`. Ama asıl kusur bu satırda değildir. Bellek o satırda biter, çünkü önceki dizilerin hiçbiri serbest bırakılmamıştır. Kusur 9. satırdaki `onbellek.add(veri)`'dır: eklenen veri bir daha hiç çıkarılmaz.

256 MB'lık bir heap'te yalnız 120 civarında 1 MB'lık dizinin sığması şaşırtıcı görünebilir. Nedeni JVM'in varsayılan çöp toplayıcısı G1'in belleği bölgelere ayırmasıdır. Bu heap boyutunda bölgeler 1 MB'tır. 1 MB'lık bir dizi, nesne başlığıyla birlikte bir bölgeye sığmaz ve iki bölge kaplar. Bu yüzden heap yarı dolulukta tükenir. Kesin sayı JVM'e ve ayarlara göre değişir.

7.7.11 Eclipse Memory Analyzer ve VisualVM

Sızıntıyı bulmanın yolu heap'e bakmaktır: hangi nesneler var, ne kadar yer tutuyorlar ve onları kim tutuyor? **Eclipse Memory Analyzer** (MAT) bu soruları cevaplar. Java heap'ini analiz ederek bellek sızıntılarını bulur ve bellek iyileştirmesine yardım eder.

MAT'ın girdisi bir **heap dökümüdür** (*heap dump*, `.hprof` dosyası): belirli bir andaki bütün nesnelerin fotoğrafı. Döküm iki yolla alınır:

```
// bellek dolunca otomatik
java -XX:+HeapDumpOnOutOfMemoryError Sizinti
// çalışan programdan
jps
jcmd <pid> GC.heap_dump yigin.hprof
```

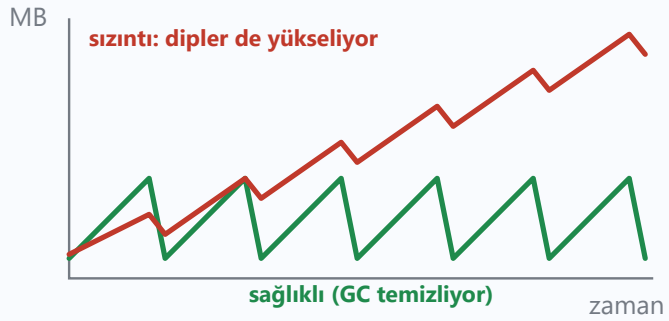
İlk yol `-XX:+HeapDumpOnOutOfMemoryError` seçeneğidir. Bu seçenekle başlatılan JVM, bellek tükendiği anda heap'i kendiliğinden bir dosyaya döker. İkinci yol çalışan bir programdan döküm almaktır. `jps` çalışan JVM'leri ve süreç numaralarını (pid) listeler. `jcmd <pid> GC.heap_dump yigin.hprof` o JVM'in heap'ini dosyaya yazar.

MAT'ın en önemli üç raporu şunlardır. **Leak Suspects** (sızıntı şüphelileri), heap'in büyük bölümünü tutan nesneleri otomatik olarak bulur. **Dominator Tree** (baskın nesne ağacı), hangi nesnenin en çok belleği tuttuğunu gösterir; bir nesne silinirse onunla birlikte silinecek bütün nesneler onun altında

listelenir. **Histogram** ise sınıf başına nesne sayısını ve kapladığı alanı verir.

`Sizinti` örneğinde Leak Suspects raporu, heap'in neredeyse tamamını `Sizinti.onbellek` listesinin ve içindeki `byte[]` dizilerinin tuttuğunu gösterir. Çözüm iki yoldan biridir: işi biten veri listeden çıkarılır ya da boyutu sınırlı bir önbellek kullanılır. Sınırlı önbellek dolunca en eski öğeyi atar ve bellek kullanımı sabit kalır.

VisualVM ise çalışan programı canlı izler. Bellek tüketimini, iş parçacıklarının kullanımını, işlemci yükünü ve daha fazlasını görselleştirir. Çalışan JVM'lere bağlanır, profil çıkarır, heap ve iş parçacığı dökümü alır. VisualVM ücretsiz ve açık kaynaktır (visualvm.github.io); eskiden JDK ile birlikte `jvisualvm` adıyla gelirdi. Komut satırında benzer bilgiyi veren araçlar `jps`, `jstat`, `jcmd` ve `jconsole`'dur.



Şekil 7.21: Kullanılan heap belleğin zamanla değişimi. Sağlıklı programda testere dişi: çöp toplama belleği her seferinde aynı dibe indirir. Sızıntıda ise dipler sürekli yükselir.

Şekil 7.21 VisualVM'deki heap grafiğinin iki tipik görünümünü gösterir. Sağlıklı bir programda bellek kullanımı yükselir, çöp toplama devreye girer ve kullanım hep aynı düzeye iner; grafik testere dişine benzer. Sızıntıda da testere dişi görülür, ama her çöp toplamadan sonraki dip bir öncekinden yüksektir. Dipler sürekli yükseliyorsa nesneler serbest bırakılmıyordur. Bu görüntü bir sızıntının ilk işaretidir. Sonra heap dökümü alınır ve MAT ile kimin neyi tuttuğu incelenir.

7.7.12 Hangi soruya hangi araç?

Bu notta anlatılan araçların her biri tek bir soruyu cevaplar. Doğru aracı seçmek, sorunun ne olduğunu doğru koymakla başlar. Tablo 7.20 soruları ve araçları eşleştirir.

Soru	Teknik / araç
Bu metot doğru sonuç veriyor mu?	Birim testi: JUnit, `@Test`, `assertEquals`, `assertThrows`
Parçalar birlikte, bütün sistem ve kullanıcı gözünden çalışıyor mu?	Modül, alt sistem, sistem, kabul (alfa / beta) testi; web için Selenium
Değişiklik eski bir şeyi bozdu mu?	Regresyon testi: tüm otomatik testler her değişiklikte (CI) çalıştırılır
Hangi girdiler seçilmeli?	Sınır değerleri, denklik sınıfları, `@ParameterizedTest`
Seçilmeyen girdilerde de doğru mu?	Özellik tabanlı test (QuickCheck, jqwik)

Soru	Teknik / araç
Testler kodun ne kadarını çalıştırıyor?	Kod kapsamı: JaCoCo, EclEmma
Kodda çalıştırmadan görülebilen hata var mı?	Statik analiz: `javac -Xlint`, SpotBugs (FindBugs)
Neden yanlış çalışıyor?	Hata ayıklayıcı: kesme noktası, Step Into / Over / Out, izleme; `jdb`
Zaman nerede harcanıyor?	Profil oluşturucu: JFR, VisualVM
Bellek neden doluyor?	VisualVM ile izleme, heap dökümünü Eclipse Memory Analyzer ile inceleme

Tablo 7.20: Hangi soruya hangi teknik ya da araç?

Tablodaki araçlar birbirinin rakibi değil, bir zincirin halkalarıdır. Regresyon testi bir arızayı haber verir. Hata ayıklayıcı kusuru bulur. Düzeltmeden sonra birim testi kusurun geri gelmesini önler. Kapsam raporu hangi testlerin eksik olduğunu, statik analiz testlerin kaçırabileceği kalıpları gösterir. Profil oluşturucu ve bellek araçları ise doğru çalışan bir programın neden yavaş ya da obur olduğunu açıklar. Test edilebilir kod genellikle iyi tasarlanmış koddur. Bağımlılıkları dışarıdan almak, zar örneğindeki `Random` gibi, hem testi hem de tasarımı kolaylaştırır.

7.8 Çözümlü problemler

Bu kesimdeki problemler notta anlatılan konuları birleştirir. Her problemin çözümü ara adımlarıyla verilir; JUnit sonuçları ve iletiler JUnit 5.11 ile çalıştırılarak doğrulanmıştır.

Örnek 7.8

Hangi test geçer, hangisi kalır?

```
class SoruTest {
  @Test void a() { assertEquals(7, 3 + 4); }
  @Test void b() { assertEquals(0.3, 0.1 + 0.2); }
  @Test void c() { assertTrue("Java".length() == 4); }
  @Test void d() { assertThrows(ArithmeticException.class,
      () -> System.out.println(10 / 2)); }
}
```

(a) 3 + 4 = 7; beklenen 7. **Geçer.**

(b) 0,1 + 0,2 ikili gösterimde 0,30000000000000004 olur. Tolerans verilmemiştir, karşılaştırma birebirdir. **Kalır:** `expected: <0.3> but was: <0.30000000000000004>`.

(c) `"Java".length()` 4'tür, koşul doğrudur. **Geçer.**

(d) `10 / 2 = 5;` bölme sorunsuzdur ve ekrana 5 yazılır. `ArithmeticException` fırlatılmaz. **Kalır:** `Expected java.lang.ArithmeticException to be thrown, but nothing was thrown.`

Örnek 7.9

Mutlak değer metodunda dal kapsamı. Aşağıdaki metot için dal kapsamının %100 olması için en az kaç test gerekir?

```
public class Mutlak {
  static int mutlak(int x) {
    if (x < 0) {
      return -x;
    }
    return x;
  }
}
```

Metotta tek bir koşul (`x < 0`) ve dolayısıyla iki dal vardır. Doğru dal için negatif bir girdi (ör. −3 → 3), yanlış dal için sıfır ya da pozitif bir girdi (ör. 5 → 5) gerekir. En az **2 test** yeterlidir. Sınır olarak −1 ve 0'ın da eklenmesi iyi olur; bu iki değer eşiği iki yandan kuşatır. Koşul yanlışlıkla `x < -1` yazılsaydı −1 için −1 dönerdi; −3 ve 5 testleri bunu göremez, yalnız −1 testi yakalar.

Örnek 7.10

Bu test sınıfının sorunu nedir?

```
class SayacTest {
  static int sayac = 0;

  @Test void artir() {
    sayac++;
    assertEquals(1, sayac);
  }

  @Test void ikiKezArtir() {
    sayac += 2;
    assertEquals(2, sayac);
  }
}
```

`sayac` alanı `static`'tir; sınıfa aittir ve iki testin test nesneleri aynı alanı paylaşır. Sonuç test sırasına bağlıdır. JUnit bu sınıfta önce `artir`'ı çalıştırır: `sayac` 1 olur ve test geçer. Ardından `ikiKezArtir` çalışır: `sayac` 1'den 3'e çıkar. **Kalır:** `expected: <2> but was: <3>`. Sıra ters olsaydı bu kez `artir` kalırdı (`expected: <1> but was: <3>`).

Çözüm iki yoldan biridir. `static` sözcüğü kaldırılır; JUnit her test için yeni nesne oluşturduğu için her test 0'dan başlar. Ya da alan `@BeforeEach` metodunda sıfırlanır.

Örnek 7.11

İki kez Step Over. `AdimOrnek` programında hata ayıklayıcı `int s = kareToplam(3, 4);` satırında (13) durmuşken art arda iki kez Step Over yapılırsa hangi satıra gelinir, ekranda ne vardır ve `s` kaçtır?

Birinci Step Over, `kareToplam`'ı içine girmeden çalıştırır ve atamayı tamamlar: `s = 25`. Program 14. satırda, `println`'dadır; ekran henüz boştur.

İkinci Step Over `println`'ı çalıştırır. Ekranda `s = 25` yazar ve program `main`'in kapanış satırına (15) gelir. `s` 25'tir.

Örnek 7.12

Hangi araç? Her durum için hangi teknik ya da araç kullanılmalıdır?

(a) Program 10 dakika sonra `OutOfMemoryError` ile çöküyor. VisualVM ile heap grafiği izlenir; dipler yükseliyorsa sızıntı vardır. Heap dökümü (`-XX:+HeapDumpOnOutOfMemoryError`) Eclipse Memory Analyzer ile incelenir.

(b) Rapor oluşturma 30 saniye sürüyor; hangi metot yavaş? Profil oluşturucu: JDK Flight Recorder ya da VisualVM.

(c) Yeni eklenen indirim kodu eski fatura hesabını bozdu mu? Regresyon testi: tüm otomatik testler yeniden çalıştırılır, tercihen CI ile her değişiklikte.

(d) Web sitesindeki kayıt formu tüm tarayıcılarda çalışıyor mu? Selenium ile tarayıcı tabanlı testler, farklı tarayıcı sürücüleriyle.

(e) `switch`'te unutulmuş `break`'ler var mı? Statik analiz: `javac -Xlint:all` (`[fallthrough]` uyarısı) ya da SpotBugs.

(f) Bir döngü 10.000. turda yanlış değer üretiyor; o anda değişkenler ne? Hata ayıklayıcıda koşullu kesme noktası: `i == 9999`. Döngü sayacı 0'dan başladığı için 10.000. tur `i = 9999`'dur.

Örnek 7.13

Asal sayı sınamasının sınırları. `static boolean asalMi(int n)` için `@ParameterizedTest` ile sınır değerleri 0, 1, 2, 3, 4, 97 ve 100 nasıl test edilir?

```
static boolean asalMi(int n) {
  if (n < 2) return false;
  for (int d = 2; d * d <= n; d++) {
    if (n % d == 0) return false;
  }
  return true;
}

@ParameterizedTest
@CsvSource({"0, false", "1, false", "2, true", "3, true",
            "4, false", "97, true", "100, false"})
void asalSinirlari(int n, boolean beklenen) {
  assertEquals(beklenen, asalMi(n));
}
```

Yedi çağrının yedisi de geçer. 0 ve 1 asal değildir; metodun ilk satırı bu iki durumu ayırır. 2 en küçük asaldır ve döngüye hiç girmez, çünkü $2 \times 2 > 2$'dir. 4 en küçük bileşik sayıdır; döngü `d = 2` 'de bölen bulur. 0, 1 ve 2 en çok hata yapılan sınırlardır: `n < 2` yerine `n < 1` yazılırsa 1 asal sanılır. Döngü koşulundaki `d * d <= n` de bir sınırdır: `<` yazılsaydı 4 asal sanılırdı, çünkü `d = 2` için $4 < 4$ yanlış çıkar ve döngü hiç çalışmazdı.

Örnek 7.14

Dikdörtgen sınıfının testi. `Dikdortgen` sınıfı (`en` , `boy` , `alan()` , `cevre()`) için bir JUnit 5 test sınıfı yazılsın; negatif kenar `IllegalArgumentException` fırlatmalıdır.

```
class DikdortgenTest {
  @Test
  void alanDogru() {
    assertEquals(12.0, new Dikdortgen(3, 4).alan(), 1e-9);
  }

  @Test
  void cevreDogru() {
    assertEquals(14.0, new Dikdortgen(3, 4).cevre(), 1e-9);
  }

  @Test
  void negatifKenarReddedilir() {
    assertThrows(IllegalArgumentException.class,
      () -> new Dikdortgen(-1, 2));
  }
}
```

3 × 4'lük dikdörtgenin alanı 12, çevresi 2 × (3 + 4) = 14'tür. Kenarlar `double` olduğu için doğrulamalarda tolerans verilir. Sınır olarak sıfır kenarlı bir dikdörtgen de denenmelidir: alanı 0'dır ve istisna fırlatmamalıdır.

Örnek 7.15

Havale metodu için TDD. `BankaHesabi`'na `havale(BankaHesabi hedef, double miktar)` eklenecektir. Kod yazılmadan önce hangi testler yazılır?

(1) Başarılı havale: iki bakiye de değişir; gönderenin bakiyesi azalır, alıcınınki artar. (2) Yetersiz bakiye: istisna fırlatılır ve *iki bakiye de değişmez*. (3) Negatif miktar: reddedilir. (4) Kendine havale: kararlaştırılan davranış (ret ya da etkisiz işlem) sınanır.

İkinci test özellikle önemlidir. Havale önce gönderenden çekip sonra alıcıya yatırıyorsa ve yatırma adımı istisna fırlatıyorsa, para havada kaybolabilir. Her test önce kırmızıdır, sonra yazılan kodla yeşile döner.

Örnek 7.16

Ortalamada tam sayı bölmesi. Bir dizinin ortalamasını hesaplayan metot `{1, 2}` için 1,5 yerine 1,0 döndürüyor. Kusur hata ayıklayıcıyla nasıl bulunur?

Önce kalan bir test yazılır: `{1, 2}` için beklenen 1,5. Kesme noktası `return` satırına konur. `toplam` değişkeni ve bölme ifadesi izlenir: `toplam` 3'tür, doğrudur. `toplam / dizi.length` ise 1 verir: iki `int` bölündüğü için bu bir tam sayı bölmesidir (3 / 2 = 1). Sonuç `double`'a

ancak bölmeden sonra çevrilir ve 1,0 olur. Düzeltme `(double) toplam / dizi.length`'tir: dönüşüm bölmeden önce yapılır, bölme ondalıklı olur ve 1,5 verir. Ardından bütün testler çalıştırılır.

7.9 Sık yapılan hatalar

Hata	Sonuç	Doğrusu
Beklenen ile gerçeği ters yazmak: `assertEquals(hm.topla(2, 3), 5)`	Çalışır, ama kalınca ileti ters okunur: `expected` gerçek sonucu gösterir	Önce beklenen, sonra gerçek
Ondalıkta toleransı unutmak: `assertEquals(0.3, 0.1 + 0.2)`	`expected: <0.3> but was: <0.30000000000000004>`	`assertEquals(0.3, 0.1 + 0.2, 1e-9)`
`import static` eksik	Derleme hatası: `cannot find symbol`, `method assertEquals(int,int)`	`import static org.junit.jupiter.api.Assertions.*;`
JUnit 4 ile 5'i karıştırmak	`org.junit.Test` ile `org.junit.jupiter.api.Test` farklıdır; yanlış içe aktarma testlerin hiç çalışmamasına yol açabilir	JUnit 5 projesinde yalnız `org.junit.jupiter.api`
`@BeforeAll` metodu `static` değil	JUnit sınıftaki testleri çalıştırmaz: `@BeforeAll method '…' must be static …`	`static void basla()`
Testler arası bağımlılık	Sıra değişince testler rastgele kalır	Her test kendi verisini kurar
Doğrulaması olmayan test	Test her zaman ✓ olur, hiçbir şey kanıtlamaz; kapsamı yine de artırır	Her testte en az bir doğrulama
Yalnız “mutlu yol”u test etmek	Sınırlar (0, boş dizi, negatif), geçersiz girdiler ve istisnalar unutulur; hatalar tam oradadır	Normal durum + sınır + hatalı girdi
Dizileri `assertEquals` ile karşılaştırmak	İçerik aynı olsa da kalır; referanslar karşılaştırılır	`assertArrayEquals`
Tohumsuz `Random` ya da `LocalDate.now()`	Kararsız test: bazen geçer, bazen kalır	Bağımlılığı dışarıdan vermek, sabit tohum

Tablo 7.21: Test yazarken sık yapılan hatalar.

Tablo 7.21'deki hataların çoğu derleyiciyi geçer. Ters yazılmış bir `assertEquals`, doğrulaması olmayan bir test ya da birbirine bağlı iki test hiçbir uyarı üretmez. Bu yüzden test kodu da üretim kodu kadar dikkatle okunmalıdır. Testin kendisinin doğru olduğunu görmenin basit bir yolu, onu bir kez bilerek kırmızıya düşürmektir: beklenen değer kasıtlı olarak yanlış yazılır ve testin gerçekten kaldığı görülür. Hiç kalmayan bir test bir şey sınamıyor olabilir.

7.10 Özet

- Test, programın beklenen davranışı gösterip göstermediğini sistemli biçimde sınar. Test hataların varlığını gösterebilir, yokluğunu gösteremez; iyi seçilmiş testler güveni artırır.
- Hata programcının yanılgısı, kusur bu yanılgının koddaki karşılığı, arıza ise çalışırken görülen yanlış davranıştır. Test arızayı ortaya çıkarır, hata ayıklama kusuru bulur.
- Test düzeyleri birim, modül, alt sistem, sistem ve kabul testidir. Aşağıda yakalanan hata ucuz, yukarıda yakalanan pahalıdır. Kara kutu testi gereksinimden, beyaz kutu testi koddan yola çıkar.
- JUnit 5'te test metotları `@Test` ile işaretlenir; sonuç `assertEquals(beklenen, gerçek)` gibi doğrulamalarla denetlenir. Ondalık sayılarda tolerans, dizilerde `assertArrayEquals` kullanılır.
- İyi bir test hazırla, çalıştır, doğrula (AAA) düzenindedir ve tek bir davranışı sınar. Test adı davranışı anlatır.
- JUnit her test için yeni bir test nesnesi oluşturur. `@BeforeEach` her testten önce, `@AfterEach` her testten sonra (test kalsa bile) çalışır. `@BeforeAll` ve `@AfterAll` bir kez çalışır ve `static` olmalıdır.
- `assertThrows` kodun beklenen tipte (ya da alt tipinde) istisna fırlattığını doğrular ve istisna nesnesini döndürür.
- Hatalar en çok sınırlarda olur. Sınır değer analizi her aralığın iki ucunu ve geçersiz bölgenin ilk değerlerini seçer; denklik sınıfından tek temsilci yeterlidir. `@ParameterizedTest` ve `@CsvSource` aynı testi birçok veriyle çalıştırır.
- İyi birim testi hızlı, bağımsız, tekrarlanabilir, kendini doğrulayan ve zamanında yazılmış testtir (FIRST). Dış bağımlılıkların yerine test dublörü konur.
- Regresyon testi, her değişiklikten sonra bütün testlerin yeniden çalıştırılmasıdır. Testler belirleyici olmalıdır; rastgelelik sabit tohumla, bağımlılık enjeksiyonu yoluyla kontrol edilir. CI sistemleri testleri her gönderimde çalıştırır.
- TDD'de önce kalan bir test yazılır (kırmızı), sonra onu geçiren en basit kod (yeşil), sonra kod temizlenir (yeniden düzenleme).
- Hata ayıklayıcı programı kesme noktasında, satır çalışmadan önce durdurur. Step Into metodun içine girer, Step Over çağrıyı tek adımda geçer, Step Out bulunulan metodu bitirir. Koşullu ve istisna kesme noktaları aranan anı yakalar. `jdb` aynı işi komut satırından yapar.
- Sistematik hata ayıklama: yeniden üret, kalan test yaz, daralt, hipotez kur ve sına, düzelt ve bütün testleri çalıştır. `assert` varsayımları `-ea` ile açıkken denetlenir.
- Profil oluşturucu (JFR, VisualVM) zamanın harcandığı sıcak noktayı ölçer: önce ölç, sonra iyileştir. Kod kapsamı (JaCoCo) hiç çalışmayan satırları ve dalları gösterir; %100 kapsam hatasızlık demek değildir.
- Statik analiz (`javac -Xlint` , SpotBugs) kodu çalıştırmadan şüpheli kalıpları bulur. Özellik tabanlı test rastgele girdilerle bir özelliği sınar; özellik zayıfsa hatalı kodu da onaylar.

- Selenium web uygulamasını gerçek tarayıcıda sınar. Bellek sızıntısı, gereksiz nesnelere hâlâ referans kalmasıdır; VisualVM'de yükselen diplerle görülür, heap dökümü Eclipse Memory Analyzer ile incelenir.

7.11 Kavram listesi

test *testing* hata / kusur / arıza *error / defect / failure* hata ayıklama *debugging* hata ayıklayıcı *debugger* birim testi *unit test* modül, alt sistem, sistem, kabul testi *module, sub-system, system, acceptance testing* kara kutu / beyaz kutu testi *black-box / white-box testing* çerçeve *framework* etiket *annotation* doğrulama *assertion* lambda ifadesi *lambda expression* test yaşam döngüsü *test lifecycle* hazırla, çalıştır, doğrula *arrange, act, assert* sınır değer analizi *boundary value analysis* denklik sınıfı *equivalence class* parametreli test *parameterized test* tümleştirme testi *integration test* test dublörü *test double* regresyon testi *regression test* kararsız test *flaky test* tohum *seed* bağımlılık enjeksiyonu *dependency injection* sürekli entegrasyon *continuous integration* test güdümlü geliştirme *test-driven development* kesme noktası *breakpoint* çağrı yığını *call stack* çerçeve *stack frame* profil oluşturucu *profiler* sıcak nokta *hot spot* kod kapsamı *code coverage* statik analiz *static analysis* özellik tabanlı test *property-based testing* karşı örnek *counterexample* çöp toplayıcı *garbage collector* bellek sızıntısı *memory leak* heap *heap (memory)* heap dökümü *heap dump*

7.12 Kendini Dene

Sorularda notta kullanılan sınıflar (`BankaHesabi`, `AdimOrnek`, `Jdb`, `Zar`) aynen geçerlidir.

1. `assertEquals(0.1 * 3, 0.3)` geçer mi? Kalıyorsa ileti nedir ve iletide ne yanıltıcıdır?

   Cevap

   Kalır: `expected: <0.30000000000000004> but was: <0.3>`. İki sorun vardır. Tolerans yoktur: 0,1 × 3 ikili gösterimde tam 0,3 etmez. Parametreler de ters yazılmıştır: hesaplanan değer "beklenen" yerine geçmiş, sabit 0,3 "gerçek" gibi görünmüştür. Doğrusu `assertEquals(0.3, 0.1 * 3, 1e-9)` 'dur.

2. Bir test sınıfında üç sıradan `@Test` metodu ve `@CsvSource`'unda üç satır bulunan bir

`@ParameterizedTest` var. `@BeforeEach` metodu kaç kez çalışır?

Cevap

6 kez. Parametreli testin her satırı ayrı bir test çağrısıdır ve her çağrıdan önce `@BeforeEach` çalışır: 3 + 3 = 6.

3. Bakiyesi 100 olan bir hesap için `assertThrows(IllegalStateException.class, () -> hesap.yatir(-10))` geçer mi?

Cevap

Kalır. `yatir(-10)` istisna fırlatır, ama tipi `IllegalArgumentException`'dır; bu sınıf `IllegalStateException`'ın alt sınıfı değildir. İleti: `Unexpected exception type thrown, expected: <java.lang.IllegalStateException> but was: <java.lang.IllegalArgumentException>`.

4. `Jdb` programında `stop at Jdb:3` ile 3. satıra (`int sonuc = x * x;`) kesme noktası konuyor. Program kaç kez durur? İlk durmada `locals` ne gösterir?

Cevap

Üç kez durur: `kare` döngüde üç kez çağrılır ve `x` sırasıyla 1, 2, 3 olur. İlk durmada `locals` yalnız `x = 1` metot argümanını gösterir; yerel değişkenler listesi boştur. 3. satır henüz çalışmadığı için `sonuc` o anda yoktur.

5. `static String tur(int n)` metodu üç çalıştırılabilir satırdan oluşuyor: `if (n % 2 == 0)`, `return "çift";` ve `return "tek";`. Yalnız `tur(4)` testi varsa satır ve dal kapsamı nedir?

Cevap

4 çifttir; `if` satırı ve `return "çift";` çalışır. Satır kapsamı 2 / 3 ≈ %67'dir. Tek koşulun iki dalından yalnız doğru dalı denenmiştir: dal kapsamı 1 / 2 = %50. Tek sayı olan bir girdiyle, örneğin `tur(3)` ile yapılan ikinci test ikisini de %100 yapar.

6. `AdimOrnek` programında hata ayıklayıcı `kare`'nin içinde, `x = 3` iken duruyor. Önce Step Out, sonra Step Over yapılıyor. Program hangi satırdadır, `k1` kaçtır?

Cevap

Step Out `kare(3)`'ü bitirir ve 7. satıra, `int k1 = kare(a);`'ya döner; dönüş değeri 9'dur ama atama henüz tamamlanmamıştır. Step Over atamayı tamamlar ve 8. satıra geçer. `k1 = 9`.

7. Bir mağazada 6 ve daha fazla ürün alana %10 indirim uygulanıyor; ürün sayısı negatif olamaz. Sınır değer analiziyle hangi girdiler seçilir?

   Cevap

   Denklik sınıfları: geçersiz (< 0), indirimsiz (0–5), indirimli (≥ 6). Sınırlar: −1 (geçersiz bölgenin ilk değeri), 0 (geçerli bölgenin başı), 5 ve 6 (indirim eşiğini iki yandan kuşatan çift). Eşik `adet >= 6` yerine yanlışlıkla `adet > 6` yazılmışsa yalnız 6 bu kusuru yakalar.

8. `MetinIslem.ters` yanlışlıkla `return s;` olarak yazılmış olsun: metni hiç değiştirmesin. "İki kez ters çevirmek metni değiştirmez" özellik testi bu kusuru yakalar mı? Nasıl güçlendirilir?

   Cevap

   Yakalamaz: `ters(ters(s))` yine `s`'dir ve 1000 girdinin hepsi geçer. Özellik doğrudur, ama tek başına zayıftır. Güçlendirmek için ikinci bir özellik eklenir: sonucun ilk karakteri girdinin son karakteridir. Ya da birkaç bilinen örnek doğrulanır: `assertEquals("cba", MetinIslem.ters("abc"))`.

7.13 İleri okuma

- JUnit Team, *JUnit 5 User Guide* (junit.org/junit5/docs/current/user-guide), "Writing Tests", "Assertions" ve "Parameterized Tests" kesimleri: etiketlerin, doğrulamaların ve parametre kaynaklarının resmî ve güncel anlatımıdır.
- K. Beck, *Test-Driven Development: By Example*: kırmızı-yeşil-yeniden düzenleme döngüsünü bir para birimi örneği üzerinde adım adım uygular; yöntemi kuran kişinin kitabıdır.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals* (12. baskı), Bölüm 7 "Exceptions, Assertions, and Logging": `assert` deyimini, günlük kaydını ve hata ayıklama ipuçlarını Java'ya özgü ayrıntılarla anlatır.
- Oracle, *Programming With Assertions*: `assert`'in ne zaman kullanılacağını ve ne zaman kullanılmayacağını açıklayan kısa resmî belgedir.

7.14 Kaynaklar

1. JUnit Team, *JUnit 5 User Guide*, junit.org/junit5/docs/current/user-guide (Writing Tests, Assertions, Parameterized Tests).
2. Oracle, *The Java Tutorials* ve *JDK 21 Tool Specifications*: `javac` (`-Xlint`), `jdb`, `jcmd`, `jfr`.
3. Oracle, *Programming With Assertions* (Java SE belgeleri).
4. J. Gosling ve diğerleri, *The Java Language Specification, Java SE 21 Edition* (14.10 The assert Statement).
5. P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson.

6. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson (Bölüm 7: Exceptions, Assertions, and Logging).
7. K. Beck, *Test-Driven Development: By Example*, Addison-Wesley.
8. Araç belgeleri: SpotBugs (spotbugs.github.io), JaCoCo (jacoco.org), Selenium (selenium.dev), Eclipse Memory Analyzer (eclipse.dev/mat), VisualVM (visualvm.github.io), jqwik (jqwik.net).

KISIM IV

Java ile Uygulamalar

Bölüm 8–11

KISIM IV · BÖLÜM 8

Giriş Çıkış İşlemleri

Bir programın değişkenleri bellekte yaşar ve program kapanınca kaybolur. Kalıcı olan her şey, yani ölçüm kayıtları, ayar dosyaları, resimler, öğrenci listeleri, programın dışında bir yerde durur: diskte, ağda ya da bir cihazda. Programın bu dış dünyayla veri alışverişine giriş çıkış (G/Ç) denir. Java bu alışverişi tek bir fikir üzerine kurar: veri, bir kaynaktan bir hedefe sırayla akan bir akıştır. Not önce akış kavramını ve klavye ile ekranın standart akışlarını anlatır. Ardından dosyaların içindeki baytlara iner: metin ile ikili dosyanın farkı, karakter kodlamaları ve bir sayının dosyada kaç bayt tuttuğu. Sonraki kesimler `File` sınıfını, metin okuma ve yazma sınıflarını, tamponlamayı, öteki bayt akışlarını ve dosya kopyalamanın yollarını işler. Son kesimler rastgele erişimi, nesnelerin serileştirilmesini ve Java'nın tampon ile kanal üzerine kurulu NIO kütüphanesini anlatır.

Bu bölümün kodu: `github.com/sercankulcu/object-oriented-programming-java/Bolum08/src`

ÖĞRENME ÇIKTILARI

Akışı açıklamak
Girdi ve çıktı akışını, `System.in`, `System.out` ve `System.err` akışlarını anlatmak; klavyeden `Scanner` ile okumak.

Metin ile ikiliyi ayırmak
Metin ve ikili dosya farkını, ASCII, ISO-8859-9 ve UTF-8 kodlamalarını açıklamak; bir sayının dosyada kaç bayt tuttuğunu hesaplamak.

Dosya okumak ve yazmak
`File`, `Scanner`, `BufferedReader`, `PrintWriter` ve `FileWriter` ile metin dosyası işlemek; kaynakları try-with-resources ile kapatmak.

Doğru akışı seçmek
Bayt (`InputStream` / `OutputStream`) ve karakter (`Reader` / `Writer`) akışlarını, tamponlu ve sarmalayıcı akışları ayırt etmek.

Kopyalamak, konumlanmak
Bayt, blok ve satır düzeyinde kopyalamak; `RandomAccessFile` ile istenen konuma atlamak.

Nesne saklamak ve NIO
Nesneleri serileştirmek (`Serializable` , `transient`); NIO tamponunun, kanalının ve seçicisinin ne işe yaradığını açıklamak.

8.1 Akışlar

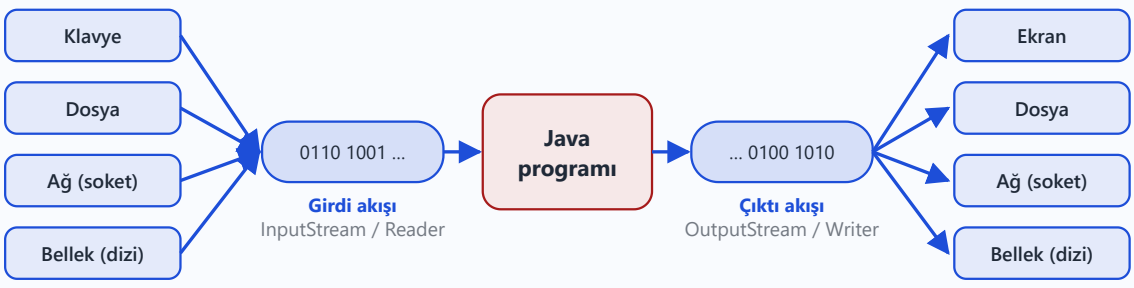
Bir program veriyi dışarıdan alır, işler ve sonucu dışarıya verir. Dışarısı her seferinde başka bir yer olabilir: klavye, ekran, diskteki bir dosya, ağdaki bir sunucu ya da bellekteki bir dizi. Bu kaynakların her biri için ayrı bir okuma yöntemi öğrenmek hem zor hem de gereksiz olurdu. Java hepsini aynı soyutlamayla ele alır: akış.

8.1.1 Akış nedir?

Tanım 8.1

Akış (*stream*), veriyi bir kaynaktan bir hedefe sırayla, parça parça taşıyan yapıdır. Verinin programa girdiği akışa **girdi akışı** (*input stream*), programdan çıktığı akışa **çıktı akışı** (*output stream*) denir. Taşınan parçalar karakter, sayı ya da ham bayt olabilir.

Akışı bir su borusuna benzetmek yerinde olur. Borunun bir ucundan giren su öbür ucundan aynı sırayla çıkar. Borunun öbür ucunda musluk mu, depo mu olduğu suyu içen kişiyi ilgilendirmez. Akış da böyledir. Program yalnız akışın kendi ucunu görür. Akışın öbür ucunda klavye, dosya ya da ağ bağlantısı olabilir; program her durumda aynı `read()` ve `write()` metotlarını çağırır. Java G/Ç (giriş/çıkış, *I/O*) kütüphanesinin gücü bu ortak arayüzden gelir. Şekil 8.1 bu yapıyı gösterir.



Şekil 8.1: Akış. Veri soldaki kaynaklardan girdi akışıyla programa, programdan çıktı akışıyla sağdaki hedeflere sırayla taşınır. Bayt akışlarının üst sınıfları `InputStream` ve `OutputStream` , karakter akışlarınınki `Reader` ve `Writer` 'dır.

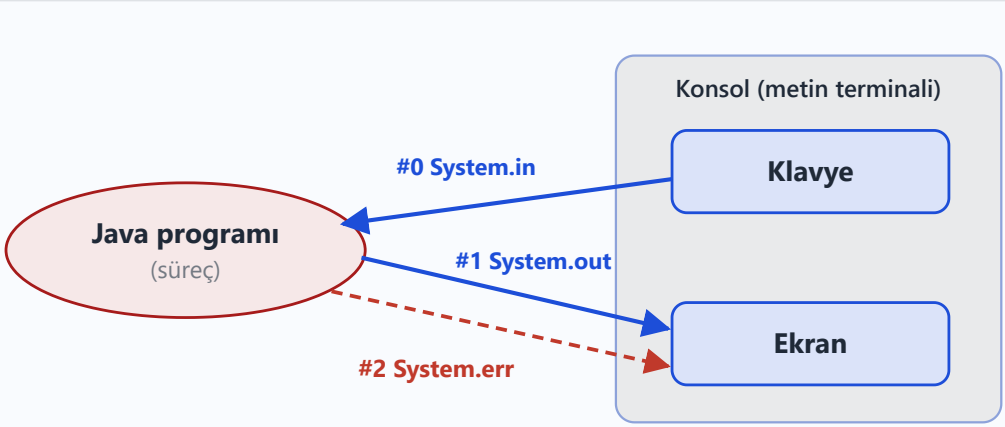
Akışın iki özelliği bütün bölüm boyunca önem taşır. Birincisi **sıradır**: akıştan okunan veri yazıldığı sırayla gelir, bir veri atlanıp sonrakine geçilemez. Bu yüzden akışlarla dosyanın ortasındaki bir kayda doğrudan gidilemez; bunun için rastgele erişim gerekir. İkincisi **yöndür**: bir akış ya girdi ya çıktı akışıdır, ikisi birden olamaz. Aynı dosyayı hem okumak hem yazmak isteyen program iki ayrı akış açar.

8.1.2 Standart akışlar

Klavye ve ekran en basit G/Ç aygıtlarıdır. Bunlarla yapılan veri işleme geçicidir: klavyeden girilen ve ekrana yazılan veri saklanmaz, program kapanınca kaybolur. Ayrıca anındadır: kullanıcı yazdıkça girdi gelir, program yazdıkça çıktı ekranda görünür. Her Java programı çalışmaya başlarken işletim sisteminden üç hazır akış alır. Bunlara **standart akışlar** denir (Tablo 8.1).

Akış	Java nesnesi	Tipi	Varsayılan uç
standart girdi	`System.in`	`InputStream`	klavye
standart çıktı	`System.out`	`PrintStream`	ekran
standart hata	`System.err`	`PrintStream`	ekran

Tablo 8.1: Java programının üç standart akışı.



Şekil 8.2: Standart akışlar. Klavye programa `System.in` ile bağlanır; program ekrana `System.out` ve `System.err` ile yazar. Numaralar işletim sistemindeki kanal numaralarıdır (0, 1, 2).

`System.out` ile `System.err` ikisi de ekrana yazar. Peki neden iki ayrı akış vardır? Fark, çıktı yönlendirildiğinde ortaya çıkar. Komut satırında `java Uygulama > sonuc.txt` yazılırsa `>` işareti yalnız standart çıktıyı `sonuc.txt` dosyasına yönlendirir. `System.err`'e yazılanlar ekranda kalır. Böylece hata iletileri hem çıktı dosyasını kirletmez hem de gözden kaçmaz. Programın asıl sonucu `System.out`'a, uyarılar ve hata iletileri `System.err`'e yazılır.

`System.in`'in tipi `InputStream`'dir; yani ham bayt verir. Kullanıcı klavyede `42` yazıp Enter'a bastığında akışa üç bayt girer: `'4'`, `'2'` ve satır sonu. Bu baytları bir `int` değerine çevirmek programcıya düşer. Bu işi `java.util.Scanner` sınıfı yapar.

8.1.3 İlk örnek: okuma ve yazma

Aşağıdaki program üç standart akışı birlikte kullanır. Klavyeden bir sayı okur, iki satırı standart çıktıya, bir satırı standart hataya yazar.

```
import java.util.Scanner;

public class AkisOrnek {
  public static void main(String[] args) {
    // Giriş akışından veri okuma
    Scanner scanner = new Scanner(System.in);
    int kullaniciGirdisi = scanner.nextInt();
    // Çıkış akışına veri yazma
    System.out.println("Merhaba, Dünya!");
    System.out.println("Girdiğiniz sayı: " + kullaniciGirdisi);
    System.err.println("Bu satır hata akışına yazıldı.");
  }
}
```

Kullanıcı `42` yazarsa konsolda şu satırlar görünür (ilk satır kullanıcının yazdığı girdidir):

```
42
Merhaba, Dünya!
Girdiğiniz sayı: 42
Bu satır hata akışına yazıldı.
```

Program satır satır şöyle çalışır:

- `new Scanner(System.in)` klavye akışını okuyan bir tarayıcı oluşturur. `Scanner` `java.util` paketindedir; bu yüzden dosyanın başındaki `import` satırı şarttır.
- `scanner.nextInt()` kullanıcı bir sayı yazıp Enter'a basana kadar bekler. Bu beklemeye **bloklanma** (*blocking*) denir: program o satırda durur ve girdi gelince devam eder.
- `System.out.println` çıktı akışına bir satır yazar ve satırı bitirir.
- `System.err.println` hata akışına yazar. Çoğu geliştirme ortamı bu satırı kırmızı gösterir.

Kullanıcı sayı yerine `abc` yazarsa `nextInt()` bu metni sayıya çeviremez ve `java.util.InputMismatchException` fırlatır. Program yakalanmamış istisnayla sona erer:

```
Exception in thread "main" java.util.InputMismatchException
        at java.base/java.util.Scanner.throwFor(Scanner.java:947)
        at java.base/java.util.Scanner.next(Scanner.java:1602)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2267)
        at java.base/java.util.Scanner.nextInt(Scanner.java:2221)
        at AkisOrnek.main(AkisOrnek.java:7)
```

Bu istisna kontrol edilmeyen bir istisnadır; derleyici onu yakalamayı zorunlu tutmaz. Kullanıcıdan gelen girdiye güvenilmeyen programlarda `hasNextInt()` ile önce denetim yapılır ya da istisna yakalanıp kullanıcıdan yeniden girdi istenir.

8.1.4 Kod izleme: klavyeden okuma

Aşağıdaki program kullanıcının adını ve yaşını sorar, ertesi yılki yaşını hesaplar. Adı okumak için `nextLine()`, yaşı okumak için `nextInt()` kullanılır. İki metodun satır sonunu farklı ele alması bu bölümün en sık görülen hatalarından birine yol açar; izleme bu farkı görünür kılar.

```
import java.util.Scanner;

public class KonsolOkuma {
  public static void main(String[] args) {
    Scanner klavye = new Scanner(System.in);
    System.out.print("Adınız: ");
    String ad = klavye.nextLine();
    System.out.print("Yaşınız: ");
    int yas = klavye.nextInt();
    int seneye = yas + 1;
    System.out.println("Merhaba " + ad + "!");
    System.out.println("Seneye " + seneye + " yaşındasın.");
    klavye.close();
  }
}
```

Kullanıcı ad olarak `Ayşe Yılmaz`, yaş olarak `19` yazsın. Tablo 8.2 programın her adımını verir. Satır numaraları yukarıdaki kodun satırlarıdır.

Adım	Satır	Değişkenler	Yeni çıktı
1	5: `Scanner klavye = new Scanner(System.in);`	klavye=Scanner(System.in)	—
2	6: `System.out.print("Adınız: ");`	klavye=Scanner(System.in)	`Adınız:`
3	7: `String ad = klavye.nextLine();`	ad="Ayşe Yılmaz"	`Ayşe Yılmaz`
4	8: `System.out.print("Yaşınız: ");`	ad="Ayşe Yılmaz"	`Yaşınız:`
5	9: `int yas = klavye.nextInt();`	ad="Ayşe Yılmaz", yas=19	`19`
6	10: `int seneye = yas + 1;`	ad="Ayşe Yılmaz", yas=19, seneye=20	—
7	11: `System.out.println("Merhaba " + ad + "!");`	ad="Ayşe Yılmaz", yas=19, seneye=20	`Merhaba Ayşe Yılmaz!`
8	12: `System.out.println("Seneye " + seneye + " yaşındasın.");`	ad="Ayşe Yılmaz", yas=19, seneye=20	`Seneye 20 yaşındasın.`

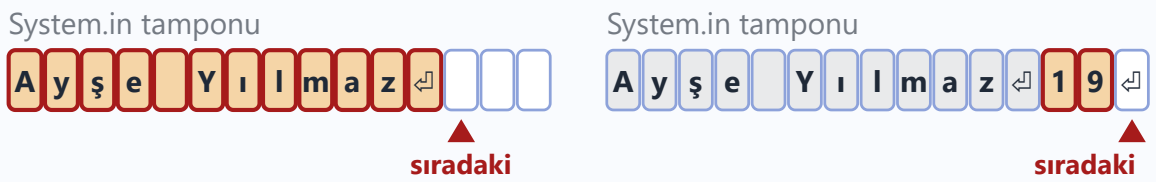
Adım	Satır	Değişkenler	Yeni çıktı
9	13: `klavye.close();`	ad="Ayşe Yılmaz", yas=19, seneye=20	—

Tablo 8.2: `KonsolOkuma` programının izlemesi. "Yeni çıktı" sütunu o adımda konsola eklenen metni gösterir; ⏎ satır sonudur. Konsolda kullanıcının yazdığı `Ayşe Yılmaz` ve `19` de görünür.

Birinci adımda `Scanner` nesnesi `System.in` akışına bağlanır. Henüz hiçbir şey okunmamıştır ve klavye tamponu boştur. İkinci adımda `print` (`println` değil) `Adınız:` sorusunu yazar. `print` satırı bitirmediği için kullanıcının cevabı sorunun hemen yanına yazılır.

Üçüncü adım önemlidir. Kullanıcı `Ayşe Yılmaz` yazıp Enter'a basar. Klavye tamponunda artık `Ayşe Yılmaz⏎` vardır. `nextLine()` satır sonuna kadar olan *her şeyi*, boşluk dahil alır ve satır sonunu da tampondan atar. `ad` değişkenine `"Ayşe Yılmaz"` atanır; satır sonu bu metnin parçası değildir.

Beşinci adımda kullanıcı `19` yazıp Enter'a basar. `nextInt()` baştaki boşlukları atlar, rakamları okur ve `int` değeri 19'a çevirir. Satır sonuna ise dokunmaz: `⏎` tamponda kalır. Bu programda sorun çıkmaz, çünkü ardından bir okuma daha yapılmıyor. Sonraki adımlarda `yas` artık sıradan bir `int` değeridir: `19 + 1 = 20`. Çıktı satırları dizgi birleştirmesiyle kurulur: `"Seneye " + 20 + " yaşındasın."`.



Şekil 8.3: Klavye tamponunun iki anı. (a) `nextLine()` satırın tamamını alır ve satır sonunu atar. (b) `nextInt()` rakamları alır; satır sonu tamponda kalır.

Son satırdaki `klavye.close()` yalnız tarayıcıyı değil, onun sardığı `System.in` akışını da kapatır. Kapanan standart girdi yeniden açılamaz. Program içinde daha sonra yeni bir `Scanner(System.in)` oluşturulsa bile klavyeden okuma yapılamaz. Aşağıdaki küçük deneme bunu gösterir: ilk tarayıcı kapatıldıktan sonra ikinci tarayıcı, girdide bekleyen bir satır olduğu hâlde `NoSuchElementException` fırlatır.

```
okunan: bir
java.util.NoSuchElementException: No line found
```

Bu yüzden klavyeyi okuyan tarayıcı programda bir kez oluşturulur ve gerekiyorsa yalnız programın sonunda kapatılır. Dosyayı okuyan tarayıcılar ise iş bitince hemen kapatılmalıdır; bu fark dosya kesimlerinde yeniden ele alınır.

8.1.5 `nextInt`'ten sonra `nextLine`

Tampondaki satır sonunun etkisi, sıra tersine çevrildiğinde ortaya çıkar. Önce yaş, sonra ad okunsun:

```
import java.util.Scanner;
public class SonraSatir {
  public static void main(String[] args) {
    Scanner klavye = new Scanner(System.in);
    System.out.print("Yaş: ");
    int yas = klavye.nextInt();
    System.out.print("Ad: ");
    String ad = klavye.nextLine();
    System.out.println("[" + ad + "] " + yas);
  }
}
```

Kullanıcı `19` yazıp Enter'a basar. `nextInt()` 19'u okur ve satır sonunu tamponda bırakır. Ardından `nextLine()` çağrılır. Bu metot satır sonuna kadar olan metni ister. Tamponda zaten bir satır sonu bekler; önündeki metin boştur. `nextLine()` hiç beklemeden boş dizgiyi döndürür. Kullanıcı adını yazmaya fırsat bulamadan program biter:

```
Yaş: Ad: [] 19
```

Sık yapılan hata

`nextInt()`, `nextDouble()` ya da `next()` çağrısından hemen sonra `nextLine()` ile metin okumak. İlk `nextLine()` önceki satırdan kalan satır sonunu okur ve boş dizgi döndürür. İki çözüm vardır: araya satır sonunu atan fazladan bir `klavye.nextLine();` konur ya da sayı da satır olarak okunup çevrilir: `int yas = Integer.parseInt(klavye.nextLine());`.

İkinci çözüm genellikle daha temizdir. Her okuma bir satırın tamamını tükettiği için tamponda artık bir şey kalmaz. Kullanıcı sayının yanına fazladan boşluk yazarsa `parseInt` hata verir; bu durumda `klavye.nextLine().trim()` kullanılır.

8.2 Dosyalar ve kodlama

Klavye ve ekranla yapılan alışveriş geçicidir. Kalıcı veri için dosya gerekir. Bu kesim bir dosyanın aslında ne olduğunu, metin ve ikili dosyaların farkını ve karakterlerin baytlara nasıl çevrildiğini anlatır. Bu bilgiler sonraki kesimlerde hangi akış sınıfının seçileceğini belirler.

8.2.1 Dosya: diskteki bayt dizisi

Dosya, veriyi diskte kalıcı olarak saklar. Program kapansa ya da bilgisayar yeniden başlasa da veri yerinde durur. RAM'deki değişkenler ise programla birlikte kaybolur. Her dosyadaki veri sonuçta 0'lar ve 1'lerden, yani bir **bayt dizisinden** oluşur. Dosyalar arasındaki fark bu baytların nasıl yorumlandığından doğar.

> **Tanım 8.2**
>
> **Metin dosyasında** (*text file*) baytlar bir karakter kodlamasına göre harflere çevrilir; insan dosyayı bir metin düzenleyicisiyle okuyabilir. **İkili dosyada** (*binary file*) baytlar sayı, piksel, ses örneği ya da makine komutu gibi verilerdir; onları ancak dosyanın biçimini bilen bir program anlamlandırır.

Bir dosyanın baytlarını görmek için onaltılık (*hexadecimal*, hex) düzenleyici kullanılır. Bu araç her baytı iki onaltılık rakamla gösterir. İçinde `Merhaba, dünya!` yazan `dosya.txt` onaltılık düzenleyicide şöyle görünür:

```
0000  4D 65 72 68 61 62 61 2C  Merhaba,
0008  20 64 C3 BC 6E 79 61 21   d..nya!
```

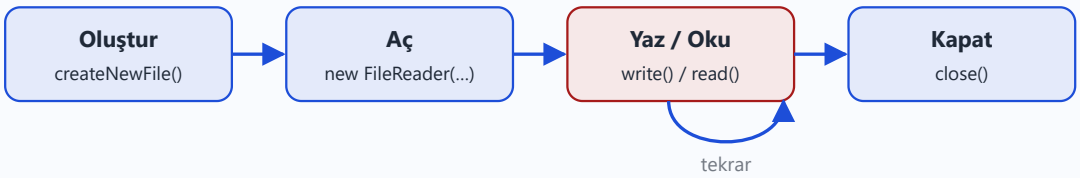
Soldaki sütun satırın ilk baytının konumudur: `0000` ve `0008`. Ortada her baytın onaltılık değeri, sağda ASCII karşılığı vardır. İlk bayt `4D` 'dir: 4 · 16 + 13 = 77, ASCII'de büyük `M` harfi. Metin 15 karakterdir ama dosya 16 bayt tutar. Fark `ü` harfinden gelir: UTF-8 kodlamasında `ü` iki bayttır, `C3 BC`. Sağ sütundaki iki nokta bu iki baytın ASCII'de karşılığı olmadığını gösterir. Bu dosya Java ile üretilip baytları yazdırıldığında aynı 16 değer elde edilir:

```
16 bayt: 4D 65 72 68 61 62 61 2C 20 64 C3 BC 6E 79 61 21
1 bayt: 7F
1382 bayt: CA FE BA BE 00 00 00 41 00 5C 0A 00 02 00 03 07 00 04 0C 00 05 00 06 01 00 10
6A 61 76 61 2F 6C
·······A·\·················java/lang/Object···<init>···()V·····java/io/FileWriter·····dos
ya.txt·············(Ljava/lang
major = 65
```

Çıktının ilk satırı bu dosyanın, ikinci satırı `veri.dat` dosyasının baytlarını verir. Kalan satırlar bir `.class` dosyasının başını gösterir; bunlar ikili dosyalar ele alınırken açıklanır.

8.2.2 Dosya işleme süreci

Hangi dil kullanılırsa kullanılsın, bir dosyayla çalışmak aynı adımlardan geçer. Şekil 8.4 adımları, Tablo 8.3 her adımın Java'daki karşılığını verir.



Şekil 8.4: Dosya işleme süreci. Dosya oluşturulur ya da açılır, okunur veya yazılır ve sonunda kapatılır. Okuma ve yazma adımı istenen kadar tekrarlanabilir.

Adım	Ne olur?	Java'da
Oluşturma	Diskte yeni, boş bir dosya oluşur.	`file.createNewFile()`; yazmak için açmak da dosyayı oluşturur
Açma	Var olan dosyaya bir akış bağlanır.	`new FileInputStream(ad)`, `new FileReader(ad)`, `new Scanner(file)`
Yazma	Veri dosyaya aktarılır.	`write()`, `print()`, `println()`
Okuma	Veri dosyadan programa alınır.	`read()`, `readLine()`, `nextLine()`
Kapatma	Tamponda kalan veri diske yazılır, işletim sistemi kaynağı bırakılır.	`close()` ya da try-with-resources

Tablo 8.3: Dosya işleme adımları ve Java'daki karşılıkları.

Son adım en sık unutulan adımdır. Kapatılmayan bir dosya iki soruna yol açar. Birincisi **veri kaybıdır**: yazılan baytların bir kısmı bellekteki bir tamponda bekler ve dosya kapatılmadan diske gitmeyebilir. İkincisi **kaynak sızıntısıdır**: işletim sistemi her açık dosya için bir tanıtıcı (*file handle*) ayırır. Bir sürecin aynı anda açık tutabileceği dosya sayısı sınırlıdır. Uzun süre çalışan ve dosyalarını kapatmayan bir program bir süre sonra yeni dosya açamaz hâle gelir. Windows'ta açık kalan bir dosya ayrıca silinemez ve taşınamaz.

8.2.3 İlk dosya programı

Aşağıdaki program bir metin dosyasına bir cümle, bir ikili dosyaya tek bir bayt yazar. Sonra ikili dosyadaki baytı geri okur ve metin dosyasının boyutunu yazdırır.

```
import java.io.*;

public class IlkDosya {
  public static void main(String[] args) throws IOException {
    // Metin dosyasına yazma
    FileWriter writer = new FileWriter("dosya.txt");
    writer.write("Merhaba, dünya!");
    writer.close();
    // İkili dosyaya bir byte yazma
    FileOutputStream cikis = new FileOutputStream("veri.dat");
    cikis.write(127);
    cikis.close();
    // İkili dosyadan okuma
    FileInputStream stream = new FileInputStream("veri.dat");
    int veri = stream.read();
    stream.close();
    System.out.println("Okunan byte: " + veri);
    File metin = new File("dosya.txt");
    System.out.println("dosya.txt: " + metin.length() + " byte");
  }
}
```

```
Okunan byte: 127
dosya.txt: 16 byte
```

Program çalıştıktan sonra `dosya.txt` 'nin içeriği şudur:

```
Merhaba, dünya!
```

Programda üç farklı akış sınıfı vardır:

- `FileWriter` **metin** yazar. Kendisine verilen karakterleri bir karakter kodlamasıyla baytlara çevirip dosyaya gönderir. Java 18'den beri varsayılan kodlama UTF-8'dir; `ü` bu yüzden iki bayt olur ve 15 karakterlik metin 16 bayt tutar.
- `FileOutputStream` **ham bayt** yazar. `write(127)` çağrısı dosyaya tek bir bayt koyar: onaltılık `7F`, ikilik `0111 1111`. Bir karakter dönüşümü yapılmaz.
- `FileInputStream` ham bayt okur. `read()` bir baytı 0–255 arasında bir `int` olarak döndürür. Okunacak bayt kalmamışsa −1 döndürür; bu ayrıntı kopyalama kesiminde önem kazanır.

`main` metodunun başlığındaki `throws IOException`, dosya işlemlerinin kontrol edilen istisna fırlatabileceğini bildirir. Dosya bulunamayabilir, disk dolabilir, izin olmayabilir. Derleyici bu durumların ele alınmasını zorunlu tutar: ya `try-catch` ile yakalanır ya da metot başlığında `throws` ile bildirilir. Bu kısa örnekler ikinci yolu seçer. Gerçek programlarda hata iletisini kullanıcıya anlaşılır biçimde vermek için yakalamak daha uygundur.

8.2.4 Metin dosyaları ve karakter kodlaması

Metin dosyaları yazdırılabilir karakterler içerir: harfler, rakamlar, noktalama işaretleri, boşluk ve satır sonu. Not Defteri ya da VS Code gibi metin düzenleyicileriyle açılıp düzenlenebilirler. Kaynak kodlar (`.java` , `.c`), `.txt` , `.csv` ve `.html` dosyaları metin dosyasıdır.

Bilgisayar harf saklayamaz, yalnız sayı saklar. Bu yüzden her karaktere bir sayı karşılık getirilir. Bu eşleme kuralına karakter kodlaması denir.

Tanım 8.3

Karakter kodlaması (*character encoding*) ya da karakter kümesi (*character set*), her karakterin hangi sayıyla, yani hangi bayt ya da baytlarla saklanacağını belirleyen kuraldır. Bir metin dosyasını yazan ve okuyan program aynı kodlamayı kullanmalıdır.

Tarih içinde birçok kodlama ortaya çıkmıştır. Bu derste karşılaşılan dördü Tablo 8.4'te özetlenir.

Kodlama	Karakter başına	Kapsam
ASCII	7 bit (1 bayt)	128 karakter: İngilizce harfler, rakamlar, noktalama ve kontrol karakterleri.
ISO-8859-1 (Latin-1)	1 bayt	256 karakter, Batı Avrupa dilleri. `é` , `ü` , `ö` , `ç` vardır; `ş` , `ğ` , `ı` yoktur.
ISO-8859-9 (Latin-5)	1 bayt	Latin-1'in Türkçe sürümü. Latin-1'deki altı harfin yerine `ğ Ğ ı İ ş Ş` gelir.
UTF-8	1–4 bayt	Unicode'un evrensel kodlaması; bütün dünya dilleri. Web'in ve Java 18'den beri Java'nın varsayılan kodlaması.

Tablo 8.4: Sık kullanılan karakter kodlamaları.

UTF-8 değişken uzunluklu bir kodlamadır. ASCII'deki 128 karakter UTF-8'de de aynı tek baytla yazılır; bu yüzden yalnız İngilizce harf içeren bir ASCII dosyası aynı zamanda geçerli bir UTF-8 dosyasıdır. Türkçe harfler ve öteki Latin harfleri iki bayt, avro işareti `€` gibi semboller üç bayt, emojiler dört bayt tutar. Aşağıdaki deneme `getBytes` metoduyla bu sayıları ölçer:

```
32 karakter, UTF-8 41, ISO-8859-9 32, UTF-16BE 64
IÅ<9f>Ä±k, Å<9f>imÅ<9f>ek, Ã§iÄ<9f>, Ã¶Ä<9f>renci, Ã<9c>nye
ş: ASCII 3F, UTF-8 C5 9F, ISO-8859-9 FE, cp1254 FE
þimdi
şimdi
Çağrı UTF-8: 8, ISO-8859-9: 5, UTF-16BE: 10
€ UTF-8: 3, 😀 UTF-8: 4, 😀 length(): 2
varsayılan: UTF-8 / native.encoding=Cp1254
4D 65 72 68 61 62 61 2C 20 64 C3 BC 6E 79 61 21
```

İlk satır, 32 karakterlik Türkçe bir cümlenin UTF-8'de 41, ISO-8859-9'da 32, UTF-16'da 64 bayt tuttuğunu gösterir. Cümlede iki baytlık dokuz Türkçe harf vardır: 32 + 9 = 41. İkinci satır aynı baytların yanlış kodlamayla çözülmesidir; bu bozulma kodlama belirtme kesiminde ayrıntılı ele alınır. `ş` satırı ise bir harfin dört kodlamadaki baytlarını verir. ASCII'de `ş` yoktur; Java onun yerine soru işaretini (`3F`) yazar. "Çağrı" satırı da ilginç bir ayrıntı verir: `"Çağrı"` beş karakterdir ama UTF-8'de 8 bayt tutar. `Ç` , `ğ` ve `ı` ikişer, `a` ve `r` birer bayttır: 2 + 1 + 2 + 1 + 2 = 8. Aynı metin ISO-8859-9 ile 5, UTF-16 ile 10 bayttır. 😀 için `length()` 2 döndürür. Java'nın `char` tipi 16 bittir ve 65.535'ten büyük kodlu karakterler iki `char` ile temsil edilir. Dolayısıyla bir dizginin uzunluğu, ne karakter sayısına ne de bayt sayısına her zaman eşittir.

Çıktıdaki "varsayılan" satırı da dikkat ister. Java 18 ve sonrasında varsayılan kodlama işletim sisteminden bağımsız olarak UTF-8'dir. İşletim sisteminin kendi kodlaması ise `native.encoding` özelliğinde durur; Türkçe Windows'ta bu Windows-1254'tür (Cp1254). Java 17 ve öncesinde varsayılan kodlama doğrudan bu işletim sistemi ayarıydı. Aynı program Windows'ta ve Linux'ta farklı baytlar yazabiliyordu.

8.2.5 İkili dosyalar

İkili dosyalar baytları karakter olarak değil, programın belirlediği bir biçimde (*format*) saklar: sayılar, pikseller, ses örnekleri, makine kodu. Bu baytlar ancak biçimi bilen program için anlamlıdır; insan onları doğrudan okuyamaz. Çalıştırılabilir dosyalar (`.exe` , Java'nın `.class` dosyaları), görüntüler (`.png` , `.jpg`), müzik (`.mp3`), video (`.mp4`) ve veri dosyaları (`.dat`) ikili dosyadır. İkili dosyaların avantajı daha küçük olmaları ve daha hızlı işlenmeleridir: sayıyı metne, metni sayıya çevirmek gerekmez.

Önceki programın derlenmesiyle oluşan `IlkDosya.class` 'ın ilk 32 baytı onaltılık düzenleyicide şöyle görünür:

```
0000  CA FE BA BE 00 00 00 41  .......A
0008  00 5C 0A 00 02 00 03 07  .\......
0010  00 04 0C 00 05 00 06 01  ........
0018  00 10 6A 61 76 61 2F 6C  ..java/l
```

Her `.class` dosyası `CA FE BA BE` baytlarıyla başlar. Bu dört bayta **sihirli sayı** (*magic number*) denir; JVM dosyanın gerçekten bir sınıf dosyası olduğunu bu sayıdan anlar. Sonraki dört bayt sürüm bilgisidir. `00 41` = 65, Java 21'in sınıf dosyası sürümüdür; bu sayı JDK 21 ile derlenen dosyada Java ile de okunmuştur (önceki çıktıdaki `major = 65`). Ardından sabitler havuzu gelir. Okunabilir parçalar da görünür: `java/lang/Object` gibi sınıf adları dosyada metin olarak saklanır. Aynı dosya Not Defteri ile açıldığında anlamsız bir karakter dizisi çıkar:

```
Êþº¾···A·\·····················java/lang/Object···<init>···()V·····java/io/FileWriter·····dos
ya.txt·············(
```

Not Defteri baytları bir metin kodlamasıyla çevirmeye çalışır. `CA`, `FE`, `BA` ve `BE` Latin-1'de sırasıyla `Ê`, `þ`, `º` ve `¾` karakterleridir. Yazdırılamayan baytlar ise nokta ya da boşluk gibi görünür. Bir ikili dosyayı metin düzenleyicisinde açıp kaydetmek onu bozabilir: düzenleyici bazı baytları değiştirebilir ya da satır sonlarını dönüştürebilir.

8.2.6 Genişletilmiş ASCII ve Türkçe karakterler

ASCII 7 bittir ve 128 karakteri kapsar. Bir bayt ise 8 bittir. Kalan bir bit, 128 yeni karakter için yer açar.

> **Tanım 8.4**
>
> **Genişletilmiş ASCII** (*extended ASCII*), 8 bitin tamamını kullanarak $2^8 = 256$ karakter temsil eden kodlamalara verilen genel addır. 0–127 arası standart ASCII karakterleridir ve her tabloda aynıdır. 128–255 arası özel karakterlere, sembollere ve ulusal harflere ayrılmıştır.

Tek bir "genişletilmiş ASCII" yoktur. Üst yarının hangi karakterleri tuttuğu **kod sayfasına** (*code page*) göre değişir. Türkçe için iki kod sayfası kullanılır: ISO-8859-9 (Latin-5) ve Windows-1254. İkisi Türkçe harflerde aynı baytları kullanır. Tablo 8.5 aynı baytların Latin-1 ve Latin-5'teki karşılığını yan yana verir.

Bayt	D0	DD	DE	F0	FD	FE	FC	E7
ISO-8859-1	Ð	Ý	Þ	ð	ý	þ	ü	ç
ISO-8859-9	Ğ	İ	Ş	ğ	ı	ş	ü	ç

Tablo 8.5: Aynı baytın iki kod sayfasındaki anlamı.

`FC` ve `E7` iki tabloda da `ü` ve `ç`'dir; bu harfler Batı Avrupa dillerinde de vardır. İlk altı bayt ise tablodan tabloya değişir. Aynı `FE` baytı bir tabloda `ş`, ötekinde İzlandaca `þ` harfidir. Türkçe bir dosya yanlış kod sayfasıyla açıldığında "şimdi" yerine "þimdi" gibi bozuk metinler çıkar (Şekil 8.5). Önceki çıktıdaki `þimdi` ve `şimdi` satırları aynı beş bayttan iki kod sayfasıyla elde edilmiştir.



Şekil 8.5: `FE` baytı ISO-8859-9'da `ş`, ISO-8859-1'de `þ`'dir. Dosyayı yanlış kod sayfasıyla açmak metni bozar.

Bu sorunun kalıcı çözümü bütün dilleri tek tabloda toplayan Unicode ve onun bayt kodlaması UTF-8'dir. UTF-8'de her karakterin tek bir bayt dizisi vardır; okuyanın hangi kod sayfasını seçeceğini tahmin etmesi gerekmez. Bugün yeni yazılan her metin dosyası için UTF-8 seçilir.

8.2.7 Bir dosyaya 127 yazmak

Bir sayının dosyada nasıl saklanacağı veri tipine, kullanım amacına ve dosya biçimine bağlıdır. 127 sayısı iki temelden farklı yolla yazılabilir.

- **Metin olarak** yazıldığında sayı önce karakterlere çevrilir: `"127"`. Her rakam bir ASCII baytı tutar. Burada sık karıştırılan bir nokta vardır: `'1'` karakterinin kodu 1 değil, 49'dur. `'2'` 50, `'7'` 55'tir. Dosyada `31 32 37` baytları durur.
- **İkili olarak** yazıldığında sayının ikilik değeri, `0111 1111`, tipin boyutu kadar bayta doğrudan yazılır. `byte` için 1, `short` için 2, `int` için 4 bayt kullanılır. Sayısal veri için bu yol daha verimlidir.



Şekil 8.6: 127 sayısının dosyadaki gösterimleri. Metin dosyasında her rakam bir ASCII baytıdır ve toplam 3 bayt tutar. İkili dosyada sayının kendisi tipin boyutu kadar bayta yazılır: 1, 2 ya da 4 bayt.

Aşağıdaki program bu beş gösterimi gerçekten dosyaya yazar ve dosyaların boyutunu ölçer. Metin yazmak için `OutputStreamWriter`, ikili yazmak için `DataOutputStream` kullanılır. Kısalık için `import java.io.*;` ve `import java.nio.charset.*;` satırları gösterilmemiştir.

```
public class Sayi127 {
  static void metinYaz(String ad, Charset kod) throws IOException {
    Writer w = new OutputStreamWriter(new FileOutputStream(ad), kod);
    w.write(String.valueOf(127));           // "127": 3 karakter
    w.close();
  }
  static void ikiliYaz(String ad, int boyut) throws IOException {
    FileOutputStream fos = new FileOutputStream(ad);
    DataOutputStream d = new DataOutputStream(fos);
    if (boyut == 1) d.writeByte(127);       // 01111111
    else if (boyut == 2) d.writeShort(127); // 00000000 01111111
    else d.writeInt(127);                   // 4 byte
    d.close();
  }
  public static void main(String[] args) throws IOException {
    metinYaz("metin.txt", StandardCharsets.US_ASCII);
    metinYaz("metin16.txt", StandardCharsets.UTF_16BE);
    ikiliYaz("b.dat", 1);  ikiliYaz("s.dat", 2);  ikiliYaz("i.dat", 4);
    String[] adlar = {"metin.txt", "metin16.txt",
             "b.dat", "s.dat", "i.dat"};
    for (String ad : adlar)
      System.out.println(ad + ": " + new File(ad).length());
  }
}
```

```
metin.txt: 3
metin16.txt: 6
b.dat: 1
s.dat: 2
i.dat: 4
```

`metinYaz` metodu bir `FileOutputStream`'i `OutputStreamWriter` ile sarar. `OutputStreamWriter` karakterleri, kendisine verilen kodlamayla baytlara çeviren köprü sınıftır. ASCII ile `"127"` üç bayt tutar. `UTF_16BE` ise her karakteri iki baytla yazar ve 6 bayt tutar. Bu kodlamanın adındaki BE (*big-endian*) baytların sırasını sabitler; bu yüzden dosyanın başına sıra işareti (*byte order mark*, BOM) yazılmaz. Sıra işaretli `UTF_16` kullanılsaydı başa iki bayt `FE FF` eklenir ve dosya 8 bayt tutardı.

`ikiliYaz` metodu `DataOutputStream` kullanır. Bu sınıf ilkel tipleri sabit boyutlu ikili gösterimleriyle yazar: `writeByte` 1, `writeShort` 2, `writeInt` 4 bayt. Baytlar en anlamlı bayt önce gelecek biçimde yazılır. Dosyaların gerçek içerikleri ölçümü doğrular:

```
metin.txt: 3
metin16.txt: 6
b.dat: 1
s.dat: 2
i.dat: 4
metin.txt -> 31 32 37
metin16.txt -> 00 31 00 32 00 37
b.dat -> 7F
s.dat -> 00 7F
i.dat -> 00 00 00 7F
UTF_16 (BOM'lu): 8
iki int: 8
7 2000000000
yanlış sıra/tip: 0 7
EOFException: null
```

Çıktının son satırları `DataInputStream` ile ilgili iki önemli kuralı gösterir. İkili dosyadaki değerler yalnız **yazıldıkları sırayla ve tiple** doğru geri okunur. İki `int` yazılan dosya iki `readShort()` ile okunursa 0 ve 7 gelir; yani ilk `int`'in dört baytı iki parçaya bölünür. Dosyada artık veri yokken `readInt()` çağrılırsa `EOFException` (*end of file*, dosya sonu) fırlatılır. Metin dosyası insana biçimini kendisi anlatır; ikili dosyanın biçimini ise yazan program ile okuyan programın önceden bilmesi gerekir.

8.2.8 Metin mi, ikili mi?

Tablo 8.6 127 sayısının beş gösterimini boyutlarıyla karşılaştırır.

127 nasıl yazıldı?	Baytlar	Boyut
ASCII / UTF-8 metin dosyası	`31 32 37`	3 bayt
UTF-16 metin dosyası (BOM'suz)	`00 31 00 32 00 37`	6 bayt
İkili, `byte`	`7F`	1 bayt
İkili, `short`	`00 7F`	2 bayt
İkili, `int`	`00 00 00 7F`	4 bayt

Tablo 8.6: 127 sayısının dosyadaki boyutu.



Şekil 8.7: 127'nin UTF-8 metin gösterimi (üç rakam baytı) ve `writeInt` ile yazılan dört baytlık ikili gösterimi.

Tabloda ikili gösterim her zaman daha küçük görünüyor, ama bu yalnız 127 için doğrudur. İkili gösterimde boyut **tipe** bağlıdır: bir `int` her zaman 4 bayttır, değeri 7 de olsa 2.000.000.000 de olsa. Metin gösteriminde boyut **basamak sayısına** bağlıdır: 7 bir bayt, 2.000.000.000 ise on bayt tutar. Küçük sayılarda metin daha kısa olabilir; büyük sayılarda ikili gösterim açıkça kazanır. Ayrıca ikili okuma sayıyı çözümlemek zorunda kalmadığı için hızlıdır.

Seçim kullanım amacına göre yapılır:

- **Metin** insanların okuyacağı veriler içindir. Her düzenleyiciyle açılır, platformlar arasında kolay taşınır, bozulduğunda gözle onarılabilir. Kayıt dosyaları, ayar dosyaları ve CSV tabloları metindir.
- **İkili** sayısal verinin verimli saklanması içindir. Daha küçük ve hızlıdır, ama yazan programın biçimini bilmek gerekir. Resim, ses ve veri tabanı dosyaları ikilidir.

Java'nın bu iki yol için araçları bellidir. `DataOutputStream` ilkel tipleri ikili yazar (`writeByte`, `writeShort`, `writeInt` ...). `DataInputStream` onları aynı sırayla geri okur. `OutputStreamWriter` ise karakterleri verilen kodlamayla baytlara çevirir.

8.3 Dosya sistemi: `File` sınıfı

Bir dosyanın içeriğini okumadan önce çoğu zaman dosyanın kendisi hakkında sorular sorulur. Dosya var mı? Bir dosya mı, klasör mü? Boyutu ne? Okunabilir mi? Bu soruların cevabını `java.io.File` sınıfı verir.

8.3.1 `File` bir yoldur

> **Tanım 8.5**
>
> `File` nesnesi diskteki bir **yolu** (*path*) temsil eder: bir dosyanın ya da klasörün adı ve konumu. `File` dosyanın içeriğini okumaz ve yazmaz; bu iş akışların işidir. `File` yalnız dosya sistemine erişip yol hakkındaki soruları cevaplar ve dosya ya da klasör oluşturma, silme, listeleme gibi işlemleri yapar.

Bu ayrımın en önemli sonucu şudur: `new File("ornek.txt")` diskte hiçbir şey oluşturmaz. Yapıcı yalnız yolu bir nesnede saklar. Dosya diskte olmasa bile nesne oluşur ve istisna fırlatılmaz. Olmayan bir dosya için `exists()` `false`, `length()` 0 döndürür. Dosya bulunamadığında istisna ancak dosyayı okumak için bir akış açılırken gelir; örneğin `new FileReader(f)` `FileNotFoundException` fırlatır.

```
false 0 yok.txt false
mkdir: false, mkdirs: true
ikinci createNewFile: false
dolu klasör delete: false
lastModified(yok) = 0
1 a/b
```

Bu çıktının ilk satırı olmayan `yok.txt` için `exists()`, `length()`, `getName()` ve `isFile()` değerlerini verir. Kalan satırlar aşağıda açıklanan metotların ince noktalarını gösterir. Tablo 8.7 sık kullanılan metotları özetler.

Metot	Döndürdüğü
`exists()`	yol diskte var mı (`boolean`)
`isFile()` / `isDirectory()`	dosya mı / klasör mü
`canRead()`, `canWrite()`, `canExecute()`	okuma, yazma, çalıştırma izni
`createNewFile()`	dosya oluşturuldu mu (zaten varsa `false`)
`mkdir()` / `mkdirs()`	klasör oluşturuldu mu (`mkdirs` ara klasörleri de oluşturur)
`delete()`	silindi mi (içi dolu klasör silinmez)
`list()` / `listFiles()`	klasörün içeriği: `String[]` / `File[]`
`length()`	boyut, bayt cinsinden (`long`)
`lastModified()`	son değiştirilme zamanı, 1 Ocak 1970'ten beri milisaniye
`getName()`, `getAbsolutePath()`	ad, tam yol

Tablo 8.7: `File` sınıfının sık kullanılan metotları.

Bu metotların çoğu hata durumunda istisna fırlatmaz, `false` döndürür. `mkdir()` ara klasörler yoksa başarısız olur; yukarıdaki çıktıda `a/b/c` için `mkdir` `false`, `mkdirs` ise `true` döndürür. `createNewFile()` dosya zaten varsa `false` döndürür, var olan dosyaya dokunmaz. `delete()` içi dolu bir klasörü silmez ve yine `false` döndürür. Olmayan bir dosyanın `lastModified()` değeri 0'dır. Sonuç olarak bu metotların dönüş değeri denetlenmelidir. Değer yok sayılırsa işlem sessizce başarısız olur ve hata ancak çok sonra fark edilir.

8.3.2 Dosya ve klasör işlemleri

Aşağıdaki program bir klasör ve içinde boş bir dosya oluşturur, dosyanın özelliklerini sorgular, sonra her şeyi siler.

```
public class DosyaBilgi {
  public static void main(String[] args) throws IOException {
    File klasor = new File("ornek");
    File dosya = new File("ornek/dosya.txt");
    System.out.println("var mı: " + dosya.exists());
    System.out.println("mkdir: " + klasor.mkdir());
    System.out.println("oluştu: " + dosya.createNewFile());
    System.out.println("dosya mı: " + dosya.isFile());
    System.out.println("klasör mü: " + dosya.isDirectory());
    System.out.println("okunur: " + dosya.canRead());
    System.out.println("yazılır: " + dosya.canWrite());
    System.out.println("boyut: " + dosya.length());
    String[] icerik = klasor.list();
    System.out.println("klasörde: " + icerik.length + " öğe");
    long zaman = dosya.lastModified();
    System.out.println("zaman > 0: " + (zaman > 0));
    System.out.println("silindi: " + dosya.delete());
    System.out.println("var mı: " + dosya.exists());
    klasor.delete();
  }
}
```

```
var mı: false
mkdir: true
oluştu: true
dosya mı: true
klasör mü: false
okunur: true
yazılır: true
boyut: 0
klasörde: 1 öğe
zaman > 0: true
silindi: true
var mı: false
```

İlk satırda dosya henüz yoktur. `mkdir()` `ornek` klasörünü, `createNewFile()` içindeki boş dosyayı oluşturur; ikisi de `true` döndürür. Dosya yeni oluştuğu için boyutu 0'dır. `klasor.list()` klasördeki adları bir dizi olarak verir; dizinin uzunluğu 1'dir. `lastModified()` değeri program her çalıştığında farklıdır, bu yüzden program yalnız sıfırdan büyük olup olmadığını yazdırır. Son olarak dosya silinir ve `exists()` yeniden `false` olur. En son satır klasörü siler; klasör artık boş olduğu için silme başarılıdır.

Programdaki yolların ikisi de **göreli yoldur**: `"ornek/dosya.txt"` programın çalıştığı klasöre göre yorumlanır. Bu klasör bir geliştirme ortamında genellikle projenin kök klasörüdür. `getAbsolutePath()` göreli yolu tam yola çevirerek dosyanın gerçekte nerede arandığını gösterir. “Dosya bulunamadı” hatası alındığında ilk bakılacak yer burasıdır.

Yol ayırıcı Windows'ta ters bölü (`\`), Linux ve macOS'ta bölü (`/`) işaretidir. Java, Windows'ta da `/`'yu kabul eder; bu yüzden `"ornek/dosya.txt"` her sistemde çalışır. Sistemin kendi ayırıcısı `File.separator` sabitinde durur. Windows yollarında ters bölü kullanılacaksa Java dizgisinde iki kez yazılmalıdır: `"C:\\dersler\\notlar.txt"`. Tek ters bölü bir kaçış dizisi başlatır. `"C:\dersler"` derlenmez, çünkü `\d` geçerli bir kaçış dizisi değildir:

```
Ana.java:5: error: illegal escape character
    java.io.File f = new java.io.File("C:\dersler\notlar.txt");
                                        ^
1 error
```

Daha sinsi durum, kaçış dizisinin geçerli olduğu durumdur. `"C:\notlar.txt"` derlenir, çünkü `\n` satır sonudur. Yol sessizce `C:`, satır sonu ve `otlar.txt`'ye dönüşür:

```
DERLENDİ; çalıştı: C:
otlar.txt
```

8.4 Metin okuma ve yazma

Dosya işlemlerinin büyük kısmı metin dosyalarıyla yapılır. Bu kesim metin okumanın ve yazmanın yollarını, bunların arkasındaki sınıf yapısını ve kaynakların güvenle kapatılmasını anlatır.

8.4.1 `Scanner` ile dosya okuma

`Scanner` yalnız klavyeyi değil, bir dosyayı ya da bir `String`'i de okuyabilir. Kaynağı değiştirmek için yapıcıya `System.in` yerine bir `File` nesnesi verilir. Okuma metotları aynıdır. Aşağıdaki program bir dosyayı satır satır okuyup ekrana yazar. Gereken importlar `java.io.File`, `java.io.FileNotFoundException` ve `java.util.Scanner`'dır.

```
public class ScannerDosya {
  public static void main(String[] args) {
    Scanner scanner = null;
    try {
      scanner = new Scanner(new File("ornekDosya.txt"));
      // Dosyanın içeriğini okuma ve ekrana yazdırma
      while (scanner.hasNextLine()) {
        String satir = scanner.nextLine();
        System.out.println(satir);
      }
    } catch (FileNotFoundException e) {
      System.err.println("Dosya bulunamadı: " + e.getMessage());
    } finally {
      if (scanner != null)
        scanner.close();
    }
  }
}
```

`ornekDosya.txt` üç satır içersin: `Java`, `Giriş Çıkış` ve `Akışlar`. Program bunları aynen yazar:

```
Java
Giriş Çıkış
Akışlar
```

Programın üç noktası ayrıca açıklanmalıdır:

- **Kontrol edilen istisna.** `new Scanner(File)` dosya yoksa `FileNotFoundException` fırlatır. Bu kontrol edilen bir istisnadır: ya yakalanır ya da metot başlığında bildirilir. Program onu yakalar ve hata akışına bir ileti yazar.
- **Döngü.** `hasNextLine()` okunacak satır olup olmadığını sorar, `nextLine()` satırı alır. Dosya sonunda `hasNextLine()` `false` döndürür ve döngü biter.
- `finally` **ve** `null` **denetimi.** Dosya açılamazsa `scanner` değişkeni `null` kalır. `finally` bloğu her durumda çalışır; denetim olmasaydı `null` bir referans üzerinden `close()` çağrılırdı.

Son nokta göründüğünden önemlidir. `if (scanner != null)` satırı kaldırılıp olmayan bir dosya açılmaya çalışılırsa program önce “Dosya bulunamadı” iletisini yazar, sonra `finally`'de çöker:

```
Dosya bulunamadı: yokDosya.txt (The system cannot find the file specified)
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"java.util.Scanner.close()" because "<local1>" is null
        at Yok17.main(Yok17.java:12)
```

Bu kalıp hem uzun hem de hataya açıktır. Java 7 ile gelen **try-with-resources** aynı işi kısaca ve güvenle yapar. `Scanner` `AutoCloseable` arayüzünü uygular; bu yüzden `try`'ın parantezinde açılabilir ve blok bitince kendiliğinden kapanır. `finally` ve `null` denetimi gerekmez:

```
import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;
public class Twr17 {
  public static void main(String[] args) {
    try (Scanner scanner = new Scanner(new File("ornekDosya.txt"))) {
      while (scanner.hasNextLine()) {
        System.out.println(scanner.nextLine());
      }
    } catch (FileNotFoundException e) {
      System.err.println("Dosya bulunamadı: " + e.getMessage());
    }
    Scanner s = new Scanner("3 4,5 elma\nikinci satır");
    System.out.println(s.nextInt() + " | " + s.next() + " | " + s.next() + " | " +
s.nextLine() + " | " + s.nextLine());
  }
}
```

```
Java
Giriş Çıkış
Akışlar
3 | 4,5 | elma |  | ikinci satır
```

Programın sonundaki iki satır `Scanner`'ın bir `String`'i de okuyabildiğini gösterir. `nextInt()` 3'ü, iki `next()` çağrısı boşluğa kadar olan `4,5` ve `elma` kelimelerini alır. Ardından gelen ilk `nextLine()` satırın kalanını, yani boş dizgiyi döndürür; ikinci `nextLine()` asıl ikinci satırı verir. Bu, klavyede görülen `nextInt` – `nextLine` tuzağının aynısıdır.

8.4.2 Kod izleme: try-with-resources ve kapanış sırası

Tanım 8.6

try-with-resources, `try` anahtar sözcüğünden sonraki parantezde açılan kaynakları blok bittiğinde kendiliğinden kapatan `try` biçimidir. Parantezde yalnız `AutoCloseable` arayüzünü uygulayan nesneler açılabilir. Kaynaklar açıldıkları sıranın tersiyle kapatılır; kapatma, blok normal bitse de istisnayla bitse de yapılır.

Kapanış sırasını görmek için açılırken ve kapanırken ileti yazan küçük bir kaynak sınıfı yazılır. `Kaynak`, `AutoCloseable` arayüzünü uygular; tek metodu `close()`'dur.

```
class Kaynak implements AutoCloseable {
  String ad;
  Kaynak(String ad) { this.ad = ad; System.out.println(ad + " açıldı"); }
  public void close() { System.out.println(ad + " kapandı"); }
}
public class KapanisSirasi {
  public static void main(String[] args) {
    try (Kaynak a = new Kaynak("A"); Kaynak b = new Kaynak("B")) {
      System.out.println("blok çalışıyor");
      throw new IllegalStateException("hata!");
    } catch (IllegalStateException e) {
      System.out.println("yakalandı: " + e.getMessage());
    } finally { System.out.println("finally"); }
  }
}
```

```
A açıldı
B açıldı
blok çalışıyor
B kapandı
A kapandı
yakalandı: hata!
finally
```

Tablo 8.8 programın sekiz adımını verir.

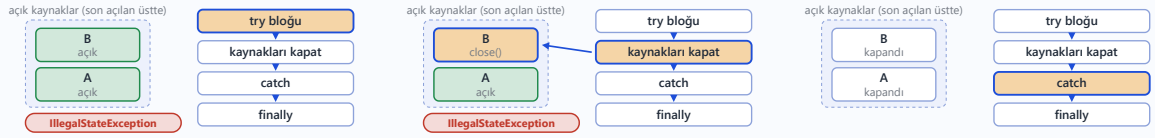
Adım	Satır	Değişkenler	Yeni çıktı
1	8: `try (Kaynak a = new Kaynak("A"); Kaynak b = new Kaynak("B")) {`	a=Kaynak("A")	`A açıldı`
2	8: `try (Kaynak a = new Kaynak("A"); Kaynak b = new Kaynak("B")) {`	a=Kaynak("A"), b=Kaynak("B")	`B açıldı`
3	9: `System.out.println("blok çalışıyor");`	a=Kaynak("A"), b=Kaynak("B")	`blok çalışıyor`
4	10: `throw new IllegalStateException("hata!");`	a=Kaynak("A"), b=Kaynak("B")	—
5	4: `public void close() { System.out.println(ad + " kapandı"); }`	this.ad="B"	`B kapandı`
6	4: `public void close() { System.out.println(ad + " kapandı"); }`	this.ad="A"	`A kapandı`
7	11: `} catch (IllegalStateException e) {` 12: `System.out.println("yakalandı: " + e.getMessage());`	e=IllegalStateException("hata!")	`yakalandı: hata!`
8	13: `} finally { System.out.println("finally"); }`	—	`finally`

Tablo 8.8: try-with-resources ile iki kaynağın açılış ve kapanış sırası. Bloktaki istisna yakalanmadan önce iki kaynak da kapatılır.

İlk iki adımda parantezdeki kaynaklar soldan sağa sırayla oluşturulur: önce `a`, sonra `b`. Üçüncü adımda blok gövdesi çalışır; iki kaynak da açıktır. Dördüncü adımda gövde bir istisna fırlatır.

Beşinci ve altıncı adımlar bu yapının özüdür. İstisna fırlatıldığı hâlde `catch`'e hemen geçilmez. Önce kaynaklar kapatılır ve kapatma **ters sırayla** yapılır: en son açılan `b` ilk kapanır, sonra `a`. Ters sıranın bir gerekçesi vardır. Sonra açılan kaynak öncekine bağlı olabilir; örneğin bir yazıcı bir dosya akışını sarıyor olabilir. Önce dıştakinin kapatılması, içteki akış henüz açıkken dıştakinin tamponunu boşaltabilmesini sağlar.

Yedinci adımda `catch` istisnayı yakalar. Bu noktada `a` ve `b` kapsam dışıdır: parantezde tanımlanan değişkenler yalnız `try` bloğunun içinde görünür. `catch` içinde ya da bloktan sonra `a.close()` gibi bir kullanım derlenmez. Sekizinci adımda `finally` çalışır. Tam sıra şudur: aç A, aç B, gövde, kapat B, kapat A, `catch`, `finally`.



Şekil 8.8: Kapanış sırasından üç an. (a) Gövde istisna fırlatır; iki kaynak da açıktır. (b) Önce son açılan `b` kapanır. (c) İki kaynak da kapandıktan sonra `catch` istisnayı yakalar.

8.4.3 `java.io` kütüphanesi

`java.io` paketi giriş ve çıkış işlemleri için güçlü ve esnek bir araç takımı sunar. Dosya işlemlerinden ağ programlamaya kadar geniş bir alanda kullanılır. Paket üç eksende esneklik sağlar:

- **Giriş ve çıkış:** dosya, ağ ve bellek gibi kaynaklardan veri alır ve onlara veri gönderir.
- **İkili ve metin:** hem bayt tabanlı (ikili) hem karakter tabanlı (metin) işlemleri destekler.
- **Sıralı ve rastgele:** dosyadaki verilere hem sırayla hem de istenen konuma atlayarak erişir.

Paketteki öğeler üç gruba ayrılır (Tablo 8.9).

Öğe tipi	Örnekler	Görevi
Sınıflar	`File`	dosya ve klasör (yol) işlemleri
	`InputStream`, `OutputStream`	temel bayt girişi ve çıkışı
	`Reader`, `Writer`	metin (karakter) girişi ve çıkışı
Arayüzler	`Closeable`, `AutoCloseable`	kaynağı serbest bırakma: `close()`
	`Flushable`	tamponu boşaltma: `flush()`
İstisnalar	`IOException`	giriş çıkış hatalarının üst sınıfı
	`FileNotFoundException`, `EOFException`	dosya yok / beklenmedik dosya sonu

Tablo 8.9: `java.io` paketinin temel öğeleri.

8.4.4 Bayt akışları ve karakter akışları

`java.io`'daki akışlar iki büyük aileye ayrılır. Ayrım, akışın taşıdığı birime göre yapılır.

> **Tanım 8.7**
>
> **Bayt akışları** (*byte streams*) veriyi 8 bitlik baytlar olarak işler; üst sınıfları `InputStream` ve `OutputStream`'dir. **Karakter akışları** (*character streams*) veriyi 16 bitlik Unicode `char` değerleri olarak işler; üst sınıfları `Reader` ve `Writer`'dır. Karakter akışları baytları bir karakter kodlamasıyla karakterlere ve karakterleri baytlara çevirir.

Bayt akışları resim, ses, video ve `.class` gibi ikili verilerle çalışır. `read()` 0–255 arası bir bayt döndürür ve baytın ne anlama geldiğiyle ilgilenmez. Karakter akışları ise metin tabanlı veriler için tasarlanmıştır. Kodlamayı çözdükleri için farklı dilleri ve alfabeleri doğru işlerler. Tablo 8.10 iki aileyi karşılaştırır.

	Bayt akışı	Karakter akışı
Okuma birimi	`byte` (8 bit)	`char` (16 bit)
Üst sınıflar	`InputStream`, `OutputStream`	`Reader`, `Writer`
Dosya sınıfları	`FileInputStream`, `FileOutputStream`	`FileReader`, `FileWriter`
Tamponlu	`BufferedInputStream`, `BufferedOutputStream`	`BufferedReader`, `BufferedWriter`
Kullanım	her tür dosya, kopyalama	yalnız metin dosyaları

Tablo 8.10: Bayt akışları ile karakter akışlarının karşılaştırması.

Metni bayt akışıyla okumak mümkündür ama zordur. Aşağıdaki deneme `Işık` kelimesini bayt bayt okuyup her baytı `(char)` ile karaktere çevirir:

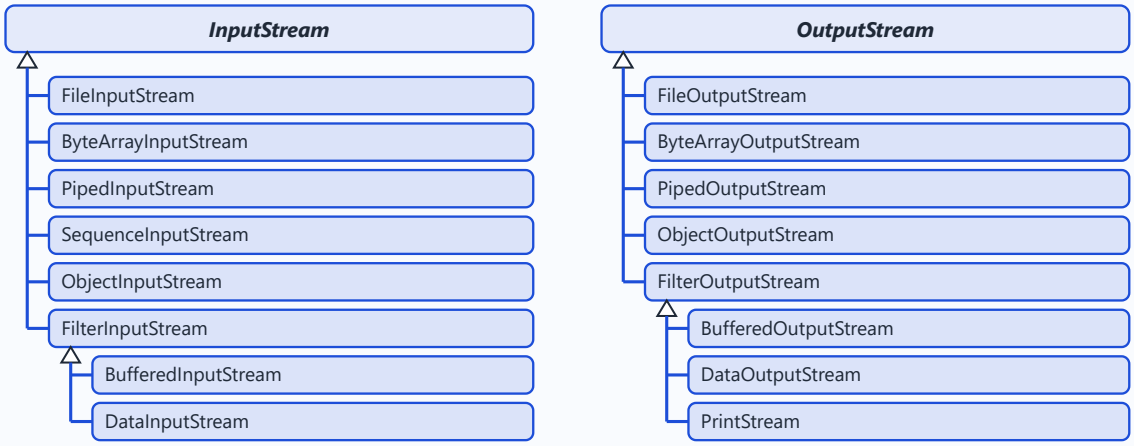
```
IÅ▯Ä±k (6 char)
```

Dört harfli kelime altı “karakter” olarak çıkar. `ş` ve `ı` UTF-8'de ikişer bayttır ve bayt akışı bu baytları ayrı ayrı verir. Her baytın tek başına `char`'a çevrilmesi yanlış harfler üretir. Karakter akışı ise iki baytı birlikte çözer ve doğru harfi verir.

İki aile arasında köprü sınıflar vardır. `InputStreamReader` bir bayt akışını karakter akışına, `OutputStreamWriter` karakterleri bayt akışına çevirir. Kodlama bu köprülerde uygulanır. Bir sayıyı UTF-16 ile yazan `metinYaz` metodu bu köprüyü kullanmıştı.

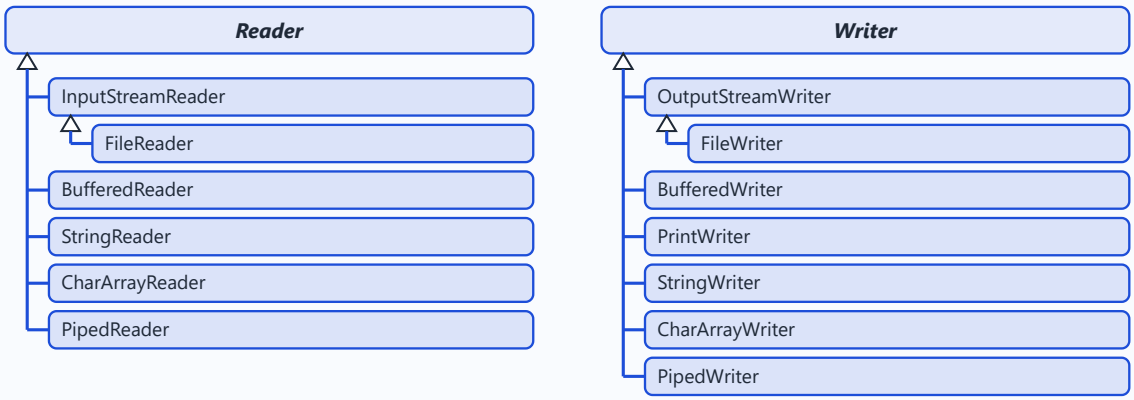
8.4.5 Sınıf hiyerarşileri

İki ailenin her biri bir soyut üst sınıftan türeyen sınıflardan oluşur. Şekil 8.9 bayt akışlarını, Şekil 8.10 karakter akışlarını gösterir.



Şekil 8.9: Bayt akışı sınıfları. İtalik kutular soyut sınıflardır; içi boş üçgenli oklar kalıtımı gösterir. `System.out` bir `PrintStream`'dir.

Hiyerarşinin tepesindeki `InputStream` ve `OutputStream` soyut sınıflardır. Onlardan doğrudan nesne oluşturulmaz. Alt sınıflar iki tür işi üstlenir. Bir kısmı bir **veri kaynağına** bağlanır: `FileInputStream` dosyaya, `ByteArrayInputStream` bellekteki bir diziye, `PipedInputStream` bir boruya bağlanır. Öbür kısmı `FilterInputStream` ve `FilterOutputStream`'den türer. Bunlar kendi başlarına bir kaynağa bağlanmaz; başka bir akışı sarmalayıp ona yeni bir yetenek ekler. `BufferedInputStream` tampon, `DataInputStream` ilkel tip okuma yeteneği ekler.



Şekil 8.10: Karakter akışı sınıfları. `FileReader` bir `InputStreamReader`'dır; `FileWriter` da `OutputStreamWriter`'dan türer.

Karakter akışlarında da aynı düzen vardır. Burada dikkat çeken ilişki `FileReader`'ın yeridir. `FileReader` aslında bir `InputStreamReader`'dır: içeride bir `FileInputStream` açar ve ondan gelen baytları seçilen kodlamayla karakterlere çevirir. `FileWriter` da aynı biçimde `OutputStreamWriter`'dan türer. Yani her karakter akışının altında, sonunda bir bayt akışı vardır. Disk yalnız bayt tanır.

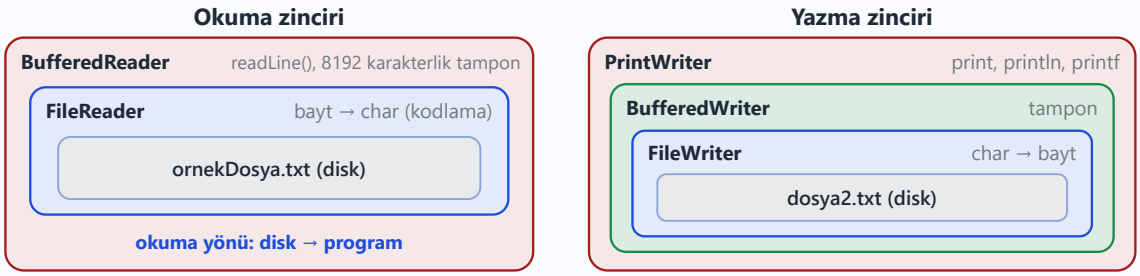
8.4.6 Sarmalayıcı akışlar

Java G/Ç kütüphanesindeki her sınıf tek bir işi iyi yapar. `FileReader` dosyadan karakter okur ama satır tanımaz. `BufferedReader` satır okur ama bir dosyaya bağlanamaz. İhtiyaç duyulan yetenekler, akışlar iç içe sarmalanarak birleştirilir.

> **Tanım 8.8**
>
> **Sarmalayıcı akış** (*decorator*, *wrapper stream*), yapıcısında başka bir akışı alan ve ona tampon, satır okuma ya da ilkel tip okuma gibi yeni bir yetenek ekleyen akıştır. Sarmalayıcının metotları işin bir kısmını kendisi yapar, kalanını içteki akışa iletir. Bu düzenin adı Dekoratör (*Decorator*) tasarım desenidir.

```
BufferedReader okuyucu = new BufferedReader(new FileReader("ornekDosya.txt"));
PrintWriter yazici = new PrintWriter(new BufferedWriter(new FileWriter("dosya2.txt")));
```

Birinci satırda iki katman vardır: `FileReader` dosyayı açar ve baytları karakterlere çevirir; `BufferedReader` onu sarar, tampon ve `readLine()` ekler. İkinci satırda üç katman vardır: `FileWriter` karakterleri baytlara çevirip dosyaya yazar, `BufferedWriter` tampon ekler, `PrintWriter` ise `print`, `println` ve `printf` gibi rahat yazma metotlarını ekler (Şekil 8.11).



Şekil 8.11: Okuma ve yazma zincirleri. Veri dıştaki akıştan içtekine ya da içtekinden dıştakine katman katman geçer.

Bunu bir bahçe hortumuna benzetmek mümkündür: hortumun ucuna önce bir filtre, sonra bir püskürtme başlığı takılır. Su, yani veri, hepsinden sırayla geçer. Her parça kendi işini yapar ve parçalar istenen sırayla birleştirilebilir. Bir yeteneği eklemek için yeni bir sınıf yazmak gerekmez; var olan sarmalayıcılar birleştirilir.

Sarmalamanın iki kuralı vardır. Birincisi, **en dıştaki akışı kapatmak içtekilerin hepsini kapatır**. `yazici.close()` önce kendi tamponunu, sonra `BufferedWriter`'ın tamponunu boşaltır, en sonunda `FileWriter`'ı ve dosyayı kapatır. İçteki akışlar ayrıca kapatılmaz. İkincisi, **sarmalayıcı veriyi yorum-**

lar ama doğrulamaz. Bir metin dosyası `DataInputStream` ile sarılıp `readInt()` çağrılırsa ilk dört bayt, yani `J`, `a`, `v`, `a` harflerinin kodları, 1.247.901.281 tam sayısı olarak okunur. Akış hata vermez; yanlış sarmalayıcı yanlış ama geçerli görünen bir sonuç üretir.

8.4.7 Kod izleme: `BufferedReader`

Metin dosyasını satır satır okumanın en yaygın yolu `BufferedReader`'ın `readLine()` metodudur. Metot bir sonraki satırı satır sonu olmadan döndürür; dosya bittiğinde `null` döndürür.

```
import java.io.*;

public class SatirOku {
  public static void main(String[] args) {
    File dosya = new File("ornekDosya.txt");
    try (BufferedReader okuyucu =
        new BufferedReader(new FileReader(dosya))) {
      String satir;
      while ((satir = okuyucu.readLine()) != null) {
        System.out.println(satir);
      }
    } catch (IOException e) {
      System.err.println("Okuma hatası: " + e.getMessage());
      e.printStackTrace();
    }
  }
}
```

Döngü koşulundaki `(satir = okuyucu.readLine()) != null` ifadesi Java'nın G/Ç kodunda sık görülen bir kalıptır. Önce parantez içindeki atama yapılır: okunan satır `satir` değişkenine konur. Atama ifadesinin değeri, atanan değerin kendisidir. Bu değer `null` ile karşılaştırılır. Böylece okuma ve dosya sonu denetimi tek satırda yapılır. Dış parantezler şarttır. Onlar olmadan `!=` önce çalışır ve bir `boolean` değer `String` değişkenine atanmaya çalışılır; kod derlenmez. Dosyada `Java`, `Giriş Çıkış` ve `Akışlar` satırları varken izleme Tablo 8.11'deki gibidir.

Adım	Satır	Değişkenler	Yeni çıktı
1	5: `File dosya = new File("ornekDosya.txt");`	dosya="ornekDosya.txt"	—
2	6: `try (BufferedReader okuyucu =` 7: `new BufferedReader(new FileReader(dosya))) {`	okuyucu=BufferedReader	—
3	9: `while ((satir = okuyucu.readLine()) != null) {`	satir="Java"	—
4	10: `System.out.println(satir);`	satir="Java"	`Java`
5	9: `while ((satir = okuyucu.readLine()) != null) {`	satir="Giriş Çıkış"	—
6	10: `System.out.println(satir);`	satir="Giriş Çıkış"	`Giriş Çıkış`
7	9: `while ((satir = okuyucu.readLine()) != null) {`	satir="Akışlar"	—
8	10: `System.out.println(satir);`	satir="Akışlar"	`Akışlar`

Adım	Satır	Değişkenler	Yeni çıktı
9	9: `while ((satir = okuyucu.readLine()) != null) {`	satir=null	—
10	12: `} catch (IOException e) {`	—	—

Tablo 8.11: `SatirOku` programının izlemesi. Döngü koşulu (satır 9) her turda bir satır okur; dosya sonunda `null` alır.

Birinci adımda yalnız bir `File` nesnesi oluşur; dosya henüz açılmamıştır. İkinci adımda `FileReader` dosyayı açar ve `BufferedReader` onu sarmalar. Okuma konumu dosyanın başındadır. Üçüncü adımda ilk `readLine()` çağrısı `"Java"`'yı döndürür. Bu sırada `BufferedReader` dosyadan tek seferde büyük bir parça okuyup tamponuna koymuştur; küçük bir dosyada tamponun içine dosyanın tamamı girer. Sonraki satırlar diske gitmeden tampondan verilir.



Şekil 8.12: `readLine()` izlemesinden iki an. (a) İlk satır okunur, satır sonu atılır. (b) Dosya sonunda `readLine()` `null` döndürür ve döngü biter.

Dokuzuncu adımda dosya biter ve `readLine()` `null` döndürür. Döngü sona erer. Onuncu adımda `try` bloğu biter; okuyucu kendiliğinden kapanır. Hata olmadığı için `catch` çalışmaz. Okunan her satır bir kez yazıldığı için programın süresi dosyanın boyutuyla doğru orantılıdır. Bellekte ise aynı anda yalnız bir satır ve tampon durur; dosya ne kadar büyük olursa olsun bellek kullanımı sabittir. Bu özellik, gigabaytlarca büyüklükteki kayıt dosyalarının bile satır satır işlenebilmesini sağlar.

Sık yapılan hata

Döngü içinde `readLine()`'ı iki kez çağırmak: `while (okuyucu.readLine() != null) System.out.println(okuyucu.readLine());`. Koşuldaki çağrı bir satırı okuyup atar, gövdedeki çağrı sonrakini yazar. Satırların yarısı kaybolur ve tek sayıda satırlı dosyanın sonunda `null` yazdırılır. Her satır bir kez okunur ve bir değişkende tutulur.

8.4.8 `PrintWriter`

`PrintWriter` metin dosyalarına yazmanın en rahat yoludur. `System.out`'tan tanıdık olan `print`, `println` ve `printf` metotlarının hepsi vardır. Sayı, karakter ve `boolean` gibi öteki tipleri kendisi metne çevirerek yazar.

```
public class PrintWriterOrnek {
  public static void main(String[] args) {
    PrintWriter yazici = null;
    try {
      yazici = new PrintWriter("yeniDosya.txt");
      yazici.println("Merhaba, dünya!");
      yazici.println("Bu bir örnek dosyadır.");
      yazici.printf(Locale.ROOT, "Pi yaklaşık %.2f%n", Math.PI);
      yazici.println(42);
    } catch (IOException e) {
      System.err.println("Dosya hatası: " + e.getMessage());
      e.printStackTrace();
    } finally {
      if (yazici != null)
        yazici.close();
    }
  }
}
```

Program çalıştıktan sonra `yeniDosya.txt`'nin içeriği şudur:

```
Merhaba, dünya!
Bu bir örnek dosyadır.
Pi yaklaşık 3.14
42
```

`new PrintWriter("yeniDosya.txt")` dosya yoksa onu oluşturur. Dosya varsa **içini siler** ve baştan yazar. Dosya oluşturulamazsa yapıcı `FileNotFoundException` fırlatır; program bunu `IOException` olarak yakalar. `printf`'teki `%.2f` sayıyı iki ondalık basamakla, `%n` ise işletim sisteminin satır sonuyla yazar. İlk argüman olan `Locale.ROOT` (`java.util.Locale`) biçimlendirmenin yerel ayardan bağımsız olmasını sağlar.

`Locale.ROOT`'un neden verildiği şöyle açıklanır. `printf` yerel ayar verilmeden çağrılırsa ondalık ayırıcıyı bilgisayarın **yerel ayarına** (*locale*) göre seçer. Türkçe bir sistemde `3,14`, İngilizce bir sistemde `3.14` yazılır. Aynı program iki makinede farklı dosya üretir. Dosyayı daha sonra bir program okuyacaksa bu fark sorun çıkarır: `Double.parseDouble("3,14")` hata verir. `Locale.ROOT` ile çıktı her makinede `3.14`'tür. `println(3.5)` ise zaten yerel ayardan bağımsızdır ve her zaman noktayla yazar. Aşağıdaki deneme Türkçe yerel ayarda çalıştırılmıştır:

```
import java.io.*;
import java.util.Locale;
public class Locale25 {
  public static void main(String[] args) throws Exception {
    try (PrintWriter p = new PrintWriter("loc.txt")) {
      p.printf("Pi yaklaşık %.2f%n", Math.PI);
      p.printf(Locale.ROOT, "Pi yaklaşık %.2f%n", Math.PI);
      p.println(3.5);
      p.println('A');
      p.println(true);
    }
    Goster.main(new String[] {"loc.txt"});
  }
}
```

```
Pi yaklaşık 3,14
Pi yaklaşık 3.14
3.5
A
true
[54 bayt]
```

`PrintWriter`'ın önemli bir özelliği daha vardır: yazma metotları istisna fırlatmaz. Yazma sırasında bir G/Ç hatası olursa `PrintWriter` bunu bir iç bayrağa kaydeder ve sessizce devam eder. Hata ancak `checkError()` çağrılarak görülür. Tamponlama kesimindeki izleme bu davranışı gösterir. Son olarak `close()` unutulmamalıdır: `PrintWriter` içinde bir tampon vardır ve kapatılmazsa son yazılanlar diske gitmeyebilir.

8.4.9 `FileWriter`: yazma ve ekleme

`FileWriter` dosyalara karakter yazmak için kullanılan temel sınıftır; `Writer` sınıfının yazma işlevini bir dosyaya bağlar. Yalnız `write(String)`, `write(char[])` ve `write(int c)` metotları vardır. Metin biçimlendirme yeteneği sınırlıdır: satır sonu (`\n`) ayrıca yazılır, sayılar önce dizgiye çevrilir. Daha düşük düzeyli ve esnek bir yaklaşımdır; bu yüzden çoğunlukla `BufferedWriter` ya da `PrintWriter` ile sarmalanır.

`write(int c)` sık karıştırılır. Bu metot sayıyı değil, kodu o sayı olan **karakteri** yazar. `w.write(65)` dosyaya `A` yazar; `65` yazmak için `w.write(String.valueOf(65))` gerekir. Aşağıdaki programın son çıktı satırındaki `A65` bu iki çağrının sonucudur.

`FileWriter`'ın en kullanışlı yeteneği **ekleme** (*append*) kipidir. `new FileWriter(ad, true)` dosyayı silmeden sonuna ekler. `false` verilirse ya da ikinci parametre hiç verilmezse dosya sıfırlanır.

```
public class FileWriterOrnek {
  public static void main(String[] args) {
    try (FileWriter yazici = new FileWriter("yeniDosya.txt")) {
      yazici.write("Merhaba, dünya!\n");
      yazici.write("Bu bir FileWriter örneğidir.\n");
      System.out.println("Dosya oluşturuldu ve veri yazıldı.");
    } catch (IOException e) {
      System.err.println("Dosya hatası: " + e.getMessage());
      e.printStackTrace();
    }
    // true: dosyanın sonuna ekle (append)
    try (FileWriter ekle = new FileWriter("yeniDosya.txt", true)) {
      ekle.write("Sona eklenen satır.\n");
    } catch (IOException e) {
      System.err.println("Dosya hatası: " + e.getMessage());
    }
  }
}
```

```
Dosya oluşturuldu ve veri yazıldı.
```

İlk `try` bloğu dosyayı oluşturur ve iki satır yazar. İkinci blok dosyayı `true` ile açtığı için mevcut iki satırı korur ve üçüncü satırı sona ekler. Program bittiğinde `yeniDosya.txt` şudur:

```
Merhaba, dünya!
Bu bir FileWriter örneğidir.
Sona eklenen satır.
```

```
Merhaba, dünya!
Bu bir FileWriter örneğidir.
Sona eklenen satır.
[69 bayt]
A65
```

Dosya 69 bayttır: üç satırdaki 62 karakter ve üç satır sonu 65 eder. `ü`, `ö`, `ğ` ve `ı` UTF-8'de ikişer bayt olduğu için dört bayt daha eklenir. Ekleme kipi kayıt dosyalarında vazgeçilmezdir: program her çalıştığında yeni kayıtlar eskilerin sonuna eklenir. İkinci `try` bloğunda `true` unutulsaydı ilk iki satır silinir ve dosyada yalnız “Sona eklenen satır.” kalırdı. Tablo 8.12 üç yazıcı sınıfını karşılaştırır.

	`FileWriter`	`BufferedWriter`	`PrintWriter`
Yazma metotları	`write`	`write`, `newLine()`	`print`, `println`, `printf`
Sayı yazmak	önce `String`'e çevrilir	önce `String`'e çevrilir	doğrudan: `println(42)`

	`FileWriter`	`BufferedWriter`	`PrintWriter`
Tampon	küçük iç tampon	8192 karakterlik tampon	sardığı akışa bağlı
Hata bildirimi	`IOException`	`IOException`	`checkError()`

Tablo 8.12: `FileWriter`, `BufferedWriter` ve `PrintWriter`.

Seçim yapılacak işe göre değişir. Biçimli rapor ya da tablo yazılacaksa `PrintWriter`, çok sayıda kısa yazma yapılacaksa araya `BufferedWriter` konur. Hataların istisna olarak hemen bildirilmesi isteniyorsa `BufferedWriter` tercih edilir; `PrintWriter`'da hata ancak `checkError()` sorulunca görülür.

8.4.10 Karakter kodlamasını belirtmek

Kodlama verilmeyen karakter akışları varsayılan kodlamayı kullanır. Java 18 ve sonrasında bu UTF-8'dir. Daha eski sürümlerde ise işletim sisteminin ayarıydı; Türkçe Windows'ta Windows-1254. Aynı dosya iki farklı makinede farklı okunabiliyordu. Taşınabilir kodda kodlama açıkça verilir. Java 11'den beri `FileReader` ve `FileWriter`'ın kodlama alan yapıcıları vardır. Java 10 ve öncesinde aynı iş `InputStreamReader` ve `OutputStreamWriter` köprüleriyle yapılırdı.

```
public class KodlamaOku {
  public static void main(String[] args) throws IOException {
    File dosya = new File("turkce.txt");
    try (BufferedWriter yazici = new BufferedWriter(
        new FileWriter(dosya, StandardCharsets.UTF_8))) {    // Java 11+
      yazici.write("Işık, şimşek, çiğ, öğrenci, Ünye");
    }
    try (BufferedReader dogru = new BufferedReader(
        new FileReader(dosya, StandardCharsets.UTF_8))) {
      System.out.println(dogru.readLine());
    }
    try (BufferedReader yanlis = new BufferedReader(
        new FileReader(dosya, StandardCharsets.ISO_8859_1))) {
      System.out.println(yanlis.readLine());
    }
  }
}
```

```
Işık, şimşek, çiğ, öğrenci, Ünye
IÅÄ±k, Åimåek, Ã§iÄ, Ã¶Ärenci, Ãnye
```

Dosya UTF-8 ile yazılıp UTF-8 ile okunduğunda metin doğru gelir. Aynı baytlar ISO-8859-1 ile okunduğunda her Türkçe harf iki ayrı karaktere dönüşür. Örneğin `ş`'nin UTF-8 baytları `C5 9F`'dur. Latin-1'de `C5` `Å` harfidir. `9F` ise yazdırılamayan bir kontrol karakteridir ve ekranda görünmez. Bu yüzden `şimşek`, `ÅimÅek` olarak görünür. Bu tür bozulmaya Türkçede “karakter bozulması”, programcı jargonunda *mojibake* denir. Kodlama hatası istisna üretmez; program çalışır, ama veri bozuk gelir.

Karşılığı olmayan karakter yazılırken de bilgi kaybolur. ASCII ile yazan bir akışa `$` verilirse bu harfin ASCII'de karşılığı olmadığı için soru işareti (`?` , `3F`) yazılır. Aynı harf UTF-8'de `C5 9F` olarak saklanırdı. Bu kayıp geri alınamaz: dosyada artık yalnız bir soru işareti vardır.

Sık yapılan hata

Dosyayı bir kodlamayla yazıp başka bir kodlamayla okumak. Eski bir Windows programının Windows-1254 ile ürettiği dosya Java 18 ve sonrasında kodlama verilmeden okunursa Türkçe harfler `?` işaretine dönüşür. Dosyanın kodlaması biliniyorsa okurken açıkça verilir: `new FileReader(dosya, Charset.forName("windows-1254"))` .

8.5 Tamponlama

Akışların temel metotları küçük parçalarla çalışır. `read()` bir bayt ya da bir bayt dizisi okur, `write()` bir bayt ya da bir bayt dizisi yazar. Her bayt için diske ayrı ayrı gidilirse uygulama ciddi biçimde yavaşlar. Bu kesim sorunun nedenini ve çözümünü anlatır.

8.5.1 Neden tampon?

Disk, bellekten binlerce kat yavaştır. Üstelik her disk erişimi yalnız aktarım süresinden ibaret değildir. Program işletim sistemine bir **sistem çağrısı** yapar, işletim sistemi aygıt sürücüsünü çalıştırır, sonuç programa geri döner. Bu sabit maliyet, bir bayt okunsa da sekiz bin bayt okunsa da aşağı yukarı aynıdır. Dolayısıyla bin baytı bin ayrı çağrıyla okumak, bir çağrıyla okumaktan çok daha pahalıdır.

Tanım 8.9

Tampon (*buffer*), verinin bellekte geçici olarak biriktirildiği alandır. **Tamponlama** (*buffering*), okuma ve yazmayı bu alan üzerinden toplu yaparak fiziksel disk erişimlerinin sayısını azaltma tekniğidir. Okurken tampon bir seferde büyük bir parçayla doldurulur ve program küçük okumalarını tampondan karşılar. Yazarken küçük yazmalar tamponda birikir ve tampon dolunca tek seferde diske gönderilir.

Gündelik bir benzetme farkı açıklar. Marketten on ürün almak için her ürüne ayrı bir yolculuk yapmak yerine hepsi bir sepete konur ve tek seferde taşınır. Yolculuğun kendisi, yani sabit maliyet, bir kez ödenir. Şekil 8.13 tamponsuz ve tamponlu yazmayı karşılaştırır.



Şekil 8.13: Tamponsuz ve tamponlu yazma. (a) Tamponsuz akışta her `write()` doğrudan diske gider. (b) Tamponlu akışta `write()` baytları bellekteki tampona koyar. (c) Tampon dolunca ya da `flush()` veya `close()` çağrılınca tamponun tamamı tek seferde diske yazılır.

8.5.2 Tamponlu akış sınıfları ve `flush`

Java'da tamponlama, sarmalayıcı akışlarla eklenir. Dört tamponlu sınıf vardır (Tablo 8.13).

Sınıf	Sardığı akış	Ne yapar?
`BufferedInputStream`	`InputStream`	Okumayı hızlandırır: tampon boşalınca diskten tek seferde 8192 bayt okur.
`BufferedReader`	`Reader`	Aynı işi karakterler için yapar; ayrıca `readLine()` ekler.
`BufferedOutputStream`	`OutputStream`	Yazmayı hızlandırır: `write()` verisini tamponda biriktirir.
`BufferedWriter`	`Writer`	Aynı işi karakterler için yapar; ayrıca `newLine()` ekler.

Tablo 8.13: Tamponlu akış sınıfları.

Yazma tamponunun bir yan etkisi vardır: `write()` döndüğünde veri henüz diskte olmayabilir. Verinin diske gittiği üç an vardır. Tampon dolduğunda kendiliğinden boşaltılır. `flush()` çağrıldığında tampon hemen diske boşaltılır. `close()` çağrıldığında önce `flush()` yapılır, sonra akış kapatılır. Aşağıdaki program bu davranışı dosya boyutunu ölçerek gösterir:

```
public class FlushOrnek {
  public static void main(String[] args) throws IOException {
    File log = new File("log.txt");
    BufferedWriter bw = new BufferedWriter(new FileWriter(log));
    bw.write("Kayıt 1");
    System.out.println("önce: " + log.length());
    bw.flush();
    System.out.println("sonra: " + log.length());
    bw.close();
  }
}
```

```
önce: 0
sonra: 8
```

`bw.write("Kayıt 1")` döndükten sonra dosya hâlâ 0 bayttır; yedi karakter `BufferedWriter`'ın tamponunda bekler. `flush()`'tan sonra dosya 8 bayt olur. Yedi karakter sekiz bayt eder, çünkü `ı` UTF-8'de iki bayttır.

> **Sık yapılan hata**
>
> Tamponlu bir yazıcıyı kapatmamak. `bw.write("kayit");` yazılıp akış kapatılmazsa veri tamponda kalır. Program bittiğinde de tampon boşaltılmaz: JVM kapanırken açık akışların tamponlarını diske yazmaz. Dosya boş görünür. `close()` ya da `flush()` çağrılır; en güvenli yol try-with-resources'tır.

Bu hata gerçekten ölçülebilir. Aşağıdaki iki program arka arkaya çalıştırılmıştır. Birincisi yazıp kapatmadan biter, ikincisi dosyanın boyutuna bakar:

```
not.txt boyutu: 0
```

```
program bitti, not.txt boyutu: 0
```

Program sona erdikten sonra bile dosya 0 bayttır. Beş karakter hiçbir zaman diske ulaşmamıştır.

8.5.3 Kod izleme: tampon ve `flush`

Aşağıdaki kod parçası bir `PrintWriter`'ın altındaki tamponu adım adım izler. Her yazmadan sonra dosyanın boyutu ölçülür. Son iki satır kapatılmış bir akışa yazmanın ne olduğunu gösterir.

```
File f = new File("log.txt");
PrintWriter pw = new PrintWriter(new FileWriter(f));
System.out.println("0: " + f.length());
pw.print("kayit1;");
System.out.println("1: " + f.length());
pw.print("kayit2;");
System.out.println("2: " + f.length());
pw.flush();
System.out.println("3: " + f.length());
pw.print("kayit3;");
pw.close();
System.out.println("4: " + f.length());
pw.print("kayit4;");   // kapalı: sessizce yok sayılır
System.out.println("hata? " + pw.checkError());
```

```
0: 0
1: 0
2: 0
3: 14
4: 21
hata? true
```

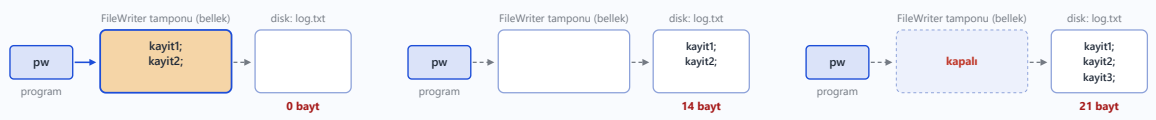
Adım	Satır	Değişkenler	Yeni çıktı
1	1: `File f = new File("log.txt");` 2: `PrintWriter pw = new PrintWriter(new FileWriter(f));`	f=log.txt, pw=PrintWriter	—
2	3: `System.out.println("0: " + f.length());`	f.length()=0	`0: 0`
3	4: `pw.print("kayit1;");`	tampon="kayit1;"	—
4	5: `System.out.println("1: " + f.length());`	f.length()=0	`1: 0`
5	6: `pw.print("kayit2;");` 7: `System.out.println("2: " + f.length());`	tampon="kayit1;kayit2;"	`2: 0`
6	8: `pw.flush();`	tampon=""	—
7	9: `System.out.println("3: " + f.length());`	f.length()=14	`3: 14`
8	10: `pw.print("kayit3;");`	tampon="kayit3;"	—
9	11: `pw.close();`	pw=kapalı	—
10	12: `System.out.println("4: " + f.length());`	f.length()=21	`4: 21`

11	13: `pw.print("kayit4;"); // kapalı: sessizce yok sayılır`
	14: `System.out.println("hata? " + pw.checkError());`
		pw.checkError()=true	`hata? true`

Tablo 8.14: Tampon ve `flush` izlemesi. “Değişkenler” sütunu tamponun içeriğini ve ölçülen dosya boyutunu verir.

Tablo 8.14'ün ilk adımında `FileWriter` dosyayı oluşturur; dosya boştur. Bu zincirde tampon `FileWriter`'ın içindedir. `FileWriter` karakterleri baytlara çeviren bir kodlayıcı kullanır ve bu kodlayıcının bir bayt tamponu vardır. `PrintWriter` kendi başına tampon eklemez; yazdıklarını bu tampona iletir.

Üçüncü ve beşinci adımlarda `print` çağrıları yedişer karakteri tampona koyar ama diske dokunmaz. Dosya 0 bayt kalır. Altıncı adımda `flush()` tamponu diske boşaltır ve 14 bayt yazılır. Sekizinci adımda üçüncü kayıt tampona girer. Dokuzuncu adımdaki `close()` önce `flush()` yapar, sonra dosyayı kapatır. Onuncu adımda dosya 21 bayttır. `close()` unutulsaydı üçüncü kayıt kaybolabilirdi.



Şekil 8.14: Tamponun üç anı. (a) İki kayıt tamponda bekliyor; dosya 0 bayt. (b) `flush()`'tan sonra dosya 14 bayt. (c) `close()`'tan sonra dosya 21 bayt.

On birinci adım `PrintWriter`'ın istisna fırlatmama özelliğini gösterir. Kapatılmış bir akışa `print` yapmak normalde bir `IOException` gerektirir. `PrintWriter` ise hata vermez; yazma sessizce başarısız olur ve iç bayrak kurulur. `checkError()` `true` döndürür. Bu davranış `System.out` için uygundur, çünkü her `println`'i `try` içine yazmak zahmetli olurdu. Önemli bir dosyaya yazılırken ise sonda `checkError()` denetlenmeli ya da istisna fırlatan bir yazıcı seçilmelidir.

8.5.4 Tamponlu okuma

Okuma tarafında tamponlama iki biçimde kullanılır: bayt bayt ve satır satır. `dosya.txt` iki satır içersin: `Java I/O` ve `Tampon`.

```
try {
  FileInputStream fis = new FileInputStream("dosya.txt");
  BufferedInputStream bis = new BufferedInputStream(fis);
  int veri;
  while ((veri = bis.read()) != -1) {
    System.out.print((char) veri);
  }
  bis.close();     // içteki fis'i de kapatır
} catch (IOException e) {
  System.err.println("Dosya hatası: " + e.getMessage());
}
```

Çıktı: bayt bayt okuma

```
Java I/O
Tampon
```

Bu döngü `bis.read()`'i her bayt için bir kez çağırır. Ama diskten her seferinde bir bayt okunmaz. İlk çağrıda tampon boştur; `BufferedInputStream` içteki `FileInputStream`'den tek seferde 8192 bayta kadar okur. Sonraki çağrılar tampondan karşılanır ve diske gidilmez. Satır sonu da bir bayttır ve `(char)` ile yazdırıldığında satırı bitirir. Bu yüzden çıktı iki satırdır. Bu yöntem yalnız ASCII metinde doğru sonuç verir; Türkçe harfler ikişer bayt olduğu için bozulur.

```
try {
  FileReader fr = new FileReader("dosya.txt");
  BufferedReader br = new BufferedReader(fr);
  String satir;
  while ((satir = br.readLine()) != null) {
    System.out.println(satir);
  }
  br.close();     // içteki fr'yi de kapatır
} catch (IOException e) {
  System.err.println("Dosya hatası: " + e.getMessage());
}
```

İkinci parça aynı dosyayı `BufferedReader` ile satır satır okur ve aynı iki satırı yazar. Karakter akışı kodlamayı çözdüğü için Türkçe harfler de doğru gelir. Her iki parçada da `close()` yalnız en dıştaki akışa uygulanır: `bis.close()` içteki `fis`'i, `br.close()` içteki `fr`'yi kapatır.

Bu iki parça bir zayıflık taşır. Okuma sırasında bir istisna olursa `close()` satırına ulaşılmaz ve dosya açık kalır. Bölümün geri kalanındaki programların çoğu bu yüzden try-with-resources kullanır.

8.5.5 Tamponlama ne kadar hızlandırır?

Aşağıdaki program 2 MB'lık bir dosya oluşturur ve onu iki kez bayt bayt okur: önce tamponsuz, sonra tamponlu. Süre `System.nanoTime()` ile milisaniye cinsinden ölçülür.

```
public class TamponOlcum {
  static long oku(InputStream in) throws IOException {
    long bas = System.nanoTime();
    while (in.read() != -1) { }          // her seferinde 1 byte
    in.close();
    return (System.nanoTime() - bas) / 1_000_000;
  }
  public static void main(String[] args) throws IOException {
    File f = new File("buyuk.dat");
    try (FileOutputStream fos = new FileOutputStream(f)) {
      fos.write(new byte[2 * 1024 * 1024]);   // 2 MB
    }
    long t1 = oku(new FileInputStream(f));
    long t2 = oku(new BufferedInputStream(new FileInputStream(f)));
    System.out.println("Tamponsuz: " + t1 + " ms");
    System.out.println("Tamponlu : " + t2 + " ms");
    f.delete();
  }
}
```

Sunumdaki örnek çalıştırmanın çıktısı şudur:

```
Tamponsuz: 3151 ms
Tamponlu : 94 ms
```

Bu not hazırlanırken aynı program üç kez daha çalıştırıldı. Süreler 3194/58, 3241/62 ve 3250/63 milisaniye oldu. Süreler makineye, diske ve işletim sisteminin önbelleğine göre değişir; oran ise benzer kalır. Bu makinede tamponlu okuma elli kattan fazla hızlıdır.

Farkın nedeni çağrı sayısındadır. Dosya 2 MB, yani $2 \cdot 1024 \cdot 1024 = 2.097.152$ bayttır. Tamponsuz akışta her `read()` bir sistem çağrısıdır: 2.097.152 çağrı, üstüne dosya sonunu bildiren bir çağrı daha. Tamponlu akışta sistem çağrısı 8192 baytta birdir: 2.097.152 / 8192 = 256 veri taşıyan çağrı ve dosya sonu için bir çağrı. Programın kendi `read()` çağrılarının sayısı iki durumda da aynıdır; ama tamponlu durumda bunlar bellekteki bir diziden okunan sıradan metot çağrılarıdır. Sistem çağrısı sayısı yaklaşık 8000 kat azalır. Süre ise yalnız elli kat kadar azalır, çünkü her bayt için bir metot çağrısı hâlâ yapılır. Blok blok okuma bu maliyeti de ortadan kaldırır.

8.6 Diğer bayt akışları

`InputStream` ve `OutputStream` ailesinde dosya akışlarından başka sınıflar da vardır. Bunlar veri kaynağı olarak bellekteki bir diziyi, başka bir iş parçacığını ya da birden çok akışı kullanır. Hepsi aynı `read()` ve `write()` metotlarını taşıdığı için bir dosya akışının beklendiği her yerde kullanılabilirler.

8.6.1 `FileInputStream` ve parça parça okuma

`FileInputStream` `InputStream`'den türer ve dosyadan doğrudan bayt okur. Düşük düzeyli bir sınıftır; resim ve ses gibi metin dışı veriler için uygundur. İki okuma metodu vardır:

- `read()` tek bir bayt okur ve 0–255 arası bir `int` döndürür. Dosya sonunda −1 döndürür.
- `read(byte[] b)` en fazla `b.length` bayt okuyup diziye koyar ve **okuduğu bayt sayısını** döndürür. Dosya sonunda −1 döndürür. Dönen sayı dizinin boyundan küçük olabilir.

```
public class ParcaOku {
  public static void main(String[] args) throws IOException {
    try (FileInputStream fis = new FileInputStream("dosya.txt")) {
      byte[] dizi = new byte[4];
      int adet;
      while ((adet = fis.read(dizi)) != -1) {
        System.out.println(adet + ": " + new String(dizi, 0, adet));
      }
    }
  }
}
```

`dosya.txt`'de `Merhaba Java` yazsın: 12 bayt. Dizi 4 baytlık olduğu için dosya üç parçada okunur:

```
4: Merh
4: aba
4: Java
```

İkinci satırdaki `aba `'nın sonunda bir boşluk vardır. `new String(dizi, 0, adet)` dizinin yalnız okunan kısmını, yani 0'dan başlayarak `adet` baytı metne çevirir. Bu örnekte dosya tam 4'ün katı olduğu için her parça doludur. Dosya 13 bayt olsaydı dördüncü okuma 1 döndürürdü ve dizinin son üç hücresinde önceki parçadan kalan baytlar dururdu. `new String(dizi)` yazılsaydı bu eski baytlar da metne katılırdı. Çalışma sorularındaki `JAVA-IO` örneği bu hatayı adım adım gösterir.

Metin için `FileInputStream` kullanmak zordur. Türkçe harfler gibi çok baytlı karakterler parça sınırında bölünebilir. `Gündüz` kelimesi 2 baytlık parçalarla okunup her parça ayrı ayrı UTF-8 olarak çözülürse `ü`'nün iki baytı iki ayrı parçaya düşer:

```
[G�][�n][d�][�z]
```

Yarım kalan her bayt geçersiz bir UTF-8 dizisidir ve `�` işaretine dönüşür. Metin dosyaları için `FileReader` gibi karakter tabanlı sınıflar daha uygundur; onlar parça sınırındaki yarım karakteri bir sonraki okumaya saklar.

8.6.2 Bellekteki akışlar

`ByteArrayInputStream` de `InputStream`'den türer, ama bir dosyadan değil, bellekteki bir bayt dizisinden okur. Fiziksel bir kaynağa erişmediği için hızlıdır. Yalnız okunabilir; belleğe yazmak için eşi `ByteArrayOutputStream` kullanılır. `ByteArrayOutputStream` yazılan baytları büyüyen bir dizide toplar; `toByteArray()` bu diziyi, `toString()` onun metin karşılığını verir.

```
public class ByteDiziOku {
  public static void main(String[] args) throws IOException {
    byte[] dizi = {74, 97, 118, 97};           // "Java"
    ByteArrayInputStream bais = new ByteArrayInputStream(dizi);
    int b;
    while ((b = bais.read()) != -1) {
      System.out.print(b + "=" + (char) b + " ");
    }
    System.out.println();
    // Belleğe yazma: ByteArrayOutputStream
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    baos.write(dizi);
    baos.write('!');
    byte[] sonuc = baos.toByteArray();
    System.out.println(sonuc.length + " byte: " + baos);
  }
}
```

```
74=J 97=a 118=v 97=a
5 byte: Java!
```

İlk döngü dizinin dört baytını okur. Her baytı sayı ve karakter olarak yazar: 74 = `J`, 97 = `a`, 118 = `v`. İkinci kısımda dört bayt ve `'!'` karakteri belleğe yazılır. Sonuç 5 baytlık bir dizidir.

Bu sınıflar iki durumda işe yarar. Birincisi, akış bekleyen bir metodu dosyasız denemektir: bir sınama, bir `ByteArrayInputStream` ile istenen girdiyi verip metodun çıktısını bir `ByteArrayOutputStream`'de toplayabilir. İkincisi, verinin tamamı bellekte hazırlanıp sonra tek seferde gönderilecekse, örneğin ağ üzerinden. Bütün veri bellekte durduğu için çok büyük verilerde bellek tüketimi sorun olur.

8.6.3 Kullanımdan kaldırılmış bir sınıf

`java.io.StringBufferInputStream` bir dizgiden bayt okumak için yazılmış eski bir sınıftır. Java 1.1'den beri **kullanımdan kaldırılmıştır** (*deprecated*). Sınıf hâlâ derlenir, ama derleyici uyarı verir ve kullanılması önerilmez. Sorunu, karakterleri bayta doğru çevirmemesidir: her karakterin yalnız alt 8 bitini alır.

```
public class EskiAkis {
  @SuppressWarnings("deprecation")
  public static void main(String[] args) throws IOException {
    InputStream eski = new StringBufferInputStream("ş€");
    int b;
    while ((b = eski.read()) != -1) {
      System.out.print(b + " ");
    }
    System.out.println();
    byte[] dogru = "ş€".getBytes("UTF-8");
    InputStream yeni = new ByteArrayInputStream(dogru);
    while ((b = yeni.read()) != -1) {
      System.out.print(b + " ");
    }
    System.out.println();
  }
}
```

```
95 172
197 159 226 130 172
```

`ş`'nin Unicode kodu 351'dir, onaltılık `015F`. Alt 8 biti `5F` = 95'tir; bu da `_` karakteridir. `€`'nun kodu `20AC`'dir ve alt 8 biti `AC` = 172'dir. İki karakter iki anlamsız bayta dönüşür ve asıl bilgi kaybolur. Doğrusu UTF-8 baytlarıdır: `ş` için 197.159, `€` için 226.130.172. Bu baytlar `"ş€".getBytes("UTF-8")` ile elde edilip `ByteArrayInputStream` ile okunur. Dizgiyi karakter olarak okumak gerekiyorsa `StringReader` kullanılır. `@SuppressWarnings("deprecation")` etiketi, eski sınıfı bilerek kullanan bu örnekte derleyici uyarısını susturur.

8.6.4 Borular

`PipedInputStream` ve `PipedOutputStream`, iki **iş parçacığı** (*thread*) arasında güvenli veri iletişimi sağlar. İş parçacığı, aynı programın içinde bağımsız ilerleyen bir yürütme akışıdır. Bir iş parçacığı `PipedOutputStream` ile veri yazar, öbürü `PipedInputStream` ile bu veriyi okur. İki uç `new PipedOutputStream(pis)` ile birbirine bağlanır (Şekil 8.15).



Şekil 8.15: Boru. Yazıcı iş parçacığı `PipedOutputStream`'e yazar; baytlar 1024 baytlık dairesel bir tamponda

bekler; okuyucu iş parçacığı onları `PipedInputStream`'den okur.

Borunun dört özelliği vardır:

- Okuma ve yazma **eş zamanlıdır** (*synchronized*): boru boşsa okuyucu bekler, doluysa yazıcı bekler.
- Veri **tek yönde** akar. Çift yönlü iletişim için iki boru gerekir.
- Borunun tamponu varsayılan olarak 1024 bayttır.
- İki ucu **aynı iş parçacığında** kullanmak kilitlenmeye (*deadlock*) yol açabilir. Tampon dolunca yazıcı okuyucuyu bekler; okuyucu da aynı iş parçacığı olduğu için hiçbir zaman çalışmaz.

Aşağıdaki program bir okuyucu ve bir yazıcı iş parçacığı oluşturur. İş parçacıklarının gövdeleri lambda ifadeleriyle yazılmıştır.

```
PipedInputStream pis = new PipedInputStream();
PipedOutputStream pos = new PipedOutputStream(pis);
```

```
// Okuyucu iş parçacığı
Thread okuyucu = new Thread(() -> {
  try {
    int b;
    while ((b = pis.read()) != -1)
      System.out.print((char) b);
    System.out.println();
    pis.close();
  } catch (IOException e) {
    e.printStackTrace();
  }
});
```

```
// Yazıcı iş parçacığı
Thread yazici = new Thread(() -> {
  try {
    pos.write("Merhaba, boru!".getBytes());
    pos.close();
  } catch (IOException e) {
    e.printStackTrace();
  }
});
okuyucu.start();
yazici.start();
yazici.join();
okuyucu.join();
```

```
Merhaba, boru!
```

`start()` iş parçacığını başlatır; `join()` onun bitmesini bekler. İki iş parçacığı aynı anda çalışır ve hangisinin önce ilerleyeceği her çalıştırmada değişebilir. Okuyucu önce başlarsa `pis.read()`'te bekler, çünkü boru boştur. Yazıcı 14 baytı boruya yazar ve `pos.close()` der. Okuyucunun bekleyen `read()` çağrısı baytları sırayla alır. Boru boşalıp yazıcı ucu kapalı olduğunda `read()` −1 döndürür ve döngü biter. Çıktı her çalıştırmada aynıdır, çünkü boru baytların sırasını korur. Program bu not için iki kez çalıştırıldı ve iki çalıştırmada da aynı satırı yazdı.

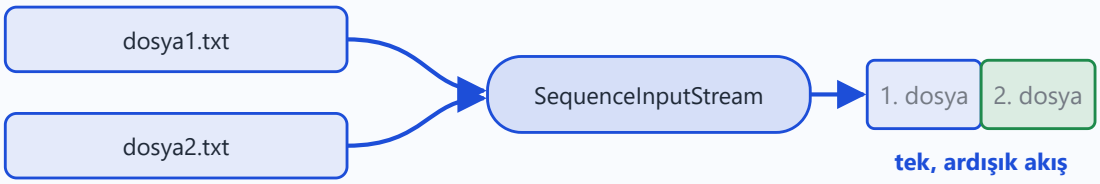
Kilitlenme uyarısını görmek için borunun tamponu 4 bayta küçültülür ve yazıcı 6 bayt yazmaya çalışır. Okuyan kimse yoktur:

```
yazıcı 1,5 s sonra hâlâ çalışıyor mu? true, borudaki bayt: 4
```

Yazıcı ilk 4 baytı yazar, sonra boru dolduğu için bekler. 1,5 saniye sonra da beklemeyi sürdürür. Okuyucu aynı iş parçacığı olsaydı program sonsuza dek dururdu. Bu deneme iş parçacığını arka plan iş parçacığı yaparak programın kapanabilmesini sağlar.

8.6.5 `SequenceInputStream`

`SequenceInputStream`, birden fazla `InputStream`'i uç uca birleştirerek tek bir ardışık akış oluşturur (Şekil 8.16). Birinci akış bitip −1 döndürdüğünde okuma kendiliğinden ikinci akışa geçer. Böylece farklı kaynaklardan gelen veri tek bir akış gibi işlenir.



Şekil 8.16: `SequenceInputStream` iki dosya akışını uç uca ekler. Okuyan program tek, ardışık bir akış görür.

```
try {
  FileInputStream is1 = new FileInputStream("dosya1.txt");
  FileInputStream is2 = new FileInputStream("dosya2.txt");
  SequenceInputStream sis = new SequenceInputStream(is1, is2);
  int okunanByte;
  while ((okunanByte = sis.read()) != -1)
    System.out.print((char) okunanByte);
  sis.close();     // is1 ve is2'yi de kapatır
} catch (IOException e) {
  System.err.println("Okuma hatası: " + e.getMessage());
}
```

```
Birinci dosya.
Ikinci dosya.
```

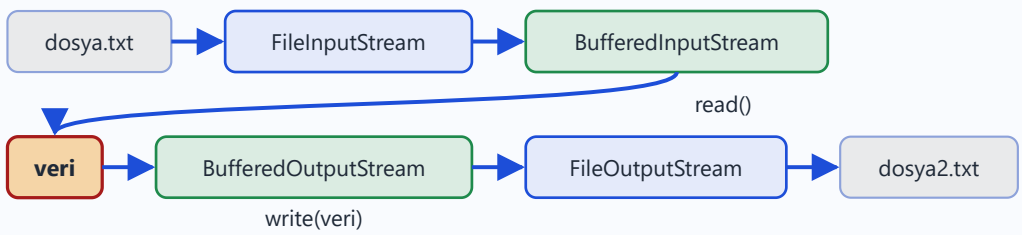
`dosya1.txt` `Birinci dosya.` satırını, `dosya2.txt` `Ikinci dosya.` satırını içerir. Döngü yalnız `sis.read()` çağırır ve dosyalar arasındaki geçişi fark etmez. `sis.close()` içteki iki akışı da kapatır. Okuma sırasında akışlardan birinde hata olursa `IOException` fırlatılır ve öbür akışa geçilmez. İkinci dosyanın metninde `İ` yerine `I` yazılması bilinçlidir: bayt bayt okunup `(char)` ile yazdırılan metinde Türkçe harfler bozulurdu.

8.7 Dosya kopyalama

Dosya kopyalamak, akışları öğrenmek için iyi bir alıştırmadır: bir girdi akışından okunan her şey bir çıktı akışına yazılır. Aynı iş farklı birimlerle yapılabilir: bayt bayt, blok blok ya da satır satır. Bu kesim yolları karşılaştırır ve her birinin tuzaklarını gösterir.

8.7.1 Bayt bayt kopyalama

En basit kopyalama, kaynak dosyadan okunan her baytı hedef dosyaya yazmaktır. Bayt akışları baytların anlamıyla ilgilenmediği için bu yöntem her tür dosyayı, yani metni, resmi ya da `.exe`'yi bozmadan kopyalar. Okuma ve yazmayı hızlandırmak için iki akış tamponlu akışlarla sarılır (Şekil 8.17).



Şekil 8.17: Bayt bayt kopyalama zinciri. Baytlar `dosya.txt`'den `FileInputStream` ve `BufferedInputStream` üzerinden `veri` değişkenine, oradan `BufferedOutputStream` ve `FileOutputStream` üzerinden

`dosya2.txt` 'ye gider.

```
import java.io.*;

public class BaytKopya {
  public static void main(String[] args) throws IOException {
    File dosya = new File("dosya.txt");
    File dosya2 = new File("dosya2.txt");
    FileInputStream fis = new FileInputStream(dosya);
    FileOutputStream fos = new FileOutputStream(dosya2);
    BufferedInputStream bis = new BufferedInputStream(fis);
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    int veri;
    // Bir byte oku. Dosyanın sonunda -1 dönecektir.
    while ((veri = bis.read()) != -1)
      bos.write(veri);    // Okunan byte'ı çıkışa yaz
    bos.close();
    bis.close();
    System.out.println(dosya2.length() + " byte kopyalandı");
  }
}
```

Kopyalama döngüsü tek satırdır: `while ((veri = bis.read()) != -1) bos.write(veri);`. `bis.read()` bir bayt okur ve dosya sonunda −1 döndürür. `bos.write(veri)` okunan baytı çıkış akışına yazar. Okunan değer bir `int` değişkende tutulmalıdır. Bunun nedeni −1 ile 255 değerli baytı ayırt etmektir. `read()` 0–255 aralığındaki her değeri geçerli bir bayt olarak döndürebilir; −1 bu aralığın dışında kaldığı için dosya sonunu belirsizlik olmadan bildirir.

Sık yapılan hata

`read()` 'in sonucunu `byte` tipinde tutmak: `byte b; while ((b = (byte) in.read()) != -1)`. Java'da `byte` işaretlidir ve −128 ile 127 arasında değer alır. Değeri 255 olan bir bayt, yani `FF`, `(byte)` dönüşümüyle −1'e dönüşür. Döngü bu baytı dosya sonu sanır ve durur; dosyanın geri kalanı kopyalanmaz. Doğru tip `int` 'tir.

Hatanın etkisi küçük bir denemeyle görülür. Üç baytlık `{10, (byte) 255, 20}` dizisi bu döngüyle okunursa yalnız ilk bayt işlenir:

```
işlenen: 1
```

Metin dosyalarında `FF` baytı nadiren görüldüğü için bu hata metin dosyalarıyla yapılan denemelerde fark edilmez. Resim ya da sıkıştırılmış dosyalarda ise `FF` sık görülür ve kopya yarıda kesilir.

8.7.2 Kod izleme: bayt bayt kopyalama

`dosya.txt` dört bayt içersin: `Java`. Tablo 8.15 programın her adımında iki tamponun ve hedef dosyanın durumunu verir.

Adım	Satır	`veri`	`bis` tamponu	`bos` tamponu	`dosya2.txt`	Ne oldu?
1	5–6	—	—	—	yok	İki `File` nesnesi yalnız yolları tutar.
2	7–8	—	—	—	0 bayt	`FileOutputStream` hedef dosyayı oluşturur; varsa içini siler.
3	9–10	?	boş	boş	0 bayt	Akışlar tamponlu akışlarla sarılır.
4	13	74	`Java`	boş	0 bayt	Tampon boş; diskten tek okumayla 4 bayt gelir. İlk bayt `'J'` döner.
5	14	74	`Java`	`J`	0 bayt	`write` baytı çıkış tamponuna koyar.
6–7	13–14	97	`Java`	`Ja`	0 bayt	Bayt tampondan gelir; diske gidilmez.
8–9	13–14	118	`Java`	`Jav`	0 bayt	Bayt tampondan gelir.
10–11	13–14	97	`Java`	`Java`	0 bayt	Çıkış tamponunda 4 bayt var.
12	13	−1	boş	`Java`	0 bayt	Tampon tükendi; ikinci disk okuması dosya sonunu bildirir.
13	15	−1	boş	boş	4 bayt	`bos.close()` önce `flush()` yapar: 4 bayt tek yazmayla diske gider.
14	16	−1	—	—	4 bayt	Giriş akışları kapatılır.
15	17	−1	—	—	4 bayt	Çıktı: `4 byte kopyalandı`

Tablo 8.15: `BaytKopya` programının izlemesi. Satır numaraları programın satırlarıdır; tamponlar 8192 baytlıktır.



Şekil 8.18: Bayt bayt kopyalamadan üç an. (a) İlk `read()` dört baytın tamamını giriş tamponuna alır. (b) Dört bayt çıkış tamponunda bekler; hedef dosya hâlâ 0 bayt. (c) `close()`'tan sonra dört bayt diskte.

İzlemenin en önemli sonucu, programın beş `read()` ve dört `write()` çağrısı yapmasına karşın diske çok daha az gidilmesidir. Bunu doğrulamak için `FileInputStream` ve `FileOutputStream`'in alt sınıfları yazıldı. Bu alt sınıflar diske giden her çağrıyı sayar ve yazdırır:

```
  disk read(dizi, 0, 8192) -> 4
  disk read(dizi, 0, 8192) -> -1
  disk write(dizi, 0, 4)
tamponlu: 2 okuma, 1 yazma
  disk read() -> 74
  disk read() -> 97
  disk read() -> 118
  disk read() -> 97
  disk read() -> -1
tamponsuz: 5 okuma, 4 yazma
```

Tamponlu kopyada diskten iki okuma yapılır. Birincisi 8192 bayt ister ve dört bayt alır. İkincisi tampon tükendikten sonra yapılır ve dosya sonunu (−1) bildirir. Diske yalnız bir yazma yapılır. Tamponsuz kopyada ise her bayt için bir okuma ve bir yazma, dosya sonu için de bir okuma daha gerekir: beş okuma, dört yazma. Dört baytlık bir dosyada fark küçüktür. Bir megabaytlık dosyada ise tamponsuz kopya bir milyondan fazla okuma ve yazma çağrısı yapar. Tamponlu kopya 129 okuma ve 128 yazmayla, yani yaklaşık 260 çağrıyla işi bitirir.

8.7.3 Blok blok kopyalama

Tamponlu akışlar disk erişimini azaltır, ama programın kendisi hâlâ her bayt için bir `read()` ve bir `write()` çağırır. Daha hızlı yol, veriyi programın kendi dizisine bloklar hâlinde okumaktır. `read(byte[] parca)` diziyi doldurmaya çalışır ve okuduğu bayt sayısını döndürür. `write(parca, 0, boyut)` dizinin yalnız ilk `boyut` baytını yazar.

```
public class ParcaKopya {
  public static void main(String[] args) throws IOException {
    // Veri, 16K'lık parçalar hâlinde okunacak
    byte[] parca = new byte[1024 * 16];
    File dosya = new File("dosya.txt");
    File dosya2 = new File("dosya2.txt");
    FileInputStream fis = new FileInputStream(dosya);
    FileOutputStream fos = new FileOutputStream(dosya2);
    BufferedInputStream bis = new BufferedInputStream(fis);
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    int boyut;
    // 16K'ya kadar oku. Dosyanın sonunda -1 dönecektir.
    while ((boyut = bis.read(parca)) > -1)
      bos.write(parca, 0, boyut);   // Okunan parçayı çıkışa yaz
    bos.close();
    bis.close();
    System.out.println(dosya2.length() + " byte kopyalandı");
  }
}
```

40.000 baytlık bir dosya için çıktı şudur:

```
40000 byte kopyalandı
```

`new byte[1024 * 16]` 16 KB'lık bir dizi oluşturur. Veri bu boyuttaki parçalar hâlinde işlenir: dosya ne kadar büyük olursa olsun programın bellek kullanımı sabit kalır. 40.000 baytlık dosya üç okumayla gelir: 16.384, 16.384 ve 7.232 bayt. Son parça dizinin tamamını doldurmaz. Bu not için 40.000 rastgele baytlık bir dosyayla yapılan deneme parça boylarını ve kopyanın kaynakla aynı olduğunu doğrular:

```
okunan parçalar: 16384 16384 7232
aynı mı: true
```

Döngü koşulu `> -1` biçimindedir; `!= -1` ile aynı anlama gelir, çünkü `read` −1'den küçük bir değer döndürmez. `read(byte[])` 'in 0 döndürmesi ise yalnız dizinin boyu 0 olduğunda mümkündür.

8.7.4 Kod izleme: blok kopyalama

Blok kopyalamada son parçanın ele alınışı kritiktir. İzlemeyi küçük tutmak için 16 baytlık bir dizi ve 39 baytlık bir dosya kullanılır; gerçek programlarda dizi 8–16 KB olur. Bu kez tamponlu akışlar kullanılmaz, kaynaklar try-with-resources ile açılır.

```
public class ParcaIzle {
  public static void main(String[] args) throws IOException {
    byte[] parca = new byte[16];
    try (InputStream in = new FileInputStream("dosya.txt");
      OutputStream out = new FileOutputStream("dosya2.txt")) {
      int boyut;
      while ((boyut = in.read(parca)) > -1) {
        out.write(parca, 0, boyut);
        System.out.println("boyut = " + boyut);
      }
    }
    long n = new File("dosya2.txt").length();
    System.out.println("Bitti: " + n + " byte");
  }
}
```

Adım	Satır	Değişkenler	Yeni çıktı
1	3: `byte[] parca = new byte[16];`	parca=byte[16]	—
2	4: `try (InputStream in = new FileInputStream("dosya.txt");` 5: `OutputStream out = new FileOutputStream("dosya2.txt")) {`	parca=byte[16]	—
3	7: `while ((boyut = in.read(parca)) > -1) {`	boyut=16	—
4	8: `out.write(parca, 0, boyut);`	boyut=16	—
5	9: `System.out.println("boyut = " + boyut);`	boyut=16	`boyut = 16`

Adım	Satır	Değişkenler	Yeni çıktı
6	7: `while ((boyut = in.read(parca)) > -1) {`	boyut=16	—
7	8: `out.write(parca, 0, boyut);`	boyut=16	—
8	9: `System.out.println("boyut = " + boyut);`	boyut=16	`boyut = 16`
9	7: `while ((boyut = in.read(parca)) > -1) {`	boyut=7	—
10	8: `out.write(parca, 0, boyut);`	boyut=7	—
11	9: `System.out.println("boyut = " + boyut);`	boyut=7	`boyut = 7`
12	7: `while ((boyut = in.read(parca)) > -1) {`	boyut=-1	—
13	13: `System.out.println("Bitti: " + n + " byte");`	boyut=-1, n=39	`Bitti: 39 byte`

Tablo 8.16: `ParcaIzle` programının izlemesi; `dosya.txt` 39 bayttır.

Tablo 8.16'nın ilk iki turunda `read(parca)` diziyi doldurur ve 16 döndürür. `write(parca, 0, 16)` okunan 16 baytı yazar. İkinci turda yeni 16 bayt aynı dizinin üzerine yazılır; dizi her turda yeniden kullanılır. Üçüncü turda dosyada yalnız 7 bayt kalmıştır. `read` dizinin ilk 7 hücresini yeniler ve 7 döndürür. Kalan 9 hücrede bir önceki bloktan kalan baytlar durur.



Şekil 8.19: Son blok. (a) Yalnız 7 bayt kaldı; dizinin ilk 7 hücresi yenilenir, kalan 9 hücre eski bloktan kalmadır. (b) `write(parca, 0, 7)` yalnız okunan 7 baytı yazar.

Burada `write(parca, 0, boyut)` yerine `write(parca)` yazılmış olsaydı dizinin 16 baytının tamamı yazılırdı. Eski 9 bayt da kopyaya girer ve kopya 16 + 16 + 16 = 48 bayt olurdu. Aynı dosyayla yapılan deneme bunu doğrular. Kopyanın son 9 baytı, ikinci bloktan kalan `kislar ve` parçasıdır:

```
write(parca) ile: 48 byte
son 9 bayt: kislar ve | gerçek son blok:  tampon
```

Dördüncü turda `read` −1 döndürür, `> -1` koşulu yanlış olur ve döngü biter. Kaynaklar kapanır ve kopya kaynakla aynı boydadır: 16 + 16 + 7 = 39 bayt. Okuma ve yazma sayısı dosya boyunun dizi boyuna bölümünün tavanıdır: ⌈39/16⌉ = 3. Bir megabaytlık bir dosya 16 KB'lık diziyle 64 turda kopyalanır.

Sık yapılan hata

Blok kopyalamada `out.write(parca);` yazmak. Son blokta dizi tam dolmaz; dizinin sonundaki eski veri de kopyaya gider ve kopya kaynaktan büyük olur. Doğrusu her zaman `write(parca, 0, boyut)` 'tur; `boyut` , `read` 'in son döndürdüğü değerdir. Aynı kural metne çevirmede de geçerlidir: `new String(parca, 0, boyut)` .

8.7.5 Satır satır kopyalama

Metin dosyaları satır satır da kopyalanabilir. Bu yol karakter akışlarıyla çalışır ve satırları kopyalarken işlemek gerektiğinde kullanışlıdır: satırları süzmek, numaralandırmak ya da değiştirmek gibi.

```
public class SatirKopya {
  public static void main(String[] args) throws IOException {
    File dosya = new File("dosya.txt");
    File dosya2 = new File("dosya2.txt");
    FileReader fr = new FileReader(dosya);
    BufferedReader br = new BufferedReader(fr);
    FileWriter fw = new FileWriter(dosya2);
    BufferedWriter bw = new BufferedWriter(fw);
    PrintWriter pw = new PrintWriter(bw);
    String satir;
    // dosya sonuna kadar bir satır oku
    while ((satir = br.readLine()) != null)
      pw.println(satir);    // satırı yaz
    pw.close();
    br.close();
    System.out.println("Kopyalandı: " + dosya2.getName());
  }
}
```

```
Kopyalandı: dosya2.txt
```

Okuma tarafında `FileReader` ve `BufferedReader` , yazma tarafında `FileWriter` , `BufferedWriter` ve `PrintWriter` zinciri vardır. Döngü her satırı `readLine()` ile alır ve `println` ile yazar. `pw.close()` zincirin tamamını, yani `bw` ve `fw` 'yi de kapatır.

Bu yöntemin iki sınırı vardır. Birincisi, yalnız metin dosyaları için uygundur. İkili bir dosya karakter akışından geçerse kodlama dönüşümü baytları bozar. İkincisi, kopya bayt bayt aynı olmayabilir. `readLine()` satır sonunu atar; `println` ise sistemin satır sonunu ekler: Windows'ta `\r\n` , Linux'ta `\n` . Kaynaktaki son satırın sonunda satır sonu yoksa kopyada bir satır sonu fazladan bulunur. Aşağıdaki deneme Windows'ta, son satırı satır sonu olmadan biten 41 baytlık bir dosyayla yapılmıştır; kopya 43 bayt tutar:

```
Kopyalandı: dosya2.txt
41 -> 43 bayt; line.separator uzunluğu 2
```

Kaynağın satır sonları Linux biçiminde (`\n`) olsaydı Windows'taki kopyada her satır bir bayt daha uzun olurdu. Bu nedenle bayt bayt aynı bir kopya gerekiyorsa her zaman bayt akışları kullanılır.

8.7.6 Web sayfası indirme

Kopyalama döngüsünün kaynağı bir dosya olmak zorunda değildir. Aşağıdaki program bir web sayfasını indirip diske kaydeder. Kaynak bu kez **ağdır**: `url.openStream()` bir `InputStream` döndürür. Döngünün geri kalanı dosya kopyalamayla aynıdır.

```
public class WebIndir {
  public static void main(String[] args) throws IOException {
    URL url = URI.create("https://sercankulcu.github.io/").toURL();
    BufferedInputStream bis = new BufferedInputStream(url.openStream());
    FileOutputStream fos = new FileOutputStream(new File("kopya.html"));
    BufferedOutputStream bos = new BufferedOutputStream(fos);
    for (int c = bis.read(); c != -1; c = bis.read()) {
      bos.write(c);
    }
    bis.close();
    bos.close();
    System.out.println("kopya.html: " + new File("kopya.html").length() + " byte");
  }
}
```

```
kopya.html: 14283 byte
```

Çıktıdaki boyut sayfaya ve indirme anına göre değişir. Bu değer, not hazırlanırken sayfanın sunucudan gelen boyutudur. Kopyalama döngüsü bu kez `for` ile yazılmıştır: ilk okuma başlatma kısmında yapılır, −1 değilse bayt yazılır ve güncelleme kısmında yeniden okunur. `while` ile yazılan döngüyle aynı işi görür.

Programdaki `URI.create(…).toURL()` kullanımı bilinçlidir. `new URL(String)` yapıcısı Java 20'den beri kullanımdan kaldırılmıştır; yerine bu iki adımlı yol önerilir. `URI` adresin söz dizimini daha sıkı denetler. Ağ bağlantısı kurulamazsa ya da sunucunun sertifikası doğrulanamazsa `openStream()` bir `IOException` alt sınıfı fırlatır. Bu program da `main` başlığındaki `throws IOException` ile hatayı JVM'e bırakır.

8.7.7 Kısa yollar ve karşılaştırma

Kopyalama döngüsünü her seferinde yeniden yazmak gerekmez. Java kütüphanesinde bu işi yapan iki metot vardır. `in.transferTo(out)` (Java 9+) bir girdi akışındaki bütün baytları bir çıktı akışına aktarır ve aktardığı bayt sayısını döndürür. `Files.copy(kaynak, hedef)` (Java 7+) bir dosyayı doğrudan başka bir dosyaya kopyalar. İkisi de içeride blok kopyalama yapar.

```
public class KopyaKisa {
  public static void main(String[] args) throws IOException {
    // Java 9+: transferTo
    try (InputStream in = new FileInputStream("dosya.txt");
      OutputStream out = new FileOutputStream("dosya2.txt")) {
      long adet = in.transferTo(out);
      System.out.println(adet + " byte aktarıldı");
    }
    // Java 7+: Files.copy
    Files.copy(Path.of("dosya.txt"), Path.of("dosya3.txt"),
         StandardCopyOption.REPLACE_EXISTING);
    System.out.println(Files.size(Path.of("dosya3.txt")) + " byte kopyalandı");
  }
}
```

```
4 byte aktarıldı
4 byte kopyalandı
```

`Files.copy` hedef dosya varsa, üzerine yazma izni verilmedikçe `FileAlreadyExistsException` fırlatır. Bu davranış bir güvenlik önlemidir: var olan bir dosyanın yanlışlıkla ezilmesini önler. Üzerine yazmak için üçüncü argüman olarak `StandardCopyOption.REPLACE_EXISTING` verilir. Seçenek verilmeden yapılan deneme istisnayı ve hedef dosyanın değişmediğini gösterir:

```
java.nio.file.FileAlreadyExistsException: k2.txt
k2.txt: eski
```

Tablo 8.17 kopyalama yöntemlerini karşılaştırır, Şekil 8.20 ise 1 MB'lık bir dosyada çağrı sayılarının ne kadar farklı olduğunu gösterir.

Yöntem	Birim	Uygun olduğu durum
Bayt bayt (`read()` / `write(int)`)	1 bayt	öğrenmek için; tamponsuz çok yavaş
Blok blok (`read(byte[])`)	8–16 KB	her tür dosya, hızlı
Satır satır (`readLine` / `println`)	1 satır	yalnız metin; satırları işlerken
`in.transferTo(out)` (Java 9+)	blok	iki akış arasında tek satırda kopya

Yöntem	Birim	Uygun olduğu durum
`Files.copy(kaynak, hedef)` (Java 7+)	blok	dosyadan dosyaya en kısa yol

Tablo 8.17: Kopyalama yöntemleri.



Şekil 8.20: 1 MB'lık dosyayı kopyalarken `read` çağrısı sayısı (çubuk boyları logaritmiktir). Bayt bayt okuma 1.048.576, 8 KB'lık diziyle blok okuma 128 çağrı yapar; `Files.copy` tek metot çağrısıdır.

Hangi yöntem seçilirse seçilsin üç kural geçerlidir: tamponlu çalışılır, akışlar kapatılır (tercihen try-with-resources ile) ve `IOException` ele alınır. Gerçek bir programda dosyadan dosyaya kopya için `Files.copy`, iki akış arasında kopya için `transferTo` kullanılır. Elle yazılan blok döngüsü, kopyalanırken veri üzerinde bir işlem yapılacaksa gerekir; örneğin ilerleme çubuğu göstermek ya da baytları şifrelemek için.

8.8 Rastgele erişim

Akışların hepsi sıralıdır: veriyi baştan sona okur ya da yazar. Bazı işler ise dosyanın ortasındaki bir kayda doğrudan gitmeyi gerektirir. Bir öğrenci dosyasındaki bininci öğrencinin notunu okumak için önceki 999 kaydı okumak gereksiz bir yüktür.

8.8.1 Sıralı ve rastgele erişim

Tanım 8.10

Sıralı erişimde (*sequential access*) her okuma ya da yazma bir öncekinin bittiği konumdan devam eder; geri dönülemez. **Rastgele erişimde** (*random access*) program dosya içinde istenen konuma atlar ve oradan okur ya da yazar. Konum, dosya başından itibaren bayt sayısıdır ve 0'dan başlar.

Sıralı erişim bir kaset gibidir: bir şarkıya ulaşmak için öncekiler sarılır. Rastgele erişim bir plak gibidir: iğne istenen yere konur (Şekil 8.21). Akışların (`FileInputStream`, `BufferedReader` ...) hepsi sıralıdır. Sıralı erişim, dosyanın yapısı bilinmediğinde ya da dosyanın tamamı işlenirken kullanılır; kopyalama bunun tipik örneğidir.



Şekil 8.21: Sıralı erişim baytları baştan sona okur. Rastgele erişim `seek` ile istenen konuma atlar.

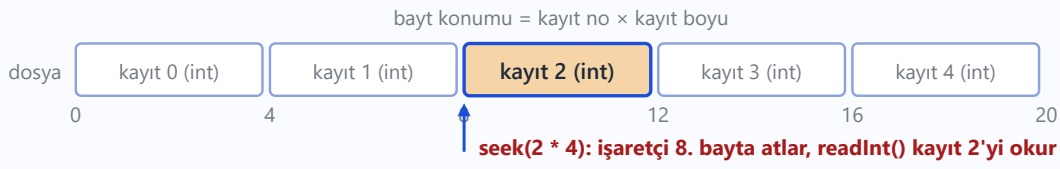
Rastgele erişim, kayıtların boyu ve konumu biliniyorsa kullanılabilir. Bütün kayıtlar aynı boyda olursa k numaralı kaydın konumu çarpmayla bulunur: konum = k · kayıt boyu (numaralar 0'dan başlar). Belirli bir kaydı okumak ya da yerinde değiştirmek için bu yol çok kullanışlıdır. Sabit boy şartı önemlidir. Satırları farklı uzunluktaki bir metin dosyasında k'ıncı satırın nerede başladığı hesaplanamaz.

8.8.2 `RandomAccessFile`

Java'da rastgele erişim `java.io.RandomAccessFile` sınıfıyla yapılır. Bu sınıf akışlardan farklıdır: tek bir nesne hem okur hem yazar ve `InputStream` / `OutputStream` hiyerarşisinde yer almaz. Nesnenin içinde bir **dosya işaretçisi** (*file pointer*) vardır. Her okuma ve yazma işaretçinin gösterdiği konumda yapılır ve işaretçiyi okunan ya da yazılan bayt sayısı kadar ilerletir. Tablo 8.18 sınıfın temel üyelerini verir.

Üye	Görevi
`RandomAccessFile(String ad, String kip)`	kip `"r"`: yalnız okuma; `"rw"`: okuma ve yazma (dosya yoksa oluşturulur)
`seek(long konum)`	dosya işaretçisini verilen konuma taşır
`getFilePointer()`	işaretçinin dosya başına uzaklığı (bayt)
`length()`	dosyanın uzunluğu (bayt)
`read()` / `write(int b)`	bir bayt okur (`int` olarak döndürür) / yazar
`readInt()`, `writeInt(int)`, `readDouble()` ...	ilkel tipleri ikili okur ve yazar (`int`: 4 bayt)
`close()`	sistem kaynaklarını serbest bırakır

Tablo 8.18: `RandomAccessFile` sınıfının temel üyeleri.



Şekil 8.22: 4 baytlık `int` kayıtları. `seek(2 * 4)` işaretçiyi 8. bayta taşır; `readInt()` 2 numaralı kaydı okur. Bayt konumu = kayıt numarası · kayıt boyu.

Aşağıdaki program aynı dosyayı önce sıralı, sonra rastgele erişimle okur. `dosya` burada 124 baytlık bir veri dosyasıdır. Bu notta program, ilk baytı 1 ve 100–103. baytları `00 00 01 00` olan bir dosyayla çalıştırılmıştır.

```
public class RasgeleErisim {
  public static void main(String[] args) throws IOException {
    // Sıralı Erişim
    FileInputStream sirali = new FileInputStream("dosya");
    int bayt = sirali.read();
    sirali.close();
    System.out.println("İlk byte: " + bayt);
    // Rasgele Erişim
    RandomAccessFile rast = new RandomAccessFile("dosya", "rw");
    // Dosyanın başından 100 byte ilerideki konuma git
    rast.seek(100);
    // 100. konumdan itibaren 4 byte'lık bir int oku
    int veri = rast.readInt();
    System.out.println("100. konumdaki int: " + veri);
    rast.close();
  }
}
```

```
İlk byte: 1
100. konumdaki int: 256
```

`FileInputStream` dosyanın başından okur ve ilk baytı, 1'i verir. `seek(100)` işaretçiyi 100 konumuna, yani **101. bayta** taşır, çünkü konumlar 0'dan başlar. Oradaki dört bayt `00 00 01 00`'dır. `readInt()` en anlamlı baytı önce okur: $0 \cdot 2^{24} + 0 \cdot 2^{16} + 1 \cdot 2^{8} + 0 = 256$.

`RandomAccessFile`'ın sınırlarda davranışı akışlardan biraz farklıdır. İşaretçi dosyanın sonunun ötesine taşınabilir; orada yazma yapılırsa dosya o noktaya kadar büyür ve aradaki baytlar tanımsız kalır (bu denemede sıfırdır). Dosya sonunda `read()` −1 döndürür, `readInt()` ise `EOFException` fırlatır. Kip yalnız `"r"`, `"rw"`, `"rws"` ya da `"rwd"` olabilir; `"w"` geçersizdir:

```
seek(10)+write: uzunluk 11, işaretçi 11
ilk bayt: 0
sonda read(): -1
sonda readInt(): EOFException
r kipi, dosya yok: FileNotFoundException
w kipi: Illegal mode "w" must be one of "r", "rw", "rws", or "rwd"
```

8.8.3 Kod izleme: kayıtlara rastgele erişim

Aşağıdaki program beş notu `int` olarak bir dosyaya yazar. Sonra üçüncü notu doğrudan okur, dördüncü notu yerinde değiştirir ve bütün notları toplar.

```
import java.io.*;

public class Kayitlar {
  public static void main(String[] args) throws IOException {
    RandomAccessFile f = new RandomAccessFile("n.dat", "rw");
    int[] notlar = {70, 85, 90, 45, 60};
    for (int n : notlar)
      f.writeInt(n);              // her kayıt 4 byte
    System.out.println("Uzunluk: " + f.length());
    f.seek(2 * 4);                  // 3. kayda git
    System.out.println("3. not: " + f.readInt());
    System.out.println("Konum: " + f.getFilePointer());
    f.seek(3 * 4);                  // 4. kayıt
    f.writeInt(50);                 // 45 yerine 50
    f.seek(0);
    int toplam = 0;
    while (f.getFilePointer() < f.length())
      toplam += f.readInt();
    System.out.println("Toplam: " + toplam);
    f.close();
  }
}
```

Adım	Satır	Değişkenler	Yeni çıktı
1	5: `RandomAccessFile f = new RandomAccessFile("n.dat", "rw");`	—	—
2	6: `int[] notlar = {70, 85, 90, 45, 60};`	notlar=[70, 85, 90, 45, 60]	—
3	7: `for (int n : notlar)` 8: `f.writeInt(n); // her kayıt 4 byte`	n=70	—
4	7: `for (int n : notlar)` 8: `f.writeInt(n); // her kayıt 4 byte`	n=60	—
5	9: `System.out.println("Uzunluk: " + f.length());`	—	`Uzunluk: 20`
6	10: `f.seek(2 * 4); // 3. kayda git`	—	—
7	11: `System.out.println("3. not: " + f.readInt());`	—	`3. not: 90`
8	12: `System.out.println("Konum: " + f.getFilePointer());`	—	`Konum: 12`
9	13: `f.seek(3 * 4); // 4. kayıt`	—	—
10	14: `f.writeInt(50); // 45 yerine 50`	—	—
11	15: `f.seek(0);`	toplam=0	—

Adım	Satır	Değişkenler	Yeni çıktı
12	17: `while (f.getFilePointer() < f.length())` 18: `toplam += f.readInt();`	toplam=355	—
13	19: `System.out.println("Toplam: " + toplam);`	toplam=355	`Toplam: 355`

Tablo 8.19: `Kayitlar` programının izlemesi.

Tablo 8.19'un ilk adımında `"rw"` kipi dosya yoksa onu oluşturur; işaretçi 0'dadır. Döngüde her `writeInt` 4 bayt yazar ve işaretçiyi 4 ilerletir. Döngü bitince dosya 5 · 4 = 20 bayttır ve işaretçi dosyanın sonunda, 20'dedir. Dosyanın gerçek baytları şöyledir:

```
20: 00 00 00 46 00 00 00 55 00 00 00 5A 00 00 00 32 00 00 00 3C
```

`46` = 70, `55` = 85, `5A` = 90'dır. Bu satırdaki dördüncü kayıt `32` = 50'dir, çünkü baytlar program bittikten sonra yazdırılmıştır; ilk yazılan değer 45, yani `2D` idi.

Altıncı adımda `seek(2 * 4)` işaretçiyi 8. bayta taşır. Burası üçüncü kaydın, yani 2 numaralı kaydın başıdır. Önceki kayıtlar okunmaz; doğrudan oraya atlanır. `readInt()` 8–11. baytları okur ve 90 döndürür. İşaretçi 12'ye ilerler. Dokuzuncu adımdaki `seek(3 * 4)` işaretçiyi 12'ye taşır; işaretçi zaten oradadır. Onuncu adımda `writeInt(50)` 45'in yerine yazılır. Dosyanın geri kalanı değişmez. Sıralı bir akışla bu yapılamaz: dosyanın ortasındaki bir kaydı değiştirmek için bütün dosyanın yeniden yazılması gerekirdi.



Şekil 8.23: Rastgele erişimden iki an. (a) `seek(8)` işaretçiyi üçüncü kaydın başına taşır. (b) `writeInt(50)` dördüncü kaydı yerinde değiştirir.

Son adımlarda `seek(0)` başa döner. Döngü, işaretçi dosya uzunluğuna ulaşana kadar `int` okur: 70 + 85 + 90 + 50 + 60 = 355. Programın çıktısı sunumdaki değerlerle aynıdır.

```
Uzunluk: 20
3. not: 90
Konum: 12
Toplam: 355
```

Maliyet açısından fark açıktır. Sıralı akışla k numaralı kayda ulaşmak için önceki k kaydın okunması gerekir; süre k ile orantılıdır, $O(k)$. Rastgele erişimde konum bir çarpmayla hesaplanır ve tek `seek` yapılır; süre kayıt numarasından bağımsızdır, $O(1)$.

Örnek 8.1

Bir öğrenci dosyasında her kayıt sırasıyla bir `int` öğrenci numarası, bir `double` ortalama ve bir `int` kredi tutsun. Kayıtlar 0'dan numaralanıyor. 37 numaralı kaydın ortalaması nasıl okunur?

Kayıt boyu 4 + 8 + 4 = 16 bayttır. 37 numaralı kayıt $37 \cdot 16 = 592$. baytta başlar. Ortalama, kaydın içinde öğrenci numarasından sonra gelir; kayıt başına göre 4 bayt ileridedir. `f.seek(37 * 16 + 4); double ort = f.readDouble();` yazılır. Okumadan sonra işaretçi 592 + 4 + 8 = 604'tedir; bu da aynı kaydın kredi alanının başıdır. Kayıtlardan biri `double` yerine `float` ile yazılmış olsaydı bütün konum hesapları bozulurdu. Rastgele erişim, dosya biçiminin kesin olarak bilinmesini gerektirir.

8.9 Serileştirme

Şimdiye kadar dosyalara sayı ve metin yazıldı. Nesne yönelimli bir programda ise saklanmak istenen şey çoğu zaman bir nesnedir: bir oyunun durumu, bir öğrenci kaydı, bir ayar nesnesi. Nesnenin her alanını tek tek yazıp geri okumak zahmetlidir. Java bu işi tek çağrıyla yapan bir mekanizma sunar.

8.9.1 Serileştirme nedir?

Tanım 8.11

Serileştirme (*serialization*), bir nesnenin durumunu, yani alanlarının değerlerini, bir bayt dizisine çevirmektir. Bu bayt dizisi bir dosyaya yazılabilir ya da ağ üzerinden başka bir ortama taşınabilir. Bayt dizisinden yeniden nesne kurmaya **serileştirmeyi geri alma** (*deserialization*) denir.

Bir sınıfın nesnelerinin serileştirilebilmesi için sınıf `java.io.Serializable` arayüzünü uygulamalıdır. Bu arayüzün hiç metodu yoktur. Böyle arayüzlere **işaret arayüzü** (*marker interface*) denir; tek görevi sınıfı "serileştirilebilir" diye işaretlemektir. Serileştirme mekanizması nesneyi yazmadan önce bu işarete bakar.

Serileştirmenin üç akışı vardır (Şekil 8.24). Bellekte bir nesne durur. `ObjectOutputStream.writeObject` onu baytlara çevirir; dosya `AC ED` sihirli sayısıyla başlar. `ObjectInputStream.readObject` bu baytlardan yeni bir nesne kurar. Yeni nesnenin değerleri aynıdır ama kendisi farklı bir nesnedir.



Şekil 8.24: Serileştirme ve geri alma. `writeObject` bellekteki `Ogrenci` nesnesini 72 baytlık bir diziye çevirip dosyaya yazar; `readObject` aynı durumda, ama farklı bir nesne kurar.

Serileştirme oyun kayıtlarında, önbelleklerde ve nesnelerin ağ üzerinden başka bir JVM'e gönderilmesinde kullanılır. Biçimi Java'ya özgüdür ve sınıfın yapısına sıkı bağlıdır. Bu yüzden uzun süre saklanacak ya da başka dillerle paylaşılacak veriler için JSON gibi metin biçimleri daha çok tercih edilir. Ayrıca güvenilmeyen bir kaynaktan gelen serileştirilmiş veriyi geri almak güvenlik açığı oluşturabilir, çünkü geri alma sırasında programın sınıfları bilinmeyen verilerle kurulur.

8.9.2 Serileştirilebilir sınıf

```
import java.io.Serializable;

class Ogrenci implements Serializable {
  private static final long serialVersionUID = 1L;
  private String ad;
  private int yas;

  public Ogrenci(String ad, int yas) {
    this.ad = ad;
    this.yas = yas;
  }

  public String toString() {
    return "Ogrenci [ad=" + ad + ", yas=" + yas + "]";
  }
}
```

Sınıftaki üç ayrıntı serileştirmeyle ilgilidir:

- `implements Serializable` sınıfı serileştirilebilir yapar. Metot yazmak gerekmez.

- `serialVersionUID` sınıfın **sürüm kimliğidir**. Serileştirilmiş baytlara bu sayı da yazılır. Okunurken dosyadaki sayı ile programdaki sınıfın sayısı karşılaştırılır; farklıysa `InvalidClassException` fırlatılır. Sayının elle yazılması önerilir. Yazılmazsa serileştirme mekanizması onu sınıfın yapısından hesaplar; sınıfa bir alan ya da metot eklenince hesaplanan sayı değişir ve eski dosyalar okunamaz.
- Alanlar da serileştirilebilir olmalıdır. `String` ve `int` serileştirilebilir. Bir alanın tipi `Serializable` değilse yazma sırasında `NotSerializableException` fırlatılır.

Sürüm denetimi gerçekten çalışır. `serialVersionUID = 1L` ile yazılan bir dosya, aynı sınıfın `2L` sürümüyle okunmaya çalışıldığında şu istisna gelir:

```
java.io.InvalidClassException: Ogrenci; local class incompatible: stream classdesc
serialVersionUID = 1, local class serialVersionUID = 2
```

8.9.3 Nesneyi yazma ve geri okuma

Serileştirme için `ObjectOutputStream`, geri almak için `ObjectInputStream` kullanılır. İkisi de sarmalayıcı akıştır ve bir dosya akışını sarar.

```
// Serileştirme işlemi
try (ObjectOutputStream cikti = new ObjectOutputStream(
    new FileOutputStream("ogrenci.ser"))) {
  Ogrenci ogrenci = new Ogrenci("Ahmet", 20);
  cikti.writeObject(ogrenci);
  System.out.println("Serileştirme başarılı.");
} catch (IOException e) {
  e.printStackTrace();
}
System.out.println("ogrenci.ser: " + new File("ogrenci.ser").length() + " byte");
// Seriyi okuma işlemi
try (ObjectInputStream girdi = new ObjectInputStream(
    new FileInputStream("ogrenci.ser"))) {
  Ogrenci ogrenci = (Ogrenci) girdi.readObject();
  System.out.println("Ogrenci: " + ogrenci);
} catch (IOException | ClassNotFoundException e) {
  e.printStackTrace();
}
```

```
Serileştirme başarılı.
ogrenci.ser: 72 byte
Ogrenci: Ogrenci [ad=Ahmet, yas=20]
```

İlk blok bir `Ogrenci` nesnesi oluşturur ve `writeObject` ile yazar. Dosya 72 bayttır. Bu küçük nesne için çok gibi görünür. Ama dosyada yalnız iki alanın değeri değil, sınıfın tanımı da vardır: sınıfın adı, sürüm kimliği, alanların adları ve tipleri. Dosyanın baytları bunu gösterir:

```
AC ED 00 05 73 72 00 07 |
····sr··Ogrenci··········I··yasL··adt··Ljava/lang/String;xp····t··Ahmet
```

İlk dört bayt `AC ED 00 05` akış başlığıdır: sihirli sayı ve serileştirme protokolünün sürümü. Okunabilir kısımda `Ogrenci`, `yas`, `ad`, `java/lang/String` ve `Ahmet` görünür.

İkinci blok nesneyi geri okur. `readObject` `Object` tipinde bir değer döndürür; bu yüzden `(Ogrenci)` dönüşümü gerekir. `readObject` iki kontrol edilen istisna fırlatabilir: `IOException` ve `ClassNotFoundException`. İkincisi, dosyadaki sınıf programda bulunamadığında fırlatılır. Program ikisini çoklu `catch` ile birlikte yakalar. `ClassNotFoundException` yakalanmaz ya da bildirilmezse kod derlenmez:

```
Ana.java:6: error: unreported exception ClassNotFoundException; must be caught or declared
to be thrown
      Object o = g.readObject();
                              ^
1 error
```

Dönüşüm yanlış tipe yapılırsa çalışma zamanında `ClassCastException`, dosya hiç serileştirilmiş veri değilse `StreamCorruptedException` fırlatılır:

```
ClassCastException
StreamCorruptedException: invalid stream header: 6D657268
```

`6D657268`, düz metin dosyasının ilk dört baytıdır: `m`, `e`, `r`, `h`. Akış başlığı `AC ED` olmadığı için okuma en baştan reddedilir.

8.9.4 `transient` ve `static` alanlar

Bir nesnenin her alanı saklanmak istenmez. Şifre saklanmamalıdır. Geçici bir önbellek ya da açık bir dosya bağlantısı saklanamaz bile. Bu tür alanlar `transient` (geçici) anahtar sözcüğüyle işaretlenir.

Tanım 8.12

`transient` olarak işaretlenen alan serileştirmede atlanır. Nesne geri okunduğunda bu alan tipinin varsayılan değerini alır: referanslar `null`, sayılar 0, `boolean` `false`. `static` alanlar ise nesneye değil sınıfa ait oldukları için hiç serileştirilmez.

```
class Kullanici implements Serializable {
  private static final long serialVersionUID = 1L;
  static int sayac;
  private String ad;
  private transient String sifre;   // yazılmaz

  Kullanici(String ad, String sifre) {
    this.ad = ad;
    this.sifre = sifre;
  }

  public String toString() {
    return "Kullanici [ad=" + ad + ", sifre=" + sifre + "]";
  }
}
```

Aşağıdaki deneme bir `Kullanici` nesnesini yazar, `static` sayacı değiştirir ve nesneyi geri okur:

```
Kullanici k = new Kullanici("ayse", "gizli123");
Kullanici.sayac = 5;
ObjectOutputStream c = new ObjectOutputStream(
    new FileOutputStream("k.ser"));
c.writeObject(k);
c.close();
Kullanici.sayac = 99;        // static: sınıfa ait
ObjectInputStream g = new ObjectInputStream(
    new FileInputStream("k.ser"));
Kullanici okunan = (Kullanici) g.readObject();
g.close();
System.out.println(okunan);
System.out.println("sayac = " + Kullanici.sayac);
System.out.println("aynı nesne mi? " + (okunan == k));
```

```
Kullanici [ad=ayse, sifre=null]
sayac = 99
aynı nesne mi? false
```

Üç satırın her biri bir kuralı gösterir. `sifre` `null` gelmiştir, çünkü `transient` alan dosyaya yazılmamıştır. `sayac` 99'dur, çünkü dosyada `sayac` yoktur ve okuma onu değiştiremez; değer bellekteki son değerdir. `okunan == k` `false`'tur, çünkü `readObject` yeni bir nesne kurar.

8.9.5 Kod izleme: serileştirme

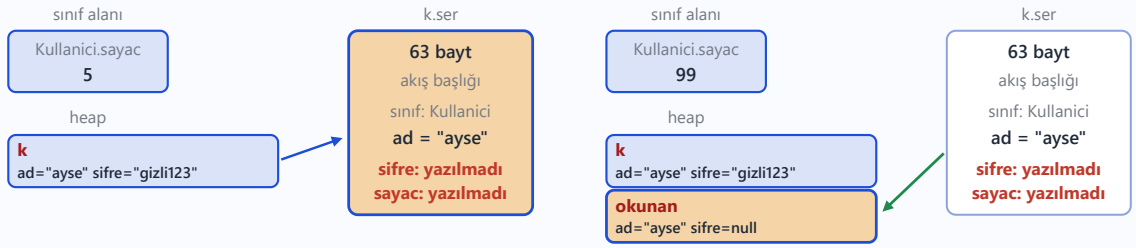
Aynı kod parçası adım adım izlendiğinde dosyanın boyutu ve nesnelerin durumu her an görülür. İzleme programı, sunumdaki koda iki satır ekleyerek dosya boyutunu da yazdırmıştır:

```
[başlıktan sonra: 4]
[k.ser: 63]
Kullanici [ad=ayse, sifre=null]
sayac = 99
aynı nesne mi? false
```

Adım	Satır	Değişkenler	Yeni çıktı
1	1: `Kullanici k = new Kullanici("ayse", "gizli123");`	k=Kullanici@1	—
2	2: `Kullanici.sayac = 5;`	k=Kullanici@1, Kullanici.sayac=5	—
3	3: `ObjectOutputStream c = new ObjectOutputStream(` 4: `new FileOutputStream("k.ser"));`	c=ObjectOutputStream	—
4	5: `c.writeObject(k);`	c=ObjectOutputStream	—
5	6: `c.close();`	c=kapalı	—
6	7: `Kullanici.sayac = 99; // static: sınıfa ait`	Kullanici.sayac=99	—
7	8: `ObjectInputStream g = new ObjectInputStream(` 9: `new FileInputStream("k.ser"));`	g=ObjectInputStream	—
8	10: `Kullanici okunan = (Kullanici) g.readObject();`	okunan=Kullanici@2	—
9	12: `System.out.println(okunan);`	okunan=Kullanici@2	`Kullanici [ad=ayse, sifre=null]`
10	13: `System.out.println("sayac = " + Kullanici.sayac);`	Kullanici.sayac=99	`sayac = 99`
11	14: `System.out.println("aynı nesne mi? " + (okunan == k));`	okunan == k sonucu=false	`aynı nesne mi? false`

Tablo 8.20: Serileştirme izlemesi. `Kullanici@1` ve `Kullanici@2` heap'teki iki ayrı nesneyi gösterir.

Tablo 8.20'nin birinci adımında heap'te yeni bir `Kullanici` nesnesi oluşur; `ad` ve `sifre` doludur. `static` `sayac` sınıfın alanıdır ve tek kopyası vardır. Üçüncü adımda `ObjectOutputStream` açılır ve dosyaya hemen 4 baytlık akış başlığını yazar. Dördüncü adımda `writeObject` sınıf tanımını ve alanları yazar; dosya 63 bayt olur. `transient` `sifre` ve `static` `sayac` yazılmaz.



Şekil 8.25: Serileştirme izlemesinden iki an. (a) `writeObject` yalnız `ad`'ı yazar; dosya 63 bayt. (b) `readObject` yeni bir nesne kurar; `sifre` `null`'dır.

Sekizinci adım serileştirmenin ince bir noktasını taşır. `readObject` heap'te yeni bir nesne kurar ama `Kullanici`'nin **yapıcısını çağırmaz**. Alanların değerleri doğrudan dosyadan gelir. Dosyada olmayan alanlar varsayılan değerde kalır. Bu yüzden `transient` bir alanın tanımındaki ilk değer de çalışmaz: `transient int b = 2;` yazılmış olsa bile geri okunan nesnede `b` 0'dır. Son adımda iki referans karşılaştırılır; içerikleri benzese de iki ayrı nesnedir.

> **Sık yapılan hata**
>
> `Serializable`'ı unutmak. `writeObject` ile yazılan sınıf ya da alanlarından birinin tipi bu arayüzü uygulamıyorsa çalışma zamanında `java.io.NotSerializableException: Araba` gibi bir istisna fırlatılır. Derleyici bunu varsayılan ayarlarla denetlemez. Çözüm sınıfı `Serializable` yapmak ya da alanı `transient` işaretlemektir.

8.10 NIO

Java NIO (*New I/O*, yeni G/Ç), Java 1.4 ile `java.nio` paketinde geldi. Klasik `java.io` akışlarına göre daha yüksek başarım ve esneklik sağlar. Java 7 ile gelen NIO.2 (`java.nio.file`) dosya işlemleri için `Path` ve `Files` sınıflarını ekledi.

8.10.1 NIO'nun yapı taşları

NIO dört temel kavram üzerine kuruludur:

- **Tampon** (`Buffer`): veriyi geçici olarak tutan bellek alanları.
- **Kanal** (`Channel`): giriş ve çıkış işlemlerini yöneten, çift yönlü olabilen bağlantı nesneleri.
- **Seçici** (`Selector`): birden çok kanalı tek bir iş parçacığıyla izleyen ve hazır olanları bildiren nesne.
- **Yol** (`Path`): dosya ve klasör yollarını temsil eden NIO.2 sınıfı.

8.10.2 Akış mı, kanal mı?

Klasik G/Ç'de program bir akışla çalışır ve veri tek yönde, bayt bayt ya da karakter karakter akar. NIO'da program bir tampona yazar ya da tampondan okur; kanal tampon dolusu veriyi tampon ile kaynak ya da hedef arasında taşır (Şekil 8.26).



Şekil 8.26: Klasik akış tek yönlü ve sıralıdır. NIO'da program kanal ve tampon üzerinden iki yönlü blok aktarımı yapar.

	Klasik G/Ç (`java.io`)	NIO (`java.nio`)
Temel kavram	akış	kanal ve tampon
Yön	tek yön: girdi ya da çıktı	kanal çift yönlü olabilir
Birim	bayt ya da karakter	tampon dolusu blok
Bekleme	her zaman bloklayan	bloklamayan kip de var
Çok bağlantı	bağlantı başına bir iş parçacığı	tek iş parçacığı ve seçici
Ne zaman?	basit dosya ve konsol işleri	çok bağlantılı sunucular, büyük dosyalar

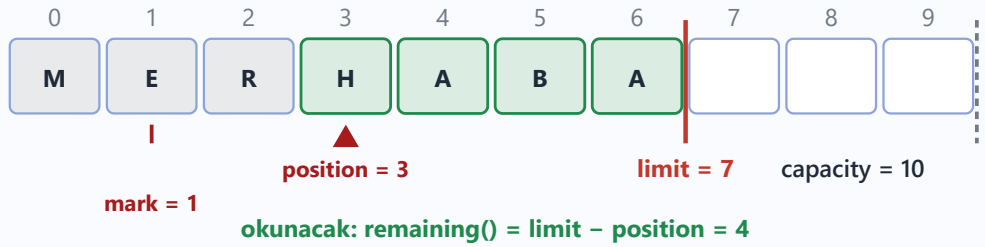
Tablo 8.21: Klasik G/Ç ile NIO'nun karşılaştırması.

"Bloklamayan" (*non-blocking*) kip, NIO'nun en önemli farkıdır. Klasik akışta `read()` veri gelene kadar bekler; iş parçacığı o sürede başka bir iş yapamaz. Bin istemcili bir sunucu bin iş parçacığı açmak zorunda kalır. Bloklamayan bir kanal ise veri yoksa hemen döner. Tek bir iş parçacığı böylece birçok kanalı sırayla yoklayabilir.

8.10.3 Tampon ve özellikleri

Tanım 8.13

NIO **tamponu**, verileri geçici olarak saklayan sabit boyutlu bir bellek alanıdır: bir dizi ve onun üzerinde konum bilgileri. Dört özelliği vardır. **Kapasite** (*capacity*) tutulabilecek en fazla öğe sayısıdır ve hiç değişmez. **Konum** (*position*) bir sonraki okumanın ya da yazmanın yapılacağı indistir. **Sınır** (*limit*) okunamayan ya da yazılamayan ilk indistir. **İşaret** (*mark*), `mark()` ile saklanan konumdur; `reset()` oraya döner. Her zaman 0 ≤ işaret ≤ konum ≤ sınır ≤ kapasite.



Şekil 8.27: Kapasitesi 10 olan bir tampon. Konum 3, sınır 7, işaret 1. Okunacak öğe sayısı `remaining()` = sınır − konum = 4'tür.

Her ilkel tip için bir tampon sınıfı vardır: `ByteBuffer`, `CharBuffer`, `ShortBuffer`, `IntBuffer`, `LongBuffer`, `FloatBuffer`, `DoubleBuffer`. Kanallar bayt taşıdığı için en çok `ByteBuffer` kullanılır. Tampon ya `ByteBuffer.allocate(1024)` ile oluşturulur ya da var olan bir dizi `ByteBuffer.wrap(dizi)` ile sarılır.

Bir tamponun iki kipi vardır. **Yazma kipinde** konum bir sonraki boş hücreyi, sınır kapasiteyi gösterir. **Okuma kipinde** konum bir sonraki okunacak hücreyi, sınır yazılan verinin sonunu gösterir. Kipler arasında `flip()` ile geçilir. Tampon kendisinin hangi kipte olduğunu bilmez; programcı bunu izlemek zorundadır.

8.10.4 `ByteBuffer` örnekleri

```
public class TamponOrnek {
  public static void main(String[] args) {
    // 1 KB boyutunda bir ByteBuffer oluştur
    ByteBuffer buffer = ByteBuffer.allocate(1024);
    // Veri yazma
    buffer.put("Merhaba, Java!".getBytes());
    System.out.println("put sonrası  : " + buffer);
    // Buffer'ı okuma moduna geçirme
    buffer.flip();
    System.out.println("flip sonrası : " + buffer);
    // Veriyi okuma
    while (buffer.hasRemaining()) {
      System.out.print((char) buffer.get());
    }
    System.out.println();
  }
}
```

```
put sonrası  : java.nio.HeapByteBuffer[pos=14 lim=1024 cap=1024]
flip sonrası : java.nio.HeapByteBuffer[pos=0 lim=14 cap=1024]
Merhaba, Java!
```

`put` 14 baytı yazar ve konumu 14'e ilerletir. `flip()` sınırı 14'e çeker ve konumu 0'a döndürür; tampon artık okuma kipindedir. `hasRemaining()` konumun sınırdan küçük olup olmadığını sorar. Döngü tam 14 bayt okur. `flip()` unutulsaydı okuma 14. konumdan başlar ve 1010 boş bayt okunurdu.

Bu döngü yalnız ASCII metinde doğru sonuç verir. Her baytı tek başına `char`'a çevirmek, çok baytlı karakterleri bozar:

```
public class TamponUmlaut {
  public static void main(String[] args) {
    ByteBuffer b = ByteBuffer.allocate(64);
    b.put("Dünya".getBytes(StandardCharsets.UTF_8));
    b.flip();
    while (b.hasRemaining())
      System.out.print((char) b.get());      // yanlış
    System.out.println();
    b.rewind();
    System.out.println(StandardCharsets.UTF_8.decode(b));  // doğru
  }
}
```

```
D╟EnyaDünya
```

“Dünya” UTF-8'de 6 bayttır; `ü` = `C3 BC`. `(char) b.get()` işaretli `byte` değerini `char`'a çevirir. `C3` bir `byte` olarak −61'dir ve `char`'a çevrilince işaret biti genişler: `U+FFC3`, yani `╟`. `BC` de `E` olur. Metni çözmek için `Charset.decode` kullanılır. `rewind()` konumu yeniden 0'a getirdiği için aynı baytlar bu kez doğru çözülür.

8.10.5 Kod izleme: tampon işlemleri

Aşağıdaki program altı tampon işlemini sırayla kullanır: `put`, `flip`, `get`, `compact`, `rewind` ve `clear`.

```
import java.nio.ByteBuffer;

public class TamponIzle {
  public static void main(String[] args) {
    ByteBuffer b = ByteBuffer.allocate(8);
    b.put((byte) 'J');
    b.put((byte) 'A');
    b.put((byte) 'V');
    b.put((byte) 'A');
    b.flip();
    char c1 = (char) b.get();
    char c2 = (char) b.get();
    b.compact();
    b.put((byte) '!');
    b.flip();
    b.get();
    b.rewind();
    b.clear();
    System.out.println(c1 + "" + c2 + " " + b);
  }
}
```

Adım	Satır	Değişkenler	Yeni çıktı
1	5: `ByteBuffer b = ByteBuffer.allocate(8);`	pos=0, lim=8, cap=8	—
2	6: `b.put((byte) 'J');`	pos=1, lim=8, cap=8	—
3	7: `b.put((byte) 'A');`	pos=2, lim=8, cap=8	—
4	8: `b.put((byte) 'V');`	pos=3, lim=8, cap=8	—
5	9: `b.put((byte) 'A');`	pos=4, lim=8, cap=8	—
6	10: `b.flip();`	pos=0, lim=4, cap=8	—
7	11: `char c1 = (char) b.get();`	pos=1, lim=4, cap=8, c1='J'	—
8	12: `char c2 = (char) b.get();`	pos=2, lim=4, cap=8, c1='J', c2='A'	—
9	13: `b.compact();`	pos=2, lim=8, cap=8, c1='J', c2='A'	—
10	14: `b.put((byte) '!');`	pos=3, lim=8, cap=8, c1='J', c2='A'	—
11	15: `b.flip();`	pos=0, lim=3, cap=8, c1='J', c2='A'	—
12	16: `b.get();`	pos=1, lim=3, cap=8, c1='J', c2='A'	—
13	17: `b.rewind();`	pos=0, lim=3, cap=8, c1='J', c2='A'	—

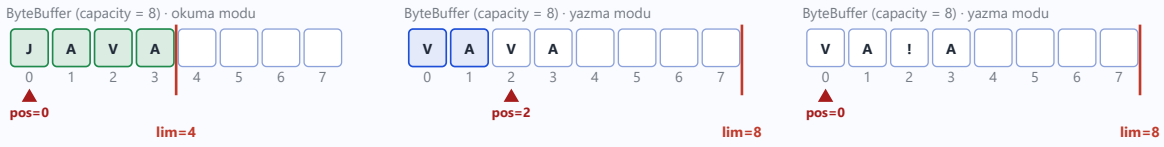
Adım	Satır	Değişkenler	Yeni çıktı
14	18: `b.clear();`	pos=0, lim=8, cap=8, c1='J', c2='A'	—
15	19: `System.out.println(c1 + "" + c2 + " " + b);`	pos=0, lim=8, cap=8, c1='J', c2='A'	`JA java.nio.HeapByteBuffer[pos=0 lim=8 cap=8]`

Tablo 8.22: Tampon işlemlerinin izlemesi; `pos` konum, `lim` sınır, `cap` kapasitedir.

Tablodaki değişkenler tamponun içini göstermez. Her işlemden sonra tamponun sekiz hücresini de yazdıran bir izleme programı bu eksiği giderir (`·` boş hücre):

```
allocate   pos=0 lim=8 [········]
put J      pos=1 lim=8 [J·······]
put A      pos=2 lim=8 [JA······]
put V      pos=3 lim=8 [JAV·····]
put A      pos=4 lim=8 [JAVA····]
flip       pos=0 lim=4 [JAVA····]
get J      pos=1 lim=4 [JAVA····]
get A      pos=2 lim=4 [JAVA····]
compact    pos=2 lim=8 [VAVA····]
put !      pos=3 lim=8 [VA!A····]
flip       pos=0 lim=3 [VA!A····]
get V      pos=1 lim=3 [VA!A····]
rewind     pos=0 lim=3 [VA!A····]
clear      pos=0 lim=8 [VA!A····]
```

İlk beş adımda dört `put` `JAVA`'yı yazar ve konum 4'e ilerler. `flip()` sınırı eski konuma, 4'e çeker; konum 0 olur. Artık yalnız yazılan dört bayt okunabilir. İki `get` `J` ve `A`'yı okur; okunmamış iki bayt kalır.



Şekil 8.28: Tampon işlemlerinden üç an. (a) `flip()`: sınır 4, konum 0. (b) `compact()`: okunmamış `V` ve `A` başa taşınır, konum 2, sınır 8. (c) `clear()`: konum 0, sınır 8; veri silinmez.

`compact()` okunmamış `V` ve `A`'yı tamponun başına kopyalar. Konum kalan öğe sayısına, 2'ye, sınır kapasiteye, 8'e çekilir; tampon yeniden yazma kipine geçer. Dikkat edilecek nokta şudur: 2 ve 3 numaralı hücrelerdeki eski `V` ve `A` silinmez. Hücre içeriği `VAVA`'dır; yeni yazılan veri bu eski baytların üzerine yazılır. Sonraki `put('!')` konum 2'deki eski `V`'nin üzerine yazar: `VA!A`.

İkinci `flip()` sınırı 3'e çeker; okunabilir veri `V`, `A`, `!` 'dir. Bir `get` `V` 'yi okur. `rewind()` konumu 0'a döndürür ama sınırı değiştirmez; aynı veri baştan yeniden okunabilir. `clear()` konumu 0'a, sınırı kapasiteye çeker. Adına rağmen veriyi silmez; tampon yalnız "boş" sayılır ve üzerine yazılır. Son satır `toString()` 'in konum, sınır ve kapasiteyi verdiğini gösterir: `JA java.nio.HeapByteBuffer[pos=0 lim=8 cap=8]` . Tablo 8.23 işlemleri özetler.

Metot	Ne yapar?	Konum	Sınır
`put(x)`	konuma veri yazar	+1	—
`get()`	konumdaki veriyi okur	+1	—
`get(i)`	mutlak indisteki veriyi okur	değişmez	—
`flip()`	yazmadan okumaya geçer	0	eski konum
`rewind()`	aynı veriyi baştan yeniden okumak için	0	değişmez
`clear()`	yeniden yazmaya hazırlar; veriyi silmez	0	kapasite
`compact()`	okunmamış veriyi başa taşır, yazmaya devam eder	kalan sayısı	kapasite
`mark()` / `reset()`	konumu işaretler / işarete döner	— / işaret	—

Tablo 8.23: Tampon işlemlerinin konum ve sınıra etkisi.

Tamponun artısı, aynı bellek alanının tekrar tekrar kullanılmasıdır. Veri üzerinde esnek gezinme ve değiştirme yapılabilir. Eksisi, kullanımın karmaşık olmasıdır: `flip` 'i unutmak en sık hatadır. Tamponun boyutu sonradan değiştirilemez; sınırdan fazla yazma `BufferOverflowException` , sınırdan fazla okuma `BufferUnderflowException` fırlatır:

```
java.nio.HeapByteBuffer[pos=0 lim=5 cap=5]
BC sonra reset: pos=1 get=B
BufferOverflowException
BufferUnderflowException
java.nio.HeapIntBuffer[pos=0 lim=2 cap=3] 7 8
putInt+putShort pos=6
```

Bu çıktının ikinci satırı `mark` ile `reset` 'i gösterir: konum 1'deyken işaret konur, iki okumadan sonra `reset()` konumu yeniden 1'e getirir. Son satırda `putInt` ve `putShort` konumu 4 + 2 = 6 ilerletir. `ByteBuffer` da `DataOutputStream` gibi ilkel tipleri ikili yazabilir.

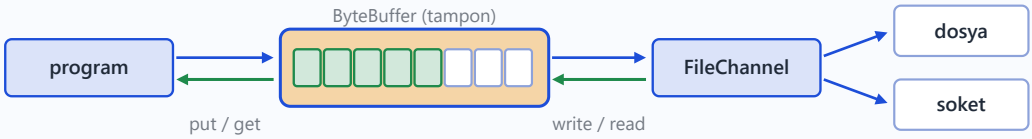
Sık yapılan hata

`flip()` 'i unutmak. `b.put((byte) 'J'); b.put((byte) 'A');` yazıldıktan sonra doğrudan `b.get()` çağrılırsa konum 2'deki boş hücre okunur ve 0 döner. Yazılan veri değil, sonrası okunur. Yazmadan sonra okumadan önce `flip()` şarttır; yeniden yazmadan önce de `clear()` ya da `compact()` . Bir de tip hatası vardır: `ByteBuffer` 'ın `put` metodu `byte` ister ve `b.put('J')` derlenmez, çünkü `char` 'dan `byte` 'a dönüşüm kendiliğinden yapılmaz.

8.10.6 Kanallar

Tanım 8.14

Kanal (*channel*), bir dosya, bir ağ soketi ya da başka bir G/Ç aygıtıyla veri alışverişi yapan bağlantıdır. Kanaldan veri her zaman bir tampona okunur (`read(tampon)`) ve bir tampondan yazılır (`write(tampon)`).



Şekil 8.29: Program tampona `put` eder ve tampondan `get` eder; kanal tamponu dosyaya ya da sokete yazar ve onlardan tampona okur.

Dört temel kanal türü vardır: `FileChannel` dosyada okuma ve yazma, `SocketChannel` TCP ağ soketi üzerinden veri iletimi, `ServerSocketChannel` gelen TCP bağlantılarını kabul etme, `DatagramChannel` UDP soketi üzerinden veri iletimi içindir. Kanallar `FileChannel.open(yol, seçenekler…)` ve `SocketChannel.open()` gibi fabrika metotlarıyla açılır. İş bitince `close()` ile ya da try-with-resources ile kapatılır.

```
public class KanalAcKapa {
  public static void main(String[] args) throws IOException {
    // FileChannel oluşturma
    FileChannel fileChannel = FileChannel.open(
        Paths.get("dosya.txt"), StandardOpenOption.READ);
    // SocketChannel oluşturma
    SocketChannel socketChannel = SocketChannel.open();
    socketChannel.connect(new InetSocketAddress("example.com", 80));
    System.out.println("Bağlandı: " + socketChannel.isConnected());
    // Kanal kapatma
    fileChannel.close();
    socketChannel.close();
  }
}
```

```
Bağlandı: true
```

Bu program iki kanal açar: bir dosya kanalı ve `example.com`'un 80 numaralı portuna bağlanan bir ağ kanalı. `connect` bloklayan kipte bağlantı kurulana kadar bekler. Program bu not için gerçekten çalıştırıldı ve bağlantı kuruldu. Kanalları açık bırakmamak için iki `close()` çağrısı yapılır; bir istisna olur-

sa bu satırlara ulaşılmaz. Bu yüzden try-with-resources daha güvenlidir.

Dosya kanalıyla okuma, tampon kullanımının tipik kalıbını izler:

```
public class KanalOku {
  public static void main(String[] args) throws IOException {
    try (FileChannel channel = FileChannel.open(
        Paths.get("dosya.txt"), StandardOpenOption.READ)) {
      ByteBuffer buffer = ByteBuffer.allocate(1024);
      // Dosyadan veriyi okuma
      int bytesRead = channel.read(buffer);
      System.out.println(bytesRead + " byte okundu");
      // Buffer'ı okuma moduna geçirme
      buffer.flip();
      // Veriyi ekrana yazma
      while (buffer.hasRemaining()) {
        System.out.print((char) buffer.get());
      }
    }
  }
}
```

```
17 byte okundu
NIO kanal ornegi
```

`dosya.txt`'de `NIO kanal ornegi` ve bir satır sonu vardır: 16 + 1 = 17 bayt. `channel.read(buffer)` kanaldan tampona okur ve okunan bayt sayısını döndürür; dosya sonunda −1 döndürür. Okuma tampona bir *yazmadır*; tampondan okumadan önce `flip()` gerekir. Büyük bir dosyada tek `read` dosyanın tamamını okuyamaz; o zaman okuma bir döngüde yapılır ve her turun sonunda tampon `clear()` ya da `compact()` ile boşaltılır.

```
public class KanalYaz {
  public static void main(String[] args) throws IOException {
    try (FileChannel kanal = FileChannel.open(Paths.get("cikti.txt"),
        StandardOpenOption.CREATE, StandardOpenOption.WRITE)) {
      byte[] veri = "Kanal ile yazıldı\n".getBytes(StandardCharsets.UTF_8);
      ByteBuffer tampon = ByteBuffer.wrap(veri);
      int yazilan = kanal.write(tampon);
      System.out.println(yazilan + " byte yazıldı");
    }
  }
}
```

```
20 byte yazıldı
```

`ByteBuffer.wrap(veri)` hazır bir diziyi tampon olarak kullanır. Konum 0, sınır dizinin boyudur; tampon zaten okuma kipinde olduğu için `flip()` gerekmez. Metin UTF-8'de 20 bayttır: 17 karakter, `ı`'ların her biri için birer bayt daha ve satır sonu. `CREATE` seçeneği dosya yoksa onu oluşturur. Dosya varsa ve daha uzunsa `CREATE` ile `WRITE` onu kısaltmaz; yeni baytlar başa yazılır, eski içeriğin sonu kalır. Dosyayı sıfırlamak için `TRUNCATE_EXISTING` eklenir.

Dosya kanalı çift yönlüdür. Aynı kanal hem okuyup hem yazabilir ve `position` ile istenen konuma gidebilir; bu yönüyle `RandomAccessFile`'a benzer:

```
import java.io.IOException;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.file.*;
public class KanalCift {
  public static void main(String[] args) throws IOException {
    try (FileChannel k = FileChannel.open(Path.of("cift.dat"), StandardOpenOption.CREATE,
        StandardOpenOption.READ, StandardOpenOption.WRITE,
StandardOpenOption.TRUNCATE_EXISTING)) {
      k.write(ByteBuffer.wrap("ABCDEF".getBytes()));
      System.out.println("yazdıktan sonra konum " + k.position() + ", boyut " + k.size());
      k.position(2);
      ByteBuffer t = ByteBuffer.allocate(3);
      int n = k.read(t);
      t.flip();
      System.out.println(n + " bayt: " + new String(t.array(), 0, t.limit()));
      ByteBuffer u = ByteBuffer.allocate(10);
      System.out.println("sonda read: " + k.read(u));
    }
  }
}
```

```
yazdıktan sonra konum 6, boyut 6
3 bayt: CDE
sonda read: 1
```

Ağ kanalı da aynı kalıpla çalışır. Aşağıdaki parça bir `main` gövdesidir: bir web sunucusuna HTTP isteği gönderir ve cevabın ilk satırını yazar.

```
try (SocketChannel socketChannel = SocketChannel.open()) {
  socketChannel.connect(new InetSocketAddress("example.com", 80));
  ByteBuffer buffer = ByteBuffer.allocate(1024);
  buffer.put(("GET / HTTP/1.1\r\nHost: example.com\r\n"
      + "Connection: close\r\n\r\n").getBytes(StandardCharsets.US_ASCII));
  buffer.flip();
  // Veriyi sokete yazma
  socketChannel.write(buffer);
  // Cevabı okuma
  buffer.clear();
  socketChannel.read(buffer);
  buffer.flip();
  String cevap = StandardCharsets.US_ASCII.decode(buffer).toString();
  System.out.println(cevap.substring(0, cevap.indexOf("\r\n")));
} catch (IOException e) {
  e.printStackTrace();
}
```

```
HTTP/1.1 200 OK
```

İstek önce tampona `put` edilir, sonra `flip` edilip kanala yazılır. Cevap için tampon `clear()` ile boşaltılır, kanaldan okunur ve yeniden `flip` edilir. HTTP/1.1 isteklerinde `Host` başlığı zorunludur. Cevabın ilk satırı `HTTP/1.1 200 OK`'dir; bu satır not hazırlanırken programın gerçek çıktısıdır. Tek `read` çağrısı cevabın tamamını almayabilir. Gerçek bir istemci, sunucu bağlantıyı kapatıp `read` −1 döndürene kadar okumaya devam eder.

8.10.7 `Path` ve `Files`

NIO.2'nin `Path` sınıfı `File`'ın modern karşılığıdır. `Files` sınıfı ise yaygın dosya işlerini tek çağrıya indiren `static` metotlar içerir.

```
public class DosyalarNio {
  public static void main(String[] args) throws IOException {
    Path yol = Path.of("veri", "notlar.txt");      // Java 11+
    Files.createDirectories(yol.getParent());
    Files.writeString(yol, "Ali 70\nAyşe 85\nCan 90\n");
    System.out.println(yol.getFileName() + ": " + Files.exists(yol));
    System.out.println("Boyut: " + Files.size(yol) + " byte");
    List<String> satirlar = Files.readAllLines(yol);
    for (String satir : satirlar) {
      System.out.println(satir.toUpperCase(Locale.ROOT));
    }
    Files.write(yol, List.of("Deniz 60"), StandardOpenOption.APPEND);
    System.out.println(Files.readAllLines(yol).size() + " satır");
  }
}
```

```
notlar.txt: true
Boyut: 23 byte
ALI 70
AYŞE 85
CAN 90
4 satır
```

`Path.of("veri", "notlar.txt")` yolu parçalardan kurar ve sistemin ayırıcısını kendisi koyar. `createDirectories` ara klasörlerle birlikte klasörü oluşturur. `writeString` metni UTF-8 ile tek çağrıda yazar: `Ayşe`'deki `ş` iki bayt olduğu için 22 karakter 23 bayt tutar. `readAllLines` bütün dosyayı satır listesi olarak okur. `Files.write` ile `APPEND` seçeneği yeni bir satırı sona ekler ve dosya 4 satır olur.

Programdaki `toUpperCase(Locale.ROOT)` çağrısı bilinçlidir. `toUpperCase()` parametresiz çağrıldığında bilgisayarın yerel ayarını kullanır. Türkçe yerel ayarda küçük `i`'nin büyüğü `İ`'dir; program Türkçe bir sistemde `ALİ 70`, İngilizce bir sistemde `ALI 70` yazardı. `Locale.ROOT` (`java.util.Locale`) dönüşümü dilden bağımsız kılar ve çıktı her makinede aynı olur. Dile bağlı olmaması gereken dönüşümlerde, örneğin dosya adlarında ya da anahtar sözcüklerde, bu biçim kullanılır. Fark Türkçe yerel ayarda şöyle görünür:

```
tr: ALİ 70 / ROOT: ALI 70 / tıtle
```

Metot	Görevi
`Path.of(…)` / `Paths.get(…)`	yol oluşturur (`File`'ın modern karşılığı)
`Files.readAllLines`, `Files.readString`	bütün dosyayı tek çağrıda okur (UTF-8)
`Files.write`, `Files.writeString`	tek çağrıda yazar

Metot	Görevi
`Files.exists`, `size`, `copy`, `move`, `delete`	dosya sistemi işlemleri

Tablo 8.24: `Path` ve `Files`'ın sık kullanılan metotları.

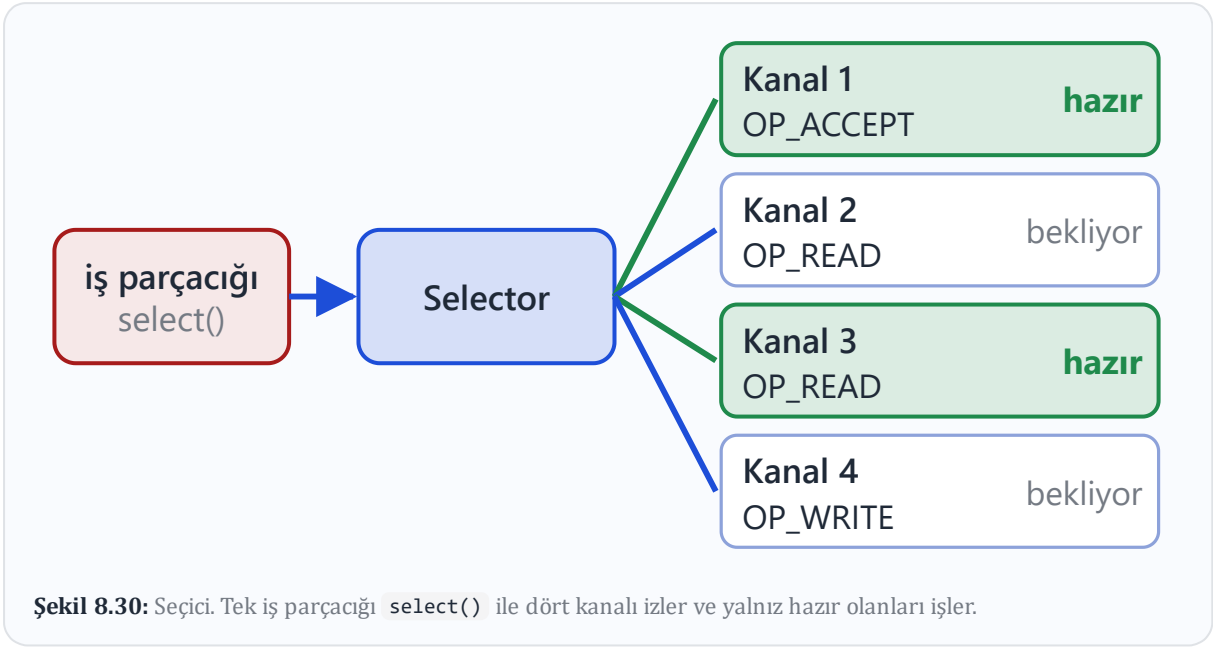
`Files`'ın metotları `File`'ınkilerden bir yönüyle daha güvenlidir: başarısızlığı `false` döndürerek değil, açıklayıcı bir istisnayla bildirir. `Files.delete` dolu bir klasörü silemezse `DirectoryNotEmptyException` fırlatır. `readAllLines` ve `readString` bütün dosyayı belleğe aldığı için küçük ve orta boy dosyalar içindir; çok büyük dosyalar `BufferedReader` ile satır satır okunur.

8.10.8 Seçici

Tanım 8.15

Seçici (*selector*), birden çok kanalı tek bir iş parçacığında izleyen NIO nesnesidir. Kanallar bloklamayan kipe alınıp seçiciye kaydedilir. `select()` en az bir kanal hazır olana kadar bekler ve hazır kanal sayısını döndürür.

Seçici `Selector.open()` ile oluşturulur. Kanal `configureBlocking(false)` ile bloklamayan kipe alınır ve `channel.register(selector, olaylar)` ile kaydedilir. Kanalın hangi olayları beklediği `SelectionKey` sabitleriyle belirlenir (Tablo 8.25).



Şekil 8.30: Seçici. Tek iş parçacığı `select()` ile dört kanalı izler ve yalnız hazır olanları işler.

Sabit	Anlamı
`OP_READ`	kanaldan okuma yapılabilir
`OP_WRITE`	kanala yazma yapılabilir

Sabit	Anlamı
`OP_CONNECT`	bağlantı kurma tamamlandı
`OP_ACCEPT`	yeni bir bağlantı kabul edilebilir

Tablo 8.25: `SelectionKey` olay sabitleri.

Aşağıdaki iskelet bir sunucunun seçici döngüsüdür. Döngü sonsuzdur; program bu yüzden yalnız derlenerek denetlendi.

```
public class SeciciGoster {
  public static void main(String[] args) throws IOException {
    ServerSocketChannel kanal = ServerSocketChannel.open();
    Selector secici = Selector.open();
    kanal.bind(new InetSocketAddress(8080));
    kanal.configureBlocking(false);
    kanal.register(secici, SelectionKey.OP_ACCEPT);
    while (true) {
      int hazirKanallar = secici.select();
      if (hazirKanallar > 0) {
        Set<SelectionKey> secilenler = secici.selectedKeys();
        for (SelectionKey secilen : secilenler) {
          if (secilen.isAcceptable()) {
            // Bağlantı kabul
          } else if (secilen.isReadable()) {
            // Veri okuma
          }
        }
        secilenler.clear(); // İşlenen anahtarları temizle
      }
    }
  }
}
```

Sunucu kanalı 8080 portuna bağlanır, bloklamayan kipe alınır ve `OP_ACCEPT` için kaydedilir. Döngüde `select()` hazır kanal sayısını, `selectedKeys()` hazır anahtarları verir. İşlenen anahtarlar `clear()` ile temizlenmezse sonraki turda yeniden gelir. Gerçek bir sunucuda kabul edilen istemci kanalı da `OP_READ` ile kaydedilir ve veri okunur. İki kural kayıt sırasında denetlenir. Bloklayan bir kanal kaydedilemez; sunucu kanalı da yalnız `OP_ACCEPT` olayını destekler:

```
bloklayan kanal kaydı: IllegalBlockingModeException
sunucu kanalında OP_READ: IllegalArgumentException
validOps == OP_ACCEPT: true
```

8.10.9 NIO'nun artıları ve eksileri

NIO'nun artıları üç başlıkta toplanır. **Başarım:** veri bloklar hâlinde aktarılır; büyük dosyalarda klasik G/Ç'den verimlidir. **Esneklik:** tampon ve kanallarla esnek veri işleme yapılır; kanallar çift yönlüdür ve dosya ile ağ aynı yöntemle işlenir. **Çoklu bağlantı:** seçiciyle birçok kanal tek iş parçacığında izlenir; yalnız etkin kanallar işlenir.

Eksileri ise karmaşıklıktır. Klasik G/Ç'ye göre kod daha uzun ve daha zordur. Tampon durumu ve yarım okuma, kopan bağlantı gibi hata durumları elle yönetilir. Seçicinin öğrenilmesi de zaman alır, çünkü çok sayıda kanalın durumunu doğru izlemek gerekir. Kural olarak tek bir dosyayı satır satır okumak için `BufferedReader` ya da `Files` yeterlidir. Kanallar çok büyük dosyalar ve çok sayıda ağ bağlantısı için öne çıkar.

8.11 Örnek problemler ve alıştırmalar

Örnek 8.2

12 ve 345 önce metin, sonra ikili olarak yazılıyor. Program hangi iki sayıyı yazdırır?

```
try (PrintWriter pw = new PrintWriter("s.txt")) {
  pw.print(12);
  pw.print(' ');
  pw.print(345);
}
System.out.println(new File("s.txt").length());
try (DataOutputStream d = new DataOutputStream(new FileOutputStream("s.dat"))) {
  d.writeInt(12);
  d.writeInt(345);
}
System.out.println(new File("s.dat").length());
```

`PrintWriter` sayıları karakterlere çevirir: `"12 345"` , altı karakter. Karakterlerin hepsi ASCII olduğu için dosya 6 bayttır. `DataOutputStream` ise iki `int` yazar: $2 \cdot 4 = 8$ bayt. Çıktı **6** ve **8**'dir. Burada metin kısa çıkmıştır, çünkü sayılar az basamaklıdır.

Örnek 8.3

`a.txt` başta yok. Program ne yazdırır?

```
try (FileWriter w = new FileWriter("a.txt")) {
  w.write("ab");
}
try (FileWriter w = new FileWriter("a.txt")) {
  w.write("cd");
}
try (FileWriter w = new FileWriter("a.txt", true)) {
  w.write("ef");
}
try (BufferedReader r = new BufferedReader(new FileReader("a.txt"))) {
  System.out.println(r.readLine());
}
```

Birinci `FileWriter` dosyayı oluşturur ve `ab` yazar. İkinci `FileWriter` ekleme kipinde değildir; dosyayı sıfırlar, `ab` silinir ve `cd` yazılır. Üçüncüsü `true` ile sona ekler. Dosyada tek satır vardır ve çıktı `cdef` 'tir.

Örnek 8.4

Program üç satırda ne yazdırır?

```
ByteBuffer b = ByteBuffer.allocate(10);
b.put((byte) 1).put((byte) 2).put((byte) 3);
b.flip();
b.get();
System.out.println(b.position() + " " + b.limit() + " " + b.remaining());
b.compact();
System.out.println(b.position() + " " + b.limit());
b.clear();
System.out.println(b.position() + " " + b.limit() + " " + b.get(0));
```

Üç `put` 'tan sonra konum 3'tür. `flip` konumu 0'a, sınırı 3'e çeker. Bir `get` konumu 1'e ilerletir; kalan 3 − 1 = 2'dir. İlk satır `1 3 2`. `compact` okunmamış 2 ve 3'ü başa taşır; konum kalan sayısı olan 2, sınır kapasite olan 10 olur. İkinci satır `2 10`. `clear` konumu 0'a, sınırı 10'a çeker ama veriyi silmez; `get(0)` başa taşınmış olan 2'yi verir. Üçüncü satır `0 10 2`.

Örnek 8.5

Program ne yazdırır?

```
try (RandomAccessFile f = new RandomAccessFile("r.dat", "rw")) {
  for (int i = 1; i <= 4; i++) f.writeInt(i * 10);
  f.seek(4);
  int a = f.readInt();
  f.seek(f.length() - 4);
  int b = f.readInt();
  System.out.println(a + " " + b + " " + f.getFilePointer());
}
```

Dosyada dört kayıt vardır: 10, 20, 30, 40; her biri 4 bayt, toplam 16 bayt. `seek(4)` ikinci kaydın başıdır ve `a` = 20 olur. `length() - 4` = 12, son kaydın başıdır; `b` = 40. Okumadan sonra işaretçi 16'dadır, yani dosyanın sonunda. Çıktı `20 40 16` 'dır.

Sunumdaki dört alıştırmanın çözümleri de JDK 21 ile çalıştırılarak doğrulanmıştır. Birincisi, "Java ile dosya / okuma ve yazma / (boş satır) / çok kolay" içeriğindeki `metin.txt` 'nin satır, kelime ve karakter sayısını bulur:

```
int satirSayisi = 0, kelimeSayisi = 0, karakterSayisi = 0;
try (BufferedReader br = new BufferedReader(new FileReader("metin.txt"))) {
  String satir;
  while ((satir = br.readLine()) != null) {
    satirSayisi++;
    karakterSayisi += satir.length();
    String temiz = satir.trim();
    if (!temiz.isEmpty()) kelimeSayisi += temiz.split("\\s+").length;
  }
} catch (IOException e) {
  System.err.println("Hata: " + e.getMessage());
}
System.out.println(satirSayisi + " satır, " + kelimeSayisi + " kelime, "
    + karakterSayisi + " karakter");
```

```
4 satır, 8 kelime, 37 karakter
```

Boş satır için `trim()` ve `isEmpty()` denetimi gereklidir. `"".split("\\s+")` uzunluğu 1 olan bir dizi döndürür; denetim olmasaydı boş satır bir kelime sayılırdı. Karakterler 14 + 14 + 0 + 9 = 37'dir; satır sonlarını `readLine` attığı için onlar sayılmaz.

İkinci alıştırma klavyeden `n` okur, 1'den `n` 'ye kadar sayıları ve karelerini `kareler.txt` 'ye yazar, sonra dosyayı okuyup karelerin toplamını bulur:

```
Scanner klavye = new Scanner(System.in);
try (PrintWriter pw = new PrintWriter(new FileWriter("kareler.txt"))) {
  int n = klavye.nextInt();
  for (int i = 1; i <= n; i++) {
    pw.println(i + " " + i * i);
  }
}
int toplam = 0;
try (Scanner dosya = new Scanner(new File("kareler.txt"))) {
  while (dosya.hasNextInt()) {
    dosya.nextInt();               // i
    toplam += dosya.nextInt();     // i * i
  }
}
System.out.println("Kareler toplamı: " + toplam);
```

```
Kareler toplamı: 55
```

Yazma bloğu kapandıktan sonra okuma başlar; aksi hâlde veri tamponda kalır ve dosya boş okunurdu. `n` = 5 için 1 + 4 + 9 + 16 + 25 = 55. Üçüncü alıştırma bir `int` dizisini serileştirir ve geri okur. Diziler serileştirilebilir:

```
int[] dizi = {3, 1, 4, 1, 5};
try (ObjectOutputStream c = new ObjectOutputStream(new FileOutputStream("dizi.ser"))) {
  c.writeObject(dizi);
}
try (ObjectInputStream g = new ObjectInputStream(new FileInputStream("dizi.ser"))) {
  int[] okunan = (int[]) g.readObject();
  int toplam = 0;
  for (int x : okunan) toplam += x;
  System.out.println(okunan.length + " eleman, toplam " + toplam);
}
```

```
5 eleman, toplam 14
```

Dördüncü alıştırmada beş not `RandomAccessFile` ile `int` olarak yazılır ve verilen sıra numarasındaki not dosyanın tamamı okunmadan bulunur: `f.seek((sira - 1) * 4); int not = f.readInt();`. Üçüncü notun konumu 8'dir. Aynı yaklaşım 10.000 satırlık bir kayıt dosyasının son kaydı için de geçerlidir, ama yalnız kayıtlar sabit boyluysa: `seek((n - 1) * boy)`. Satır boyları farklıysa konum hesaplanamaz. Ya dosya baştan sıralı okunur ya da sondan geriye doğru `\n` aranır.

8.12 Sık yapılan hatalar

Bölüm boyunca yer yer anılan hatalar, derleyicinin ya da JVM'in gerçek iletileriyle birlikte burada toplanır.

Kontrol edilen istisnayı ele almamak. `new FileReader("a.txt")` `FileNotFoundException`, `readLine()` `IOException` fırlatabilir. Kod `try-catch` içinde olmalı ya da metot `throws IOException` demelidir:

```
Ana.java:5: error: unreported exception FileNotFoundException; must be caught or declared
to be thrown
    BufferedReader r = new BufferedReader(new FileReader("a.txt"));
                                          ^
Ana.java:6: error: unreported exception IOException; must be caught or declared to be
thrown
    String s = r.readLine();
                         ^
2 errors
```

import'u unutmak. `Scanner` `java.util`'de, G/Ç sınıfları `java.io`'dadır. `import` yazılmazsa derleyici sınıfı bulamaz:

```
Ana.java:3: error: cannot find symbol
    Scanner sc = new Scanner(System.in);
    ^
  symbol:   class Scanner
  location: class Ana
Ana.java:3: error: cannot find symbol
    Scanner sc = new Scanner(System.in);
                     ^
  symbol:   class Scanner
  location: class Ana
2 errors
```

Öteki hataların her biri ilgili kesimde gerçek çıktısıyla gösterildi: Windows yolunda tek ters bölü yazmak, `finally`'de `null` denetimi yapmamak, `nextInt`'ten sonra `nextLine` kullanmak, tamponlu yazıcıyı kapatmamak, `flip()`'i ya da `Serializable`'ı unutmak, `read()`'in sonucunu `byte`'ta tutmak ve blok kopyada `write(parca)` yazmak. Bunların çoğunda program derlenir ve çalışır, ama yanlış sonuç verir. Bu yüzden G/Ç kodu yazıldıktan sonra kopyanın boyu, okunan değerler ve dosyanın gerçek içeriği mutlaka denetlenir.

8.13 Özet

- Akış, veriyi kaynaktan hedefe sırayla taşıyan yapıdır. Her Java programının üç standart akışı vardır: `System.in`, `System.out` ve `System.err`. Çıktı yönlendirildiğinde `System.err` ekranda kalır.
- `Scanner` klavyeyi, dosyaları ve dizgileri okur. `nextInt()` satır sonunu tamponda bırakır; ardından gelen `nextLine()` boş dizgi döndürür.
- Her dosya bir bayt dizisidir. Metin dosyasında baytlar bir karakter kodlamasıyla harflere çevrilir; ikili dosyada sayı, piksel ya da komut olarak yorumlanır.
- ASCII 7 bit, ISO-8859-9 Türkçe için 8 bittir; UTF-8 bir karakteri 1–4 baytla yazar ve Java 18'den beri varsayılandır. Aynı dosya yazıldığı kodlamayla okunmalıdır.
- 127 metin olarak 3 bayt, `int` olarak 4 bayttır. İkili boyut tipe, metin boyutu basamak sayısına bağlıdır. İkili veri yazıldığı sırayla ve tiple geri okunur.
- `File` yalnız bir yoldur, içeriği okumaz. Metotlarının çoğu hatayı `false` döndürerek bildirir.
- Bayt akışları (`InputStream` / `OutputStream`) her tür veri, karakter akışları (`Reader` / `Writer`) metin içindir. Sarmalayıcı akışlar yetenek ekler; en dıştakini kapatmak hepsini kapatır.
- `readLine()` dosya sonunda `null`, `read()` −1 döndürür. `read()`'in sonucu `int`'te tutulur.
- `FileWriter(ad, true)` sona ekler; parametresiz hâli dosyayı sıfırlar. `PrintWriter` istisna fırlatmaz, hata `checkError()` ile görülür.
- Tamponlama disk erişimlerini toplu yapar. Yazılan veri `flush()` ya da `close()`'a kadar bellekte bekleyebilir.
- try-with-resources kaynakları açılış sırasının tersiyle ve `catch`'ten önce kapatır.
- Blok kopyada yalnız okunan `n` bayt yazılır: `write(parca, 0, n)`. Kısa yollar `transferTo` ve `Files.copy`'dir.
- `RandomAccessFile` `seek` ile istenen konuma atlar; sabit boylu kayıtlarda k. kaydın konumu $k \cdot$ kayıt boyudur.
- Serileştirme nesneyi baytlara çevirir. Sınıf `Serializable`'ı uygular; `static` ve `transient` alanlar yazılmaz; geri okumada yapıcı çağrılmaz.
- NIO tampon, kanal ve seçiciyle çalışır. Tampona yazdıktan sonra okumadan önce `flip()`, yeniden yazmadan önce `clear()` ya da `compact()` çağrılır.

8.14 Kavram listesi

akış *stream* · standart girdi / çıktı *standard input / output* · metin dosyası / ikili dosya *text file / binary file* · karakter kodlaması *character encoding* · bayt akışı / karakter akışı *byte stream / character stream* · sarmalayıcı akış *decorator* · tampon / tamponlama *buffer / buffering* · try-with-resources · kontrol edilen istisna *checked exception* · rastgele erişimli dosya *random access file* · serileştirme *serialization*

kanal *channel* · seçici *selector* · yerel ayar *locale* · iş parçacığı *thread*

8.15 Kendini Dene

1. `"Ilık"` kelimesi UTF-8 ile ve ISO-8859-9 ile kaç bayt tutar?

   Cevap

   UTF-8'de `I`, `l` ve `k` birer, `ı` iki bayttır: **5**. ISO-8859-9'da her harf bir bayttır: **4**.

2. Bir `PrintWriter` ile `print(5L)`, `print('A')`, `print(1.5f)`; bir `DataOutputStream` ile `writeLong(5)`, `writeChar('A')`, `writeFloat(1.5f)` yapılıyor. İki dosya kaç bayttır?

   Cevap

   Metin dosyasında `5A1.5` yazar: **5** bayt. İkili dosyada `long` 8, `char` 2, `float` 4 bayttır: **14** bayt. `print(1.5f)` yerel ayardan bağımsızdır ve nokta yazar.

3. 10.000 baytlık bir dosya 4096 baytlık bir diziyle `read(dizi)` çağrılarak okunuyor. Çağrılar hangi değerleri döndürür? Kopyada `write(dizi)` kullanılsaydı kopya kaç bayt olurdu?

   Cevap

   4096, 4096, 1808 ve dosya sonunda −1. `write(dizi)` her turda dizinin tamamını yazar: 3 · 4096 = **12.288** bayt. Son turda 2288 eski bayt fazladan yazılır.

4. `ByteBuffer t = ByteBuffer.allocate(5);` sonra `put` ile 1, 2, 3 yazılıyor; `flip()`, iki `get()`, `compact()`, `put((byte) 9)` ve yeniden `flip()` yapılıyor. Konum, sınır, kalan ve 0 ile 1 numaralı hücreler nedir?

   Cevap

   İki `get`'ten sonra okunmamış tek öğe 3'tür. `compact` onu başa taşır: konum 1, sınır 5. `put(9)` 1 numaralı hücreye yazar, konum 2 olur. `flip` sınırı 2'ye, konumu 0'a çeker. Sonuç: konum **0**, sınır **2**, kalan **2**, hücreler **3** ve **9**.

5. Boş bir dosyaya `RandomAccessFile` ile 7, 8, 9 `int` olarak yazılıyor. Sonra `seek(4); writeInt(100); seek(length()); writeShort(1);` yapılıyor. Dosyanın uzunluğu nedir ve

`seek(0)` 'dan sonra `readShort()` ne döndürür?

> **Cevap**
>
> Üç `int` 12 bayttır; `writeInt(100)` 8'in yerine yazılır ve uzunluk değişmez. `writeShort` sona 2 bayt ekler: **14**. `readShort()` ilk iki baytı okur. 7'nin baytları `00 00 00 07` olduğu için sonuç **0**'dır.

6. Yapıcısı "kuruldu" yazan, `transient String not = "geçici";` ve `int n = 5;` alanları olan serileştirilebilir bir sınıfın nesnesi oluşturuluyor, `n` 8 yapılıyor, nesne yazılıp geri okunuyor. "kuruldu" kaç kez yazılır ve okunan nesnede `not` ile `n` nedir?

> **Cevap**
>
> **Bir kez**; `readObject` yapıcıyı çağırmaz. `not` `null` 'dır, çünkü `transient` alan yazılmaz ve alan ilklendiricisi de çalışmaz. `n` **8**'dir.

7. `try (R a = new R("A"); R b = new R("B"); R c = new R("C")) { … }` bloğu istisnasız biter; `R` açılırken ve kapanırken ileti yazıyor. İletilerin sırası nedir?

> **Cevap**
>
> **A açıldı, B açıldı, C açıldı**, gövde, **C kapandı, B kapandı, A kapandı**. Kaynaklar soldan sağa açılır ve ters sırayla kapanır.

8. 3 MB'lık bir dosya varsayılan `BufferedInputStream` ile bayt bayt okunuyor. Alttaki `FileInputStream` 'den kaç okuma yapılır?

> **Cevap**
>
> 3 · 1.048.576 / 8192 = 384 veri taşıyan okuma ve dosya sonu için bir okuma: **385**.

8.16 İleri okuma

- Oracle, *The Java Tutorials*, "Basic I/O": akışları, `Scanner` ve biçimlendirmeyi, NIO.2 ile dosya işlemlerini kısa örneklerle anlatan resmî ve ücretsiz kaynaktır.
- C. S. Horstmann, *Core Java, Volume II: Advanced Features* (12. baskı), Bölüm 2 "Input and Output": akış sınıflarını, karakter kodlamalarını, serileştirmeyi ve bellek eşlemeli dosyaları ayrıntılı işler.
- Oracle, *Java Object Serialization Specification*: serileştirilmiş baytların biçimini, `serialVersionUID` 'nin nasıl hesaplandığını ve sürüm uyumluluğu kurallarını tanımlar.
- OpenJDK, JEP 400 "UTF-8 by Default": Java 18'de varsayılan kodlamanın neden UTF-8 yapıldığını ve eski programlara etkisini açıklar.

8.17 Kaynaklar

1. Oracle, *The Java Tutorials*: “Basic I/O” (I/O Streams, File I/O featuring NIO.2), “Scanning and Formatting”.
2. Oracle, *Java SE 21 API Documentation*: `java.io`, `java.nio`, `java.nio.channels`, `java.nio.file` paketleri.
3. Oracle, *Java Object Serialization Specification*.
4. J. Gosling ve diğerleri, *The Java Language Specification, Java SE 21 Edition* (3.10.6 kaçış dizileri, 14.20.3 try-with-resources).
5. P. Deitel ve H. Deitel, *Java: How to Program, Early Objects*, 11. baskı, Pearson, 2017 (Bölüm 15: Files, Input/Output Streams, NIO and XML Serialization).
6. C. S. Horstmann, *Core Java, Volume II: Advanced Features*, 12. baskı, 2022 (Bölüm 2: Input and Output).
7. K. Sierra ve B. Bates, *Head First Java*, 3. baskı, O'Reilly, 2022 (Bölüm 16: Saving Objects (and Text)).
8. A. B. Downey ve C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019.
9. OpenJDK, *JEP 400: UTF-8 by Default*, JDK 18, 2022. https://openjdk.org/jeps/400

KISIM IV · BÖLÜM 9

Grafikler ve Kullanıcı Arayüzleri

Şimdiye kadarki programlar konsolda çalıştı: soruyu program sordu, kullanıcı klavyeden yanıt verdi, sonuç alt alta yazıldı. Gündelik hayatta kullanılan programlar ise pencere, düğme, metin alanı ve çizimlerle çalışır. Bu not Java'da grafik kullanıcı arayüzü kurmayı anlatır. Önce olay güdümlü programın konsol programından farkı, AWT ve Swing kütüphaneleri ve ekran koordinatları ele alınır. Ardından `JFrame` ve `JPanel` ile pencere kurmak, `paintComponent` ve `Graphics` ile çizmek, `Point`, `Color` ve `Font` sınıfları gelir. Yerleşim yöneticileri bileşenleri pencereye dizer; düzenin piksel piksel nasıl hesaplandığı adım adım izlenir. Fare olayları, dinleyici arayüzleri, adaptör sınıfları ve olay iş parçacığı notun merkezindedir. Not, üç küçük uygulamayla biter: fareyle çizim programı, metin dönüştürücü ve para üstü hesaplayıcı.

Bu bölümün kodu: `github.com/sercankulcu/object-oriented-programming-java/Bolum09/src`

ÖĞRENME ÇIKTILARI

Pencere kurmak
`JFrame` ve `JPanel` ile pencere ve panel oluşturmak, bileşen eklemek, pencereyi göstermek ve kapatmak.

Çizmek
`paintComponent` metodunu geçersiz kılıp `Graphics` ile şekil ve metin çizmek; ekran koordinatlarını ve sınırlayıcı dikdörtgeni kullanmak.

Yerleştirmek
`BorderLayout`, `FlowLayout`, `GridLayout` ve `BoxLayout` ile bileşenleri düzenlemek; bileşenlerin konum ve boyutunu hesaplamak.

Olaylara yanıt vermek
Dinleyici arayüzleri, adaptör sınıfları, anonim sınıflar ve lambda ifadeleriyle fare, düğme ve metin olaylarını işlemek.

Güvenli başlatmak
Arayüzü `SwingUtilities.invokeLater` ile olay iş parçacığında kurmanın nedenini açıklamak; `invokeLater` ile `invokeAndWait` arasındaki farkı izlemek.

Uygulama yazmak
Çizim programı, metin dönüştürücü ve para üstü hesaplayıcıyı okuyup değiştirebilmek.

9.1 Grafik programlama

9.1.1 Bileşen, olay ve görsel oluşturma

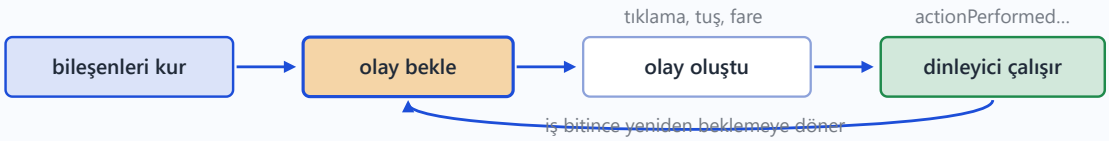
Bir kullanıcı bilgisayarı çoğunlukla pencereler üzerinden kullanır. Pencerenin başlığı, düğmeleri, menüleri ve metin alanları vardır. Kullanıcı fareyle tıklar, klavyeyle yazar, pencereyi büyütür. Program da bu eylemlere yanıt verir. Bu türden bir arayüze grafik kullanıcı arayüzü denir.

Tanım 9.1

Grafik kullanıcı arayüzü (*graphical user interface*, GUI), kullanıcının pencere, düğme, menü, metin alanı ve çizim gibi görsel öğelerle programla etkileştiği arayüzdür.

Java, GUI geliştirmek için kapsamlı bir sınıf kütüphanesi sunar. Bu kütüphaneyle bir GUI programı yazmak için üç kavramı anlamak gerekir.

- **Bileşen** (*component*): Pencere, düğme, etiket ve metin alanı gibi arayüzün yapı taşlarıdır. Her bileşen bir sınıfın nesnesidir. Bir düğme, `JButton` sınıfından oluşturulmuş bir nesnedir; metni, rengi ve boyutu bu nesnenin alanlarında saklanır.
- **Olay işleme** (*event handling*): Fare tıklaması, tuşa basma gibi kullanıcı eylemleri yakalanır ve bunlara yanıt veren metotlar çağrılır.
- **Görsel oluşturma** (*painting*): Şekiller çizilir, ekran güncellenir ve bileşenler görüntülenir.



Şekil 9.1: GUI programının yaşamı. Bileşenler kurulur, program olay bekler, bir olay gelince ilgili dinleyici metot çalışır; ardından program yeniden bekler.

Bir GUI programı, müşteriyi bekleyen bir dükkâna benzer. Önce vitrin hazırlanır: bileşenler oluşturulur ve pencereye yerleştirilir. Sonra program bekler. Müşteri ne zaman neye dokunursa o işle görevli kişi çalışır. Programda vitrinin karşılığı bileşenler, dokunuşun karşılığı olay, görevlinin karşılığı da dinleyici metottur (Şekil 9.1).

9.1.2 Konsol programı ve GUI programı

Konsol programında akış sıralıdır. `main` metodu baştan sona çalışır. Programın bir girdiye ihtiyacı olduğunda soruyu program sorar ve `Scanner` ile klavyeden bir satır okur. Sonuç `System.out.println` ile yazılır. `main` bitince program da biter.

GUI programının akışı başkadır. `main` yalnız arayüzü kurar ve biter. Program bundan sonra kullanıcının hamlesini bekler. Kullanıcı bir düğmeye bastığında ya da panele tıkladığında ilgili metot çalışır. Bu çalışma biçimine olay güdümlü programlama denir.

> **Tanım 9.2**
>
> **Olay** (*event*), kullanıcının tıklama, tuşa basma, fareyi sürükleme gibi bir eylemi ya da sistemden gelen bir bildirimdir. **Olay güdümlü** (*event-driven*) programda akışı koddaki satır sırası değil, olayların geliş sırası belirler.

	Konsol programı	GUI programı
Akış	Sıralı: `main` baştan sona çalışır, soruyu program sorar	Olay güdümlü: program kullanıcının hamlesini bekler
Girdi	`Scanner` ile klavyeden satır okunur	Düğme, metin alanı, fare; dinleyici metotlar çağrılır
Çıktı	`System.out.println`	Bileşenlerin metni değişir ya da panel yeniden çizilir
Bitiş	`main` bitince program biter	`main` bitse de pencere kapanana kadar çalışır
Kütüphane	`java.util`, `java.io`	`java.awt`, `javax.swing`, `java.awt.event`

Tablo 9.1: Konsol programı ile GUI programının karşılaştırması.

Tablodaki son satırların ikisi önemlidir. GUI programında `main` bittiği hâlde program sürer. Bunun nedeni, Java'nın pencere açılınca ayrı bir iş parçacığı başlatmasıdır. Bu iş parçacığı olay kuyruğunu bekler ve gelen her olayı ilgili dinleyiciye iletir. JVM, bu iş parçacığı çalıştıkça kapanmaz.

GUI programında “sonra ne olacak?” sorusunun cevabı koddaki sıra değildir; kullanıcının sıradaki eylemidir. Bu yüzden kodun büyük kısmı “şu olursa şunu yap” biçimindeki dinleyicilerden oluşur. Programcı her eylem için bir yanıt yazar, ama eylemlerin hangi sırayla geleceğini bilmez.

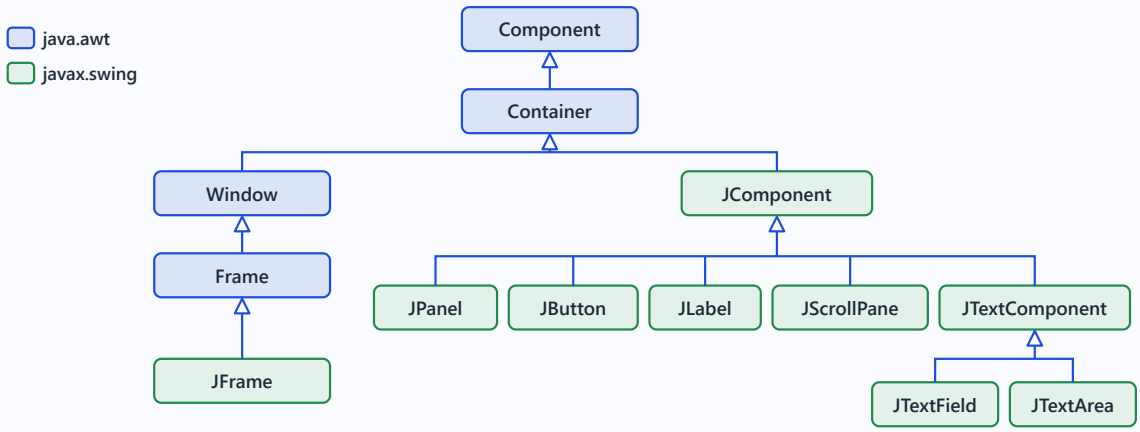
> **Sık yapılan hata**
>
> GUI programında kullanıcı girdisini bir `while` döngüsünde beklemeye çalışmak. Konsol alışkanlığıyla yazılan “kullanıcı düğmeye basana kadar dön” döngüsü olay iş parçacığını meşgul eder ve pencere donar. GUI programında beklemeyi Java yapar; programcı yalnız dinleyiciyi kaydeder.

9.1.3 AWT ve Swing

Java'nın grafik sınıfları iki kütüphaneye dağılmıştır. İkisi de JDK ile gelir; ayrıca bir şey kurmak gerekmez.

- **AWT** (*Abstract Window Toolkit*), `java.awt` paketi: Java'nın ilk grafik kütüphanesidir. Bugün daha çok ortak sınıfları için kullanılır: `Graphics`, `Color`, `Font`, `Point`, yerleşim yöneticileri ve olay sınıfları (`java.awt.event`).
- **Swing**, `javax.swing` paketi: AWT'nin üzerine kurulmuş, bileşenleri Java'nın kendisinin çizdiği kütüphanedir. Sınıf adları `J` ile başlar: `JFrame`, `JPanel`, `JButton`, `JLabel`.

İki kütüphane arasındaki fark bileşenlerin nasıl çizildiğindedir. AWT bir düğme istediğinde işletim sisteminin kendi düğmesini kullanır. Bu yüzden aynı program Windows'ta ve Linux'ta farklı görünür. Swing ise düğmeyi kendisi boyar. Görünüm her sistemde aynıdır ve Java'nın denetimindedir. Bu notta bileşenler Swing'den, çizim ve yardımcı sınıflar AWT'den gelir. Tipik bir GUI programı bu yüzden iki paketi birlikte içe aktarır: `import java.awt.*;` ve `import javax.swing.*;`.



Şekil 9.2: GUI sınıflarının kalıtım ağacı. İçi boş üçgen "türetilmiştir" anlamındadır. `JFrame` bir AWT penceresidir; diğer Swing bileşenleri `JComponent`'ten türer.

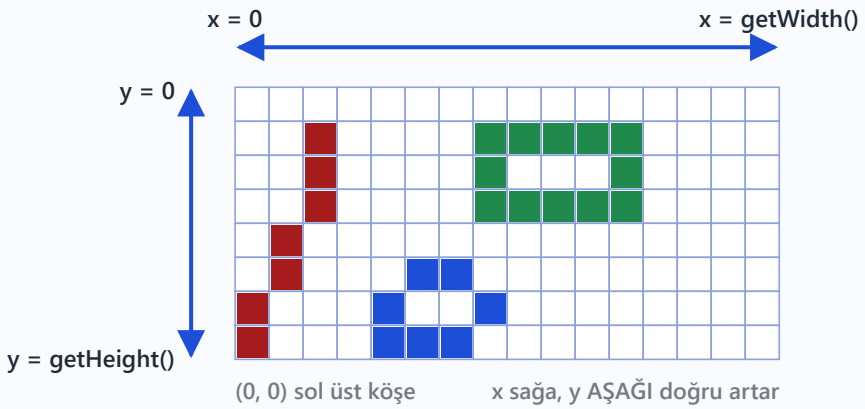
Şekil 9.2, kalıtımın GUI kütüphanesinde nasıl kullanıldığını gösterir. Ağacın kökünde `Component` vardır: ekranda yeri ve boyutu olan her şey bir bileşendir. `Container`, başka bileşenleri içinde tutabilen bileşendir. `Container`'dan iki dal ayrılır. Birinci dalda `Window`, `Frame` ve en sonda `JFrame` bulunur. İkinci dalda Swing bileşenlerinin ortak üst sınıfı `JComponent` vardır. `JPanel`, `JLabel` ve metin bileşenleri doğrudan ya da kısa bir yoldan `JComponent`'ten türer. `JButton` ise araya `AbstractButton` sınıfını alır.

Bu ağacın pratik bir sonucu vardır. Bir metot parametre olarak `Component` bekliyorsa ona düğme de, etiket de, panel de verilebilir. Çok biçimlilik burada gündelik bir araçtır. Örneğin `panel.add(...)` metodu herhangi bir bileşeni kabul eder.

9.1.4 Ekran koordinatları

Grafik bir uygulama üç adımda kurulur. Önce Java'nın grafik sınıflarıyla arayüze bileşenler eklenir. Sonra kullanıcının eylemlerine yanıt veren olay işleme metotları tanımlanır. Son olarak `paint()` ya da `paintComponent()` gibi metotlarla görsel oluşturulur. Çizime geçmeden önce ekranın nasıl adreslendiğini bilmek gerekir.

Ekran bir piksel ızgarasıdır. Her pikselin bir sütun numarası (`x`) ve bir satır numarası (`y`) vardır. Koordinatlar tam sayıdır ve bileşenin sol üst köşesinden ölçülür.



Şekil 9.3: Ekran koordinat sistemi. Orijin sol üst köşededir; `x` sağa, `y` aşağı doğru büyür. Çizgi, dikdörtgen ve elips piksellerden oluşur.

Tanım 9.3

Ekran koordinatları, bir bileşenin içindeki pikseli `(x, y)` tam sayı çiftiyle gösterir. `(0, 0)` bileşenin sol üst pikselidir. `x` sağa, `y` aşağı doğru büyür. Sağ alt piksel `(getWidth() - 1, getHeight() - 1)`'dir.

Matematikteki eksenle en önemli fark `y`'nin yönüdür. Matematikte `y` yukarı doğru büyür; ekranda aşağı doğru. Bu tercih ekranın tarihinden gelir: katot ışınlı tüpler görüntüyü üstten alta, satır satır çiziyordu. Sonuç olarak `y` değeri büyüyen bir nokta ekranda aşağı iner. `y`'yi azaltmak noktayı yukarı taşır.

Örnek 9.1

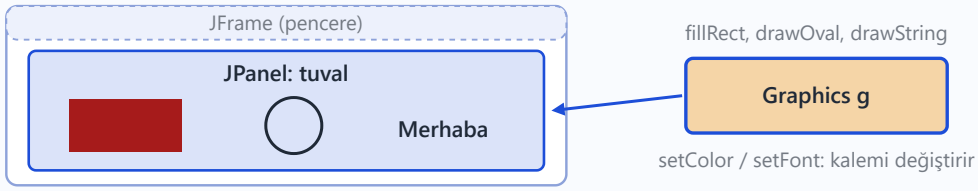
Bir panel 300 piksel genişliğinde ve 160 piksel yüksekliğindedir. Sağ üst, sol alt ve orta pikselin koordinatları nedir?

Sütunlar 0'dan 299'a, satırlar 0'dan 159'a numaralanır. Sağ üst piksel `(299, 0)`, sol alt piksel `(0, 159)` 'dur. Ortadaki piksel için genişlik ve yükseklik ikiye bölünür: `(300 / 2, 160 / 2) = (150, 80)`. Sayılar tek olsaydı tam sayı bölmesi ondalığı atardı; örneğin 261 / 2 = 130 olur.

9.1.5 Tuval, kalem, renk ve yazı tipi

Çizim için üç şey gerekir: üzerine çizilecek bir yüzey, çizimi yapan bir kalem ve kalemin özellikleri.

- **Tuval** (*canvas*): Üzerine resim çizilebilen yüzeydir. Swing'de genellikle `JPanel` kullanılır. AWT'de bu iş için `Canvas` sınıfı vardır. Programcı `JPanel` 'den bir alt sınıf türetir ve çizimi onun içine yapar.
- `Graphics` **sınıfı**: Çizim işlemlerini yapan "kalem"dir. Metotları `drawLine()`, `drawRect()`, `fillOval()`, `drawString()` ve `drawImage()` gibi adlar taşır.
- `Color` **ve** `Font`: Kalemin rengini `Color`, yazı tipini `Font` belirler: `g.setColor(Color.RED)`, `g.setFont(new Font("Serif", Font.BOLD, 24))`.



Şekil 9.4: `JFrame` penceresinin içinde bir `JPanel` tuvali. Çizimi `Graphics` nesnesi yapar.

Kalem benzetmesi bir kuralı da açıklar. `Graphics` nesnesi programcının elindeki kalemdir. `setColor` kalemi değiştirir. Kalem değiştirilene kadar bütün çizimler aynı renkle yapılır. Yazı tipi için de durum aynıdır: `setFont` çağrısından sonraki bütün `drawString` 'ler yeni yazı tipini kullanır. `Graphics` nesnesi bu yüzden durum tutan bir nesnedir: kendi rengini ve yazı tipini hatırlar.

9.1.6 `paintComponent` ile çizim

Bir panele kalıcı çizim yapmanın yolu, `JPanel` 'den türetilen bir sınıfta `paintComponent` metodunu geçersiz kılmaktır.

```
import java.awt.*;
import javax.swing.*;

class CizimPaneli extends JPanel {
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);     // zemini temizle
        g.setColor(Color.BLUE);
        g.drawRect(20, 30, 50, 70);  // x, y, en, boy
        g.drawString("Merhaba, Java Grafik!", 100, 100);
    }
}
```

Sınıf `JPanel`'i genişletir ve üst sınıftaki `paintComponent(Graphics g)` metodunu geçersiz kılar. `@Override` etiketi derleyiciye "bu metot üst sınıftaki bir metodun yerini alıyor" der. Metot adı yanlış yazılırsa derleyici bu etiketi görüp hata verir. Gövdenin ilk satırı `super.paintComponent(g)`'dir: üst sınıfın kendi çizimi, yani zeminin arka plan rengiyle boyanması yapılır. Ardından kalem maviye çevrilir. `drawRect(20, 30, 50, 70)` sol üst köşesi `(20, 30)`'da olan, 50 piksel geniş ve 70 piksel yüksek bir dikdörtgen çerçevesi çizer. Son satır `(100, 100)` noktasına bir metin yazar.



Şekil 9.5: Panel 300 × 160 iken `CizimPaneli`'nin çıktısı. Mavi çerçeve ve aynı renkte metin.

Bu kodda dikkat çeken bir şey vardır: `paintComponent`'i programın hiçbir yeri çağırmaz. Metodu Swing çağırır. Pencere ilk açıldığında, boyutu değiştiğinde, üstündeki başka bir pencere çekildiğinde ya da program `repaint()` ile yeniden çizim istediğinde Swing bu metodu çalıştırır. Çağırırken de hazır bir `Graphics g` nesnesi verir. Bu nesne o anda çizilecek bölgeye bağlıdır.

Metodun gerçekte hangi çizim çağrılarını yaptığını görmek için Swing'in `DebugGraphics` sınıfı kullanılabilir. Bu sınıf, her çizim çağrısını bir satır olarak yazan bir kalemdir. `CizimPaneli` 300 × 160 boyutunda bir resim üzerinde bu kalemle çalıştırıldığında şu çağrılar görülür (kısaltılmış):

```
setColor(238, 238, 238)
fillRect(0, 0, 300, 160)
setColor(0, 0, 255)
drawRect(20, 30, 50, 70)
drawString("Merhaba, Java Grafik!", 100, 100)
```

İlk iki satırı programcı yazmadı. Onlar `super.paintComponent(g)`'nin işidir: kalem panelin arka plan rengine, `(238, 238, 238)` açık griye çevrilir ve bütün panel bu renkle doldurulur. Sonraki üç satır programcının yazdığı çağrılardır.

Tanım 9.4

`paintComponent(Graphics g)`, bir Swing bileşeninin kendi içeriğini çizen metottur. Swing bileşeni yeniden boyaması gerektiğinde bu metodu çağırır. Kendi çizimini yapmak isteyen sınıf metodu geçersiz kılar ve gövdeye önce `super.paintComponent(g)` yazar.

Bir grafik nesnesi `panel.getGraphics()` ile de alınabilir. Ancak bu nesneyle yapılan çizim kalıcı değildir. Pencere bir sonraki kez yeniden çizildiğinde, örneğin küçültülüp büyütüldüğünde, Swing `paintComponent`'i çağırır ve `getGraphics()` ile çizilenler silinir. Kalıcı çizimin yolu `paintComponent`'tir. Çizimin verisi alanlarda saklanır ve metot her çağrıldığında bu veriden yeniden üretilir.

Metnin konumu da bir ayrıntı taşır. `drawString`'in `y` parametresi metnin üst kenarı değil, taban çizgisidir. Taban çizgisi harflerin "oturduğu" çizgidir. `drawString("Merhaba, Java Grafik!", 100, 100)` çağrısında harflerin gövdesi 100. satırın üstünde kalır; yalnız "g", "j" gibi harflerin kuyrukları bu çizginin altına iner.

Sık yapılan hata

`drawString(metin, x, 0)` yazıp metni panelin en üstüne koymayı beklemek. `y = 0` taban çizgisi olduğu için harflerin neredeyse tamamı panelin dışında, görünmez bölgede kalır. Metnin görünmesi için `y` en az yazı tipi yüksekliği kadar olmalıdır; 24 puntoluk bir yazı için 30–40 civarı güvenli bir değerdir.

9.1.7 `Graphics` çizim metotları

`Graphics` sınıfının sık kullanılan metotları Tablo 9.2'de toplanmıştır. Adlarda bir düzen vardır: `draw…` ile başlayan metotlar yalnız çerçeveyi, `fill…` ile başlayanlar içi dolu şekli çizer.

Metot	Ne yapar?
`drawLine(x1, y1, x2, y2)`	İki nokta arasında çizgi

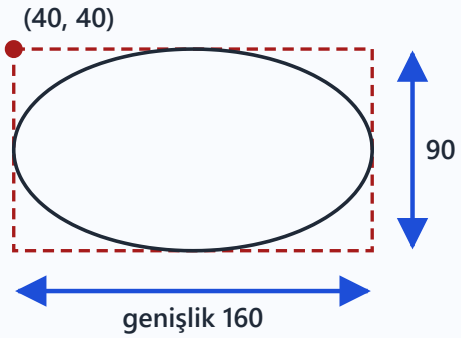
Metot	Ne yapar?
`drawRect(x, y, en, boy)`	Dikdörtgen çerçevesi; `(x, y)` sol üst köşe
`fillRect(x, y, en, boy)`	İçi dolu dikdörtgen
`drawOval(x, y, en, boy)`	Bu dikdörtgene sığan elips (en = boy ise çember)
`fillOval(x, y, en, boy)`	İçi dolu elips ya da daire
`drawString(s, x, y)`	Metin; `y` taban çizgisi
`setColor(c)` / `setFont(f)`	Sonraki çizimlerin rengi ya da yazı tipi
`drawImage(resim, x, y, null)`	Bir resmi `(x, y)` noktasına çizer

Tablo 9.2: `Graphics` sınıfının temel çizim metotları.

Elips ve daireler merkezleriyle değil, sınırlayıcı dikdörtgenleriyle verilir. Elipsin sığdığı dikdörtgenin sol üst köşesi ve boyutu yazılır.

Tanım 9.5

Bir şeklin **sınırlayıcı dikdörtgeni** (*bounding rectangle*), şekli tam içine alan en küçük, kenarları eksenlere paralel dikdörtgendir. `drawOval(x, y, en, boy)` elipsi, sol üst köşesi `(x, y)` ve boyutu `en × boy` olan dikdörtgenin içine çizer.



Şekil 9.6: `drawOval(40, 40, 160, 90)`. Elips, sol üst köşesi `(40, 40)` olan 160 × 90'lık dikdörtgene sığar.

Merkezi `(cx, cy)`, yarıçapı `r` olan bir daire çizmek için sınırlayıcı karenin sol üst köşesi hesaplanır. Kare, merkezden sola ve yukarı `r` kadar uzanır; kenarı `2r`'dir. Sonuç: `fillOval(cx - r, cy - r, 2r, 2r)`. Tersine, verilen bir `drawOval(x, y, en, boy)` çağrısında elipsin merkezi `(x + en / 2, y + boy / 2)`'dir.

Örnek 9.2

`g.drawOval(50, 50, 100, 60)` ile çizilen elipsin merkezi neresidir? Elips hangi piksel aralığını kaplar?

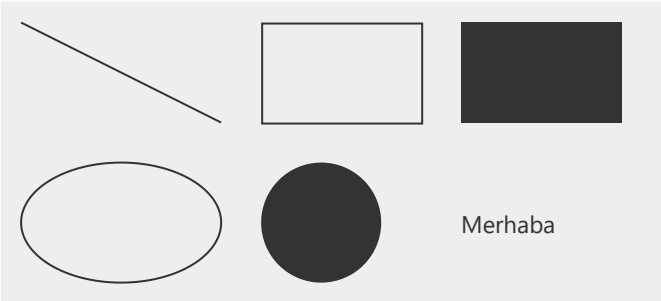
Merkez `(50 + 100 / 2, 50 + 60 / 2) = (100, 80)` 'dir. Elips yatayda 50'den, dikeyde 50'den başlar. Bir resim üzerinde çizilip boyalı pikseller sayıldığında `x` 50 ile 150, `y` 50 ile 110 arasında çıkar. Çerçeve çizen metotlar, sınırın sağ ve alt kenarını da boyar: `draw…` şekli `en + 1` piksel geniş ve `boy + 1` piksel yüksektir. `fill…` ise tam `en × boy` piksel boyar.

Son cümledeki fark küçük ama gerçektir. `drawRect(20, 30, 50, 40)` çağrısında çerçevenin sağ alt köşe pikseli `(70, 70)` 'tir; `(71, 70)` boş kalır. Aynı argümanlarla `fillRect` ise `(69, 69)` 'a kadar boyar. Bir çerçeveyi dolu bir dikdörtgenin üstüne çizerken bu bir piksellik fark görünür hâle gelir.

9.1.8 Kod izleme: şekillerin çizim sırası

Aşağıdaki panel, temel metotların her birini bir kez kullanır.

```
class SekillerPaneli extends JPanel {
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.drawLine(10, 10, 110, 60);
        g.drawRect(130, 10, 80, 50);
        g.fillRect(230, 10, 80, 50);
        g.drawOval(10, 80, 100, 60);
        g.fillOval(130, 80, 60, 60);
        g.drawString("Merhaba", 230, 115);
    }
}
```



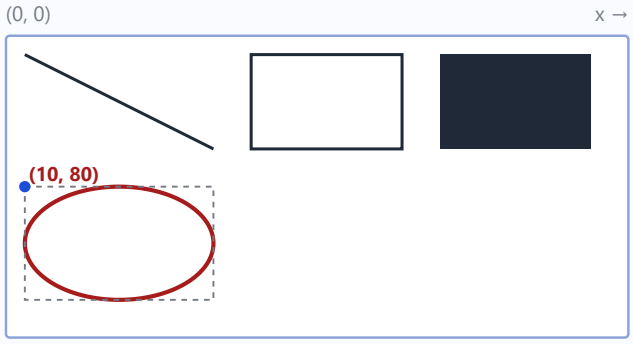
Şekil 9.7: `SekillerPaneli` 'nin çıktısı: çizgi, dikdörtgen çerçevesi, dolu dikdörtgen, elips, dolu daire ve metin.

Kodda hiç `setColor` çağrısı yoktur. Renk verilmediğinde kalem panelin ön plan rengiyle çizer. Swing'in varsayılan görünümünde bu renk `(51, 51, 51)` koyu gridir; çalıştırılan programda `getForeground()` bu değeri döndürür. Tablo 9.3 metodun adımlarını izler.

Adım	Satır	Değişkenler	Açıklama
1	4: `super.paintComponent(g);`	g=Graphics (koyu gri kalem)	Swing bu metodu kendisi çağırır. `super.paintComponent(g)` tuvali arka plan rengiyle **temizler**.
2	5: `g.drawLine(10, 10, 110, 60);`	x1=10, y1=10, x2=110, y2=60	İki nokta arasına çizgi. y aşağı doğru büyüdüğü için çizgi sağa ve **aşağı** iner.
3	6: `g.drawRect(130, 10, 80, 50);`	x=130, y=10, en=80, boy=50	`drawRect` : (x, y) **sol üst köşe**; yalnız çerçeve çizilir, içi boş kalır.
4	7: `g.fillRect(230, 10, 80, 50);`	x=230, y=10, en=80, boy=50	`fillRect` : aynı boyutta, içi kalemin rengiyle dolu dikdörtgen.
5	8: `g.drawOval(10, 80, 100, 60);`	x=10, y=80, en=100, boy=60	`drawOval` : kesikli kutuya **sığan** elips. (10, 80) elipsin değil kutunun köşesi.
6	9: `g.fillOval(130, 80, 60, 60);`	x=130, y=80, en=60, boy=60	En = boy olduğundan dolu bir **daire**: merkez (160, 110), yarıçap 30.
7	10: `g.drawString("Merhaba", 230, 115);`	s="Merhaba", x=230, y=115	`drawString` 'de y, metnin **taban çizgisidir**: harfler 115'in üstüne yazılır.
8	11: `}`	—	Metot biter; pencere her yeniden boyandığında (boyut değişimi, `repaint()`) aynı sırayla **baştan** çizilir.

Tablo 9.3: `SekillerPaneli.paintComponent` izlemesi.

2. adımdaki çizgi `(10, 10)` 'dan `(110, 60)` 'a gider. İkinci noktanın `y` 'si daha büyük olduğu için çizgi sağa ve aşağı iner. 3. ve 4. adımlar aynı boyutta iki dikdörtgen çizer. Biri yalnız çerçevedir, öbürü kalemin rengiyle doludur. 5. adımda `(10, 80)` noktası elipsin değil, sınırlayıcı kutunun köşesidir.



Şekil 9.8: 5. adım. `drawOval(10, 80, 100, 60)` : elips kesikli çizgiyle gösterilen sınırlayıcı kutuya sığar.

6. adımda en ve boy eşittir, bu yüzden dolu bir daire çizilir. Merkez `(130 + 30, 80 + 30) = (160, 110)` , yarıçap 30'dur. 7. adımda metnin taban çizgisi `y = 115` 'tir; harfler bu çizginin üstüne yazılır. Metot bittiğinde panel tamamlanmıştır. Pencere her yeniden boyandığında aynı çağrılar aynı sırayla baştan yapılır.

Çizimler üst üste boyanır. Sonradan çizilen şekil öncekini örter. Bu programda şekiller çakışmadığı için sıra görünmez. Şekiller çakıştığında ise son çizilen üstte kalır.

Sık yapılan hata

`super.paintComponent(g)` satırını unutmak. Bu satır zemini temizler. Yazılmazsa panel arka planıyla boyanmaz; önceki çizimden ya da başka pencerelerden kalan görüntü artıkları ekranda kalabilir. Özellikle hareketli çizimlerde şekil, geçtiği her yerde iz bırakır.

9.2 Pencere ve panel

9.2.1 `JFrame` sınıfı

Bir GUI programının bütün bileşenleri bir pencerenin içinde yaşar. Swing'de pencere `JFrame` sınıfıyla oluşturulur.

Tanım 9.6

`JFrame`, ekranda başlık çubuğu, simge durumuna küçültme ve kapatma düğmeleri olan üst düzey penceredir (*top-level window*). Başka bir bileşenin içine konmaz; diğer bileşenler onun içerik bölmesine eklenir.

Kod	Görevi
`JFrame pencere = new JFrame("Başlık");`	Başlığı verilen pencereyi oluşturur (henüz görünmez)
`pencere.add(bilesen);`	Pencereye (içerik bölmesine) bileşen ekler
`pencere.setSize(400, 300);`	Genişlik ve yükseklik (piksel, kenarlıklar dahil)
`pencere.pack();`	Boyutu içerikteki bileşenlerin tercih ettiği boyuta göre ayarlar
`pencere.setLocation(x, y);`	Pencerenin ekrandaki konumu; `setLocationRelativeTo(null)` ortalar
`pencere.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);`	Kapatılınca programı sonlandırır
`pencere.setVisible(true);`	Pencereyi görünür yapar (genellikle en son çağrılır)

Tablo 9.4: `JFrame` ile sık kullanılan çağrılar.

Pencere boyutunun iki yolu vardır. `setSize` boyutu doğrudan verir; bu boyut başlık çubuğunu ve kenarlıkları da içerir. `pack()` ise içerikteki bileşenlerin tercih ettiği boyutları toplar ve pencereyi tam o kadar büyütür. Bileşenleri yerleşim yöneticisiyle dizilmiş bir pencerede `pack()` genellikle daha iyi sonuç verir, çünkü yazı tipi ve ekran çözünürlüğü değişse de içerik pencereye sığar.

`add` çağrısı bileşeni aslında pencerenin kendisine değil, **içerik bölmesine** (*content pane*) ekler. İçerik bölmesi, başlık çubuğunun altında kalan iç alandır. Eski Java sürümlerinde `pencere.getContentPane().add(...)` yazmak zorunluydu. Java 5'ten beri `pencere.add(...)` bu çağrıyı kendisi yapar. Bu nottaki örnekler iki yazımı da kullanır; anlamları aynıdır.

9.2.2 İlk pencere

```
import javax.swing.JFrame;

public class JFrameOrnegi {
    public static void main(String[] args) {
        JFrame pencere = new JFrame("Java JFrame Örneği");
        pencere.setSize(400, 300);
        pencere.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pencere.setVisible(true);
    }
}
```

Program yalnız `JFrame` sınıfını içe aktarır. `main` başlığı "Java JFrame Örneği" olan bir pencere oluşturur, boyutunu 400 × 300 yapar, kapatma davranışını belirler ve pencereyi görünür kılar. Pencerenin içi boştur. `main` bu dört satırdan sonra biter, ama program pencere kapanana kadar çalışmayı sürdürür.

Bu kısa programda üç tuzak vardır:

- `setVisible(true)` unutulursa pencere bellekte oluşur ama ekranda görünmez. Program hata vermez; yalnız hiçbir şey olmuyormuş gibi görünür.
- `EXIT_ON_CLOSE` verilmezse varsayılan davranış `HIDE_ON_CLOSE`'dur. Kullanıcı × düğmesine basınca pencere gizlenir, ama program arka planda çalışmaya devam eder. Geliştirme ortamında bu, durdurulması unutulan program örnekleri olarak birikir.
- Boyut verilmezse pencere 0 × 0'a yakın, yalnız başlık çubuğu kadar açılır.

`setDefaultCloseOperation`'ın kabul ettiği dört sabit `WindowConstants` arayüzünde tanımlıdır. `DO_NOTHING_ON_CLOSE` (0) kapatmayı tümüyle engeller. `HIDE_ON_CLOSE` (1) pencereyi gizler. `DISPOSE_ON_CLOSE` (2) pencereyi yok eder; başka pencere kalmamışsa program bitebilir. `EXIT_ON_CLOSE` (3) `System.exit(0)` çağırır.

Bu program, ekranı olmayan bir ortamda çalıştırılırsa ilginç bir sonuç verir. Java, ekransız ortamı "başsız" (*headless*) kip olarak tanır. Bu kipte `new JFrame(...)` çağrısı `java.awt.HeadlessException` fırlatır, çünkü gösterilecek ekran yoktur. Panel, düğme ve metin alanı gibi bileşenler ise bu kipte de oluşturulabilir. Notta verilen çıktıların çoğu bu yolla, pencere açılmadan, bileşenleri doğrudan çalıştırarak elde edilmiştir.

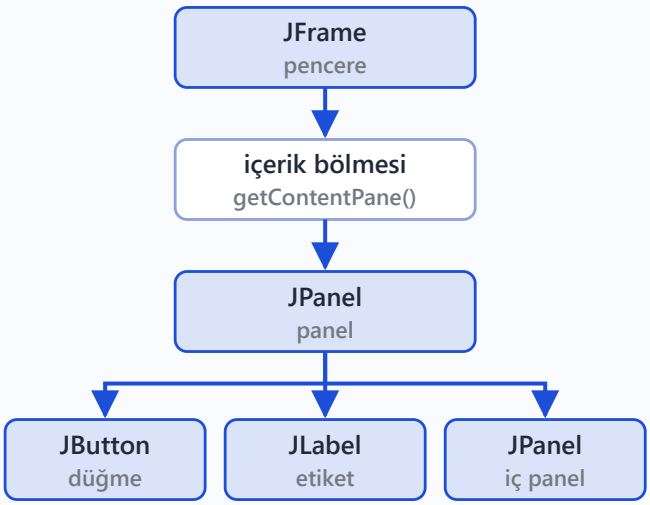
9.2.3 `JPanel` sınıfı

Tanım 9.7

Kapsayıcı (*container*), içinde başka bileşenler barındıran bileşendir. `JPanel`, `javax.swing` paketindeki genel amaçlı kapsayıcıdır; pencere içindeki bileşenleri gruplamak, düzenlemek ve üzerine çizim yapmak için kullanılır.

`JPanel`'in dört kullanımı vardır:

- **Gruplamak:** Birden fazla panel iç içe konarak karmaşık düzenler kurulur. Örneğin bir pencerenin altındaki düğme sırası ayrı bir panelde toplanır.
- **Özelleştirmek:** `setBackground()`, `setBorder()` ve `setPreferredSize()` ile rengi, kenarlığı ve tercih edilen boyutu değiştirilir.
- **Tuval olmak:** Çizim, animasyon ve grafikler panelde yapılır: `JPanel`'den türetilir ve `paintComponent` geçersiz kılınır.
- **Kodu bölmek:** Her panel ayrı bir sınıf olabilir. Büyük bir arayüz böylece küçük, ayrı ayrı anlaşılabilen parçalara bölünür.



Şekil 9.9: İç içe bileşenler. `JFrame` içerik bölmesini, içerik bölmesi bir `JPanel`'i, bu panel de bir düğme, bir etiket ve başka bir paneli içerir.

Şekil 9.9'daki yapı bir ağaçtır. Kök penceredir, yapraklar düğme ve etiket gibi bileşenlerdir. Ara düğümler kapsayıcılardır. Bir kapsayıcı yeniden boyanırken önce kendisini, sonra içindeki bileşenleri çizer. Bileşen ağacının her düzeyinde aynı kural işler.

9.2.4 Panel örneği

```
import java.awt.Color;
import javax.swing.*;

public class PanelOrnegi {
    public static void main(String[] args) {
        JPanel panel = new JPanel();
        panel.setBackground(Color.LIGHT_GRAY);
        panel.add(new JButton("Tıkla"));
        JFrame pencere = new JFrame("JPanel Örneği");
        pencere.add(panel);
        pencere.setSize(300, 150);
        pencere.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        pencere.setVisible(true);
    }
}
```

Program bir panel oluşturur ve arka planını `Color.LIGHT_GRAY`, yani `(192, 192, 192)` açık gri yapar. Panele "Tıkla" yazılı bir düğme eklenir. Ardından pencere oluşturulur, panel pencereye eklenir ve pencere 300 × 150 boyutunda gösterilir. Düğmenin bir dinleyicisi yoktur; tıklanınca görsel olarak basılır, ama hiçbir şey olmaz.

Pencerede düğme panelin üst ortasında durur. Panel ise bütün pencereyi kaplar. Bu iki gözlemin nedeni yerleşim yöneticileridir. `JPanel`'in varsayılan yerleşim yöneticisi `FlowLayout`'tur. Bu yönetici bileşenleri tercih ettikleri boyutta, satır satır ve ortalı dizer. Panelin içinde tek bir düğme olduğu için düğme ilk satırın ortasına konur. `JFrame`'in içerik bölmesi ise `BorderLayout` kullanır. Bölge belirtilmeden eklenen panel merkez bölgeye yerleşir. Merkez de geriye kalan bütün alanı alır.

Örnek 9.3

Bu panel 284 × 111 piksel olsun. "Tıkla" düğmesinin tercih ettiği boyut 61 × 26 ise düğmenin konumu ne olur?

`FlowLayout` varsayılan olarak bileşenler arasında ve kenarlarda 5 piksel boşluk bırakır. Kullanılabilir genişlik $284 - 2 \times 5 = 274$'tür. Düğme bu genişliğin ortasına konur: `x = 5 + (274 - 61) / 2 = 5 + 106 = 111`. Satır üst kenardan 5 piksel aşağıda başlar: `y = 5`. Başsız kipte panel bu boyuta getirilip düzen hesaplatıldığında düğmenin sınırları `[x=111, y=5, width=61, height=26]` çıkar. Düğmenin boyutu yazı tipine bağlıdır; başka bir sistemde birkaç piksel farklı olabilir.

9.3 Çizim araçları

Çizimde üç küçük sınıf sürekli kullanılır: bir noktayı tutan `Point`, rengi tutan `Color` ve yazı tipini tutan `Font`. Üçü de `java.awt` paketindedir.

9.3.1 `Point` sınıfı

Tanım 9.8

`java.awt.Point`, ekrandaki bir `(x, y)` noktasını tutan sınıftır. `x` ve `y` alanları `public int`'tir. `getX()` ve `getY()` aynı değerleri `double` olarak döndürür.

`Point`, sarmalama ilkesinin bilinçli bir istisnasıdır: alanları `public`'tir ve doğrudan okunup yazılabilir. Grafik kodunda nokta koordinatlarına çok sık erişildiği için kütüphane sadeliği tercih etmiştir. Sınıfın metotları da vardır. Aşağıdaki kod bunların çoğunu kullanır.

```
Point nokta = new Point(30, 40);
double xKoordinati = nokta.getX();
double yKoordinati = nokta.getY();
System.out.println(xKoordinati + " " + yKoordinati);
System.out.println(nokta.x + " " + nokta.y);
System.out.println(nokta.distance(0, 0));
nokta.setLocation(60, 80);
System.out.println(nokta);
nokta.translate(5, -10);   // 5 sağa, 10 yukarı
System.out.println(nokta);
System.out.println(Point.distance(0, 0, 3, 4));
```

```
30.0 40.0
30 40
50.0
java.awt.Point[x=60,y=80]
java.awt.Point[x=65,y=70]
5.0
```

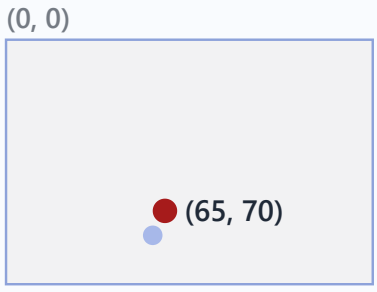
Adım	Satır	Değişkenler	Yeni çıktı
1	1: `Point nokta = new Point(30, 40);`	nokta=(30, 40)	—
2	2: `double xKoordinati = nokta.getX();`	nokta=(30, 40), xKoordinati=30.0	—
3	3: `double yKoordinati = nokta.getY();`	nokta=(30, 40), xKoordinati=30.0, yKoordinati=40.0	—

Adım	Satır	Değişkenler	Yeni çıktı
4	4: `System.out.println(xKoordinati + " " + yKoordinati);`	nokta=(30, 40), xKoordinati=30.0, yKoordinati=40.0	`30.0 40.0`
5	5: `System.out.println(nokta.x + " " + nokta.y);`	nokta=(30, 40), xKoordinati=30.0, yKoordinati=40.0	`30 40`
6	6: `System.out.println(nokta.distance(0, 0));`	nokta=(30, 40), xKoordinati=30.0, yKoordinati=40.0	`50.0`
7	7: `nokta.setLocation(60, 80);`	nokta=(60, 80), xKoordinati=30.0, yKoordinati=40.0	—
8	8: `System.out.println(nokta);`	nokta=(60, 80), xKoordinati=30.0, yKoordinati=40.0	`java.awt.Point[x=60,y=80]`
9	9: `nokta.translate(5, -10); // 5 sağa, 10 yukarı`	nokta=(65, 70), xKoordinati=30.0, yKoordinati=40.0	—
10	10: `System.out.println(nokta);`	nokta=(65, 70), xKoordinati=30.0, yKoordinati=40.0	`java.awt.Point[x=65,y=70]`
11	11: `System.out.println(Point.distance(0, 0, 3, 4));`	nokta=(65, 70), xKoordinati=30.0, yKoordinati=40.0	`5.0`

Tablo 9.5: `Point` örneğinin izlemesi.

1. adımda `(30, 40)` noktası oluşur. 2. ve 3. adımlarda `getX()` ve `getY()` değerleri `double` değişkenlere alınır. 4. adım bu değerleri yazar: `30.0 40.0`. Ondalık nokta, değerin `double` olduğunu gösterir. 5. adım alanlara doğrudan erişir; `nokta.x` bir `int` olduğu için `30 40` yazılır. Aynı sayının iki farklı biçimde yazılması, tipin çıktıyı nasıl etkilediğini gösterir.

6. adımda `distance(0, 0)` noktanın orijine uzaklığını hesaplar. Pisagor bağıntısıyla `√(30² + 40²) = √2500 = 50.0` bulunur. 7. adımda `setLocation(60, 80)` noktayı yeni bir konuma taşır. `xKoordinati` ve `yKoordinati` değişmez. Onlar 2. ve 3. adımda alınmış kopyalardır; nesneyle bağları kalmamıştır. 8. adımda `println(nokta)` nesnenin `toString()` metodunu çağırır ve sınıf adıyla alanları yazar: `java.awt.Point[x=60,y=80]`.



Şekil 9.10: 9. adım. `translate(5, -10)` noktayı 5 piksel sağa, 10 piksel yukarı taşır: `(60, 80)` 'den `(65, 70)` 'e.

9. adımdaki `translate(dx, dy)` göreli bir kaydırmadır: `x`'e `dx`, `y`'ye `dy` eklenir. `dy = -10` olduğu için `y` 80'den 70'e azalır ve nokta ekranda yukarı çıkar (Şekil 9.10). Son satırdaki `Point.distance(0, 0, 3, 4)` bir sınıf metodudur. Nesne gerektirmez; iki noktanın koordinatlarını alır ve `5.0` döndürür.

Örnek 9.4

`Point p = new Point(10, 20); Point q = p; q.translate(5, 5); p.setLocation(p.x * 2, p.y);` satırlarından sonra `p`, `q == p` ve `q.distance(30, 29)` nedir?

`Point q = p;` yeni bir nokta oluşturmaz. İki değişken aynı nesneyi gösterir. `q.translate(5, 5)` bu ortak nesneyi `(15, 25)`'e taşır. `p.setLocation(15 * 2, 25)` nesneyi `(30, 25)`'e taşır. `p` ve `q` aynı nesne olduğu için `q == p` `true`'dur. `(30, 25)` ile `(30, 29)` arası dikey bir doğrudur: uzaklık `4.0`. Bağımsız bir kopya isteniyorsa `new Point(p)` yazılır.

```
java.awt.Point[x=30,y=25] true
4.0
140 70 35
```

Çıktının üçüncü satırı bir sonraki sınıfla ilgilidir: `new Color(200, 100, 50).darker()` rengin bileşenlerini verir.

9.3.2 `Color` sınıfı

`java.awt.Color`, arayüzün ve çizimlerin rengini tutar. Ekrandaki her renk, üç ana rengin karışımıdır: kırmızı (*red*), yeşil (*green*) ve mavi (*blue*). Bu düzene RGB denir.

Tanım 9.9

RGB rengi, kırmızı, yeşil ve mavi bileşenlerin her biri 0–255 aralığında bir tam sayı olan üçlüdür. `new Color(int red, int green, int blue)` bu üçlüden bir renk nesnesi oluşturur. `(0, 0, 0)` siyah, `(255, 255, 255)` beyazdır.

Rengi elde etmenin iki yolu vardır. Birincisi hazır sabitlerdir: `Color.RED`, `Color.GREEN`, `Color.WHITE`, `Color.LIGHT_GRAY` gibi. Bu sabitlerin küçük harfli eş adları da vardır (`Color.red`); ikisi aynı nesneyi gösterir. İkincisi RGB üçlüsüyle yeni bir nesne oluşturmaktır. 0–255 dışındaki bir değer `IllegalArgumentException` fırlatır; `new Color(300, 0, 0)` çağrısının iletisi `Color parameter outside of expected range: Red`'dir.

```
Color turuncu = new Color(255, 128, 0);
System.out.println(turuncu);
System.out.println(turuncu.getGreen());
System.out.println(Color.RED.darker());
```

```
java.awt.Color[r=255,g=128,b=0]
128
java.awt.Color[r=178,g=0,b=0]
```

Bir rengin koyu ve açık tonları için iki metot vardır. `darker()` her bileşeni 0,7 ile çarpar ve tam sayıya keser. `brighter()` ise her bileşeni 0,7'ye böler, sonucu 255'te sınırlar. İki metot da asıl rengi değiştirmez; yeni bir `Color` nesnesi döndürür. `Color` nesneleri değiştirilemezdir.

İfade	RGB	Hesap
`Color.RED`	(255, 0, 0)	hazır sabit
`Color.BLUE`	(0, 0, 255)	hazır sabit
`Color.LIGHT_GRAY`	(192, 192, 192)	hazır sabit
`new Color(255, 128, 0)`	(255, 128, 0)	turuncu
`turuncu.darker()`	(178, 89, 0)	255 × 0,7 = 178,5 → 178; 128 × 0,7 = 89,6 → 89
`Color.RED.darker()`	(178, 0, 0)	0 × 0,7 = 0
`Color.GREEN.darker()`	(0, 178, 0)	giriş yasak işaretinin zemini
`new Color(100, 50, 0).brighter()`	(142, 71, 0)	100 / 0,7 = 142,8 → 142; 50 / 0,7 = 71,4 → 71
`Color.RED.brighter()`	(255, 0, 0)	255 zaten sınırda; 0 bileşenler 0 kalır
`Color.RED.darker().brighter()`	(254, 0, 0)	178 / 0,7 = 254,2 → 254: kesme kaybı

Tablo 9.6: Bazı renkler ve RGB bileşenleri (çalıştırılarak doğrulandı).

Tablonun son satırı bir uyarı taşır. `darker()` ve `brighter()` birbirinin tam tersi değildir. Her işlem sonucu tam sayıya keser ve kesilen kesir geri gelmez. Kırmızıyı koyultup yeniden açınca `(255, 0, 0)` değil, `(254, 0, 0)` elde edilir.

`brighter()` 'ın sıfır bileşenleri neden değiştirmediği de sorulabilir. Sıfırın 0,7'ye bölümü yine sıfırdır. Bu yüzden saf kırmızı “açılınca” pembeye dönmez, aynı kalır. Kütüphane tek bir özel durumu ayrıca ele alır: siyah `(0, 0, 0)` açılınca `(3, 3, 3)` olur, çünkü aksi hâlde siyah hiç açılamazdı.

Örnek 9.5

`new Color(200, 100, 50).darker()` hangi bileşenleri verir?

Her bileşen 0,7 ile çarpılıp kesilir: 200 × 0,7 = 140, 100 × 0,7 = 70, 50 × 0,7 = 35. Sonuç `(140, 70, 35)` 'tir. Bu değerler bir önceki örneğin çıktısının üçüncü satırıyla aynıdır.

9.3.3 `Font` sınıfı

`java.awt.Font`, metnin yazı tipini, stilini ve boyutunu belirler. Yapıcı üç parametre alır: `new Font(ad, stil, punto)`.

Parametre	Değerler
`ad`	`"Serif"`, `"SansSerif"`, `"Monospaced"`, `"Dialog"` (her sistemde bulunan mantıksal adlar) ya da `"Arial"` gibi gerçek bir yazı tipi
`stil`	`Font.PLAIN` (0), `Font.BOLD` (1), `Font.ITALIC` (2); ikisi birden: `Font.BOLD | Font.ITALIC`
`punto`	Tam sayı boyut, örneğin 24

Tablo 9.7: `Font` yapıcısının parametreleri.

Mantıksal ad, Java'nın her sistemde gerçek bir yazı tipine eşlediği addır. `"Serif"` Windows'ta çoğunlukla Times New Roman'a, `"Monospaced"` Courier New'e karşılık gelir. Gerçek bir yazı tipi adı verilir ve o yazı tipi sistemde yoksa Java `"Dialog"`'a döner. Taşınabilir programlar bu yüzden mantıksal adları tercih eder.

Stil sabitleri bit bayraklarıdır. `BOLD` sayının 1. bitini, `ITALIC` 2. bitini kullanır. `|` işleci (bit düzeyinde VEYA) iki bayrağı birleştirir: `1 | 2 = 3`. `new Font("Serif", Font.BOLD | Font.ITALIC, 20)` nesnesinde `getStyle()` 3 döndürür; `isBold()` ve `isItalic()` ikisi de `true`'dur.

```
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(new Color(0, 102, 204));
    g.setFont(new Font("Serif", Font.BOLD, 24));
    g.drawString("Java", 20, 40);
    g.setFont(new Font("Monospaced", Font.PLAIN, 14));
    g.drawString("Grafik", 20, 70);
}
```

Metot kalemi önce `(0, 102, 204)` mavisine çevirir. İlk metin kalın, 24 puntoluk Serif ile `(20, 40)`'a yazılır. Ardından yazı tipi değiştirilir ve ikinci metin 14 puntoluk Monospaced ile `(20, 70)`'e yazılır. Renk değiştirilmediği için iki metin de aynı mavidedir.



Şekil 9.11: `YaziPaneli`'nin çıktısı: kalın Serif 24 punto “Java” ve Monospaced 14 punto “Grafik”.

`JLabel` gibi bileşenlerde de aynı `Font` nesnesi kullanılır: `etiket.setFont(f)`. Var olan bir yazı tipinden yeni bir tane türetmek için `deriveFont` metodu vardır. `f.deriveFont(Font.ITALIC, 18f)` aynı ailede, italik ve 18 puntoluk yeni bir nesne döndürür. `f` değişmez.

Sık yapılan hata

Yalnız boyutu değiştirmek için `f.deriveFont(18)` yazmak. `deriveFont`'un tek `int` parametreli sürümü boyutu değil **stili** değiştirir. 24 puntoluk kalın bir yazı tipinde bu çağrı 18'i stil olarak yorumlar ve boyutu 24'te bırakır. Denendiğinde sonuç stil 0, boyut 24'tür. Boyut için `float` sürümü kullanılır: `f.deriveFont(18f)`.

9.4 Yerleşim yöneticileri

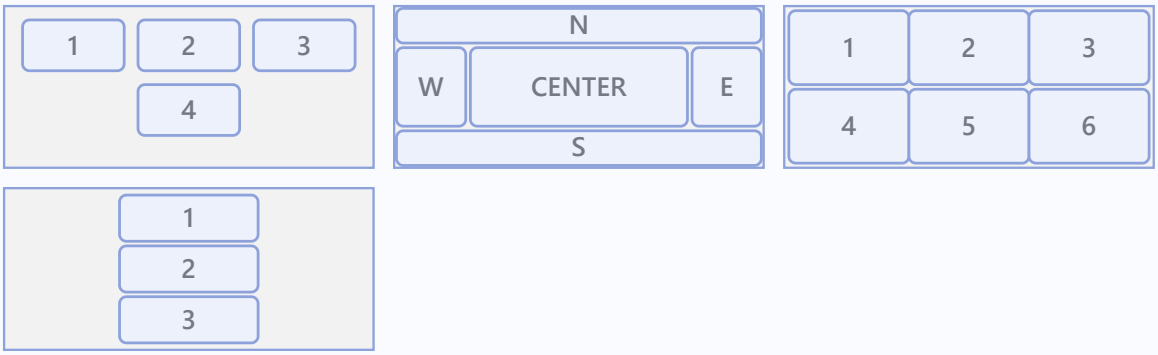
9.4.1 Yerleşim yöneticisi nedir?

Bir pencereye beş düğme konacağını düşünelim. Her düğmenin konumu ve boyutu elle verilebilir: "birinci düğme `(10, 10)`'da, 80 × 30". Bu yöntem bir sorunla karşılaşır. Kullanıcı pencereyi büyütünce düğmeler sol üstte kalır. Ekran çözünürlüğü ya da yazı tipi değişince düğme metinleri kutulara sığmaz. Java bu işi bir nesneye devreder.

Tanım 9.10

Yerleşim yöneticisi (*layout manager*), bir kapsayıcının içindeki bileşenlerin konumunu ve boyutunu kurallara göre hesaplayan nesnedir. Kapsayıcıya `setLayout` ile verilir: `panel.setLayout(new GridLayout(2, 3));`.

Yerleşim yöneticisi koordinatları elle vermenin yerine kullanılır. Pencere büyüyünce, yazı tipi ya da ekran çözünürlüğü değişince düzen kendiliğinden yeniden hesaplanır. Programcı yalnız kuralı seçer: "satır satır diz", "beş bölgeye yerleştir", "eşit hücrelere böl". Java'nın standart yöneticilerinden dördü bu notta işlenir.



Şekil 9.12: Dört yerleşim yöneticisi. Soldan sağa: `FlowLayout` (soldan sağa, satır dolunca alta; `JPanel` varsayılanı), `BorderLayout` (beş bölge; `JFrame` içerik bölmesi varsayılanı), `GridLayout(2, 3)` (eşit boyutlu hücreler), `BoxLayout.Y_AXIS` (tek sütun).

Daha esnek ama karmaşık olan `GridBagLayout`, farklı boyutlarda hücrelerden oluşan bir ızgara kurar. Çoğu zaman buna gerek kalmaz. Panelleri iç içe koyup her birine basit bir yönetici vermek yeterlidir. Örneğin bir pencerenin içerik bölmesi `BorderLayout` kullanır, alttaki düğme şeridi ise `FlowLayout` kullanan ayrı bir paneldir.

Yerleşim yöneticilerinin davranışı etkileşimli olarak Swing Yerleşim Yöneticileri simülatöründe izlenebilir. Simülatörde pencere boyutu değiştikçe dört yöneticinin düzeni nasıl yeniden hesapladığı görülür.

9.4.2 Kod izleme: `FlowLayout` satırları nasıl doldurur?

Yerleşim yöneticisinin çalışmasını anlamanın en iyi yolu, bir düzeni elle hesaplamaktır. Aşağıdaki kod, 260 piksel genişliğindeki bir panele genişlikleri farklı beş düğme ekler ve hesaplanan konumları yazar.

```
int[] gen = {90, 60, 110, 70, 80};
JPanel panel = new JPanel();   // FlowLayout(CENTER, 5, 5)
panel.setSize(260, 120);
for (int i = 0; i < gen.length; i++) {
    JButton b = new JButton("B" + i);
    b.setPreferredSize(new Dimension(gen[i], 30));
    panel.add(b);
}
panel.doLayout();              // konumları hesapla
for (Component c : panel.getComponents())
    System.out.println(c.getX() + ", " + c.getY());
```

```
52, 5
147, 5
37, 40
152, 40
90, 75
```

`FlowLayout`'un varsayılan ayarları `FlowLayout(CENTER, 5, 5)`'tir: satırlar ortalanır, bileşenler arasında yatayda ve dikeyde 5 piksel boşluk kalır. Aynı boşluk panelin kenarlarında da bırakılır. Her düğmenin tercih ettiği boyut `setPreferredSize` ile verilmiştir: genişlikler 90, 60, 110, 70 ve 80; yükseklikler 30.

Adım	Satır	Değişkenler	Yeni çıktı	Açıklama

1	1: `int[] gen = {90, 60, 110, 70, 80};`
	2: `JPanel panel = new JPanel(); // FlowLayout(CENTER, 5, 5)`
	3: `panel.setSize(260, 120);`
		gen=[90, 60, 110, 70, 80], panel=260×120	—	`JPanel`'in varsayılan yöneticisi `FlowLayout`: ortalı, bileşenler arası 5 piksel boşluk. Kesikli çizgiler kenar boşluğu.

2	5: `JButton b = new JButton("B" + i);` 6: `b.setPreferredSize(new Dimension(gen[i], 30));` 7: `panel.add(b);`	i=0, b=B0 (90×30)	—	`add` düğmeyi yalnızca listeye ekler; konumu henüz (0, 0), boyutu 0 × 0.
3	4: `for (int i = 0; i < gen.length; i++) {` 5: `JButton b = new JButton("B" + i);` 6: `b.setPreferredSize(new Dimension(gen[i], 30));` 7: `panel.add(b);`	i=4, b=B4 (80×30)	—	Döngü bitti: beş düğme eklendi, hiçbiri yerleşmedi. Yerleşim ancak düzen hesaplanınca yapılır.
4	9: `panel.doLayout(); // konumları hesapla`	satır=1, genişlik=90+5+60=155	—	`doLayout()`: kullanılabilir genişlik 260 − 2 × 5 = 250. B0 ve B1 sığar: 155 piksel.
5	9: `panel.doLayout(); // konumları hesapla`	satır=1, deneme=155+5+110 = 270 > 250	—	B2 de eklenirse satır 270 piksel olur: sığmaz, **alt satıra** iner.
6	9: `panel.doLayout(); // konumları hesapla`	satır=1, x0=5+(250−155)/2 = 52	—	Satır 1 ortalanır: B0 x = 52, B1 x = 52 + 90 + 5 = 147, y = 5.
7	9: `panel.doLayout(); // konumları hesapla`	satır=2, genişlik=110+5+70 = 185	—	Satır 2: B2 ve B3 (185 piksel); B4 eklense 270 olur. Ortalama: x0 = 5 + (250 − 185)/2 = 37.

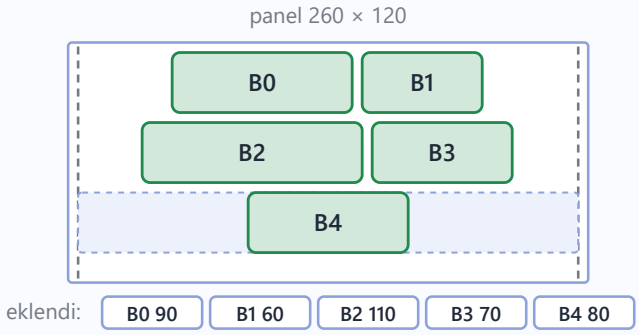
Adım	Satır	Değişkenler	Yeni çıktı	Açıklama
8	9: `panel.doLayout(); // konumları hesapla`	satır=3, x0=5+(250−80)/2 = 90	—	B4 tek başına üçüncü satırda; y = 5 + 30 + 5 + 30 + 5 = 75.
9	10: `for (Component c : panel.getComponents())` 11: `System.out.println(c.getX() + ", " + c.getY());`	c=B0 … B4	`52, 5 ⏎ 147, 5 ⏎ 37, 40 ⏎ 152, 40 ⏎ 90, 75`	Hesaplanan sınırlar yazdırılır. Panel genişlerse satırlar yeniden hesaplanır ve düğmeler yer değiştirir.

Tablo 9.8: `FlowLayout` yerleşim izlemesi.

İzlemenin ilk dersi 2. ve 3. adımlardadır. `add` düğmeyi kapsayıcının bileşen listesine ekler, ama yerleştirmez. Döngü bittiğinde beş düğmenin de konumu `(0, 0)`, boyutu 0 × 0'dır. Yerleşim ancak düzen hesaplanınca yapılır. Gerçek bir pencerede bu iş `pack()`, `setVisible(true)` ya da pencere boyutunun değişmesiyle kendiliğinden tetiklenir. Örnekte ise `doLayout()` açıkça çağrılır.

`doLayout()` satırları şöyle doldurur. Kullanılabilir genişlik, panel genişliğinden iki kenar boşluğunun çıkarılmasıyla bulunur: 260 − 2 × 5 = 250. Düğmeler sırayla satıra eklenir. B0 (90) ve B1 (60) aralarındaki 5 piksel boşlukla 155 piksel tutar; sığar. B2 de eklense satır 155 + 5 + 110 = 270 piksel olurdu. 270 > 250 olduğu için B2 alt satıra iner.

Bir satır kapanınca ortalanır. Satırın boşta kalan genişliği iki yana eşit dağıtılır: `x0 = 5 + (250 − 155) / 2 = 5 + 47 = 52`. Tam sayı bölmesi 47,5'i 47'ye keser. B0 `x = 52`'ye, B1 `x = 52 + 90 + 5 = 147`'ye konur; ikisinin `y`'si 5'tir.



Şekil 9.13: 8. adım. Üç satır kuruldu; B4 tek başına üçüncü satırda ortalanmış.

İkinci satır B2 (110) ve B3 (70) ile 185 piksel tutar. B4 eklense 185 + 5 + 80 = 270 olurdu; yine sığmaz. Ortalama: `x0 = 5 + (250 − 185) / 2 = 5 + 32 = 37`. B2 `x = 37`'ye, B3 `x = 37 + 110 + 5 = 152`'ye konur. Satırın `y`'si, önceki satırın altı artı boşluktur: `5 + 30 + 5 = 40`. B4 üçüncü satırda yalnızdır: `x0 = 5 + (250 − 80) / 2 = 90`, `y = 5 + 30 + 5 + 30 + 5 = 75`. Hesap çıktıyla birebir tutar.

Panel genişlerse satırlar yeniden hesaplanır. Aynı panel 400 piksele çıkarılıp düzen yeniden hesaplatıldığında kullanılabilir genişlik 390 olur. İlk dört düğme (90 + 60 + 110 + 70 + 3 × 5 = 345) tek satıra sığar; B4 ikinci satıra iner:

```
(27, 5) (122, 5) (187, 5) (302, 5) (160, 40)
```

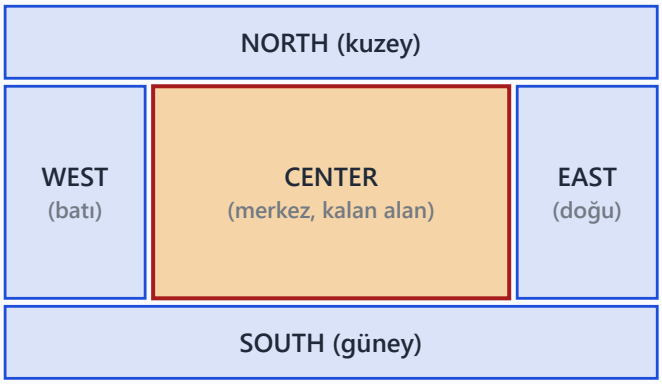
Örnek 9.6

200 piksel genişliğindeki bir `FlowLayout` paneline 80 × 30 boyutunda üç bileşen eklenir. Bileşenlerin konumları nedir?

Kullanılabilir genişlik 200 − 10 = 190'dır. İlk iki bileşen 80 + 5 + 80 = 165 piksel tutar; üçüncüsü eklense 250 olur, sığmaz. Birinci satırın başlangıcı `5 + (190 − 165) / 2 = 5 + 12 = 17` 'dir. Bileşenler `(17, 5)` ve `(102, 5)` 'e konur. Üçüncü bileşen ikinci satırdadır: `x = 5 + (190 − 80) / 2 = 60` , `y = 40` . Başsız kipte çalıştırılan program aynı değerleri verdi.

9.4.3 `BorderLayout`

`BorderLayout` , kapsayıcıyı beş bölgeye ayırır: `NORTH` (kuzey), `SOUTH` (güney), `WEST` (batı), `EAST` (doğu) ve `CENTER` (merkez). Önceliği farklı bileşenleri düzenli göstermek için uygundur: araç çubuğu üstte, durum satırı altta, asıl içerik ortada.



Şekil 9.14: `BorderLayout` bölgeleri.

Yöneticinin kuralları şunlardır:

- Bölge belirtilmeden eklenen bileşen merkeze yerleşir: `add(b)` ile `add(b, BorderLayout.CENTER)` aynıdır.
- Her bölgede tek bileşen olur. Aynı bölgeye ikinci ekleme öncekinin yerini alır.

- Kuzey ve güney bütün genişliği alır; yükseklikleri bileşenin tercih ettiği yüksekliktir. Batı ve doğu, kuzeyle güney arasında kalan yüksekliği alır; genişlikleri tercih edilen genişliktir. Merkez geriye kalan her şeyi doldurur.
- `setHgap()` ve `setVgap()` bölgeler arasındaki yatay ve dikey boşluğu piksel olarak ayarlar. Aynı değerler yapıcıya da verilebilir: `new BorderLayout(5, 5)`.

İkinci kuralın ince bir yanı vardır. Aynı bölgeye ikinci bileşen eklendiğinde birincisi kapsayıcıdan silinmez. Kapsayıcının bileşen sayısı 2'dir; ama yönetici yalnız son ekleneni o bölgeye bağlar. Birinci bileşen yerleşime katılmaz ve boyutu 0 × 0 kalır. Ekranda görünmez.

> **Sık yapılan hata**
>
> Pencereye bölge belirtmeden iki bileşen eklemek: `f.add(new JButton("Bir")); f.add(new JButton("İki"));`. İçerik bölmesi `BorderLayout` kullandığı için ikisi de merkeze gider ve son eklenen kazanır. Ekranda yalnız "İki" görünür ve bütün alanı kaplar. 300 × 200'lük bir `BorderLayout` panelinde denendiğinde "İki" `(0, 0, 300, 200)`, "Bir" ise 0 × 0 çıkar. Yan yana dizilim `FlowLayout`'un davranışıdır; bunun için düğmeler önce bir `JPanel`'e konur.

9.4.4 Kod izleme: bölgeleri doldurmak

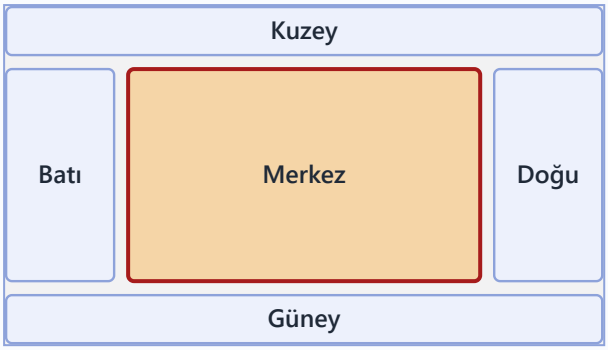
```
JPanel panel = new JPanel();
BorderLayout yerlesim = new BorderLayout();
yerlesim.setHgap(5);
yerlesim.setVgap(5);
panel.setLayout(yerlesim);
panel.add(new JButton("Kuzey"), BorderLayout.NORTH);
panel.add(new JButton("Güney"), BorderLayout.SOUTH);
panel.add(new JButton("Batı"), BorderLayout.WEST);
panel.add(new JButton("Doğu"), BorderLayout.EAST);
panel.add(new JButton("Merkez"), BorderLayout.CENTER);
```

Kod bir panel ve bir `BorderLayout` nesnesi oluşturur, boşlukları 5 piksel yapar ve yöneticiyi panele verir. Ardından beş bölgenin her birine bir düğme eklenir. Tablo 9.9 her eklemenin düzene etkisini anlatır.

Adım	Satır	Açıklama
1	1: `JPanel panel = new JPanel();` 2: `BorderLayout yerlesim = new BorderLayout();` 3: `yerlesim.setHgap(5);` 4: `yerlesim.setVgap(5);` 5: `panel.setLayout(yerlesim);`	Panel `BorderLayout` kullanacak; bölgeler arasında 5 piksel boşluk var. Henüz bileşen yok.

Adım	Satır	Açıklama
2	6: `panel.add(new JButton("Kuzey"), BorderLayout.NORTH);`	Kuzey düğmesi **tüm genişliği** alır, yüksekliği düğmenin tercih ettiği kadardır.
3	7: `panel.add(new JButton("Güney"), BorderLayout.SOUTH);`	Güney de tüm genişlikte, en alta yerleşir.
4	8: `panel.add(new JButton("Batı"), BorderLayout.WEST);`	Batı, kuzey ile güney arasında kalan yüksekliği alır; genişliği metne göre.
5	9: `panel.add(new JButton("Doğu"), BorderLayout.EAST);`	Doğu sağ kenara yerleşir.
6	10: `panel.add(new JButton("Merkez"), BorderLayout.CENTER);`	Merkez **geriye kalan bütün alanı** doldurur. Pencere büyütülürse en çok merkez büyür.

Tablo 9.9: `BorderLayout` bölgelerinin dolması.



Şekil 9.15: 6. adım. Beş bölge dolu; merkez geriye kalan bütün alanı aldı.

Piksel değerleri panel boyutuna ve düğmelerin tercih ettiği boyuta bağlıdır. Panel 400 × 250 yapılıp düzen hesaplatıldığında değerler şöyle çıkar. Düğmeler varsayılan yazı tipinde 26 piksel yüksektir. Kuzey `(0, 0)`'da 400 × 26, güney `(0, 224)`'te 400 × 26'dır. Batı ve doğu kuzeyin altından 5 piksel boşlukla başlar: `y = 26 + 5 = 31`. Yükseklikleri `250 − 26 − 26 − 2 × 5 = 188`'dir. Batı 56, doğu 63 piksel geniştir; bunlar "Batı" ve "Doğu" metinlerinin gerektirdiği genişliklerdir. Merkez `x = 56 + 5 = 61`'den başlar ve `400 − 56 − 63 − 2 × 5 = 271` piksel genişliğindedir. Merkezin tercih ettiği 76 piksel bu hesapta hiç kullanılmaz.

Pencere büyütülürse en çok merkez büyür. Kuzey ve güney yalnız genişler, batı ve doğu yalnız uzar. Bu özellik `BorderLayout`'u ana pencere düzeni için uygun kılar: metin alanı ya da çizim tuvali merkeze konur ve pencereyle birlikte büyür.

Örnek 9.7

300 × 200 piksellik ve boşluksuz bir `BorderLayout` paneline beş bileşen eklenir. Tercih edilen boyutlar şöyledir: kuzey 100 × 30, güney 100 × 20, batı 60 × 50, doğu 40 × 50, merkez 10 × 10. Ardından kuzeye 80 × 40 tercihli ikinci bir bileşen eklenir. Bileşenlerin sınırları nedir?

Kuzey bütün genişliği alır; yüksekliği tercih edilen 30'dur: `(0, 0, 300, 30)`. Tercih edilen en olan 100 yok sayılır. Güney en alttadır: `y = 200 − 20 = 180`, yani `(0, 180, 300, 20)`. Batı ile doğu aradaki `200 − 30 − 20 = 150` piksel yüksekliği alır: batı `(0, 30, 60, 150)`, doğu `(300 − 40, 30, 40, 150) = (260, 30, 40, 150)`. Merkez kalanı alır: `(60, 30, 200, 150)`. İkinci kuzey bileşeni ilkinin yerini alır ve 40 piksel yüksektir. Batı, doğu ve merkez 10 piksel aşağı kayar ve kısalır: batı `(0, 40, 60, 140)`, merkez `(60, 40, 200, 140)`. Eski kuzey bileşeni panelde kalır ama boyutu güncellenmez. Başsız kipte çalıştırılan program bu değerlerin hepsini verdi.

9.4.5 `GridLayout` ve `BoxLayout`

`GridLayout(satır, sütun)`, kapsayıcıyı eşit boyutlu hücrelere böler. Bileşenler satır satır, soldan sağa yerleşir. Her bileşen hücresini tümüyle doldurur; tercih edilen boyut dikkate alınmaz.

Örnek 9.8

`new GridLayout(2, 3)` kullanan 300 × 120 piksellik bir panele sırayla beş düğme eklenir. Beşinci düğmenin konumu ve boyutu nedir?

Hücre genişliği 300 / 3 = 100, yüksekliği 120 / 2 = 60'tır. Düğme 1–3 üst satırı doldurur. Düğme 4 alt satırın ilk hücresine, düğme 5 ikinci hücresine yerleşir: `(100, 60, 100, 60)`. Altıncı hücre boş kalır. Hücre boyutu bileşen sayısına göre değişmez; çalıştırılan program aynı sınırları verdi.

Satır ya da sütun sayısından biri 0 verilebilir. `new GridLayout(0, 2)` "iki sütun, satır sayısı gerektiği kadar" demektir. 200 × 150'lik bir panele beş düğme eklenirse üç satır oluşur. Hücreler 100 × 50 olur ve beşinci düğme `(0, 100)`'e konur. Bileşen sayısı satır × sütun sayısını aşarsa ve ikisi de sıfırdan farklıysa, Java satır sayısını korur ve sütun sayısını artırır. `GridLayout(2, 3)`'e yedi düğme eklenince ızgara 2 × 4 olur ve hücreler 75 piksele daralır.

`BoxLayout` bileşenleri tek satıra ya da tek sütuna dizer. Yapıcısı, yönettiği kapsayıcıyı da parametre olarak alır: `panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS))`. `Y_AXIS` alt alta, `X_AXIS` yan yana dizer. Formlarda etiketleri ve alanları alt alta sıralamak için kullanılır.

Sık yapılan hata

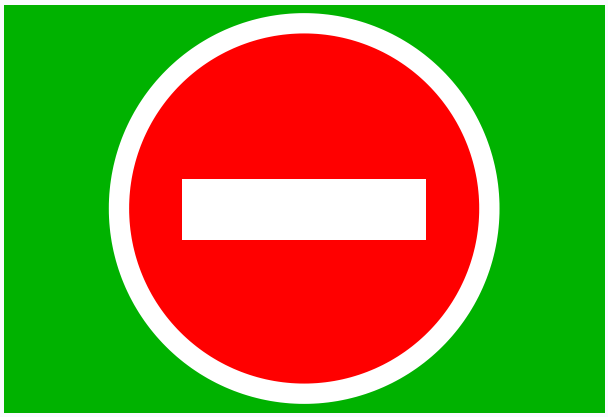
Yerleşim yöneticisi tanımlı bir kapsayıcıda bileşenin konumunu `setBounds` ya da `setLocation` ile vermek. Yönetici bir sonraki düzen hesabında bu değerleri ezer; elle verilen konum kaybolur. Bileşenin boyutunu etkilemek için `setPreferredSize` kullanılır. Yönetici bu tercihi kuralına göre dikkate alır ya da almaz.

9.5 Uygulama: giriş yasak işareti

9.5.1 Hedef ve katmanlar

Bu uygulama, trafikteki “girişi olmayan yol” işaretini çizen bir panel yazar. İşaret, üst üste boyanan dört katmandan oluşur:

1. Yeşil arka plan (`setBackground` ile).
2. Beyaz büyük daire (dış halka).
3. Kırmızı, biraz küçük daire.
4. Ortada beyaz yatay bar.



Şekil 9.16: Giriş yasak işareti: yeşil zemin üzerinde beyaz halka, kırmızı daire ve ortada beyaz yatay bar.

Programın önemli bir tasarım kararı vardır: bütün ölçüler `getWidth()` ve `getHeight()`'tan hesaplanır. Sabit piksel değeri kullanılmaz. Pencere büyütülünce işaret de büyür ve her zaman ortada kalır. Çizim, panelin o anki boyutuna göre her seferinde baştan hesaplanır.

9.5.2 `paintComponent` kodu

```
public class GirisYasaktirIsareti extends JPanel {
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        var c = new Point(getWidth() / 2, getHeight() / 2); // merkez
        var r = Math.min(getWidth() / 2, getHeight() / 2) - 5; // yarıçap
        var r2 = (int) (r * 0.9);   // iç yarıçap
        var bw = (int) (r2 * 1.4);  // bar genişliği
        var bh = (int) (r2 * 0.35); // bar yüksekliği
        g.setColor(Color.WHITE);
        g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);
        g.setColor(Color.RED);
        g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);
        g.setColor(Color.WHITE);
        g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);
    }
}
```

Kod beş değişken hesaplar, sonra üç şekil çizer. Değişkenler `var` ile tanımlanmıştır. Java 10'dan beri `var`, yerel değişkenin tipini derleyicinin sağdaki ifadeden çıkarmasını sağlar. Burada `c`'nin tipi `Point`, diğerlerinin tipi `int`'tir. Klasik yazım şöyledir: `Point c = new Point(…); int r = …;`. `var` tipi ortadan kaldırmaz; tip derleme anında belirlenir ve sonradan değişmez.

Satırları sırayla okuyalım:

- `c`: Panelin merkezi. Genişlik ve yükseklik ikiye bölünür. İki işlenen de `int` olduğu için tam sayı bölmesi yapılır.
- `r`: Dış dairenin yarıçapı. Merkezden kısa kenara olan uzaklık seçilir (`Math.min`) ve 5 piksel pay bırakılır. Daire böylece panelin dışına taşmaz.
- `r2`: İç dairenin yarıçapı, `r`'nin %90'ı. `r * 0.9` bir `double`'dır; `(int)` dönüşümü ondalık kısmı atar.
- `bw` ve `bh`: Barın genişliği ve yüksekliği, iç yarıçapın 1,4 ve 0,35 katı.

Çizim üç adımda yapılır. Önce kalem beyaza çevrilir ve dış daire çizilir. Sonra kalem kırmızıya çevrilir ve iç daire çizilir. Kırmızı daire beyazın üstüne boyanır; beyazdan geriye yalnız dışta ince bir halka kalır. Son olarak kalem yeniden beyaza çevrilir ve bar çizilir. Bar, sınırlayıcı dikdörtgenin sol üst köşesiyle verilir: merkezden yarı genişlik kadar sola, yarı yükseklik kadar yukarı.

Gerekli içe aktarmalar `java.awt.*` ve `javax.swing.*`'tır. `Point`, `Graphics` ve `Color` birincisinden, `JPanel` ikincisinden gelir.

9.5.3 Kod izleme: işareti çizmek

Panel 384 × 261 piksel olsun. Bu boyut, 400 × 300'lük bir pencereden başlık çubuğu ve kenarlıklar düşülünce kalan iç alanın tipik değeridir. Gerçek değer işletim sistemine ve tema ayarlarına göre birkaç piksel değişir.

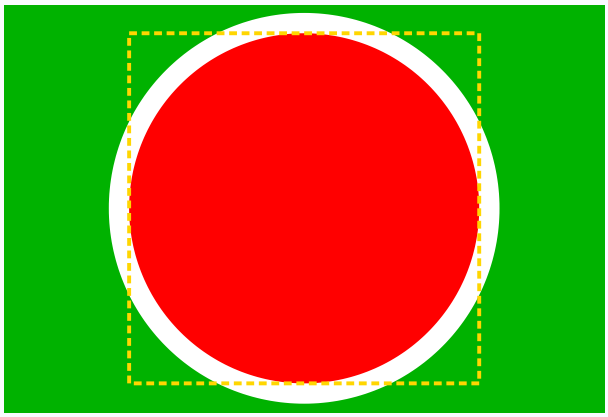
Adım	Satır	Değişkenler	Açıklama
1	1: `super.paintComponent(g);`	getWidth()=384, getHeight()=261	Panel 384 × 261 piksel. `super.paintComponent` zemini arka plan rengiyle (yeşil) boyar.
2	2: `var c = new Point(getWidth() / 2, getHeight() / 2);`	getWidth()=384, getHeight()=261, c=(192, 130)	Merkez: (384 / 2, 261 / 2) = (192, 130). Tam sayı bölmesi: 130,5 → 130.
3	3: `var r = Math.min(getWidth() / 2, getHeight() / 2) - 5;`	getWidth()=384, getHeight()=261, c=(192, 130), r=125	Yarıçap: min(192, 130) − 5 = 125. Kısa kenara göre seçilir, 5 piksel pay kalır.
4	4: `var r2 = (int) (r * 0.9);`	getWidth()=384, getHeight()=261, c=(192, 130), r=125, r2=112	İç yarıçap: (int) 112,5 = 112.
5	5: `var bw = (int) (r2 * 1.4);`	getWidth()=384, getHeight()=261, c=(192, 130), r=125, r2=112, bw=156	Bar genişliği: (int) (112 × 1,4) = (int) 156,8 = 156.
6	6: `var bh = (int) (r2 * 0.35);`	getWidth()=384, getHeight()=261, c=(192, 130), r=125, r2=112, bw=156, bh=39	Bar yüksekliği: (int) 39,2 = 39.
7	7: `g.setColor(Color.WHITE);` 8: `g.fillOval(c.x - r, c.y - r, 2 * r, 2 * r);`	getWidth()=384, getHeight()=261, c=(192, 130), r=125, r2=112, bw=156, bh=39	`fillOval(67, 5, 250, 250)` : beyaz daire, sınırlayıcı kare kesikli çizgide.
8	9: `g.setColor(Color.RED);` 10: `g.fillOval(c.x - r2, c.y - r2, r2 * 2, r2 * 2);`	getWidth()=384, getHeight()=261, c=(192, 130), r=125, r2=112, bw=156, bh=39	`fillOval(80, 18, 224, 224)` : kırmızı daire beyazın üstüne boyanır; 13 piksellik beyaz halka kalır.
9	11: `g.setColor(Color.WHITE);` 12: `g.fillRect(c.x - bw / 2, c.y - bh / 2, bw, bh);`	getWidth()=384, getHeight()=261, c=(192, 130), r=125, r2=112, bw=156, bh=39	`fillRect(114, 111, 156, 39)` : beyaz bar en üste. İşaret tamam.

Tablo 9.10: Panel 384 × 261 iken `paintComponent` izlemesi.

2. adımda merkez `(384 / 2, 261 / 2)` hesaplanır. 261 / 2 matematikte 130,5'tir, ama tam sayı bölmesi ondalığı atar: `(192, 130)`. 3. adımda yarıçap için `min(192, 130) − 5 = 125` bulunur. Kısa kenar yükseklik olduğu için yarıçapı o belirler.

4.–6. adımlar tip dönüşümünü gösterir. `125 × 0,9 = 112,5`; `(int)` bunu 112'ye keser. Yuvarlama yapılmaz: 112,5 bile aşağı iner. `112 × 1,4` ise 156,8 olmalıdır. `double` aritmetiği bu değeri tam gösteremez; Java'da çarpımın sonucu `156.79999999999998` 'dir. Kesme yine 156 verir. Bar yüksekliği `112`

`× 0,35 = 39,2` ; kesilince 39'dur.



Şekil 9.17: 8. adım. `fillOval(80, 18, 224, 224)` : kırmızı daire beyazın üstüne boyandı; sınırlayıcı kare kesikli çizgiyle gösterilmiştir.

7. adımdaki `fillOval(c.x - r, c.y - r, 2 * r, 2 * r)` çağrısının argümanları `(192 - 125, 130 - 125, 250, 250) = (67, 5, 250, 250)` 'dir. 8. adımda iç daire `(192 - 112, 130 - 112, 224, 224) = (80, 18, 224, 224)` ile çizilir. İki yarıçap arasındaki fark 125 − 112 = 13 piksellik beyaz halkayı verir. 9. adımda bar `(192 - 78, 130 - 19, 156, 39) = (114, 111, 156, 39)` 'a çizilir. `bw / 2 = 78` ve `bh / 2 = 19` değerleri de tam sayı bölmesidir.

Hesaplanan çağrılar Java'da doğrulandı. Panel başsız kipte 384 × 261 boyutuna getirildi ve `paintComponent` bir resim üzerinde `DebugGraphics` kalemiyle çalıştırıldı. Kayıtta tam olarak `fillOval(67, 5, 250, 250)` , `fillOval(80, 18, 224, 224)` ve `fillRect(114, 111, 156, 39)` görüldü. Resimdeki pikseller de beklenen renkleri taşıyordu: merkez beyaz (bar), `(192, 25)` kırmızı, `(192, 10)` beyaz (halka), `(5, 5)` yeşil.

Örnek 9.9

Aynı `paintComponent` 240 × 180 piksellik bir panelde çalışırsa değişkenler ve üç çizim çağrısının argümanları ne olur?

Merkez `(120, 90)` 'dır. `r = min(120, 90) - 5 = 85` . `r2 = (int) (85 × 0,9) = (int) 76,5 = 76` . `bw = (int) (76 × 1,4) = (int) 106,4 = 106` ; `bh = (int) (76 × 0,35) = (int) 26,6 = 26` . Kesme 26,6'yı bile 26'ya indirir. Çağrılar şunlardır:

```
setColor(0, 178, 0)
fillRect(0, 0, 240, 180)
setColor(255, 255, 255)
fillOval(35, 5, 170, 170)
setColor(255, 0, 0)
fillOval(44, 14, 152, 152)
setColor(255, 255, 255)
fillRect(67, 77, 106, 26)
```

Dış daire `(120 - 85, 90 - 85, 170, 170)`, iç daire `(120 - 76, 90 - 76, 152, 152)` 'dir; halka 9 piksel kalınlığındadır. Bar için `bw / 2 = 53`, `bh / 2 = 13` : `(67, 77, 106, 26)`.

Örnek 9.10

Panel 200 × 100 piksel olsaydı `r`, `r2`, `bw` ve `bh` kaç olurdu?

`r = min(100, 50) - 5 = 45`. `r2 = (int) 40,5 = 40`. `bw = (int) (40 × 1,4)` ve `bh = (int) (40 × 0,35)` 'tir. Burada bir kayan nokta tuzağı akla gelebilir: `1.4` ve `0.35` ikilik sistemde tam gösterilemez, çarpım 55,999... çıkıp 55'e kesilebilir mi? Java'da denendiğinde `40 * 1.4` tam `56.0`, `40 * 0.35` tam `14.0` verdi. Sonuç `bw = 56`, `bh = 14` 'tür. Bu tür sınır durumlarında sonuca güvenmenin yolu hesabı çalıştırmaktır.

Sık yapılan hata

Katmanların sırasını karıştırmak. Kırmızı daire beyazdan önce çizilirse büyük beyaz daire onu tümüyle örter ve işaret beyaz bir daireye döner. Bar kırmızıdan önce çizilirse kırmızı daire barı örter. Sonradan çizilen her zaman üsttedir; çizim sırası katman sırasıdır.

9.5.4 `main` ve pencere

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        var panel = new GirisYasaktirIsareti();
        panel.setBackground(Color.GREEN.darker());
        var frame = new JFrame("Basit bir grafik programı");
        frame.setSize(400, 300);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.getContentPane().add(panel, BorderLayout.CENTER);
        frame.setVisible(true);
    });
}
```

`main` metodunun tek işi, arayüzü kuracak kodu `SwingUtilities.invokeLater`'a vermektir. `() -> { … }` bir lambda ifadesidir: "sonra çalıştırılacak kod parçası". `invokeLater` bu parçayı olay iş parçacığında çalıştırılmak üzere kuyruğa koyar. Bu kalıp her Swing programının `main`'inde bulunur; gerekçesi olay iş parçacığının çalışma kuralıdır.

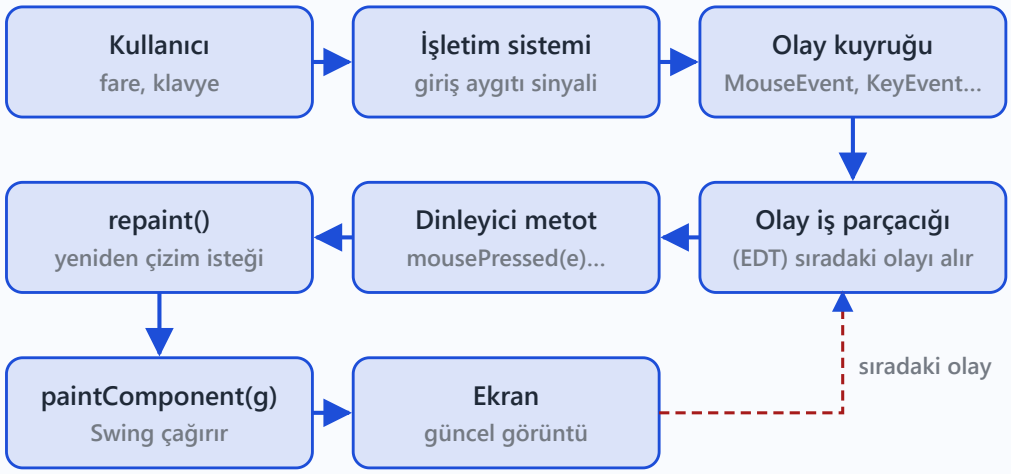
Lambdanın içinde panel oluşturulur ve arka planı `Color.GREEN.darker()`, yani `(0, 178, 0)` yapılır. Saf yeşil `(0, 255, 0)` göz yoracak kadar parlaktır; koyu ton trafik işaretine daha çok benzer. `super.paintComponent(g)` zemini bu renkle boyar. Panel, pencerenin içerik bölmesinin merkezine eklenir. Tek bileşen olduğu için bütün alanı kaplar.

400 × 300 pencerede kenarlık ve başlık çubuğu düşülünce panel yaklaşık 384 × 261 olur. Bu değer işletim sistemine göre değişir. Programın doğru çalışması bu değere bağlı değildir: ölçüler her çizimde `getWidth()` ve `getHeight()`'tan yeniden hesaplanır. Panel 768 × 522'ye büyütülürse dış daire `fillOval(128, 5, 512, 512)` ile çizilir ve işaret yine ortadadır.

9.6 Fare olayları ve olay iş parçacığı

9.6.1 Etkileşimli grafik ve olay döngüsü

Bilgisayar grafikleri, görsel içerik oluşturma ve işleme sürecidir. Etkileşimli (*interactive*) grafiklerde kullanıcı fare, klavye gibi giriş aygıtlarıyla görüntüyü değiştirir. Çizim programları, oyunlar ve haritalar bu türdendir. Grafik kütüphaneleri şekil çizmeyi ve renklendirmeyi sağlar; etkileşim ise bir olay döngüsüyle kurulur.



Şekil 9.18: Olay döngüsü. Kullanıcının eylemi işletim sisteminden geçip olay kuyruğuna girer. Olay iş parçacığı olayı dinleyici metoda verir; metot veriyi günceller ve `repaint()` ister. Swing uygun anda `paintComponent`'i çağırır ve ekran güncellenir.

Döngüdeki adımlar sırasıyla şunlardır. Kullanıcı fareye basar. İşletim sistemi bu eylemi algılar ve Java'ya bildirir. Java bir olay nesnesi oluşturur ve olay kuyruğuna koyar. Olay iş parçacığı kuyruktaki sıradaki olayı alır ve o bileşene kayıtlı dinleyicinin metodunu çağırır. Dinleyici metot programın verisini günceller ve `repaint()` ile yeniden çizim ister. Swing uygun bir anda `paintComponent`'i çağırır; ekran yeni veriyle çizilir. Olay iş parçacığı sıradaki olaya geçer.

Bu döngünün bir kuralı vardır: dinleyici metot ekranı doğrudan boyamaz. Yalnız veriyi değiştirir ve yeniden çizim ister. Çizimi her zaman `paintComponent` yapar. Bu ayrım, çizimin tek bir yerde toplanmasını ve pencere her yeniden boyandığında doğru görüntünün oluşmasını sağlar.

9.6.2 Dinleyici arayüzü ve adaptör sınıfı

Tanım 9.11

Olay dinleyici (*event listener*), bir olay olunca çağrılacak metotları tanımlayan arayüz ya da bu arayüzü uygulayan nesnedir. Dinleyici nesne bir bileşene `add…Listener` metoduyla kaydedilir.

Fare olayları için Java'nın arayüzü `MouseListener`'dır. Arayüzün beş soyut metodu vardır. Bir sınıf bu arayüzü uygularsa beş metodun hepsini yazmak zorundadır. İlgilenmediği metotların gövdesi boş kalır.

```
class Dinleyici implements MouseListener {
    public void mouseClicked(MouseEvent e) {
        System.out.println("tık");
    }
    public void mousePressed(MouseEvent e) { }
    public void mouseReleased(MouseEvent e) { }
    public void mouseEntered(MouseEvent e) { }
    public void mouseExited(MouseEvent e) { }
}
```

Beş metottan dördü boştur. Yalnız tıklamayla ilgilenen bir sınıf için bu fazla yazımdır. Java bu durum için hazır sınıflar sunar.

Tanım 9.12

Adaptör sınıfı (*adapter class*), bir dinleyici arayüzünün bütün metotlarını boş gövdelerle uygulayan hazır sınıftır. Ondan türeyen sınıf yalnız ilgilendiği metodu geçersiz kılar.

```
class Dinleyici extends MouseAdapter {
    @Override
    public void mouseClicked(MouseEvent e) {
        System.out.println("tık");
    }
}
// panel.addMouseListener(new Dinleyici());
```

`MouseAdapter` soyut bir sınıftır, ama soyut metodu yoktur; bütün metotları boş gövdelidir. Soyut tanımlanmasının amacı, doğrudan nesnesinin oluşturulmasını engellemektir. `MouseAdapter` üç arayüzü birden uygular: `MouseListener`, `MouseMotionListener` ve `MouseWheelListener`. Tek bir alt sınıf böylece tıklama, hareket ve tekerlek olaylarının hepsini yazabilir.

Hareket olayları ayrı bir arayüzdedir: `MouseMotionListener`. Bu arayüzün iki metodu vardır: `mouseDragged` ve `mouseMoved`. Kendi adaptörü `MouseMotionAdapter`'dır. Kayıt da ayrıdır: tıklama türü olaylar `addMouseListener(…)`, hareket olayları `addMouseMotionListener(…)` ile kaydedilir.

Dinleyici sınıfı ayrı bir dosyada yazmak da gerekmez. Java, tek kullanımlık sınıfı kullanıldığı yerde tanımlamaya izin verir.

Tanım 9.13

Anonim iç sınıf (*anonymous inner class*), adı olmayan, tanımlandığı yerde tek nesnesi oluşturulan iç sınıftır. `new MouseAdapter() { … }` yazımı, `MouseAdapter`'dan türeyen adsız bir sınıf tanımlar ve hemen bir nesnesini oluşturur.

Sık yapılan hata

Bir arayüzü uygulayıp metotlarının bir kısmını yazmamak. `class Dinleyici implements MouseListener` yalnız `mouseClicked`'ı yazarsa derleyici şu hatayı verir:

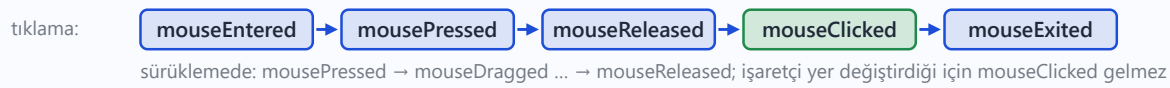
```
Dinleyici.java:6: error: Dinleyici is not abstract and does not override abstract
method mouseExited(MouseEvent) in MouseListener
class Dinleyici implements MouseListener {
```

Derleyici eksik metotlardan yalnız birini adıyla söyler. `mousePressed` de yazılsa ileti yine `mouseExited`'ı gösterir; ileti eksik metotların listesi değildir. Çözüm ya beş metodu da yazmak ya da `MouseAdapter`'dan türetmektir.

9.6.3 Fare olay metotları

Metot	Arayüz	Ne zaman çağrılır?
`mousePressed(e)`	`MouseListener`	Fare düğmesine basıldığı an
`mouseReleased(e)`	`MouseListener`	Düğme bırakıldığında
`mouseClicked(e)`	`MouseListener`	Basıp aynı yerde bırakınca (pressed → released → clicked)
`mouseEntered(e)` / `mouseExited(e)`	`MouseListener`	İşaretçi bileşenin içine girince / dışına çıkınca
`mouseMoved(e)`	`MouseMotionListener`	Düğmeye basmadan hareket edince
`mouseDragged(e)`	`MouseMotionListener`	Düğme basılıyken hareket edince (sürükleme)
`mouseWheelMoved(e)`	`MouseWheelListener`	Tekerlek döndürülünce

Tablo 9.11: Fare olay metotları.



Şekil 9.19: Fare olaylarının sırası. Tıklama üç olay üretir; sürükleme, basma ile bırakma arasında çok sayıda `mouseDragged` üretir.

Bir tıklama tek bir olay değildir. Kullanıcı düğmeye basar: `mousePressed`. Bırakır: `mouseReleased`. Basma ve bırakma aynı noktada olduysa Java bir de `mouseClicked` üretir. Basma ile bırakma arasında fare hareket ettiyse `mouseClicked` gelmez; araya `mouseDragged` olayları girer. Bu yüzden çizim programı gibi uygulamalar tıklamayı değil, basmayı ve sürüklemeyi dinler.

Başsız kipte bir panele sırayla basma, bırakma ve tıklama olayları gönderildiğinde dinleyiciler bu sırayla çağrıldı. Aynı panele hem `Dinleyici` (arayüzü uygulayan) hem `Dinleyici2` (adaptörden türeyen) kaydedildiği için tıklamada iki kez “tık” yazıldı. Bir bileşene birden çok dinleyici kaydedilebilir; olay geldiğinde hepsi çağrılır.

9.6.4 `MouseEvent` sınıfı

Her fare metodu bir `MouseEvent e` parametresi alır. Olayın nerede, hangi düğmeyle ve hangi tuşlar basılıyken olduğu bu nesnededir.

Metot	Döndürdüğü
`getX()`, `getY()`	İşaretçinin bileşene göre konumu
`getPoint()`	Aynı konum, `Point` olarak
`getButton()`	`BUTTON1` sol, `BUTTON2` orta, `BUTTON3` sağ (değerleri 1, 2, 3)
`getClickCount()`	Art arda tıklama sayısı (2: çift tık)

Metot	Döndürdüğü
`isShiftDown()`, `isCtrlDown()`, `isAltDown()`	O anda tuş basılı mı?

Tablo 9.12: `MouseEvent` sınıfının sık kullanılan metotları.

```
panel.addMouseListener(new MouseAdapter() {
    @Override
    public void mousePressed(MouseEvent e) {
        String dugme = "sol";
        if (e.getButton() == MouseEvent.BUTTON3) {
            dugme = "sağ";
        }
        System.out.println(dugme + " (" + e.getX()
                + ", " + e.getY() + ")"
                + (e.isShiftDown() ? " + Shift" : ""));
    }
});
```

Kod bir panele anonim bir adaptör kaydeder. Basılan düğme sağ düğmeyse `dugme` değişkeni "sağ" olur. Ardından düğme adı, konum ve Shift tuşu basılıysa " + Shift" yazılır. Panele `(40, 120)`'de sol tıklandığında `sol (40, 120)`, Shift basılıyken `(70, 30)`'da sağ tıklandığında `sağ (70, 30) + Shift` yazdığı çalıştırılarak görüldü.

Koordinatlar ekrana değil, olayı alan bileşene göredir. Panelin sol üst köşesi, panel pencerenin neresinde olursa olsun, `(0, 0)`'dır. Bu sayede panelin içinde çizilen şekillerle fare koordinatları doğrudan karşılaştırılabilir.

Özetle, bir tıklama `pressed → released → clicked` sırasıyla üç olay üretir; sürüklerken ise çok sayıda `mouseDragged` gelir. Sağ tık `BUTTON3`'tür.

9.6.5 Olay iş parçacığı (EDT)

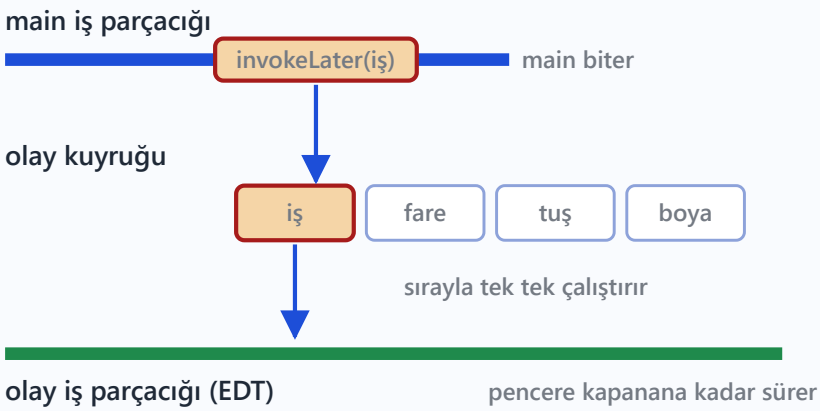
Swing programlarının `main`'i `SwingUtilities.invokeLater` kalıbıyla yazılır. Bu kalıbın nedeni, Swing'in iş parçacıklarıyla ilişkisidir.

> **Tanım 9.14**
>
> **Olay iş parçacığı** (*Event Dispatch Thread*, EDT), Swing bileşenlerinin kurulduğu, değiştirildiği ve olayların işlendiği tek iş parçacığıdır. Olay kuyruğundaki işleri sırayla, birer birer çalıştırır. Java 21'de adı `AWT-EventQueue-0`'dır.

Swing bileşenleri iş parçacığı güvenli değildir (*not thread-safe*). Aynı bileşene aynı anda birden çok iş parçacığı dokunursa bileşenin iç durumu tutarsız hâle gelebilir. Swing bu sorunu kilitlerle değil, basit bir kuralla çözer: bileşenler yalnız olay iş parçacığından kurulur ve değiştirilir. Dinleyici metotlar

zaten EDT'de çalışır; onların içinden bileşenlere dokunmak güvenlidir.

Bir benzetme kuralı açıklar. Mutfakta tek bir aşçı vardır: EDT. Garsonlar yemeği kendileri yapmaz; sipariş fişi bırakır. Fiş bir `Runnable` nesnesidir, yani çalıştırılacak bir kod parçası. Aşçı fişleri geliş sırasıyla, birer birer hazırlar. Tek aşçı olduğu için iki yemek aynı tavaya aynı anda girmez.



Şekil 9.20: `main` iş parçacığı işi `invokeLater` ile olay kuyruğuna koyar ve biter. Olay iş parçacığı kuyruktaki işleri sırayla çalıştırır.

Metot	Davranış
`invokeLater(iş)`	İşi kuyruğa koyar ve hemen döner (asenkron). GUI başlatmanın standart yolu.
`invokeAndWait(iş)`	İş EDT'de bitene kadar bekler (senkron). EDT'nin içinden çağrılırsa `Error` fırlatır: EDT kendini bekleyemez.

Tablo 9.13: `SwingUtilities`'in iki çağırma metodu.

`invokeAndWait`'in EDT içinden çağrılması gerçekten bir `Error` fırlatır. EDT, kuyruğa koyduğu işin bitmesini beklerse o işi çalıştıracak kimse kalmaz ve program sonsuza kadar kilitlenir. Swing bunu önceden yakalar:

```
AWT-EventQueue-0 daemon=false
java.lang.Error: Cannot call invokeAndWait from the event dispatcher thread
```

İlk satır, `invokeAndWait` ile çalıştırılan işin hangi iş parçacığında olduğunu gösterir: `AWT-EventQueue-0`. Bu iş parçacığı arka plan (*daemon*) iş parçacığı değildir. JVM, arka plan olmayan bir iş parçacığı çalıştıkça kapanmaz. GUI programının `main` bittikten sonra da sürmesinin nedeni budur.

9.6.6 Kod izleme: `invokeLater` ve `invokeAndWait`

```
public static void main(String[] a) throws Exception {
  System.out.println("1 main");
  SwingUtilities.invokeLater(
      () -> System.out.println("3 iş A"));
  SwingUtilities.invokeLater(
      () -> System.out.println("4 iş B"));
  System.out.println("2 main devam ediyor");
  SwingUtilities.invokeAndWait(() -> System.out.println(
      "5 iş C, EDT mi? " + SwingUtilities.isEventDispatchThread()));
  System.out.println("6 main, EDT mi? "
      + SwingUtilities.isEventDispatchThread());
}
```

```
1 main
2 main devam ediyor
3 iş A
4 iş B
5 iş C, EDT mi? true
6 main, EDT mi? false
```

Adım	Satır	Değişkenler	Yeni çıktı
1	2: `System.out.println("1 main");`	iş parçacığı=main	`1 main`
2	3: `SwingUtilities.invokeLater(` 4: `() -> System.out.println("3 iş A"));`	kuyruk=[A]	—
3	5: `SwingUtilities.invokeLater(` 6: `() -> System.out.println("4 iş B"));`	kuyruk=[A, B]	—
4	7: `System.out.println("2 main devam ediyor");`	kuyruk=[A, B]	`2 main devam ediyor`
5	8: `SwingUtilities.invokeAndWait(() -> System.out.println(` 9: `"5 iş C, EDT mi? " + SwingUtilities.isEventDispatchThread()));`	kuyruk=[A, B, C]	—
6	4: `() -> System.out.println("3 iş A"));`	kuyruk=[B, C]	`3 iş A`
7	6: `() -> System.out.println("4 iş B"));`	kuyruk=[C]	`4 iş B`
8	8: `SwingUtilities.invokeAndWait(() -> System.out.println(` 9: `"5 iş C, EDT mi? " + SwingUtilities.isEventDispatchThread()));`	kuyruk=[]	`5 iş C, EDT mi? true`
9	10: `System.out.println("6 main, EDT mi? "` 11: `+ SwingUtilities.isEventDispatchThread());`	iş parçacığı=main	`6 main, EDT mi? false`

Tablo 9.14: Olay kuyruğu izlemesi.

Program `main` iş parçacığında başlar ve `1 main` yazar. 2. adımda `invokeLater` iş A'yı kuyruğa koyar ve hemen döner. Olay iş parçacığı henüz yoksa bu ilk çağrı onu başlatır. 3. adımda iş B de kuyruğa eklenir; `main` yine beklemez. 4. adımda `main` kendi işine devam eder ve `2 main devam ediyor` yazar.



Şekil 9.21: 5. adım. `invokeAndWait` iş C'yi kuyruğa koydu ve `main`'i durdurdu. Kuyrukta A, B ve C bekliyor.

5. adımda `invokeAndWait` iş C'yi kuyruğa koyar ve `main`'i durdurur. EDT kuyruğu baştan işler: önce A (`3 iş A`), sonra B (`4 iş B`), sonra C. Kuyruk sıralıdır; işler kuyruğa giriş sırasıyla çalışır. İş C, `isEventDispatchThread()` ile kendi iş parçacığını sorar ve `true` alır. C bitince bekleyen `main` uyandırılır. Son satırda `main` aynı soruyu sorar ve `false` alır.

Çıktının sırası hakkında dikkatli olmak gerekir. 1, 2, 3, 4, 5, 6 sırası bu çalıştırmada görüldü. Program 20 kez çalıştırıldığında da sıra her seferinde aynı çıktı. Ama sıranın bir kısmı garanti değildir. `2 main devam ediyor` satırı ile `3 iş A` satırı farklı iş parçacıklarından gelir. EDT, `main` 2. satırı yazmadan önce A'yı çalıştırabilirdi. Kesin olan şunlardır: A, B'den önce; B, C'den önce; C, 6'dan önce çalışır. Aynı ayrım `invokeLater(B)`, `println(A)`, `invokeAndWait(C)`, `println(D)` dizisinde de geçerlidir. B, C'den ve C, D'den önce gelir. 20 denemede hep `ABCD` görüldü, ama A ile B'nin yeri çalıştırmaya göre değişebilir.

Örnek 9.11

Bir düğme dinleyicisi sırasıyla `System.out.println("1")`, `SwingUtilities.invokeLater(() -> System.out.println("3"))` ve `System.out.println("2")` çalıştırıyor. Çıktı nedir? Sıra kesin midir?

`1`, `2`, `3`. Dinleyici EDT'de çalışır. `invokeLater` işi kuyruğun sonuna koyar ve döner; dinleyici `2`'yi yazar ve biter. EDT ancak bundan sonra kuyruktaki sıradaki işi alabilir. Bu kez sıra kesindir, çünkü iki parça da aynı iş parçacığında çalışır. Kuyruğa konan iş, o anda çalışan işten önce başlayamaz.

Sık yapılan hata

Arayüzü doğrudan `main` iş parçacığında kurmak. Küçük programlarda çoğu zaman sorun çıkmaz; bu yüzden hata uzun süre fark edilmez. Ama pencere görünür olduktan sonra EDT çalışmaya başlar ve `main` ile aynı bileşenlere dokunabilir. Sonuç, nadiren ve rastgele görülen çizim

bozuklukları ya da istisnalardır. Güvenli kalıp tektir: `SwingUtilities.invokeLater(() -> { … })`.

9.7 Uygulama: çizim programı

Bu uygulama, fareyle basılı tutup sürükleyerek çizim yapılan bir programdır. Çalışma ilkesi üç cümleyle özetlenir:

- Basınca (`mousePressed`) yeni bir eğri başlar.
- Sürükledikçe (`mouseDragged`) eğrinin sonuna nokta eklenir ve yeniden çizim istenir.
- Çizerken (`paintComponent`) her eğrinin ardışık noktaları çizgilerle birleştirilir.

Şekil 9.22: Fareyle çizim yapılan panel.

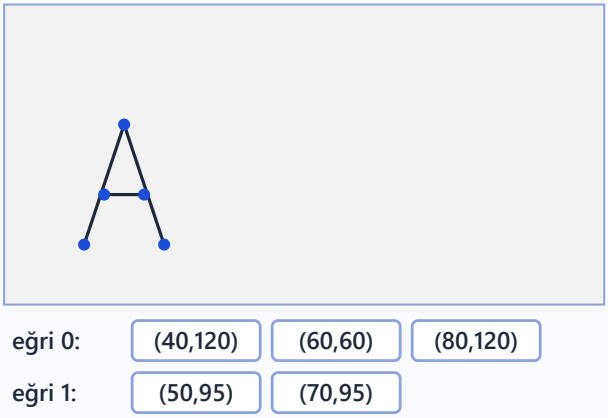
Programın temel tasarım kararı, çizimi bir resim olarak değil, noktaların listesi olarak saklamaktır. Ekran her seferinde bu veriden yeniden çizilir. Pencere küçültülüp büyütülse de, üstüne başka bir pencere gelip çekilse de çizim kaybolmaz. Veri ile görüntünün ayrılması, olay döngüsünün kuralını uygular: dinleyiciler veriyi değiştirir, `paintComponent` veriyi çizer.

9.7.1 Eğrileri saklamak

```
public class BasitCizimPaneli extends JPanel {
    private List<List<Point>> egriler = new ArrayList<>();

    public BasitCizimPaneli() {
        addMouseListener(new MouseAdapter() {
            @Override
            public void mousePressed(MouseEvent e) {
                var yeniEgri = new ArrayList<Point>();
                yeniEgri.add(new Point(e.getX(), e.getY()));
                egriler.add(yeniEgri);
            }
        });
        addMouseMotionListener(new MouseMotionAdapter() {
            @Override
            public void mouseDragged(MouseEvent e) {
                egriler.get(egriler.size() - 1)
                       .add(new Point(e.getX(), e.getY()));
                repaint(0, 0, getWidth(), getHeight());
            }
        });
    }
}
```

Sınıfın tek alanı `egriler`'dir. Tipi `List<List<Point>>`, yani listelerin listesidir. Dış liste eğrileri tutar. Her iç liste bir eğrinin noktalarını sırayla tutar.



Şekil 9.23: İki eğrili bir çizim ve onu tutan veri. Eğri 0 üç, eğri 1 iki noktadan oluşur.

Yapıcı iki dinleyici kaydeder. Birincisi `MouseAdapter`'dan türeyen anonim bir sınıftır ve yalnız `mousePressed`'ı yazar. Metot yeni ve boş bir nokta listesi oluşturur, basılan noktayı listeye ekler ve listeyi dış listeye koyar. İkinci dinleyici `MouseMotionAdapter`'dan türer ve yalnız `mouseDragged`'ı

yazar. Metot dış listenin son öğesini, yani en son başlayan eğriyi alır ve sonuna yeni noktayı ekler. Ardından paneli yeniden çizdirir.

Son eğriye `egriler.get(egriler.size() - 1)` ile ulaşılır. Liste indisleri 0'dan başladığı için son öğenin indisi `size() - 1`'dir. `get(size())` her zaman listenin bir ötesini ister.

Sık yapılan hata

Son öğeyi `get(size())` ile istemek. Listede bir eğri varken bu çağrı şu istisnayı fırlatır:

```
java.lang.IndexOutOfBoundsException: Index 1 out of bounds for length 1
```

Hata derleme anında yakalanmaz; ilk sürüklemede ortaya çıkar. Doğru indis `size() - 1`'dir.

Kodun bir de içe aktarma tuzağı vardır. `java.awt.*` ve `java.util.*` birlikte içe aktarılırsa `List` adı iki pakette birden bulunur: `java.awt.List` bir liste kutusu bileşeni, `java.util.List` ise koleksiyon arayüzüdür. Derleyici hangisinin kastedildiğini bilemez:

```
Ana.java:4: error: reference to List is ambiguous
  List<String> x;
  both interface java.util.List in java.util and class java.awt.List in java.awt match
```

Çözüm, kastedilen sınıfı ayrıca açıkça içe aktarmaktır: `import java.util.List;`. Tek tip içe aktarma, yıldızlı içe aktarmadan önceliklidir.

9.7.2 Kod izleme: fare olayları

Kullanıcı şu hareketleri yapsın: `(40, 120)`'de basar, `(60, 60)` ve `(80, 120)`'ye sürükler, bırakır; sonra `(50, 95)`'te yeniden basar ve `(70, 95)`'e sürükler.

```
addMouseListener(new MouseAdapter() {
    @Override
    public void mousePressed(MouseEvent e) {
        var yeniEgri = new ArrayList<Point>();
        yeniEgri.add(new Point(e.getX(), e.getY()));
        egriler.add(yeniEgri);
    }
});
addMouseMotionListener(new MouseMotionAdapter() {
    @Override
    public void mouseDragged(MouseEvent e) {
        egriler.get(egriler.size() - 1)
               .add(new Point(e.getX(), e.getY()));
        repaint(0, 0, getWidth(), getHeight());
    }
});
```

Adım	Satır	Değişkenler	Açıklama
1	4: `var yeniEgri = new ArrayList<Point>();` 5: `yeniEgri.add(new Point(e.getX(), e.getY()));` 6: `egriler.add(yeniEgri);`	e.getX()=40, e.getY()=120, egriler.size()=1	**mousePressed**: Basıldı (40, 120): yeni bir iç liste oluşur, ilk noktası eklenir, dış listeye konur.
2	12: `egriler.get(egriler.size() - 1)` 13: `.add(new Point(e.getX(), e.getY()));` 14: `repaint(0, 0, getWidth(), getHeight());`	e.getX()=60, e.getY()=60, egriler.size()=1	**mouseDragged**: Sürüklendi (60, 60): son eğriye nokta eklenir; `repaint` yeniden çizimi ister, çizgi görünür.
3	12: `egriler.get(egriler.size() - 1)` 13: `.add(new Point(e.getX(), e.getY()));` 14: `repaint(0, 0, getWidth(), getHeight());`	e.getX()=80, e.getY()=120, egriler.size()=1	**mouseDragged**: Sürüklendi (80, 120): aynı eğri uzar. Gerçekte saniyede onlarca `mouseDragged` gelir.
4	1: `addMouseListener(new MouseAdapter() {`	e.getX()=80, e.getY()=120, egriler.size()=1	**mouseReleased**: Bırakıldı: `mouseReleased`'ı yazmadık; `MouseAdapter`'ın **boş gövdesi** çalışır, hiçbir şey olmaz.

Adım	Satır	Değişkenler	Açıklama
5	4: `var yeniEgri = new ArrayList<Point>();` 5: `yeniEgri.add(new Point(e.getX(), e.getY()));` 6: `egriler.add(yeniEgri);`	e.getX()=50, e.getY()=95, egriler.size()=2	**mousePressed**: Yeniden basıldı (50, 95): **ikinci** eğri başlar. Eski eğri olduğu gibi kalır.
6	12: `egriler.get(egriler.size() - 1)` 13: `.add(new Point(e.getX(), e.getY()));` 14: `repaint(0, 0, getWidth(), getHeight());`	e.getX()=70, e.getY()=95, egriler.size()=2	**mouseDragged**: Sürüklendi (70, 95): ekleme yine **son** eğriye, yani eğri 1'e yapılır.

Tablo 9.15: Fare olaylarının izlemesi.

1. adımda dış liste boştur. `mousePressed` yeni bir iç liste oluşturur, `(40, 120)`'yi ekler ve iç listeyi dış listeye koyar: `egriler.size()` 1 olur. 2. ve 3. adımlarda `mouseDragged` iki nokta ekler. İkisi de son eğriye, yani eğri 0'a gider. Her eklemeden sonra `repaint` yeniden çizim ister ve yeni çizgi parçası ekranda görünür.

Gerçek kullanımda olay sayısı çok daha fazladır. Fare hareket ettikçe işletim sistemi saniyede onlarca konum bildirir ve her biri bir `mouseDragged` olur. İzlemedeki iki sürükleme, bu akışın iki örneğidir.

4. adımda kullanıcı fareyi bırakır. `mouseReleased`'ı yazmadık. Olay yine gelir, ama `MouseAdapter`'ın boş gövdesi çalışır ve hiçbir şey olmaz. 5. adımda yeni bir basma ikinci bir iç liste başlatır; eski eğri olduğu gibi kalır. 6. adımdaki sürükleme yine son eğriye, artık eğri 1'e eklenir.

Olay dizisi başsız kipte panele doğrudan gönderilerek denendi. Her adımdan sonra yazılan veri, tablonun son durumunu verdi: `[[(40,120), (60,60), (80,120)], [(50,95), (70,95)]]`.

9.7.3 Kod izleme: eğrileri çizmek

```
@Override
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    for (var egri : egriler) {
        var onceki = egri.get(0);
        for (var nokta : egri) {
            g.drawLine(onceki.x, onceki.y, nokta.x, nokta.y);
            onceki = nokta;
        }
    }
}
```

Metot iç içe iki döngü kullanır. Dış döngü eğriler, iç döngü bir eğrinin noktaları üzerinde gezinir. `onceki` değişkeni, çizginin başlayacağı noktayı tutar. Her eğrinin başında `onceki` o eğrinin ilk noktasına ayarlanır. İç döngü her noktada `onceki` 'den o noktaya bir çizgi çizer ve `onceki` 'yi ilerletir.

Adım	Satır	Değişkenler	Yeni çıktı
1	3: `super.paintComponent(g);`	—	—
2	4: `for (var egri : egriler) {` 5: `var onceki = egri.get(0);`	egri=eğri 0, onceki=(40,120)	—
3	6: `for (var nokta : egri) {` 7: `g.drawLine(onceki.x, onceki.y, nokta.x, nokta.y);` 8: `onceki = nokta;`	egri=eğri 0, onceki=(40,120), nokta=(40,120)	`drawLine(40, 120, 40, 120)`
4	6: `for (var nokta : egri) {` 7: `g.drawLine(onceki.x, onceki.y, nokta.x, nokta.y);` 8: `onceki = nokta;`	egri=eğri 0, onceki=(40,120), nokta=(60,60)	`drawLine(40, 120, 60, 60)`
5	6: `for (var nokta : egri) {` 7: `g.drawLine(onceki.x, onceki.y, nokta.x, nokta.y);` 8: `onceki = nokta;`	egri=eğri 0, onceki=(60,60), nokta=(80,120)	`drawLine(60, 60, 80, 120)`
6	4: `for (var egri : egriler) {` 5: `var onceki = egri.get(0);`	egri=eğri 1, onceki=(50,95)	—
7	6: `for (var nokta : egri) {` 7: `g.drawLine(onceki.x, onceki.y, nokta.x, nokta.y);` 8: `onceki = nokta;`	egri=eğri 1, onceki=(50,95), nokta=(50,95)	`drawLine(50, 95, 50, 95)`
8	6: `for (var nokta : egri) {` 7: `g.drawLine(onceki.x, onceki.y, nokta.x, nokta.y);` 8: `onceki = nokta;`	egri=eğri 1, onceki=(50,95), nokta=(70,95)	`drawLine(50, 95, 70, 95)`

Tablo 9.16: `paintComponent` izlemesi; "Yeni çıktı" sütunu yapılan çizim çağrısıdır.

3. adımda bir ayrıntı göze çarpar. İç döngü ilk noktadan başlar ve `onceki` de ilk noktadır. İlk çağrı `drawLine(40, 120, 40, 120)` olur: nokta kendisiyle birleşir ve tek bir piksel çizilir. Ardından `onceki = nokta` yapılır. 4. ve 5. adımlar gerçek çizgileri çizer: `(40,120) → (60,60)` ve `(60,60) → (80,120)` .



Şekil 9.24: 8. adım. Beş `drawLine` çağrısıyla çizim tamamlandı; iki eğri birbirine bağlanmadı.

6. adımda dış döngü ikinci eğriye geçer ve `onceki` yeniden bu eğrinin ilk noktası olur. İki eğrinin birleşmemesi bu satırın işidir. `onceki` her eğride sıfırlanmasaydı, eğri 0'ın son noktası `(80, 120)` ile eğri 1'in ilk noktası `(50, 95)` arasında istenmeyen bir çizgi çizilirdi. Toplam çağrı sayısı 3 + 2 = 5'tir; görünen çizgi parçası ise 2 + 1 = 3'tür.

`DebugGraphics` kaydı da aynı beş çağrıyı verdi:

```
setColor(238, 238, 238)
fillRect(0, 0, 200, 200)
drawLine(40, 120, 40, 120)
drawLine(40, 120, 60, 60)
drawLine(60, 60, 80, 120)
drawLine(50, 95, 50, 95)
drawLine(50, 95, 70, 95)
```

İlk iki satırı yine `super.paintComponent(g)` üretti: zemin, panelin arka plan rengiyle temizlendi. Önceki çizimden iz kalmaz. Yeni görüntü her seferinde boş bir zeminden, verinin tamamından kurulur.

Örnek 9.12

Kullanıcı `(10, 10)` 'da basar, `(20, 10)` ve `(30, 20)` 'ye sürükler, bırakır; `(5, 5)` 'te basar, `(6, 6)` 'ya sürükler, bırakır. `egriler.size()` , her eğrinin nokta sayısı ve yeniden çizimdeki `drawLine` sayısı nedir?

Her basış bir iç liste eklediği için `egriler.size()` 2'dir. Eğri 0'da üç nokta, eğri 1'de iki nokta vardır. `drawLine` sayısı nokta sayısına eşittir: 3 + 2 = 5. Kayıt şöyledir:

```
setColor(238, 238, 238)
fillRect(0, 0, 200, 200)
drawLine(10, 10, 10, 10)
drawLine(10, 10, 20, 10)
drawLine(20, 10, 30, 20)
drawLine(5, 5, 5, 5)
drawLine(5, 5, 6, 6)
```

9.7.4 `main` ve `repaint`

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        var frame = new JFrame("Basit Çizim Programı");
        frame.getContentPane().add(new BasitCizimPaneli(),
                BorderLayout.CENTER);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(400, 300);
        frame.setVisible(true);
    });
}
```

`main` bilinen kalıbı izler: arayüz `invokeLater` ile EDT'de kurulur, panel içerik bölmesinin merkezine eklenir, pencere 400 × 300 boyutunda gösterilir.

`repaint` metodu iki biçimde kullanılır ve ikisi de bir istektir:

- `repaint()` çizimi hemen yapmaz; kuyruğa bir "yeniden boya" isteği koyar. Art arda gelen istekler tek bir boyamada birleştirilebilir. Fare saniyede onlarca olay üretirken bu birleştirme gereksiz çizimleri önler.
- `repaint(x, y, en, boy)` yalnız o dikdörtgenin yeniden çizilmesini ister. Swing bu bölgenin dışını kırpar (*clipping*): `g` bölgenin dışına boyamaz ve iş azalır. `repaint(0, 0, getWidth(), getHeight())`, `repaint()` ile aynıdır.

Tanım 9.15

`repaint()`, bir bileşenin yeniden çizilmesi için olay kuyruğuna istek bırakan metottur. Swing uygun anda bileşenin `paintComponent` metodunu çağırır. Programcı `paintComponent`'i kendisi çağırmaz; veriyi değiştirir ve `repaint()` ister.

Sık yapılan hata

Dinleyicinin içinden `paintComponent(getGraphics())` çağırmak. Bu çağrı Swing'in boyama düzenini atlar. Çift arabellekleme, yani görüntünün önce bellekte hazırlanması devre dışı kalır ve görüntü titreyebilir. Bir sonraki gerçek boyamada da çizilen her şey silinir. Doğru yol `repaint()` 'tir.

9.8 Düğme ve metin bileşenleri

9.8.1 `JButton` sınıfı

`JButton`, tıklanabilir düğmeler oluşturur. Metni `setText()`, ikonu `setIcon()` ile belirlenir. `setEnabled(false)` düğmeyi pasif yapar: düğme soluk görünür ve tıklanınca olay üretmez.

Düğmenin tıklanmasına yanıt vermek için `addActionListener()` ile bir dinleyici eklenir. Dinleyici `ActionListener` arayüzünü uygular. Bu arayüzün tek metodu vardır: `actionPerformed(ActionEvent e)`. Düğme, klavyeden boşluk tuşuyla da tetiklenebilir; iki durumda da aynı metot çağrılır. Bu yüzden düğme için fare olayı değil, “eylem” olayı dinlenir.

Dinleyici eklemek, Gözlemci (*Observer*) tasarım deseninin bir örneğidir. Düğme “konu”dur; kendisine kayıtlı gözlemcilerin listesini tutar. Bir tıklama olunca listedeki her gözlemciye haber verir. Düğme, gözlemcilerin ne yaptığını bilmez. Yeni bir davranış eklemek için düğmenin koduna dokunmak gerekmez; yalnız yeni bir gözlemci kaydedilir.

```
class SayacPaneli extends JPanel {
    private int sayac = 0;
    private JLabel etiket = new JLabel("Henüz tıklanmadı");

    SayacPaneli() {
        JButton dugme = new JButton("Tıkla");
        dugme.addActionListener(e -> {
            sayac++;
            etiket.setText(sayac + " kez tıklandı");
        });
        add(dugme);
        add(etiket);
    }
}
```

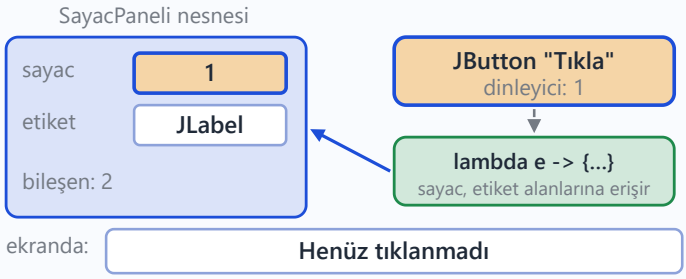
9.8.2 Kod izleme: sayaç düğmesi

Adım	Satır	Çağrı yığını	Değişkenler	Açıklama
1	2: `private int sayac = 0;` 3: `private JLabel etiket = new JLabel("Henüz tıklanmadı");`	main() → new SayacPaneli()	sayac=0, etiket=JLabel	Nesne oluşurken önce alanlara ilk değerleri verilir: `sayac` 0, etiket hazır.
2	6: `JButton dugme = new JButton("Tıkla");`	main() → SayacPaneli()	dugme=JButton("Tıkla")	Yapıcı bir düğme oluşturur; `dugme` yerel değişkendir.
3	7: `dugme.addActionListener(e -> {` 8: `sayac++;` 9: `etiket.setText(sayac + " kez tıklandı");` 10: `});`	main() → SayacPaneli()	dinleyici sayısı=1	Lambda **çalıştırılmaz**, yalnızca düğmeye kaydedilir. Gövdesi tıklanınca çalışacak.
4	11: `add(dugme);` 12: `add(etiket);`	main()	bileşen sayısı=2	Düğme ve etiket panele eklenir (`FlowLayout` : yan yana). Yapıcı biter.
5	8: `sayac++;`	EDT → actionPerformed(e)	sayac=1	Kullanıcı tıklar: EDT `actionPerformed` 'ı, yani lambda gövdesini çağırır. `sayac` bir **alan** olduğu için artırılabilir.
6	9: `etiket.setText(sayac + " kez tıklandı");`	EDT → actionPerformed(e)	sayac=1	`setText` etiketin metnini değiştirir; Swing etiketi yeniden boyar.
7	8: `sayac++;`	EDT → actionPerformed(e)	sayac=2	İkinci tıklama: aynı lambda yeniden çağrılır, alan 2 olur.
8	9: `etiket.setText(sayac + " kez tıklandı");`	EDT → actionPerformed(e)	sayac=2	Etiket güncellenir. Tıklamalar arasında program hiçbir kod çalıştırmaz; yalnızca **olay bekler**.

Tablo 9.17: Sayaç düğmesinin izlemesi.

1. adımda `new SayacPaneli()` çalışmaya başlar. Yapıcının gövdesinden önce alanlara ilk değerleri verilir: `sayac` 0 olur, `etiket` "Henüz tıklanmadı" metniyle oluşturulur. 2. adımda yapıcı bir düğme oluşturur. `dugme` yerel bir değişkendir; yapıcı bitince değişken kaybolur, ama düğme panelin bileşeni olarak yaşamaya devam eder.

3. adım izlemenin en önemli adımıdır. Lambda ifadesi bu anda çalıştırılmaz. Yalnız bir nesne olarak düğmeye kaydedilir. Gövdesindeki `sayac++` ve `setText` ancak tıklanınca çalışacaktır. 4. adımda düğme ve etiket panele eklenir. `FlowLayout` onları yan yana dizer. Yapıcı biter; panelin bileşen sayısı 2, düğmenin dinleyici sayısı 1'dir.



Şekil 9.25: 5. adım. Kullanıcı tıkladı; EDT düğmeye kayıtlı dinleyicinin `actionPerformed`'ını, yani lambda gövdesini çağırdı.

5. adımda kullanıcı tıklar. EDT `actionPerformed`'ı, yani lambda gövdesini çağırır. `sayac` bir alan olduğu için artırılabilir: 1 olur. 6. adımda `setText` etiketin metnini "1 kez tıklandı" yapar ve Swing etiketi yeniden boyar. 7. ve 8. adımlar ikinci tıklamadır: aynı lambda yeniden çağrılır, alan 2 olur, etiket güncellenir.

Tıklamalar arasında program hiçbir kod çalıştırmaz; yalnız olay bekler. Bu davranış, konsol programının sürekli çalışan akışından çok farklıdır. Başsız kipte panel oluşturulup düğmenin `doClick` metoduyla iki tıklama verildiğinde etiket "1 kez tıklandı" ve "2 kez tıklandı" oldu. Düğme `setEnabled(false)` ile pasif yapılıp yeniden tıklandığında etiket değişmedi.

```
2 1 Henüz tıklanmadı
1 kez tıklandı
2 kez tıklandı
pasif: 2 kez tıklandı
```

9.8.3 Anonim sınıf ve lambda

Aynı dinleyici iki biçimde yazılabilir. Klasik yazım anonim iç sınıftır:

```
dugme.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        sayac++;
        etiket.setText("Sayı: " + sayac);
    }
});
```

Java 8'den beri aynı iş bir lambda ifadesiyle yazılır:

```
dugme.addActionListener(e -> {
    sayac++;
    etiket.setText("Sayı: " + sayac);
});

// tek ifade ise süslü parantez gerekmez:
dugme.addActionListener(e -> System.exit(0));
```

Lambdanın kullanılabilmesinin nedeni `ActionListener`'ın yapısıdır. Arayüzün yalnız bir soyut metodu vardır.

Tanım 9.16

Fonksiyonel arayüz (*functional interface*), tek soyut metodu olan arayüzdür. Bir **lambda ifadesi** (*lambda expression*), böyle bir arayüzün tek metodunun gövdesi olarak kabul edilir: `e -> …`, `actionPerformed(e)`'nin gövdesidir.

Derleyici `addActionListener`'ın parametre tipine bakar, `ActionListener`'ı görür ve tek metodunu bulur. Lambdanın parametresi `e`, o metodun `ActionEvent` parametresi olur. Gövde tek bir ifadeyse süslü parantez de gerekmez: `dugme.addActionListener(e -> System.exit(0));`.

`MouseListener`'ın beş metodu olduğu için lambda ile yazılamaz. Derleyici hangi metodun gövdesinin yazıldığını bilemez:

```
Ana.java:9: error: incompatible types: MouseListener is not a functional interface
  p.addMouseListener(e -> System.out.println("tık"));
    multiple non-overriding abstract methods found in interface MouseListener
```

Fare olayları için bu yüzden adaptör sınıfı kullanılır.

Lambdanın bir sınırı daha vardır: içinden yerel değişken değiştirilemez. Lambda, kullandığı yerel değişkenin değerini yakalar (kopyalar); değişkenin kendisine bağlanmaz. Bu yüzden yerel değişken final ya da "fiilen final" olmalıdır, yani tanımlandıktan sonra hiç değiştirilmemelidir.

Sık yapılan hata

Sayacı yerel değişken olarak tutmak: `int sayac = 0; … d.addActionListener(e -> sayac++);`. Derleyici şu iletiyi verir:

```
Ana.java:10: error: local variables referenced from a lambda expression must be
final or effectively final
  d.addActionListener(e -> sayac++);
```

Sayaç gibi değişen değerler sınıfın alanı (*field*) olmalıdır. Alanlar nesneye aittir; lambda nesneye bir referans tutar ve alanı değiştirebilir. Sayaç paneli bu yüzden `private int sayac` alanını kullanır.

9.8.4 `JLabel` ve `JTextField`

`JLabel` bir etikettir. Metin ya da resim gösterir; kullanıcı onu değiştiremez. `setText()` metni, `setIcon()` resmi, `setFont()` yazı tipini, `setForeground()` metin rengini belirler.

`JTextField` tek satırlık metin alanıdır. Kullanıcının metin girmesi içindir; formlardan veri bu yolla toplanır. `getText()` ve `setText()` alanı okur ve yazar. `setColumns()` görünen sütun sayısını belirler. Bu sayı alanın tercih ettiği genişliği etkiler, alabileceği metnin uzunluğunu sınırlamaz. Başsız kipte 12 sütunluk alan 136, 24 sütunluk alan 268 piksel genişlik istedi. `addActionListener()` ile eklenen dinleyici, kullanıcı alanda Enter'a basınca çalışır.

```
JLabel etiket = new JLabel("Adınız:");
etiket.setFont(new Font("SansSerif", Font.BOLD, 14));
etiket.setForeground(Color.BLUE);
JTextField adAlani = new JTextField(12);
JLabel selam = new JLabel(" ");
adAlani.addActionListener(e -> selam.setText(
        "Merhaba, " + adAlani.getText() + "!"));
```

Kod iki etiket ve bir metin alanı oluşturur. Birinci etiket kalın, 14 puntoluk ve mavidir. Metin alanı 12 sütun genişliğindedir. İkinci etiket başta bir boşluk içerir: boş metinli bir etiketin tercih ettiği boyut 0 × 0'dır. Tek boşluklu etiket ise 3 × 16 piksel yer tutar ve selamın satırını baştan ayırır. Dinleyici, Enter'a basılınca alandaki metni okur ve selam etiketine yazar. Başsız kipte alana “Ayşe” yazılıp Enter olayı üretildiğinde selam etiketi `Merhaba, Ayşe!` oldu.

Lambda içinde `adAlani` ve `selam` yerel değişkenleri kullanılır. Bu değişkenler tanımlandıktan sonra değiştirilmediği için fiilen finaldir ve lambda onları yakalayabilir. Değişkenin gösterdiği nesnenin içeriği değişebilir; `selam.setText(…)` buna örnektir. Kural yalnız değişkenin başka bir nesneye yönlendirilmesini yasaklar.

9.8.5 `JTextArea` ve `JScrollPane`

`JTextArea` çok satırlı metin alanıdır. İçeriği okunabilir, atanabilir ve güncellenebilir. `new JTextArea(satır, sütun)` ile görünür boyut önceden verilir. Sık kullanılan metotları şunlardır: `getText()`, `setText()`, `append()` (sona ekler), `setEditable(false)` (salt okunur yapar) ve `setLineWrap(true)` (uzun satırları kaydırır).

`JScrollPane` bir kaydırma bölmesidir. İçine konan bileşen (metin alanı, tablo, resim) ekrana sığmadığında kaydırma çubukları ekler. İçeriğin boyutu değişince kendini uyarlar. `setPreferredSize()` ile kendi boyutu, `setViewportView()` ile içeriği değiştirilir. Metin alanı kaydırma bölmesine yapıcıda ve-

rilir ve pencereye metin alanı değil, kaydırma bölmesi eklenir.

```
JTextArea alan = new JTextArea(5, 20);
alan.setText("Birinci satır\n");
alan.append("İkinci satır\n");
JScrollPane kaydirma = new JScrollPane(alan);
System.out.print(alan.getText());
System.out.println(alan.getLineCount());
```

```
Birinci satır
İkinci satır
3
```

`setText` eski içeriği siler ve yenisini yazar; `append` sona ekler. Metin `"Birinci satır\nİkinci satır\n"` olur. `getLineCount()` satır sonu sayısının bir fazlasını döndürür: metinde iki satır sonu vardır, sonuç 3'tür. Üçüncü satır boştur, ama imleç oraya konabildiği için sayılır. Boş bir metin alanında bile `getLineCount()` 1'dir.

Örnek 9.13

Aşağıdaki kod hangi üç satırı yazdırır?

```
JTextArea a = new JTextArea("x");
a.append("\ny");
a.append("\n");
System.out.println(a.getLineCount());
a.setText("tek");
a.append(" satır");
System.out.println(a.getText() + " " + a.getLineCount());
JTextField f = new JTextField(10);
f.setText(" 42 ");
System.out.println(Integer.parseInt(f.getText().trim()) + 1);
```

Metin `"x\ny\n"` olur: iki satır sonu, üç satır. İlk satır `3`'tür. `setText` eski içeriği siler, `append` ekler: `"tek satır"`, satır sonu yok, satır sayısı 1. `getText()` her zaman `String` döndürür. `trim()` baştaki ve sondaki boşlukları atar; `42 + 1 = 43`. Boşluklar atılmasaydı `parseInt` şu istisnayı fırlatırdı: `NumberFormatException: For input string: " 42 "`.

```
3
tek satır 1
43
```

9.9 Uygulama: metin dönüştürücü

Bu uygulama, bir metin alanındaki metni iki düğmeyle küçük ya da büyük harfe çevirir. Pencerenin düzeni iki yerleşim yöneticisinin birlikte kullanımına iyi bir örnektir:

- Ortada (`CENTER`) bir `JScrollPane` içinde 8 satır × 50 sütunluk bir `JTextArea` vardır.
- Altta (`SOUTH`) iki düğmeyi taşıyan bir `JPanel` vardır. Panel `FlowLayout` kullandığı için düğmeler ortalı ve yan yanadır.
- `pack()` pencere boyutunu bu bileşenlerin tercih ettiği boyuttan hesaplar.
- Düğmeler metni `toLowerCase()` ya da `toUpperCase()` ile dönüştürür ve alana geri yazar.

```
import java.awt.BorderLayout;
import javax.swing.*;

public class BuyukHarfeCevirici {
    public static void main(String[] args) {
        SwingUtilities.invokeLater(() -> {
            var alan = new JTextArea("Java Programlama", 8, 50);
            var kucukHarfButton = new JButton("Küçük Harfe Çevir");
            var buyukHarfButton = new JButton("Büyük Harfe Çevir");
            kucukHarfButton.addActionListener(e ->
                    alan.setText(alan.getText().toLowerCase()));
            buyukHarfButton.addActionListener(e ->
                    alan.setText(alan.getText().toUpperCase()));
            var panel = new JPanel();
            panel.add(kucukHarfButton);
            panel.add(buyukHarfButton);
            var frame = new JFrame("BuyukHarfeCevirici");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.getContentPane().add(new JScrollPane(alan),
                    BorderLayout.CENTER);
            frame.getContentPane().add(panel, BorderLayout.SOUTH);
            frame.pack();
            frame.setVisible(true);
        });
    }
}
```

Bütün arayüz tek bir lambda içinde, EDT'de kurulur. Önce metin alanı ve iki düğme oluşturulur. Her düğmeye bir dinleyici eklenir. Dinleyici alanın metnini okur, dönüştürür ve sonucu alana geri yazar. Sonra düğmeler bir panele konur. Pencere oluşturulur, kaydırma bölmesi merkeze, düğme paneli güneye eklenir. `pack()` boyutu hesaplar ve pencere gösterilir.

Kodda üç ayrıntı vardır:

- `alan` yerel bir değişkendir, ama lambda içinde kullanılabilir. Değişken tanımlandıktan sonra başka bir nesneye yönlendirilmez; fiilen finaldir. Lambda `alan` değişkeninin kendisini değil, gösterdiği nesneyi kullanır ve o nesnenin metnini değiştirir.
- `toUpperCase()` yeni bir `String` döndürür. `String` değiştirilemez olduğu için asıl metin değişmez; sonucu `setText` ile alana geri yazmak gerekir. `alan.getText().toUpperCase();` tek başına yazılsaydı ekranda hiçbir şey değişmezdi.
- `BorderLayout.CENTER` tam adıyla yazılır. Yalnız `CENTER` yazmak derlenmez, çünkü sabit `BorderLayout` sınıfının içindedir.

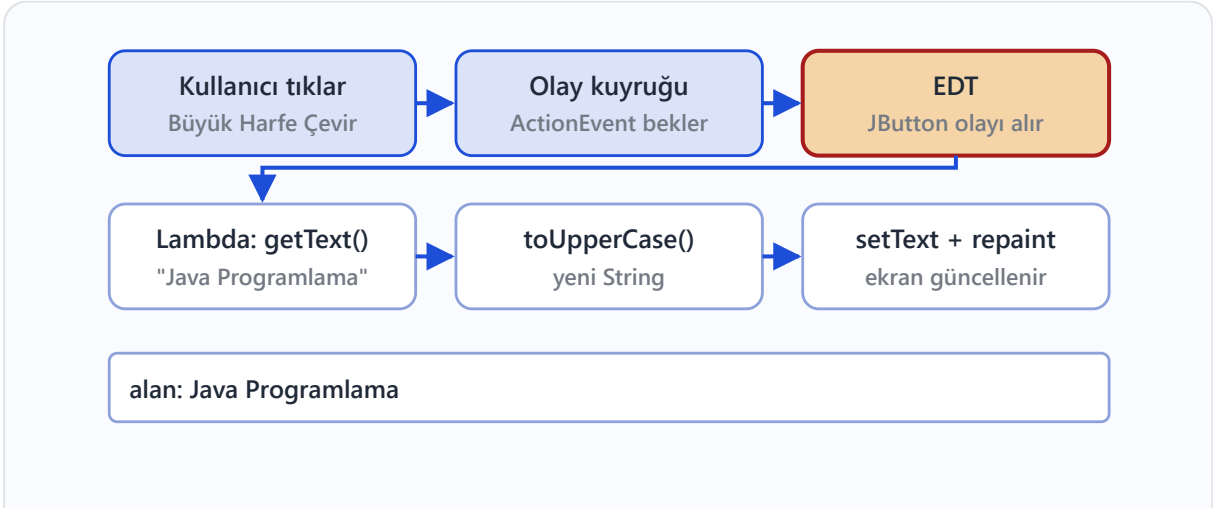
Başsız kipte aynı iki düğme ve metin alanı oluşturulup düğmelere `doClick` ile basıldığında metin önce büyük, sonra küçük harfe döndü:

```
JAVA PROGRAMLAMA
java programlama
varsayılan yerel ayar tr_TR: İSTANBUL / ENGLISH ISTANBUL / tr İSTANBUL
```

9.9.1 Bir tıklamada neler olur?

Kullanıcının “Büyük Harfe Çevir” düğmesine tıklamasıyla ekranda metnin değişmesi arasında altı adım vardır.

1. Kullanıcı düğmeye tıklar; işletim sistemi bunu Java'ya iletir.
2. Java bir olay nesnesi oluşturur ve olay kuyruğuna koyar.
3. EDT sıradaki olayı alır. Düğme, kayıtlı bütün `ActionListener`'ların `actionPerformed`'ını çağırır: lambdamız çalışır.
4. `alan.getText()` mevcut metni okur.
5. `toUpperCase()` yeni bir dizgi üretir; eski dizgi değişmez.
6. `setText` alanın modelini günceller. Alan kendini `repaint` ile yeniden çizdirir. EDT sıradaki olaya geçer.



Şekil 9.26: 3. adım. EDT olayı kuyruktan aldı ve düğmeye kayıtlı dinleyiciyi çağırdı.

6. adımda "model" sözcüğü geçer. Swing'de metin bileşenleri iki parçadan oluşur: metni tutan bir model ve onu ekranda gösteren bir görünüm. `setText` modeli değiştirir; görünüm bu değişikliği fark eder ve kendini yeniden çizdirir. Para üstü uygulaması bu ayrımı doğrudan kullanır.

9.9.2 Büyük harf ve yerel ayar

`toUpperCase()` ve `toLowerCase()` parametresiz çağrılırsa bilgisayarın yerel ayarını (*locale*) kullanır. Türkçe, harf dönüşümünde özel kuralları olan dillerden biridir: noktalı ve noktasız iki ayrı "i" vardır. İngilizce ayarlı bir makinede `"istanbul".toUpperCase()` sonucu `"ISTANBUL"`, Türkçe ayarlıda `"İSTANBUL"` 'dur. Aynı program, çalıştığı bilgisayara göre farklı sonuç verir.

Sonucu sabitlemek için yerel ayar açıkça verilir: `toUpperCase(Locale.forLanguageTag("tr"))`. Aşağıdaki deneme iki dilin kurallarını karşılaştırır.

```
String s = "Kış İzmir";
Locale tr = Locale.forLanguageTag("tr");
System.out.println(s.toUpperCase(tr));
System.out.println(s.toLowerCase(tr));
System.out.println(s.toUpperCase(Locale.ENGLISH));
System.out.println(s);
```

```
KIŞ İZMİR
kış izmir
KIŞ İZMIR
Kış İzmir
```

Türkçe kurallarında `i → İ`, `ı → I`, `İ → i` ve `I → ı` 'dır. İngilizce kurallarında hem `ı` hem `i` büyüyünce `I` olur; `İ` ise olduğu gibi kalır. Bu yüzden üçüncü satır `KIŞ İZMIR` 'dir. Son satır, dizgilerin değiştirilemezliğini bir kez daha gösterir: üç dönüşümden sonra `s` hâlâ `"Kış İzmir"` 'dir.

Yerel ayar nesnesi eskiden `new Locale("tr")` ile oluşturulurdu. Bu yapıcı Java 19'dan beri kullanımdan kaldırılmıştır (*deprecated*); derleyici uyarı verir. Yerine `Locale.forLanguageTag("tr")` ya da `Locale.of("tr")` kullanılır.

Sık yapılan hata

Programın mantığını yerel ayara bağlı bir dönüşüme dayandırmak. Örneğin bir komut `komut.toUpperCase().equals("QUIT")` ile denetlenirse, Türkçe ayarlı makinede `"quit"` `"QUİT"` olur ve karşılaştırma başarısız olur. Kullanıcıya gösterilmeyen, program içi metinlerde `Locale.ROOT` ya da `Locale.ENGLISH` verilir; kullanıcıya gösterilen Türkçe metinlerde

Türkçe yerel ayar verilir.

9.10 Uygulama: para üstü hesaplayıcı

Bu uygulama, girilen kuruş miktarının en az sayıda madeni parayla (50, 25, 10, 5 ve 1 kuruş) nasıl ödeneceğini gösterir. Pencerede düğme yoktur: rapor her tuş vuruşunda kendiliğinden güncellenir. Bunu metin alanının belgesine eklenen bir `DocumentListener` sağlar.

9.10.1 Alanlar ve yapıcı

```
public class ParaUstuHesaplama extends JFrame {
    private JTextField miktarAlani = new JTextField(12);
    private Document metin = miktarAlani.getDocument();
    private JTextArea rapor = new JTextArea(8, 40);

    public ParaUstuHesaplama() {
        var ustPanel = new JPanel();
        ustPanel.add(new JLabel("Miktar:"));
        ustPanel.add(miktarAlani);
        getContentPane().add(ustPanel, BorderLayout.NORTH);
        getContentPane().add(new JScrollPane(rapor),
                BorderLayout.CENTER);
        setBackground(Color.LIGHT_GRAY);
        rapor.setEditable(false);
        metin.addDocumentListener(new DocumentListener() {
            public void insertUpdate(DocumentEvent e) { raporuGuncelle(); }
            public void removeUpdate(DocumentEvent e) { raporuGuncelle(); }
            public void changedUpdate(DocumentEvent e) { raporuGuncelle(); }
        });
    }
```

Sınıf bu kez `JPanel`'den değil, `JFrame`'den türer. Pencerenin kendisi uygulamadır. Üç alan vardır: 12 sütunluk bir miktar alanı, bu alanın belgesi ve 8 satır × 40 sütunluk bir rapor alanı.

Swing'de veri ile görünüm ayrıdır. Bu düzene Model-Görünüm-Denetleyici (*Model-View-Controller*, MVC) denir. `JTextField` görünümdür: metni ekranda gösterir ve kullanıcının yazdıklarını alır. `getDocument()` ile alınan `Document` nesnesi ise metni tutan modeldir. Kullanıcı bir tuşa bastığında model değişir ve modeli dinleyen herkese haber verilir.

Yapıcı üst paneli kurar: bir “Miktar:” etiketi ve miktar alanı. Panel pencerenin kuzeyine, rapor alanı bir kaydırma bölmesiyle merkeze eklenir. Rapor salt okunur yapılır. Son olarak belgeye anonim bir `DocumentListener` eklenir.

`DocumentListener`'ın üç metodu vardır: `insertUpdate` (metin eklendi), `removeUpdate` (metin silindi) ve `changedUpdate` (stil değişti). Düz metin alanında stil olmadığı için `changedUpdate` hiç çağrılmaz. Ama arayüzün üç metodu da soyuttur ve üçü de yazılmalıdır. Üçü de aynı işi yapar: `raporuGuncelle()`'yi çağırır.

9.10.2 Kod izleme: `raporuGuncelle`

```
void raporuGuncelle() {
    try {
        var miktar = Integer.parseInt(
                metin.getText(0, metin.getLength()));
        int[] dizi = {50, 25, 10, 5, 1};
        rapor.setText(miktar
                + " kuruş yapmak için kullanılacaklar:\n");
        for (int x : dizi) {
            rapor.append(miktar / x + " " + x + " kuruş\n");
            miktar %= x;
        }
    } catch (NumberFormatException e) {
        rapor.setText("Tam sayı değil veya aralık dışında");
    } catch (Exception e) {
        rapor.setText(e.toString());
    }
}
```

Metot belgenin metnini okur ve tam sayıya çevirir. `metin.getText(0, metin.getLength())` belgenin 0. karakterinden başlayıp bütün uzunluğu kadar metni döndürür. Ardından para değerleri büyükten küçüğe bir diziye konur. Rapor `setText` ile sıfırdan başlatılır. Döngü her para değeri için iki iş yapar: kaç adet kullanılacağını (`miktar / x`) yazar ve kalan miktarı (`miktar %= x`) hesaplar.

Adım	Satır	Değişkenler	Yeni çıktı
1	3: `var miktar = Integer.parseInt(` 4: `metin.getText(0, metin.getLength()));`	miktar=93	—
2	5: `int[] dizi = {50, 25, 10, 5, 1};`	miktar=93, dizi=[50, 25, 10, 5, 1]	—
3	6: `rapor.setText(miktar` 7: `+ " kuruş yapmak için kullanılacaklar:\n");`	miktar=93, dizi=[50, 25, 10, 5, 1]	`93 kuruş yapmak için kullanılacaklar:`
4	9: `rapor.append(miktar / x + " " + x + " kuruş\n");`	miktar=93, x=50	`1 50 kuruş`
5	10: `miktar %= x;`	miktar=43, x=50	—
6	9: `rapor.append(miktar / x + " " + x + " kuruş\n");`	miktar=43, x=25	`1 25 kuruş`
7	10: `miktar %= x;`	miktar=18, x=25	—

Adım	Satır	Değişkenler	Yeni çıktı
8	9: `rapor.append(miktar / x + " " + x + " kuruş\n");`	miktar=18, x=10	`1 10 kuruş`
9	10: `miktar %= x;`	miktar=8, x=10	—
10	9: `rapor.append(miktar / x + " " + x + " kuruş\n");`	miktar=8, x=5	`1 5 kuruş`
11	10: `miktar %= x;`	miktar=3, x=5	—
12	9: `rapor.append(miktar / x + " " + x + " kuruş\n");`	miktar=3, x=1	`3 1 kuruş`
13	10: `miktar %= x;`	miktar=0, x=1	—

Tablo 9.18: Miktar 93 iken `raporuGuncelle` izlemesi.

Yöntem, açgözlü (*greedy*) bir algoritmadır: her adımda kullanılabilecek en büyük para seçilir. 93 kuruşta önce bir 50'lik kullanılır, 43 kalır. Sonra bir 25'lik, 18 kalır. Bir 10'luk, 8 kalır. Bir 5'lik, 3 kalır. Son olarak üç 1'lik. Toplam yedi madeni para. Tam sayı bölmesi adet hesabını, kalan işleci de kalan miktarı verir.



Şekil 9.27: 13. adım. Döngü bitti: 1 × 50, 1 × 25, 1 × 10, 1 × 5 ve 3 × 1 kuruş.

50, 25, 10, 5, 1 dizisi için açgözlü yöntem her zaman en az sayıda parayı verir. Bu her para sistemi için doğru değildir. Örneğin paralar 1, 3 ve 4 olsaydı, 6 için açgözlü yöntem 4 + 1 + 1 (üç para) seçerdi. Oysa 3 + 3 iki parayla aynı tutarı verir.

Örnek 9.14

Kullanıcı miktar alanına 187 yazdı. Rapor hangi satırları içerir?

187 / 50 = 3, kalan 37. 37 / 25 = 1, kalan 12. 12 / 10 = 1, kalan 2. 2 / 5 = 0, kalan 2. 2 / 1 = 2, kalan 0. Toplam 3 + 1 + 1 + 0 + 2 = 7 madeni para. Sıfır adetli satır da yazılır, çünkü döngü her para değeri için bir satır ekler:

```
187 kuruş yapmak için kullanılacaklar:
3 50 kuruş
1 25 kuruş
1 10 kuruş
0 5 kuruş
2 1 kuruş
```

9.10.3 `main` ve olayların sırası

```
public static void main(String[] args) {
    SwingUtilities.invokeLater(() -> {
        var frame = new ParaUstuHesaplama();
        frame.setTitle("Para Üstü Hesaplama");
        frame.pack();
        frame.setDefaultCloseOperation(
                JFrame.EXIT_ON_CLOSE);
        frame.setVisible(true);
    });
}
```

`main` pencereyi EDT'de oluşturur, başlığını verir, `pack()` ile boyutunu hesaplar ve gösterir. Programın asıl ilginç yanı, kullanıcının yazdıklarına göre hangi olayların geldiğidir. `JFrame` başsız kipte oluşturulamadığı için aynı alanlar, dinleyici ve `raporuGuncelle` metodu pencere olmadan bir sınıfta denendi; belgeye ekleme ve silme doğrudan yapıldı. Gelen olaylar şunlardır:

```
9 yazar              insert
3 yazar              insert
hepsini siler        remove
abc                  insert
99999999999          remove insert
setText 287          remove insert
setText boş          remove
boştan setText 287   insert
-7                   remove insert
68                   remove insert
187                  remove insert
boşluklu             remove insert
```

Kullanıcı / kod	Olay	Rapor
9 yazar	`insertUpdate`	9 kuruş: 0, 0, 0, 1 × 5, 4 × 1
3 yazar (alan 93)	`insertUpdate`	93 kuruş: 1, 1, 1, 1, 3

Kullanıcı / kod	Olay	Rapor
Hepsini siler	`removeUpdate`	`parseInt("")` : "Tam sayı değil veya aralık dışında"
`abc` ya da `99999999999`	`NumberFormatException`	Aynı ileti
`setText("287")` (alan doluyken)	Önce `removeUpdate`, sonra `insertUpdate`	5, 1, 1, 0, 2
`setText("287")` (alan boşken)	Yalnız `insertUpdate`	5, 1, 1, 0, 2
`-7`	Geçerli sayı	0, 0, 0, −1, −2: denetim eksik

Tablo 9.19: Kullanıcı eylemleri, olaylar ve rapor (çalıştırılarak doğrulandı).

Tablodaki her satır bir dersi taşır. Kullanıcı "93" yazarken iki olay gelir: "9" ve "3" için birer `insertUpdate`. Rapor her tuşta güncellenir; ara durum (9 kuruş) da bir an için görünür. Alan tamamen silinince metin boş dizgidir ve `parseInt("")` `NumberFormatException` fırlatır. İlk `catch` bunu yakalar.

`99999999999` bir sayıdır, ama `int`'in en büyük değeri olan 2.147.483.647'yi aşar. `parseInt` bu durumda da `NumberFormatException` fırlatır; ileti bu yüzden "tam sayı değil *veya aralık dışında*" der. `setText` ise bir bütün olarak değil, iki adımda çalışır: önce eski metni siler, sonra yenisini ekler. Alan doluysa iki olay gelir. Birinci olayda alan bir an boştur ve rapor hata iletisini gösterir; ikinci olay doğru raporu yazar.

Negatif sayı ise bir eksikliği ortaya çıkarır. `-7` geçerli bir tam sayıdır. Java'da tam sayı bölmesi sıfıra doğru keser ve kalanın işareti bölünenin işaretini alır: `-7 / 5 = -1`, `-7 % 5 = -2`. Rapor `-1 5 kuruş` ve `-2 1 kuruş` gibi anlamsız satırlar içerir:

```
-7 kuruş yapmak için kullanılacaklar:
0 50 kuruş
0 25 kuruş
0 10 kuruş
-1 5 kuruş
-2 1 kuruş
```

Program negatif değeri reddetmelidir. `parseInt`'ten hemen sonra bir denetim yeterlidir: `if (miktar < 0) { rapor.setText("Miktar pozitif olmalı"); return; }`.

İki `catch` bloğunun görevleri farklıdır. İlki beklenen hatayı, yani sayı olmayan metni ve `int` sınırını aşan değeri yakalar. İkincisi diğer her şey için genel bir güvencedir. Bu blok gereklidir, çünkü `getText(0, n)` kontrol edilen bir istisna olan `BadLocationException`'ı fırlatabileceğini bildirir. Derleyici bu istisnanın ele alınmasını ister; `catch (Exception e)` onu da kapsar.

Örnek 9.15

Para üstü programına 68 yazılınca rapor ne olur?

68 / 50 = 1, kalan 18. 18 / 25 = 0, kalan 18. 18 / 10 = 1, kalan 8. 8 / 5 = 1, kalan 3. 3 / 1 = 3. Rapor "68 kuruş yapmak için kullanılacaklar:" satırıyla başlar ve beş satırla devam eder:

```
68 kuruş yapmak için kullanılacaklar:
1 50 kuruş
0 25 kuruş
1 10 kuruş
1 5 kuruş
3 1 kuruş
```

9.11 İyi uygulamalar

GUI programlarının kalitesini birkaç alışkanlık belirler.

Yerleşim yöneticisi kullanmak
: Koordinatlar elle verilmez. Yerleşim yöneticisiyle düzen farklı ekran boyutu, çözünürlük ve yazı tipinde bozulmaz. Karmaşık düzenler iç içe panellerle kurulur.

Anlamlı adlar seçmek
: `b1`, `t2` yerine `buyukHarfButton`, `miktarAlani` gibi adlar kullanılır. Onlarca bileşenli bir arayüzde kod ancak böyle okunur kalır.

Olay güdümlü düşünmek
: Tasarım "Kullanıcı şunu yapınca ne olmalı?" sorusuyla yapılır. Her eylem için bir dinleyici yazılır; dinleyiciler birbirinin sırasına güvenmez.

Arayüz ile iş mantığını ayırmak
: Para üstü hesabı gibi kurallar ayrı bir metotta ya da sınıfta tutulur. Böylece arayüz olmadan test edilebilir ve başka bir arayüzde de kullanılabilir. Notta verilen çıktıların çoğu bu ayrım sayesinde pencere açılmadan elde edildi.

Arayüzü EDT'de kurmak
: `main`'in içi tek bir çağrıdır: `SwingUtilities.invokeLater(() -> { … })`.

EDT'yi meşgul etmemek
: Dinleyicide uzun bir iş (dosya indirme, büyük bir hesap) yapılırsa EDT o süre boyunca başka olay işleyemez ve pencere donar. Bu işler ayrı bir iş parçacığına taşınır; sonuç `invokeLater` ile EDT'ye geri verilir.

Örnek 9.16

Para üstü hesabını arayüzden ayırmak için nasıl bir metot yazılabilir?

Hesap, girdi olarak bir miktar alıp adetleri döndüren statik bir metoda taşınır:

```
static int[] paraUstu(int miktar) {
    int[] dizi = {50, 25, 10, 5, 1};
    int[] adet = new int[dizi.length];
    for (int i = 0; i < dizi.length; i++) {
        adet[i] = miktar / dizi[i];
        miktar %= dizi[i];
    }
    return adet;
}
```

`raporuGuncelle` artık yalnız metni okur, bu metodu çağırır ve sonucu rapora yazar. Metot ise bir konsol programında ya da bir testte doğrudan denenebilir: `paraUstu(93)` `[1, 1, 1, 1, 3]` döndürür.

9.12 Çözümlü problemler

Örnek 9.17 (Problem 1)

`BorderLayout` kullanan bir panele önce `A`, sonra `B` düğmesi aynı `NORTH` bölgesine eklenirse kuzeyde hangisi görünür?

`B`. Her bölgede tek bileşen olur; son eklenen kazanır. `A` panelde kalır (bileşen sayısı 2'dir), ama yerleşime katılmaz. Başsız kipte 200 × 100'lük panelde `B` `(0, 0, 200, 26)`, `A` ise 0 × 0 çıktı.

Örnek 9.18 (Problem 2)

Kullanıcı panele bir kez tıklayıp bırakırsa `MouseListener`'ın hangi metotları hangi sırayla çağrılır?

`mousePressed` → `mouseReleased` → `mouseClicked`. İşaretçi panele dışarıdan girerken ayrıca `mouseEntered` gelir. Basma ile bırakma arasında fare kıpırdarsa `mouseDragged` olayları araya girer ve `mouseClicked` gelmeyebilir.

Örnek 9.19 (Problem 3)

`Color.RED.brighter()` hangi rengi verir?

Yine `(255, 0, 0)`. Kırmızı bileşen zaten en büyük değerdedir; 255 / 0,7 = 364 sınırda 255'e iner. Yeşil ve mavi bileşenler 0'dır ve 0 / 0,7 = 0 kalır.

Örnek 9.20 (Problem 4: hedef tahtası)

Panel boyutuna göre ölçeklenen, kırmızı-beyaz beş iç içe halkadan oluşan bir hedef tahtası çizilecektir. `paintComponent` nasıl yazılır? 300 × 200'lük panelde hangi çağrılar yapılır?

Merkez ve en büyük yarıçap işaretteki gibi hesaplanır. Döngü her turda yarıçapı beşte bir azaltır ve renkleri sırayla değiştirir:

```
int cx = getWidth() / 2, cy = getHeight() / 2;
int r = Math.min(cx, cy) - 5;
for (int i = 0; i < 5; i++) {
    int ri = r - i * r / 5;
    g.setColor(i % 2 == 0 ? Color.RED : Color.WHITE);
    g.fillOval(cx - ri, cy - ri, 2 * ri, 2 * ri);
}
```

300 × 200'de `cx = 150`, `cy = 100`, `r = 95` 'tir. `i * r / 5` soldan sağa hesaplanır: önce çarpma, sonra tam sayı bölmesi. Yarıçaplar 95, 76, 57, 38 ve 19 olur; çağrılar `fillOval(55, 5, 190, 190)`, `(74, 24, 152, 152)`, `(93, 43, 114, 114)`, `(112, 62, 76, 76)` ve `(131, 81, 38, 38)` 'dir. Daireler büyükten küçüğe çizilir. Küçükten büyüğe çizilseydi en son çizilen büyük daire hepsini örterdi.

Örnek 9.21 (Problem 5: sağ tıkla geri alma)

Çizim programında sağ tık son eğriyi silsin; sağ düğmeyle sürükleme de çizim yapmasın. Hangi değişiklikler gerekir?

`mousePressed` 'ın başına bir denetim eklenir. Sağ düğmeyse son eğri silinir ve metot döner:

```
if (e.getButton() == MouseEvent.BUTTON3) {
    if (!egriler.isEmpty()) egriler.remove(egriler.size() - 1);
    repaint();
    return;
}
```

`mouseDragged` 'ın başına da `if (!SwingUtilities.isLeftMouseButton(e)) return;` yazılır. Bu satır olmasaydı sağ düğmeyle sürüklemek, az önce silinen eğrinin yerine bir önceki eğriye nokta eklerdi. Liste boşken sağ sürükleme ise `get(-1)` ile `IndexOutOfBoundsException: Index -1 out of bounds for length 0` fırlatırdı. Değişiklikler başsız kipte olay gönderilerek denendi: iki eğri çizildikten sonra sağ tık eğri sayısını 1'e indirdi ve sağ sürükleme hiçbir noktayı değiştirmedi.

Örnek 9.22 (Problem 6: sıcaklık dönüştürücü)

Bir `JTextField`'a °C girilip düğmeye basılınca °F gösterilsin. Dinleyici nasıl yazılır?

```
cevir.addActionListener(e -> {
    try {
        double c = Double.parseDouble(alan.getText());
        sonuc.setText(c * 9 / 5 + 32 + " °F");
    } catch (NumberFormatException ex) {
        sonuc.setText("Sayı değil");
    }
});
```

100 girildiğinde `212.0 °F`, −40 girildiğinde `-40.0 °F` yazılır; iki ölçeğin eşit olduğu tek sıcaklık −40'tır. `abc` girildiğinde `parseDouble` `NumberFormatException` fırlatır ve "Sayı değil" görünür. `c * 9 / 5 + 32 + " °F"` ifadesinde sıra önemlidir: önce sayısal işlemler soldan sağa yapılır, en son dizgi eklenir. Lambdanın içindeki istisna parametresine `e` yerine `ex` adı verilir, çünkü `e` zaten lambdanın parametresidir.

Daha küçük iki alıştırma da aynı kalıpları kullanır. Sayaç penceresinde "−1", "+1" ve "Sıfırla" düğmeleri bir `private static int sayac` alanını değiştirir ve etiketi günceller. İki "+1" ve bir "−1" tıklamasından sonra etiket `1`, "Sıfırla"dan sonra `0` gösterir. Çizim programına "Temizle" düğmesi eklemek için düğme dinleyicisi `egriler.clear(); repaint();` çalıştırır. Panel ve düğme pencerenin `CENTER` ve `SOUTH` bölgelerine konur.

9.13 Sık yapılan hatalar

Grafik programlarında hataların bir kısmı derleyiciden gelir, bir kısmı ise programın sessizce yanlış davranmasıyla ortaya çıkar. Derleyici iletileri JDK 21 ile alınmıştır.

Hata	Belirti	Çözüm
İçe aktarmayı unutmak	`cannot find symbol … class JFrame`	`import javax.swing.*;`
`setVisible(true)` yok	Pencere hiç görünmez	Kurulumun sonunda `setVisible(true)`
`EXIT_ON_CLOSE` yok	Pencere kapanır, program arka planda sürer	`setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE)`
Metot adını yanlış yazmak (`paintcomponent`)	Hiç çağrılmaz; `@Override` varsa `method does not override or implement a method from a supertype`	Doğru ad ve her zaman `@Override`
`paintComponent`'i `public`/`protected` olmadan yazmak	`attempting to assign weaker access privileges; was protected`	`protected` ya da `public`

Hata	Belirti	Çözüm
`super.paintComponent(g)` unutmak	Zemin temizlenmez, eski çizim artıkları kalır	Gövdenin ilk satırı
Arayüzün bütün metotlarını yazmamak	`… is not abstract and does not override abstract method mouseExited(MouseEvent) in MouseListener`	`MouseAdapter`'dan türetmek
Hareket dinleyicisini kaydetmemek	`mouseDragged` hiç çağrılmaz	`addMouseMotionListener(…)`
Son öğe için `get(size())`	`IndexOutOfBoundsException: Index 1 out of bounds for length 1`	`get(size() - 1)`
Lambda içinde yerel değişken değiştirmek	`local variables referenced from a lambda expression must be final or effectively final`	Değişkeni alan yapmak
Çok metotlu arayüze lambda vermek	`MouseListener is not a functional interface`	Adaptör ya da anonim sınıf
Sabiti sınıf adı olmadan yazmak	`cannot find symbol … variable CENTER`	`BorderLayout.CENTER`
`java.awt.*` ve `java.util.*` ile `List`	`reference to List is ambiguous`	`import java.util.List;`
EDT içinden `invokeAndWait`	`java.lang.Error: Cannot call invokeAndWait from the event dispatcher thread`	EDT'de doğrudan çağırmak ya da `invokeLater`
Dinleyicide uzun iş	Pencere donar, tıklamalara yanıt vermez	İşi ayrı iş parçacığına taşımak

Tablo 9.20: Sık yapılan hatalar, belirtileri ve çözümleri.

İçe aktarma unutulduğunda derleyici aynı satır için iki hata verir: biri değişkenin tipi, biri `new` ifadesi içindir.

```
Ana.java:3: error: cannot find symbol
    JFrame f = new JFrame("x");
  symbol:   class JFrame
  location: class Ana
Ana.java:3: error: cannot find symbol
    JFrame f = new JFrame("x");
  symbol:   class JFrame
  location: class Ana
```

`@Override` etiketinin değeri en açık biçimde yanlış yazılmış metot adında görülür. Etiket olmadan `paintcomponent` (küçük c) yazan sınıf sorunsuz derlenir. Yeni bir metot tanımlanmıştır ve Swing onu hiç çağırmaz; panel boş kalır. Etiket eklenince derleyici hatayı hemen bulur:

```
Ana.java:7: error: method does not override or implement a method from a supertype
  @Override
```

`paintComponent`, `JComponent`'te `protected` olarak tanımlıdır. Geçersiz kılan metodun erişimi daraltılamaz: `protected` ya da `public` olabilir. Erişim belirteci hiç yazılmazsa metot paket erişimli olur ve derleyici reddeder:

```
Ana.java:7: error: paintComponent(Graphics) in Ana cannot override
paintComponent(Graphics) in JComponent
  void paintComponent(Graphics g) {
  attempting to assign weaker access privileges; was protected
```

Sabitin sınıf adı olmadan yazılması da benzer bir ileti verir:

```
Ana.java:9: error: cannot find symbol
  p.add(new JButton("x"), CENTER);
  symbol:   variable CENTER
  location: class Ana
```

Örnek 9.23

Bir çizim paneli derleniyor, ama eğriler görünmüyor, eski çizim artıkları kalıyor ve ilk sürüklemede istisna fırlıyor. Kodda şu satırlar var: `egriler.get(egriler.size()).add(e.getPoint())` ve `public void paintcomponent(Graphics g) { for (…) … }`. Üç hata nedir?

Birincisi metodun adıdır: `paintcomponent` yeni bir metottur ve Swing onu çağırmaz. `@Override` eklenseydi derleme hatası alınırdı. İkincisi, metodun içinde `super.paintComponent(g)` yoktur; ad düzeltilince zemin temizlenmez ve artıklar kalır. Üçüncüsü son öğenin indisidir: `get(size())` `IndexOutOfBoundsException` fırlatır, doğrusu `size() - 1`'dir. Kodun derlenmesi hatasız olduğunu göstermez; üç hatanın hiçbiri derleyiciye takılmaz.

9.14 Özet

- GUI programı olay güdümlüdür: `main` yalnız arayüzü kurar; akışı kullanıcının eylemleri belirler. Kodun büyük kısmı dinleyicilerden oluşur.
- `java.awt` paketi `Graphics`, `Color`, `Font`, `Point`, yerleşim yöneticileri ve olay sınıflarını; `javax.swing` paketi adı `J` ile başlayan bileşenleri içerir. Swing bileşenleri Java'nın kendisi çizer.
- Ekran koordinatlarında `(0, 0)` bileşenin sol üst köşesidir; `x` sağa, `y` aşağı doğru büyür.
- Kalıcı çizim için `JPanel`'den türetilir ve `paintComponent(Graphics g)` geçersiz kılınır. Gövdenin ilk satırı `super.paintComponent(g)`'dir. Metodu Swing çağırır.

- `draw…` metotları çerçeve, `fill…` metotları içi dolu şekil çizer. Elips ve daireler sınırlayıcı dikdörtgenle verilir: merkezi `(cx, cy)`, yarıçapı `r` olan daire `fillOval(cx – r, cy – r, 2r, 2r)`'dir. `drawString`'in `y`'si taban çizgisidir.
- `Graphics` nesnesi rengini ve yazı tipini hatırlar; `setColor` ve `setFont` sonraki bütün çizimleri etkiler. Sonradan çizilen şekil öncekini örter.
- `JFrame` üst düzey penceredir. Boyut `setSize` ya da `pack()` ile verilir, kapatma davranışı `setDefaultCloseOperation(EXIT_ON_CLOSE)` ile belirlenir, en son `setVisible(true)` çağrılır. Varsayılan kapatma davranışı `HIDE_ON_CLOSE`'dur.
- `JPanel` hem kapsayıcı hem tuvaldir; varsayılan yerleşim yöneticisi `FlowLayout`'tur. `JFrame`'in içerik bölmesi `BorderLayout` kullanır.
- `Point`'in `x` ve `y` alanları `public int`'tir; `getX()` ve `getY()` `double` döndürür. `translate` göreli, `setLocation` mutlak taşır.
- `Color` RGB üçlüsüyle (0–255) oluşturulur; değiştirilemezdir. `darker()` bileşenleri 0,7 ile çarpar, `brighter()` 0,7'ye böler; ikisi de sonucu tam sayıya keser ve birbirinin tam tersi değildir.
- `Font(ad, stil, punto)`; stil sabitleri bit bayraklarıdır: `BOLD | ITALIC = 3`. `deriveFont(float)` boyutu, `deriveFont(int)` stili değiştirir.
- Yerleşim yöneticisi bileşenlerin konum ve boyutunu kurala göre hesaplar. `add` yalnız listeye ekler; yerleşim düzen hesaplanınca yapılır.
- `FlowLayout` satır satır, ortalı ve 5 piksel boşlukla dizer. `BorderLayout` beş bölgeye yerleştirir: her bölgede tek bileşen, son eklenen kazanır, merkez kalanı alır. `GridLayout` eşit hücrelere böler. `BoxLayout` tek satır ya da tek sütun kurar.
- Dinleyici arayüzü bir olayın metotlarını tanımlar; dinleyici `add…Listener` ile kaydedilir. `MouseListener` basma, bırakma, tıklama, giriş ve çıkışı; `MouseMotionListener` hareket ve sürüklemeyi bildirir. Kayıt arayüze göre yapılır.
- Adaptör sınıfı (`MouseAdapter`) bütün metotları boş gövdeyle uygular; alt sınıf yalnız gerekeni yazar. Anonim iç sınıf tek kullanımlık dinleyici için uygundur.
- Bir tıklama `pressed → released → clicked` üretir. `MouseEvent` konumu bileşene göre verir; `getButton()` düğmeyi (`BUTTON3` sağ) söyler.
- Swing tek iş parçacıklıdır: bileşenler yalnız EDT'de kurulur ve değiştirilir. `invokeLater` işi kuyruğa koyar ve döner; `invokeAndWait` iş bitene kadar bekler ve EDT içinden çağrılırsa `Error` fırlatır. Kuyruk sıralıdır.
- Dinleyici ekrana çizmez; veriyi günceller ve `repaint()` ister. `repaint()` bir istektir; Swing uygun anda `paintComponent`'i çağırır.
- `ActionListener` tek metotlu, fonksiyonel bir arayüzdür ve lambda ile yazılabilir. Lambda içinden yerel değişken değiştirilemez; değişen değerler alan olur.
- `JLabel` metin gösterir, `JTextField` tek satır, `JTextArea` çok satır metin alır, `JScrollPane` kaydırır. `getLineCount()` satır sonu sayısının bir fazlasıdır.
- `toUpperCase()` yerel ayara bağlıdır; Türkçede `i → İ`. Sonucu sabitlemek için yerel ayar açıkça verilir.

- Metin bileşenlerinde model (`Document`) ile görünüm ayrıdır. `DocumentListener` her ekleme ve silmede çağrılır; `setText` dolu alanda iki olay üretir.
- İyi uygulamalar: yerleşim yöneticisi kullanmak, anlamlı adlar seçmek, iş mantığını arayüzden ayırmak, arayüzü EDT'de kurmak ve EDT'yi uzun işlerle meşgul etmemek.

9.15 Kavram listesi

grafik kullanıcı arayüzü (GUI) *graphical user interface* · bileşen *component* · kapsayıcı *container* · yerleşim yöneticisi *layout manager* · olay *event* · olay dinleyici *event listener* · olay iş parçacığı (EDT) *Event Dispatch Thread* · iş parçacığı *thread* · arka plan iş parçacığı *daemon thread* · iş parçacığı güvenliği *thread safety* · anonim sınıf *anonymous class* · iç sınıf *inner class* · lambda ifadesi *lambda expression* · fonksiyonel arayüz *functional interface* · geçersiz kılma *method overriding* · uygulamak (arayüzü) *implement (an interface)* · değiştirilemez *immutable* · yerel ayar *locale*

olay güdümlü programlama *event-driven programming* · ekran koordinatları *screen coordinates* · sınırlayıcı dikdörtgen *bounding rectangle* · tuval *canvas* · içerik bölmesi *content pane* · adaptör sınıfı *adapter class* · RGB rengi *RGB color* · kırpma *clipping* · model-görünüm-denetleyici *Model-View-Controller (MVC)* · gözlemci deseni *Observer pattern* · başsız kip *headless mode*

9.16 Kendini Dene

1. `g.drawRect(10, 20, 30, 40)` ve `g.fillRect(10, 20, 30, 40)` çağrılarının boyadığı en sağ alttaki piksel hangisidir?

   Cevap

   `drawRect` için `(40, 60)`, `fillRect` için `(39, 59)`. Çerçeve, sınırlayıcı dikdörtgenin sağ ve alt kenarını da çizer ve `en + 1` piksel geniş olur. Dolu dikdörtgen tam `en × boy` piksel boyar. İki değer de bir resim üzerinde boyalı pikseller sayılarak bulundu.

2. `new Color(10, 20, 30).brighter()` hangi bileşenleri verir?

   Cevap

   `(14, 28, 42)`. Her bileşen 0,7'ye bölünüp kesilir: 10 / 0,7 = 14,28 → 14; 20 / 0,7 = 28,57 → 28; 30 / 0,7 = 42,85 → 42. Hiçbir bileşen 255'e ulaşmadığı için sınır devreye girmez.

3. 300 × 200 piksellik bir panel `new BorderLayout(5, 5)` kullanıyor. Kuzeye 30, güneye 20 piksel yüksekliği; batıya 60, doğuya 40 piksel genişliği tercih eden bileşenler ve merkeze bir bileşen ekleniyor. Merkez bileşeninin sınırları nedir?

   Cevap

   `(65, 35, 190, 140)`. Merkez, batının sağından 5 piksel boşlukla başlar: `x = 60 + 5`. Kuzeyin altından 5 piksel boşlukla başlar: `y = 30 + 5`. Genişlik `300 − 60 − 40 − 2 × 5 = 190`, yükseklik `200 − 30 − 20 − 2 × 5 = 140`'tır. Batı ve doğu da aynı yükseklikte ve `y = 35`'tedir.

4. Giriş yasak işaretinin `paintComponent` metodu 300 × 300 piksellik bir panelde çalışıyor. `r`, `r2`, `bw` ve `bh` kaçtır?

   Cevap

   `r = min(150, 150) − 5 = 145`. `r2 = (int) 130,5 = 130`. `bw = (int) (130 × 1,4) = 182`. `bh = (int) (130 × 0,35) = (int) 45,5 = 45`. Kesme yarımı da atar; yuvarlama olsaydı 46 bulunurdu.

5. Para üstü programına 41 yazılınca rapordaki adetler nelerdir?

   Cevap

   0 × 50, 1 × 25, 1 × 10, 1 × 5, 1 × 1. 41 / 50 = 0 (kalan 41), 41 / 25 = 1 (kalan 16), 16 / 10 = 1 (kalan 6), 6 / 5 = 1 (kalan 1), 1 / 1 = 1. Toplam dört madeni para.

6. Aşağıdaki kod derlenir mi?

```
String ad = " Ayşe ";
ad = ad.trim();
JButton b = new JButton("x");
b.addActionListener(e -> System.out.println(ad));
```

   Cevap

   Derlenmez: `local variables referenced from a lambda expression must be final or effectively final`. Lambda `ad`'ı yalnız okuyor, ama `ad` ikinci satırda yeniden atandığı için fiilen final değildir. Çözüm, kırpılmış değeri yeni bir değişkene almaktır: `String temiz = ad.trim();` ve lambda içinde `temiz`.

7. Yalnız `mouseClicked`'ı geçersiz kılan bir `MouseAdapter` nesnesi panele

`addMouseMotionListener` ile kaydediliyor. Panele tıklanınca metot çalışır mı?

Cevap

Çalışmaz. `MouseAdapter` üç arayüzü birden uyguladığı için kayıt derlenir, ama panel bu nesneyi yalnız hareket dinleyicisi olarak tanır. Tıklama olayları yalnız `addMouseListener` ile kaydedilen dinleyicilere gider. Panele bir tıklama olayı gönderildiğinde metot 0 kez çağrıldı.

8. Metni `"a\n\nb\n"` olan bir `JTextArea`'da `getLineCount()` kaç döndürür?

Cevap

4. Metinde üç satır sonu vardır: “a”, boş satır, “b” ve son satır sonundan sonraki boş satır. Satır sayısı, satır sonu sayısının bir fazlasıdır.

9.17 İleri okuma

- Oracle, *The Java Tutorials*, “Creating a GUI With Swing”: bileşenlerin, yerleşim yöneticilerinin ve olay dinleyicilerinin resmî ve ücretsiz anlatımıdır. “Concurrency in Swing” dersi olay iş parçacığını ayrıntılı açıklar.
- Oracle, *The Java Tutorials*, “Performing Custom Painting”: `paintComponent`, `repaint` ve kırpmanın nasıl birlikte çalıştığını adım adım bir örnekle gösterir.
- C. S. Horstmann, *Core Java, Volume I: Fundamentals* (12. baskı), Bölüm 10–11: grafik programlama, olay işleme ve Swing bileşenlerini MVC bakışıyla işler.
- E. Gamma, R. Helm, R. Johnson ve J. Vlissides, *Design Patterns*: Gözlemci deseninin kaynak anlatımıdır; Swing dinleyicilerinin neden böyle tasarlandığını açıklar.

9.18 Kaynaklar

1. Oracle, *The Java Tutorials*: Creating a GUI With Swing; Performing Custom Painting; Laying Out Components Within a Container; Writing Event Listeners; Concurrency in Swing.
2. Oracle, *Java SE 21 API Documentation*: `java.awt.Graphics`, `java.awt.Color`, `javax.swing.JFrame`, `javax.swing.SwingUtilities`.
3. P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (çevrim içi bölümler: Swing GUI Components; Graphics and Java 2D).
4. K. Sierra, B. Bates ve T. Gee, *Head First Java*, 3. baskı, O'Reilly (Bölüm 14–15: GUI, Swing ve olaylar).
5. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson (Bölüm 10–11: grafik programlama, olay işleme, Swing).
6. A. B. Downey ve C. Mayfield, *Think Java*, 2. baskı, O'Reilly (Ek C: Graphics).

7. E. Gamma, R. Helm, R. Johnson ve J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

KISIM IV · BÖLÜM 10

İş Parçacıkları

Bir program aynı anda birden çok iş yürütebilir. Kelime işlemci yazarken yazım denetimi yapar, tarayıcı bir sayfayı indirirken öbür sekmeye tepki verir, bir sunucu yüzlerce isteği birlikte karşılar. Java bu işleri iş parçacığı denen bağımsız yürütme akışlarıyla yapar. Not önce iş parçacığını, süreci, eş zamanlılığı ve paralelliği tanıtır. Ardından `Thread` sınıfını, iş parçacığının yaşam döngüsünü, oluşturma ve başlatma yollarını ve iş parçacığını güvenle durdurmayı işler. Notun merkezinde paylaşılan verinin korunması vardır: yarış durumu, `synchronized`, `volatile`, açık kilitler, kilitlenme ve koşul beklemesi. Son kesimler `java.util.concurrent` paketinin hazır araçlarını, atomik sınıfları, kesmeyi, `ThreadLocal`'ı ve iş parçacığı havuzlarını tanıtır. Birden çok iş parçacığı içeren programların çıktısı çoğu zaman çalıştırmadan çalıştırmaya değişir. Notta bu tür her çıktı JDK 21 ile defalarca çalıştırılarak ölçüldü; hangi kısmın değişebileceği her örnekte belirtilir.

Bu bölümün kodu: `github.com/sercankulcu/object-oriented-programming-java/Bolum10/src`

ÖĞRENME ÇIKTILARI

Kavramlar
İş parçacığı ile süreç, eş zamanlılık ile paralellik arasındaki farkı açıklamak.

Oluşturma
`Runnable` arayüzünü uygulayarak ya da `Thread` sınıfını genişleterek iş parçacığı yazmak ve `start()` ile başlatmak.

Yaşam döngüsü
Altı durumu ve `sleep`, `join`, `wait`, kilit beklemesi gibi geçişleri izlemek.

Senkronizasyon
Yarış durumunu tanımak; `synchronized`, `volatile`, `Lock` ve atomik sınıflarla önlemek.

Koşul bekleme
`wait` / `notify` ve `Condition` ile üretici–tüketici yazmak; kilitlenmeyi önlemek.

Hazır araçlar
`CountDownLatch`, `CyclicBarrier`, `Exchanger`, `Semaphore`, `ThreadLocal` ve iş parçacığı havuzlarını kullanmak.

10.1 Temel kavramlar

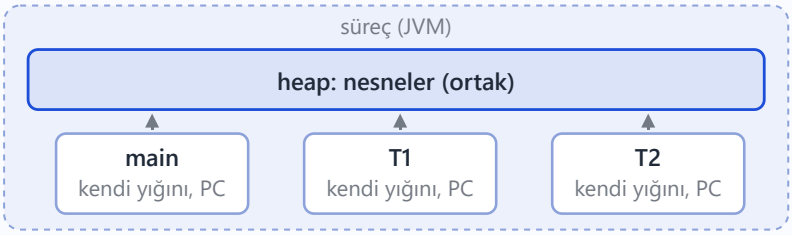
10.1.1 İş parçacığı

Şimdiye kadar yazılan bütün programlar tek bir akışla çalıştı. `main` metodunun ilk satırı çalışır, sonra ikincisi; bir metot çağrılırsa akış o metoda girer ve dönünce kaldığı yerden sürer. Herhangi bir anda programın "nerede olduğu" tek bir satırla anlatılabilir. Bu tek akış, iş parçacığının kendisidir.

Tanım 10.1

İş parçacığı (*thread*), bir programın içinde bağımsız ilerleyen yürütme akışıdır. Komutları sırayla çalıştırır; kendi çağrı yığını ve kendi program sayacı vardır. Java'da her iş parçacığı `java.lang.Thread` sınıfının bir nesnesiyle temsil edilir.

Her Java programı en az bir iş parçacığıyla başlar. JVM, `main()` metodunu çalıştırmak için bir iş parçacığı açar; bunun adı **ana iş parçacığı**dır (*main thread*). Program yeni iş parçacıkları açarsa bunlar ana iş parçacığıyla yan yana ilerler. Birden çok iş parçacığıyla çalışmaya **çoklu iş parçacığı** (*multithreading*) denir. Birden çok işin aynı zaman aralığında ilerlediği programlamanın adı da **eş zamanlı programlama**dır (*concurrent programming*).



Şekil 10.1: Bir süreçteki iş parçacıkları. Heap ortaktır; her iş parçacığının kendi yığını vardır.

Sunum durumu bir arı kovanına benzetir. Kovandaki her arı kendi işini yapar; bu, bir iş parçacığıdır. Ama bütün arılar aynı kovanı ve aynı petekleri kullanır; bu da ortak bellektir. İki arı aynı göze aynı anda bal koymaya kalkarsa sorun çıkar. Notun büyük kısmı bu çarpışmaların nasıl düzenleneceğini anlatır.

Birden çok iş parçacığına neden ihtiyaç duyulur? Üç neden öne çıkar:

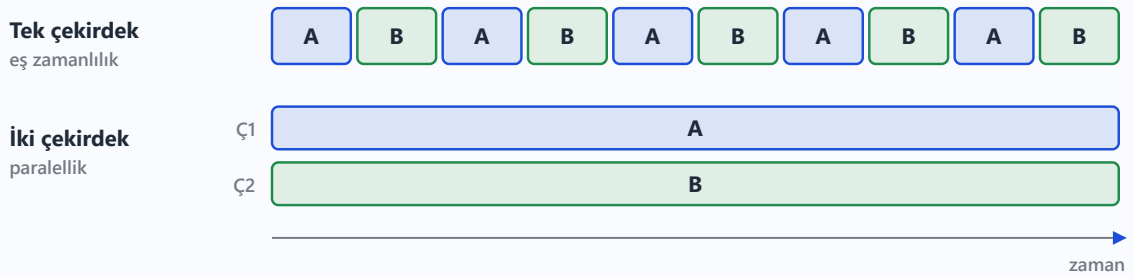
- **Yanıt veren arayüz.** Grafik arayüzlü bir programda uzun bir hesap, arayüzü çizen iş parçacığında yapılırsa pencere donar. Hesap ayrı bir iş parçacığına verilirse kullanıcı düğmelere basmayı sürdürebilir.
- **Çok çekirdekli işlemciyi kullanmak.** Bugünkü işlemcilerin çoğunun 4, 8 ya da daha fazla çekirdeği vardır. Tek iş parçacıklı bir program bunlardan yalnız birini kullanır. İş parçalara bölünürse çekirdekler aynı anda çalışır ve hesap hızlanır.
- **Beklerken çalışmak.** Ağdan veri gelmesi ya da diskten dosya okunması işlemciye göre çok yavaştır. Bekleyen iş parçacığı uyurken başka bir iş parçacığı işini sürdürür.

10.1.2 Eş zamanlılık ve paralellik

Gündelik dilde "aynı anda" sözü iki farklı durumu anlatır. Teknik dilde bu iki durum ayrı adlar taşır.

Tanım 10.2

Eş zamanlılık (*concurrency*), birden çok görevin aynı zaman aralığında ilerlemesidir. Tek çekirdekte bile olur: işletim sistemi iş parçacıklarını çok kısa dilimlerle sırayla çalıştırır. **Paralellik** (*parallelism*) ise görevlerin gerçekten aynı anda, farklı çekirdeklerde çalışmasıdır ve birden çok çekirdek gerektirir.



Şekil 10.2: Solda tek çekirdek: A ve B zaman dilimleriyle sırayla çalışır. Sağda iki çekirdek: A ve B aynı anda çalışır.

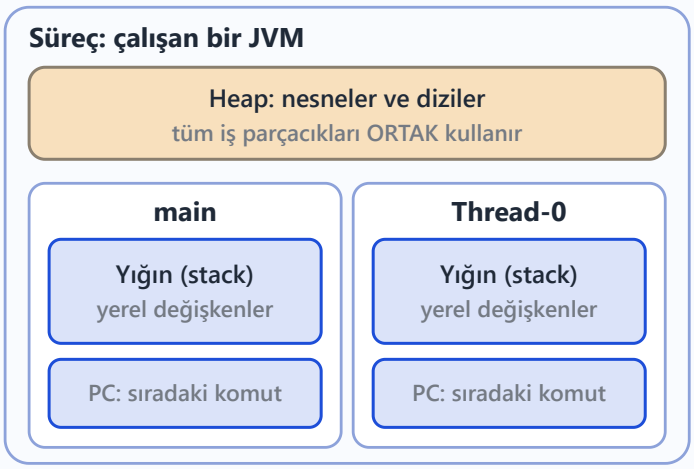
Şekil 10.2'nin sol yarısında tek çekirdek vardır. A bir dilim çalışır, sonra B, sonra yine A. Dilimler milisaniyeler mertebesinde olduğu için kullanıcı iki işi aynı anda ilerliyor sanır. Bu eş zamanlılıktır, ama paralellik değildir: belirli bir anda işlemcide tek bir iş parçacığı vardır. Sağ yarıda iki çekirdek vardır ve A ile B gerçekten aynı anda çalışır. Paralellik eş zamanlılığın özel bir durumudur.

Hangi iş parçacığının ne zaman ve hangi çekirdekte çalışacağına JVM ve işletim sisteminin **çizelgeleyicisi** (*scheduler*) karar verir. Programcı bu sırayı bilemez ve denetleyemez. Aynı program bir çalıştırmada A'yı önce, ötekinde B'yi önce çalıştırabilir. Bu belirsizlik, çok iş parçacıklı programların en belirgin özelliğidir.

10.1.3 Süreç ve iş parçacığı

Tanım 10.3

Süreç (*process*), kendi bellek alanı olan, çalışmakta olan programdır. İşletim sistemi her sürece ayrı bir adres alanı verir. Bir sürecin içinde bir ya da daha çok iş parçacığı çalışır; bu iş parçacıkları sürecin belleğini paylaşır.



Şekil 10.3: Bir süreç: ortak heap, her iş parçacığına ayrı yığın ve program sayacı.

	Süreç	İş parçacığı
Bellek	Kendi adres alanı	Sürecin heap'ini paylaşır
Yığın	—	Her birinin kendi yığını
Oluşturma	Pahalı	Ucuz (hafif süreç)
İletişim	Dosya, soket, boru	Ortak nesneler
Risk	Birbirini bozamaz	Ortak veriyi bozabilir

Tablo 10.1: Süreç ile iş parçacığının karşılaştırması.

Çalışan bir Java programı bir süreçtir; içinde birçok iş parçacığı çalışır. Tablo 10.1 iki kavramın farkını özetler. Heap'teki nesneler sürecin bütün iş parçacıklarına açıktır. İki iş parçacığı aynı nesnenin referansını tutuyorsa ikisi de o nesneyi okuyup değiştirebilir. Yerel değişkenler ise her iş parçacığının kendi yığınındadır ve paylaşılmaz. Bir metodu iki iş parçacığı aynı anda çalıştırsa bile metodun yerel değişkeni `i`'nin iki ayrı kopyası olur.

Bu ayrım iki yönlüdür. İyi yanı, iş parçacıkları arasında veri paylaşmanın çok kolay olmasıdır: ortak bir nesneye yazmak yeter. İki süreç ise birbirine ancak dosya, soket ya da boru gibi işletim sistemi araçlarıyla veri gönderebilir. Kötü yanı, birbirinden habersiz iki iş parçacığının ortak nesneyi aynı anda değiştirebilmesidir. Bir süreç ötekinin belleğini bozamaz; bir iş parçacığı ise aynı süreçteki her şeyi bozabilir.

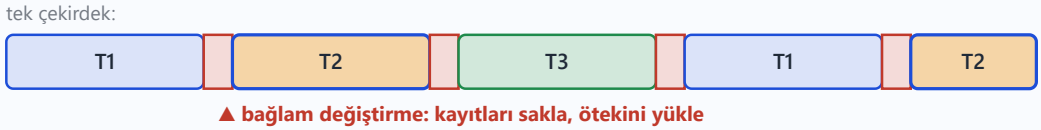
İş parçacığı oluşturmak süreç oluşturmaktan çok daha ucuzdur, çünkü yeni bir adres alanı açılmaz; yalnız yeni bir yığın ayrılır. Bu yüzden iş parçacığına **hafif süreç** (*lightweight process*) da denir.

10.1.4 Bağlam değiştirme

Bir bilgisayarda çoğu zaman çekirdek sayısından çok daha fazla iş parçacığı vardır. İşletim sistemi bunları zaman dilimleriyle sırayla çalıştırır. Bir iş parçacığının dilimi dolunca onun durumu saklanır, sıradakinin durumu yüklenir.

Tanım 10.4

Bağlam değiştirme (*context switching*), işlemcinin bir iş parçacığından ötekine geçmesidir. Geçişte çalışan iş parçacığının durumu (program sayacı, yazmaçlar, yığın işaretçisi) saklanır ve sıradakinin durumu yüklenir.



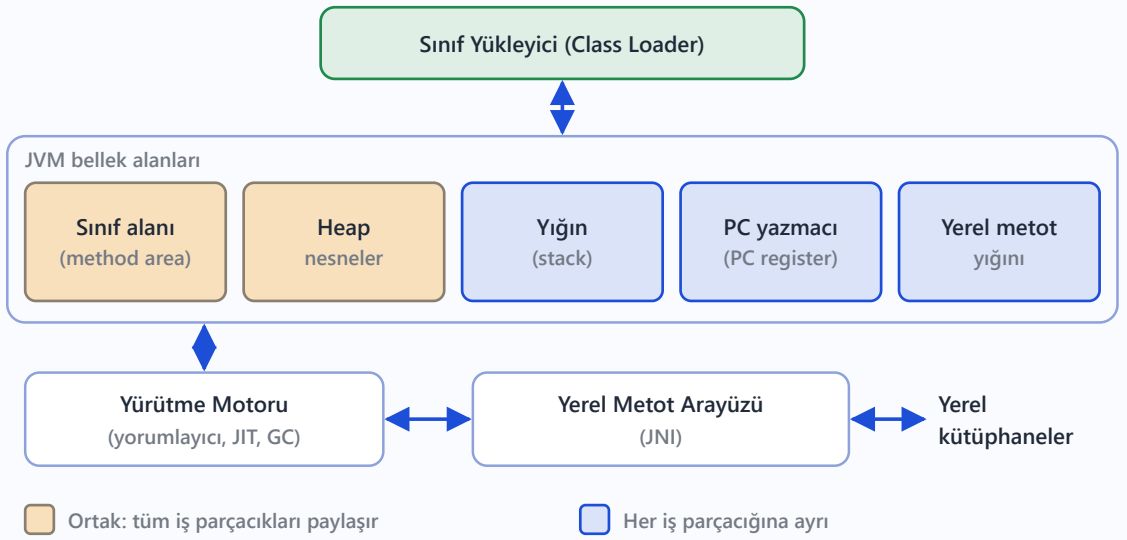
Şekil 10.4: Tek çekirdekte iş parçacıkları sırayla çalışır; aralarda bağlam değiştirilir.

Bağlam değiştirmenin bir bedeli vardır. Her geçiş birkaç mikrosaniye sürer. Daha önemlisi, işlemcinin önbelleğindeki verinin bir kısmı yeni iş parçacığı için işe yaramaz ve yeniden bellekten okunması gerekir. Bu yüzden iş parçacığı sayısını artırmak her zaman hızlandırmaz. Binlerce iş parçacığı açan bir program, zamanının önemli bir kısmını geçişlere harcar ve yavaşlar.

Pratik bir kural vardır. Hesap ağırlıklı işlerde iş parçacığı sayısı yaklaşık çekirdek sayısı kadar tutulur. Çekirdek sayısı `Runtime.getRuntime().availableProcessors()` ile öğrenilir; notun örnekleri 12 mantıksal çekirdekli bir bilgisayarda çalıştırıldı. Bekleme ağırlıklı işlerde (ağ, dosya) iş parçacıklarının çoğu zaman uyuduğu için daha fazlası yararlı olabilir. Sayıyı denetim altında tutmak için iş parçacığı havuzu kullanılır.

10.1.5 JVM ve iş parçacıkları

Java programları doğrudan işletim sisteminde değil, Java Sanal Makinesi (JVM, *Java Virtual Machine*) içinde çalışır. JVM, iş parçacıklarını bağımsız yürütme birimleri olarak yöneten sanal bir ortamdır. Günümüzün JVM'lerinde her Java iş parçacığı bir işletim sistemi iş parçacığına karşılık gelir; çizelgelemeyi işletim sistemi yapar.



Şekil 10.5: JVM mimarisi. Sınıf alanı ve heap tek kopyadır; yığın, PC yazmacı ve yerel metot yığını her iş parçacığına ayrıdır.

Şekil 10.5 JVM'in bellek alanlarını gösterir. Bir iş parçacığı oluşturulduğunda JVM ona üç özel alan ayırır:

- **Yığın** (*stack*): metot çağrılarının çerçevelerini, yerel değişkenleri ve parametreleri tutar.
- **PC yazmacı** (*program counter register*): iş parçacığının o anda çalıştırdığı komutun adresini tutar.
- **Yerel metot yığını** (*native method stack*): Java dışında, örneğin C ile yazılmış yerel metotların çağrıları içindir.

Sınıf alanı (sınıfların kodu ve `static` alanlar) ile heap (bütün nesneler) ise tek kopyadır ve bütün iş parçacıklarınca paylaşılır. `static` bir alan ve heap'teki bir nesne bu yüzden iki iş parçacığı arasında ortaktır; yerel değişkenler ortak değildir.

Sık yapılan hata

Yerel değişkenin de iş parçacıkları arasında paylaşıldığını sanmak. `void say() { int i = 0; i++; }` metodunu iki iş parçacığı aynı anda çalıştırırsa her biri kendi yığınında kendi `i`'sini artırır; ikisi de 1 bulur ve birbirini etkilemez. Sorun, iki iş parçacığının aynı nesnenin alanına ya da aynı `static` alana yazmasıyla başlar.

10.2 `Thread` sınıfı

10.2.1 Sınıf ve temel metotlar

Her iş parçacığı, `java.lang` paketindeki `Thread` sınıfının bir nesnesidir. Paket `java.lang` olduğu için `import` gerekmez. `Thread` nesnesi iş parçacığının yaşam döngüsünü yönetir: başlatır, bekletir, adını ve durumunu tutar.

İş parçacığı yazmanın bir yolu `Thread`'den türeyen bir alt sınıf yazmaktır. Alt sınıf `run()` metodunu geçersiz kılar (*override*) ve iş parçacığının yapacağı işi bu metoda yazar. İş parçacığı ise `start()` çağrılarak başlatılır. İki metodun ayrımı kritiktir: `run()` işin kendisidir, `start()` yeni akışı açan çağrıdır. Tablo 10.2 sınıfın sık kullanılan metotlarını özetler.

Metot	Görevi
`start()`	Yeni iş parçacığını başlatır; o da `run()`'ı çalıştırır
`run()`	İş parçacığının yapacağı iş
`sleep(ms)`	`static`: çalışan iş parçacığını belirli süre uyutur
`join()`	O iş parçacığının bitmesini bekler
`yield()`	`static`: yürütmeyi geçici olarak başkasına bırakır
`interrupt()`	Kesme isteği gönderir
`currentThread()`	`static`: şu an çalışan iş parçacığı
`getName()`, `getState()`, `isAlive()`	Bilgi okuma

Tablo 10.2: `Thread` sınıfının temel metotları.

`sleep`, `yield` ve `currentThread` metotlarının `static` olması anlamlıdır. Bunlar belirli bir `Thread` nesnesine değil, o satırı çalıştıran iş parçacığına etki eder. `Thread.sleep(100)` yazan iş parçacığı kendisi uyur; başka bir iş parçacığını uyutmanın yolu yoktur.

10.2.2 İş parçacığının nitelikleri

Bir `Thread` nesnesi, iş parçacığını tanımlayan birkaç nitelik taşır. Bunlar nesnenin alanlarıdır ve okumak ya da değiştirmek için get/set metotları vardır.

Çalıştırılabilir (*Runnable*)
: İş parçacığının çalıştıracağı `run()` metodunu içeren nesne. İş parçacığının ana mantığı burada tanımlanır.

Ad (*name*)
: `main`, `Thread-0`, `Thread-1` gibi. Ad verilmezse JVM `Thread-` ön ekine artan bir numara ekler. Kayıt dosyalarında ve hata ayıklamada iş parçacığını tanımak için kullanılır: `getName()`, `setName()`.

Kimlik (*ID*)

İş parçacığı oluşturulurken otomatik atanan tekil numara. Java 19'dan beri `threadId()` ile okunur; daha eski sürümlerde `getId()` kullanılırdı.

Grup (*ThreadGroup*)

İş parçacığının ait olduğu grup. İş parçacıklarını düzenlemek için kullanılır.

Arka plan (*daemon*)

Diğer iş parçacıklarına hizmet eden yardımcı iş parçacığı. JVM kapanırken bitmesi beklenmez.

Bağlam sınıf yükleyici (*context class loader*)

İş parçacığının sınıfları çalışma sırasında yüklerken kullandığı sınıf yükleyici.

Öncelik (*priority*)

`MIN_PRIORITY` (1) ile `MAX_PRIORITY` (10) arasında bir tam sayı; varsayılan `NORM_PRIORITY` (5).

Durum (*state*)

`NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING`, `TERMINATED` değerlerinden biri.

Kesilme (*interrupted*)

Kesme isteği gelip gelmediğini gösteren bayrak; `interrupt()` ile kurulur.

Aşağıdaki kod parçası `main` içinde çalışır ve o anda çalışan iş parçacığının niteliklerini yazar.

```
Thread t = Thread.currentThread();   // çalışan
System.out.println("Ad      : " + t.getName());
// threadId(): Java 19+ (öncesinde getId())
System.out.println("Kimlik  : " + t.threadId());
System.out.println("Öncelik : " + t.getPriority());
String grup = t.getThreadGroup().getName();
System.out.println("Grup    : " + grup);
System.out.println("Daemon  : " + t.isDaemon());
System.out.println("Durum   : " + t.getState());
```

```
Ad      : main
Kimlik  : 1
Öncelik : 5
Grup    : main
Daemon  : false
Durum   : RUNNABLE
```

`Thread.currentThread()`, o satırı çalıştıran iş parçacığını döndürür. Kod `main` içinde olduğu için bu, ana iş parçacığıdır. Adı `main`'dir, kimliği 1'dir, önceliği varsayılan 5'tir. Grubu da `main` adını taşır. Ana iş parçacığı arka plan iş parçacığı değildir. Durumu `RUNNABLE`'dır, çünkü `getState()`'i çağıran iş parçacığı o anda çalışıyor. JDK 21 ile çalıştırılan program bu altı satırı her seferinde aynen yazdı.

Öncelik ve arka plan niteliği, iş parçacığını oluşturan iş parçacığından kalıtılır. `main` içinde oluşturulan bir iş parçacığının önceliği 5'tir ve arka plan iş parçacığı değildir. Bu değerler `start()`'tan önce değiştirilebilir.

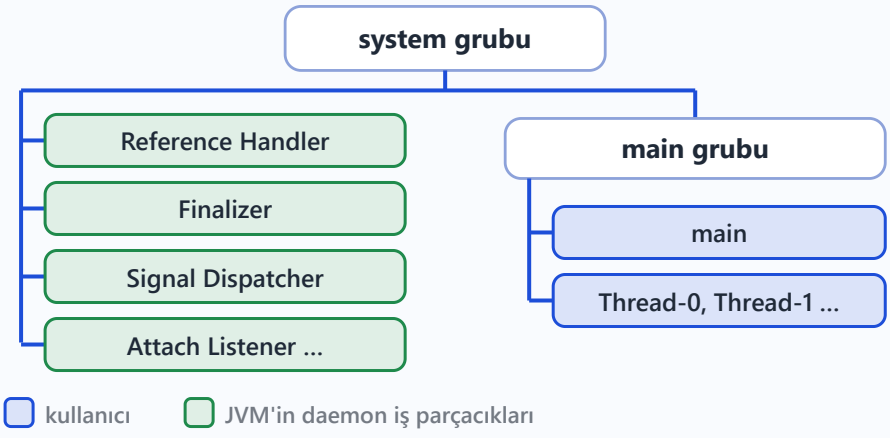
Örnek 10.1

`main` içinde, başka iş parçacığı oluşturulmadan önce şu satırlar çalışıyor: `Thread a = new Thread(() -> { }, "isci");`, `Thread b = new Thread(() -> { });`, `Thread c = new Thread(() -> { });` ve `c.setPriority(Thread.MAX_PRIORITY);`. `a`, `b`, `c` 'nin adları ile `b` ve `c` 'nin öncelikleri nedir?

Adlar `isci`, `Thread-0` ve `Thread-1` 'dir. Ad verilen iş parçacığı otomatik numara almaz; numara yalnız adsız iş parçacıklarına sırayla verilir. `b` 'nin önceliği oluşturan iş parçacığından (`main`) kalıtılan 5'tir; `c` 'ninki 10'dur. Üçü de hiç başlatılmadığı için durumları `NEW` 'dur. Program JDK 21'de `isci Thread-0 Thread-1 ve 5 10 false` yazdı.

10.2.3 İş parçacığı grupları

JVM başlarken iki iş parçacığı grubu oluşturur. `system` **grubu** JVM'in kendi iş parçacıklarını içerir: referansları yöneten `Reference Handler`, sonlandırıcı `Finalizer`, sinyalleri karşılayan `Signal Dispatcher` ve benzerleri. `main` **grubu** `system` 'in alt grubudur; ana iş parçacığı ve programın oluşturduğu iş parçacıkları varsayılan olarak buraya aittir.



Şekil 10.6: İş parçacığı grupları: `system` grubu altında JVM iş parçacıkları ve `main` grubu.

`Thread.getAllStackTraces()`, canlı bütün iş parçacıklarını ve yığınlarını bir eşlem olarak döndürür. Anahtar kümesinde gezinilerek hepsi listelenir:

```
Set<Thread> hepsi =
        Thread.getAllStackTraces().keySet();
for (Thread t : hepsi) {
    System.out.println(t.getThreadGroup().getName()
            + "  " + t.getName()
            + "  daemon=" + t.isDaemon());
}
```

```
system  Notification Thread  daemon=true
main  main  daemon=false
system  Signal Dispatcher  daemon=true
system  Reference Handler  daemon=true
system  Attach Listener  daemon=true
InnocuousThreadGroup  Common-Cleaner  daemon=true
system  Finalizer  daemon=true
```

Bu çıktı JDK 21'de bir çalıştırmadan alındı. Listede yedi iş parçacığı vardır ve `main` dışındakilerin hepsi arka plan iş parçacığıdır. `Common-Cleaner` ayrı bir grupta, `InnocuousThreadGroup`'tadır. Satırların sırası eşlemin iç düzenine bağlıdır. On çalıştırmada sıra hep aynıydı, ama bu bir garanti değildir. Listede hangi JVM iş parçacıklarının bulunacağı da JVM sürümüne ve ayarlarına göre değişir.

10.2.4 JVM başlarken çalışan iş parçacıkları

Sunum, JVM başladığında var olan dört temel iş parçacığını öne çıkarır:

Ana iş parçacığı (*main thread*)
: JVM başladığında ilk çalışan iş parçacığıdır. `main()` metodu burada çalışır; uygulamanın temel mantığı burada yürür.

Çöp toplayıcı (*garbage collector*)
: Bellek yönetiminden sorumludur. Referansı kalmamış nesneleri bulur ve temizler. HotSpot JVM'inde çöp toplayıcının iş parçacıkları JVM'in iç iş parçacıklarıdır; Java nesnesi olarak görünmezler, bu yüzden yukarıdaki listede yer almazlar.

Sinyal gönderici (*Signal Dispatcher*)
: İşletim sisteminden gelen sinyalleri karşılar. Örneğin Ctrl+C ile gelen kapatma isteğinde JVM'in düzenli kapanmasını başlatır.

Sonlandırıcı (*Finalizer*)
: Nesnelerin `finalize()` metodunu çağırır. `finalize()` Java 9'da kullanımdan kaldırıldı (*deprecated*), Java 18'den beri de ileride silinecek olarak işaretlidir. Yerine `try-finally` ya da `AutoCloseable` ile try-with-resources kullanılır.

10.3 Yaşam döngüsü

10.3.1 Altı durum

Bir iş parçacığı oluşturulduğu andan bittiği ana kadar farklı durumlardan geçer. `getState()` metodu o anki durumu `Thread.State` numaralandırmasının bir sabiti olarak döndürür. Java altı durum tanımlar.

Durum	Anlamı	Nasıl girilir?
`NEW` (yeni)	Nesne oluşturuldu, henüz başlatılmadı	`new Thread(…)` ; çıkış: `start()`

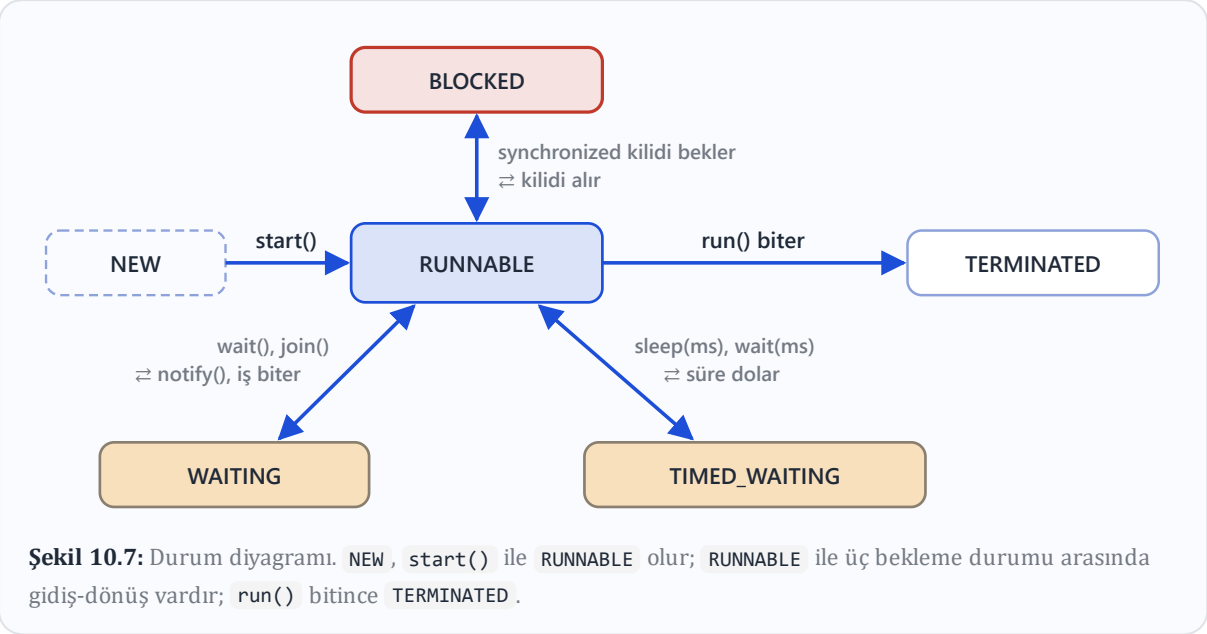
Durum	Anlamı	Nasıl girilir?
`RUNNABLE` (çalışabilir)	Çalışıyor ya da işlemci sırası bekliyor	`start()` sonrası
`BLOCKED` (engellenmiş)	`synchronized` blok ya da metoda girmek için monitör kilidini bekliyor	Kilit başkasındayken
`WAITING` (bekliyor)	Süresiz bekliyor; başka bir iş parçacığı uyandırmalı	`Object.wait()`, `Thread.join()`, `LockSupport.park()`
`TIMED_WAITING` (süreli bekliyor)	Süre sınırlı bekliyor	`Thread.sleep(ms)`, `wait(ms)`, `join(ms)`, `parkNanos`, `parkUntil`
`TERMINATED` (sonlandı)	Yürütmeyi tamamladı	`run()` bitince

Tablo 10.3: İş parçacığı durumları.

Tablo 10.3 bir noktayı açıkça gösterir: Java'da ayrı bir "çalışıyor" (*RUNNING*) durumu yoktur. İşlemcide o anda çalışan iş parçacığı da, sırasını bekleyen iş parçacığı da `RUNNABLE`'dır. Hangisinin gerçekten işlemcide olduğunu çizelgeleyici bilir; JVM bu ayrımı programa göstermez.

Üç bekleme durumu birbirine karıştırılmamalıdır. `BLOCKED` yalnız bir monitör kilidini beklemektir: iş parçacığı bir `synchronized` bloğa girmek ister, ama kilit başka bir iş parçacığındadır. `WAITING` bir olayı süresiz beklemektir: başka bir iş parçacığının bitmesi (`join`) ya da uyandırması (`notify`) gerekir. `TIMED_WAITING` ise süreli beklemedir; süre dolunca iş parçacığı kendiliğinden döner. `sleep` bu yüzden her zaman `TIMED_WAITING` üretir.

10.3.2 Durum diyagramı



Şekil 10.7: Durum diyagramı. `NEW`, `start()` ile `RUNNABLE` olur; `RUNNABLE` ile üç bekleme durumu arasında gidiş-dönüş vardır; `run()` bitince `TERMINATED`.

Şekil 10.7 geçişleri gösterir. Ana yol tek yönlüdür: `NEW` → `RUNNABLE` → `TERMINATED`. Geri dönüş yoktur. Biten bir iş parçacığı yeniden başlatılamaz; aynı işi bir kez daha yapmak için yeni bir `Thread` nesnesi gerekir. Üç bekleme durumunun her birinden ise yeniden `RUNNABLE`'a dönülür: kilit boşalınca, beklenen olay olunca ya da süre dolunca.

10.3.3 Kod izleme: durumları gözlemlemek

Aşağıdaki program dört ayrı iş parçacığını farklı durumlara sokar ve durumlarını yazar. `uyu(ms)` ve `katil(t)`, `Thread.sleep` ve `t.join()` çağrılarını `try-catch` ile saran yardımcı metotlardır. `kilit` ise `static final Object` türünde bir alandır.

```
public static void main(String[] args) {
    Thread uyuyan = new Thread(() -> uyu(1000));
    System.out.println("1) " + uyuyan.getState());
    uyuyan.start();
    uyu(100);
    System.out.println("2) " + uyuyan.getState());
    Thread bekleyen = new Thread(() -> katil(uyuyan));
    bekleyen.start();
    uyu(100);
    System.out.println("3) " + bekleyen.getState());
    Thread engellenen = new Thread(() -> {
        synchronized (kilit) { }
    });
    synchronized (kilit) {
        engellenen.start();
        uyu(100);
        System.out.println("4) " + engellenen.getState());
    }
    katil(bekleyen);
    System.out.println("5) " + uyuyan.getState());
}
```

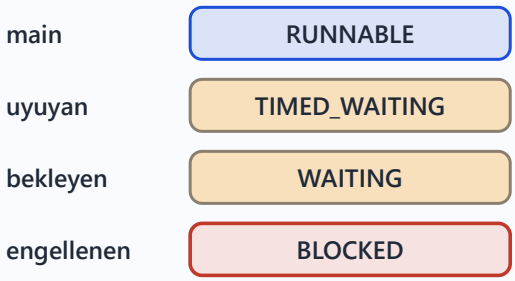
Adım	Satır	Değişkenler / durum	Yeni çıktı
1	2: `Thread uyuyan = new Thread(() -> uyu(1000));`	—	—
2	3: `System.out.println("1) " + uyuyan.getState());`	—	`1) NEW`
3	4: `uyuyan.start();`	—	—
4	5: `uyu(100);`	—	—
5	6: `System.out.println("2) " + uyuyan.getState());`	—	`2) TIMED_WAITING`
6	8: `bekleyen.start();`	—	—
7	10: `System.out.println("3) " + bekleyen.getState());`	—	`3) WAITING`
8	15: `engellenen.start();`	—	—
9	17: `System.out.println("4) " + engellenen.getState());`	—	`4) BLOCKED`
10	19: `katil(bekleyen);`	—	—

Adım	Satır	Değişkenler / durum	Yeni çıktı
11	20: `System.out.println("5) " + uyuyan.getState());`	—	`5) TERMINATED`

Tablo 10.4: Durum gözlemi programının izlemesi.

1. ve 2. adımda `uyuyan` nesnesi oluşturulur ama başlatılmaz; durumu `NEW`'dur. 3. adımda `start()` çağrılır. Yeni iş parçacığı hemen `uyu(1000)`'e girer ve bir saniye uyur. Bu sırada `main` de 100 ms uyur. `main` uyandığında `uyuyan` hâlâ uykudadır: `TIMED_WAITING`.

6. adımda `bekleyen` başlatılır ve `uyuyan.join()`'i çağırır. `join` süresizdir; `bekleyen`, `uyuyan` bitene kadar `WAITING` durumunda kalır. 8. adımda `main`, `kilit`'in monitörünü alır ve ardından `engellenen`'i başlatır. `engellenen` aynı kilitle bir `synchronized` bloğa girmek ister, ama kilit `main`'dedir. `main` bloğun içinde 100 ms uyurken durumu yazar: `BLOCKED`.



Şekil 10.8: 9. adım. `main` kilidi tutuyor ve uyuyor; `uyuyan` uykuda, `bekleyen` onu bekliyor, `engellenen` kilidi bekliyor.

Şekil 10.8 9. adımdaki durumu gösterir: dört iş parçacığı dört farklı durumdadır. Bir noktaya dikkat edilmelidir. `main` `sleep` içinde uyurken kilidi bırakmaz; `engellenen` bu yüzden içeri giremez. Blok bitince kilit bırakılır, `engellenen` girip hemen çıkar ve biter. `main` ardından `bekleyen`'i bekler. `uyuyan` bir saniyesini doldurup biter; bunun üzerine `bekleyen` de biter. Son satır `uyuyan`'ın durumunu yazar: `TERMINATED`.

Bu programın çıktısı çok iş parçacıklı olmasına rağmen sabittir. Bekleme süreleri, her gözlem anında iş parçacıklarının hangi durumda olacağını belirler: 100 ms, bir iş parçacığının başlayıp `sleep`'e ya da `join`'e girmesine yeter. Program 8 kez çalıştırıldı ve her seferinde aynı beş satırı yazdı. Yine de bu tür süre varsayımları ağır yüklü bir bilgisayarda bozulabilir. Süreler senkronizasyon için değil, yalnız gözlem için kullanılmıştır.

10.4 Oluşturma ve başlatma

10.4.1 `Runnable` arayüzü

Her iş parçacığı bir `Runnable` nesneyi çalıştırır. `Runnable` arayüzü `java.lang` paketindedir ve tek bir metot içerir:

```
package java.lang;

public interface Runnable {
    void run();
}
```

Bu arayüzü uygulayan sınıf `run()` metodunu yazar. Metodun gövdesi, iş parçacığının yapacağı iştir.

```
public class MyRunnable implements Runnable {
    @Override
    public void run() {
        // İş parçacığının yapması gereken görevler
    }
}
```

`Runnable`'ın tek soyut metodu olduğu için o bir fonksiyonel arayüzdür. Java 8'den beri böyle bir arayüzün yerine lambda ifadesi yazılabilir. Kısa görevler için ayrı bir sınıf yazmak gerekmez:

```
Runnable r = () -> System.out.println("merhaba");
```

Lambda, `run()` metodunun gövdesini doğrudan verir. Parametre listesi boştur, çünkü `run()` parametre almaz; dönüş tipi `void` olduğu için lambda değer döndürmez.

10.4.2 İş parçacığını başlatmak

Bir `Runnable` nesnesi tek başına iş parçacığı değildir; yalnızca yapılacak işi anlatır. İşi yeni bir iş parçacığında çalıştırmak için nesne `Thread` yapıcısına verilir ve `start()` çağrılır.

```
public class Worker implements Runnable {
    // İş parçacığının görevleri burada tanımlanır
    public void run() {
        String ad = Thread.currentThread().getName();
        System.out.println("Worker çalışıyor: " + ad);
    }
    public static void main(String[] args) {
        // İş parçacığını başlatma
        Worker w = new Worker();
        Thread t = new Thread(w);
        t.start();
        System.out.println("main devam ediyor");
    }
}
```

Sunum bir çalıştırmanın çıktısını şöyle verir:

```
main devam ediyor
Worker çalışıyor: Thread-0
```

`main`, `Worker` nesnesini oluşturur ve onu yapıcıya vererek bir `Thread` oluşturur. `t.start()` yeni iş parçacığını açar ve hemen döner. Bundan sonra iki iş parçacığı vardır: `main` bir sonraki satıra geçip `main devam ediyor` yazar; `Thread-0` ise `run()`'ı çalıştırıp kendi adını yazar. Ad otomatik verilmiştir: programın oluşturduğu ilk adsız iş parçacığı `Thread-0`'dır.

İki satırdan hangisinin önce yazılacağı belli değildir. `start()` döner dönmez `main` çalışmayı sürdürür; yeni iş parçacığının ne zaman işlemciye alınacağını çizelgeleyici belirler. Program JDK 21 ile 40 kez çalıştırıldı: hepsinde önce `main devam ediyor` satırı geldi. Ters sıra da geçerlidir ve başka bir bilgisayarda ya da yük altında görülebilir. Ölçülen dağılım bilgisayara ve yüke göre değişir; programın doğruluğu hiçbir sıraya dayanmamalıdır.

Sık yapılan hata

`start()` yerine `run()`'ı çağırmak. `t.run()` sıradan bir metot çağrısıdır: kod çağıranın iş parçacığında, yani `main`'de çalışır ve yeni iş parçacığı açılmaz. Program hata vermez, ama hiçbir şey paralel çalışmaz. Bu yüzden hata çoğu zaman fark edilmez.

10.4.3 Kod izleme: iki yığın, iki akış

Aynı programı adım adım izlemek, `start()`'ın gerçekte ne yaptığını gösterir. İzleme, olası sıralardan birini, `main`'in önce yazdığı sırayı gösterir.

Adım	Satır	Değişkenler / durum	Yeni çıktı
1	9: `Worker w = new Worker();`	w=Worker nesnesi, t=—	—
2	10: `Thread t = new Thread(w);`	w=Worker nesnesi, t=Thread-0 (NEW)	—
3	11: `t.start();`	w=Worker nesnesi, t=Thread-0 (RUNNABLE)	—
4	12: `System.out.println("main devam ediyor");`	w=Worker nesnesi, t=Thread-0 (RUNNABLE)	`main devam ediyor`
5	13: `}`	t=Thread-0 (RUNNABLE)	—
6	4: `String ad = Thread.currentThread().getName();`	t=Thread-0 (RUNNABLE), ad="Thread-0"	—
7	5: `System.out.println("Worker çalışıyor: " + ad);`	t=Thread-0 (RUNNABLE), ad="Thread-0"	`Worker çalışıyor: Thread-0`
8	6: `}`	t=Thread-0 (TERMINATED)	—

Tablo 10.5: `Worker` programının bir olası sırası.

2. adımda `Thread` nesnesi oluşturulmuştur ve durumu `NEW`'dur. Ama henüz ayrı bir yürütme akışı yoktur; nesne heap'te bekleyen sıradan bir nesnedir. 3. adımda `start()` çağrılır. JVM yeni bir yığın açar ve `run()`'ı o yığında başlatır. `start()` ise `run()`'ın bitmesini beklemeden döner.



Şekil 10.9: 6. adım. `main()` bitti, ama `Thread-0` kendi yığınında `run()`'ı çalıştırıyor. `ad` onun yerel değişkenidir.

5. adımda `main()` metodu biter. Program ise bitmez. JVM, arka plan iş parçacığı olmayan bütün iş parçacıkları bitene kadar çalışmayı sürdürür. Şekil 10.9 bu anı gösterir: `main`'in yığını boşalmıştır, `Thread-0`'ın yığınında `run()` çerçevesi vardır. `ad` değişkeni bu çerçevededir ve başka hiçbir iş parçacığı onu göremez. 8. adımda `run()` biter ve `Thread-0` `TERMINATED` olur. Çalışan iş parçacığı kalmadığı için JVM kapanır.

10.4.4 `Thread` sınıfını genişletmek

İş parçacığı, `Thread` sınıfı genişletilerek de yazılabilir. `Thread` sınıfı zaten `Runnable`'ı uygular; alt sınıf yalnız `run()`'ı geçersiz kılar.

```
public class Worker2 extends Thread {
    // İş parçacığının görevleri burada tanımlanır
    public void run() {
        String ad = getName();   // Thread'den kalıtıldı
        System.out.println("Worker çalışıyor: " + ad);
    }
    public static void main(String[] args) {
        // İş parçacığını oluşturma ve başlatma
        Worker2 w = new Worker2();
        w.start();
        System.out.println("main devam ediyor");
    }
}
```

```
main devam ediyor
Worker çalışıyor: Thread-0
```

Bu yolda `Runnable` nesnesi ile `Thread` nesnesi aynıdır. Nesnenin kendisi bir `Thread` olduğu için `w.start()` doğrudan çağrılır. `run()` içinde de `getName()` ön eksiz yazılır, çünkü metot `Thread`'den kalıtılmıştır. Çıktının sırası yine değişebilir: 40 çalıştırmada hepsinde yine önce `main` yazdı.

10.4.5 `start()` mı, `run()` mı?

`start()` ile `run()` arasındaki fark, aynı nesne üzerinde ikisini art arda çağırarak görülür.

```
public class StartRunFarki {
    public static void main(String[] args)
            throws InterruptedException {
        Thread t = new Thread(() -> {
            String ad = Thread.currentThread().getName();
            System.out.println("run() içinde: " + ad);
        });
        t.run();     // YANLIŞ: main çalıştırır
        t.start();   // DOĞRU: yeni iş parçacığı
        t.join();
    }
}
```

```
run() içinde: main
run() içinde: Thread-0
```

İlk satır `t.run()`'dan gelir. Bu çağrı, `main` içinde sıradan bir metot çağrısıdır; `currentThread()` `main`'i döndürür. İkinci satır `t.start()`'tan gelir: `run()` artık yeni iş parçacığında, `Thread-0`'da çalışır. Çıktının sırası sabittir: `t.run()` çağrısı bitmeden `t.start()` satırına gelinmez. `join()` ise `main`'i `Thread-0` bitene kadar bekletir.

Çağrı	Ne olur?
`t.run()`	Sıradan bir metot çağrısı: aynı iş parçacığında (`main`) çalışır, yeni yığın yok, bitene kadar `main` bekler.
`t.start()`	JVM yeni bir iş parçacığı ve yığın oluşturur; `run()` orada çalışır, `start()` hemen döner.
`t.start()` ikinci kez	`IllegalThreadStateException` : bir `Thread` nesnesi yalnızca bir kez başlatılabilir.

Tablo 10.6: `run()` ile `start()` çağrılarının farkı.

Kural kısadır: iş parçacığının işi `run()` içine yazılır, iş parçacığı `start()` ile başlatılır. `start()` aynı nesnede ikinci kez çağrılırsa JDK 21 hem iş parçacığı çalışırken hem de bittikten sonra `java.lang.IllegalThreadStateException` fırlattı. Bunun nedeni durum diyagramındaki tek yönlü yoldur: `start()` yalnız `NEW` durumundaki bir iş parçacığında geçerlidir.

Örnek 10.2

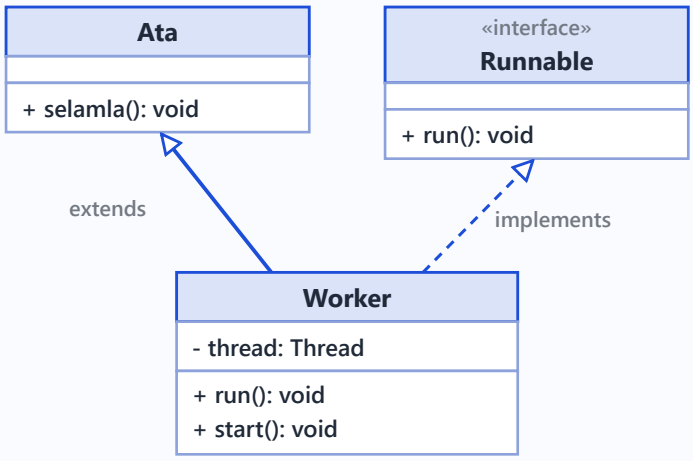
Adı `T` olan bir iş parçacığı oluşturuluyor; işi `"run: " + Thread.currentThread().getName()` yazmaktır. Ardından sırasıyla `t.run();`, `System.out.println("A " + t.getState());`, `t.start();`, `t.join();` ve `System.out.println("B " + t.getState());` çalışıyor. Program ne yazdırır?

`t.run()` çağrısı `main`'de çalışır: `run: main`. İş parçacığı hâlâ hiç başlatılmamıştır: `A NEW`. `start()` sonrası `run()`, `T`'nin kendi yığınında çalışır: `run: T`. `join()` `T` bitene kadar bekler; son satır `B TERMINATED`'dır. `join()` sırayı sabitlediği için bu dört satır her çalıştırmada aynıdır; program beş kez çalıştırıldı ve hep aynı çıktıyı verdi.

10.4.6 Tek kalıtım sorunu

Java bir sınıfın yalnızca tek bir sınıftan kalıtım almasına izin verir. Çalıştırılmak istenen sınıf zaten başka bir sınıftan türemişse `Thread`'i genişletemez. `class Worker extends Ata, Thread` yazmak derleme hatasıdır; `javac` virgülün olduğu yerde `'{' expected` hatası verir.

Çözüm `Runnable` arayüzündedir. Bir sınıfın uygulayabileceği arayüz sayısında sınır yoktur. Sınıf `Ata`'yı genişletir, `Runnable`'ı uygular ve gerekirse içinde `Thread` tipinde bir alan tutar.



Şekil 10.10: `Worker`, `Ata` sınıfını genişletir ve `Runnable` arayüzünü uygular; `thread` adlı bir `Thread` alanı vardır.

```
class Ata {
    public void selamla() {
        System.out.println("Ata sınıfından merhaba");
    }
}
class Worker extends Ata implements Runnable {
    private Thread thread;   // Thread alan olarak
    public void run() {
        String ad = Thread.currentThread().getName();
        System.out.println("run(): " + ad);
    }
    public void start() {
        thread = new Thread(this);
        thread.start();
    }
}
public class TekMirasOrnegi {
    public static void main(String[] args) {
        Worker w = new Worker();
        w.selamla();   // Ata'dan kalıtılan metot
        w.start();     // kendi start() metodumuz
    }
}
```

```
Ata sınıfından merhaba
run(): Thread-0
```

`Worker` hem `Ata`'nın metotlarını kalıtır hem de bir iş parçacığında çalışabilir. `w.selamla()` `Ata`'dan gelen metottur ve `main`'de çalışır. `w.start()` ise `Thread`'in değil, `Worker`'ın kendi yazdığı `start()` metodudur. Bu metot `new Thread(this)` ile kendini bir `Thread`'e verir ve onu başlatır. `this`, `Runnable` tipindendir; `Thread` yapıcısı bu yüzden onu kabul eder. Çıktıdaki iki satırın sırası sabittir: `selamla()`, iş parçacığı başlatılmadan önce biter.

Bu kalıp, iş ile iş parçacığını birbirinden ayırır. `Worker` "ne yapılacağını" anlatır, `Thread` nesnesi "nasıl çalıştırılacağını". Aynı `Runnable` nesnesi ileride bir iş parçacığı havuzuna da verilebilir. `Runnable`'ın çoğu durumda tercih edilmesinin nedeni budur.

10.4.7 Üç yolun karşılaştırması

Aşağıdaki program iş parçacığı oluşturmanın üç yolunu yan yana koyar. `static class`, başka bir sınıfın içinde tanımlanan sınıftır; dış sınıfın nesnesi olmadan kullanılabilir.

```
public class UcYol {
    static class Gorev implements Runnable {     // 1)
        public void run() {
            System.out.println("1) Runnable sınıfı");
        }
    }
    static class IsParcacigi extends Thread {    // 2)
        public void run() {
            System.out.println("2) Thread alt sınıfı");
        }
    }
    public static void main(String[] args)
            throws InterruptedException {
        Thread t1 = new Thread(new Gorev());
        Thread t2 = new IsParcacigi();
        // 3) Lambda (Java 8+)
        Thread t3 = new Thread(
                () -> System.out.println("3) Lambda"));
        t1.start(); t1.join();
        t2.start(); t2.join();
        t3.start(); t3.join();
    }
}
```

```
1) Runnable sınıfı
2) Thread alt sınıfı
3) Lambda
```

Her `start()`'tan hemen sonra `join()` geldiği için bir iş parçacığı bitmeden öteki başlamaz. Çıktı sırası bu yüzden sabittir. Elbette bu kullanım iş parçacıklarının bütün yararını ortadan kaldırır; program yalnız üç yolu göstermek için yazılmıştır.

Yol	Ne zaman?
`implements Runnable`	Sınıf başka sınıftan türemişse; iş ile iş parçacığı ayrı tutulur
`extends Thread`	Kısa, bağımsız örnekler; kalıtım hakkı harcanır
Lambda (Java 8+)	Kısa görevler için en pratik yol

Tablo 10.7: İş parçacığı oluşturmanın üç yolu.

10.4.8 Kod izleme: iki iş parçacığı

İki iş parçacığı gerçekten birlikte çalıştığında çıktı nasıl oluşur? Aşağıdaki programda `A` ve `B` adlı iki iş parçacığı aynı `gorev()` metodunu çalıştırır. Thread yapıcısının ikinci argümanı iş parçacığının adıdır.

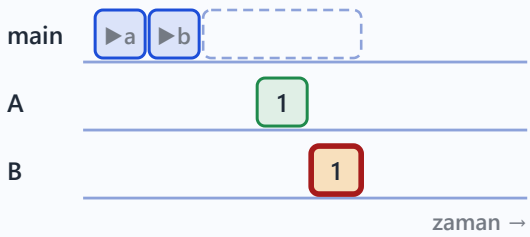
```
public class IkiIsParcacigi {
    static void gorev() {
        String ad = Thread.currentThread().getName();
        for (int i = 1; i <= 2; i++) {
            System.out.println(ad + ": adım " + i);
        }
    }
    public static void main(String[] args)
            throws InterruptedException {
        Thread a = new Thread(() -> gorev(), "A");
        Thread b = new Thread(() -> gorev(), "B");
        a.start();
        b.start();
        a.join();
        b.join();
        System.out.println("main: bitti");
    }
}
```

Adım	Satır	Çağrı yığını	Değişkenler / durum	Yeni çıktı
1	12: `a.start();`	main()	a=RUNNABLE, b=NEW	—
2	13: `b.start();`	main()	a=RUNNABLE, b=RUNNABLE	—
3	14: `a.join();`	main()	a=RUNNABLE, b=RUNNABLE, main=WAITING	—
4	5: `System.out.println(ad + ": adım " + i);`	A: run() → gorev()	A.i=1, B.i=—, main=WAITING	`A: adım 1`
5	5: `System.out.println(ad + ": adım " + i);`	B: run() → gorev()	A.i=1, B.i=1, main=WAITING	`B: adım 1`
6	5: `System.out.println(ad + ": adım " + i);`	A: run() → gorev()	A.i=2, B.i=1, main=WAITING	`A: adım 2`
7	5: `System.out.println(ad + ": adım " + i);`	B: run() → gorev()	A.i=2, B.i=2, main=WAITING	`B: adım 2`

Adım	Satır	Çağrı yığını	Değişkenler / durum	Yeni çıktı
8	16: `System.out.println("main: bitti");`	main()	A=TERMINATED, B=TERMINATED, main=RUNNABLE	`main: bitti`

Tablo 10.8: İki iş parçacığının bir olası sırası (sunumdaki sıra).

1. ve 2. adımda `main` iki iş parçacığını başlatır. İkisi de `RUNNABLE`'dır, ama hangisinin ne zaman çalışacağı belli değildir. 3. adımda `main`, `a.join()` ile `A`'nın bitmesini beklemeye başlar ve `WAITING` olur. Artık çalışan `A` ve `B`'dir.



Şekil 10.11: 5. adım. Bağlam değişimi: şimdi `B` çalışıyor. `B`'nin `i`'si `A`'nınkinden ayrı bir yerel değişkendir.

4.–7. adımlarda `A` ve `B` sırayla birer satır yazar. Her bağlam değişiminde öteki iş parçacığı kaldığı yerden devam eder. `A`'nın `i`'si ile `B`'nin `i`'si ayrı yığınlardadır; birinin artması ötekini etkilemez. 8. adımda `a.join()` döner. `B` zaten bittiği için `b.join()` beklemeden döner ve `main` son satırı yazar.

Bu sıra olası sıralardan yalnız biridir. Program JDK 21 ile 60 kez çalıştırıldı; görülen çıktılar ve sayıları:

```
32 kez:  B: adım 1 · B: adım 2 · A: adım 1 · A: adım 2 · main: bitti
25 kez:  A: adım 1 · A: adım 2 · B: adım 1 · B: adım 2 · main: bitti
 3 kez:  B: adım 1 · A: adım 1 · A: adım 2 · B: adım 2 · main: bitti
```

Aynı iki iş parçacığı tek bir JVM içinde art arda başlatıldığında tablo değişti: 6.000 denemede A1 · A2 · B1 · B2 5.958 kez, B1 · B2 · A1 · A2 35 kez, A1 · B1 · A2 · B2 5 kez, A1 · B1 · B2 · A2 2 kez. Isınmış bir JVM'de yeni iş parçacığı çok çabuk başlar ve kısa işini öteki başlamadan bitirir. Sunumdaki sıra seyrek de olsa gerçekten görüldü. Kesin olan iki şey vardır. Birincisi, `main: bitti` her zaman en sondadır, çünkü iki `join()`'den sonra yazılır. İkincisi, aynı iş parçacığının iki satırı kendi aralarında sırasını korur: `A: adım 2` hiçbir zaman `A: adım 1`'den önce gelmez. İki iş parçacığının satırları ise birbirine her biçimde karışabilir.

Örnek 10.3

`main` içinde sırasıyla şunlar çalışıyor: `a`'yı başlat, `a.join()`, `b`'yi başlat, `"m"` yaz, `b.join()`. `a` `"a"`, `b` `"b"` yazıyor. Hangi çıktılar mümkündür?

`a.join()`, `a` bitmeden `main`'i ilerletmez; `a` her zaman ilk harftir. `b` başlatıldıktan sonra `main`'in `m` yazması ile `b`'nin çalışması yarışır. Mümkün çıktılar `abm` ve `amb`'dir; `mab` olamaz. `b.join()` yalnız programın `b`'den önce bitmesini engeller. Tek bir JVM içinde 300'er denemelik beş turda toplam 1.500 denemenin hepsinde `amb` çıktı; ayrı JVM'lerle yapılan 40 çalıştırmanın hepsinde de `amb` görüldü. Yeni iş parçacığı başlayana kadar `main` çoğu zaman satırını çoktan yazmıştır. Yine de `abm` geçerli bir sıradır; hiçbir sıra garanti edilmez.

10.5 Yönetim ve durdurma

10.5.1 `sleep`, `join` ve `yield`

Bir iş parçacığının akışını düzenleyen üç temel metot vardır.

`sleep(ms)`
: Çalışan iş parçacığını belirli bir süre uyutur; iş parçacığı bu sürede `TIMED_WAITING` durumundadır. Uyuyan iş parçacığı tuttuğu kilitleri bırakmaz.

`join()`
: Çağıran iş parçacığı, hedef iş parçacığı bitene kadar bekler. `join(ms)` biçimi en çok `ms` milisaniye bekler; süre dolunca hedef bitmemiş olsa bile döner.

`yield()`
: Yürütmeyi geçici olarak başka bir iş parçacığına bırakma önerisidir. Çizelgeleyici bu öneriyi yok sayabilir; bu yüzden `yield()` ile hiçbir sıra garanti edilmez.

`join()`'in neden gerektiğini aşağıdaki program gösterir. `hesap` iş parçacığı 1'den 1.000.000'a kadar sayıları toplar ve sonucu `static` bir alana yazar.

```
public class JoinOrnegi {
    static long sonuc = 0;
    public static void main(String[] args)
            throws InterruptedException {
        Thread hesap = new Thread(() -> {
            long toplam = 0;
            for (int i = 1; i <= 1_000_000; i++) {
                toplam += i;
            }
            sonuc = toplam;
        });
        hesap.start();
        System.out.println("join öncesi : " + sonuc);
        hesap.join();   // hesap bitene kadar bekle
        System.out.println("join sonrası: " + sonuc);
    }
}
```

```
join öncesi : 0
join sonrası: 500000500000
```

`start()`'tan hemen sonra `main` `sonuc`'u okur. Hesap o sırada daha bitmemiştir; `main` başlangıç değerini, yani 0'ı görür. Program 30 kez çalıştırıldı ve ilk satır her seferinde 0 çıktı. Bu bir garanti değildir: yavaş başlayan bir `main` ya da çok hızlı bir hesap, ilk satırda da sonucu gösterebilirdi. `join()` sonrasında ise durum kesindir. Hesap bitmiştir ve yazdığı değer `main`'e görünür: $1 + 2 + \ldots + 1.000.000 = 1.000.000 \cdot 1.000.001 / 2 = 500.000.500.000$.

Sonucun `long` tutulması önemlidir. Toplam yaklaşık $5 \cdot 10^{11}$'dir; `int`'in üst sınırı olan 2.147.483.647'yi çok aşar.

`join()` ikinci bir güvence daha verir: hedef iş parçacığının bütün yazdıkları, `join()`'den dönen iş parçacığına görünür. `sonuc`'un `volatile` olmaması bu yüzden sorun değildir.

`sleep` ve `join`, kontrol edilen bir istisna olan `InterruptedException`'ı fırlatabilir. Bu yüzden çağrıldıkları yerde ya yakalanmaları ya da `throws InterruptedException` ile bildirilmeleri gerekir. Sunumdaki örneklerin çoğu `main`'i `throws InterruptedException` ile bildirir. Lambda içinde ise `throws` yazılamaz, çünkü `run()` hiçbir kontrol edilen istisna bildirmez; orada `try-catch` şarttır.

Sık yapılan hata

`Thread.sleep(100);` satırını `try-catch` olmadan, `throws` bildirmeyen bir metoda yazmak. Derleyici `unreported exception InterruptedException; must be caught or declared to be thrown` hatası verir. Çözüm `try-catch` ya da metot imzasına `throws InterruptedException` eklemektir.

10.5.2 Arka plan iş parçacıkları

Tanım 10.5

Arka plan iş parçacığı (*daemon thread*), JVM'in kapanmasını engellemeyen yardımcı iş parçacığıdır. JVM, arka plan iş parçacığı olmayan bütün iş parçacıkları bitince kapanır; arka plan iş parçacıklarının bitmesini beklemez.

```
public class DaemonOrnegi {
    public static void main(String[] args)
            throws InterruptedException {
        Thread saat = new Thread(() -> {
            int s = 0;
            while (true) {
                System.out.println("tik " + s++);
                try {
                    Thread.sleep(300);
                } catch (InterruptedException e) {
                    return;
                }
            }
        });
        saat.setDaemon(true);   // start()'tan ÖNCE
        saat.start();
        Thread.sleep(1000);
        System.out.println("main bitti, JVM kapanıyor");
    }
}
```

```
tik 0
tik 1
tik 2
tik 3
main bitti, JVM kapanıyor
```

`saat` sonsuz bir döngüdedir ve her 300 ms'de bir satır yazar. `setDaemon(true)` onu arka plan iş parçacığı yapar. `main` bir saniye uyur, son satırını yazar ve biter. Geriye yalnız arka plan iş parçacığı kaldığı için JVM kapanır; `saat`'in döngüsü yarıda kalır.

Kaç `tik` yazılır? `saat` 0, 300, 600 ve 900 ms civarında yazar; dördüncü uykusu 1200 ms'de bitecektir. `main` 1000 ms'de uyanır. Bu yüzden beklenen çıktı dört `tik`'tir. Program 10 kez çalıştırıldı ve hepsinde dört `tik` satırı yazıldı. Zamanlamaya dayalı her çıktı gibi bu da garanti değildir; 900 ms ile 1000 ms arasındaki pay küçüktür.

`setDaemon(true)`, `start()`'tan önce çağrılmalıdır. İş parçacığı başladıktan sonra çağrılırsa `IllegalThreadStateException` fırlar; bu davranış JDK 21 ile denendi. Arka plan iş parçacıkları kayıt tutma, önbellek temizleme ya da düzenli denetim gibi yardımcı işler içindir. Dosyaya yazan ya da yarıda kalmaması gereken bir iş arka plan iş parçacığına verilmez, çünkü JVM kapanırken `finally` blokları bile çalışmadan kesilebilir.

10.5.3 İş parçacığını durdurmak

Bir iş parçacığı, `run()` metodu tamamlandığında durur. JVM ise iki durumda kapanır: `System.exit()` (içeride `Runtime.exit()`) çağrıldığında ya da arka plan iş parçacığı olmayan bütün iş parçacıkları bittiğinde.

Bir iş parçacığını dışarıdan durdurmanın doğru yolu iş birliğidir. Durdurulacak iş parçacığı düzenli aralıklarla bir bayrağı denetler; bayrak kalkınca işini düzgünce bitirip `run()`'dan çıkar. Bayrak `volatile` tanımlanır, böylece başka bir iş parçacığının yaptığı değişiklik görünür olur. İkinci yol `interrupt()` ile kesme isteği göndermektir.

Eski Java sürümlerinde `Thread.stop()` adlı bir metot vardı. Bu metot iş parçacığını işinin ortasında öldürürdü. İş parçacığının tuttuğu kilitler birden bırakılır ve koruduğu nesneler yarım güncellenmiş hâlde kalırdı. Örneğin bir hesaptan para çekilmiş ama öteki hesaba yatırılmamış olabilirdi. Metot uzun süre kullanımdan kaldırılmış olarak kaldı; Java 20'den beri de çalışmaz:

```
Thread t = new Thread(() -> { });
try {
    t.stop();
} catch (UnsupportedOperationException e) {
    System.out.println("stop(): desteklenmiyor");
}
```

```
stop(): desteklenmiyor
```

JDK 21'de `stop()` her zaman `UnsupportedOperationException` fırlatır. Aynı denemede `suspend()` ve `resume()` da aynı istisnayı fırlattı.

10.5.4 Bayrakla durdurmak

```
class Worker extends Thread {
    private volatile boolean flag = false;
    public void finish() { flag = true; }
    public void run() {
        while (!flag) {
            System.out.println("çalışıyor...");
            try { Thread.sleep(100); }
            catch (InterruptedException e) { return; }
        }
        System.out.println("bayrak görüldü, bitiyor");
    }
}
public class BayrakDurdurma {
    public static void main(String[] args)
            throws InterruptedException {
        Worker w = new Worker();
        w.start();
        Thread.sleep(350);
        w.finish();   // bayrağı kaldır
        w.join();     // bitmesini bekle
        System.out.println("main: işçi durdu");
    }
}
```

```
çalışıyor...
çalışıyor...
çalışıyor...
çalışıyor...
bayrak görüldü, bitiyor
main: işçi durdu
```

`finish()` yalnız bayrağı kaldırır; iş parçacığını zorla durdurmaz. İşçi her turun başında `!flag` koşulunu denetler. Bayrağın kalktığını gördüğü ilk turda döngüden çıkar, son satırını yazar ve `run()` 'dan kendisi döner. `main` ise `join()` ile işçinin bitmesini bekler.

Zamanlama şöyledir: işçi 0, 100, 200 ve 300 ms civarında yazar. `main` 350 ms'de uyanıp bayrağı kaldırır. İşçi 400 ms'de uyandığında bayrağı görür. Program 15 kez çalıştırıldı ve hepsinde dört `çalışıyor...` satırı görüldü. Sayı zamanlamaya bağlıdır; bayrak yönteminin doğruluğu ise sayıya bağlı değildir.

Bayrak `volatile` olmasaydı ne olurdu? İşçi bayrağı ilk okuduğu değerle önbellekte ya da bir yazmaçta tutabilir ve değişikliği hiç görmeyebilirdi. Bu örnekte döngüdeki `sleep` çağrısı sorunu çoğu zaman gizler; boş bir döngüde ise iş parçacığı gerçekten takılabilir.

10.5.5 Referansla durdurmak

İş birliğiyle durdurmanın ikinci bir biçimi, bayrak yerine iş parçacığının kendisine referans tutmaktır.

```
class Worker implements Runnable {
    private volatile Thread thread;
    public void start() {
        thread = new Thread(this);
        thread.start();
    }
    public void finish() { thread = null; }
    public void run() {
        Thread ben = Thread.currentThread();
        while (thread == ben) {   // ... iş ...
        }
        System.out.println("run() bitti");
    }
}
public class ReferansDurdurma {
    public static void main(String[] args) throws Exception {
        Worker w = new Worker();
        w.start();
        Thread.sleep(200);
        w.finish();
        Thread.sleep(200);
        System.out.println("main: bitti");
    }
}
```

```
run() bitti
main: bitti
```

Sınıf `Runnable`'ı uygular ve çalışan `Thread` nesnesini `volatile` bir alanda tutar. Döngü, alan hâlâ kendi iş parçacığını gösterdikçe sürer. `Thread.currentThread()` ile karşılaştırma bunu denetler. `finish()` alanı `null` yapar; koşul bozulur ve `run()` biter. `main` 200 ms sonra durdurur, 200 ms daha bekler ve son satırını yazar. On çalıştırmada çıktı hep bu iki satırdı.

Bir ayrıntı: `currentThread()`, `Thread`'in `static` metodudur. `Thread`'i genişletmeyen bir sınıfta `Thread.` ön ekiyle yazılmak zorundadır. Ön ek unutulursa derleyici `cannot find symbol` hatası verir. Döngünün gövdesi boş olduğu için iş parçacığı bu 200 ms boyunca bir çekirdeği tam meşgul eder. Gerçek bir programda gövde bir iş yapar ya da kısa bir süre uyur.

10.6 Senkronizasyon

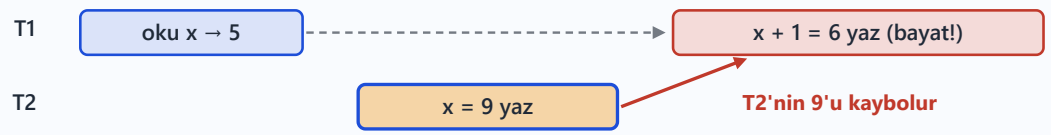
10.6.1 Sıra neden bozulur?

Birden çok iş parçacığı aynı veriyle çalıştığında işlemlerin doğru sırada yapılması üç nedenle güçtür.

Hız belirsizdir
: Her iş parçacığının çalışma hızı ve sıklığı değişkendir. İşletim sistemleri öncelikleri göz ardı edebilir; sıra için önceliğe güvenilmez.

Değer bayatlayabilir
: Bir iş parçacığı bir değişkeni okuduktan sonra başkası değeri güncelleyebilir. İlki güncel olmayan değerle çalışmayı sürdürür.

Komutlar yeniden sıralanabilir
: Derleyici, JIT (*just-in-time*, çalışma anında) derleyicisi ve işlemci, hız için komutların sırasını değiştirebilir. Tek iş parçacığında bu fark edilmez; birden fazlasında beklenmeyen davranış doğurur.



Şekil 10.12: Bayat değer: T1 okuduktan sonra T2 yazar; T1 eski değerle devam eder.

Paylaşılan ve değişen veriye birden fazla iş parçacığı erişiyorsa erişim senkronize edilmelidir. Aksi hâlde sonuç, iş parçacıklarının hangi sırayla çalıştığına bağlı olur.

Tanım 10.6

Yarış durumu (*race condition*), bir programın sonucunun iş parçacıklarının çalışma sırasına bağlı olmasıdır. **Senkronizasyon** (*synchronization*) ise iş parçacıklarının paylaşılan veriye erişimini, sonuç sıraya bağlı kalmayacak biçimde düzenlemektir.

10.6.2 Yarış durumu

En basit yarış örneği, iki iş parçacığının aynı sayacı artırmasıdır. Her iş parçacığı 100.000 kez `sayac++` yapar; beklenen sonuç 200.000'dir.

```
public class YarisDurumu {
    static int sayac = 0;   // paylaşılan veri
    public static void main(String[] args)
            throws InterruptedException {
        Runnable arttir = () -> {
            for (int i = 0; i < 100_000; i++) {
                sayac++;   // oku, 1 ekle, yaz: atomik DEĞİL
            }
        };
        Thread t1 = new Thread(arttir);
        Thread t2 = new Thread(arttir);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Beklenen: 200000");
        System.out.println("Bulunan : " + sayac);
    }
}
```

Sunumdaki üç çalıştırma 138001, 135352 ve 136776 buldu. Not için program JDK 21 ile 20 kez daha çalıştırıldı. Bulunan değerler:

```
136210, 145954, 155340, 149684, 132748, 125554, 126412, 156356, 126897, 144040, 137525,
127748, 117573, 164660, 129167, 134217, 140315, 129857, 128366, 133715
(en küçük 117573, en büyük 164660)
```

Her çalıştırma farklı bir sonuç verdi ve hiçbiri 200.000'e ulaşmadı. Artışların bir kısmı kaybolur. Kaybın miktarı tamamen çalıştırmaya göre değişir; bilgisayarın çekirdek sayısına, o anki yüke ve JIT'in kodu ne zaman derlediğine bağlıdır. Az sayıda artış yapan bir programda (örneğin 1.000) sonuç çoğu zaman doğru çıkar, çünkü iş parçacıkları hiç çakışmadan bitebilir. Hata bu yüzden rastgele görünür ve test ile yakalanması zordur.

10.6.3 Kod izleme: `sayac++` neden kaybolur?

`sayac++` kaynak kodda tek bir işlem gibi görünür. İşlemcide ise üç adımdır: değeri bellekten oku, bir ekle, belleğe yaz.

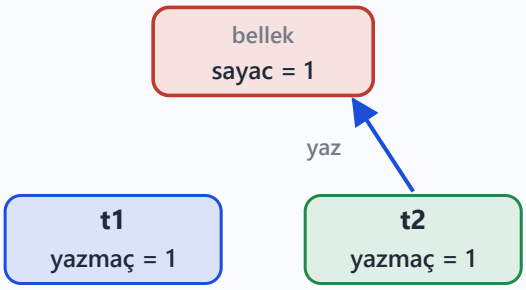
```
// sayac++ işlemcide üç adımdır:
int yazmac = sayac;     // 1) oku
yazmac = yazmac + 1;    // 2) 1 ekle
sayac = yazmac;         // 3) yaz
```

İki iş parçacığı bu üç adımı aynı anda yaparsa adımlar birbirine karışabilir. Aşağıdaki izleme başlangıçta 0 olan sayacın iki artışla 1'de kaldığı bir sırayı gösterir.

Adım	Satır	Çağrı yığını	Değişkenler / durum
1	1: `// sayac++ işlemcide üç adımdır:`	—	sayac=0, t1.yazmac=—, t2.yazmac=—
2	2: `int yazmac = sayac; // 1) oku`	t1	sayac=0, t1.yazmac=0, t2.yazmac=—
3	2: `int yazmac = sayac; // 1) oku`	t2	sayac=0, t1.yazmac=0, t2.yazmac=0
4	3: `yazmac = yazmac + 1; // 2) 1 ekle`	t1	sayac=0, t1.yazmac=1, t2.yazmac=0
5	4: `sayac = yazmac; // 3) yaz`	t1	sayac=1, t1.yazmac=1, t2.yazmac=0
6	3: `yazmac = yazmac + 1; // 2) 1 ekle`	t2	sayac=1, t1.yazmac=1, t2.yazmac=1
7	4: `sayac = yazmac; // 3) yaz`	t2	sayac=1, t1.yazmac=1, t2.yazmac=1

Tablo 10.9: `sayac++` adımlarının bir olası karışması. "Çağrı yığını" sütunu o anda çalışan iş parçacığını gösterir.

2. adımda `t1` sayacı okur: yazmacına 0 gelir. Tam bu noktada bir bağlam değişimi olur. 3. adımda `t2` de sayacı okur; `t1` henüz yazmadığı için o da 0 görür. 4. ve 5. adımlarda `t1` kendi yazmacında 1 ekler ve belleğe 1 yazar.



Şekil 10.13: 7. adım. `t2` kendi bayat değerine 1 ekleyip yazdı. İki artış yapıldı, ama `sayac` 1'dir.

6. ve 7. adımlarda `t2` kendi yazmacındaki eski değere, yani 0'a 1 ekler ve 1 yazar. `t1`'in yazdığı 1'in üstüne yine 1 yazılmıştır. İki artış yapılmış, ama sonuç 1'dir: `t1`'in artışı kaybolmuştur (Şekil 10.13). Bu olaya **kayıp güncelleme** (*lost update*) denir.

Bu, olası sıralardan yalnızca biridir. Araya girme olmazsa, yani bir iş parçacığı üç adımını öteki başlamadan bitirirse sonuç doğru çıkar. 100.000'er artışlık programda bu karışma binlerce kez olur ve her seferinde bir artış kaybolur. İş Parçacığı Yarışı simülatörü bu adımları elle ya da rastgele sıralamakta ve kaybın nasıl oluştuğunu tek tek izlemekte kullanılabilir.

Örnek 10.4

Bir hesabın bakiyesi 500'dür. `T1` `yatir(100)`, `T2` `cek(30)` çağırır; metotlar `bakiye += m` ve `bakiye -= m` satırlarından ibarettir ve senkronize değildir. Sıra şöyle olsun: `T1` okur, `T2` okur, `T2` çıkarıp yazar, `T1` ekleyip yazar. Bakiye sonunda kaç olur, doğrusu kaç olmalıydı?

Doğru sonuç 500 + 100 − 30 = 570'tir. `T1` 500 okur. `T2` de 500 okur, 30 çıkarır ve 470 yazar. `T1` bayat 500'e 100 ekler ve 600 yazar. Sonuç 600'dür: `T2` 'nin para çekmesi kaybolmuştur. Son iki yazma ters sırada olsaydı sonuç 470 olurdu ve bu kez yatırılan para kaybolurdu. Çözüm, iki metodu da aynı nesnenin kilidiyle (`synchronized`) korumaktır.

10.6.4 Senkronizasyon seçenekleri

Java iş parçacıklarını eşgüdümlemek için birçok araç sunar. Bunların her biri farklı bir soruna çözümdür:

- `join()` : bir iş parçacığının bitmesini beklemek.
- `synchronized` : nesnenin monitör kilidiyle karşılıklı dışlama.
- `Lock` **nesneleri** (`ReentrantLock` , `ReadWriteLock`): açık kilitleme.
- **Bariyer, semafor, değiştirici, geri sayım mandalı**: `CyclicBarrier` , `Semaphore` , `Exchanger` , `CountDownLatch` .
- `volatile` **değişkenler**: okuma ve yazmaların bütün iş parçacıklarına görünmesi.
- **Atomik nesneler**: `AtomicInteger` gibi bölünemez işlemler.

Senkronizasyona benzeyen ama senkronizasyon olmayan araçlar da vardır: öncelikler, `sleep` ile uyutma, `yield` ile yol verme ve zamanlayıcılar. Bunlar sırayı garanti etmez. "Biraz bekleyeyim, öbürü bitirmiş olur" düşüncesiyle yazılan kod, bir gün öbürünün bitirmediği bir anda yanlış sonuç üretir.

10.6.5 `synchronized` ve monitör

Tanım 10.7

Monitör (*monitor*), her Java nesnesine bağlı, kilitlenip açılabilen iç kilit ve bekleme düzeneğidir. Monitörü aynı anda yalnızca bir iş parçacığı tutabilir. `synchronized` anahtar sözcüğü bir nesnenin monitörünü alır; blok bitince monitör otomatik olarak bırakılır.

Monitörün çalışması şöyledir. `t1` bir nesnenin monitörünü tutarken `t2` de aynı monitörü isterse `t2` bekler; durumu `BLOCKED` 'dır. Monitör bırakılınca bekleyenler yarışır ve yalnızca biri monitörü alır. Blok normal biterse de, bloktan bir istisnayla çıkılırsa da monitör bırakılır. Böylece **karşılıklı dışlama** (*mutual exclusion*) sağlanır ve veri bütünlüğü korunur.

Tanım 10.8

Kritik bölge (*critical section*), paylaşılan veriye erişen ve aynı anda yalnızca bir iş parçacığının çalıştırması gereken kod parçasıdır. **Karşılıklı dışlama**, kritik bölgede aynı anda yalnızca bir iş parçacığının bulunmasıdır.

Söz dizimi kısadır. Parantez içine monitörü kullanılacak nesne yazılır:

```
synchronized (o) {
    // o'nun monitörü bu blokta bizde
}
```

Yarış örneğindeki sayaç artışı böyle bir blokla korunur:

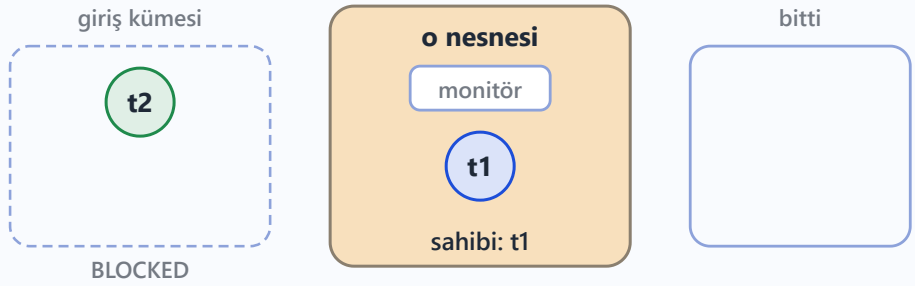
```
Runnable r = () -> {
    for (int i = 0; i < 50_000; i++) {
        synchronized (o) {   // o'nun monitörünü al
            ortak++;
        }                    // monitör bırakılır
    }
};
```

İki iş parçacığı bu görevi 50.000'er kez çalıştırdığında sonuç her seferinde 100000'dir. Program 10 kez çalıştırıldı ve hep bu değeri verdi. Her `ortak++`, monitör tutulurken yapılır; bir iş parçacığının oku–ekle–yaz adımları bitmeden öteki bloğa giremez.

Monitörün işleyişi adım adım şöyledir:

```
// t1 ve t2 aynı kodu çalıştırır
synchronized (o) {
    ortak++;      // kritik bölge
}
```

1. `t1` ve `t2` aynı anda `synchronized (o)` satırına gelir; monitör boştur.
2. `t1` kazanır ve monitörü alır. `t2` giriş kümesinde (*entry set*) `BLOCKED` bekler.
3. `t1` bloktan çıkar; monitör otomatik bırakılır. Bekleyenler yarışır.
4. `t2` monitörü alır ve kritik bölgeye girer.
5. `t2` de çıkar. `ortak++` iki kez, birbirine karışmadan yapılmıştır.



Şekil 10.14: 2. adım. `t1` monitörün sahibi; `t2` giriş kümesinde `BLOCKED` bekliyor.

Hangi iş parçacığının önce gireceği belirsizdir. Önemli olan, kritik bölgenin hiçbir zaman iki iş parçacığınca aynı anda çalıştırılmamasıdır. Monitör kilitlemesinin doğru çalışması için bütün iş parçacıklarının **aynı nesneyi** kilitlemesi gerekir. Her iş parçacığı kendi nesnesinin monitörünü alırsa hiçbir koruma oluşmaz.

10.6.6 Senkronize metotlar

`synchronized` bir metodun önüne de yazılabilir. Bu, metodun bütün gövdesini bir `synchronized` bloğa almakla aynıdır.

```
class C {
    synchronized void p() { ... }
    static synchronized void q() { ... }
    //...
}
```

Yukarıdaki tanımın eşdeğeri şudur:

```
class C {
    void p() { synchronized (this) { ... } }
    static void q() { synchronized (C.class) { ... } }
    //...
}
```

Hangi nesnenin kilitlendiği metodun türüne bağlıdır:

- **Örnek metot**: `this` nesnesinin monitörü kilitlenir.
- `static` **metot**: nesne olmadığı için sınıfın `C.class` nesnesinin monitörü kilitlenir.

Bu yüzden bir `static synchronized` metot ile bir örnek `synchronized` metot birbirini engellemez; farklı kilitler kullanırlar. Bu davranış JDK 21 ile denendi: başka bir iş parçacığı `C.class`'ın monitörünü 500 ms tutarken `x.p()` beklemeden çalıştı, `C.q()` ise monitör bırakılana kadar bekledi.

Sunumdaki kodda `...` yerine gövde yazıldığında (örneğin `n++`, `s++`) iki biçim de derlenir ve aynı davranır.

Blok biçiminin bir üstünlüğü vardır: yalnız gerçekten gereken satırlar kilitlenir. Uzun bir metodun yalnız iki satırı paylaşılan veriye dokunuyorsa bütün metodu kilitlemek, öteki iş parçacıklarını boşuna bekletir.

Örnek 10.5

`Sayac` sınıfı `Runnable`'ı uygular; `static int toplam` alanı vardır ve `run()` metodu `synchronized`'dır. `run()`, `toplam`'ı 100.000 kez artırır. İki iş parçacığı `new Thread(new Sayac())` ile iki ayrı nesneyi çalıştırıyor. Toplam neden 200.000'den küçük çıkabilir?

Her iş parçacığı kendi nesnesinin monitörünü (`this`) alır. İki farklı kilit birbirini engellemez; korunan veri ise ortak `static toplam`'dır. Program on kez çalıştırıldı ve bulunan toplamlar 111.143 ile 169.427 arasında değişti; hiçbiri 200.000 değildi. Düzeltme, herkesin aynı kilidi kullanmasıdır: döngüdeki artış `synchronized (Sayac.class)` bloğuna alınır. Düzeltilmiş program beş çalıştırmada da 200000 verdi. `AtomicInteger` da bir çözümdür.

10.6.7 Yarışı önlemek

Yarış örneğindeki sayaç, metotları senkronize bir sınıfın içine taşınarak düzeltilir.

```
class Sayac {
    private int deger = 0;
    public synchronized void arttir() { deger++; }
    public synchronized int getDeger() { return deger; }
}

public class SenkronSayac {
    public static void main(String[] args)
            throws InterruptedException {
        Sayac sayac = new Sayac();
        Runnable gorev = () -> {
            for (int i = 0; i < 100_000; i++) {
                sayac.arttir();
            }
        };
        Thread t1 = new Thread(gorev);
        Thread t2 = new Thread(gorev);
        t1.start();
        t2.start();
        t1.join();
        t2.join();
        System.out.println("Sonuç: " + sayac.getDeger());
    }
}
```

```
Sonuç: 200000
```

`arttir()` artık `synchronized`'dır: oku–ekle–yaz adımları bölünmeden yapılır. Program 10 kez çalıştırıldı ve her seferinde `Sonuç: 200000` yazdı. Okuyan `getDeger()` metodu da senkronizedir. Bu ilk bakışta gereksiz görünebilir, çünkü bir `int`'i okumak zaten tek adımdır. Ama okuma senkronize olmazsa okuyan iş parçacığının güncel değeri görmesi garanti değildir. Kilit bırakılırken yazılanlar, aynı kilidi sonradan alan iş parçacığına görünür hâle gelir.

Senkronizasyonun bir bedeli vardır. İş parçacıkları kritik bölgeye sırayla girer ve birbirini bekler; program biraz yavaşlar. Bu yüzden yalnız paylaşılan veri korunur. Bir iş parçacığının yalnız kendisinin kullandığı yerel değişkenler için kilide gerek yoktur.

10.6.8 Karşılıklı dışlama: öncelikli kuyruk

Senkronize metotlar, bir veri yapısını bütünüyle iş parçacığı güvenli hâle getirmenin en kısa yoludur. Aşağıdaki öncelikli kuyruk her çağrıda en küçük öğeyi verir. Bütün metotlar aynı monitörü, `this`'i kullanır.

```
class OncelikliKuyruk {
    private int[] veri;
    private int boyut;
    public OncelikliKuyruk(int n) { veri = new int[n]; }
    public synchronized void ekle(int o) {
        if (boyut == veri.length)
            throw new HeapFullException();
        veri[boyut++] = o;
    }
    public synchronized int cikar() {   // en küçüğü
        int enk = 0;
        for (int i = 1; i < boyut; i++) {
            if (veri[i] < veri[enk]) enk = i;
        }
        int sonuc = veri[enk];
        veri[enk] = veri[--boyut];
        return sonuc;
    }
    public synchronized String toString() {
        return Arrays.toString(Arrays.copyOf(veri, boyut));
    }
}
```

`HeapFullException` sunumda tanımlanmamıştır; burada `RuntimeException`'dan türeyen ve `"Kuyruk dolu"` iletisini taşıyan bir sınıf olarak yazıldı. Kuyruk tek bir iş parçacığından şöyle kullanılır:

```
OncelikliKuyruk k =
        new OncelikliKuyruk(3);
k.ekle(5);
k.ekle(2);
k.ekle(8);
System.out.println(k);
int x = k.cikar();
System.out.println(x);
System.out.println(k);
k.ekle(1);
try {
    k.ekle(9);
} catch (HeapFullException e) {
    String m = e.getMessage();
    System.out.println(m);
}
```

```
[5, 2, 8]
2
[5, 8]
Kuyruk dolu
```

5, 2 ve 8 eklenince dizi `[5, 2, 8]` olur. `cikar()` en küçüğün yerini doğrusal aramayla bulur: 1. konumdaki 2. Bu öğe döndürülür, yerine son öğe (8) taşınır ve boyut bir azalır: `[5, 8]`. 1 eklenince kuyruk yeniden dolar. `ekle(9)` istisna fırlatır ve ileti yazdırılır.

Bu sınıfta senkronizasyon neden şarttır? `ekle`'deki `veri[boyut++] = o` satırı da oku–değiştir–yaz içerir. İki iş parçacığı aynı anda eklerse ikisi de aynı `boyut`'u okuyup aynı göze yazabilir; bir öğe kaybolur. `cikar`'ın ortasında bir `ekle` araya girerse en küçük öğenin yeri değişebilir. Bütün metotlar aynı monitörü kullandığından bir metot çalışırken öteki hiçbir metot başlayamaz. `toString()`'in de senkronize olması, yarım değişmiş bir diziyi yazdırmayı önler.

10.7 Görünürlük

10.7.1 `volatile` alanlar

Yarış durumu, iki iş parçacığının aynı anda yazmasından doğar. Daha sinsi bir sorun da vardır: bir iş parçacığının yazdığı değeri öteki hiç görmeyebilir. İşlemciler hız için değerleri çekirdeğe yakın önbelleklerde ya da yazmaçlarda (*register*) tutar. Derleyici de bir döngüde değişmediğini düşündüğü bir değişkeni bir kez okuyup yazmaçta tutabilir. Bu durumda başka bir çekirdekteki iş parçacığı ana belleğe yeni değeri yazsa bile ilk iş parçacığı eski değerle çalışmayı sürdürür.

Tanım 10.9

`volatile` anahtar sözcüğüyle tanımlanan bir alan, her okunuşta ana bellekten okunur ve her yazılışta hemen bütün iş parçacıklarına görünür olur. `volatile` görünürlüğü garanti eder; birden çok adımlı işlemleri atomik yapmaz.



Şekil 10.15: `main` yeni değeri ana belleğe yazar; `t` iş parçacığı değeri yazmacında eski hâliyle tuttuğu için değişikliği görmez.

Java dilinin kuralına göre `long` ve `double` dışındaki bütün değişkenlerin tek bir okunması ya da yazılması zaten atomiktir. 64 bitlik `long` ve `double` değerleri ise bazı JVM'lerde iki 32 bitlik yarım olarak yazılabilir. `volatile long` ve `volatile double` alanlarda bu yarım yazma olmaz; okuma ve yazma atomik hâle gelir.

Paylaşılan basit bir bayrak için `volatile` yeterlidir; `synchronized` ya da kilit gerekmez. Ama yalnız tek bir okuma ya da tek bir yazma atomiktir. `x++` gibi oku–değiştir–yaz işlemleri `volatile` alanda da atomik değildir.

10.7.2 Görünürlük sorunu

```
public class GorunurlukSorunu {
    static boolean dur = false;   // volatile DEĞİL!
    public static void main(String[] args)
            throws InterruptedException {
        Thread t = new Thread(() -> {
            long tur = 0;
            while (!dur) {
                tur++;
            }
            System.out.println("iş parçacığı durdu");
        });
        t.setDaemon(true);
        t.start();
        Thread.sleep(500);
        dur = true;
        System.out.println("main: dur = true");
        t.join(2000);   // en çok 2 sn bekle
        boolean canli = t.isAlive();
        System.out.println("hâlâ çalışıyor mu? " + canli);
    }
}
```

İş parçacığı boş bir döngüde `dur`'u denetler. `main` 500 ms sonra `dur`'u `true` yapar ve en çok iki saniye bekler. `volatile` olmadan sunumdaki çıktı şudur:

```
main: dur = true
hâlâ çalışıyor mu? true
```

İş parçacığı değişikliği hiç görmemiş ve iki saniye sonra hâlâ çalışıyor. Nedeni JIT derleyicisidir. Döngü birkaç bin kez döndükten sonra JIT onu makine koduna çevirir. Döngünün içinde `dur`'u değiştiren hiçbir şey olmadığı için `dur` bir kez okunur ve döngü `while (true)` gibi çalışır. Bu dönüşüm tek iş parçacıklı bir program için tamamen doğrudur; Java kuralları da `volatile` olmayan bir alanda buna izin verir.

Program JDK 21 ile 10 kez çalıştırıldı ve iş parçacığı her seferinde değişikliği görmedi; çıktı sunumdaki gibiydi. Alan `static volatile boolean dur` olarak tanımlandığında ise on çalıştırmanın hepsinde iş parçacığı durdu:

```
main: dur = true
iş parçacığı durdu
hâlâ çalışıyor mu? false
```

Bu çalıştırmalardan birinde `iş parçacığı durdu` satırı `main: dur = true`'dan önce yazıldı. Bu bir çelişki değildir: `main` önce alanı değiştirir, sonra yazar; iş parçacığı değişikliği bu iki adımın arasında görüp kendi satırını önce yazabilir.

Sonuç JVM'e, işlemciye ve JIT'in ne zaman devreye girdiğine göre değişebilir. Bir bilgisayarda sorun hiç görünmeyebilir, ötekinde her seferinde görünür. Doğruluk bu tür şansa bırakılmaz: başka bir iş parçacığının değiştirdiği bir bayrak her zaman `volatile` tanımlanır. Örnekteki `setDaemon(true)` çağrısının amacı da budur: takılan iş parçacığı programın kapanmasını engellemesin.

10.7.3 `volatile`'ın sınırları

`volatile`, sayaç sorununu çözmez. Yarış örneğindeki sayaç `volatile` tanımlansa bile artışlar kaybolur:

```
static volatile int sayac = 0;
// main içinde:
Runnable r = () -> {
    for (int i = 0; i < 100_000; i++) sayac++;
};
```

Sunumdaki üç çalıştırma 163299, 164441 ve 199332 buldu. Not için yapılan 20 çalıştırmada bulunan değerler:

```
157591, 188201, 171134, 181949, 163135, 175082, 150206, 177784, 184817, 187816, 178180,
174558, 168901, 171061, 179658, 173836, 168144, 176917, 170584, 172321
(en küçük 150206, en büyük 188201)
```

Görünürlük sorunu çözülmüştür: her iş parçacığı her okumada güncel değeri görür. Ama iki iş parçacığı yine aynı değeri okuyup aynı sonucu yazabilir; okuma ile yazma arasına öteki iş parçacığı girebilir. Çözüm `synchronized` ya da `AtomicInteger`'dır.

`volatile`, `final` ile de birlikte kullanılamaz:

```
public class FinalVolatile {
    private final volatile int x = 5;
}
```

```
FinalVolatile.java:2: error: illegal combination of modifiers: final and volatile
    private final volatile int x = 5;
                              ^
1 error
```

İki anahtar sözcük aynı soruna farklı yollardan çözüm getirir. Tablo 10.10 farkı özetler.

`final`	`volatile`
Bir kez atanır (yapıcıda); sonra değişmez	Değeri değişebilir
Yapıcı bitince herkes aynı değeri görür	Her okuma güncel değeri döndürür

Tablo 10.10: `final` ve `volatile`.

`final` alan hiç değişmediği için görünürlük sorunu yoktur; Java, yapıcı bittikten sonra bütün iş parçacıklarının `final` alanın doğru değerini göreceğini garanti eder. `volatile` ise yalnız değişen alanlar için anlamlıdır. İkisini birlikte yazmak çelişkili olduğu için derleyici reddeder.

10.7.4 Java Bellek Modeli ve önce-olur ilişkisi

Bir iş parçacığının yazdığı değeri öteki ne zaman görür? Bu sorunun kesin cevabını Java Bellek Modeli (JMM, *Java Memory Model*) verir. Programcı işlemcinin önbelleklerini ve bellek bariyerlerini düşünmek zorunda kalmaz; bunun yerine önce-olur ilişkileriyle düşünür.

Tanım 10.10

A işlemi B işleminden **önce-olur** (*happens-before*) ise A'nın yazdığı her şey B'de kesinlikle görünür. Önce-olur ilişkisi geçişlidir: A, B'den ve B, C'den önce-olursa A da C'den önce-olur.

Kural	A önce-olur B
Kilit	Bir monitörün ya da kilidin bırakılması → aynı kilidin sonraki alınması
`volatile`	Bir `volatile` alana yazma → aynı alanın sonraki okunması
Başlatma	`t.start()` çağrısından önceki her şey → `t` 'nin içindeki işlemler
Bitirme	`t` 'nin bütün işlemleri → `t.join()` 'den dönen iş parçacığı
Hazır yapılar	`BlockingQueue.put` → aynı öğeyi alan `take` ; `Future` 'ın işi → `get()`

Tablo 10.11: Önce-olur ilişkisi kuran başlıca kurallar.

İş parçacıklı programlardaki görünürlük davranışı bu kurallarla açıklanır. `join()` örneğinde `sonuc` `volatile` değildi, ama `main` doğru değeri gördü: bitirme kuralı. Senkronize sayaçta `getDeger()` 'in de senkronize olması kilit kuralını işletir. Bayrak örneğinde `volatile` kuralı işler. Bu ilişkilerden biri kurulmamışsa iki iş parçacığı arasında görünürlük garantisi yoktur.

Görünürlükle karıştırılmaması gereken bir başarım sorunu da vardır: **yanlış paylaşım** (*false sharing*). Farklı iş parçacıklarının birbirinden bağımsız değişkenleri aynı önbellek satırında (çoğunlukla 64 bayt) bulunur ve sık güncellenirse çekirdekler bu satırı birbirinden sürekli çeker. Sonuç yanlış olmaz, ama program yavaşlar. Bu bir doğruluk değil, başarım sorunudur.

10.8 Açık kilitler ve kilitlenme

10.8.1 `Lock` arayüzü

`synchronized` basit ve güvenlidir, ama esnek değildir. Kilit beklenirken vazgeçilemez, bekleme süreyle sınırlanamaz, beklemekte olan iş parçacığı kesilemez. Karmaşık senkronizasyon senaryoları için `java.util.concurrent.locks` paketi açık kilitler sunar.

> **Tanım 10.11**
>
> **Açık kilit** (*explicit lock*), `Lock` arayüzünü uygulayan bir nesnedir. Kilit `lock()` ile alınır ve `unlock()` ile açılır; ikisini de programcı çağırır. `ReentrantLock` ve `ReentrantReadWriteLock` bu arayüzün başlıca gerçeklemeleridir.

Açık kilitler `synchronized` 'ın yapamadığı üç şeyi yapar: beklemeden denemek (`tryLock()`), süreli beklemek (`tryLock(süre)`) ve kesilebilir beklemek (`lockInterruptibly()`). Bunun karşılığında sorumluluk programcıdadır: kilidi açmak unutulmamalıdır. Kullanmak için `java.util.concurrent.locks.*` içe aktarılır.

```
public class Ornek {
    private final Lock kilit = new ReentrantLock();
    private int sayac = 0;
    public void calistir() {
        kilit.lock();
        try {
            // kritik bölge
            sayac++;
        } finally {
            kilit.unlock();
        }
    }
    public static void main(String[] args)
            throws InterruptedException {
        Ornek o = new Ornek();
        Runnable r = () -> {
            for (int i = 0; i < 100_000; i++) o.calistir();
        };
        Thread a = new Thread(r), b = new Thread(r);
        a.start(); b.start();
        a.join(); b.join();
        System.out.println("Sonuç: " + o.sayac);
    }
}
```

```
Sonuç: 200000
```

Program, yarış örneğinin açık kilitle yazılmış hâlidir. Her `sayac++` kilit tutulurken yapılır. Program 10 kez çalıştırıldı ve her seferinde `Sonuç: 200000` yazdı. Kilit nesnesi `final` bir alandadır; bütün iş parçacıkları aynı kilidi kullanır.

10.8.2 Kilidi `finally` bloğunda açmak

Açık kilidin doğru kullanım kalıbı her zaman aynıdır:

```
public void calistir() {
    Lock kilit = ...;
    kilit.lock();
    try {
        // korunan kaynağa erişim
    } finally {
        kilit.unlock();
    }
}
```

`lock()`, `try`'dan hemen önce çağrılır. `unlock()` ise `finally`'nin içindedir. Kritik bölgede bir istisna oluşsa bile `finally` çalışır ve kilit bırakılır. `lock()`'un `try`'ın içine değil, önüne yazılmasının da nedeni vardır. `lock()` başarısız olursa `finally`'nin alınmamış bir kilidi açmaya çalışmasını istemeyiz.

```
kilit.lock();
islemYap();      // istisna fırlatırsa...
kilit.unlock();  // ...bu satıra hiç gelinmez
```

Bu yanlış biçimde `islemYap()` istisna fırlatırsa `unlock()` satırına hiç gelinmez. Kilit kilitli kalır ve onu isteyen her iş parçacığı sonsuza dek bekler. `synchronized`'da bu sorun yoktur: bloktan istisnayla çıkılsa da monitör kendiliğinden bırakılır.

Örnek 10.6

`bol(a, b)` metodu kilidi alıyor, `a / b`'yi yazıyor ve kilidi açıyor; `finally` yok. `t1` `bol(10, 0)` çağırıyor; `t1` bittikten sonra `t2` `bol(10, 2)` çağırıyor. Ekranda ne görülür, `t2` hangi durumda kalır?

`t1`'de `10 / 0` `ArithmeticException` fırlatır. `t1` sonlanır, ama `unlock()`'a gelinmediği için kilit kilitli kalır. Ekranda yalnız `t1`'in yığın izi görünür; `5` hiç yazılmaz. `t2` `lock()`'ta sonsuza dek bekler. Deneme programı `t2`'yi arka plan iş parçacığı yaparak ve 300 ms sonra durumunu yazdırarak asılı kalmadan bitirildi; üç çalıştırmada da durum `WAITING` çıktı. `Lock` beklemesi bir monitör beklemesi olmadığı için durum `BLOCKED` değil, `WAITING`'dir. `unlock()` `finally`'ye alınınca `t2` `5` yazıp bitti.

İkinci bir hata da alınmamış bir kilidi açmaktır:

```
Lock kilit = new ReentrantLock();
kilit.unlock();   // hiç alınmamış kilit
```

```
Exception in thread "main" java.lang.IllegalMonitorStateException
    at
java.base/java.util.concurrent.locks.ReentrantLock$Sync.tryRelease(ReentrantLock.java:175)
    at
java.base/java.util.concurrent.locks.AbstractQueuedSynchronizer.release(AbstractQueuedSync
hronizer.java:1059)
    at java.base/java.util.concurrent.locks.ReentrantLock.unlock(ReentrantLock.java:494)
    at AcmaHatasi.main(AcmaHatasi.java:5)
```

`ReentrantLock`, kilidin sahibini tutar. Sahibi olmayan bir iş parçacığı `unlock()` çağırırsa `IllegalMonitorStateException` fırlar. JDK 21'in yazdığı tam iletide istisnanın oluştuğu yer de görünür.

10.8.3 `tryLock()`

`tryLock()` beklemeden kilit almayı dener. Kilit boşsa alır ve `true` döndürür; doluysa hiç beklemeden `false` döndürür. `tryLock(3, TimeUnit.SECONDS)` biçimi en çok üç saniye bekler. Süreli bekleme kesilebildiği için `InterruptedException` bildirilmelidir.

```
static void calistir()
        throws InterruptedException {
    if (kilit.tryLock()) {
        // alındı
        try { System.out.println("1: alındı"); }
        finally { kilit.unlock(); }
    } else {
        // alınamadı, başka bir şey yap
        System.out.println("1: alınamadı");
    }
    if (kilit.tryLock(3, TimeUnit.SECONDS)) {
        try { System.out.println("2: alındı"); }
        finally { kilit.unlock(); }
    } else {
        // 3 saniye içinde alınamadı
        System.out.println("2: alınamadı");
    }
}
```

Metot şöyle sınanır: `main` kilidi alır, yeni bir iş parçacığında `calistir()`'ı başlatır ve bir saniye sonra kilidi bırakır.

```
kilit.lock();   // kilidi main tutuyor
Thread t = new Thread(() -> {
    try { calistir(); }
    catch (InterruptedException e) { }
});
t.start();
Thread.sleep(1000);
System.out.println("main: kilidi bırakıyor");
kilit.unlock();
t.join();
```

```
1: alınamadı
main: kilidi bırakıyor
2: alındı
```

İlk denemede kilit `main`'dedir; `tryLock()` beklemeden `false` döndürür ve `1: alınamadı` yazılır. İş parçacığı ardından en çok üç saniye beklemeye başlar. Bir saniye sonra `main` kilidi bırakır; bekleyen iş parçacığı kilidi hemen alır ve `2: alındı` yazar. Çıktı beş çalıştırmada da aynıydı. `main` kilidi hiç bırakmasaydı ikinci deneme üç saniye sonra `false` dönerdi; bu durum ayrıca denendi ve `2: alınamadı (3 sn)` görüldü.

10.8.4 `ReentrantLock` ve yeniden giriş

`ReentrantLock`'un adı "yeniden girişe izin veren" anlamındadır. Kilidi tutan iş parçacığı aynı kilidi bir kez daha isterse beklemez; yalnız bir sayaç artar. Kilit, sayaç 0'a inince gerçekten bırakılır.

Tanım 10.12

Yeniden girişli kilit (*reentrant lock*), kendisini tutan iş parçacığının aynı kilidi beklemeden yeniden almasına izin veren kilittir. Kilit bir sahip ve bir **tutma sayısı** (*hold count*) saklar; her `lock()` sayıyı bir artırır, her `unlock()` bir azaltır.

Birden fazla kilitle iç içe kritik bölgelere de girilebilir:

```
ReentrantLock kilit1 = new ReentrantLock();
// true: adil (fair) kilit
ReentrantLock kilit2 = new ReentrantLock(true);
// ...
kilit1.lock();
try {
    // kritik bölgeye erişim
    kilit2.lock();
    try {
        // iç içe kritik bölgeye erişim
        System.out.println("iki kilit de alındı");
    } finally {  kilit2.unlock();  }
} finally {  kilit1.unlock();  }
System.out.println("adil mi? " + kilit2.isFair());
```

```
iki kilit de alındı
adil mi? true
```

`new ReentrantLock(true)` bir **adil kilit** (*fair lock*) oluşturur: kilit boşalınca en uzun bekleyen iş parçacığına verilir. Parametresiz yapıcı adil olmayan kilit verir; `kilit1.isFair()` `false` döndürür. Adil olmayan kilit daha hızlıdır, ama bir iş parçacığının uzun süre bekleyip durmasına izin verebilir. İki kilit iç içe alınırken bırakma sırası, alma sırasının tersidir: önce `kilit2`, sonra `kilit1`.

`synchronized` da yeniden girişlidir. `synchronized` bir metot, aynı nesnenin başka bir `synchronized` metodunu çağırabilir; iş parçacığı kendi tuttuğu monitörü beklemez. Yeniden giriş olmasaydı böyle bir çağrı iş parçacığını kendi kendisiyle kilitlerdi.

Tutma sayısının nasıl değiştiğini aşağıdaki program gösterir.

```
import java.util.concurrent.locks.ReentrantLock;

public class YenidenGiris {
    static final ReentrantLock kilit =
            new ReentrantLock();
    static void yaz(String yer) {
        System.out.println(yer + kilit.getHoldCount());
    }
    static void dis() {
        kilit.lock();
        try { yaz("dis(): "); ic(); }
        finally { kilit.unlock(); }
    }
    static void ic() {
        kilit.lock();   // beklemeden girer
        try { yaz("ic():  "); }
        finally { kilit.unlock(); }
    }
    public static void main(String[] args) {
        dis();
        yaz("main:  ");
    }
}
```

Adım	Satır	Çağrı yığını	Değişkenler / durum	Yeni çıktı
1	20: `dis();`	main()	sahibi=yok, sayı=0	—
2	10: `kilit.lock();`	main() → dis()	sahibi=main, sayı=1	—
3	11: `try { yaz("dis(): "); ic(); }`	main() → dis() → yaz()	sahibi=main, sayı=1	`dis(): 1`
4	15: `kilit.lock(); // beklemeden girer`	main() → dis() → ic()	sahibi=main, sayı=2	—
5	16: `try { yaz("ic(): "); }`	main() → dis() → ic() → yaz()	sahibi=main, sayı=2	`ic(): 2`
6	17: `finally { kilit.unlock(); }`	main() → dis() → ic()	sahibi=main, sayı=1	—
7	12: `finally { kilit.unlock(); }`	main() → dis()	sahibi=yok, sayı=0	—

Adım	Satır	Çağrı yığını	Değişkenler / durum	Yeni çıktı
8	21: `yaz("main: ");`	main() → yaz()	sahibi=yok, sayı=0	`main: 0`

Tablo 10.12: Yeniden giriş programının izlemesi.

2. adımda `dis()` kilidi alır: sahibi `main`, tutma sayısı 1. 4. adımda `ic()` aynı kilidi ister. Kilidin sahibi zaten `main` olduğu için beklemez; sayı 2 olur.



Şekil 10.16: 4. adım. Kilidin sahibi `main`, tutma sayısı 2.

6. adımda `ic()`'nin `finally`'si `unlock()` çağırır. Sayı 1'e iner; kilit hâlâ `main`'dedir ve başka bir iş parçacığı onu alamaz. 7. adımda `dis()`'in `finally`'si sayıyı 0'a indirir ve kilit gerçekten bırakılır. `main`'deki son `yaz` çağrısı bu yüzden 0 yazar. Çıktı tek iş parçacıklı olduğu için sabittir. Kural açıktır: her `lock()` için bir `unlock()` gerekir. Fazladan bir `unlock()` `IllegalMonitorStateException` fırlatır; eksik bir `unlock()` ise kilidi sonsuza dek kilitli bırakır.

10.8.5 `ReadWriteLock`

Birçok veri yapısı sık okunur, seyrek yazılır: bir ayar tablosu, bir önbellek, bir sözlük. Okumalar veriyi değiştirmediği için birbirini engellemelerine gerek yoktur. `ReadWriteLock` bu durumu iki ayrı kilitle karşılar.



Şekil 10.17: `ReadWriteLock`: okuyucular aynı anda okur; yazıcı yazarken kimse okuyamaz.

Okuma kilidi
: Birden fazla iş parçacığı veriyi aynı anda okuyabilir; okurken birbirini engellemez.

Yazma kilidi

Bir iş parçacığı yazarken diğerlerinin okuması da yazması da engellenir; yazma güvenli ve atomik olur.

```
private final Map<String, String> tablo =
        new HashMap<>();
private final ReadWriteLock kilit =
        new ReentrantReadWriteLock();

public String oku(String anahtar) {
    kilit.readLock().lock();   // okuma kilidi al
    try {
        return tablo.get(anahtar);   // veriyi oku
    } finally {
        kilit.readLock().unlock();
    }
}
public void yaz(String anahtar, String deger) {
    kilit.writeLock().lock();  // yazma kilidi al
    try {
        tablo.put(anahtar, deger);   // veriyi yaz
    } finally {
        kilit.writeLock().unlock();
    }
}
```

`ReentrantReadWriteLock`, `ReadWriteLock` arayüzünün gerçeklemesidir. `oku` okuma kilidini, `yaz` yazma kilidini kullanır. Sınıfı iki okuyucuyla deneyen kod şudur:

```
AyarTablosu a = new AyarTablosu();
a.yaz("dil", "tr");
Runnable okuyucu = () -> System.out.println(
        Thread.currentThread().getName()
        + ": " + a.oku("dil"));
Thread o1 = new Thread(okuyucu, "Okuyucu-1");
Thread o2 = new Thread(okuyucu, "Okuyucu-2");
o1.start(); o2.start();
o1.join(); o2.join();
```

```
Okuyucu-2: tr
Okuyucu-1: tr
```

İki okuyucu okuma kilidini aynı anda tutabilir. Bu ayrıca denendi: bir iş parçacığı okuma kilidini tutarken ikinci bir iş parçacığının `readLock().tryLock()` çağrısı `true`, `writeLock().tryLock()` çağrısı `false` döndü. Satırların sırası ise değişir: 40 çalıştırmada 22 kez `Okuyucu-1`, 18 kez `Okuyucu-2` önce yazdı.

10.8.6 Açık ve dolaylı kilitleme

	Açık: `Lock`	**Dolaylı:** `synchronized`
Denetim	Programcı `lock()` / `unlock()` çağırır	Otomatik: blok girişinde alınır, çıkışta bırakılır
Sınıflar	`Lock` arayüzü, `ReentrantLock`, `ReentrantReadWriteLock`	Anahtar sözcük; her nesnenin monitörü
Beklemeden deneme	`tryLock()`, `tryLock(süre)`	Yok: kilit boşalana kadar bekler
Kesilebilir bekleme	`lockInterruptibly()`	Yok
Adillik, koşullar	`new ReentrantLock(true)`, birden çok `Condition`	Yok; tek bekleme kümesi
Yeniden giriş	`ReentrantLock` yeniden girişlidir	Her zaman yeniden girişlidir
Risk	`unlock()` unutulursa kilit sonsuza dek kalır	Unutma riski yok
Ne zaman?	Gelişmiş denetim gerektiğinde	Basit senaryolar, daha okunur kod

Tablo 10.13: Açık (`Lock`) ve dolaylı (`synchronized`) kilitleme.

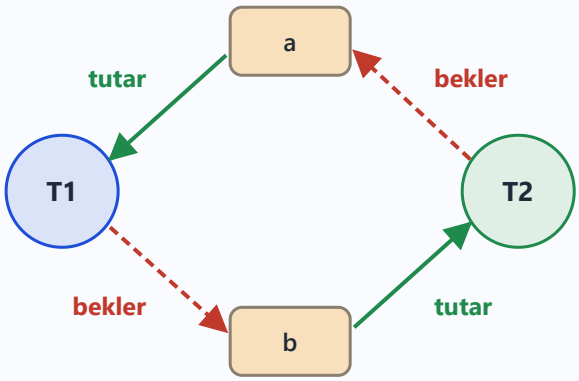
Tablo 10.13 seçimi kolaylaştırır. Varsayılan seçim `synchronized`'dır: kısa, okunur ve hataya kapalıdır. `Lock`, tablodaki ek yeteneklerden biri gerçekten gerektiğinde kullanılır.

10.8.7 Kilitlenme

Tanım 10.13

Kilitlenme (*deadlock*), iki ya da daha fazla iş parçacığının birbirinin tuttuğu kilidi sonsuza dek beklemesidir. Kilitler farklı iş parçacıklarınca farklı sırada alındığında oluşur.

```
static void calis(Object x, Object y, String ad) {
    synchronized (x) {        // 1. kilit
        uyu(100);
        synchronized (y) {    // 2. kilit
            System.out.println(ad + " bitti");
        }
    }
}
public static void main(String[] args)
        throws InterruptedException {
    Thread t1 = new Thread(() -> calis(a, b, "T1"));
    Thread t2 = new Thread(() -> calis(b, a, "T2"));
    t1.setDaemon(true);
    t2.setDaemon(true);
    t1.start();
    t2.start();
    Thread.sleep(500);
    String d = t1.getState() + " " + t2.getState();
    System.out.println(d);
    long[] k = ManagementFactory.getThreadMXBean()
            .findDeadlockedThreads();
    int n = (k == null) ? 0 : k.length;
    System.out.println("Kilitlenmiş: " + n);
}
```



Şekil 10.18: Kilitlenme döngüsü: T1 `a`'yı tutar ve `b`'yi bekler; T2 `b`'yi tutar ve `a`'yı bekler.

```
BLOCKED BLOCKED
Kilitlenmiş: 2
```

`calis(x, y, ad)`, önce `x`'in sonra `y`'nin monitörünü alır. T1 `calis(a, b)`, T2 ise ters sırayla `calis(b, a)` çağırır. `uyu(ms)`, `sleep`'i `try-catch` ile saran yardımcıdır; iki iş parçacığının da ilk kilidi alıp ikinciyi istemeden önce beklemesini sağlar. 500 ms sonra `main` iki iş parçacığının durumu-

nu yazar: ikisi de `BLOCKED`. `ThreadMXBean.findDeadlockedThreads()` kilitlenmiş iş parçacıklarının kimliklerini döndürür; sayı 2'dir.

İki iş parçacığı arka plan iş parçacığı yapılmıştır. Böylece `main` bitince JVM kapanır ve kilitlenme programı asılı bırakmaz. Program 10 kez çalıştırıldı ve her seferinde `BLOCKED BLOCKED` ve `Kilitlenmiş: 2` görüldü. 100 ms'lik uyku yüzünden iki iş parçacığı neredeyse her zaman ilk kilitlerini birlikte alır. Uyku olmasaydı kilitlenme yalnız bazı çalıştırmalarda oluşurdu.

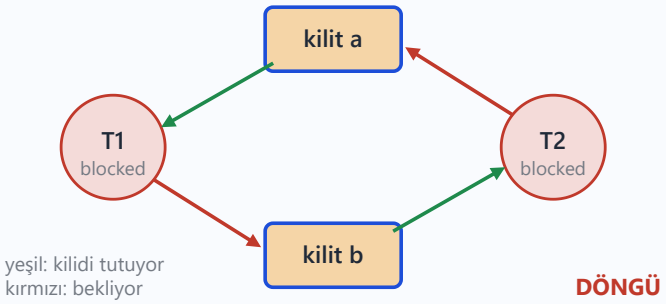
Kilitlenmenin nasıl kurulduğunu sunumdaki daha kısa sürüm üzerinde adım adım izleyelim. Bu sürümde iş parçacıkları arka plan iş parçacığı değildir; program gerçekten sonsuza dek bekler.

```
static final Object a = new Object(), b = new Object();
static void calis(Object x, Object y, String ad) {
    synchronized (x) {
        uyu(100);
        synchronized (y) {
            System.out.println(ad + " bitti");
        }
    }
}
public static void main(String[] args) {
    new Thread(() -> calis(a, b, "T1")).start();
    new Thread(() -> calis(b, a, "T2")).start();
}
```

Adım	Satır	Çağrı yığını	Değişkenler / durum
1	1: `static final Object a = new Object(), b = new Object();` 10: `public static void main(String[] args) {`	main()	a=Object, b=Object
2	11: `new Thread(() -> calis(a, b, "T1")).start();` 3: `synchronized (x) {`	T1: calis(a, b)	T1: x=a, y=b
3	12: `new Thread(() -> calis(b, a, "T2")).start();` 3: `synchronized (x) {`	T2: calis(b, a)	T2: x=b, y=a
4	4: `uyu(100);`	T1: uyu(100) → T2: uyu(100)	a=T1 tutuyor, b=T2 tutuyor
5	5: `synchronized (y) {`	T1: calis(a, b)	T1 istiyor=b
6	5: `synchronized (y) {`	T1: calis(a, b) → T2: calis(b, a)	T2 istiyor=a
7	6: `System.out.println(ad + " bitti");`	T1: calis(a, b) → T2: calis(b, a)	T1=BLOCKED, T2=BLOCKED
8	12: `new Thread(() -> calis(b, a, "T2")).start();`	T1: calis(a, b) → T2: calis(a, b)	T2 çağrısı=calis(a, b, "T2")

Tablo 10.14: Kilitlenmenin adım adım oluşması.

2. ve 3. adımlarda T1 `a`'yı, T2 `b`'yi alır. Şimdilik sorun yoktur. 4. adımda ikisi de 100 ms uyur; `sleep` tutulan kilidi bırakmaz. 5. adımda T1 uyanır ve iç blok için `b`'yi ister. `b` T2'dedir: T1 `BLOCKED` olur. 6. adımda T2 de `a`'yı ister; `a` T1'dedir. Her biri ötekinin bırakmasını bekler ve bir döngü oluşur.



Şekil 10.19: 6. adım. Kaynak ayırma çizgesinde döngü kapandı: T1 → `b` → T2 → `a` → T1.

`"bitti"` satırına hiçbiri ulaşamaz. Bu program not için 6 saniyelik zaman aşımıyla üç kez çalıştırıldı. Üçünde de hiçbir satır yazılmadı ve süreç zaman aşımında dışarıdan sonlandırıldı. Kilitlenme dışarıdan bir müdahale olmadan asla çözülmez; JVM bunu kendiliğinden fark edip düzeltmez.

10.8.8 Kilitlenmeyi önlemek

Kilitlenmenin en etkili ilacı tek bir değişikliktir: bütün iş parçacıkları kilitleri aynı sırayla alır.

```
Thread t1 = new Thread(() -> calis(a, b, "T1"));
Thread t2 = new Thread(() -> calis(a, b, "T2"));
```

```
T1 bitti
T2 bitti
TERMINATED TERMINATED
Kilitlenmiş: 0
```

Herkes önce `a`'yı, sonra `b`'yi alır. T2 `a`'yı bekler ve `b`'yi hiç tutmaz; döngü kurulamaz. T1 bitince kilitleri bırakır, T2 alır ve o da biter. Program 10 kez çalıştırıldı ve çıktı her seferinde bu dört satırdı.

Kilitlenmeyi önlemenin dört yolu şunlardır:

Sabit kilit sırası
: Bütün iş parçacıkları kilitleri aynı sırayla alır; örneğin nesnelerin kimlik numarasına göre.

Süreli deneme
: `tryLock(süre)` kilidi alamazsa iş parçacığı elindeki kilitleri bırakır ve daha sonra yeniden dener.

Kısa kritik bölge
: Bir kilit tutulurken başka kilit istenmez ve uzun işler yapılmaz.

Teşhis
: JDK'nın `jstack` aracı ya da `ThreadMXBean.findDeadlockedThreads()` kilitlenmiş iş parçacıklarını ve tuttukları kilitleri gösterir.

Sık yapılan hata

Kilitlenmenin "nasılsa olmuyor" diye göz ardı edilmesi. Uykusuz bir sürümde T1 `a` sonra `b`, T2 `b` sonra `a` alırsa çoğu çalıştırmada biri iki kilidi de alıp çıkar ve sorun görünmez. Not için iki iş parçacığı bu iki bloğu 1.000'er kez çalıştırdı ve deneme en çok 200 kez yinelendi. Üç ayrı çalıştırmanın üçünde de kilitlenme bu sınır dolmadan oluştu. Kilitlenmenin ne zaman ortaya çıkacağı çalıştırmaya göre değişir; olasılığı sıfır değilse er geç olur.

10.8.9 Canlı kilit ve açlık

Kilitlenme tek ilerleme sorunu değildir. İki akrabası daha vardır.



Şekil 10.20: Kilitlenme, canlı kilit ve açlık.

Canlı kilit (*livelock*)
: İş parçacıkları engellenmiş değildir; birbirine sürekli "yol vererek" durum değiştirirler, ama hiçbiri işini bitiremez. Dar bir koridorda karşılaşan iki kişinin aynı anda aynı yana kayması gibidir. Örnek: iki iş parçacığı `tryLock` başarısız olunca elindeki kilidi bırakır, aynı anda yeniden dener ve hep çakışır.

Açlık (*starvation*)
: Bir iş parçacığı kaynağa uzun süre ya da hiç erişemez: kilidi hep başkaları kapar ya da uzun kritik bölgeler onu bekletir. Öncelik (`setPriority`) değerleri bu sorunu çözmek için güvenilir değildir.

```
Lock adil = new ReentrantLock(true);  // adil
if (kilit.tryLock(50, TimeUnit.MILLISECONDS)) {
    // ... iş, sonra kilit.unlock()
} else {
    Thread.sleep(rnd.nextInt(20)); // rastgele
}
```

Çareler kısadır. Adil kilit, kilidi en uzun bekleyene verir; bunun bedeli biraz daha yavaş çalışmadır. Yeniden denemeden önce rastgele bir süre beklemek iki iş parçacığının simetrisini bozar ve canlı kilidi önler; kodda `rnd.nextInt(20)` bu işi görür. Kritik bölgeleri kısa tutmak da herkese sıra gelmesini sağlar.

10.9 Koşul senkronizasyonu

10.9.1 Koşul beklemek

Karşılıklı dışlama, iki iş parçacığının aynı anda kritik bölgeye girmesini engeller. Bazı durumlarda bu yetmez: iş parçacığı kritik bölgeye girdiğinde yapacağı iş için uygun koşul henüz oluşmamış olabilir. Paylaşılan bir kuyruğu düşünelim. Tüketici kuyruktan öğe alır, ama kuyruk boşsa alacak bir şey yoktur. Üretici öğe ekler, ama kuyruk doluysa yer yoktur.

> **Tanım 10.14**
>
> **Koşul senkronizasyonu** (*condition synchronization*), ortak bir kaynağa yalnızca belirli bir koşul sağlandığında erişilmesini sağlar. İş parçacığı koşul sağlanana kadar bekler; koşulu değiştiren iş parçacığı bekleyeni uyandırır.

Kuyruğu güvenle güncellemek için nesne kilidi gerekir. Ayrıca koşul sağlanana kadar beklemek gerekir. Ama kilidi tutarken beklenemez. Tüketici kilidi tutarak kuyruğa öğe gelmesini beklerse üretici kilidi hiç alamaz ve kuyruğa öğe ekleyemez; herkes takılır.

Akla ilk gelen çözüm, koşulu bir döngüde sürekli denetlemektir:

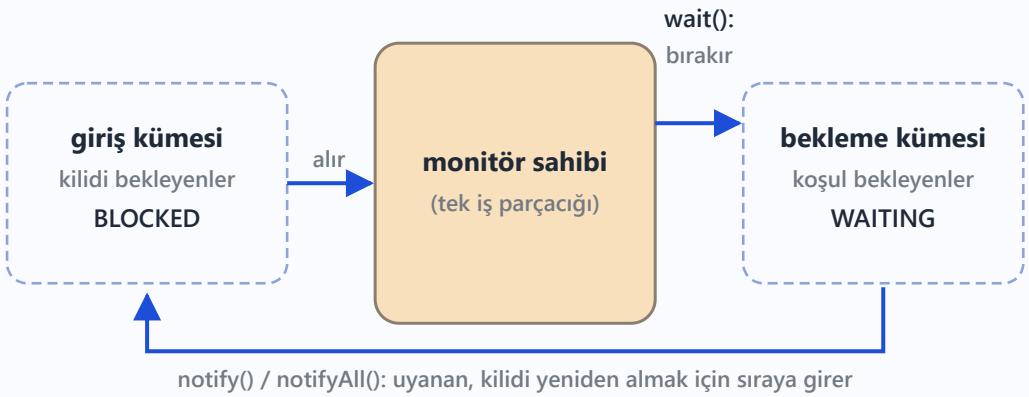
```
while (kuyruk.bosMu()) {
    // boşuna işlemci yakar; kilit tutuluyorsa
    // üretici hiç ekleyemez
}
```

Bu **meşgul bekleme** (*busy waiting*) iki yönden yanlıştır. Döngü boşuna bir çekirdeği tam meşgul eder. Kilit tutuluyorsa da üretici hiç ekleyemez ve döngü sonsuza dek sürer. Java'nın sunduğu üç doğru çözüm şunlardır:

1. `Object.wait()` ve `Object.notify()` / `notifyAll()`.
2. `Lock` ile birlikte kullanılan `Condition` arayüzü.
3. Senkronizasyon nesneleri (`CountDownLatch`, `Semaphore` …) ve hazır `BlockingQueue` sınıfları.

10.9.2 `wait()` ve `notify()`

Her nesnenin monitörüne bağlı iki küme vardır. Giriş kümesi, monitörü almak isteyen iş parçacıklarını tutar. **Bekleme kümesi** (*wait set*) ise bir koşulun değişmesini bekleyen iş parçacıklarını tutar.



Şekil 10.21: Monitör: giriş kümesi, sahip ve bekleme kümesi. `wait` ile bekleme kümesine geçilir, `notify` ile giriş kümesine dönülür.

- Kilidi tutan iş parçacığı `wait()` çağırır: bekleme kümesine girer, kilidi bırakır ve engellenir.
- `notify()` bekleme kümesinden bir iş parçacığını, `notifyAll()` hepsini uyandırır. `wait(ms)` biçimi süre dolunca da döner.
- Uyanan iş parçacığı önce kilidi yeniden almak zorundadır. Kilidi alınca `wait()` 'ten döner ve kaldığı yerden devam eder.

İki kural vardır. Birincisi, `wait` ve `notify` yalnızca aynı nesnenin `synchronized` bloğu ya da metodu içinde çağrılır. Aksi hâlde JDK 21 `java.lang.IllegalMonitorStateException: current thread is not owner` fırlatır. İkincisi, koşul `if` ile değil `while` ile denetlenir. Uyanan iş parçacığı kilidi yeniden alana kadar başka bir iş parçacığı koşulu yine bozmuş olabilir. Ayrıca Java, hiçbir `notify` olmadan uyanmaya, yani **sahte uyanmaya** (*spurious wakeup*) izin verir. `while` döngüsü her uyanışta koşulu yeniden denetler.

Örnek 10.7

`synchronized` bir blok içinde `kilit.wait()` ile `Thread.sleep(1000)` arasındaki fark nedir?

`wait()` monitörü bırakır ve bekleme kümesine girer; `notify` ya da `notifyAll` ile uyanınca kilidi yeniden alıp devam eder. `sleep()` ise kilitle ilgilenmez: kilidi tutarak uyur ve o süre boyunca kilidi isteyen herkes `BLOCKED` bekler. Koşul beklerken kilidi tutmak, koşulu değiştirecek iş parçacığını içeri sokmaz; bu yüzden koşul beklemesi için `wait()` kullanılır.

10.9.3 Engelleyen kuyruk

Koşul senkronizasyonunun klasik örneği **üretici–tüketici problemi**dir (*producer–consumer problem*). Üretici iş parçacıkları paylaşılan bir tampona öğe koyar, tüketici iş parçacıkları alır. Tampon doluysa üretici, boşsa tüketici bekler. Aşağıdaki `BlockingQueue` sınıfı bu tamponu sabit boyutlu bir diziyle yazar.

```
class BlockingQueue {
    private Object[] veri;
    private int bas = 0;
    private int son = 0;
    private int buyukluk = 0;
    public BlockingQueue(int boyut) {
        veri = new Object[boyut];
    }
    public synchronized int getSize() {
        return buyukluk;
    }
    public synchronized void add(Object oge)
            throws InterruptedException {
        while (buyukluk == veri.length) {
            wait();
        }
        veri[son] = oge;
        son = (son + 1) % veri.length;
        buyukluk++;
        notifyAll();
    }
```

```
    public synchronized Object remove()
            throws InterruptedException {
        while (buyukluk == 0) {
            wait();
        }
        Object oge = veri[bas];
        bas = (bas + 1) % veri.length;
        buyukluk--;
        notifyAll();
        return oge;
    }
}
```

Dizi dairesel (*circular*) kullanılır. `bas` çıkarılacak öğenin, `son` eklenecek öğenin konumudur. Her ekleme ve çıkarmada konum bir artar; `% veri.length` sona gelince başa sarar. `buyukluk` kuyruktaki öğe sayısıdır.

`add`, kuyruk doluyken (`buyukluk == veri.length`) bekler; `remove`, kuyruk boşken bekler. Her ekleme ya da çıkarmadan sonra `notifyAll()` bekleyenleri uyandırır. Burada `notifyAll()` bilinçli bir seçimdir. Aynı bekleme kümesinde hem üreticiler hem tüketiciler bekleyebilir. `notify()` yanlış türden birini uyandırırsa o iş parçacığı koşulu bozuk bulup yeniden bekler ve doğru iş parçacığı hiç uyanmayabilir.

10.9.4 Kod izleme: üretici ve tüketici

Kapasitesi 2 olan bir kuyrukta üreticinin 1, 2, 3 eklediği ve tüketicinin aldığı bir olası sırayı izleyelim. Satır numaraları aşağıdaki iki metodun numaralarıdır.

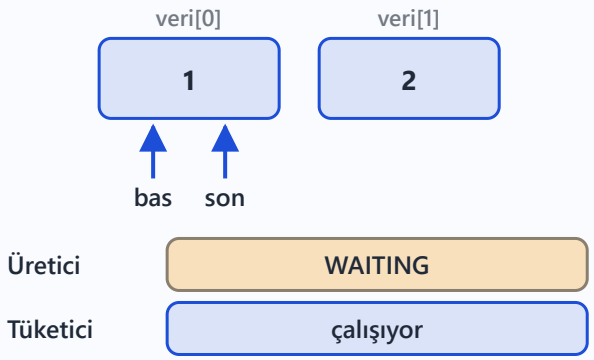
```
public synchronized void add(Object oge)
        throws InterruptedException {
    while (buyukluk == veri.length) {
        wait();
    }
    veri[son] = oge;
    son = (son + 1) % veri.length;
    buyukluk++;
    notifyAll();
}
public synchronized Object remove()
        throws InterruptedException {
    while (buyukluk == 0) {
        wait();
    }
    Object oge = veri[bas];
    bas = (bas + 1) % veri.length;
    buyukluk--;
    notifyAll();
    return oge;
}
```

Adım	Satır	Değişkenler / durum
1	6: `veri[son] = oge;` 7: `son = (son + 1) % veri.length;` 8: `buyukluk++;`	bas=0, son=1, buyukluk=1
2	6: `veri[son] = oge;` 7: `son = (son + 1) % veri.length;` 8: `buyukluk++;`	bas=0, son=0, buyukluk=2
3	4: `wait();`	bas=0, son=0, buyukluk=2
4	16: `Object oge = veri[bas];` 17: `bas = (bas + 1) % veri.length;` 18: `buyukluk--;`	bas=1, son=0, buyukluk=1
5	19: `notifyAll();`	bas=1, son=0, buyukluk=1

Adım	Satır	Değişkenler / durum
6	3: `while (buyukluk == veri.length) {` 6: `veri[son] = oge;` 7: `son = (son + 1) % veri.length;` 8: `buyukluk++;`	bas=1, son=1, buyukluk=2
7	16: `Object oge = veri[bas];` 17: `bas = (bas + 1) % veri.length;` 18: `buyukluk--;`	bas=0, son=1, buyukluk=1
8	16: `Object oge = veri[bas];` 17: `bas = (bas + 1) % veri.length;` 18: `buyukluk--;`	bas=1, son=1, buyukluk=0
9	14: `wait();`	bas=1, son=1, buyukluk=0

Tablo 10.15: Üretici–tüketici izlemesi (kapasite 2, bir olası sıra).

1. adımda `add(1)` yer bulur: 1, `veri[0]`'a yazılır ve `son` 1'e ilerler. 2. adımda `add(2)` 2'yi `veri[1]`'e yazar; `son` 2 % 2 = 0'a sarar. Kuyruk doludur. 3. adımda `add(3)` dolu kuyrukla karşılaşır ve `wait()` çağırır. Üretici bekleme kümesine girer ve kilidi bırakır.



Şekil 10.22: 3. adım. Kuyruk dolu; üretici `wait()` ile bekleme kümesinde ve kilit boşta.

Kilit boşaldığı için 4. adımda tüketici içeri girebilir. `remove()` 1'i alır ve `bas` 1 olur. 5. adımda `notifyAll()` üreticiyi uyandırır. Ama kilit hâlâ tüketicidedir; üretici kilidi bekler. `remove` 1'i döndürüp kilidi bırakınca 6. adımda üretici kilidi alır. `while` koşulunu yeniden denetler: kuyruk artık dolu değildir. 3, `veri[0]`'a yazılır. 7. ve 8. adımlarda tüketici 2'yi ve 3'ü alır; kuyruk boşalır. 9. adımda bir sonraki `remove()` boş kuyrukla karşılaşır ve tüketici bekler. Üretici 4'ü ekleyip `notifyAll()` yapınca uyanacaktır.

Bu, olası sıralardan biridir; gerçek sıra çizelgeleyiciye bağlıdır. Ama hiçbir sırada kuyruk taşmaz ya da boşken okunmaz. Bu güvenceyi `while` ile `wait` birlikte verir.

Sunumdaki deneme programı kuyruğu iki iş parçacığıyla kullanır. Tüketici her öğeden önce 100 ms uyur; böylece üretici kuyruğu doldurup beklemek zorunda kalır.

```
BlockingQueue kuyruk = new BlockingQueue(2);
Thread uretici = new Thread(() -> {
    try {
        for (int i = 1; i <= 4; i++) {
            kuyruk.add(i);
            System.out.println("üretildi: " + i);
        }
    } catch (InterruptedException e) { }
});
Thread tuketici = new Thread(() -> {
    try {
        for (int i = 1; i <= 4; i++) {
            Thread.sleep(100);
            Object x = kuyruk.remove();
            System.out.println("tüketildi: " + x);
        }
    } catch (InterruptedException e) { }
});
uretici.start();
tuketici.start();
uretici.join();
tuketici.join();
System.out.println("kalan: " + kuyruk.getSize());
```

```
üretildi: 1
üretildi: 2
üretildi: 3
tüketildi: 1
üretildi: 4
tüketildi: 2
tüketildi: 3
tüketildi: 4
kalan: 0
```

Sunumdaki bu çıktıda `üretildi: 3`, `tüketildi: 1`'den önce görünür. Bu ilk bakışta tuhaftır: 3, ancak 1 çıkarıldıktan sonra eklenebilir. Açıklama, `println`'ın kilidin dışında olmasıdır. Tüketici `remove`'dan döner ve tam yazdıracakken çizelgeleyici üreticiyi çalıştırır; üretici 3'ü ekleyip kendi satırını önce yazar. Program 40 kez çalıştırıldı; görülen sıralar:

```
20 kez:  üretildi: 1 · üretildi: 2 · üretildi: 3 · tüketildi: 1 · tüketildi: 2 · üretildi:
4 · tüketildi: 3 · tüketildi: 4
20 kez:  üretildi: 1 · üretildi: 2 · üretildi: 3 · tüketildi: 1 · üretildi: 4 · tüketildi:
2 · tüketildi: 3 · tüketildi: 4
```

Bütün çalıştırmalarda son satır `kalan: 0`'dı ve her öğe tam bir kez üretilip tüketildi. Satırların sırası ise çalıştırmaya göre değişti.

10.9.5 `Condition` arayüzü

`wait` ve `notify` monitör dünyasına aittir. `Lock` kullanan bir sınıfta aynı işi `Condition` arayüzü görür.

Tanım 10.15

`Condition`, bir `Lock`'a bağlı bekleme kümesidir ve `kilit.newCondition()` ile oluşturulur. `await()` kilidi bırakır ve iş parçacığını askıya alır; `signal()` bekleyen bir iş parçacığını, `signalAll()` hepsini uyandırır.

`Condition`'ın asıl üstünlüğü, bir kilide birden çok koşul bağlanabilmesidir. Üreticiler ve tüketiciler ayrı kümelerde bekler; her iş parçacığı yalnız ilgili olanı uyandırır.

```
class BlockingQueue {
    private final Lock kilit = new ReentrantLock();
    private final Condition doluDegil =
            kilit.newCondition();
    private final Condition bosDegil =
            kilit.newCondition();
    private Object[] veri;
    private int bas = 0;
    private int son = 0;
    private int buyukluk = 0;
    public BlockingQueue(int boyut) {
        veri = new Object[boyut];
    }
    // add() ve remove() metotları
}
```

```
public void add(Object oge)
        throws InterruptedException {
    kilit.lock();
    try {
        while (buyukluk == veri.length)
            doluDegil.await();
        veri[son] = oge;
        son = (son + 1) % veri.length;
        buyukluk++;
        bosDegil.signal();
    } finally {
        kilit.unlock();
    }
}
```

```
public Object remove()
        throws InterruptedException {
    kilit.lock();
    try {
        while (buyukluk == 0)
            bosDegil.await();
        Object oge = veri[bas];
        bas = (bas + 1) % veri.length;
        buyukluk--;
        doluDegil.signal();
        return oge;
    } finally {
        kilit.unlock();
    }
}
```

Ekleyen yalnız `bosDegil` koşulunu, çıkaran yalnız `doluDegil` koşulunu uyarır. `bosDegil`'de yalnız tüketiciler, `doluDegil`'de yalnız üreticiler bekler. Gereksiz uyandırma olmadığı için `signalAll()` yerine `signal()` yeterlidir. Kilit `finally`'de açılır; `await` da `wait` gibi bir `while` döngüsü içindedir.

Sunum, sınıfı bir üretici ve `main` ile dener: üretici 1'den 4'e kadar ekler, `main` dört kez çıkarıp aralarında boşlukla yazar.

```
1 2 3 4
```

Kuyruk ilk giren ilk çıkar (FIFO) ilkesiyle çalıştığı için bu çıktı sıra açısından her zaman aynıdır. Program on kez çalıştırıldı ve hep `1 2 3 4` yazdı.

10.9.6 Hazır çözüm: `ArrayBlockingQueue`

Kendi engelleyen kuyruğumuzu yazmak, düzeneği anlamak için yararlıdır. Gerçek projelerde ise `java.util.concurrent` paketindeki hazır `BlockingQueue` gerçeklemeleri kullanılır: `ArrayBlockingQueue`, `LinkedBlockingQueue` ve benzerleri.

```
BlockingQueue<Integer> kuyruk =
        new ArrayBlockingQueue<>(2);
Thread uretici = new Thread(() -> {
    try {
        // put(): kuyruk doluysa bekler
        for (int i = 1; i <= 4; i++) kuyruk.put(i);
    } catch (InterruptedException e) { }
});
uretici.start();
for (int i = 1; i <= 4; i++) {
    // take(): kuyruk boşsa bekler
    System.out.print(kuyruk.take() + " ");
}
System.out.println();
uretici.join();
```

```
1 2 3 4
```

`put()` kuyruk doluysa, `take()` boşsa bekler. Bekleme ve uyandırma sınıfın içinde yapılır; programcı kilit ya da koşul yazmaz. Sonuç on çalıştırmada da `1 2 3 4` idi. Tablo 10.16 üç yolu karşılaştırır.

Yol	Bekleme	Uyandırma
`synchronized`	`wait()`	`notify()`, `notifyAll()`
`Lock` + `Condition`	`await()`	`signal()`, `signalAll()`
`ArrayBlockingQueue`	`put()`, `take()` içinde	otomatik

Tablo 10.16: Koşul beklemenin üç yolu.

Örnek 10.8

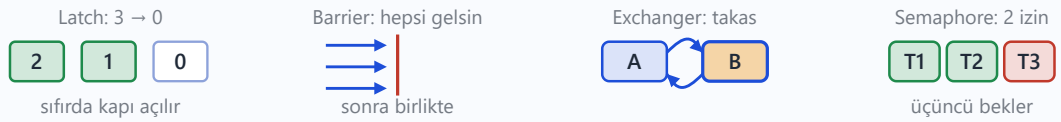
Kapasitesi 2 olan bir `ArrayBlockingQueue<Integer>` ile üretici iş parçacığı 1'den 5'e kadar sayıları, ardından bitiş işareti olarak −1'i koysun. `main` −1 gelene kadar alıp toplasın. Çıktı nedir?

Üretici `for (int i = 1; i <= 5; i++) q.put(i);` ve `q.put(-1);` çalıştırır. `main` bir döngüde `int v = q.take();` yapar, `v < 0` ise döngüden çıkar, değilse `toplam += v` yapar. Kuyruk FIFO olduğu için −1 her zaman en son gelir ve beş sayının hepsi toplanır: `Toplam: 15`. Kapasitenin 2 olması yalnız üreticinin daha sık beklemesine yol açar; sonucu değiştirmez. Program beş kez çalıştırıldı ve hep `Toplam: 15` yazdı.

10.10 Senkronizasyon nesneleri

10.10.1 Dört hazır araç

`java.util.concurrent` paketi, iş parçacıkları arasında sık görülen eşgüdüm biçimleri için hazır sınıflar sunar. Her biri belirli bir senaryoya çözümdür.



Şekil 10.23: `CountDownLatch`, `CyclicBarrier`, `Exchanger` ve `Semaphore`.

`CountDownLatch` (geri sayım mandalı)
: "N olay olsun, sonra devam et." Bekleyenler sayaç sıfırlanana kadar durur. Tek kullanımlıktır.

`CyclicBarrier` (döngüsel bariyer)
: "Hepimiz buluşma noktasına gelelim, sonra birlikte devam edelim." Yeniden kullanılabilir.

`Exchanger` (değiştirici)
: İki iş parçacığı bir noktada buluşur ve veri takası yapar.

`Semaphore` (semafor)
: N tane izin: aynı anda en çok N iş parçacığı kaynağa girebilir.

10.10.2 `CountDownLatch`

Geri sayım mandalı, belirli sayıda olayı bekleyen iş parçacıklarını eşgüdümler. Yapıcıya bir başlangıç sayısı verilir. Her `countDown()` sayacı bir azaltır. `await()` çağıran iş parçacığı sayaç 0 olana kadar bekler.

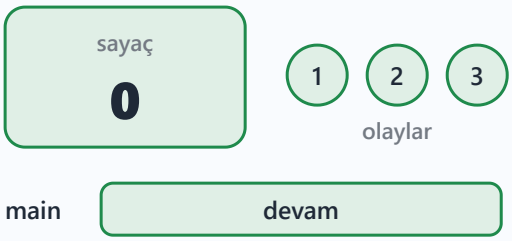
```
public static void main(String[] args)
        throws Exception {
    int sayac = 3;
    CountDownLatch mandal = new CountDownLatch(sayac);
    for (int i = 0; i < sayac; i++) {
        int no = i + 1;   // lambda için kopya
        new Thread(() -> {
            System.out.println(no + ". olay gerçekleşti");
            // Olay gerçekleştiğinde sayaç azaltılır.
            mandal.countDown();
        }).start();
    }
    // iş parçacıklarının tamamlanması beklenir.
    mandal.await();
    System.out.println("Tüm olaylar tamamlandı!");
}
```

Döngüdeki `int no = i + 1;` satırı önemlidir. Lambda içinden yalnız `final` ya da etkin olarak `final` (bir kez atanıp değişmeyen) yerel değişkenlere erişilebilir. Döngü değişkeni `i` her turda değiştiği için doğrudan kullanılamaz; `javac` şu hatayı verir: `local variables referenced from a lambda expression must be final or effectively final`. Her turda yeni oluşturulan `no` ise hiç değişmez.

Adım	Satır	Değişkenler / durum	Yeni çıktı
1	4: `CountDownLatch mandal = new CountDownLatch(sayac);`	sayaç=3	—
2	11: `}).start();`	sayaç=3	—
3	14: `mandal.await();`	sayaç=3	—
4	8: `System.out.println(no + ". olay gerçekleşti");` 10: `mandal.countDown();`	sayaç=2	`1. olay gerçekleşti`
5	8: `System.out.println(no + ". olay gerçekleşti");` 10: `mandal.countDown();`	sayaç=1	`3. olay gerçekleşti`
6	8: `System.out.println(no + ". olay gerçekleşti");` 10: `mandal.countDown();`	sayaç=0	`2. olay gerçekleşti`
7	15: `System.out.println("Tüm olaylar tamamlandı!");`	sayaç=0	`Tüm olaylar tamamlandı!`

Tablo 10.17: `CountDownLatch` programının bir çalıştırması (sunumdaki sıra).

Üç iş parçacığı başlatılır ve `main` `await()`'te bekler. Her iş parçacığı satırını yazar ve sayacı bir azaltır. Sunumdaki çalıştırmada sıra 1, 3, 2'dir. Son `countDown()` sayacı 0'a indirince mandal açılır ve `main` uyanır.



Şekil 10.24: 6. adım. Son olay gerçekleşti, sayaç 0; mandal açıldı ve `main` uyanıyor.

`Tüm olaylar tamamlandı!` satırı her zaman en sondadır; olay satırlarının sırası ise değişir. Program 60 kez çalıştırıldı; görülen sıralar:

```
13 kez:  1 · 3 · 2
13 kez:  1 · 2 · 3
12 kez:  2 · 3 · 1
 9 kez:  3 · 2 · 1
 7 kez:  3 · 1 · 2
 6 kez:  2 · 1 · 3
```

Mandal tek kullanımlıktır. Sayaç hiç artmaz ve yeniden kurulamaz. Sayaç 0'dayken `countDown()` çağrısı etkisizdir; sayaç eksiye inmez. Açılmış bir mandalda `await()` beklemeden döner.

10.10.3 `CyclicBarrier`

Döngüsel bariyer, bir grup iş parçacığını bir buluşma noktasında bekletir. Yapıcıya **parti** denen beklenecek iş parçacığı sayısı ve isteğe bağlı bir bariyer eylemi verilir.

```
public static void main(String[] args) {
    int parti = 3;
    Runnable eylem = () -> {
        System.out.println("Parti tamamlandı!");
    };
    CyclicBarrier bariyer =
            new CyclicBarrier(parti, eylem);
    for (int i = 0; i < parti; i++) {
        int no = i + 1;
        new Thread(() -> {
            try {
                System.out.println(no + " bekliyor");
                // iş parçacıklarını bekle
                bariyer.await();
                System.out.println(no + " devam ediyor");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

```
1 bekliyor
2 bekliyor
3 bekliyor
Parti tamamlandı!
3 devam ediyor
1 devam ediyor
2 devam ediyor
```

Her iş parçacığı `await()` çağırınca bekler. Partinin son üyesi de `await()` çağırınca bariyer eylemi bir kez çalışır, ardından herkes devam eder. Eylemi partiyi tamamlayan son iş parçacığı çalıştırır; bu ayrıca denendi. Bu çıktı da bir çalıştırmadır. `Parti tamamlandı!` satırı her zaman üç `bekliyor` satırından sonra, üç `devam ediyor` satırından öncedir. Her grubun kendi içindeki sırası ise değişir: 60 çalıştırmada 8 farklı çıktı görüldü.

Bariyerin "döngüsel" olması, parti dolup herkes geçtikten sonra aynı bariyerin yeniden kullanılabilmesidir. Bir oyunda her turun sonunda oyuncuların birbirini beklemesi bunun tipik örneğidir. Bekleyenlerden biri istisna alırsa, zaman aşımına uğrarsa ya da kesilirse bariyer kırılır; öteki bekleyenler `BrokenBarrierException` alır.

10.10.4 `Exchanger`

`Exchanger`, iki iş parçacığının veri alışverişi için kullanılır. Bir kilitten çok bir iletişim nesnesidir.

```
static void takas(Exchanger<String> ex, String veri, int ms) {
    try {
        System.out.println("gönderiliyor: " + veri);
        Thread.sleep(ms);
        String alinan = ex.exchange(veri);   // karşıyı bekle
        System.out.println(veri + " sahibi aldı: " + alinan);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}
public static void main(String[] args) {
    Exchanger<String> exchanger = new Exchanger<>();
    // İlk iş parçacığı "Veri 1" gönderir, 2 sn sonra
    new Thread(() -> takas(exchanger, "Veri 1", 2000)).start();
    // İkinci iş parçacığı "Veri 2" gönderir, 1 sn sonra
    new Thread(() -> takas(exchanger, "Veri 2", 1000)).start();
```

```
gönderiliyor: Veri 2
gönderiliyor: Veri 1
Veri 1 sahibi aldı: Veri 2
Veri 2 sahibi aldı: Veri 1
```

`exchange(veri)`, karşı taraf da `exchange` çağırana kadar bekler. `Veri 2`'nin sahibi bir saniye sonra `exchange`'e gelir ve bir saniye daha bekler. `Veri 1`'in sahibi iki saniyede gelince buluşurlar ve her biri ötekinin verisini alır. İlk iki satırın sırası iki iş parçacığının başlama sırasına bağlıdır; son iki satır da buluşma anında yarışır. On çalıştırmada 4 farklı çıktı görüldü.

10.10.5 `Semaphore`

Tanım 10.16

Semafor (*semaphore*), aynı anda en çok N iş parçacığının bir kaynağa erişmesine izin veren sayaçtır. `acquire()` bir izin alır; izin yoksa bekler. `release()` bir izni geri verir.

```
public static void main(String[] args) {
    int izin = 3;
    Semaphore semaphore = new Semaphore(izin);
    for (int i = 0; i < 5; i++) {
        int no = i + 1;
        new Thread(() -> {
            try {
                semaphore.acquire();   // izin al
                System.out.println(no + " girdi");
                Thread.sleep(2000);
                System.out.println(no + " çıkıyor");
                semaphore.release();   // izni bırak
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
        }).start();
    }
}
```

Adım	Satır	Değişkenler / durum	Yeni çıktı
1	3: `Semaphore semaphore = new Semaphore(izin);`	izin=3	—
2	16: `}).start();`	izin=3	—

3	8: `semaphore.acquire(); // izin al`
	9: `System.out.println(no + " girdi");`
		izin=2	`1 girdi`

4	8: `semaphore.acquire(); // izin al`
	9: `System.out.println(no + " girdi");`
		izin=1	`3 girdi`

5	8: `semaphore.acquire(); // izin al`
	9: `System.out.println(no + " girdi");`
		izin=0	`2 girdi`

6	8: `semaphore.acquire(); // izin al`	izin=0	—
7	11: `System.out.println(no + " çıkıyor");` 12: `semaphore.release(); // izni bırak`	izin=1	`2 çıkıyor`
8	11: `System.out.println(no + " çıkıyor");` 12: `semaphore.release(); // izni bırak`	izin=2	`1 çıkıyor`

9	8: `semaphore.acquire(); // izin al`
	9: `System.out.println(no + " girdi");`
		izin=0	`4 girdi` ⏎ `5 girdi`

10	11: `System.out.println(no + " çıkıyor");` 12: `semaphore.release(); // izni bırak`	izin=3	`3 çıkıyor` ⏎ `4 çıkıyor` ⏎ `5 çıkıyor`

Tablo 10.18: Üç izinli semafor ve beş iş parçacığı (sunumdaki çalıştırma).

Beş iş parçacığı başlatılır. İlk üçü birer izin alıp girer; sunumdaki çalıştırmada bunlar 1, 3 ve 2'dir. İzin kalmadığı için 4 ve 5 `acquire()` 'da bekler.



Şekil 10.25: 6. adım. Üç izin de kullanımda; 4 ve 5 `acquire()`'da bekliyor.

İki saniye sonra içerdekiler çıkar ve izinlerini bırakır. Bekleyen 4 ve 5 izinleri alıp girer ve iki saniye sonra onlar da çıkar. Sonunda üç izin geri dönmüştür. İçeride aynı anda hiçbir zaman üçten fazla iş parçacığı bulunmaz. Hangi iş parçacıklarının önce gireceği ve satırların sırası ise değişir: 12 çalıştırmada 12 farklı çıktı görüldü. Bütün çalıştırmalarda ilk üç satır üç `girdi` satırıydı ve dördüncü giriş ancak bir çıkıştan sonra geldi. İlk girenler her zaman 1, 2 ve 3 değildi; iki çalıştırmada 4 de ilk üçün arasındaydı. Çıktı satırları kilidin dışında yazıldığı için bazen bir `girdi` satırı, izni bırakan iş parçacığının `çıkıyor` satırından bile önce göründü.

`new Semaphore(n, true)` adil bir semafor oluşturur: izinler bekleme sırasına göre dağıtılır. Semaforun iki özelliği şaşırtıcıdır. Semafor izinlerin sahibini tutmaz; izni almamış bir iş parçacığı da `release()` çağırabilir. Başlangıç sayısı da bir üst sınır değildir: fazladan her `release()` izin sayısını artırır.

Örnek 10.9

Tek bir iş parçacığında şunlar çalışıyor: `Semaphore s = new Semaphore(2);`, `s.acquire();`, `x1 = s.tryAcquire();`, `x2 = s.tryAcquire();`, ardından üç kez `s.release();`. Sonra `CountDownLatch m = new CountDownLatch(2);` ve üç kez `m.countDown();`. `x1`, `x2`, `s.availablePermits()`, `m.getCount()` ve `m.await(10, TimeUnit.MILLISECONDS)` nedir?

Semafor 2 izinle başlar. `acquire()` bir izin alır: 1. `tryAcquire()` kalan izni alır ve `true` döner: 0. İkinci `tryAcquire()` izin bulamaz ve beklemeden `false` döner; `acquire()` olsaydı sonsuza dek beklerdi. Üç `release()` izni 3'e çıkarır, başlangıçtaki 2'yi aşar. Mandal iki `countDown()` ile 0'a iner; üçüncüsü etkisizdir. Açık mandalda `await` beklemeden `true` döner. Program `true false 3` ve `0 true` yazdı.

10.11 Diğer araçlar

10.11.1 Atomik sınıflar

Tek bir sayacı korumak için kilit kullanmak, sinek avlamak için top atmaya benzer. `java.util.concurrent.atomic` paketi kilitsiz ve iş parçacığı güvenli sayaçlar sunar.

Tanım 10.17

Atomik işlem (*atomic operation*), bölünmeden, tek adımda tamamlanan işlemdir; başka bir iş parçacığı işlemin ortasına giremez. `AtomicInteger`, `AtomicLong`, `AtomicBoolean` gibi sınıflar değerleri üzerinde atomik işlemler sunar.

```
public class AtomikSayac {
    public static void main(String[] args)
            throws InterruptedException {
        AtomicInteger sayac = new AtomicInteger(0);
        Runnable r = () -> {
            for (int i = 0; i < 100_000; i++) {
                sayac.incrementAndGet();   // atomik ++sayac
            }
        };
        Thread a = new Thread(r), b = new Thread(r);
        a.start(); b.start();
        a.join(); b.join();
        System.out.println("Sonuç: " + sayac.get());
        boolean b1 = sayac.compareAndSet(200000, 0);
        System.out.println(b1 + " " + sayac.get());
        boolean b2 = sayac.compareAndSet(5, 10);
        System.out.println(b2 + " " + sayac.get());
    }
}
```

```
Sonuç: 200000
true 0
false 0
```

`incrementAndGet()`, `++x`'in atomik karşılığıdır: değeri bir artırır ve yeni değeri döndürür. Program 10 kez çalıştırıldı ve her seferinde `Sonuç: 200000` buldu. Ardından iki `compareAndSet` çağrısı vardır. Değer 200000 olduğu için ilki tutar, 0 yazar ve `true` döner. İkincisi 5 bekler, ama değer 0'dır; hiçbir şey yazmaz ve `false` döner.

Tanım 10.18

Karşılaştır ve değiştir (*compare-and-set, CAS*): `compareAndSet(beklenen, yeni)`, değer hâlâ `beklenen` ise `yeni`'yi yazar ve `true` döndürür. Değer farklıysa ona dokunmaz ve `false` döndürür. Karşılaştırma ve yazma tek bir atomik işlemdir; işlemci bunu tek bir komutla yapar.

Kabaca, bir atomik nesne bir `volatile` alan ile atomik bir `compareAndSet` işleminin birleşimidir. `volatile` görünürlüğü, CAS ise bölünmezliği sağlar.

10.11.2 Kod izleme: `compareAndSet`

Kilit yoksa `incrementAndGet` nasıl güvenli olabilir? Aşağıdaki metot onun mantığını CAS ile yazar.

```
AtomicInteger sayac = new AtomicInteger(5);

int artir() {                  // incrementAndGet'in mantığı
    while (true) {
        int eski = sayac.get();
        int yeni = eski + 1;
        if (sayac.compareAndSet(eski, yeni))
            return yeni;
        // başarısız: araya biri girdi, tekrar dene
    }
}
```

Adım	Satır	Çağrı yığını	Değişkenler / durum
1	1: `AtomicInteger sayac = new AtomicInteger(5);`	—	sayac=5
2	5: `int eski = sayac.get();`	T1: artir()	T1.eski=5
3	5: `int eski = sayac.get();`	T1: artir() → T2: artir()	T1.eski=5, T2.eski=5
4	6: `int yeni = eski + 1;`	T1: artir() → T2: artir()	T1.yeni=6, T2.yeni=6
5	7: `if (sayac.compareAndSet(eski, yeni))` 8: `return yeni;`	T1: artir()	sayac=6, T1 → 6
6	7: `if (sayac.compareAndSet(eski, yeni))`	T2: artir()	sayac=6, T2 CAS(5, 6)=false
7	5: `int eski = sayac.get();`	T2: artir()	T2.eski=6
8	6: `int yeni = eski + 1;`	T2: artir()	T2.yeni=7
9	7: `if (sayac.compareAndSet(eski, yeni))` 8: `return yeni;`	T2: artir()	sayac=7

Tablo 10.19: İki iş parçacığının aynı anda `artir()` çağırması.

2. ve 3. adımlarda T1 ve T2 güncel değeri okur; ikisi de 5 görür. Kilit olmadığı için ikisi de çalışmayı sürdürür. 4. adımda ikisi de yerel olarak 6 hesaplar. Sıradan bir `sayac++` olsaydı ikisi de 6 yazar ve bir artış kaybolurdu.



Şekil 10.26: 6. adım. T1'in CAS'ı başarılı oldu, değer 6. T2'nin `compareAndSet(5, 6)` çağrısı başarısız: beklenen 5, değer 6.

5. adımda T1 `compareAndSet(5, 6)` çağırır. Değer hâlâ 5 olduğu için 6 yazılır ve `true` döner. 6. adımda T2 de `compareAndSet(5, 6)` çağırır. Değer artık 6'dır, beklenen 5 değildir; hiçbir şey yazılmaz ve `false` döner. T2 kaybolmaz, döngüye döner: 7. adımda güncel değeri, 6'yı yeniden okur ve 7 hesaplar. 9. adımda CAS bu kez başarılıdır: `sayac = 7`. İki artış da sayılmıştır. Kimse kilit beklemedi; yalnız T2 bir tur fazla döndü.

Bu metot iki iş parçacığıyla 100.000'er kez çağrıldığında sayaç 5 + 200.000 = 200.005'e ulaştı; beş çalıştırmada da sonuç aynıydı. Kilitsiz yöntemin üstünlüğü, iş parçacıklarının hiç uyumamasıdır. Çakışma az olduğunda bu kilitten daha hızlıdır; çok yoğun çakışmada ise iş parçacıkları boşuna tur atar.

10.11.3 Kesme

Bir iş parçacığını durdurmanın iş birliğine dayalı ikinci yolu kesmedir. `t.interrupt()`, `t`'ye bir kesme isteği gönderir; `t`'yi zorla durdurmaz.

> **Tanım 10.19**
>
> **Kesme** (*interrupt*), bir iş parçacığına "işini bırak" isteği göndermektir. `interrupt()` hedefin kesme bayrağını kurar. Hedef `sleep`, `wait` ya da `join` içinde bekliyorsa bekleme hemen kesilir ve `InterruptedException` fırlar; bu sırada bayrak temizlenir.

Bayrağı okumanın iki yolu vardır. `Thread.interrupted()` `static` bir metottur; çalışan iş parçacığının bayrağını okur ve temizler. `isInterrupted()` ise bir örnek metodudur ve bayrağı yalnız okur.

```
public class KesmeOrnegi {
    public static void main(String[] args)
            throws InterruptedException {
        Thread isci = new Thread(() -> {
            try {
                while (!Thread.interrupted()) {
                    // İş parçacığının çalışma mantığı
                    System.out.println("çalışıyor...");
                    Thread.sleep(400);
                }
            } catch (InterruptedException e) {
                System.out.println("İş parçacığı iptal edildi!");
            }
        });
        isci.start();
        Thread.sleep(1000);
        isci.interrupt();   // sleep() içindeyken kes
        isci.join();
        System.out.println("main: bitti");
    }
}
```

```
çalışıyor...
çalışıyor...
çalışıyor...
İş parçacığı iptal edildi!
main: bitti
```

İşçi, bayrak kurulmadıkça döngüde kalır. Her turda bir satır yazar ve 400 ms uyur. `main` bir saniye sonra `interrupt()` çağırır.

Adım	Satır	Değişkenler / durum	Yeni çıktı
1	1: `Thread isci = new Thread(() -> {` 11: `isci.start();`	isci=RUNNABLE	—
2	3: `while (!Thread.interrupted()) {` 4: `System.out.println("çalışıyor...");` 5: `Thread.sleep(400);`	isci=TIMED_WAITING	`çalışıyor...`
3	12: `Thread.sleep(1000);`	main=TIMED_WAITING	—
4	3: `while (!Thread.interrupted()) {` 4: `System.out.println("çalışıyor...");` 5: `Thread.sleep(400);`	zaman=≈400 ms	`çalışıyor...`
5	3: `while (!Thread.interrupted()) {` 4: `System.out.println("çalışıyor...");` 5: `Thread.sleep(400);`	zaman=≈800 ms	`çalışıyor...`
6	13: `isci.interrupt(); // isci sleep() içinde`	zaman=≈1000 ms	—

Adım	Satır	Değişkenler / durum	Yeni çıktı
7	7: `} catch (InterruptedException e) {` 8: `System.out.println("iptal edildi!");`	e=InterruptedException	`iptal edildi!`
8	14: `isci.join();`	isci=TERMINATED	—
9	15: `System.out.println("main: bitti");`	—	`main: bitti`

Tablo 10.20: Kesme programının izlemesi.

İşçi 0, yaklaşık 400 ve 800 ms'de yazar. Üçüncü uykusu 1200 ms'de bitecektir. `main` 1000 ms'de uyanır ve `interrupt()` çağırır. 7. adımda işçi `sleep` içindedir; uyku hemen kesilir, `InterruptedException` fırlar ve bayrak yeniden `false` olur.



Şekil 10.27: 7. adım. `interrupt()` işçiyi `sleep` içinde yakaladı; uyku kesildi ve `catch` çalıştı.

`catch` bloğu iletisini yazar ve `run()` biter. `main` `join()` ile işçinin bitmesini bekler ve son satırını yazar. Program on kez çalıştırıldı ve hep üç `çalışıyor...` satırı yazdı. Kesme bir istektir: iş parçacığı bayrağı ya da istisnayı kendisi ele alır ve çıkar.

Örnek 10.10

Bir iş parçacığı `Thread.sleep(5000)` içindeyken `main` ona `interrupt()` gönderiyor. `catch` içinde `Thread.currentThread().isInterrupted()` yazdırılıyor. Ardından `main` kendi bayrağını `Thread.currentThread().interrupt()` ile kurup iki kez `Thread.interrupted()` yazdırıyor. Çıktı nedir?

`sleep` `InterruptedException` fırlatırken bayrağı temizler; `catch` içinde `isInterrupted()` `false` 'tur ve `kesildi false` yazılır. `main` 'in ilk `Thread.interrupted()` çağrısı kurulu bayrağı okur ve temizler: `true` . İkincisi temizlenmiş bayrağı okur: `false` . Program üç çalıştırmada da `kesildi false` ve `true false` yazdı.

Sık yapılan hata

`InterruptedException` 'ı boş bir `catch` ile yutmak. İstisna fırlarken bayrak temizlenmiştir. `catch` hiçbir şey yapmazsa kesme isteği kaybolur ve iş parçacığı çalışmayı sürdürür. Doğrusu döngüden çıkmak ya da isteği yukarıya iletmek için `catch` içinde

`Thread.currentThread().interrupt()` ile bayrağı yeniden kurmaktır.

10.11.4 Askıya alma

Eski Java'da `suspend()` bir iş parçacığını dondurur, `resume()` devam ettirirdi. `suspend()` iş parçacığını tuttuğu kilitlerle birlikte dondurduğu için kilitlenmeye çok açıktı; JDK 21'de iki metot da `UnsupportedOperationException` fırlatır. Aynı iş `wait()` ve `notifyAll()` ile güvenle yapılır.

```
class Gorev implements Runnable {
    private boolean duraklatildi = false;
    public synchronized void duraklat() {
        duraklatildi = true;
    }
    public synchronized void devamEt() {
        duraklatildi = false;
        notifyAll();   // bekleyeni uyandır
    }
    public void run() {
        try {
            for (int i = 1; i <= 5; i++) {
                synchronized (this) {
                    while (duraklatildi) wait();
                }
                System.out.println("adım " + i);
                Thread.sleep(200);
            }
        } catch (InterruptedException e) { }
    }
}
```

Görev her adımın başında `duraklatildi` bayrağını kilit altında denetler. Bayrak kalkmışsa `wait()` ile bekler. `devamEt()` bayrağı indirir ve `notifyAll()` ile bekleyeni uyandırır. Bayrak her iki metotta da aynı monitörle korunur; ayrıca `volatile` olmasına gerek yoktur.

```
Gorev g = new Gorev();
Thread t = new Thread(g);
t.start();
Thread.sleep(500);
g.duraklat();
System.out.println("main: duraklatıldı");
Thread.sleep(1000);
System.out.println("main: devam");
g.devamEt();
t.join();
```

```
adım 1
adım 2
adım 3
main: duraklatıldı
main: devam
adım 4
adım 5
```

Görev 0, 200 ve 400 ms'de adım yazar. `main` 500 ms'de duraklatır. Görev 600 ms'de uyanınca bayrağı görür ve bekler. Bir saniye boyunca hiçbir adım yazılmaz. `main` devam ettirince kalan iki adım yazılır. Program on kez çalıştırıldı ve hep bu yedi satırı yazdı. Duraklatma, görev bir adımın ortasındayken değil, adımlar arasında devreye girer; bu yüzden hiçbir nesne yarım kalmaz.

10.11.5 `ThreadLocal`

Yarışın en kesin çaresi paylaşmamaktır. `ThreadLocal`, her iş parçacığına aynı değişkenin ayrı bir kopyasını verir.

```
static final ThreadLocal<String> deger =
        new ThreadLocal<>();
public static void main(String[] args)
        throws InterruptedException {
    Thread thread1 = new Thread(() -> {
        // İlk iş parçacığı
        deger.set("Değer 1");
        System.out.println("thread1: " + deger.get());
    });
    Thread thread2 = new Thread(() -> {
        // İkinci iş parçacığı
        deger.set("Değer 2");
        System.out.println("thread2: " + deger.get());
    });
    thread1.start();
    thread2.start();
    thread1.join();
    thread2.join();
    System.out.println("main: " + deger.get());
```

```
thread1: Değer 1
thread2: Değer 2
main: null
```

İki iş parçacığı aynı `deger` nesnesine yazar, ama her biri kendi kopyasına yazar ve kendi kopyasını okur. Kopyalar birbirinden bağımsızdır. `main` hiç `set` etmediği için kendi kopyası boştur ve `null` görür. İlk iki satırın sırası değişir: 40 çalıştırmada 20 kez `thread1`, 20 kez `thread2` önce yazdı.

`main: null` satırı ise hep sondaydı.

Her iş parçacığına ayrı bir başlangıç değeri vermek için `ThreadLocal.withInitial(() -> 0)` (Java 8+) yazılır ya da `initialValue()` metodu geçersiz kılınır. Tipik kullanım, iş parçacığı güvenli olmayan bir nesnenin, örneğin bir tarih biçimleyicinin, her iş parçacığına ayrı bir kopyasını vermektir.

Örnek 10.11

`ThreadLocal<Integer> t = ThreadLocal.withInitial(() -> 0);` ardından `main`'de `t.set(5)`. Yeni bir `x` iş parçacığı `t.set(t.get() + 1)` yapıp `"x: " + t.get()` yazıyor; `main` `x`'i bekleyip `"main: " + t.get()` yazıyor. Çıktı nedir?

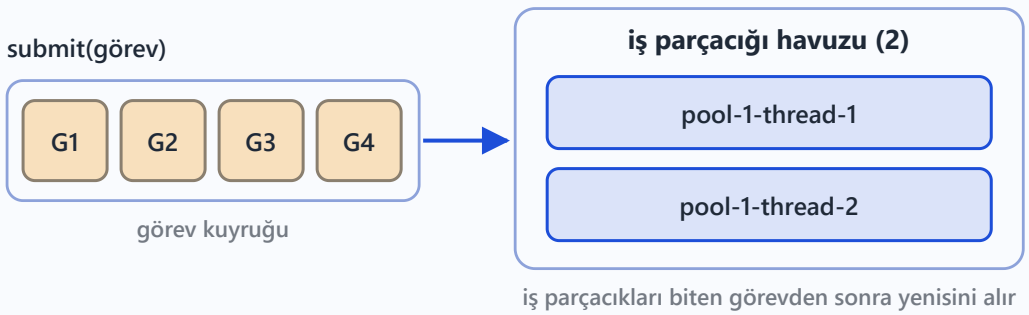
`main`'in 5'i `x`'e geçmez. `x` ilk `get()`'te `withInitial`'daki 0 ile başlar ve 1 yapar: `x: 1`. `x`'in yazdığı da `main`'in kopyasını etkilemez: `main: 5`.

10.11.6 İş parçacığı havuzları

Her görev için yeni bir iş parçacığı açmak iki sorun doğurur. İş parçacığı oluşturmak ucuz değildir: yığın ayrılır, işletim sistemi kaydı yapılır. Daha önemlisi sayı denetimsiz büyür: bin istek gelirse bin iş parçacığı açılır.

Tanım 10.20

İş parçacığı havuzu (*thread pool*), kullanılacak iş parçacıklarını önceden oluşturan ve görevleri bunlara dağıtan yapıdır. Görevler havuza gönderilir (`submit`); boştaki bir işçi görevi alır, bitirince sıradakine geçer. Boş işçi yoksa görev kuyrukta bekler.



Şekil 10.28: Görevler kuyruğa eklenir; havuzdaki iki iş parçacığı sırayla görev alır.

Havuz iş parçacıklarını yeniden kullanır ve sayılarını sınırlar. Java'da havuzları `ExecutorService` arayüzü yönetir. `Executors.newFixedThreadPool(n)` sabit `n` işçili hazır bir havuz verir. Ayrıntılı ayar gerektiğinde havuzu oluşturan ve yöneten `ThreadPoolExecutor` sınıfı doğrudan kullanılır.

```
public class HavuzOrnegi {
    static void gorev(int no) {
        String ad = Thread.currentThread().getName();
        System.out.println("Görev " + no + " -> " + ad);
    }
    public static void main(String[] args) throws Exception {
        ExecutorService havuz = Executors.newFixedThreadPool(2);
        for (int i = 1; i <= 4; i++) {
            int no = i;
            havuz.submit(() -> gorev(no));
        }
        // Callable: değer döndüren görev
        Future<Integer> f = havuz.submit(() -> 6 * 7);
        System.out.println("Callable sonucu: " + f.get());
        havuz.shutdown();   // yeni görev kabul etme
        havuz.awaitTermination(5, TimeUnit.SECONDS);
        System.out.println("Havuz kapandı");
    }
}
```

Sunumdaki ilk çalıştırma şudur:

```
Görev 1 -> pool-1-thread-1
Görev 2 -> pool-1-thread-2
Görev 3 -> pool-1-thread-2
Görev 4 -> pool-1-thread-2
Callable sonucu: 42
Havuz kapandı
```

Dört görev iki işçiye dağılır. Görevlerin hangi işçiye düştüğü ve satırların sırası değişir; sunum ikinci bir çalıştırmada son iki satırı `Görev 3 -> pool-1-thread-1` ve `Görev 4 -> pool-1-thread-2` olarak verir. Not için 40 çalıştırmada 6 farklı çıktı görüldü. Değişmeyenler de vardır: işçi adları hep `pool-1-thread-1` ve `pool-1-thread-2` 'dir; havuz üçüncü bir iş parçacığı açmaz. Son iki satır da hep aynıdır.

`Runnable` değer döndürmez. Değer döndüren görevler için `Callable` arayüzü vardır; lambda `() -> 6 * 7` bir `Callable<Integer>` 'dır. `submit` hemen bir `Future` döndürür. `Future`, ileride hazır olacak sonucu temsil eder. `get()` sonuç hazır olana kadar bekler.

`shutdown()`, havuzun yeni görev almayı bırakmasını ister; kuyruktaki görevler yine de bitirilir. `awaitTermination` bütün görevlerin bitmesini en çok verilen süre kadar bekler. `shutdown()` unutulursa havuzun işçileri arka plan iş parçacığı olmadığı için program açık kalır.

Java 21 yeni bir seçenek getirdi: `Executors.newVirtualThreadPerTaskExecutor()` her göreve ayrı bir **sanal iş parçacığı** (*virtual thread*) verir. Sanal iş parçacıklarını işletim sistemi değil JVM yönetir; milyonlarcası açılabilir. Bekleme ağırlıklı, çok sayıda kısa görev için uygundurlar. Bir denemede böyle bir görevin içinde `isVirtual()` ve `isDaemon()` ikisi de `true` döndü.

10.11.7 Kod izleme: havuz ve `Future`

Havuzun zaman içinde nasıl çalıştığını görmek için görevler uyuyarak uzun iş taklidi yapar. `is(ms, d)`, `ms` milisaniye uyuyup `d`'yi döndürür.

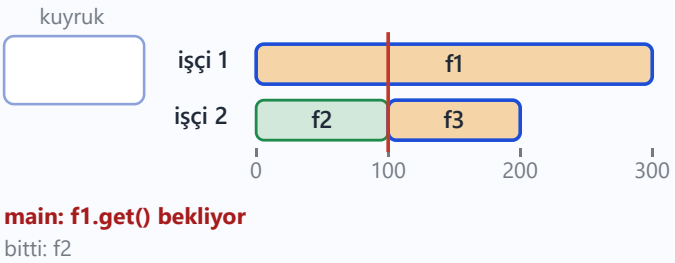
```
static int is(int ms, int d) throws Exception {
    Thread.sleep(ms);   // uzun süren iş yerine
    return d;
}
// main içinde:
ExecutorService havuz = Executors.newFixedThreadPool(2);
Future<Integer> f1 = havuz.submit(() -> is(300, 10));
Future<Integer> f2 = havuz.submit(() -> is(100, 20));
Future<Integer> f3 = havuz.submit(() -> is(100, 30));
System.out.println(f1.isDone() + " " + f3.isDone());
int toplam = f1.get() + f2.get() + f3.get();
System.out.println("toplam = " + toplam);
System.out.println(f2.isDone() + " " + f3.isDone());
havuz.shutdown();
```

Adım	Satır	Değişkenler / durum	Yeni çıktı
1	6: `ExecutorService havuz = Executors.newFixedThreadPool(2);`	havuz=2 işçi	—
2	7: `Future<Integer> f1 = havuz.submit(() -> is(300, 10));`	f1=Future (bekliyor)	—
3	8: `Future<Integer> f2 = havuz.submit(() -> is(100, 20));`	f2=Future	—
4	9: `Future<Integer> f3 = havuz.submit(() -> is(100, 30));`	f3=Future	—
5	10: `System.out.println(f1.isDone() + " " + f3.isDone());`	f1.isDone()=false	`false false`
6	11: `int toplam = f1.get() + f2.get() + f3.get();`	zaman=≈100 ms	—
7	11: `int toplam = f1.get() + f2.get() + f3.get();`	zaman=≈200 ms	—
8	11: `int toplam = f1.get() + f2.get() + f3.get();`	toplam=10+20+30=60	—
9	12: `System.out.println("toplam = " + toplam);` 13: `System.out.println(f2.isDone() + " " + f3.isDone());`	toplam=60	`toplam = 60` ⏎ `true true`
10	14: `havuz.shutdown();`	havuz=kapanıyor	—

Tablo 10.21: İki işçili havuzda üç görevin izlemesi (süreler yaklaşık).

Havuzun iki işçisi vardır. `f1` işçi 1'de, `f2` işçi 2'de başlar. İki işçi de meşgul olduğu için `f3` kuyrukta bekler; yeni iş parçacığı açılmaz. 5. adımda hiçbir görev bitmemiştir; `isDone()` beklemeden `false` döner. 6. adımda `f1.get()` sonuç hazır olana kadar `main`'i durdurur. Bu sırada, yaklaşık 100 ms'de

`f2` biter ve işçi 2 kuyruktan `f3`'ü alır.



Şekil 10.29: 6. adım. `main` `f1.get()`'te bekliyor; `f2` bitti ve işçi 2 kuyruktaki `f3`'ü aldı.

Yaklaşık 200 ms'de `f3` de biter; `main` hâlâ `f1`'i bekler. 300 ms'de `f1` hazır olur. `f2` ve `f3` zaten bittiği için onların `get()`'leri beklemeden döner: toplam 10 + 20 + 30 = 60. Görevler tek iş parçacığında sırayla çalışsaydı 300 + 100 + 100 = 500 ms sürerdi; havuzla toplam süre yaklaşık 300 ms'dir. Program on kez çalıştırıldı; çıktı hep `false false`, `toplam = 60`, `true true` idi ve ölçülen süre her seferinde 300 ile 350 ms arasındaydı. Görevlerin hangi işçiye düştüğü ise çalıştırmadan çalıştırmaya değişebilir.

10.11.8 `CompletableFuture` ve eş zamanlı koleksiyonlar

```
CompletableFuture<Integer> cf = CompletableFuture
    .supplyAsync(() -> 6)
    .thenApply(x -> x * 7)              // 42
    .thenCombine(CompletableFuture
        .supplyAsync(() -> 8), Integer::sum);
System.out.println(cf.get());           // 50

Map<String, Integer> say =
    new ConcurrentHashMap<>();
String[] k = {"elma", "armut", "elma"};
Arrays.stream(k).parallel()
    .forEach(w -> say.merge(w, 1, Integer::sum));
System.out.println(say);  // {elma=2, armut=1}
```

Bu kod iki farklı aracı gösterir ve on çalıştırmada da `50` ile `{elma=2, armut=1}` yazdı.

`CompletableFuture`
: `get()` ile beklemek yerine işlemleri zincirler; her adım öncekinin bitmesiyle arka planda çalışır. Kodda 6 üretilir, 7 ile çarpılır (42) ve ayrı bir görevin ürettiği 8 ile toplanır: 50.

`ConcurrentHashMap`
: İş parçacığı güvenli eşlemdir. `merge` ve `compute` metotları atomiktir. Paralel akış, dizinin öğelerini farklı iş parçacıklarında işler; `merge` sayımın kaybolmamasını sağlar.

`CopyOnWriteArrayList`

Her yazmada iç diziyi kopyalar. Çok okunan, az yazılan listeler için uygundur.

`ForkJoinPool`
: Böl ve fethet görevleri (`RecursiveTask`) içindir. Boştaki işçi başkasının kuyruğundan iş çalar (*work stealing*). Paralel akışlar (`parallelStream()`) bu havuzu kullanır.

Paralel akış her zaman hızlı değildir. Küçük veride iş parçacıklarını eşgüdümleme maliyeti kazancı aşar. Paylaşılan değiştirilebilir bir değişkene yazan bir paralel akış ise sonucu bozar. Paralelleştirme kararı ölçümle verilir.

10.11.9 İş parçacığı güvenliği

Tanım 10.21

Bir sınıf, birden çok iş parçacığından aynı anda kullanıldığında da doğru davranıyorsa **iş parçacığı güvenlidir** (*thread-safe*).

İş parçacığı güvenliğine dört yoldan ulaşılır:

Değiştirilemez nesneler
: Oluşturulduktan sonra değişmeyen nesneler (`String`, bütün alanları `final` olan sınıflar, Java 16+ `record`) senkronizasyonsuz paylaşılabilir. Değişmeyen veri yarışmaz.

Paylaşmamak
: Yerel değişkenler her iş parçacığının kendi yığınındadır; `ThreadLocal` da her iş parçacığına kopya verir.

Senkronizasyon
: Paylaşılan değişen veriye her erişim aynı kilitle korunur: `synchronized` ya da `Lock`.

Hazır yapılar
: `AtomicInteger`, `ConcurrentHashMap`, `CopyOnWriteArrayList`, `BlockingQueue`.

Koleksiyon çerçevesinin sıradan sınıfları, `ArrayList` ve `HashMap`, iş parçacığı güvenli değildir. İki iş parçacığı aynı `ArrayList`'e 10.000'er öğe eklediğinde not için yapılan on çalıştırmanın hiçbirinde boyut 20.000 çıkmadı; bulunan değerler 10.004 ile 14.121 arasındaydı. Dört çalıştırmada iş parçacıklarından biri, iç dizi büyütülürken `ArrayIndexOutOfBoundsException` ile öldü. Bu sınıflar ya bir kilitle korunur ya da yerlerine eş zamanlı karşılıkları kullanılır.

10.12 Çözümlü problemler

Örnek 10.12 (Problem 1)

Çıktı nedir? Her çalıştırmada aynı mı?

```
Thread t = new Thread(() -> System.out.print("B"));
System.out.print("A");
t.start();
t.join();
System.out.print("C");
```

`ABC`, her zaman. `A`, iş parçacığı başlatılmadan önce yazılır. `B` yeni iş parçacığında yazılır; `join()`, `B` yazılıp iş parçacığı bitmeden `main`'i `C`'ye geçirmez. On çalıştırmada da çıktı `ABC` idi.

Örnek 10.13 (Problem 2)

Çıktı nedir?

```
Thread t = new Thread(() -> System.out.print("X"));
System.out.print(t.getState() + " ");
t.run();
System.out.print(" " + t.getState());
```

`NEW X NEW`. `run()` sıradan bir çağrıdır: `X`'i `main` yazar, iş parçacığı hiç başlamaz ve `NEW` kalır.

Örnek 10.14 (Problem 3)

"İki iş parçacığı" örneğinin olası çıktıları hangileridir? a) `A1 A2 B1 B2 main`; b) `B1 A1 A2 B2 main`; c) `A1 main B1 A2 B2`; d) `A2 A1 B1 B2 main`. (Kısaltma: `A1`, `A: adım 1` satırıdır.)

a ve b. c olamaz, çünkü `main` iki `join()`'den sonra yazar. d olamaz, çünkü aynı iş parçacığının adımları kendi içinde sırasını korur. a, not için yapılan ölçümlerde en sık görülen sıraydı. b'nin aynası olan `A1 B1 B2 A2` de tek JVM içindeki 6.000 denemede iki kez görüldü.

Örnek 10.15 (Problem 4)

Her durumda iş parçacığının durumu nedir? a) `sleep(500)` içinde; b) dolu bir `synchronized` bloğa girerken; c) `wait()` sonrası; d) `run()` bitince.

a) `TIMED_WAITING`; b) `BLOCKED`; c) `WAITING`; d) `TERMINATED`. c'de iş parçacığı bekleme kümesindedir. `notify` gelip kilidi yeniden almaya çalışırken ise `BLOCKED` olur.

Örnek 10.16 (Problem 5)

Aşağıdaki sınıf iş parçacığı güvenli midir?

```
class Hesap {
    private int bakiye = 0;
    void yatir(int m) { bakiye += m; }
    int getBakiye() { return bakiye; }
}
```

Hayır: `bakiye += m` oku–ekle–yaz'dır. İki iş parçacığı 100.000'er kez `yatir(1)` çağırdığında on çalıştırmanın hiçbirinde bakiye 200.000 olmadı; sonuçlar 126.397 ile 171.138 arasında değişti. Çözüm: metotlar `synchronized` olur ya da alan `AtomicInteger` yapılıp `addAndGet(m)` kullanılır.

Örnek 10.17 (Problem 6)

İki iş parçacığı bunları çalıştırıyor. Ne olabilir?

```
// T1                    // T2
synchronized (a) {       synchronized (b) {
  synchronized (b) { }     synchronized (a) { }
}                        }
```

Kilitlenme: T1 `a` 'yı, T2 `b` 'yi alırsa ikisi de ötekini sonsuza dek bekler. Uyku olmadığı için kilitlenme her çalıştırmada olmaz; bu yüzden tehlikelidir. Bloklar birer döngüde binlerce kez çalıştırıldığında kilitlenme üç denemenin üçünde de görüldü. Çözüm: iki iş parçacığı da önce `a` 'yı, sonra `b` 'yi alır.

Örnek 10.18 (Alıştırma: iki yarım toplam)

1..1000 sayılarının toplamı iki iş parçacığıyla hesaplanacak: biri ilk yarıyı, öteki ikinci yarıyı toplar; `main` ikisini bekleyip toplamı yazar.

```
Thread sol = new Thread(() -> {
    for (int i = 0; i < 500; i++) parca[0] += dizi[i];
});
Thread sag = new Thread(() -> {
    for (int i = 500; i < 1000; i++) parca[1] += dizi[i];
});
sol.start(); sag.start();
sol.join(); sag.join();
```

Her iş parçacığı kendi gözüne yazar; iki iş parçacığı aynı göze hiç yazmadığı için yarış yoktur. `join()` sonrası okuma güvenlidir. `dizi[i] = i + 1` iken parçalar 125.250 ve 375.250'dir; toplam 500.500 = 1000 · 1001 / 2.

Örnek 10.19 (Alıştırma: çift sayıları saymak)

1..1000 arasındaki çift sayılar dört iş parçacığıyla, her biri 250 sayı alarak, ortak bir sayaçta sayılacak.

Sayaç `AtomicInteger cift` olur. Her iş parçacığı `for (int i = bas; i < bas + 250; i++) if (i % 2 == 0) cift.incrementAndGet();` çalıştırır; `bas` değerleri 1, 251, 501 ve 751'dir. Bütün `join()`'lerden sonra çıktı `Çift sayı: 500`'dür. Sayaç sıradan bir `int` olsaydı sonuç yarış yüzünden 500'den küçük çıkabilirdi.

Örnek 10.20 (Alıştırma: karelerin toplamı)

1..5 sayılarının kareleri 3 iş parçacıklı bir havuzda `Callable` görevleriyle hesaplanacak ve `Future`'lardan toplanacak.

```
ExecutorService havuz = Executors.newFixedThreadPool(3);
List<Future<Long>> sonuclar = new ArrayList<>();
for (int i = 1; i <= 5; i++) {
    int n = i;
    sonuclar.add(havuz.submit(() -> kare(n)));
}
long toplam = 0;
for (Future<Long> f : sonuclar) toplam += f.get();
havuz.shutdown();
System.out.println("Karelerin toplamı: " + toplam);
```

Yardımcı metot `static long kare(int n) { return (long) n * n; }`'dir. `Future`'lar gönderilme sırasıyla listede durur; `get()` her birinin sonucunu bekler. Çıktı `Karelerin toplamı: 55`'tir (1 + 4 + 9 + 16 + 25).

Örnek 10.21 (Alıştırma: saat)

Her saniye `tik` yazan bir iş parçacığı yazılacak; `main` 3,5 saniye sonra onu `interrupt()` ile durduracak.

Kesme örneğindeki kalıp kullanılır: döngüde `while (!Thread.interrupted())`, gövdede `tik` yazıp `sleep(1000)`; `catch`'ten sonra `saat durdu`. `main` `sleep(3500)`, `interrupt()` ve `join()` çalıştırır. İş parçacığı 0, 1, 2 ve 3. saniyelerde yazar; 3,5. saniyede uykusu kesilir. Çıktı dört `tik` ve `saat durdu`'dur; beş çalıştırmada da böyleydi.

10.13 Sık yapılan hatalar

Aşağıdaki iki tablo, sunumdaki sık yapılan hataları toplar. Derleyici iletileri JDK 21'in `javac`'ıyla, çalışma zamanı belirtileri programlar çalıştırılarak doğrulandı.

Hata	Belirti	Doğrusu
`t.run()` çağırmak	Yeni iş parçacığı yok; kod `main`'de sırayla çalışır	`t.start()`
`start()`'ı iki kez çağırmak	`IllegalThreadStateException`	Her çalıştırma için yeni `Thread`
`Thread.sleep(100);` yalnız başına	`error: unreported exception InterruptedException; must be caught or declared to be thrown`	`try`-`catch` ya da `throws InterruptedException`
Lambda içinde döngü değişkeni `i`	`error: local variables referenced from a lambda expression must be final or effectively final`	`int no = i;` kopyası
Fırlatılmayan istisnayı yakalamak: `try { while (!Thread.interrupted()) { } } catch (InterruptedException e)`	`error: exception InterruptedException is never thrown in body of corresponding try statement`	Gövdede `sleep`/`wait` olmalı ya da `catch` kaldırılmalı
Sınıf `Thread`'i genişletmiyorken `currentThread()`	`error: cannot find symbol`	`Thread.currentThread()`
`join()`'u unutmak	`main`, sonuç hazır olmadan okur (ör. 0)	Sonucu okumadan önce `join()`

Tablo 10.22: İş parçacığı oluşturma ve yönetme hataları.

Hata	Belirti	Doğrusu
`wait()`/`notify()`'ı `synchronized` dışında çağırmak	`IllegalMonitorStateException: current thread is not owner`	Aynı nesne üzerinde `synchronized` içinde
Koşulu `if` ile beklemek	Uyanınca koşul yine bozuk olabilir; taşan ya da boşken okunan kuyruk	`while (koşul) wait();`
`unlock()`'u `finally` dışında yazmak	İstisnada kilit hiç açılmaz; herkes sonsuza dek bekler	`lock(); try { … } finally { unlock(); }`

Hata	Belirti	Doğrusu
Alınmamış kilidi açmak	`IllegalMonitorStateException`	Her `lock()` için bir `unlock()`
`volatile` sayaçta `sayac++`	Artışlar kaybolur (ör. 163299)	`AtomicInteger` ya da `synchronized`
`final volatile` alan	`error: illegal combination of modifiers: final and volatile`	Yalnız biri kullanılır
`stop()`, `suspend()` kullanmak	Java 20+: `UnsupportedOperationException`	Bayrak, `interrupt()`, `wait` / `notify`
`InterruptedException`'ı boş `catch` ile yutmak	Kesme isteği kaybolur, iş parçacığı durmaz	Döngüden çıkmak ya da `Thread.currentThread().interrupt()`
Kilitleri farklı sırada almak	Kilitlenme: program donar	Herkes aynı sırayla alır

Tablo 10.23: Senkronizasyon hataları.

Bu hataların bir kısmını derleyici yakalar; geri kalanı ancak çalışma sırasında, çoğu zaman da yalnız bazı çalıştırmalarda ortaya çıkar. Yarış ve kilitlenme hataları yükün yüksek olduğu bir anda, örneğin bir sunucuda yoğun saatte görünür hâle gelir. Bu yüzden çok iş parçacıklı kodda "denedim, çalıştı" bir kanıt değildir. Paylaşılan her değişken için hangi kilidin onu koruduğu açıkça bilinmelidir.

10.14 Özet

- İş parçacığı, bir süreç içinde bağımsız ilerleyen yürütme akışıdır. Aynı sürecin iş parçacıkları heap'i ve sınıf alanını paylaşır; her birinin kendi yığını, PC yazmacı ve yerel değişkenleri vardır.
- Eş zamanlılık, görevlerin aynı zaman aralığında ilerlemesidir ve tek çekirdekte bile olur. Paralellik, görevlerin farklı çekirdeklerde gerçekten aynı anda çalışmasıdır.
- Hangi iş parçacığının ne zaman çalışacağına çizelgeleyici karar verir. Birden çok iş parçacıklı programların çıktı sırası bu yüzden çalıştırmaya göre değişebilir.
- İş parçacığı `Runnable`'ı uygulayarak, `Thread`'i genişleterek ya da lambda ile yazılır. İş `run()`'a yazılır; iş parçacığı `start()` ile başlatılır. `run()`'ı doğrudan çağırmak yeni iş parçacığı açmaz.
- Altı durum vardır: `NEW`, `RUNNABLE`, `BLOCKED`, `WAITING`, `TIMED_WAITING`, `TERMINATED`. Biten iş parçacığı yeniden başlatılamaz.
- `join()` bir iş parçacığının bitmesini bekler ve onun yazdıklarını görünür kılar. `sleep` kilidi bırakmaz.
- Arka plan iş parçacıkları JVM'in kapanmasını engellemez; `setDaemon(true)`, `start()`'tan önce çağrılır.
- İş parçacığı iş birliğiyle durdurulur: `volatile` bir bayrak ya da `interrupt()`. `stop()` Java 20'den beri çalışmaz.
- Yarış durumu, paylaşılan veride bölünebilir işlemlerden doğar. `sayac++` oku–ekle–yaz adımlarından oluşur ve araya giren iş parçacığı artışları kaybettirir.

- `synchronized`, bir nesnenin monitörünü alarak karşılıklı dışlama sağlar. Örnek metot `this`'i, `static` metot `Sınıf.class`'ı kilitler. Koruma ancak herkes aynı kilidi kullanırsa işe yarar.
- `volatile` görünürlüğü sağlar, atomikliği sağlamaz. Önce-olur ilişkileri (kilit, `volatile`, `start`, `join`) bir iş parçacığının yazdığının ötekine ne zaman görüneceğini belirler.
- `Lock` açık kilittir: `tryLock`, süreli ve kesilebilir bekleme, adillik ve birden çok `Condition` sunar. `unlock()` her zaman `finally`'dedir. `ReentrantLock` ve `synchronized` yeniden girişlidir.
- `ReadWriteLock`, okuyucuların aynı anda okumasına izin verir, yazıcıyı tek başına bırakır.
- Kilitlenme, kilitlerin farklı sırada alınmasıyla oluşur. Sabit kilit sırası, süreli deneme ve kısa kritik bölgeler onu önler.
- Koşul beklemesi için `wait` / `notifyAll` ya da `Condition`'ın `await` / `signal`'ı kullanılır; koşul `while` ile denetlenir. Hazır `ArrayBlockingQueue` üretici–tüketici problemini kendisi çözer.
- `CountDownLatch` olayları sayar, `CyclicBarrier` iş parçacıklarını buluşturur, `Exchanger` takas yaptırır, `Semaphore` eş zamanlı girişi sınırlar.
- Atomik sınıflar `compareAndSet` ile kilitsiz ve güvenli güncelleme yapar. `ThreadLocal` her iş parçacığına ayrı kopya verir.
- İş parçacığı havuzları iş parçacıklarını yeniden kullanır ve sayılarını sınırlar. `Callable` değer döndürür, `Future.get()` sonucu bekler, `shutdown()` unutulmaz.
- İş parçacığı güvenliğinin yolları: değiştirilemez nesneler, paylaşmamak, senkronizasyon ve hazır eş zamanlı yapılar. `ArrayList` ve `HashMap` güvenli değildir.

10.15 Kavram listesi

iş parçacığı *thread* · süreç *process* · eş zamanlılık / paralellik *concurrency / parallelism* · iş parçacığı durumları *thread states* · arka plan iş parçacığı *daemon thread* · Java Sanal Makinesi (JVM) *Java Virtual Machine* · fonksiyonel arayüz *functional interface* · lambda ifadesi *lambda expression* · yarış durumu *race condition* · senkronizasyon *synchronization* · monitör *monitor* · karşılıklı dışlama *mutual exclusion* · kilit *lock* · kilitlenme *deadlock* · üretici–tüketici problemi *producer–consumer problem* · semafor *semaphore* · atomik işlem *atomic operation* · iş parçacığı havuzu *thread pool* · iş parçacığı güvenliği *thread safety* · değiştirilemez *immutable*

bağlam değiştirme *context switching* · çizelgeleyici *scheduler* · kritik bölge *critical section* · görünürlük, `volatile` *visibility* · önce-olur ilişkisi *happens-before* · yeniden girişli kilit *reentrant lock* · canlı kilit / açlık *livelock / starvation* · bekleme kümesi *wait set* · karşılaştır ve değiştir *compare-and-set* · kesme *interrupt*

10.16 Kendini Dene

1. Bir `static` metot `int toplam = 0;` tanımlayıp bir döngüde 1.000 kez `toplam++` yapıyor ve sonucu yazıyor. A ve B adlı iki iş parçacığı bu metodu çalıştırırsa her biri ne yazar? Yarış olur mu?

   Cevap

   İkisi de 1000 yazar. `toplam` yerel bir değişkendir ve her iş parçacığının kendi yığınındadır; iki kopya birbirini görmez. Yarış ancak paylaşılan bir alana (nesne alanı ya da `static` alan) yazılınca olur.

2. Bir sınıfta `static synchronized void s() { ortak++; }` ve `synchronized void o() { ortak++; }` var; `ortak` bir `static int`'tir. Bir iş parçacığı 100.000 kez `s()`, öteki aynı nesne üzerinde 100.000 kez `o()` çağırıyor. Sonuç her zaman 200.000 müdür?

   Cevap

   Hayır. `s()` sınıfın `.class` nesnesini, `o()` `this`'i kilitler. İki farklı kilit birbirini engellemez ve `ortak++` iki iş parçacığından aynı anda çalışabilir. Beş denemede sonuç her seferinde 200.000'den küçük çıktı (180.586 ile 198.518 arası). Çözüm iki metodun da aynı kilidi kullanmasıdır.

3. `ReentrantLock k` üzerinde `main` iki kez `lock()`, bir kez `unlock()` çağırıyor. Ardından başka bir iş parçacığı `k.tryLock()` çağırıyor. Ne döner? `main`'de `k.getHoldCount()` kaçtır?

   Cevap

   `tryLock()` `false` döner; `getHoldCount()` 1'dir. Tutma sayısı 2'den 1'e inmiştir ve kilit hâlâ `main`'dedir. Kilit ancak ikinci `unlock()` ile bırakılır.

4. `CountDownLatch m = new CountDownLatch(3);` üzerinde iki kez `countDown()` çağrılıyor. `m.await(100, TimeUnit.MILLISECONDS)` ne döndürür? `m.getCount()` kaçtır?

   Cevap

   `false` ve 1. Sayaç 0'a inmediği için mandal açılmaz; süreli `await` 100 ms bekler ve süre dolunca `false` döndürür. Süresiz `await()` çağrılsaydı iş parçacığı üçüncü olay gelene kadar beklerdi.

5. `Executors.newFixedThreadPool(1)` ile oluşturulan havuza, her biri kendi numarasını ve iş parça-

cığının adını yazan üç görev gönderiliyor. Çıktının sırası ve iş parçacığı adları nedir?

Cevap

Havuzda tek işçi olduğu için görevler kuyruk sırasıyla, birbiri ardına çalışır: 1, 2, 3. Üçü de `pool-1-thread-1` 'de çalışır; havuz yeni iş parçacığı açmaz. Tek işçili havuz bu yüzden görevlerin sırasını korur. Beş denemede çıktı hep aynıydı.

6. `AtomicInteger a = new AtomicInteger(10);` ardından `a.compareAndSet(10, 20)`, `a.compareAndSet(10, 30)` ve `a.getAndIncrement()` çağrılıyor. Üç çağrı ne döndürür, `a` 'nın son değeri nedir?

Cevap

`true`, `false`, `20`; son değer 21. İlk CAS 10'u bulur ve 20 yazar. İkincisi 10 bekler, ama değer 20'dir; dokunmaz. `getAndIncrement()` önce eski değeri (20) döndürür, sonra artırır.

7. `main` önce `Thread.currentThread().interrupt()` çağırıyor, sonra `Thread.sleep(1000)` 'i bir `try-catch` içinde çalıştırıyor. `sleep` bir saniye sürer mi? `catch` içinde `isInterrupted()` nedir?

Cevap

Sürmez. Kesme bayrağı kurulu bir iş parçacığı `sleep` 'e girerse `InterruptedException` hemen fırlar. Denemede istisna 100 ms'den kısa sürede geldi. İstisna fırlarken bayrak temizlenir; `isInterrupted()` `false` 'tur.

8. Üç iş parçacığı üç kilidi şöyle alıyor: T1 önce `a` sonra `b`; T2 önce `b` sonra `c`; T3 önce `c` sonra `a`. Kilitlenme olabilir mi? Olabilirse hangi kural bunu önler?

Cevap

Olabilir. Üçü de ilk kilidini alırsa T1 `b` 'yi, T2 `c` 'yi, T3 de `a` 'yı bekler. Beklenen her kilit bir sonraki iş parçacığındadır ve T1 → T2 → T3 → T1 döngüsü kurulur. Sabit bir sıra, örneğin `a` < `b` < `c` konur ve her iş parçacığı kilitlerini bu sırayla alır. T3 bu durumda önce `a` 'yı, sonra `c` 'yi alır ve döngü kurulamaz.

9. Engelleyen kuyruk örneğinde `notifyAll()` yerine `notify()` yazılırsa ne gibi bir sorun çıkabilir?

Cevap

Aynı bekleme kümesinde hem üreticiler hem tüketiciler bekleyebilir. `notify()` tek bir iş parçacığını rastgele seçer. Bir tüketici öğe aldıktan sonra yine bir tüketiciyi uyandırırsa, o tüketici kuyruğu boş bulup yeniden bekler. Bekleyen üretici ise hiç uyanmaz ve program takılabilir. `notifyAll()` herkesi uyandırır; ilgisiz olanlar `while` koşulunu bozuk bulup yeniden bekler. `Condition` ile üreticilere ve tüketicilere ayrı küme verilirse `signal()` yeterli olur.

10.17 İleri okuma

- Oracle, *The Java Tutorials*, "Lesson: Concurrency": süreçler ve iş parçacıkları, senkronizasyon, canlılık sorunları, korumalı bloklar ve yüksek düzeyli eş zamanlılık nesneleri üzerine resmî ve ücretsiz anlatımdır.
- B. Goetz ve diğerleri, *Java Concurrency in Practice*: iş parçacığı güvenliği, görünürlük, kilitler ve havuzlar üzerine alanın temel kitabıdır; örnekleri bugün de geçerlidir.
- J. Bloch, *Effective Java* (3. baskı), Bölüm 11, Madde 78–84: paylaşılan değişen veriye erişimi senkronize etmeyi, `wait` / `notify` yerine hazır araçları tercih etmeyi ve aşırı senkronizasyondan kaçınmayı gerekçeleriyle anlatır.
- J. Gosling ve diğerleri, *The Java Language Specification, Java SE 21 Edition*, Bölüm 17 "Threads and Locks": önce-olur ilişkisinin ve `volatile`'ın kesin tanımıdır.

10.18 Kaynaklar

1. Oracle, *The Java Tutorials*: Lesson: Concurrency (Processes and Threads, Synchronization, Liveness, Guarded Blocks, High Level Concurrency Objects).
2. J. Gosling ve diğerleri, *The Java Language Specification, Java SE 21 Edition*, Bölüm 17: Threads and Locks.
3. Oracle, *Java SE 21 API Documentation*: `java.lang.Thread`, `java.util.concurrent`, `java.util.concurrent.locks`, `java.util.concurrent.atomic`.
4. B. Goetz ve diğerleri, *Java Concurrency in Practice*, Addison-Wesley, 2006.
5. P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (Bölüm 23: Concurrency).
6. C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson (Bölüm 12: Concurrency).
7. K. Sierra, B. Bates ve T. Gee, *Head First Java*, 3. baskı, O'Reilly (iş parçacıkları ve eş zamanlılık bölümleri).
8. J. Bloch, *Effective Java*, 3. baskı, Addison-Wesley (Bölüm 11: Concurrency, Madde 78–84).

KISIM IV · BÖLÜM 11

Veri Tabanı Programlama

Programın değişkenleri ve nesneleri bellekte yaşar, program kapanınca kaybolur. Öğrenci notları, banka hesapları, sensör ölçümleri ise yıllarca saklanmalı, hızlı aranabilmeli ve aynı anda birçok program onları güvenle değiştirebilmelidir. Bu işi veri tabanları yapar. Not önce veri tabanı kavramını, dosyadan farkını ve ilişkisel modeli anlatır. Ardından gömülü bir veri tabanı olan SQLite'ı ve ilişkisel veri tabanlarının ortak dili SQL'i işler. SQL kesimi tablo kurmayı, kayıt eklemeyi, sorgulamayı, güncellemeyi ve silmeyi; tabloları birleştirmeyi ve gruplamayı gösterir. Sonraki kesimler Java'nın veri tabanı arayüzü JDBC'yi ve kullanıcı girdisini güvenle kullanmanın yolunu anlatır. Son kesimler satırları nesnelere dönüştürmeyi ve birden çok komutu tek bir iş birimi olarak yürüten işlemleri işler. Bütün kodlar JDK 21 ve SQLite JDBC sürücüsüyle çalıştırılmıştır.

Öğrenci Yönetimi Uygulaması: `github.com/sercankulcu/StudentManagement` · **Bu bölümün kodu:** `github.com/sercankulcu/object-oriented-programming-java/Bolum11/src`

ÖĞRENME ÇIKTILARI

Kavramları açıklamak
Veri tabanının neden gerektiğini, düz dosyadan farkını ve ilişkisel modeli (tablo, satır, sütun, birincil anahtar) açıklamak.

SQL yazmak
`CREATE TABLE`, `INSERT`, `SELECT`, `UPDATE` ve `DELETE` komutlarını yazmak; sorguyu `WHERE`, `ORDER BY` ve `LIMIT` ile daraltmak.

İlişkileri kullanmak
Yabancı anahtarla tabloları bağlamak; `JOIN` ile birleştirmek, `GROUP BY` ve `HAVING` ile gruplamak; `NULL` değerleri doğru karşılaştırmak.

JDBC ile çalışmak
`Connection`, `Statement` ve `ResultSet` ile Java'dan SQLite'a bağlanıp kayıt eklemek, okumak, güncellemek ve silmek; kaynakları try-with-resources ile kapatmak.

Güvenlik ve bütünlük
SQL enjeksiyonunu `PreparedStatement` ile önlemek; `commit` ve `rollback` ile birden çok komutu tek bir işlem olarak yürütmek.

Nesnelere dönüştürmek
Sorgu sonucundaki satırları nesnelere çevirmek ve bir tablonun SQL'ini veri erişim nesnesinde (DAO) toplamak.

11.1 Veri tabanı kavramı

Bu kesim veri tabanının ne olduğunu, hangi sorunu çözdüğünü ve veriyi nasıl düzenlediğini anlatır. Kod yoktur; burada tanımlanan kavramlar bütün JDBC kodunun temelidir.

11.1.1 Veri tabanı neden gerekir?

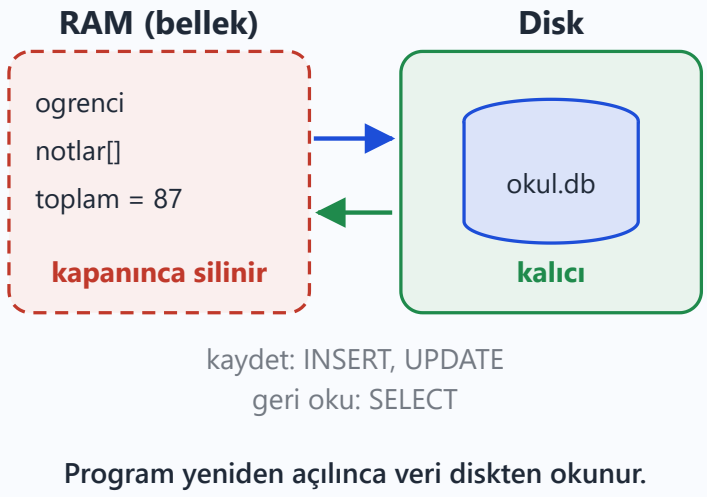
Bir Java programında `int toplam = 87;` yazıldığında 87 değeri RAM'de, yani ana bellekte bir yere konur. Nesneler de yığın (*heap*) bölgesinde yaşar. Bu bellek hızlıdır ama kalıcı değildir. Program kapandığında işletim sistemi programın belleğini geri alır; değişkenler ve nesneler kaybolur. Elektrik kesilirse bellek içeriği de silinir.

Oysa birçok verinin programdan uzun yaşaması gerekir. Bir öğrencinin vize notu dönem boyunca, bir banka hesabının bakiyesi yıllarca, bir oyunun en yüksek skoru oyun her açıldığında gerekir. Verinin bu biçimde saklanmasına **kalıcı depolama** (*persistent storage*) denir. Kalıcı depolama disk gibi elektrik kesilince silinmeyen bir ortamda yapılır.

Tanım 11.1

Veri tabanı (*database*), düzenli biçimde saklanan ve kolayca sorgulanabilen veri topluluğudur. Veri tabanını oluşturan, koruyan, sorgulayan ve güncelleyen yazılıma **veri tabanı yönetim sistemi** (VTYS; *database management system, DBMS*) denir.

Günlük dilde ikisi çoğu zaman karıştırılır: “MySQL veri tabanı” denir ama MySQL aslında bir VTYS'dir. Veri tabanı ise o sistemin yönettiği tablolar ve içlerindeki kayıtlardır. Şekil 11.1 iki ortamın farkını gösterir. Program çalışırken veriyi bellekte işler, kalıcı olması gerekenleri veri tabanına yazar (`INSERT`, `UPDATE`). Program yeniden açıldığında veri diskten geri okunur (`SELECT`).



Şekil 11.1: Bellek ve disk. RAM'deki değişkenler program kapanınca silinir; diskteki veri tabanı dosyası (`okul.db`) kalıcıdır.

Bir benzetme yapılabilir. Veri tabanı, kütüphanenin kataloglu arşivine benzer. Katalog olmasaydı bir kitabı bulmak için bütün rafları gezmek gerekirdi. Katalog sayesinde kitabın yeri birkaç adımda bulunur. Veri tabanı da milyonlarca kayıt arasından aranan kaydı bütün veriyi baştan sona okumadan bulur.

11.1.2 Dosya mı, veri tabanı mı?

Kalıcı depolama için veri tabanı tek yol değildir. Program veriyi bir metin dosyasına ya da CSV (*comma-separated values*, virgülle ayrılmış değerler) dosyasına da yazabilir. Ayar dosyası, günlük kaydı ya da kısa bir liste için dosya yeterlidir ve daha basittir. Veri büyüyüp karmaşıklaştıkça ve aynı veriyi birden çok program kullanmaya başladıkça dosyanın sınırları ortaya çıkar. Tablo 11.1 iki yaklaşımı altı açıdan karşılaştırır.

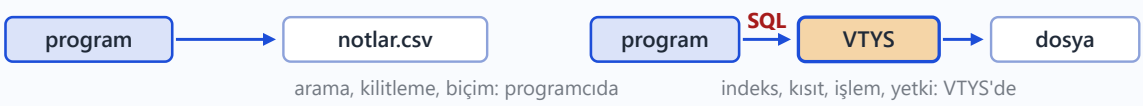
Özellik	Düz dosya (txt, csv)	Veri tabanı
Arama	Dosya baştan sona okunur	İndeksle hızlı; tek satırlık SQL sorgusu
Güncelleme	Çoğu zaman bütün dosya yeniden yazılır	Yalnız ilgili satır değişir
Eş zamanlı erişim	İki program aynı anda yazarsa veri bozulabilir	Kilitleme ve işlemlerle güvenli
Tutarlılık	Biçim denetimi tamamen programcıya kalır	Veri tipleri ve kısıtlar (`NOT NULL` , `UNIQUE` , `CHECK`)
Güvenlik	Dosya izni: ya hepsi ya hiçbiri	Kullanıcı, tablo ve alan düzeyinde yetki
Hata sonrası	Yarım kalan yazma dosyayı bozabilir	Yarım kalan işlem geri alınır

Tablo 11.1: Düz dosya ile veri tabanının karşılaştırılması.

Tablodaki farkların her biri somut bir soruna karşılık gelir. Bir okulun öğrenci listesi CSV dosyasındaysa 20.000 satır arasından tek bir öğrenciyi bulmak için dosya satır satır okunur. Bir öğrencinin telefonunu değiştirmek için genellikle dosyanın tamamı okunur, bellekte değiştirilir ve yeniden yazılır. Çünkü metin dosyasının ortasındaki bir satır uzarsa arkasındaki bütün baytlar kayar.

En ciddi sorun eş zamanlı erişimdir. İki öğretmen aynı anda not girerse ikisi de dosyayı okur, kendi değişikliğini yapar ve dosyayı geri yazar. İkinci yazan, birincinin değişikliğini fark etmeden ezer. VTYS bu durumu kilitlerle ve işlemlerle yönetir. Yazma sırasında elektrik kesilirse dosya yarım kalabilir; veri tabanı ise yarım kalan işlemi açılışta geri alır.

Tutarlılık açısından da fark büyüktür. CSV dosyasının bir hücresine yaş yerine "yirmi" yazılmasını hiçbir şey engellemez; her okuyan program bu hatayı kendisi denetlemek zorundadır. Veri tabanında kurallar tablonun tanımına yazılır ve kurala uymayan kayıt baştan reddedilir. Şekil 11.2 temel yapı farkını özetler: dosyada program dosyaya doğrudan erişir, veri tabanında ise arada VTYS vardır.



Şekil 11.2: Dosyada arama, kilitleme ve biçim denetimi programa kalır; veri tabanında program SQL ile VTYS'ye başvurur, indeks, kilit, işlem, yetki ve tip denetimini VTYS yapar.

11.1.3 Temel özellikler

Veri tabanlarını dosyalardan üstün kılan dört temel özellik vardır.

Verilere hızlı erişim. Veri tabanı, sık aranan sütunlar için indeks adı verilen ek yapılar tutar. İndeks sayesinde milyonlarca satır arasından aranan kayıt, bütün satırlara bakılmadan kısa sürede bulunur.

Alan düzeyinde güvenlik. Dosyada yetki genellikle bütün dosya içindir: dosyayı okuyabilen her şeyi okur. VTYS ise yetkiyi tablo, hatta sütun (alan) düzeyinde verebilir. Örneğin bir şirket veri tabanında maaş sütununu yalnız muhasebe birimi görebilir; öteki birimler aynı tablonun ad ve telefon sütunlarını okur.

Platform bağımsız, standart arayüzler. SQL bir standarttır; temel komutları bütün ilişkisel veri tabanlarında aynıdır. Java tarafında da JDBC adlı tek bir arayüz kümesi vardır. Aynı Java kodu, sürücü değiştirilerek farklı veri tabanlarına erişir.

Dosya yapısından bağımsızlık. Program, verinin diskte hangi dosyada, hangi sırayla, hangi biçimde tutulduğunu bilmez. Programın görevi ne istediğini SQL ile söylemektir; verinin nasıl bulunacağına VTYS karar verir (Şekil 11.3). Bu ayrım sayesinde VTYS bir indeks ekleyip verinin yerleşimini değiştirdiğinde program kodu değişmez.



Şekil 11.3: Program SQL ile *ne* istediğini söyler; VTYS indeksleri ve disk dosyalarını kullanarak *nasıl* bulacağına karar verir.

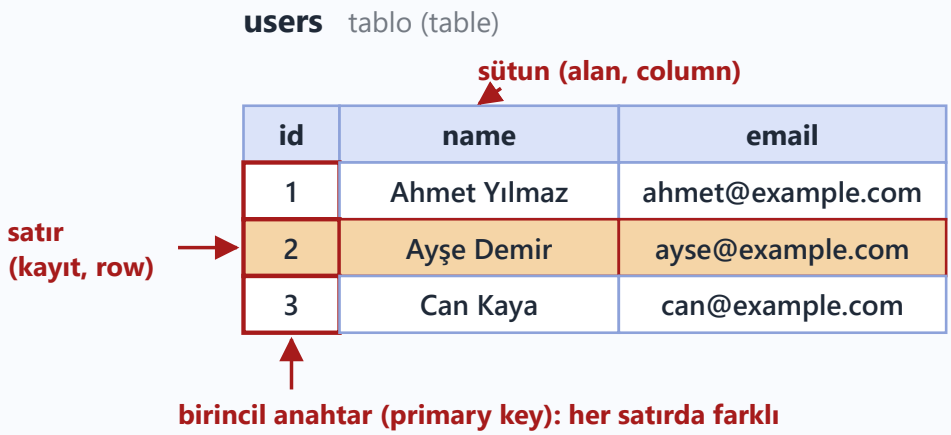
Bu dört özelliğe iki önemli özellik daha eklenir. Birincisi veri bütünlüğü kısıtlarıdır: "bu sütun boş olamaz", "bu değer tekrar edemez", "yaş negatif olamaz" gibi kurallar tabloya yazılır ve VTYS bunlara uymayan kaydı reddeder. İkincisi işlem desteğidir: birden çok değişiklik ya birlikte uygulanır ya hiç uygulanmaz.

11.1.4 İlişkisel model

Bugün en yaygın veri tabanı türü ilişkisel veri tabanıdır. Adı, verinin matematikteki "ilişki" (*relation*) kavramına dayanan tablolarla tutulmasından gelir.

Tanım 11.2

İlişkisel model (*relational model*), veriyi satır ve sütunlardan oluşan tablolarda tutar. **Sütun** verinin bir özelliğidir; adı ve veri tipi vardır. **Satır** (kayıt) bir varlığın bütün bilgisidir. **Birincil anahtar** (*primary key*), tablodaki her satırı tekil olarak tanımlayan sütundur; iki satırın birincil anahtarı aynı olamaz.



Şekil 11.4: `users` tablosu. Sütunlar `id`, `name` ve `email`'dir; her satır bir kullanıcıdır; `id` birincil anahtardır.

Şekil 11.4 bölüm boyunca kullanılan `users` tablosunu gösterir. Tablonun üç sütunu vardır. `name` sütununun tipi metindir (`TEXT`), `id` sütununun tipi tam sayıdır (`INTEGER`). İşaretli ikinci satır, Ayşe Demir adlı kullanıcının bütün bilgisidir: kimliği 2, e-postası `ayse@example.com`.

Birincil anahtar neden gerekir? İki kullanıcının adı aynı olabilir; iki "Ahmet Yılmaz" kaydı ayırt edilemezse hangisinin güncelleneceği belli olmaz. Ad ve soyad birlikte bile tekil değildir. Bu yüzden her satıra yalnız onu tanımlayan bir numara verilir. Bu numara değişmez ve başka bir satıra verilmez. Birincil anahtar başka tablolardan bu satırı göstermek için de kullanılır.

Java ile benzerlik kurmak yararlıdır. **Tablo bir sınıfa**, **sütun bir alana**, **satır bir nesneye** karşılık gelir. `users` tablosu, `int id`, `String name` ve `String email` alanlarını taşıyan bir `User` sınıfına benzer; her satır bu sınıfın bir nesnesidir. Benzerlik tam değildir: tablo yalnız veri tutar, metotları yoktur. Ama satırları nesnelere dönüştürürken bu eşleme doğrudan kullanılır.

> **Sık yapılan hata**
>
> Satırla sütunu karıştırmak. Sütun bir özelliktir ve bütün kayıtlarda bulunur (`email`). Satır bir kayıttır ve bütün özellikleri taşır (Ayşe Demir'in kimliği, adı ve e-postası). "Tabloya yeni bir öğrenci eklemek" yeni bir satır eklemektir; "öğrencilere doğum tarihi eklemek" ise yeni bir sütun eklemektir.

11.1.5 Veri tabanı türleri

Veri tabanları veriyi düzenleme biçimine göre gruplanır. Üç grup bu ders için önemlidir.

İlişkisel veri tabanları veriyi tablolarda tutar ve SQL ile sorgulanır. Oracle, SQL Server, MySQL ve PostgreSQL bu gruptadır. Bu sistemler bir sunucu olarak çalışır; birçok istemci ağ üzerinden aynı veri tabanına bağlanır. Kullanımın büyük bölümü, yaklaşık yüzde 77'si, bu gruptadır.

NoSQL ve arama sistemleri tablo kullanmaz. Veriyi belge (JSON benzeri kayıtlar), anahtar–değer çiftleri ya da arama dizini olarak tutar. MongoDB bir belge veri tabanı, Elasticsearch bir arama motorudur. Belirli senaryolarda, örneğin yapısı sık değişen ya da çok büyük hacimli veride kullanılır. Kullanım payı yaklaşık yüzde 23'tür (Şekil 11.5).



Şekil 11.5: Yaklaşık kullanım payı: ilişkisel veri tabanları yüzde 77, NoSQL ve arama sistemleri yüzde 23.

Gömülü ilişkisel veri tabanları ayrı bir sunucu olarak değil, programın içinde bir kütüphane olarak çalışır. En bilineni SQLite'tır. Veri paylaşımının az olduğu, tek bir programın kendi verisini sakladığı durumlarda yaygındır: telefon uygulamaları, tarayıcılar, gömülü cihazlar.

Bu bölümün örneklerinde kurulumu en kolay olan SQLite kullanılır. Bir sunucu kurmak, kullanıcı tanımlamak ya da ağ ayarı yapmak gerekmez. SQL komutları ve JDBC kodu öteki ilişkisel veri tabanlarında da neredeyse aynıdır. SQLite ile öğrenilen bilgi MySQL ya da PostgreSQL'e doğrudan taşınır; değişen yalnız bağlantı adresi ve sürücüdür.

11.2 SQLite

11.2.1 SQLite nedir?

SQLite, C diliyle yazılmış, açık kaynaklı ve ücretsiz bir ilişkisel veri tabanıdır. Onu öteki ilişkisel sistemlerden ayıran özellikleri şunlardır.

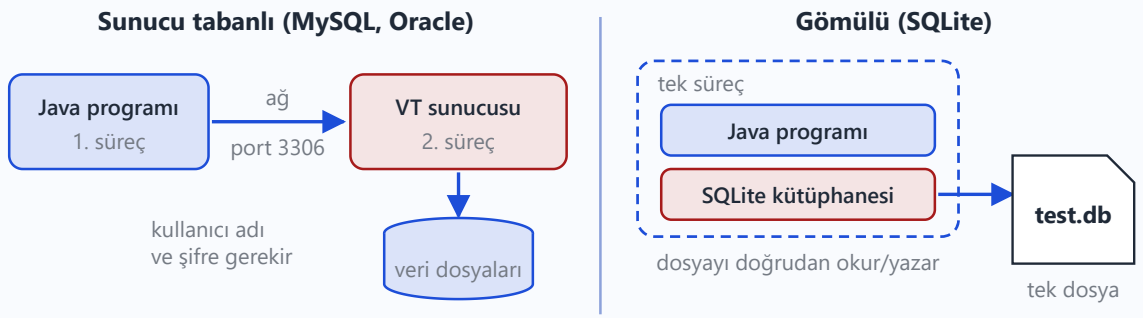
- **Gömülüdür** (*embedded*): programın içinde bir kütüphane olarak çalışır. Java programı SQLite'ı çağırdığında ayrı bir süreç başlamaz; veri tabanı kodu programın kendi sürecinde yürür.
- **Sunucusuzdur** (*serverless*): ayrı bir sunucu programı, kurulum ya da veri tabanı yöneticisi gerekmez.
- **Kendine yeter**: bütün tablolar, indeksler ve şema tek bir disk dosyasında saklanır, örneğin `sample.db`. Veri tabanını yedeklemek ya da başka makineye taşımak bu dosyayı kopyalamaktır.
- **Çapraz platformdur**: aynı `.db` dosyası Windows, Linux, macOS ve Android'de açılır.

SQLite dünyada en yaygın kullanılan gömülü ilişkisel veri tabanıdır. Her Android telefonda, her büyük tarayıcıda ve birçok masaüstü programında bulunur. Yaygın kullanım alanları mobil uygulamalar, gömülü sistemler, küçük ölçekli uygulamalar ve tarayıcı ya da e-posta istemcisi gibi masaüstü programlarıdır. Elektrik-elektronik mühendisliği açısından gömülü sistemler özellikle önemlidir: bir ölçüm cihazı, okuduğu değerleri bir SQLite dosyasında saklayıp daha sonra sorgulayabilir.

SQLite'ın uygun olmadığı yer de bellidir: ağ üzerinden çok sayıda kullanıcının aynı anda yazdığı büyük sistemler. SQLite aynı anda yalnız bir yazıcıya izin verir. Bir e-ticaret sitesinin binlerce eş zamanlı siparişi için MySQL ya da PostgreSQL gibi sunucu tabanlı bir sistem gerekir.

11.2.2 Sunucu tabanlı ve gömülü veri tabanı

İki mimarinin farkı Şekil 11.6 ile görülür. Sunucu tabanlı bir veri tabanında iki süreç vardır: Java programı ve VT sunucusu. Program sunucuya ağ üzerinden bağlanır; MySQL'in varsayılan kapısı (*port*) 3306'dır. Bağlanırken kullanıcı adı ve şifre verilir. Veriyi diskteki dosyalara sunucu yazar; program bu dosyaları hiç görmez.



Şekil 11.6: Sunucu tabanlı veri tabanında program ağ üzerinden ayrı bir sunucu sürecine bağlanır; SQLite ise programın içinde çalışır ve tek bir dosyaya doğrudan yazar.

Gömülü veri tabanında tek bir süreç vardır. SQLite kütüphanesi Java programının içinde yer alır ve `.db` dosyasını doğrudan okuyup yazar. Ağ, kullanıcı adı ve şifre yoktur. Tablo 11.2 farkları özetler.

	Sunucu tabanlı	**SQLite**
Kurulum	Sunucu kurulur, kullanıcı ve şifre tanımlanır	Projeye tek bir jar eklenir
Veri	Sunucunun yönettiği birçok dosya	Tek bir `.db` dosyası
Eş zamanlı yazma	Çok kullanıcı	Aynı anda tek yazıcı

Tablo 11.2: Sunucu tabanlı veri tabanı ile SQLite.

11.2.3 Sürücü, DB Browser ve çalıştırma

Java'nın SQLite ile konuşabilmesi için bir **JDBC sürücüsü** (*driver*) gerekir. SQLite için yaygın sürücü Xerial'in `sqlite-jdbc` kütüphanesidir (github.com/xerial/sqlite-jdbc). Sürücü tek bir jar dosyasıdır, örneğin `sqlite-jdbc-3.40.0.0.jar`. Bu jar hem Java sınıflarını hem de SQLite'ın her işletim sistemi için derlenmiş kütüphanesini içerir. Bu yüzden ayrıca SQLite kurmak gerekmez.

Veri tabanı dosyasının içini görmek için **DB Browser for SQLite** (sqlitebrowser.org) kullanılabilir. Bu program bir `.db` dosyasını açar, tabloları ve satırları görsel olarak gösterir ve SQL komutu çalıştırır. Java programının yaptığı değişiklikleri denetlemek için çok yararlıdır. Program bir satır ekledikten sonra DB Browser'da tablo yenilenir ve satırın gerçekten eklendiği görülür.

Eclipse'te sürücü projeye şöyle eklenir. Jar dosyası projede bir `lib` klasörüne konur. Projeye sağ tıklanıp *Build Path → Configure Build Path...* seçilir. *Libraries* sekmesinde *Classpath* seçilir, *Add JARs...* ile jar eklenir ve *Apply and Close* ile kapatılır. Komut satırından derleme ve çalıştırma ise şöyledir:

```
javac Baglan.java
java -cp ".;sqlite-jdbc-3.40.0.0.jar" Baglan
```

İlk satır programı derler. Derlemek için sürücü gerekmez, çünkü programın kullandığı `java.sql` paketi JDK ile gelir. İkinci satır programı çalıştırır. `-cp` seçeneği sınıf yolunu (*classpath*) verir: nokta, çalışma klasöründeki `Baglan.class` için; jar, sürücü için. Windows'ta yol öğeleri noktalı virgülle (`;`), Linux ve macOS'ta iki noktayla (`:`) ayrılır. Sürücü yalnız çalışma zamanında gerekir; jar sınıf yolunda yoksa program derlenir ama bağlanırken hata verir. Bu durumda alınan ileti "No suitable driver found ..." biçimindedir.

11.3 SQL temelleri

Java'dan veri tabanına bağlanmadan önce veri tabanının dilini tanımak gerekir. Java programı veri tabanına SQL komutlarını metin olarak gönderir; veri tabanı bu metni çözümler ve çalıştırır. Bu kesimdeki komutlar DB Browser for SQLite'ta da, bir Java programının içinde de aynen çalışır.

11.3.1 Yaygın SQL komutları

> **Tanım 11.3**
>
> **SQL** (*Structured Query Language*, yapılandırılmış sorgu dili), ilişkisel veri tabanlarının ortak dilidir. Veri tabanından veri isteyen SQL komutuna **sorgu** (*query*) denir.

SQL, Java'dan farklı olarak bildirimsel (*declarative*) bir dildir. Java'da bir dizide arama yapmak için döngü yazılır: hangi öğeye hangi sırayla bakılacağı adım adım söylenir. SQL'de yalnız istenen sonuç tarif edilir: "puanı 70'ten büyük oyuncuları getir." Sonucun nasıl üretileceğine VTYS karar verir. Tablo 11.3 en sık kullanılan beş komutu verir.

Komut	Görevi	Örnek	CRUD
`CREATE TABLE`	Yeni tablo oluşturur	`CREATE TABLE users (id INTEGER, name TEXT)`	—
`INSERT INTO`	Yeni kayıt ekler	`INSERT INTO users(name) VALUES('Ali')`	**C**reate
`SELECT`	Tabloyu sorgular	`SELECT * FROM users`	**R**ead
`UPDATE`	Var olan kayıtları günceller	`UPDATE users SET name = 'Efe' WHERE id = 1`	**U**pdate
`DELETE`	Kayıtları siler	`DELETE FROM users WHERE id = 1`	**D**elete

Tablo 11.3: Yaygın SQL komutları ve CRUD işlemleri.

Kalıcı veriyle yapılan dört temel işlem **CRUD** diye anılır. Ad, işlemlerin İngilizce baş harflerinden gelir: oluşturma (*create*), okuma (*read*), güncelleme (*update*) ve silme (*delete*). Hemen her veri tabanı uygulaması bu dört işlemin birleşimidir. Şekil 11.7 her harfin karşılık geldiği SQL komutunu gösterir. `CREATE TABLE` CRUD'un parçası değildir; veriyi değil, tablonun yapısını oluşturur.



Şekil 11.7: CRUD: tabloya ekleme (`INSERT`), okuma (`SELECT`), güncelleme (`UPDATE`) ve silme (`DELETE`).

Yazım kuralları kısadır. SQL anahtar sözcüklerinde büyük-küçük harf fark etmez: `select` ile `SELECT` aynıdır. Okunabilirlik için anahtar sözcükler büyük, tablo ve sütun adları küçük yazılır. Metin değerler tek tırnak içine alınır: `'Ali'`. Çift tırnak SQL'de değer için değil, ad için kullanılır; bu yüzden Java dizgisinin içindeki SQL'de metinler tek tırnakla yazılır ve iki dil birbirine karışmaz. `SELECT *` içindeki yıldız "bütün sütunlar" demektir. `WHERE` ise komutun hangi satırlara uygulanacağını seçer.

11.3.2 `CREATE TABLE`, veri tipleri ve kısıtlar

Bir tablo, adı ve sütunlarıyla tanımlanır. Her sütunun bir adı, bir tipi ve isteğe bağlı kısıtları vardır. Aşağıdaki komut bir üye tablosu oluşturur:

```
CREATE TABLE IF NOT EXISTS uyeler (
  id    INTEGER PRIMARY KEY,   -- benzersiz
  name  TEXT NOT NULL,         -- boş olamaz
  email TEXT UNIQUE,           -- tekrar edemez
  yas   INTEGER CHECK (yas >= 0)
)
```

Komutun her parçasının bir görevi vardır. `IF NOT EXISTS`, tablo zaten varsa hata verilmemesini sağlar. Program her açılışta bu komutu çalıştırabilir; ilk açılışta tablo oluşur, sonrakilerde komut hiçbir şey yapmaz. `--` ile başlayan kısımlar SQL'de satır sonuna kadar süren açıklamalardır, Java'daki `//` gibi.

`INTEGER PRIMARY KEY` SQLite'ta özel bir anlam taşır. Bu sütun satırın iç numarasıdır (`rowid`). Ekleme sırasında değer verilmezse SQLite tablodaki en büyük `id`'nin bir fazlasını atar; tablo boşsa 1 verir. Böylece programın numara üretmesi gerekmez.

Kısıtlar (*constraint*) tabloya yazılan kurallardır. `NOT NULL` sütunun boş bırakılamayacağını, `UNIQUE` değerin iki satırda tekrar edemeyeceğini, `CHECK` ise verilen koşulun her satırda doğru olması gerektiğini söyler. Kısıt bozulursa komut reddedilir ve tablo değişmez. Aşağıdaki deneme bu tabloya dokuz ekleme gönderir; her reddedilen ekleme için JDBC'nin verdiği iletiyi yazar:

```
kabul: INSERT INTO uyeler(name, email, yas) VALUES ('Ada', 'ada@x', 20)
[SQLITE_CONSTRAINT_NOTNULL] A NOT NULL constraint failed (NOT NULL constraint failed:
uyeler.name)
[SQLITE_CONSTRAINT_CHECK] A CHECK constraint failed (CHECK constraint failed: yas >= 0)
[SQLITE_CONSTRAINT_UNIQUE] A UNIQUE constraint failed (UNIQUE constraint failed:
uyeler.email)
[SQLITE_CONSTRAINT_PRIMARYKEY] A PRIMARY KEY constraint failed (UNIQUE constraint failed:
uyeler.id)
kabul: INSERT INTO uyeler(name) VALUES ('Naz')
kabul: INSERT INTO uyeler(id, name) VALUES (10, 'Ece')
kabul: INSERT INTO uyeler(name) VALUES ('Nur')
kabul: INSERT INTO uyeler(name, yas) VALUES ('Ali', 'yirmi')
1 Ada ada@x 20 integer
2 Naz null null null
10 Ece null null null
11 Nur null null null
12 Ali null yirmi text
```

İletilerin her biri bir kısıta karşılık gelir. Adı verilmeyen satır `NOT NULL constraint failed: uyeler.name`, yaşı −3 olan satır `CHECK constraint failed: yas >= 0` ile reddedilir. Ada'nın e-postasını ikinci kez kullanan satır `UNIQUE` kısıtına, `id`'si 1 olan ikinci satır birincil anahtara takılır. İletinin başındaki `[SQLITE_CONSTRAINT_…]` kısmı sürücünün eklediği hata kodudur; parantez içindeki asıl ileti SQLite'ın kendisinden gelir.

Çıktının son satırları `INTEGER PRIMARY KEY`'in davranışını gösterir. Naz 2 numarasını alır. Ece'ye elle 10 verilince bir sonraki satır (Nur) 11 alır: en büyük `id` + 1. Arada kalan 3–9 numaraları boş kalır; bu bir sorun değildir, çünkü birincil anahtarın ardışık olması gerekmez, yalnız tekil olması gerekir.

Son satır SQLite'a özgü bir inceliği ortaya koyar: `INTEGER` sütununa `'yirmi'` metni yazılabilmiştir. SQLite esnek tiplidir; sütun tipi bir öneridir (*type affinity*). Değer sayıya çevrilebiliyorsa sayı olarak, çevrilemiyorsa metin olarak saklanır. `CHECK (yas >= 0)` de bu satırı durdurmamıştır, çünkü SQLite karşılaştırmasında her metin her sayıdan büyük sayılır. MySQL ve PostgreSQL böyle bir değeri reddeder. Bu yüzden Java tarafında değerin tipi doğru verilmeli ve sayılar `setInt` gibi tipli metotlarla gönderilmelidir. Tablo 11.4 SQLite tiplerinin Java karşılıklarını verir.

SQLite tipi	Java karşılığı	Okuma metodu
`INTEGER`	`int`, `long`	`getInt`, `getLong`
`REAL`	`double`	`getDouble`
`TEXT`	`String`	`getString`
`BLOB`	`byte[]`	`getBytes`
`NULL`	değer yok	`getString` → `null`

Tablo 11.4: SQLite veri tipleri, Java karşılıkları ve okuma metotları.

`BLOB` (*binary large object*) ham bayt dizisidir; bir resim ya da bir ölçüm kaydı bu tiple saklanabilir. `NULL` ise bir tip değil, değerin olmadığını gösteren özel bir işarettir. Karşılaştırmalarda nasıl davrandığı ayrıca önemlidir.

11.3.3 SQL komutları adım adım

Beş temel komutun tabloyu nasıl değiştirdiği, boş bir veri tabanında sırayla çalıştırılarak izlenebilir. İlk komut tabloyu kurar:

```
CREATE TABLE users (id INTEGER PRIMARY KEY,
                    name TEXT, email TEXT)
```

Bu komuttan sonra tablonun üç sütunu vardır ama hiç satırı yoktur. Ardından iki ekleme gelir:

```
INSERT INTO users(name, email)
  VALUES('Ahmet Yılmaz', 'ahmet@example.com')
```

```
INSERT INTO users(name, email)
  VALUES('Ayşe Demir', 'ayse@example.com')
```

Eklemelerde `id` verilmemiştir; yalnız `name` ve `email` sütunları adlarıyla anılır ve değerleri aynı sırayla `VALUES` içinde verilir. SQLite birinci satıra 1, ikinci satıra en büyük `id` + 1 = 2 atar. Her ekleme bir satırı etkiler. Sonra birinci satır güncellenir ve silinir; son olarak tablo okunur:

```
UPDATE users SET name = 'Mehmet Akdeniz'
  WHERE id = 1
```

```
DELETE FROM users WHERE id = 1
```

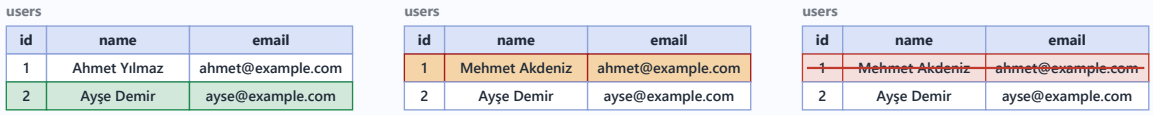
```
SELECT * FROM users
```

Komutlar bir Java programıyla tek tek çalıştırılıp her birinin ardından tablo yazdırıldığında Tablo 11.5 elde edilir. “Etkilenen satır” sütunu, JDBC'nin her komut için bildirdiği sayıdır.

Komut	Etkilenen satır	Tablonun durumu
`CREATE TABLE users …`	0	(boş)
`INSERT … 'Ahmet Yılmaz'`	1	(1, Ahmet Yılmaz, ahmet@example.com)

Komut	Etkilenen satır	Tablonun durumu
`INSERT … 'Ayşe Demir'`	1	(1, Ahmet Yılmaz, ahmet@example.com) (2, Ayşe Demir, ayse@example.com)
`UPDATE … WHERE id = 1`	1	(1, **Mehmet Akdeniz**, ahmet@example.com) (2, Ayşe Demir, ayse@example.com)
`DELETE … WHERE id = 1`	1	(2, Ayşe Demir, ayse@example.com)
`SELECT * FROM users`	—	değişmez; sonuç: (2, Ayşe Demir, ayse@example.com)

Tablo 11.5: `users` tablosunun her komuttan sonraki durumu (JDK 21 ve SQLite JDBC ile çalıştırıldı).



Şekil 11.8: İkinci eklemeden sonra (sol), güncellemeden sonra (orta; yalnız `name` değişti) ve silmeden sonra (sağ; 1 numaralı satırın üstü çizili).

Güncellemede yalnız `SET` ile anılan sütun değişir; `email` aynı kalır (Şekil 11.8). `WHERE id = 1` yalnız birinci satırı seçer. Silme de aynı koşulla yalnız birinci satırı kaldırır. Son `SELECT` tabloyu değiştirmez, yalnız okur ve tek satır döndürür. Silinen 1 numarası Ayşe'ye geçmez: Ayşe'nin `id`'si 2 olarak kalır. Birincil anahtar bir kez verildikten sonra değişmez.

Sık yapılan hata

`UPDATE` ya da `DELETE`'te `WHERE`'i unutmak. `UPDATE users SET name = 'X'` hata vermez; tablodaki *bütün* satırların adını değiştirir. `DELETE FROM users` bütün satırları siler, tablonun kendisi kalır. Üç satırlı bir tabloda iki komut da 3 döndürür. Veri tabanı bunu bir hata olarak görmez; geri almanın tek yolu bir işlem içinde çalışmak ya da yedek almaktır.

11.3.4 `SELECT`: süzme, sıralama, sınırlama

`SELECT`, SQL'in en çok kullanılan ve en zengin komutudur. Bu kesimdeki örnekler beş satırlık bir skor tablosuyla çalışır (Şekil 11.9).

skorlar

oyuncu	puan
ada	72
can	95
ece	58
efe	95
naz	81

Şekil 11.9: `skorlar` tablosu.

Bir sorgu en fazla yedi yan tümceden oluşur; bunların beşi burada kullanılır. `SELECT` hangi sütunların getirileceğini, `FROM` hangi tablodan okunacağını söyler. `WHERE` satırları bir koşula göre süzer. Koşulda `=`, `<>` (eşit değil), `<`, `>=` gibi karşılaştırmalar ve `AND`, `OR` bağlaçları kullanılır. `LIKE 'e%'` "e ile başlayan" anlamına gelir: `%` herhangi sayıda karakterin, `_` tek karakterin yerini tutar. `ORDER BY` sonucu sıralar; `DESC` büyükten küçüğe, varsayılan `ASC` küçükten büyüğe demektir. `LIMIT n` ilk `n` satırı alır. Tablo 11.6 dört sorguyu ve sonuçlarını verir.

Sorgu	Sonuç
`SELECT oyuncu, puan FROM skorlar WHERE puan < 90 ORDER BY puan DESC`	naz 81, ada 72, ece 58
`SELECT oyuncu, puan FROM skorlar ORDER BY puan DESC, oyuncu LIMIT 3`	can 95, efe 95, naz 81
`SELECT oyuncu, puan FROM skorlar WHERE oyuncu LIKE 'e%'`	ece 58, efe 95
`SELECT COUNT(*), AVG(puan), MAX(puan) FROM skorlar WHERE puan >= 60`	4 · 85.75 · 95

Tablo 11.6: `skorlar` tablosu üzerinde dört sorgu (sonuçlar çalıştırılarak alındı).

Birinci sorgu 90'dan küçük puanları seçer; can ve efe (95) elenir. Kalan üç satır puana göre büyükten küçüğe dizilir. İkinci sorguda süzme yoktur. Sıralama iki ölçütlüdür: önce puan azalan, puanlar eşitse oyuncu adı artan sırada. can ile efe 95'te eşittir; alfabede can önce gelir. `LIMIT 3` ilk üç satırı alır. İkinci ölçüt yazılmasaydı eşit puanlıların sırası belirsiz olurdu; sonuç VTYS'nin o anki yerleşimine bağlı kalırdı.

Üçüncü sorgu e ile başlayan iki adı döndürür. SQLite'ta `LIKE` ASCII harflerde büyük-küçük harf ayrımı yapmaz; `LIKE 'E%'` de aynı iki satırı verir. Dördüncü sorgu tek satır döndürür, çünkü `COUNT`, `AVG` ve `MAX` birer **toplama fonksiyonudur** (*aggregate function*): bir satır kümesinden tek bir değer üretir. 60 ve üstü dört puan vardır: 72, 95, 95 ve 81. Toplamları 343, ortalamaları 343 / 4 = 85,75'tir. En büyüğü 95'tir. `SUM` ve `MIN` da aynı ailedendir.

Bu fonksiyonların boş kümedeki davranışı akılda tutulmalıdır. Hiçbir satırın seçilmediği `WHERE puan > 100` ile `COUNT(*)` 0, `AVG(puan)` ve `SUM(puan)` ise `NULL` döndürür. Sorgu yine de tek bir satır döndürür; satırın değerleri boştur.

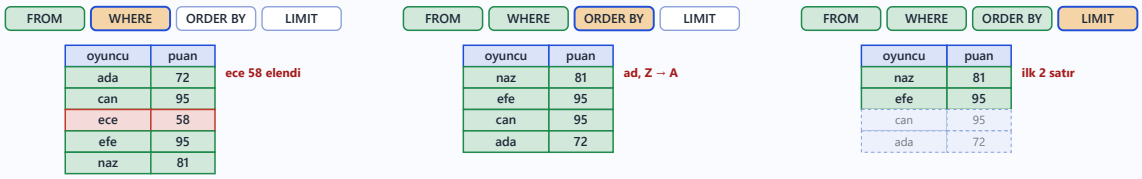
11.3.5 Kod izleme: bir `SELECT` sorgusu

SQL komutu yazılış sırasıyla çalışmaz. Aşağıdaki Java kodu bir sorguyu veri tabanına gönderir ve sonucu satır satır yazar. Burada önemli olan JDBC türleri değil, sorgunun hangi aşamalardan geçtiğidir.

```
String sql = "SELECT oyuncu, puan FROM skorlar "
           + "WHERE puan > 70 "
           + "ORDER BY oyuncu DESC LIMIT 2";
try (Statement st = conn.createStatement();
     ResultSet rs = st.executeQuery(sql)) {
  while (rs.next()) {
    System.out.println(rs.getString("oyuncu") + " "
                       + rs.getInt("puan"));
  }
}
```

```
naz 81
efe 95
```

Veri tabanı önce `FROM`'u işler: tablonun beş satırını alır. Sonra `WHERE puan > 70` koşulunu her satıra uygular ve ece'yi (58) eler; dört satır kalır. Ardından `ORDER BY oyuncu DESC` satırları ada göre ters alfabetik sıraya dizer: naz, efe, can, ada. Sıralama puana göre değil, ada göredir. En son `LIMIT 2` ilk iki satırı alır. Sonuç kümesi iki satırdır: (naz, 81) ve (efe, 95). Şekil 11.10 bu aşamaları gösterir.



Şekil 11.10: Sorgunun aşamaları. `WHERE` ece'yi eler (sol), `ORDER BY` adları ters sıraya dizer (orta), `LIMIT 2` ilk iki satırı sonuç kümesi yapar (sağ).

Sonuç Java'ya bir `ResultSet` nesnesi olarak gelir. Bu nesnenin bir **imleci** (*cursor*) vardır ve imleç başlangıçta ilk satırın *önünde* durur. Her `rs.next()` çağrısı imleci bir satır ilerletir; satır varsa `true`, kalmadıysa `false` döndürür. Bu yüzden döngü `while (rs.next())` biçimindedir: ilk çağrı imleci birinci satıra taşır, gövde o satırın sütunlarını okur. Tablo 11.7 on adımı verir.

Adım	Satır	Değişkenler	Yeni çıktı
1	1: `String sql = "SELECT oyuncu, puan FROM skorlar "` 2: `+ "WHERE puan > 70 "` 3: `+ "ORDER BY oyuncu DESC LIMIT 2";`	sql=SELECT … LIMIT 2	—
2	5: `ResultSet rs = st.executeQuery(sql)) {`	aşama=FROM	—
3	5: `ResultSet rs = st.executeQuery(sql)) {`	aşama=WHERE	—

Adım	Satır	Değişkenler	Yeni çıktı
4	5: `ResultSet rs = st.executeQuery(sql)) {`	aşama=ORDER BY	—
5	5: `ResultSet rs = st.executeQuery(sql)) {`	aşama=LIMIT	—
6	6: `while (rs.next()) {`	rs.next()=true	—
7	7: `System.out.println(rs.getString("oyuncu") + " "` 8: `+ rs.getInt("puan"));`	oyuncu=naz, puan=81	`naz 81`
8	6: `while (rs.next()) {`	rs.next()=true	—
9	7: `System.out.println(rs.getString("oyuncu") + " "` 8: `+ rs.getInt("puan"));`	oyuncu=efe, puan=95	`efe 95`
10	6: `while (rs.next()) {` 10: `}`	rs.next()=false	—

Tablo 11.7: `SELECT` sorgusunun izlenmesi. Adım 2–5 veri tabanının içinde, adım 6–10 Java döngüsünde geçer.

Mantıksal çalışma sırası bu örnekte görülür: `FROM` → `WHERE` → `ORDER BY` → `LIMIT`. `SELECT`'teki sütun seçimi sona doğru yapılır. Bu sıra gruplama sorgularında daha da önem kazanır. Bir sorgunun ne döndüreceğini kestirmek için onu yazılış sırasıyla değil, bu sırayla okumak gerekir.

Örnek 11.1

`skorlar` tablosunda `SELECT oyuncu FROM skorlar WHERE puan > 90 OR oyuncu = 'ece'` ve `SELECT oyuncu FROM skorlar LIMIT 2 OFFSET 1` sorguları ne döndürür?

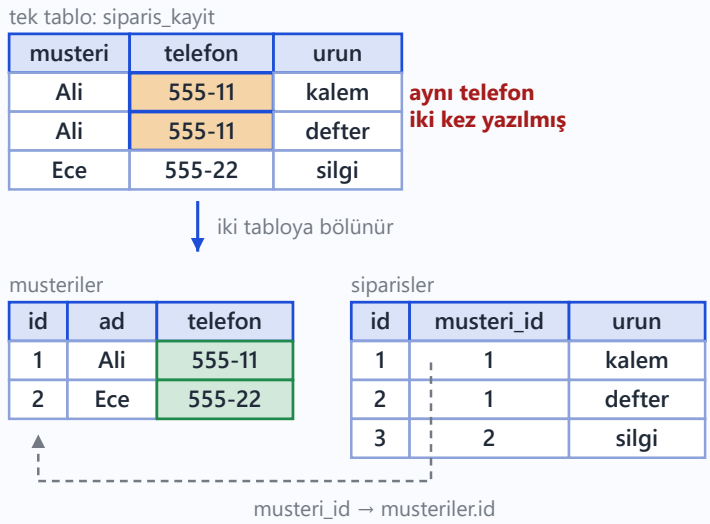
Birincisinde koşul her satır için ayrı değerlendirilir. can ve efe 95 puanla birinci kısmı, ece ikinci kısmı sağlar. ada (72) ve naz (81) iki kısmı da sağlamaz. Sonuç **can, ece, efe**'dir. Sıralama istenmediği için SQLite satırları eklenme sırasıyla verir. İkincisinde `OFFSET 1` ilk satırı atlar, `LIMIT 2` sonraki iki satırı alır: **can, ece**. Çalıştırılan sorgular bu sonuçları verdi. `ORDER BY` olmadan satır sırası standartça güvence altında değildir; sıra önemliyse her zaman `ORDER BY` yazılır.

11.4 İlişkiler ve gruplama

Gerçek bir uygulamada veri tek bir tabloya sığmaz. Öğrenciler, bölümler, dersler, siparişler ayrı tablolarda durur ve birbirine bağlanır. Bu kesim tabloların neden bölündüğünü, nasıl bağlandığını ve bağlı tabloların tek sorguda nasıl birleştirildiğini anlatır.

11.4.1 Normalleştirme

Bir mağazanın siparişleri tek bir tabloda tutulsun: her satırda müşterinin adı, telefonu ve sipariş edilen ürün. Ali üç kez sipariş verirse adı ve telefonu üç satırda tekrarlanır (Şekil 11.11, üstte). Bu tekrar yalnız yer israfı değildir; asıl sorun tutarsızlık riskidir.



Şekil 11.11: Tekrarlı tek tablo iki tabloya bölünür: `musteriler` ve `siparisler`. Siparişler müşteriye `musteri_id` ile bağlanır.

Tekrar üç tür bozulmaya, yani **anomaliye**, yol açar. **Güncelleme anomalisi**: Ali'nin telefonu değişince her satırı ayrı ayrı düzeltmek gerekir; biri unutulursa Ali'nin iki farklı telefonu olur ve hangisinin doğru olduğu bilinemez. **Ekleme anomalisi**: henüz siparişi olmayan bir müşteri kaydedilemez, çünkü her satır bir siparişitir. **Silme anomalisi**: Ece'nin tek siparişi silinince Ece'nin telefonu da kaybolur.

> **Tanım 11.4**
>
> **Normalleştirme** (*normalization*), tekrarı önlemek için veriyi her bilgi tek bir yerde duracak biçimde tablolara bölmektir. Bölünen tablolar bir anahtarla birbirine bağlanır.

Şeklin alt kısmında müşteri bilgisi `musteriler` tablosunda bir kez durur. `siparisler` tablosu müşterinin adını ve telefonunu değil, yalnız numarasını (`musteri_id`) tutar. Ali'nin telefonu değişince tek bir hücre güncellenir. Siparişi olmayan müşteri `musteriler`'e eklenebilir. Ece'nin siparişi silinse de Ece'nin kaydı kalır. Üç anomali birden ortadan kalkar.

Normalleştirmenin ilk kuralına **birinci normal biçim** denir: her hücrede tek bir değer bulunur. Bir müşterinin iki telefonu `'555-11, 555-33'` gibi tek hücrede bir liste olarak yazılmaz. Böyle bir hücrede "telefonu 555-33 olan müşteri" aramak zorlaşır, bir numarayı silmek metin işlemeye dönüşür. Telefonlar ya ayrı satırlara ya da ayrı bir `telefonlar(musteri_id, numara)` tablosuna taşınır. Daha ileri normal biçimler de vardır; bu ders için ilk kural ve "her bilgi tek yerde" ilkesi yeterlidir.

11.4.2 Yabancı anahtar

Bölünen tabloları bağlayan sütunun özel bir adı vardır.

Tanım 11.5

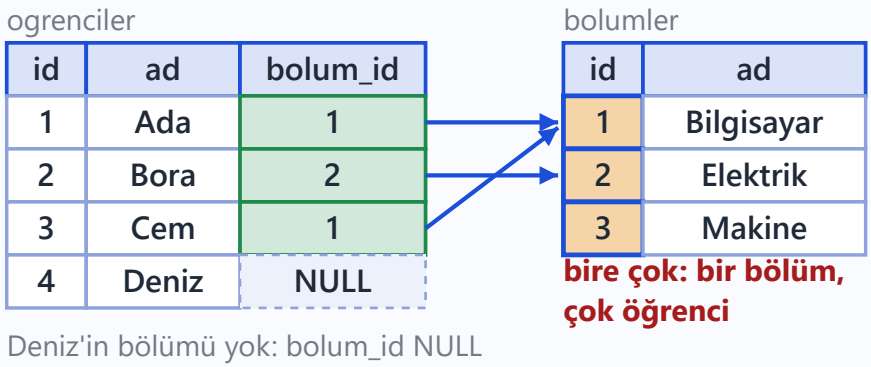
Yabancı anahtar (*foreign key*), başka bir tablonun birincil anahtarını gösteren sütundur. Yabancı anahtarın değeri ya o tablodaki bir satırın anahtarına eşittir ya da `NULL` 'dur.

Bölümün ilişki örnekleri iki tabloyla çalışır: `bolumler` ve `ogrenciler`. Her öğrencinin bir bölümü vardır ve bu bağ `bolum_id` sütunuyla kurulur:

```
CREATE TABLE bolumler (
  id INTEGER PRIMARY KEY,
  ad TEXT NOT NULL);
CREATE TABLE ogrenciler (
  id INTEGER PRIMARY KEY,
  ad TEXT NOT NULL,
  bolum_id INTEGER REFERENCES bolumler(id));
```

`REFERENCES bolumler(id)`, `bolum_id` 'nin bir yabancı anahtar olduğunu bildirir. Tablolara şu satırlar eklenir; Deniz'in bölümü henüz belli değildir:

```
INSERT INTO bolumler(ad) VALUES ('Bilgisayar'), ('Elektrik'), ('Makine');
INSERT INTO ogrenciler(ad, bolum_id)
  VALUES ('Ada', 1), ('Bora', 2), ('Cem', 1), ('Deniz', NULL);
```



Şekil 11.12: `ogrenciler.bolum_id` sütunu `bolumler.id` sütununu gösterir. Bir bölümde çok öğrenci olabilir: bire çok ilişki. Deniz'in bölümü yoktur, `bolum_id` 'si `NULL` 'dur.

Şekil 11.12 ilişkiyi gösterir. Ada ve Cem'in `bolum_id` 'si 1'dir; ikisi de Bilgisayar bölümündedir. Bir bölümde birden çok öğrenci olabilir ama bir öğrencinin tek bölümü vardır. Bu yapıya **bire çok** (*one-to-many*) ilişki denir. Java'da karşılığı, her `Ogrenci` nesnesinde bir `Bolum` referansı bulunmasıdır. Tabloda nesne referansı yerine anahtar değeri durur.

Yabancı anahtarın asıl yararı bütünlük denetimidir. Var olmayan 9 numaralı bölüme öğrenci eklemek ya da öğrencisi olan bir bölümü silmek, bağı koparır: öğrenci hiçbir yeri göstermeyen bir numara taşır. VTYS bu iki işlemi reddeder. Ancak SQLite geriye uyumluluk için bu denetimi **varsayılan olarak yapmaz**. Denetim her bağlantıda ayrıca açılır:

```
PRAGMA foreign_keys = ON;
INSERT INTO ogrenciler(ad, bolum_id) VALUES ('Efe', 9);   -- ret
DELETE FROM bolumler WHERE id = 2;                        -- ret: Bora bağlı
DELETE FROM bolumler WHERE id = 3;                        -- kabul: Makine boş
```

Aynı işlemler önce varsayılan ayarla, sonra denetim açıkken çalıştırıldığında şu çıktı elde edilir:

```
== PRAGMA yok (varsayılan)
foreign_keys = 0
kabul (1): INSERT INTO ogrenciler(ad, bolum_id) VALUES ('Efe', 9)
kabul (1): DELETE FROM bolumler WHERE id = 2
== PRAGMA foreign_keys = ON
ret: INSERT INTO ogrenciler(ad, bolum_id) VALUES ('Efe', 9) ->
[SQLITE_CONSTRAINT_FOREIGNKEY] A foreign key constraint failed (FOREIGN KEY constraint
failed)
ret: DELETE FROM bolumler WHERE id = 2 -> [SQLITE_CONSTRAINT_FOREIGNKEY] A foreign key
constraint failed (FOREIGN KEY constraint failed)
kabul (1): DELETE FROM bolumler WHERE id = 3
kabul (1): INSERT INTO ogrenciler(ad, bolum_id) VALUES ('Nil', NULL)
```

Varsayılan ayarda `PRAGMA foreign_keys` 0'dır. 9 numaralı bölüme ekleme sessizce kabul edilir; Bora'nın bölümü de silinir ve Bora'nın `bolum_id`'si artık var olmayan bir satırı gösterir. Denetim açıkken iki işlem de `FOREIGN KEY constraint failed` ile reddedilir. Öğrencisi olmayan Makine bölümü ise silinebilir. `NULL` yabancı anahtar her zaman kabul edilir, çünkü "bölümü yok" anlamına gelir. Java'da bu ayar bağlantı açıldıktan hemen sonra `stmt.execute("PRAGMA foreign_keys = ON")` ile yapılır. Ayar bağlantıya özgüdür: her yeni bağlantıda yinelenmelidir.

Bire çok dışında bir ilişki türü daha sık görülür: **çoka çok** (*many-to-many*). Bir öğrenci birçok derse kayıtlıdır, bir derste birçok öğrenci vardır. Bu ilişki iki tablonun sütunlarıyla kurulamaz; üçüncü bir ara tablo gerekir:

```
CREATE TABLE kayitlar (
  ogrenci_id INTEGER REFERENCES ogrenciler(id),
  ders_id    INTEGER REFERENCES dersler(id),
  PRIMARY KEY (ogrenci_id, ders_id));
```

`kayitlar` tablosunun her satırı "şu öğrenci şu derse kayıtlı" bilgisidir. İki sütunun her biri ayrı bir tabloya yabancı anahtardır. Birincil anahtar iki sütunun birleşimidir; böylece aynı öğrenci aynı derse iki kez kaydedilemez.

11.4.3 `JOIN` ile birleştirme

Normalleştirmenin bir bedeli vardır: “Ada hangi bölümde?” sorusunun cevabı artık iki tablodadır. `ogrenciler`'de yalnız 1 sayısı, `bolumler`'de 1'in “Bilgisayar” olduğu yazar. İki bilgiyi tek sonuçta buluşturmak için tablolar birleştirilir.

> **Tanım 11.6**
>
> **Birleştirme** (`JOIN`), iki tablonun satırlarını `ON` koşuluna göre eşleyip tek bir sonuç tablosu üreten SQL işlemidir. Yalnız `JOIN` (ya da `INNER JOIN`) eşi bulunan satırları, `LEFT JOIN` ise soldaki tablonun bütün satırlarını verir.

```
SELECT o.ad AS ogrenci, b.ad AS bolum
FROM ogrenciler o JOIN bolumler b
  ON o.bolum_id = b.id
ORDER BY o.id
```

Sorguda `o` ve `b` tablo **takma adlarıdır**: `FROM ogrenciler o` , sorgunun geri kalanında `ogrenciler` yerine `o` yazılabileceğini söyler. Takma adlar burada zorunludur, çünkü iki tabloda da `ad` adlı bir sütun vardır. `o.ad` öğrencinin, `b.ad` bölümün adıdır. `AS` sonuç sütununa yeni bir ad verir; Java tarafında sütun bu adla okunur: `rs.getString("bolum")` . `ON o.bolum_id = b.id` eşleme koşuludur: bir öğrenci satırı, `id`'si kendi `bolum_id`'sine eşit olan bölüm satırıyla eşlenir.

Aynı tablolarla `LEFT JOIN` yazıldığında sorgu şöyle olur:

```
SELECT o.ad AS ogrenci, b.ad AS bolum
FROM ogrenciler o LEFT JOIN bolumler b
  ON o.bolum_id = b.id
ORDER BY o.id
```

İki sorgunun sonuçları Tablo 11.8'de yan yanadır. Fark yalnız Deniz'dedir. Deniz'in `bolum_id`'si `NULL` olduğu için hiçbir bölümle eşleşmez. `JOIN` eşi olmayan satırı sonuca almaz. `LEFT JOIN` ise soldaki tablonun (`ogrenciler`) her satırını korur; eşi yoksa sağ tablonun sütunlarını `NULL` ile doldurur. Java'da bu değer `getString` ile okununca `null` gelir.

`ogrenci`	`JOIN` (INNER)	`LEFT JOIN`
Ada	Bilgisayar	Bilgisayar
Bora	Elektrik	Elektrik
Cem	Bilgisayar	Bilgisayar
Deniz	satır yok	`NULL`

Tablo 11.8: `JOIN` ve `LEFT JOIN` sonuçları.

Hangisinin seçileceği soruya bağlıdır. “Bölümü olan öğrencileri bölüm adıyla listele” için `JOIN`, “bütün öğrencileri listele, bölümü varsa adını da yaz” için `LEFT JOIN` kullanılır. Tabloların sırası `LEFT JOIN`'de önemlidir: `bolumler b LEFT JOIN ogrenciler o` yazılırsa korunan taraf bölümler olur. Bu sorgu Makine bölümünü öğrencisi `NULL` olarak getirir.

İki hata sık yapılır. Birincisi takma ad yazmadan `SELECT ad` yazmaktır. SQLite hangi `ad`'ın kastedildiğini bilemez ve sorguyu `ambiguous column name: ad` iletisiyle reddeder. İkincisi `ON` koşulunu unutmaktır:

```
SELECT COUNT(*) FROM ogrenciler, bolumler      -- 12
```

Koşulsuz birleştirmede her satır her satırla eşleşir. Dört öğrenci ve üç bölümden 4 × 3 = 12 satır çıkar. Buna **Kartezyen çarpım** denir. Sonuç anlamsızdır: Ada hem Bilgisayar, hem Elektrik, hem Makine bölümünde görünür. Büyük tablolarda bu hata çok pahalıdır. 10.000 öğrenci ve 100 bölüm bir milyon satır üretir.

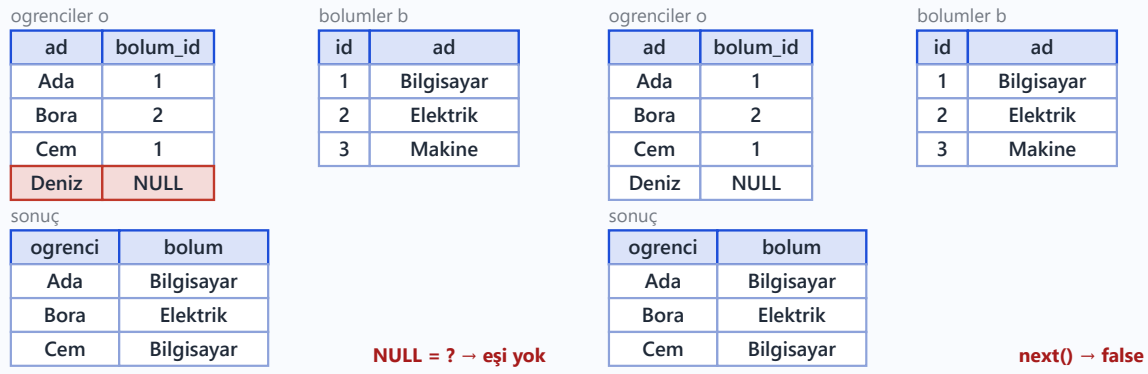
11.4.4 Kod izleme: `JOIN` sorgusu

Aşağıdaki kod birleştirme sorgusunu Java'dan çalıştırır. `stmt`, açık bir bağlantı üzerinde oluşturulmuş bir `Statement`'tır.

```
String sql = "SELECT o.ad AS ogrenci, b.ad AS bolum "
    + "FROM ogrenciler o JOIN bolumler b "
    + "ON o.bolum_id = b.id ORDER BY o.id";
try (ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    System.out.println(rs.getString("ogrenci") + " - "
        + rs.getString("bolum"));
  }
}
```

```
Ada - Bilgisayar
Bora - Elektrik
Cem - Bilgisayar
```

Veri tabanı `ogrenciler`'in her satırı için `bolumler`'de `id`'si eşleşen satırı arar. Ada'nın `bolum_id`'si 1'dir; 1 numaralı bölüm Bilgisayar'dır. Bora 2'ye, yani Elektrik'e bağlanır. Cem yine 1'e bağlanır; aynı bölüm satırı ikinci kez kullanılır. Bölüm tablosunda Bilgisayar bir kez yazılıdır ama sonuçta iki kez görünür. Deniz'in `bolum_id`'si `NULL`'dur ve `NULL` hiçbir `id`'ye eşit sayılmaz; Deniz sonuca girmez (Şekil 11.13).



Şekil 11.13: Deniz'in `NULL` bölümü hiçbir satırla eşleşmez (sol); sonuç kümesi üç satırdır (sağ).

Adım	Satır	Değişkenler	Yeni çıktı
1	1: `String sql = "SELECT o.ad AS ogrenci, b.ad AS bolum "` 2: `+ "FROM ogrenciler o JOIN bolumler b "` 3: `+ "ON o.bolum_id = b.id ORDER BY o.id";`	—	—
2	4: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
3	4: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
4	4: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
5	4: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
6	5: `while (rs.next()) {` 6: `System.out.println(rs.getString("ogrenci") + " - "` 7: `+ rs.getString("bolum"));`	satır=1	`Ada - Bilgisayar`
7	5: `while (rs.next()) {` 6: `System.out.println(rs.getString("ogrenci") + " - "` 7: `+ rs.getString("bolum"));`	satır=2	`Bora - Elektrik`
8	5: `while (rs.next()) {` 6: `System.out.println(rs.getString("ogrenci") + " - "` 7: `+ rs.getString("bolum"));`	satır=3	`Cem - Bilgisayar`
9	5: `while (rs.next()) {` 9: `}`	satır=yok	—

Tablo 11.9: `JOIN` sorgusunun izlenmesi.

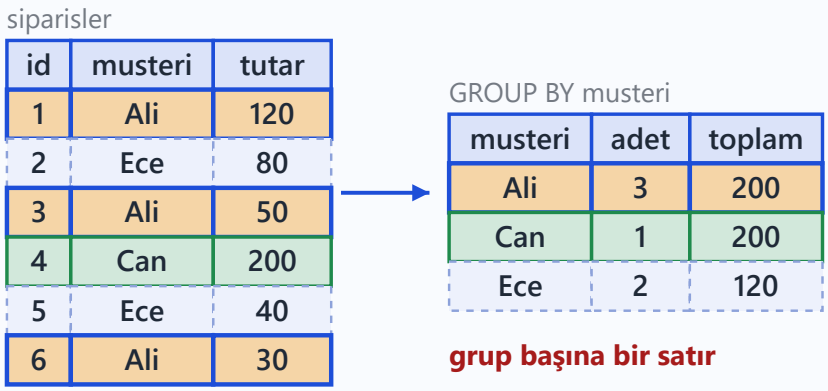
Java tarafı önceki örnekle aynıdır: imleç sonuç kümesinde satır satır ilerler. Sütunlar `AS` ile verilen adlarla okunur: `"ogrenci"` ve `"bolum"`. `rs.getString("ad")` yazılsaydı sonuçta `ad` adlı sütun olmadığı için sürücü hata verirdi. `LEFT JOIN` yazılsaydı dördüncü satır olarak `Deniz - null` yazılırdı; bu da çalıştırılarak doğrulandı.

11.4.5 `GROUP BY` ve `HAVING`

Toplama fonksiyonları bütün tabloya uygulandığında tek bir değer verir. Çoğu zaman ise her grup için ayrı bir değer istenir: her müşterinin sipariş sayısı, her bölümün öğrenci sayısı. Bu kesimin örnekleri altı satırlık bir sipariş tablosuyla çalışır:

```
INSERT INTO siparisler(musteri, tutar) VALUES
  ('Ali', 120), ('Ece', 80), ('Ali', 50),
  ('Can', 200), ('Ali', 30), ('Ece', 40);
```

`GROUP BY musteri`, `musteri` değeri aynı olan satırları bir grupta toplar. Toplama fonksiyonları her grup için ayrı hesaplanır ve sonuçta her grup tek bir satıra dönüşür (Şekil 11.14). Ali'nin üç siparişi vardır: 120 + 50 + 30 = 200. Can'ın tek siparişi 200, Ece'nin iki siparişi 80 + 40 = 120'dir.



Şekil 11.14: `siparisler` tablosu müşteriye göre üç gruba ayrılır; her grup için adet ve toplam hesaplanır.

Süzme gruplamadan önce ya da sonra yapılabilir. `WHERE` gruplamadan *önce* satırları süzer. `HAVING` ise gruplamadan *sonra* grupları süzer. Bu yüzden toplama fonksiyonu içeren bir koşul `WHERE`'e yazılamaz: `WHERE` çalışırken henüz grup yoktur, dolayısıyla `SUM(tutar)` da yoktur. `WHERE SUM(tutar) >= 150` yazılırsa SQLite `misuse of aggregate: SUM()` iletisiyle reddeder. Tablo 11.10 üç durumu karşılaştırır.

Sorgunun devamı	Sonuç
`GROUP BY musteri`	Ali (3, 200) · Can (1, 200) · Ece (2, 120)
`WHERE tutar >= 50 GROUP BY musteri`	Ali (2, 170) · Can (1, 200) · Ece (1, 80)
`GROUP BY musteri HAVING SUM(tutar) >= 150`	Ali (3, 200) · Can (1, 200)

Tablo 11.10: `SELECT musteri, COUNT(*), SUM(tutar) FROM siparisler …` sorgusunun üç biçimi.

İkinci satırda `WHERE tutar >= 50` önce tek tek siparişleri süzer: Ali'nin 30'luk ve Ece'nin 40'lık siparişi elenir. Gruplar kalan siparişlerden kurulur. Ali'nin adedi 2'ye, toplamı 170'e düşer. Üçüncü satırda bütün siparişler gruplanır, sonra toplamı 150'nin altında kalan Ece'nin grubu elenir. İki sorgu benzer görünür ama farklı soruları cevaplar: biri “50 TL ve üstü siparişleri müşteriye göre say”, öteki “toplamı 150 ve üstü olan müşterileri bul” der.

Bir sorgunun yan tümceleri mantıksal olarak şu sırayla çalışır:

> `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `ORDER BY` → `LIMIT`

Bu sıra birçok kuralı açıklar. `WHERE` toplama fonksiyonu kullanamaz, çünkü gruplamadan önce gelir. `ORDER BY` ise `SELECT`'te verilen takma adları kullanabilir, çünkü ondan sonra gelir.

Gruplama birleştirmeyle de kullanılır. Her bölümün öğrenci sayısı şöyle bulunur:

```
SELECT b.ad, COUNT(o.id)
FROM bolumler b LEFT JOIN ogrenciler o
  ON o.bolum_id = b.id
GROUP BY b.id
```

Sonuç Bilgisayar 2 · Elektrik 1 · Makine 0'dır. Burada iki ayrıntı önemlidir. `LEFT JOIN` sayesinde öğrencisi olmayan Makine de sonuca girer; düz `JOIN` yazılsaydı hiç görünmezdi. İkinci ayrıntı `COUNT(o.id)`'dir. `COUNT(sütun)` yalnız `NULL` olmayan değerleri sayar. Makine'nin tek birleşik satırında `o.id` `NULL`'dur ve sayılmaz; sonuç 0 olur. `COUNT(*)` ise satırları sayar. Yazılsaydı Makine'ye 1 verirdi; bu da çalıştırılarak doğrulandı.

Gruplamada bir tuzak daha vardır. `SELECT musteri, tutar FROM siparisler GROUP BY musteri` sorgusunda `tutar` ne gruplanmış ne de bir toplama fonksiyonunun içindedir. Ali'nin grubunda üç farklı tutar vardır; hangisi yazılacak? SQLite hata vermez ve gruptaki satırlardan birinin değerini getirir. Denemede Ali 120, Can 200, Ece 80 çıktı. PostgreSQL ve SQL Server gibi birçok veri tabanı bu sorguyu reddeder. Kural basittir: `SELECT`'teki her sütun ya `GROUP BY`'da bulunur ya da bir toplama fonksiyonunun içindedir.

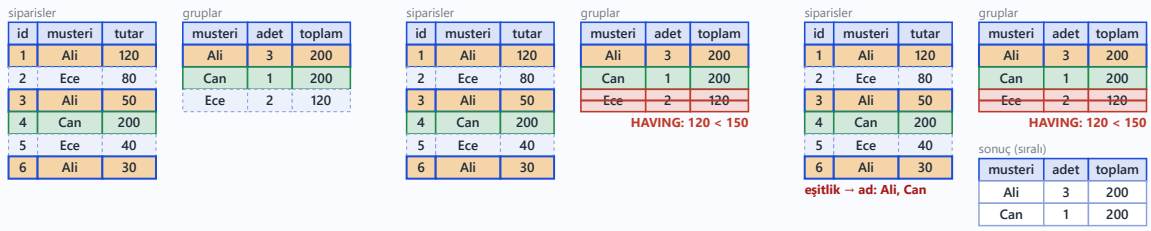
11.4.6 Kod izleme: `GROUP BY` sorgusu

Aşağıdaki kod gruplama, grup süzme ve iki ölçütlü sıralamayı tek sorguda birleştirir:

```
String sql = "SELECT musteri, COUNT(*) AS adet, "
    + "SUM(tutar) AS toplam FROM siparisler "
    + "GROUP BY musteri HAVING SUM(tutar) >= 150 "
    + "ORDER BY toplam DESC, musteri";
try (ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    System.out.println(rs.getString("musteri") + ": "
        + rs.getInt("adet") + " sipariş, "
        + rs.getInt("toplam") + " TL");
  }
}
```

```
Ali: 3 sipariş, 200 TL
Can: 1 sipariş, 200 TL
```

Veri tabanı önce altı satırı müşteriye göre üç gruba ayırır. Her grup için `COUNT(*)` ve `SUM(tutar)` hesaplanır: Ali 3 ve 200, Can 1 ve 200, Ece 2 ve 120. `HAVING SUM(tutar) >= 150` Ece'nin grubunu eler. `ORDER BY toplam DESC, musteri` kalan iki grubu toplama göre azalan sıraya dizer. Ali ile Can 200'de eşittir; ikinci ölçüt ad olduğu için Ali önce gelir (Şekil 11.15).



Şekil 11.15: Gruplar ve toplamları (sol), `HAVING` Ece'yi eler (orta), sıralanmış sonuç (sağ).

Adım	Satır	Değişkenler	Yeni çıktı
1	1: `String sql = "SELECT musteri, COUNT(*) AS adet, "` 2: `+ "SUM(tutar) AS toplam FROM siparisler "` 3: `+ "GROUP BY musteri HAVING SUM(tutar) >= 150 "` 4: `+ "ORDER BY toplam DESC, musteri";`	—	—
2	5: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
3	5: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
4	5: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
5	5: `try (ResultSet rs = stmt.executeQuery(sql)) {`	—	—
6	6: `while (rs.next()) {` 7: `System.out.println(rs.getString("musteri") + ": "` 8: `+ rs.getInt("adet") + " siparis, "` 9: `+ rs.getInt("toplam") + " TL");`	satır=1	`Ali: 3 siparis, 200 TL`
7	6: `while (rs.next()) {` 7: `System.out.println(rs.getString("musteri") + ": "` 8: `+ rs.getInt("adet") + " siparis, "` 9: `+ rs.getInt("toplam") + " TL");`	satır=2	`Can: 1 siparis, 200 TL`
8	6: `while (rs.next()) {` 11: `}`	satır=yok	—

Tablo 11.11: `GROUP BY` sorgusunun izlenmesi.

Java tarafında takma adlar (`adet`, `toplam`) sütun adı gibi okunur: `rs.getInt("adet")`. Sonuçta grup başına bir satır vardır; tek tek siparişler görünmez. Ali'nin üç siparişi tek bir "Ali: 3 sipariş, 200 TL" satırına dönüşmüştür.

11.4.7 `NULL`: değerin olmaması

Deniz'in bölümü bilinmiyor. Bu durum tabloda `NULL` ile gösterilir.

Tanım 11.7

SQL'de `NULL`, değerin bilinmediğini ya da olmadığını gösteren özel işarettir. `NULL` ne 0'dır ne de boş metindir (`''`). `NULL` ile yapılan her karşılaştırma "bilinmiyor" sonucunu verir.

SQL üç değerli bir mantık kullanır: doğru, yanlış ve bilinmiyor. `bolum_id = NULL` karşılaştırması Deniz için bile doğru değil, bilinmiyordur. Bilinmeyen bir değerin başka bir bilinmeyene eşit olup olmadığı da bilinemez. `WHERE` yalnız koşulu *doğru* olan satırları seçer; bu yüzden aşağıdaki ilk sorgu hiçbir satır döndürmez:

```
SELECT ad FROM ogrenciler WHERE bolum_id = NULL;      -- 0 satır
SELECT ad FROM ogrenciler WHERE bolum_id IS NULL;     -- Deniz
SELECT ad FROM ogrenciler WHERE bolum_id <> 1;        -- yalnız Bora
SELECT ad, COALESCE(bolum_id, 0) FROM ogrenciler;     -- Deniz 0
SELECT COUNT(*), COUNT(bolum_id) FROM ogrenciler;     -- 4 ve 3
```

Doğru yazım `IS NULL` ve `IS NOT NULL`'dur. Üçüncü satırdaki sonuç daha şaşırtıcıdır. `bolum_id <> 1` Deniz'i de seçmez, çünkü `NULL`'un 1'den farklı olduğu bile bilinmez. "Bilgisayar dışındaki öğrenciler" isteniyorsa `WHERE bolum_id <> 1 OR bolum_id IS NULL` yazılır.

`NULL` aritmetik işlemlere de yayılır: `NULL + 1`'in sonucu `NULL`'dur. `COALESCE(bolum_id, 0)`, ilk argüman `NULL` ise ikincisini döndürür; böylece `NULL` yerine 0 konur. Toplama fonksiyonları `NULL`'ları atlar. `COUNT(*)` satırları sayar ve 4 verir; `COUNT(bolum_id)` `NULL` olmayan değerleri sayar ve 3 verir. Hiç satır eşleşmezse `AVG` ve `SUM` `NULL` döndürür, `COUNT` 0 döndürür. Bu satırların hepsi bir Java programında çalıştırılarak doğrulandı.

Java tarafında da bir sorun vardır. `int` gibi temel tipler `null` olamaz. `rs.getInt` `NULL` bir sütun için 0 döndürür. Peki 0, gerçekten 0 mı yoksa `NULL` mı? Bunu ayırt etmek için okumanın hemen ardından `wasNull()` çağrılır:

```
String sql = "SELECT ad, bolum_id FROM ogrenciler "
    + "WHERE id >= 3";
try (ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    int b = rs.getInt("bolum_id");
    boolean bos = rs.wasNull();
    System.out.println(rs.getString("ad") + " " + b
        + (bos ? " (NULL)" : ""));
  }
}
```

```
Cem 1
Deniz 0 (NULL)
== geç wasNull
Cem 1
Deniz 0
getString: null, getObject: null
SELECT ad FROM ogrenciler WHERE bolum_id = NULL
  -> 0 satır:
SELECT ad FROM ogrenciler WHERE bolum_id IS NULL
  -> 1 satır: Deniz;
SELECT ad FROM ogrenciler WHERE bolum_id IS NOT NULL
  -> 3 satır: Ada; Bora; Cem;
SELECT ad FROM ogrenciler WHERE bolum_id <> 1
  -> 1 satır: Bora;
SELECT NULL + 1, NULL = NULL, NULL IS NULL, COALESCE(NULL, 0)
  -> 1 satır: null null 1 0;
SELECT ad, COALESCE(bolum_id, 0) FROM ogrenciler
  -> 4 satır: Ada 1; Bora 2; Cem 1; Deniz 0;
SELECT COUNT(*), COUNT(bolum_id) FROM ogrenciler
  -> 1 satır: 4 3;
SELECT AVG(bolum_id), SUM(bolum_id) FROM ogrenciler WHERE id > 10
  -> 1 satır: null null;
SELECT AVG(bolum_id), SUM(bolum_id) FROM ogrenciler
  -> 1 satır: 1.33333333333333 4;
SELECT ad FROM ogrenciler WHERE NOT (bolum_id = 1)
  -> 1 satır: Bora;
```

Çıktının ilk iki satırı sunumdaki programındır. Cem'in bölümü 1'dir ve `wasNull()` `false` döner. Deniz için `getInt` 0 verir, `wasNull()` `true` döner ve satıra "(NULL)" eklenir. `wasNull()` yalnız *son okunan* sütuna bakar. "geç wasNull" başlıklı ikinci denemede `bolum_id`'den sonra `getString("ad")` okunmuş, `wasNull()` ondan sonra çağrılmıştır. Bu kez ad sorulmuş olur; ad `NULL` olmadığı için Deniz'in satırına "(NULL)" yazılmaz. `getString` ve `getObject` ise `NULL` için doğrudan Java'nın `null`'unu döndürür; bu metotlarda `wasNull()` gerekmez. Çıktının kalan satırları yukarıdaki SQL sorgularının gerçek sonuçlarıdır.

Sık yapılan hata

`WHERE bolum_id = NULL` yazmak. Sorgu hata vermez, yalnız hiçbir satır döndürmez; program "bölümsüz öğrenci yok" sonucuna varır. `NULL` denetimi her zaman `IS NULL` ile yapılır. Java'daki karşılığı da benzerdir: `getInt`'in 0 döndürmesi değerin 0 olduğunu göstermez; `wasNull()` ilgili `getInt`'in hemen ardından çağrılır.

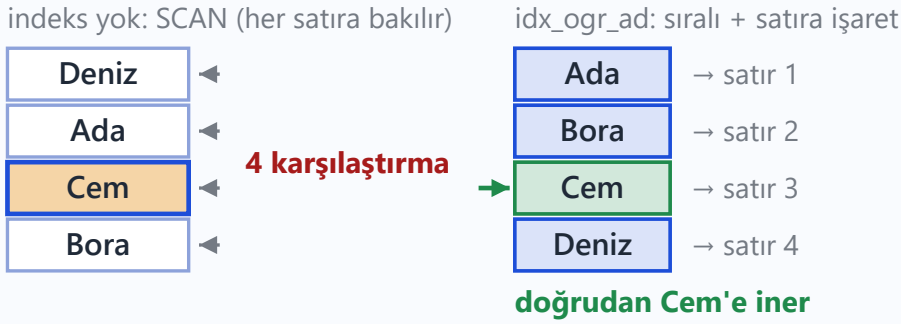
11.4.8 İndeks

Veri tabanının hızlı aramasının sırrı indekstir.

Tanım 11.8

İndeks (*database index*), bir sütunun değerlerini sıralı tutan ve her değerin bulunduğu satırı gösteren ek yapıdır. SQLite indeksi bir B-ağacı (*B-tree*) olarak saklar.

İndeks, kitabın sonundaki dizine benzer. "Kalıtım" kelimesinin geçtiği sayfalar, kitap baştan sona okunmadan dizinden bulunur. İndeks olmadan `WHERE ad = 'Cem'` sorgusu tablonun her satırına bakar; buna **tarama** (*scan*) denir. İndeks varken sıralı yapıda ikili aramaya benzer bir arama yapılır ve doğrudan Cem'in satırına gidilir (Şekil 11.16). *n* satırlı bir tabloda tarama *n* karşılaştırma, B-ağacında arama ise yaklaşık log *n* adım ister. Bir milyon satırda bu fark bir milyon ile yirmi civarı arasındadır.



Şekil 11.16: İndekssiz aramada bütün satırlar taranır (4 karşılaştırma); indeks sıralı olduğu için aranan ada doğrudan gidilir.

İndeks `CREATE INDEX` ile oluşturulur. SQLite'ın `EXPLAIN QUERY PLAN` komutu, bir sorgunun nasıl yürütüleceğini, yani tablonun taranıp taranmayacağını ya da indeksin kullanılıp kullanılmayacağını gösterir:

```
CREATE INDEX idx_ogr_ad ON ogrenciler(ad);
EXPLAIN QUERY PLAN
  SELECT * FROM ogrenciler WHERE ad = 'Cem';
```

Tablo 11.12 dört durumun planlarını verir. Planlar SQLite 3.40.0 ile alındı.

Durum	`EXPLAIN QUERY PLAN` çıktısı
İndeks yokken, `WHERE ad = 'Cem'`	`SCAN ogrenciler`
İndeks varken, `WHERE ad = 'Cem'`	`SEARCH ogrenciler USING INDEX idx_ogr_ad (ad=?)`
`WHERE id = 3`	`SEARCH ogrenciler USING INTEGER PRIMARY KEY (rowid=?)`
`WHERE ad LIKE '%em'`	`SCAN ogrenciler`

Tablo 11.12: `EXPLAIN QUERY PLAN` çıktıları.

`SCAN` tablonun taranacağını, `SEARCH` bir indeksle arama yapılacağını gösterir. Birincil anahtar için ayrıca indeks oluşturmak gerekmez; `INTEGER PRIMARY KEY` zaten tablonun kendi sıralama anahtarıdır. `UNIQUE` sütunların indeksi de kendiliğinden oluşur, çünkü VTYS tekrarı denetlemek için bu indekse ihtiyaç duyar.

Son satır indeksin sınırını gösterir. `LIKE '%em'` "em ile biten" demektir. İndeks adları baştan sıralı tutar; sonu "em" olan adlar dizinin her yerine dağılmıştır. Bu yüzden baştaki joker indeksi kullanamaz. Denemede `LIKE 'C%'` de `SCAN` üretti. SQLite'ta `LIKE` büyük-küçük harf ayırmaz, sıradan indeks ise ayırır. Bu yüzden önek araması da varsayılan ayarlarla indeksi kullanmadı.

İndeksin bir bedeli vardır. Her `INSERT`, `UPDATE` ve `DELETE` yalnız tabloyu değil, indeksi de günceller; indeks diskte de yer kaplar. Bu yüzden her sütuna indeks konmaz. İndeks, sık aranan ve `WHERE`, `JOIN` ya da `ORDER BY`'da kullanılan sütunlara konur. Etkinin büyüklüğünü görmek için 200.000 satırlık bir tabloda `ad` sütununa göre 200 arama yapıldı. İndekssiz aramalar 633 ms, indeks oluşturulduktan sonra aynı aramalar 1 ms sürdü. Süreler makineye ve çalıştırmaya göre değişir; oran ise her seferinde yüzlerce kattır.

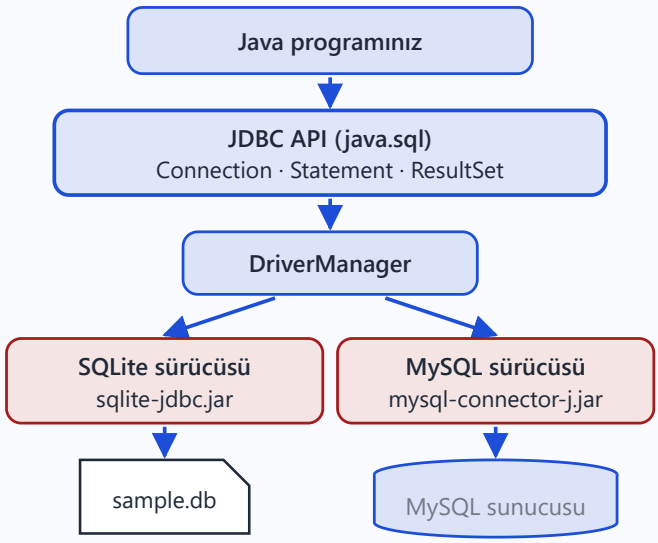
11.5 JDBC

SQL komutları veri tabanının dilidir; Java programının bu komutları veri tabanına iletmesi, sonuçları da geri alması gerekir. Bu köprüyü JDBC kurar.

11.5.1 JDBC nedir?

> **Tanım 11.9**
>
> **JDBC** (*Java Database Connectivity*), Java'nın veri tabanlarına bağlanmak için sunduğu standart API'dir (*application programming interface*, uygulama programlama arayüzü). `java.sql` paketinde bulunur ve JDK ile gelir. Belirli bir veri tabanıyla konuşan kodu ise o veri tabanının **JDBC sürücüsü** sağlar.

JDBC'nin yapısı, nesne yönelimli programlamanın arayüz kavramının güzel bir uygulamasıdır. `java.sql` paketindeki `Connection`, `Statement` ve `ResultSet` birer arayüzdür; yalnız hangi metotların bulunacağını söyler. Bu arayüzleri her veri tabanı üreticisinin sürücüsü kendi sınıflarıyla uygular. SQLite sürücüsü SQLite dosyasıyla, MySQL sürücüsü MySQL sunucusuyla nasıl konuşulacağını bilir (Şekil 11.17).



Şekil 11.17: JDBC katmanları. Program yalnız JDBC arayüzlerini kullanır; `DriverManager` uygun sürücüyü seçer; sürücü SQLite dosyasına ya da MySQL sunucusuna bağlanır.

Program yalnız arayüzleri bildiği için veri tabanını değiştirmek çoğu zaman yalnız sürücü jar'ını ve bağlantı adresini değiştirmektir. Burada çok biçimlilik iş başındadır. Programdaki değişkenin tipi `Connection`'dır. Çalışma zamanında bu değişkenin gösterdiği gerçek nesnenin sınıfı SQLite sürücüsündeki `org.sqlite.jdbc4.JDBC4Connection`'dır. Bu ad `conn.getClass().getName()` ile yazdırılarak doğrulandı. Program `conn.createStatement()` çağırdığında JVM, `Connection` arayüzünün bu sınıftaki gerçeklemesini çalıştırır. Aynı satır MySQL'de MySQL sürücüsünün gerçeklemesini çalıştırır.

Arayüzün nesnesi doğrudan oluşturulamaz. `new Connection()` yazılırsa derleyici reddeder:

```
Ana.java:4: error: Connection is abstract; cannot be instantiated
    Connection conn = new Connection();
                      ^
1 error
```

Bağlantı nesnesini sürücünün sınıfından `new` ile kurmak da mümkündür ama yanlış olur, çünkü program bir sürücüye bağımlı hâle gelir. Bağlantı her zaman `DriverManager.getConnection` ile alınır; hangi sınıfın nesnesinin döneceğine sürücü karar verir.

11.5.2 JDBC'nin temel türleri

Bir JDBC programı birkaç türle yazılır. Tablo 11.13 bunları ve önemli metotlarını verir.

Tür	Görevi	Önemli metotlar
`DriverManager`	Uygun sürücüyü bulur, bağlantı açar	`getConnection(url)`, `getConnection(url, kullanici, sifre)`

Tür	Görevi	Önemli metotlar
`Connection`	Veri tabanıyla açık bir oturum	`createStatement()`, `prepareStatement(sql)`, `commit()`, `rollback()`, `close()`
`Statement`	Sabit bir SQL metnini çalıştırır	`execute`, `executeUpdate`, `executeQuery`
`PreparedStatement`	`?` yer tutuculu, önceden hazırlanan SQL	`setInt`, `setString`, `executeUpdate()`, `executeQuery()`
`ResultSet`	Sorgu sonucundaki satırlar ve bir imleç	`next()`, `getInt("id")`, `getString("name")`
`SQLException`	Veri tabanı hatası; kontrol edilen istisna	`getMessage()`, `getErrorCode()`

Tablo 11.13: JDBC'nin temel türleri. Hepsi `java.sql` paketindedir.

Türler bir zincir oluşturur. `DriverManager` bir `Connection` verir. `Connection` bir `Statement` ya da `PreparedStatement` üretir. Bu nesne bir sorgu çalıştırınca `ResultSet` döner. Her halka bir öncekinden elde edilir; programcı yalnız `DriverManager`'ı doğrudan çağırır.

Bu türlerin kullanılabilmesi için dosyanın başına `import java.sql.*;` yazılır. Unutulursa derleyici türleri bulamaz:

```
Ana.java:3: error: cannot find symbol
    Connection conn = DriverManager.getConnection("jdbc:sqlite:sample.db");
    ^
  symbol:   class Connection
  location: class Ana
Ana.java:3: error: cannot find symbol
    Connection conn = DriverManager.getConnection("jdbc:sqlite:sample.db");
                      ^
  symbol:   variable DriverManager
  location: class Ana
2 errors
```

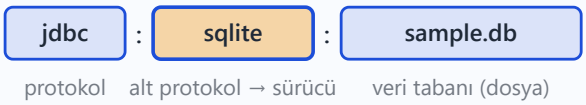
`SQLException` kontrol edilen (*checked*) bir istisnadır. JDBC metotlarının neredeyse hepsi onu fırlatabileceğini bildirir. Bu yüzden JDBC kodu ya bir `try-catch` içinde durur ya da metot `throws SQLException` der. İkisi de yapılmazsa program derlenmez:

```
Ana.java:4: error: unreported exception SQLException; must be caught or declared to be
thrown
    Connection conn = DriverManager.getConnection("jdbc:sqlite:sample.db");
                                                  ^
1 error
```

`Connection`, `Statement` ve `ResultSet` veri tabanında kaynak tutar: açık bir dosya, kilitler, bellek. Üçü de `AutoCloseable` arayüzünü uygular ve işleri bitince kapatılmalıdır. En güvenli yol try-with-resources'tır.

11.5.3 Bağlantı adresi

`DriverManager` hangi veri tabanına bağlanılacağını bir metin adresinden anlar. Bu adrese **JDBC URL**'si denir (Şekil 11.18).



Şekil 11.18: JDBC bağlantı adresinin parçaları: protokol (`jdbc`), alt protokol (sürücüyü seçer) ve veri tabanının dosyası ya da adresi.

Adresin biçimi `jdbc:<alt protokol>:<veri tabanı dosyası ya da adresi>`'dir. İlk parça her zaman `jdbc`'dir. İkinci parça sürücüyü belirler: `sqlite`, `mysql`, `postgresql`. `DriverManager` kayıtlı sürücülere adresi kabul edip etmediklerini sırayla sorar. Adresi kabul eden ilk sürücü bağlantıyı açar. Üçüncü parçanın biçimi sürücüye özgüdür. Tablo 11.14 örnekleri verir.

Bağlantı adresi (URL)	Anlamı
`jdbc:sqlite:sample.db`	Çalışma klasöründeki `sample.db`; yoksa boş olarak oluşturulur
`jdbc:sqlite:C:/veriler/okul.db`	Tam yol ile dosya
`jdbc:sqlite::memory:`	Bellekte geçici veri tabanı; bağlantı kapanınca silinir
`jdbc:mysql://localhost:3306/testdb`	MySQL sunucusu: makine, kapı, veri tabanı adı; ayrıca kullanıcı adı ve şifre

Tablo 11.14: Bağlantı adresi örnekleri.

Bellekteki veri tabanı denemeler için çok kullanışlıdır: diskte dosya bırakmaz ve her çalıştırmada temiz başlar. Bu notun birçok örneği `jdbc:sqlite::memory:` ile çalıştırıldı. Ancak bellekteki veri tabanı bağlantıya özgüdür. Bir bağlantıda oluşturulan tablo, aynı adresle açılan ikinci bir bağlantıda yoktur. Denemede ikinci bağlantı `no such table: t` iletisini aldı.

Göreli bir dosya yolu çalışma klasörüne göre çözülür. Eclipse'te çalışma klasörü proje klasörüdür; komut satırında ise `java` komutunun verildiği klasördür. Bu yüzden program iki yerden çalıştırılınca iki ayrı `sample.db` oluşabilir. SQLite olmayan bir dosyayı oluşturur ama olmayan bir klasörü oluşturmaz. `jdbc:sqlite:alt/klasor/okul.db` adresinde `alt` klasörü yoksa bağlantı "path … does not exist" iletisiyle başarısız olur.

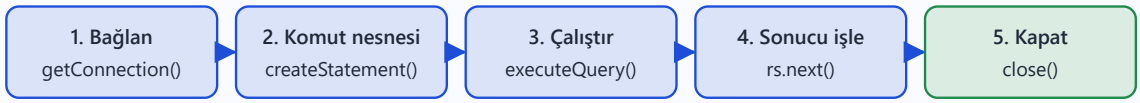
Eski kitaplarda ve kodlarda bağlantıdan önce şu satır görülür: `Class.forName("org.sqlite.JDBC");`. Bu satır sürücü sınıfını yükleyip `DriverManager`'a kaydettirirdi. JDBC 4.0'dan (Java 6) beri sürücü, sınıf yolundaki jar'dan kendiliğinden yüklenir; satır artık gerekmez. Yazılırsa zarar da vermez. Sürücü sınıf yolunda yoksa ne olur? Program sürücü jar'ı olmadan çalıştırıldığında şu çıktı alındı:

```
No suitable driver found for jdbc:sqlite:sample.db
ClassNotFoundException: org.sqlite.JDBC
```

İlk satır `getConnection`'ın fırlattığı `SQLException`'ın iletisidir: adresi kabul eden sürücü bulunamamıştır. Aynı ileti adreste yazım hatası olduğunda da gelir. Örneğin `jdbc:sqlit:okul.db` adresi sürücü yüklüyken bile bu iletiyi verir, çünkü `sqlit` alt protokolünü tanıyan sürücü yoktur. İkinci satır, eski yöntemle sürücü sınıfı aranırken alınan `ClassNotFoundException`'dır.

11.5.4 Bir JDBC programının adımları

Her JDBC programı aynı beş adımı izler (Şekil 11.19): bağlantı açılır, bir komut nesnesi oluşturulur, SQL çalıştırılır, sonuç işlenir ve kaynaklar kapatılır.



Şekil 11.19: JDBC programının beş adımı: bağlan, komut nesnesi oluştur, çalıştır, sonucu işle, kapat.

```
try (Connection conn = DriverManager.getConnection(url); // 1
     Statement stmt = conn.createStatement();             // 2
     ResultSet rs = stmt.executeQuery(sql)) {             // 3
  while (rs.next()) {                                     // 4
    System.out.println(rs.getInt(1));
  }
} // 5: rs, stmt, conn ters sırada otomatik kapanır
```

Kodun yapısı adımlarla birebir örtüşür. Birinci, ikinci ve üçüncü adım try-with-resources parantezinde sırayla yapılır: bağlantı açılır, bağlantıdan bir `Statement` alınır, `Statement` sorguyu çalıştırıp bir `ResultSet` döndürür. Dördüncü adım `while (rs.next())` döngüsüdür. Beşinci adım için kod yazılmaz: blok bitince, hata olsa bile, kaynaklar açılışın ters sırasıyla kapanır. Önce `rs`, sonra `stmt`, en son `conn`.

Ters sıra rastgele değildir. `ResultSet` `Statement`'a, `Statement` da `Connection`'a bağlıdır. Bağlantı önce kapansaydı ötekiler havada kalırdı. Kapanışı doğrulamak için kaynaklar bloğun dışındaki değişkenlere kopyalandı ve blok bittikten sonra `isClosed()` ile sorgulandı:

```
7
8
true true true
The database has been closed
```

Üç kaynak da kapalıdır (`true true true`). Kapalı `Statement` ile sorgu çalıştırma denemesi bir `SQLException` fırlattı; iletisi son satırdadır. Kapatılmayan bağlantılar SQLite dosyasını kilitli bırakabilir. Başka bir program, hatta aynı programın başka bir bağlantısı dosyaya yazamaz. Uzun süre çalışan programlarda kapatılmayan nesneler bellek sızıntısına da yol açar.

Kodda bir ayrıntı daha vardır: `rs.getInt(1)`. Sütunlar adlarıyla okunabildiği gibi sıra numarasıyla da okunabilir. JDBC'de sütun numaraları **1'den** başlar. Java dizilerinde 0'dan başlamaya alışkın bir programcı için bu sık bir hata kaynağıdır.

Sık yapılan hata

try-with-resources'ta açılan kaynağı bloğun dışında kullanmaya çalışmak. Parantezde tanımlanan değişkenin kapsamı yalnız bloktur; blok bittiğinde kaynak hem kapanmış hem de görünmez olmuştur. Derleyici dışarıdaki kullanımı reddeder:

```
Ana.java:6: error: cannot find symbol
    conn.close();
    ^
  symbol:   variable conn
  location: class Ana
1 error
```

11.6 Temel işlemler

Bu kesimdeki altı küçük program, `sample.db` adlı aynı dosya üzerinde sırayla çalışır: bağlanma, tablo oluşturma, ekleme, okuma, güncelleme, silme. Programlar bağımsızdır; her biri kendi bağlantısını açar ve kapatır. Veri, programlar arasında dosyada kalıcı olarak durur. Her program tam hâliyle `public static void main` içinde ve `import java.sql.*;` ile çalıştırıldı; notta yalnız `main`'in gövdesi verilir.

11.6.1 SQLite'a bağlanma

```
// Veri tabanı dosyasının yolu (bağlantı adresi)
String url = "jdbc:sqlite:sample.db";
try (Connection conn = DriverManager.getConnection(url)) {
  // Bağlantı başarılıysa mesaj yazdırılır.
  if (conn != null) {
    System.out.println("Bağlantı başarılı!");
  }
} catch (SQLException e) {
  // Bağlantı hatası olursa hata mesajı yazdırılır.
  System.out.println(e.getMessage());
}
```

```
Bağlantı başarılı!
```

Program bağlantı adresini bir değişkende tutar ve `getConnection` ile bağlantı açar. Dosya yoksa SQLite onu oluşturur; ilk çalıştırmadan önce olmayan `sample.db`, çalıştırmadan sonra diskte durur. İlginç bir ayrıntı: oluşan dosyanın boyu 0 bayttır. SQLite dosyaya ilk tablo yazılana kadar içerik koymaz. Bu, bir `File` nesnesiyle dosyanın varlığı ve boyu sorgulanarak görüldü.

`if (conn != null)` denetimi yalnız ek bir güvencedir. `getConnection` başarısız olursa `null` döndürmez, `SQLException` fırlatır. Bu yüzden `try` bloğunun içine gelindiyse bağlantı kesinlikle açıktır. `catch` bloğu hatanın iletisini yazar; sürücü sınıf yolunda olmasaydı burada "No suitable driver found ..." yazardı.

11.6.2 Tablo oluşturma

```
String url = "jdbc:sqlite:sample.db";
String sql = "CREATE TABLE IF NOT EXISTS users "
           + "(id INTEGER PRIMARY KEY, name TEXT, email TEXT)";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement()) {
  // Tablo oluşturuluyor.
  stmt.execute(sql);
  System.out.println("Tablo oluşturuldu!");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

```
Tablo oluşturuldu!
```

Bu kez try-with-resources parantezinde iki kaynak vardır: bağlantı ve bir `Statement`. Kaynaklar parantez içinde noktalı virgülle ayrılır. `execute(sql)` her tür SQL'i çalıştırır. Dönüş değeri, komutun bir sonuç kümesi üretip üretmediğini söyler. `CREATE TABLE` sonuç kümesi üretmez; `execute` `false` döndürür. `SELECT` için `true` dönerdi. Programda dönüş değeri kullanılmamıştır; ikisi de ayrı bir denemede yazdırılarak doğrulandı.

`IF NOT EXISTS` sayesinde program ikinci kez çalıştırıldığında hata vermez. Bu ifade olmasaydı ikinci çalıştırmada şu ileti alınırdı: `[SQLITE_ERROR] SQL error or missing database (table users already exists)`.

11.6.3 Veri ekleme

```
String url = "jdbc:sqlite:sample.db";
String sql = "INSERT INTO users(name, email) "
           + "VALUES('Ahmet Yılmaz', 'ahmet@example.com')";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement()) {
  // Veri tabanına veri ekleniyor.
  int sayi = stmt.executeUpdate(sql);
  System.out.println("Veri eklendi! (" + sayi + " satır)");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

```
Veri eklendi! (1 satır)
```

Ekleme `executeUpdate` ile yapılır. Bu metot `INSERT`, `UPDATE` ve `DELETE` için kullanılır ve komutun etkilediği satır sayısını `int` olarak döndürür. Tek bir satır eklendiği için dönen değer 1'dir. `id` verilmemiştir; SQLite tablo boş olduğu için 1 atar.

SQL metni Java'da iki parça olarak yazılmış ve `+` ile birleştirilmiştir. Birinci parçanın sonundaki boşluğa dikkat edilmelidir: `"INSERT INTO users(name, email) "`. Boşluk unutulsaydı metin `…email)VALUES(…` biçiminde birleşirdi. SQLite bunu kabul ederdi, ama başka bir yerde, örneğin `"FROM users" + "WHERE id = 1"` birleşiminde, `usersWHERE` ortaya çıkar ve sorgu bozulur. Metin değerler SQL'in kuralı gereği tek tırnak içindedir.

Program her çalıştığında yeni bir satır eklenir: `id` 2, 3, … Aynı kişi tabloda birden çok kez bulunur. Bu, sorgu sonuçlarını beklenmedik biçimde değiştirebilir.

11.6.4 Veri okuma

```
String url = "jdbc:sqlite:sample.db";
String sql = "SELECT * FROM users";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(sql)) {
  // Satırlar tek tek okunup yazdırılıyor.
  while (rs.next()) {
    System.out.println("ID: " + rs.getInt("id")
        + ", Ad: " + rs.getString("name")
        + ", E-posta: " + rs.getString("email"));
  }
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

```
ID: 1, Ad: Ahmet Yılmaz, E-posta: ahmet@example.com
```

Okuma `executeQuery` ile yapılır ve bir `ResultSet` döndürür. Bu kez sonuç kümesi de bir kaynak olduğu için try-with-resources parantezine üçüncü kaynak olarak girer. `rs.next()` imleci bir sonraki satıra taşır; satır varsa `true`, yoksa `false` döndürür. `getInt("id")` ve `getString("name")` o anki satırın sütunlarını adlarıyla okur. Tabloda tek satır olduğu için döngü bir kez döner.

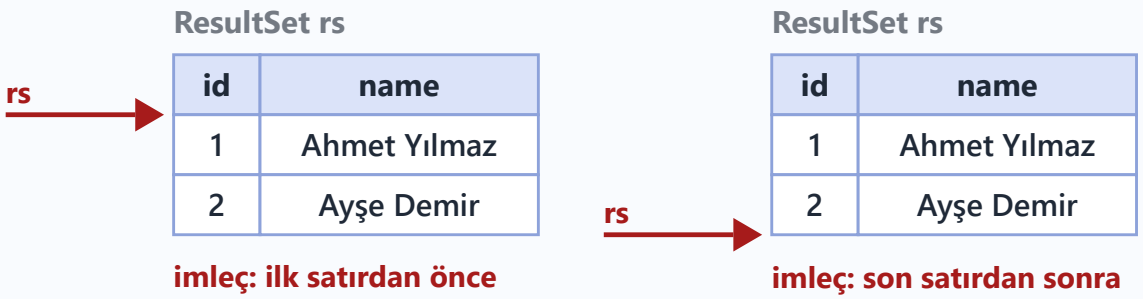
Sütun adına göre okumak, sıra numarasına göre okumaktan daha güvenlidir. `SELECT *` sütunları tablo tanımındaki sırayla getirir; tabloya ileride yeni bir sütun eklenirse numaralar kayabilir, adlar ise değişmez. SQLite sürücüsünde sütun adları büyük-küçük harfe duyarlı değildir: `getString("NAME")` da çalışır. Olmayan bir sütun adı ise bir `SQLException` fırlatır: `getString("ad")` denemesi `no such column: 'ad'` iletisini verdi.

11.6.5 Kod izleme: `ResultSet` ile okuma

`while (rs.next())` döngüsü JDBC'nin en sık görülen kalıbıdır. Aşağıdaki program iki satırlı bir tabloyu okur. `okuma.db`, programdan önce iki satırla hazırlanmıştır: (1, Ahmet Yılmaz) ve (2, Ayşe Demir).

```
String url = "jdbc:sqlite:okuma.db";
String sql = "SELECT * FROM users";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    int id = rs.getInt("id");
    String ad = rs.getString("name");
    System.out.println(id + " - " + ad);
  }
  System.out.println("Okuma bitti.");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

```
1 - Ahmet Yılmaz
2 - Ayşe Demir
Okuma bitti.
```



Şekil 11.20: İmleç başlangıçta ilk satırın önünde (sol), son `next()` çağrısından sonra son satırın ardında (sağ).

Adım	Satır	Değişkenler	Yeni çıktı
1	3: `try (Connection conn = DriverManager.getConnection(url);` 4: `Statement stmt = conn.createStatement();` 5: `ResultSet rs = stmt.executeQuery(sql)) {`	—	—
2	6: `while (rs.next()) {`	rs.next()=true	—
3	7: `int id = rs.getInt("id");` 8: `String ad = rs.getString("name");`	rs.next()=true, id=1, ad="Ahmet Yılmaz"	—
4	9: `System.out.println(id + " - " + ad);`	rs.next()=true, id=1, ad="Ahmet Yılmaz"	`1 - Ahmet Yılmaz`
5	6: `while (rs.next()) {`	rs.next()=true, id=1, ad="Ahmet Yılmaz"	—

Adım	Satır	Değişkenler	Yeni çıktı
6	7: `int id = rs.getInt("id");` 8: `String ad = rs.getString("name");`	rs.next()=true, id=2, ad="Ayşe Demir"	—
7	9: `System.out.println(id + " - " + ad);`	rs.next()=true, id=2, ad="Ayşe Demir"	`2 - Ayşe Demir`
8	6: `while (rs.next()) {`	rs.next()=false	—
9	11: `System.out.println("Okuma bitti.");`	rs.next()=false	`Okuma bitti.`
10	12: `} catch (SQLException e) {`	—	—

Tablo 11.15: `ResultSet` ile okumanın izlenmesi.

Tablo 11.15 on adımı verir. Birinci adımda bağlantı açılır ve sorgu çalışır; `rs` iki satırlık sonucu tutar, imleç ilk satırın önündedir (Şekil 11.20, sol). İkinci adımda `next()` imleci birinci satıra taşır ve `true` döndürür. Üçüncü ve dördüncü adımda sütunlar o anki satırdan okunur ve yazdırılır. Beşinci adımda imleç ikinci satıra geçer; altıncı ve yedinci adımlar aynı işi ikinci satır için yapar. Sekizinci adımda başka satır yoktur; `next()` `false` döndürür ve döngü biter. Dokuzuncu adımda "Okuma bitti." yazılır. Onuncu adımda `try` bloğu biter ve `rs`, `stmt`, `conn` bu sırayla kapanır.

`next()` toplam üç kez çağrılmıştır: iki satır için birer kez ve sonunu anlamak için bir kez. Genel olarak *n* satırlı bir sonuçta döngü *n* kez döner, `next()` *n* + 1 kez çağrılır. Boş bir sonuçta ilk `next()` `false` döner ve döngü gövdesi hiç çalışmaz. Bu, hata değildir: `WHERE id = 99` gibi eşleşmeyen bir sorgu boş bir sonuç kümesi döndürür.

> **Sık yapılan hata**
>
> İlk `next()` 'i çağırmadan sütun okumak. İmleç ilk satırın önündedir; okunacak bir satır yoktur. Standart JDBC bu durumda bir `SQLException` bekler. SQLite sürücüsü 3.40.0.0 ise denemede hata vermeden birinci satırın değerini döndürdü. Bu, bir sürücünün kendi davranışıdır ve başka sürücülerde kod çöker. Tek satırlık sonuçlarda da kalıp `if (rs.next()) { … }` biçimindedir.

11.6.6 Veri güncelleme ve silme

Güncelleme ve silme, eklemeyle aynı kalıbı kullanır; değişen yalnız SQL metnidir.

```
String url = "jdbc:sqlite:sample.db";
String sql = "UPDATE users SET name = 'Mehmet Akdeniz' WHERE id = 1";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement()) {
  // Veri tabanındaki veri güncelleniyor.
  int sayi = stmt.executeUpdate(sql);
  System.out.println("Veri güncellendi! (" + sayi + " satır)");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

```
Veri güncellendi! (1 satır)
```

Güncellemeden önce okuma programı `ID: 1, Ad: Ahmet Yılmaz, …` yazıyordu. Güncellemeden sonra aynı program şunu yazar:

```
ID: 1, Ad: Mehmet Akdeniz, E-posta: ahmet@example.com
```

`SET` yalnız yazılan sütunu değiştirir; `email` aynı kalmıştır. `WHERE id = 1` güncellenecek satırı seçer. `executeUpdate` 1 döndürür, çünkü koşulu tek satır sağlamıştır. Silme programı da aynı yapıdadır:

```
String url = "jdbc:sqlite:sample.db";
String sql = "DELETE FROM users WHERE id = 1";
try (Connection conn = DriverManager.getConnection(url);
     Statement stmt = conn.createStatement()) {
  // Veri tabanındaki veri siliniyor.
  int sayi = stmt.executeUpdate(sql);
  System.out.println("Veri silindi! (" + sayi + " satır)");
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

Program iki kez çalıştırıldığında çıktı şöyledir:

```
Veri silindi! (1 satır)
```

```
Veri silindi! (0 satır)
```

İlk çalıştırmada 1 numaralı satır silinir. İkinci çalıştırmada silinecek satır kalmamıştır. Bu bir hata değildir: `DELETE` koşulu sağlayan sıfır satırı siler ve `executeUpdate` 0 döndürür. Program yine de "Veri silindi!" yazar ve kullanıcıyı yanıltır. Bu yüzden ileti dönen sayıya göre verilmelidir. `WHERE`'siz `DELETE FROM users` ise tablodaki bütün satırları siler. Tablonun kendisi, yani yapısı, kalır; tabloyu tümüyle kaldıran komut `DROP TABLE`'dır.

11.6.7 Üç `execute` metodu

`Statement`'ın SQL çalıştıran üç metodu vardır. Tablo 11.16 hangisinin ne zaman kullanılacağını gösterir.

Metot	Döndürür	Kullanım
`execute`	`boolean`: sonuç kümesi var mı	`CREATE TABLE`, `DROP`; türü bilinmeyen SQL
`executeUpdate`	`int`: etkilenen satır sayısı	`INSERT`, `UPDATE`, `DELETE`
`executeQuery`	`ResultSet`	`SELECT`

Tablo 11.16: `Statement`'ın üç çalıştırma metodu.

Yanlış metot seçilirse ne olur? Aşağıdaki deneme her metodu "yanlış" bir komutla çağırır:

```
query does not return ResultSet
satır sayısı: 1
executeUpdate(SELECT) = 0
execute(INSERT) = false, getUpdateCount = 1
execute(SELECT) = true
execute(CREATE) = false, getUpdateCount = 1
executeUpdate(CREATE) = 0
executeUpdate(DROP) = 0
```

`executeQuery` ile `INSERT` çalıştırmak `query does not return ResultSet` iletisiyle bir `SQLException` fırlatır. İkinci satır şaşırtıcıdır: hata alınmasına rağmen satır tabloya eklenmiştir. SQLite sürücüsü komutu önce çalıştırmış, sonuç kümesi olmadığını sonra fark etmiştir. Programın "hata oldu, ekleme yapılmadı" sanması yanlış olur. `executeUpdate` ile `SELECT` bu sürücüde hata vermedi ve 0 döndürdü; standart ise bir `SQLException` bekler. `execute` her komutu kabul eder ve sonucu sonradan `getUpdateCount()` ya da `getResultSet()` ile almayı sağlar. `CREATE` ve `DROP` için `executeUpdate` 0 döndürür. Kısacası doğru metot seçilir; sürücülerin yanlış kullanıma verdiği tepki birbirinden farklıdır.

Altı program sırayla komut satırından çalıştırıldığında ekran şöyle görünür:

```
> java Baglan
Bağlantı başarılı!
> java TabloOlustur
Tablo oluşturuldu!
> java VeriEkle
Veri eklendi! (1 satır)
> java VeriOku
ID: 1, Ad: Ahmet Yılmaz, E-posta: ahmet@example.com
> java VeriGuncelle
Veri güncellendi! (1 satır)
> java VeriOku
ID: 1, Ad: Mehmet Akdeniz, E-posta: ahmet@example.com
> java VeriSil
Veri silindi! (1 satır)
> java VeriOku
> java VeriSil
Veri silindi! (0 satır)
```

Bu döküm gerçek çalıştırmayla birebir aynıdır. Silmeden sonraki `VeriOku` hiçbir şey yazmaz: tablo vardır ama boştur, döngü gövdesi hiç çalışmaz. Son `VeriSil` 0 satır siler.

11.6.8 Kod izleme: `executeUpdate`'in dönüş değerleri

`executeUpdate`'in döndürdüğü sayı, programın doğru iletiyi vermesi için gerekli bilgiyi taşır. Aşağıdaki kod bir malzeme tablosuna üç kayıt ekler, ardından bir güncelleme ve bir silme yapar. Ekleme `PreparedStatement` ile yapılmıştır. Burada bilinmesi gereken, `?` işaretlerinin değerlerin yerini tuttuğu ve `setString`, `setInt` ile doldurulduğudur.

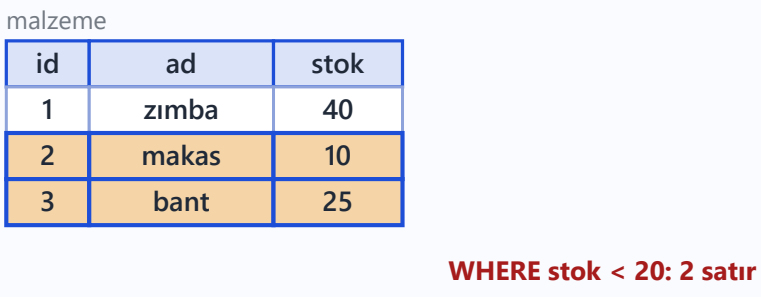
```
String[] adlar = {"zımba", "makas", "bant"};
int[] stok = {40, 0, 15};
PreparedStatement ek = conn.prepareStatement(
    "INSERT INTO malzeme(ad, stok) VALUES(?, ?)");
for (int i = 0; i < adlar.length; i++) {
  ek.setString(1, adlar[i]);
  ek.setInt(2, stok[i]);
  ek.executeUpdate();
}
int g = stmt.executeUpdate(
    "UPDATE malzeme SET stok = stok + 10 WHERE stok < 20");
int s = stmt.executeUpdate("DELETE FROM malzeme WHERE stok > 100");
System.out.println(g + " güncellendi, " + s + " silindi");
```

Program bir bağlantı ve `malzeme(id, ad, stok)` tablosuyla çalıştırıldığında ekleme döngüsünün her turunda `executeUpdate`'in döndürdüğü değer de yazdırıldı. Programın son satırı ve tablonun son durumu da çıktıdadır:

```
tur 0: 1
tur 1: 1
tur 2: 1
2 güncellendi, 0 silindi
1 zımba 40
2 makas 10
3 bant 25
```

Adım	Satır	Değişkenler	Yeni çıktı
1	1: `String[] adlar = {"zımba", "makas", "bant"};` 2: `int[] stok = {40, 0, 15};` 3: `PreparedStatement ek = conn.prepareStatement(` 4: `"INSERT INTO malzeme(ad, stok) VALUES(?, ?)");`	adlar=[zımba, makas, bant], stok=[40, 0, 15]	—
2	6: `ek.setString(1, adlar[i]);` 7: `ek.setInt(2, stok[i]);`	i=0	—
3	8: `ek.executeUpdate();`	i=0, dönen=1	—
4	5: `for (int i = 0; i < adlar.length; i++) {` 6: `ek.setString(1, adlar[i]);` 7: `ek.setInt(2, stok[i]);` 8: `ek.executeUpdate();`	i=1, dönen=1	—
5	5: `for (int i = 0; i < adlar.length; i++) {` 6: `ek.setString(1, adlar[i]);` 7: `ek.setInt(2, stok[i]);` 8: `ek.executeUpdate();`	i=2, dönen=1	—
6	10: `int g = stmt.executeUpdate(` 11: `"UPDATE malzeme SET stok = stok + 10 WHERE stok < 20");`	g=2	—
7	12: `int s = stmt.executeUpdate("DELETE FROM malzeme WHERE stok > 100");`	g=2, s=0	—
8	13: `System.out.println(g + " güncellendi, " + s + " silindi");`	g=2, s=0	`2 güncellendi, 0 silindi`

Tablo 11.17: `executeUpdate` dönüş değerlerinin izlenmesi.



Şekil 11.21: `WHERE stok < 20` makas (0) ve bant (15) satırlarını seçer; ikisine 10 eklenir.

Ekleme komutu döngüden önce bir kez hazırlanır; her turda yalnız değerler değişir. Her ekleme 1 döndürür ve SQLite satırlara sırayla 1, 2 ve 3 numaralarını verir. Güncellemede `WHERE stok < 20` koşulunu makas (0) ve bant (15) sağlar (Şekil 11.21). İkisinin stoğuna 10 eklenir: makas 10, bant 25 olur. `g` 2'dir. `SET stok = stok + 10`, sütunun eski değerini kullanarak yeni değeri hesaplar; Java'daki `stok += 10`'un karşılığıdır.

Silmede stoğu 100'den büyük ürün yoktur. Hata oluşmaz, yalnız 0 döner. Program bu sayılara bakarak "kayıt bulunamadı" gibi doğru iletiler verebilir. Bir maliyet notu: `WHERE` koşulu indekslenmemiş bir sütundaysa (`stok` gibi) güncelleme ve silme bütün tabloyu tarar; *n* satırlı tabloda maliyet *n* ile orantılıdır. `WHERE id = 1` gibi birincil anahtar koşulunda ise satır doğrudan bulunur.

11.7 `PreparedStatement`

Şimdiye kadarki SQL metinleri sabitti. Gerçek programlarda ise değerler kullanıcıdan gelir: aranacak ad, eklenecek not, silinecek kayıt numarası. Bu değerleri SQL metnine yerleştirmenin bir yanlış, bir de doğru yolu vardır.

11.7.1 SQL enjeksiyonu

En kolay görünen yol, kullanıcının yazdığı değeri dizgi birleştirmeyle SQL metnine yapıştırmaktır. Bir arama ekranında kullanıcı aranacak adı yazar ve program şu metni kurar: `"SELECT ad FROM kisiler WHERE ad = '" + girdi + "'"`. Girdi `Ali` olduğu sürece her şey yolundadır. Ama girdi bir tırnak işareti içerirse SQL metninin yapısı değişir.

> **Tanım 11.10**
>
> **SQL enjeksiyonu** (*SQL injection*), kullanıcı girdisinin SQL komutuna karışıp komutun anlamını değiştirmesi saldırısıdır. Girdi, değer olarak değil SQL'in parçası olarak yorumlanır.

```
String girdi = "x' OR '1'='1";   // kullanıcının yazdığı "ad"

// TEHLİKELİ: girdi SQL metnine yapıştırılıyor
String sql = "SELECT ad FROM kisiler WHERE ad = '"
           + girdi + "'";
System.out.println(sql);
ResultSet rs1 = stmt.executeQuery(sql);

// GÜVENLİ: girdi yalnızca değer olarak gönderiliyor
PreparedStatement ps = conn.prepareStatement(
    "SELECT ad FROM kisiler WHERE ad = ?");
ps.setString(1, girdi);
ResultSet rs2 = ps.executeQuery();
```

Kodun birinci yarısı tehlikeli, ikinci yarısı güvenli yoldur. `kisiler` tablosunda Ali, Ayşe ve Can adlı üç kişi varken iki sonuç kümesinin satırları sayıldı:

```
SELECT ad FROM kisiler WHERE ad = 'x' OR '1'='1'
Birleştirme: 3 satır
PreparedStatement: 0 satır
girdi uzunluğu: 12
[SQLITE_ERROR] SQL error or missing database (near "nin": syntax error)
PreparedStatement ile: 1
Ali'nin
INSERT INTO kisiler(ad) VALUES('Ali''nin')
tablo duruyor: 4
```

Çıktının ilk satırı veri tabanına giden SQL metnidir. Kullanıcı ad yerine `x' OR '1'='1` yazmıştır. Birleştirmede girdinin içindeki ilk tırnak, programın açtığı tırnağı kapatır. Ad değeri `'x'` olarak biter. Ardından gelen `OR '1'='1'` artık bir değer değil, SQL'in parçasıdır. Programın en sona eklediği tırnak da `'1`'i kapatır ve metin söz dizimi açısından kusursuz olur. Koşul `ad = 'x' OR '1'='1'`'dir; `'1'='1'` her satırda doğru olduğu için koşul hiçbir satırı elemez. Üç satır döner.

Bir giriş ekranında bu, şifresiz oturum açmak demektir. Daha kötüsü de mümkündür: girdi verileri silen ya da başka tablolardaki şifreleri okuyan bir SQL parçası olabilir. SQL enjeksiyonu yıllardır web uygulamalarının en yaygın güvenlik açıkları arasındadır.

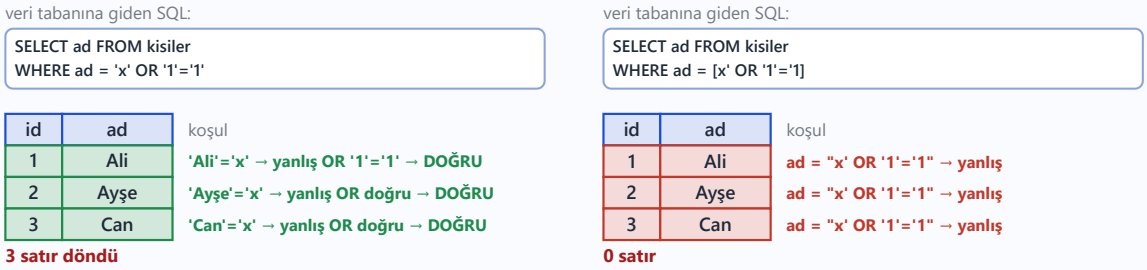
İkinci yarıdaki `PreparedStatement` aynı girdiyle 0 satır döndürür. Burada SQL metni sabittir ve girdinin yeri `?` ile işaretlenmiştir. Girdi `setString(1, girdi)` ile *ayrıca* gönderilir ve yalnız bir değer olarak kullanılır. Veri tabanı "adı tam olarak `x' OR '1'='1` olan kişiyi" arar. Bu 12 karakterlik metin hiç kimsenin adı değildir.

Çıktının son satırları iki yan sonucu daha gösterir. `Ali'nin` gibi kesme işaretli bir metin birleştirmeyle eklenmek istendiğinde SQL bozulur ve `near "nin": syntax error` iletisi alınır. Saldırı amacı olmayan sıradan bir kullanıcı bile programı bozabilir. `PreparedStatement` ile aynı metin sorunsuz eklenir ve geri okunur. Tırnağı elle ikilemek (`''`) de bir yoldur ama hataya açıktır; tek bir unutulan yer bütün programı savunmasız bırakır. SQLite sürücüsünün `executeQuery`'si noktalı virgülle eklenen ikinci bir komutu çalıştırmadı (`DROP TABLE` denemesinden sonra tablo duruyordu). Bu bir koruma sayılmaz: tek bir komutun içinde kalan saldırı, yukarıdaki gibi, yine işler.

11.7.2 Kod izleme: SQL enjeksiyonu

Aşağıdaki kod iki yolu satır sayısıyla karşılaştırır:

```
String girdi = "x' OR '1'='1";
String sql = "SELECT ad FROM kisiler WHERE ad = '"
           + girdi + "'";
int n = 0;
try (ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) n++;
}
System.out.println("Birleştirme: " + n + " satır");
PreparedStatement ps = conn.prepareStatement(
    "SELECT ad FROM kisiler WHERE ad = ?");
ps.setString(1, girdi);
n = 0;
try (ResultSet rs = ps.executeQuery()) { while (rs.next()) n++; }
System.out.println("PreparedStatement: " + n + " satır");
```



Şekil 11.22: Birleştirilmiş SQL'de koşul her satırda doğrudur ve 3 satır döner (sol); `PreparedStatement`'ta girdi tek bir metin değeridir ve hiçbir adla eşleşmez (sağ).

Adım	Satır	Değişkenler	Yeni çıktı
1	1: `String girdi = "x' OR '1'='1";`	girdi=x' OR '1'='1	—
2	2: `String sql = "SELECT ad FROM kisiler WHERE ad = '"` 3: `+ girdi + "'";`	sql=SELECT … WHERE ad = 'x' OR '1'='1'	—
3	5: `try (ResultSet rs = stmt.executeQuery(sql)) {`	rs=ResultSet	—
4	5: `try (ResultSet rs = stmt.executeQuery(sql)) {`	rs=ResultSet	—
5	6: `while (rs.next()) n++;` 7: `}`	n=3	—
6	8: `System.out.println("Birleştirme: " + n + " satır");`	n=3	`Birleştirme: 3 satır`
7	9: `PreparedStatement ps = conn.prepareStatement(` 10: `"SELECT ad FROM kisiler WHERE ad = ?");`	ps=PreparedStatement	—
8	11: `ps.setString(1, girdi);`	1. ?=x' OR '1'='1	—
9	13: `try (ResultSet rs = ps.executeQuery()) { while (rs.next()) n++; }`	n=0	—

Adım	Satır	Değişkenler	Yeni çıktı
10	14: `System.out.println("PreparedStatement: " + n + " satır");`	n=0	`PreparedStatement: 0 satır`

Tablo 11.18: SQL enjeksiyonunun izlenmesi.

Üçüncü ve dördüncü adımlarda veri tabanı koşulu her satır için değerlendirir. Ali için `'Ali'='x'` yanlıştır ama `OR`'un sağ tarafı her zaman doğrudur; satır seçilir. Ayşe ve Can da aynı nedenle seçilir (Şekil 11.22, sol). Yedinci adımda SQL *önce* derlenir; `?` yalnız bir değerin yeridir. Komutun yapısı bu noktada kesinleşmiştir ve girdi ne olursa olsun değişemez. Sekizinci adımda girdi bütünüyle tek bir metin değeri olarak bağlanır; içindeki tırnaklar sıradan karakterdir. Dokuzuncu adımda hiçbir ad bu metne eşit değildir. Aynı girdi iki farklı sonuç verir. Kural kesindir: kullanıcıdan gelen değer SQL'e her zaman `?` ile verilir.

11.7.3 Yer tutucular

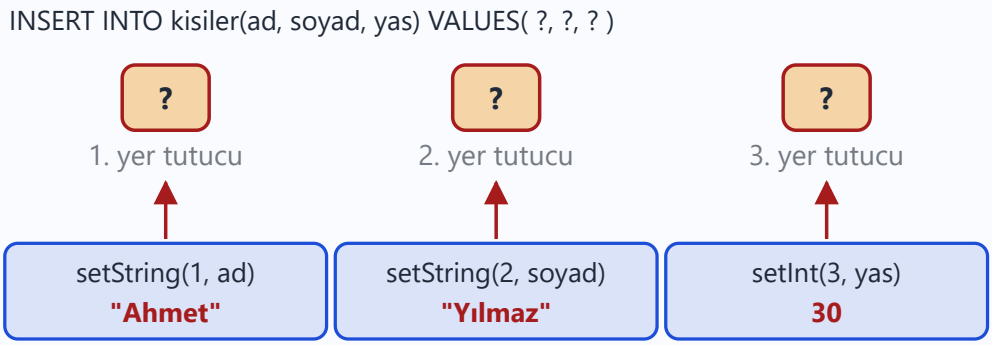
Tanım 11.11

Önceden hazırlanan deyim (`PreparedStatement`), değerlerin yerine `?` yer tutucuları içeren ve değerleri ayrıca alan SQL deyimidir. Bağlantının `prepareStatement(sql)` metoduyla oluşturulur; yer tutucular `setXxx(sıra, değer)` metotlarıyla doldurulur.

Aşağıdaki metot bir kişiyi `kisiler` tablosuna ekler:

```
static void ekle(Connection conn, String ad, String soyad,
                 int yas) throws SQLException {
  String sql = "INSERT INTO kisiler(ad, soyad, yas) "
             + "VALUES(?, ?, ?)";
  try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, ad);      // 1. soru işareti
    ps.setString(2, soyad);   // 2. soru işareti
    ps.setInt(3, yas);        // 3. soru işareti
    ps.executeUpdate();
  }
}
```

SQL metninde üç `?` vardır. `setXxx` metotlarının ilk argümanı yer tutucunun sırasıdır; sıra **1'den** başlar ve SQL metnindeki soldan sağa sırayı izler (Şekil 11.23). Metodun adı değerin Java tipini söyler: metin için `setString`, tam sayı için `setInt`, ondalıklı sayı için `setDouble`. Bütün yer tutucular doldurulduktan sonra `executeUpdate()` *parametresiz* çağrılır, çünkü SQL metni hazırlanırken verilmiştir.



Şekil 11.23: `setString` ve `setInt` çağrıları soru işaretlerini sırayla, 1'den başlayan numaralarla doldurur.

`PreparedStatement`'ın üç yararı vardır. Birincisi **güvenliktir**: SQL enjeksiyonunu önler. İkincisi **tırnak sorunu** yoktur: `"Ali'nin"` sorunsuz kaydedilir. Üçüncüsü **yeniden kullanımdır**: aynı komut bir kez hazırlanır ve farklı değerlerle tekrar tekrar çalıştırılır. VTYS komutu her seferinde yeniden çözümlemez; döngüde binlerce ekleme yapılırken bu fark hissedilir.

Yer tutucular yalnız *değerlerin* yerini tutabilir. Tablo ya da sütun adı `?` ile verilemez: `SELECT * FROM ?` hazırlanırken `near "?": syntax error` iletisi alındı. Tırnak içindeki `'?'` ise bir yer tutucu değil, soru işareti karakteridir; `WHERE ad = '?'` sorgusunda `setString(1, …)` çağrısı başarısız olur. Doldurulmayan bir yer tutucu `NULL` sayılır. Üç yer tutuculu ekleme yalnız iki değer verilerek çalıştırıldığında `NOT NULL constraint failed: kisiler.yas` iletisi alındı. Bir başka ayrıntı: doldurulan değer, yeni bir `setXxx` çağrısına kadar yerinde kalır. Bir eklemeden sonra yalnız `setString(1, "Bora")` değiştirilip yeniden çalıştırıldığında, ikinci satır öncekinin soyadını ve yaşını aldı.

Sık yapılan hata

Yer tutucu sırasını 0'dan başlatmak: `ps.setInt(0, 5)`. Standart bu durumda bir `SQLException` bekler. SQLite sürücüsü 3.40.0.0 ise denetlenmeyen bir `ArrayIndexOutOfBoundsException` fırlattı (`Index -1 out of bounds for length 3`). Bu istisna `catch (SQLException e)` ile yakalanmaz ve programı durdurur. Sıra her zaman 1'den başlar ve `?` sayısını aşmaz.

11.7.4 Kişi programı

Ekleme metodu, güncelleme, silme ve listeleme metotlarıyla birlikte küçük ama tam bir program oluşturur. Güncelleme ve silme metotları şöyledir:

```
static void guncelle(Connection conn, String ad, String soyad,
                     int yas) throws SQLException {
  String sql = "UPDATE kisiler SET soyad = ?, yas = ? "
             + "WHERE ad = ?";
  try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, soyad);
    ps.setInt(2, yas);
    ps.setString(3, ad);
    ps.executeUpdate();
  }
}

static void sil(Connection conn, String ad) throws SQLException {
  String sql = "DELETE FROM kisiler WHERE ad = ?";
  try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setString(1, ad);
    ps.executeUpdate();
  }
}
```

Her metot `Connection`'ı parametre olarak alır. Bağlantı `main`'de bir kez açılır ve birçok işte kullanılır. Her işlem için yeni bağlantı açmak hem yavaştır hem de gereksizdir. `PreparedStatement` ise her metodun kendi try-with-resources bloğunda açılır ve metot bitince kapanır; bağlantı açık kalır.

Güncelleme metodunda `?`'lerin sırasına dikkat edilmelidir. SQL metni `SET soyad = ?, yas = ? WHERE ad = ?`'dir. Metodun parametre sırası `(ad, soyad, yas)` olsa da SQL'deki sırada ad *üçüncü* yer tutucudur. Bu yüzden `setString(3, ad)` yazılır. Numaralar Java parametrelerine göre değil, SQL metnine göre verilir.

Metotlar `throws SQLException` ile hatayı çağırana iletir. `main` de `throws SQLException` dediği için bir hata olursa program yığın iziyle durur. Küçük bir örnek program için bu kabul edilebilir. Gerçek bir programda hata `main`'de yakalanır ve kullanıcıya anlaşılır bir ileti verilir. Listeleme metodu şöyledir:

```
static void listele(Connection conn) throws SQLException {
  String sql = "SELECT id, ad, soyad, yas FROM kisiler";
  try (Statement st = conn.createStatement();
       ResultSet rs = st.executeQuery(sql)) {
    while (rs.next()) {
      System.out.println(rs.getInt("id") + " " + rs.getString("ad")
          + " " + rs.getString("soyad") + " " + rs.getInt("yas"));
    }
    System.out.println("---");
  }
}
```

Bu sorguda değişken bir değer yoktur; düz `Statement` yeterlidir. Kullanıcı girdisi olmadığı için enjeksiyon riski de yoktur. Her `listele` çağrısı tablonun o anki hâlini okur ve sonuna `---` yazar. Son olarak `main` metotları sırayla çağırır:

```
public static void main(String[] args) throws SQLException {
  String url = "jdbc:sqlite:kisiler.db";
  try (Connection conn = DriverManager.getConnection(url);
       Statement stmt = conn.createStatement()) {
    stmt.execute("CREATE TABLE IF NOT EXISTS kisiler ("
        + "id INTEGER PRIMARY KEY, ad TEXT NOT NULL, "
        + "soyad TEXT NOT NULL, yas INTEGER NOT NULL)");
    ekle(conn, "Ahmet", "Yılmaz", 30);
    ekle(conn, "Ayşe", "Demir", 40);
    guncelle(conn, "Ahmet", "Akdeniz", 35);
    listele(conn);
    sil(conn, "Ahmet");
    listele(conn);
  }
}
```

Dört metot ve `main` tek bir `Kisi` sınıfında birleştirilip boş bir klasörde çalıştırıldı:

```
1 Ahmet Akdeniz 35
2 Ayşe Demir 40
---
2 Ayşe Demir 40
---
```

11.7.5 Kod izleme: kişi programı

Adım	Satır	Değişkenler	Yeni çıktı
1	3: `try (Connection conn = DriverManager.getConnection(url);` 4: `Statement stmt = conn.createStatement()) {`	—	—
2	5: `stmt.execute("CREATE TABLE IF NOT EXISTS kisiler ("` 6: `+ "id INTEGER PRIMARY KEY, ad TEXT NOT NULL, "` 7: `+ "soyad TEXT NOT NULL, yas INTEGER NOT NULL)");`	—	—
3	8: `ekle(conn, "Ahmet", "Yılmaz", 30);`	—	—
4	9: `ekle(conn, "Ayşe", "Demir", 40);`	—	—
5	10: `guncelle(conn, "Ahmet", "Akdeniz", 35);`	—	—

Adım	Satır	Değişkenler	Yeni çıktı
6	11: `listele(conn);`	—	`1 Ahmet Akdeniz 35 ⏎ 2 Ayşe Demir 40 ⏎ ---`
7	12: `sil(conn, "Ahmet");`	—	—
8	13: `listele(conn);`	—	`2 Ayşe Demir 40 ⏎ ---`
9	14: `}`	—	—

Tablo 11.19: Kişi programının izlenmesi (satır numaraları `main`'e göredir).

kisiler

id	ad	soyad	yas
1	Ahmet	Akdeniz	35
2	Ayşe	Demir	40

kisiler

id	ad	soyad	yas
2	Ayşe	Demir	40

Şekil 11.24: Güncellemeden sonra `kisiler` tablosu (sol) ve silmeden sonra (sağ).

Birinci adımda `kisiler.db` yoksa oluşturulur; bağlantı ve `stmt` açılır. İkinci adımda tablo oluşturulur ve henüz boştur. Üçüncü ve dördüncü adımlarda `ekle` çağrılır; Ahmet `id` = 1, Ayşe `id` = 2 ile eklenir. Beşinci adımda `WHERE ad = ?` Ahmet'i seçer; soyadı Akdeniz, yaşı 35 olur. Altıncı adımda `listele` iki satırı ve `---`'yü yazar. Yedinci adımda Ahmet'in satırı silinir. Sekizinci adımda tabloda yalnız Ayşe kalmıştır. Dokuzuncu adımda bağlantı kapanır; veri `kisiler.db`'de kalıcıdır (Şekil 11.24).

Bir maliyet notu eklenebilir. `guncelle` ve `sil`, `WHERE ad = ?` ile çalışır. `ad` sütununda indeks olmadığından her çağrı tabloyu tarar. Ayrıca aynı adlı iki kişi varsa ikisini birden günceller ya da siler. Birincil anahtarla (`WHERE id = ?`) çalışmak hem hızlı hem de kesindir.

11.8 İleri JDBC

11.8.1 Otomatik anahtar: `getGeneratedKeys`

Yeni satırın `id`'sini veri tabanı belirler. Program çoğu zaman bu numarayı hemen bilmek ister. Bir sipariş eklendiğinde siparişin kalemleri ayrı bir tabloya, siparişin `id`'siyle eklenir. Bu numarayı öğrenmenin yolu `getGeneratedKeys()`'tir:

```
String sql = "INSERT INTO ogrenciler(ad, bolum_id) "
    + "VALUES(?, ?)";
try (PreparedStatement ps = conn.prepareStatement(sql,
        Statement.RETURN_GENERATED_KEYS)) {
  ps.setString(1, "Ece");
  ps.setInt(2, 3);
  ps.executeUpdate();
  try (ResultSet k = ps.getGeneratedKeys()) {
    if (k.next()) {
      System.out.println("Yeni id: " + k.getInt(1));
    }
  }
}
```

```
Yeni id: 5
id INTEGER PRIMARY KEY -> 3
id INTEGER PRIMARY KEY AUTOINCREMENT -> 4
ortadan silince: 1,3,4
last_insert_rowid: 5
```

`prepareStatement`'a ikinci argüman olarak `Statement.RETURN_GENERATED_KEYS` verilir. Bu sabit, sürücüye oluşturulan anahtarları saklamasını söyler. `executeUpdate`'ten sonra `getGeneratedKeys()` bir `ResultSet` döndürür. Bu sonuç kümesinin tek satırı ve tek sütunu yeni anahtardır. Tabloda dört öğrenci (Ada, Bora, Cem, Deniz) vardı; Ece 5 numarasını aldı. Yeni anahtar da bir sonuç kümesi olduğu için try-with-resources ile açılıp kapanır.

Çıktının sonraki iki satırı numaralandırmanın bir inceliğini gösterir. Bir tabloya 1, 2, 3 numaralı satırlar eklenir, 3 silinir ve yeni bir satır eklenir. Yeni satır hangi numarayı alır? Tablo 11.20 iki tanımı karşılaştırır.

Sütun tanımı	**Yeni `id`**
`id INTEGER PRIMARY KEY`	3 (en büyük `id` + 1)
`id INTEGER PRIMARY KEY AUTOINCREMENT`	4 (kullanılmış numara verilmez)

Tablo 11.20: 1, 2, 3 eklendi, 3 silindi; yeni satırın `id`'si.

Yalın `INTEGER PRIMARY KEY`, en büyük mevcut numaranın bir fazlasını verir. En büyük numara silinirse o numara yeniden kullanılır. Ortadan bir numara silinirse (örneğin 2) yeni satır yine 4 alır; boşluk doldurulmaz. `AUTOINCREMENT` ise o güne kadar kullanılmış en büyük numarayı ayrı bir tabloda saklar ve hiçbir numarayı yeniden vermez. Silinen kaydın numarası başka bir kayda geçmemeliyse (fatura numarası, sipariş numarası) `AUTOINCREMENT` kullanılır. MySQL'deki karşılığı `AUTO_INCREMENT`'tir.

11.8.2 Satırdan nesneye: DAO deseni

Şimdiye kadarki programlar SQL'i ve satır okumayı `main`'in içine dağıtıyordu. Nesne yönelimli bir programda ise her satır bir nesneye dönüştürülür ve SQL tek bir yerde toplanır. Öğrenci satırları için basit bir sınıf yeterlidir:

```
class Ogrenci {
  private final int id;
  private final String ad;
  Ogrenci(int id, String ad) {
    this.id = id;
    this.ad = ad;
  }
  public String toString() {
    return "#" + id + " " + ad;
  }
}
```

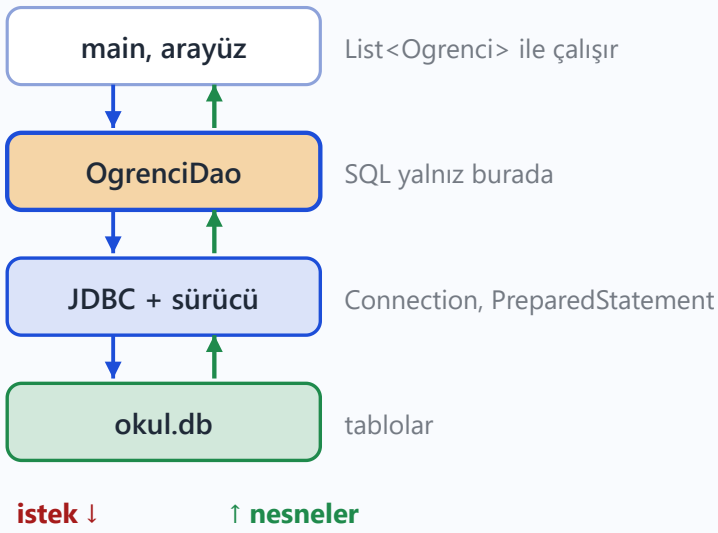
`Ogrenci` sınıfının alanları `final`'dır; nesne oluşturulduktan sonra değişmez. `toString`, nesneyi `#1 Ada` biçiminde yazar. Bu sınıf veri tabanından habersizdir: içinde ne SQL ne de JDBC vardır.

> **Tanım 11.12**
>
> **Veri erişim nesnesi** (DAO; *data access object*), bir tablonun bütün SQL işlemlerini tek bir sınıfta toplayıp sonuçları nesne olarak döndüren tasarımdır. Programın geri kalanı SQL görmez; yalnız DAO'nun metotlarını çağırır.

Tablo 11.21 bir `OgrenciDao` sınıfının metotlarını verir. Her metot bir SQL komutuna karşılık gelir ve sonucu Java'nın diliyle döndürür: satır yerine nesne, sonuç kümesi yerine liste, etkilenen satır sayısı yerine `boolean`.

`OgrenciDao` metodu	SQL	Döndürür
`ekle(ad, bolum)`	`INSERT`	yeni `id`
`bul(id)`	`SELECT … WHERE id = ?`	`Ogrenci` ya da `null`
`bolumdekiler(bolum)`	`SELECT … WHERE bolum_id = ?`	`List<Ogrenci>`
`sil(id)`	`DELETE`	silindi mi (`boolean`)

Tablo 11.21: `OgrenciDao` sınıfının metotları.



Şekil 11.25: Katmanlar: arayüz, `OgrenciDao`, JDBC ve veri tabanı dosyası. İstekler aşağı, nesneler yukarı gider.

Şekil 11.25 katmanlı yapıyı gösterir. Kullanıcı arayüzü (`main` ya da bir Swing penceresi) yalnız `OgrenciDao`'yu tanır ve ondan `List<Ogrenci>` ister. DAO, JDBC ve SQL ile çalışır. SQL yalnız bu sınıftadır. Veri tabanı değişirse ya da tablo yeniden düzenlenirse yalnız DAO değişir; arayüz kodu aynı kalır. Bu, sarmalamanın veri tabanı düzeyindeki karşılığıdır. Bağlantı DAO'ya yapıcıyla verilir; DAO bağlantıyı açmaz ve kapatmaz.

Tablodaki dört metodun tamamı yazılıp çalıştırıldı. Ada, Bora, Cem ve Deniz'in bulunduğu tabloda sonuçlar şöyledir:

```
ekle -> 5
bul(2) -> #2 Bora
bul(9) -> null
bolumdekiler(1) -> [#1 Ada, #3 Cem, #5 Ece]
sil(3) -> true
sil(3) -> false
bolumdekiler(1) -> [#1 Ada, #5 Ece]
```

`ekle`, `getGeneratedKeys` ile Ece'nin numarasını (5) döndürür. `bul(9)` satır bulamaz ve `null` döndürür; çağıran kod bunu denetlemelidir. `sil(3)` ilk çağrıda `true`, ikinci çağrıda `false` döndürür, çünkü `executeUpdate` ikinci kez 0 verir. DAO bu sayıyı `boolean`'a çevirerek programın geri kalanını SQL'in ayrıntısından korur.

Satırları nesnelere eşleme işi bu örnekte elle yapılmıştır. Büyük projelerde bu işi **ORM** (*object-relational mapping*, nesne-ilişkisel eşleme) kütüphaneleri, örneğin JPA standardı ve onun bir gerçeklemesi olan Hibernate, kendiliğinden yapar. Sınıf alanlarının hangi sütunlara karşılık geldiği açıklamalarla bildirilir; SQL'in çoğunu kütüphane kendisi üretir. ORM'nin arkasında yine JDBC çalışır; bu bölümde öğrenilenler o kütüphaneleri anlamanın da temelidir.

11.8.3 Kod izleme: satırdan nesneye

Aşağıdaki metot bir bölümdeki öğrencileri nesne listesi olarak döndürür; son iki satır onu `main`'den çağırır:

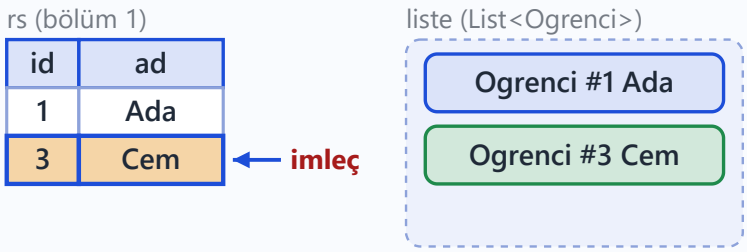
```
static List<Ogrenci> bolumdekiler(Connection conn,
    int bolum) throws SQLException {
  List<Ogrenci> liste = new ArrayList<>();
  String sql = "SELECT id, ad FROM ogrenciler "
      + "WHERE bolum_id = ? ORDER BY id";
  try (PreparedStatement ps = conn.prepareStatement(sql)) {
    ps.setInt(1, bolum);
    try (ResultSet rs = ps.executeQuery()) {
      while (rs.next()) {
        liste.add(new Ogrenci(rs.getInt("id"),
                              rs.getString("ad")));
      }
    }
  }
  return liste;
}
// main içinde:
List<Ogrenci> liste = bolumdekiler(conn, 1);
for (Ogrenci o : liste) System.out.println(o);
```

```
#1 Ada
#3 Cem
bölüm 3: []
bölüm 2: [#2 Bora]
```

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
1	18: `List<Ogrenci> liste = bolumdekiler(conn, 1);`	main() → bolumdekiler(conn, 1)	bolum=1	—
2	3: `List<Ogrenci> liste = new ArrayList<>();` 4: `String sql = "SELECT id, ad FROM ogrenciler "` 5: `+ "WHERE bolum_id = ? ORDER BY id";` 6: `try (PreparedStatement ps = conn.prepareStatement(sql)) {`	main() → bolumdekiler(conn, 1)	bolum=1, liste=[]	—
3	7: `ps.setInt(1, bolum);` 8: `try (ResultSet rs = ps.executeQuery()) {`	main() → bolumdekiler(conn, 1)	bolum=1, liste=[]	—
4	9: `while (rs.next()) {` 10: `liste.add(new Ogrenci(rs.getInt("id"),` 11: `rs.getString("ad")));`	main() → bolumdekiler(conn, 1)	liste=[#1 Ada]	—
5	9: `while (rs.next()) {` 10: `liste.add(new Ogrenci(rs.getInt("id"),` 11: `rs.getString("ad")));`	main() → bolumdekiler(conn, 1)	liste=[#1 Ada, #3 Cem]	—

Adım	Satır	Çağrı yığını	Değişkenler	Yeni çıktı
6	9: `while (rs.next()) {` 13: `}` 14: `}`	main() → bolumdekiler(conn, 1)	liste=[#1 Ada, #3 Cem]	—
7	15: `return liste;` 18: `List<Ogrenci> liste = bolumdekiler(conn, 1);`	main()	liste=[#1 Ada, #3 Cem]	—
8	19: `for (Ogrenci o : liste) System.out.println(o);`	main()	liste=[#1 Ada, #3 Cem]	`#1 Ada ⏎ #3 Cem`

Tablo 11.22: `bolumdekiler` metodunun izlenmesi.



Şekil 11.26: İmleç ikinci satırdayken (3, Cem) okunur; listede iki `Ogrenci` nesnesi vardır.

`main` metodu `bolum` = 1 ile çağırır; bağlantı `main`'de açıktır. Metot boş bir liste oluşturur ve sorguyu `?` ile hazırlar. `?` = 1 bağlanır ve sorgu çalışır; bölüm 1'de Ada ve Cem vardır. İlk satırdan `id` = 1, `ad` = Ada okunur ve `new Ogrenci(1, "Ada")` listeye eklenir. İkinci satırdan ikinci nesne oluşur: `#3 Cem` (Şekil 11.26). `next()` `false` döndürünce iç içe iki `try` bloğu biter; önce `rs`, sonra `ps` kapanır. Bağlantıyı onu açan `main` kapatır. Metot iki öğeli listeyi döndürür ve çerçevesi yığından kalkar. `main` her nesneyi `toString()` ile yazar; SQL'den habersizdir.

Çıktının son iki satırı aynı metodun öteki bölümler için sonuçlarıdır. 3 numaralı bölümde (Makine) hiç öğrenci yoktur; metot `null` değil, boş liste döndürür ve `[]` yazılır. Boş liste döndürmek `null`'dan daha güvenlidir: çağıran kod bir `for` döngüsüyle listede gezinirse döngü hiç dönmez, `NullPointerException` riski oluşmaz. Metodun maliyeti sorgunun maliyeti ile k nesne oluşturmanın toplamıdır; k, bölümdeki öğrenci sayısıdır. Liste k nesne kadar bellek kaplar. Çok büyük sonuçlar için bütün satırları listeye doldurmak yerine satır satır işlemek gerekebilir.

11.8.4 Aynı kod, başka veri tabanı: MySQL

JDBC'nin vaadi, aynı kodun başka veri tabanlarında da çalışmasıdır. Aşağıdaki program bir MySQL sunucusundaki `users` tablosunu okur:

```
String url = "jdbc:mysql://localhost:3306/testdb";
String kullanici = "root";
String sifre = "gizli";
String sql = "SELECT id, name, email FROM users";
try (Connection conn =
         DriverManager.getConnection(url, kullanici, sifre);
     Statement stmt = conn.createStatement();
     ResultSet rs = stmt.executeQuery(sql)) {
  while (rs.next()) {
    System.out.println(rs.getInt("id") + " "
        + rs.getString("name"));
  }
} catch (SQLException e) {
  System.out.println(e.getMessage());
}
```

JDBC kodu SQLite örneğiyle aynıdır. Değişenler dört şeydir: bağlantı adresi, kullanıcı adı, şifre ve sınıf yolundaki sürücü jar'ı. MySQL'in sürücüsü `mysql-connector-j`'dir. Adres `jdbc:mysql://localhost:3306/testdb` biçimindedir: `localhost` sunucunun çalıştığı makine, 3306 MySQL'in varsayılan kapısı, `testdb` veri tabanının adıdır. `getConnection`'ın üç argümanlı biçimi kullanıcı adı ve şifreyi de alır.

MySQL bir sunucudur. Program çalışmadan önce sunucu kurulmuş ve çalışıyor olmalı, `testdb` veri tabanı ve `users` tablosu oluşturulmuş olmalıdır. SQLite'taki gibi "dosya yoksa oluştur" davranışı yoktur. Bu notun hazırlandığı makinede MySQL sürücüsü yoktu. Program derlendi ve çalıştırıldığında `No suitable driver found for jdbc:mysql://localhost:3306/testdb` iletisini verdi; bu ileti beklenen davranıştır.

Sürücü sınıfının adı `com.mysql.cj.jdbc.Driver`'dır. Eski kodlarda görülen `com.mysql.jdbc.Driver` adı kullanımdan kalkmıştır. JDBC 4.0'dan beri sürücü kendiliğinden yüklendiği için bu ad kodda zaten yazılmaz. Son bir uyarı: gerçek projelerde şifre koda yazılmaz. Koda yazılan şifre derlenmiş `.class` dosyasında ve sürüm denetim sisteminde açıkça görünür. Şifre bir ayar dosyasından ya da ortam değişkeninden, örneğin `System.getenv("DB_SIFRE")` ile, okunur.

11.9 İşlemler

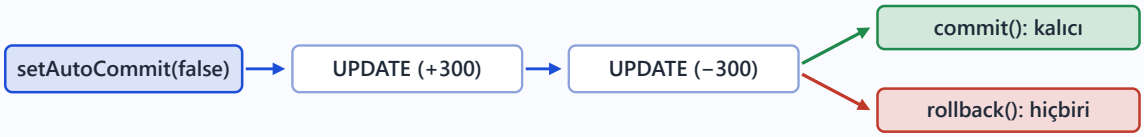
Şimdiye kadar her SQL komutu tek başına çalıştı ve hemen kalıcı oldu. Bazı işler ise birden çok komuttan oluşur ve bu komutlar ancak birlikte anlam taşır. Bu kesim böyle işlerin nasıl güvenle yürütüleceğini anlatır.

11.9.1 İşlem ve ACID

Bir havale düşünülsün: Ali, Ece'ye 300 TL gönderiyor. Bu iş iki `UPDATE` komutudur: Ece'nin hesabına 300 TL eklenir, Ali'nin hesabından 300 TL düşülür. Birinci komut çalıştıktan sonra elektrik kesilirse, ya da ikinci komut bir hata yüzünden çalışmazsa ne olur? Ece'nin parası artmış, Ali'ninki azalmamıştır: para yoktan var olmuştur. Komutların sırası ters olsaydı para kaybolurdu. İki komuttan birinin olup ötekinin olmaması kabul edilemez.

> **Tanım 11.13**
>
> **İşlem** (*transaction*), tek bir iş birimi olarak yürütülen bir ya da daha çok SQL komutudur. İşlemin komutları ya hepsi birlikte kalıcı olur ya da hiçbiri olmaz. Kalıcı yapmaya `commit`, geri almaya `rollback` denir.

JDBC'de bağlantının varsayılan ayarı **otomatik onaydır** (*auto-commit*): her komut kendi başına bir işlemdir ve çalışır çalışmaz kalıcı olur. Bu ayar SQLite bağlantısında `conn.getAutoCommit()` ile sorgulandı ve `true` çıktı. Birden çok komutu tek işlemde toplamanın kalıbı bellidir. `conn.setAutoCommit(false)` ile işlem başlatılır ve komutlar çalıştırılır. Her şey yolundaysa `conn.commit()` değişiklikleri kalıcı yapar; bir hata olursa `conn.rollback()` geri alır (Şekil 11.27).



Şekil 11.27: İşlem: `setAutoCommit(false)` 'tan sonraki komutlar geçicidir; `commit()` kalıcı yapar, `rollback()` geri alır.

Güvenilir bir işlemin dört özelliği vardır; baş harflerinden **ACID** diye anılır.

- **Bölünmezlik** (*atomicity*): işlemin ya tamamı ya hiçbiri uygulanır. Havalenin iki komutundan yalnız biri kalıcı olamaz.
- **Tutarlılık** (*consistency*): işlem sonunda bütün kısıtlar sağlanır. `CHECK (bakiye >= 0)` varsa hiçbir işlem bir bakiyeyi eksiye düşürerek bitemez.
- **Yalıtım** (*isolation*): eş zamanlı işlemler birbirinin yarım hâlini görmez. Havale sürerken başka bir program Ece'nin bakiyesini okursa ya havaleden önceki ya da sonraki değeri görür, aradaki değeri görmez.
- **Kalıcılık** (*durability*): commit edilen veri, hemen ardından elektrik kesilse de kaybolmaz. VTYS commit'i, değişiklik diske güvenle yazıldıktan sonra tamamlar.

Yalıtım bir denemeyle görülebilir. Aynı `banka.db` dosyasına iki ayrı bağlantı açıldı. Birinci bağlantı otomatik onayı kapatıp Ece'nin bakiyesini 500'den 800'e çıkardı, henüz commit etmedi. Bu sırada iki bağlantı aynı satırı okudu:

```
a görür: 800
b görür: 500
commit sonrası b görür: 800
```

Değişikliği yapan bağlantı kendi geçici değerini (800) görür. İkinci bağlantı ise commit edilmemiş değişikliği görmez ve 500 okur. Commit'ten sonra ikinci bağlantı da 800 görür.

11.9.2 Havale örneği

Havale programı iki parçadır. `main` bellekte bir veri tabanı kurar, iki hesap ekler ve iki havale dener:

```
static final String YATIR =
    "UPDATE hesaplar SET bakiye = bakiye + ? WHERE id = ?";
static final String CEK =
    "UPDATE hesaplar SET bakiye = bakiye - ? WHERE id = ?";

public static void main(String[] args) throws SQLException {
  String url = "jdbc:sqlite::memory:";  // bellekte, geçici
  try (Connection conn = DriverManager.getConnection(url);
       Statement st = conn.createStatement()) {
    st.execute("CREATE TABLE hesaplar (id INTEGER PRIMARY KEY, "
        + "sahip TEXT, bakiye INTEGER CHECK (bakiye >= 0))");
    st.execute("INSERT INTO hesaplar(sahip, bakiye) "
        + "VALUES ('Ali', 1000), ('Ece', 500)");
    havale(conn, 1, 2, 300);    // Ali → Ece 300 TL
    havale(conn, 1, 2, 5000);   // Ali'nin parası yetmez
    yazdir(conn);
  }
}
```

`hesaplar` tablosunun `bakiye` sütununda `CHECK (bakiye >= 0)` kısıtı vardır. Bakiyeyi eksiye düşüren her komut bir `SQLException` fırlatır. İkinci havalede Ali'nin 700 TL'si 5000 TL'lik havaleye yetmez. `havale` metodu şöyledir:

```
static void havale(Connection conn, int kimden, int kime,
                   int tutar) throws SQLException {
  conn.setAutoCommit(false);         // işlem başlar
  try (PreparedStatement p1 = conn.prepareStatement(YATIR);
       PreparedStatement p2 = conn.prepareStatement(CEK)) {
    p1.setInt(1, tutar);  p1.setInt(2, kime);
    p1.executeUpdate();              // alıcıya yatır
    p2.setInt(1, tutar);  p2.setInt(2, kimden);
    p2.executeUpdate();              // göndericiden çek
    conn.commit();                   // kalıcı yap
    System.out.println(tutar + " TL aktarıldı.");
  } catch (SQLException e) {
    conn.rollback();                 // geri al
    System.out.println("Geri alındı: " + tutar + " TL");
  } finally {
    conn.setAutoCommit(true);
  }
}
```

Metot önce otomatik onayı kapatır; işlem başlar. İki `PreparedStatement` ile önce alıcıya yatırır, sonra göndericiden çeker. İki komut da başarılıysa `commit()` ikisini birlikte kalıcı yapar. Herhangi bir komut `SQLException` fırlatırsa denetim `catch`'e geçer ve `rollback()` işlemdeki bütün değişiklikleri geri alır. `finally` bloğu, hangi yoldan gelinirse gelinsin, otomatik onayı yeniden açar; bağlantı sonraki kullanıcılar için varsayılan hâline döner. `yazdir` metodu, `listele` gibi bir `SELECT` döngüsüdür ve her hesabın sahibini ve bakiyesini yazar. Program çalıştırıldığında şu çıktıyı verir:

```
300 TL aktarıldı.
Geri alındı: 5000 TL
Ali: 700
Ece: 800
```

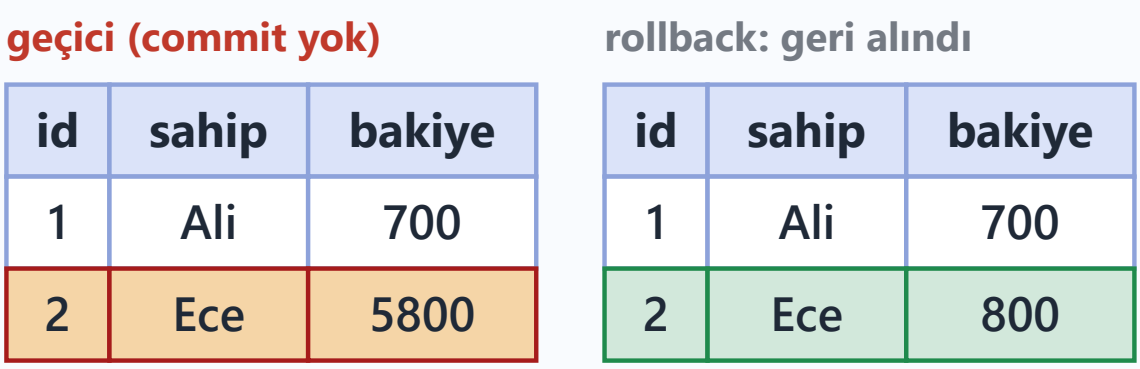
11.9.3 Kod izleme: `commit` ve `rollback`

Tablo 11.23 iki havale boyunca bakiyelerin ve işlemin durumunu verir. “Geçici” sütunu commit edilmemiş değişiklikleri gösterir.

Adım	Satır	Olay	Ali	Ece	Durum	Yeni çıktı
1	3	`setAutoCommit(false)`: birinci havale, işlem başlar	1000	500	kalıcı	—
2	6–7	Ece'ye 300 yatırılır	1000	800	geçici	—
3	8–9	Ali'den 300 çekilir; kısıt sağlanıyor	700	800	geçici	—
4	10–11	`commit()`: iki değişiklik birlikte kalıcı	700	800	kalıcı	`300 TL aktarıldı.`

Adım	Satır	Olay	Ali	Ece	Durum	Yeni çıktı
5	16	`finally` : otomatik onay yeniden açılır	700	800	kalıcı	—
6	3, 6–7	İkinci havale: Ece'ye 5000 yatırılır	700	5800	geçici	—
7	8–9	700 − 5000 < 0: `CHECK` bozulur, `SQLException`	700	5800	geçici	—
8	12–14	`rollback()` : geçici yatırma geri alınır	700	800	kalıcı	`Geri alındı: 5000 TL`

Tablo 11.23: Havale programının izlenmesi (satır numaraları `havale` metoduna göredir).



Şekil 11.28: İkinci havalede geçici yatırmadan sonra (sol) ve `rollback()` 'ten sonra (sağ).

Birinci havale sorunsuzdur: iki komut geçici olarak uygulanır, `commit()` ikisini birlikte kalıcı yapar. İkinci havalede yatırma komutu başarılı olur ve Ece'nin bakiyesi geçici olarak 5800'e çıkar (Şekil 11.28, sol). Çekme komutu Ali'nin bakiyesini −4300 yapacağı için `CHECK` kısıtına takılır ve `SQLException` fırlar. Ali'nin bakiyesi hiç değişmez: reddedilen komut uygulanmaz. `commit()` satırına gelinmez. `catch` bloğundaki `rollback()`, işlemde o ana kadar yapılmış olan yatırmayı da geri alır; Ece'nin bakiyesi 800'e döner. Para ne kaybolmuş ne çoğalmıştır: başta 1500 TL vardı, sonda da 700 + 800 = 1500 TL vardır.

Bu programın değeri, işlem olmadan ne olacağına bakınca anlaşılır. Aynı metottan `setAutoCommit(false)`, `commit()` ve `rollback()` çıkarılıp program yeniden çalıştırıldı:

```
300 TL aktarıldı.
Geri alındı: 5000 TL
  ([SQLITE_CONSTRAINT_CHECK] A CHECK constraint failed (CHECK constraint failed: bakiye >=
0))
Ali: 700
Ece: 5800
```

İkinci havalede yatırma komutu çalışır çalışmaz kalıcı oldu. Çekme komutu reddedildi ama geri alınacak bir işlem yoktu. Program yine “Geri alındı” yazdı, ama yanlış: Ece'nin hesabında 5800 TL vardır ve 5000 TL yoktan var olmuştur. Toplam para 6500 TL'ye çıkmıştır.

Sık yapılan hata

`catch` bloğunda `rollback()`'i unutmak. Otomatik onay kapalıyken bir komut başarısız olur ve `rollback()` çağrılmazsa, işlemin başarılı komutları geçici hâlde bekler. Ardından `finally`'deki `setAutoCommit(true)` çalışır. JDBC kuralına göre işlem sürerken otomatik onayı açmak işlemi *commit eder*. Denemede Ece'ye yatırılan 5000 TL bu yolla kalıcı oldu: Ali 1000, Ece 5500. Hata yakalandı ama veri bozuldu. Hata yolunda `rollback()` her zaman çağrılır.

11.9.4 Toplu işlem

Bir ölçüm cihazının 1000 değerini tabloya yazmak için 1000 `INSERT` gerekir. Bu komutlar tek tek gönderilebilir ya da bir kuyrukta biriktirilip birlikte gönderilebilir.

Tanım 11.14

Toplu işlem (*batch update*), birçok komutu `addBatch()` ile bir kuyrukta biriktirip `executeBatch()` ile birlikte göndermektir. `executeBatch()` her komutun etkilediği satır sayısını bir `int[]` dizisi olarak döndürür.

```
conn.setAutoCommit(false);
try (PreparedStatement ps = conn.prepareStatement(
        "INSERT INTO olcumler(deger) VALUES(?)")) {
  for (int i = 1; i <= 1000; i++) {
    ps.setInt(1, i);
    ps.addBatch();
  }
  int[] sonuc = ps.executeBatch();
  conn.commit();
  System.out.println(sonuc.length + " komut, ilki "
      + sonuc[0] + " satır etkiledi");
} catch (SQLException e) {
  conn.rollback();
}
conn.setAutoCommit(true);
```

```
1000 komut, ilki 1 satır etkiledi
1000 satır, toplam 500500
```

Döngünün her turunda yer tutucu doldurulur ve `addBatch()` doldurulmuş komutu kuyruğa ekler. Döngü bittiğinde kuyrukta 1000 komut vardır. `executeBatch()` hepsini gönderir ve 1000 öğeli bir dizi döndürür; her öğe 1'dir, çünkü her ekleme bir satır etkilemiştir. Çıktının ikinci satırı denetim için eklendi. Tabloda 1000 satır vardır; değerlerin toplamı 1 + 2 + … + 1000 = 1000 · 1001 / 2 = 500.500'dür.

Bu kodun hızı iki kaynaktan gelir: tek işlem ve toplu gönderim. Otomatik onay açıkken her `INSERT` ayrı bir işlemdir. SQLite her işlemin sonunda kalıcılığı sağlamak için veriyi diske yazdırır ve yazmanın bitmesini bekler. Disk üzerindeki bir `.db` dosyasıyla üç ölçüm yapıldı (Tablo 11.24).

Yöntem	1. çalıştırma	2. çalıştırma	3. çalıştırma
Tek tek, otomatik onay açık	1841	2047	1864
`addBatch` + tek işlem	7	8	9
Tek tek `executeUpdate` + tek işlem	2	2	2

Tablo 11.24: Disk üzerindeki SQLite dosyasına 1000 satır ekleme süreleri (üç çalıştırma, ms). Süreler makineye ve çalıştırmaya göre değişir.

Otomatik onay açıkken 1000 satır yaklaşık 2 saniye sürdü; tek işlemle aynı iş birkaç milisaniyede bitti. Fark yüzlerce kattır. Tablonun son satırı öğretici bir ayrıntı taşır: SQLite'ta asıl kazanç tek işlemden gelir. Komutlar tek tek gönderilse bile, tek işlem içinde olduklarında toplu gönderim kadar hızlıdır. Bunun nedeni SQLite'ın programın içinde çalışmasıdır; komut göndermek bir metot çağrısından ibarettir. MySQL gibi bir sunucuda ise her komut ağ üzerinden bir gidiş-dönüş ister ve `addBatch` bu gidiş-dönüşlerin sayısını da azaltır.

Bütün kuyruk tek işlem olduğu için biri başarısız olursa `rollback()` kuyruğun tamamının etkisini geri alır. Bunu görmek için tabloya `CHECK (deger <> 500)` kısıtı eklenip aynı kod çalıştırıldı:

```
SQLiteException: [SQLITE_CONSTRAINT_CHECK] A CHECK constraint failed (CHECK constraint
failed: deger <> 500)
tabloda 0 satır
```

500. komut reddedildi, `catch` bloğu `rollback()` çağırdı ve tablo boş kaldı: ya hepsi ya hiçbiri. Standart, toplu işlemdeki hatanın `SQLException`'ın bir alt sınıfı olan `BatchUpdateException` ile bildirilmesini öngörür. SQLite sürücüsü burada kendi `SQLiteException`'ını fırlattı. İkisi de `SQLException` olduğu için `catch (SQLException e)` iki durumu da yakalar.

Çok büyük yüklemelerde kuyruk parça parça gönderilir. Örneğin her 500 `addBatch`'te bir `executeBatch` çağrılır. Böylece bir milyon komut bellekte birikmez ve bellek şişmez; işlem yine tek bir `commit` ile bitebilir.

11.10 Örnek problemler ve alıştırmalar

Örnek 11.2

Boş bir veri tabanında aşağıdaki komutlar sırayla çalışıyor. Son `SELECT` hangi satırları döndürür?

```
CREATE TABLE users (id INTEGER PRIMARY KEY, name TEXT);
INSERT INTO users(name) VALUES ('Ali');
INSERT INTO users(name) VALUES ('Can');
DELETE FROM users WHERE id = 2;
INSERT INTO users(name) VALUES ('Ece');
UPDATE users SET name = 'Efe' WHERE id = 1;
SELECT * FROM users;
```

Ali 1, Can 2 numarasını alır. Can silinince tablodaki en büyük `id` 1 olur. Ece en büyük `id` + 1 = 2'yi alır: silinen numara yeniden kullanılmıştır. `UPDATE` yalnız 1 numaralı satırı değiştirir; Ali'nin adı Efe olur. Sonuç **(1, Efe)** ve **(2, Ece)**'dir. Komutlar JDBC ile çalıştırılarak doğrulandı. Tablo `AUTOINCREMENT` ile tanımlansaydı Ece 3 alırdı.

Örnek 11.3

Kişi programı `kisiler.db` ile ikinci kez çalıştırılıyor. Çıktının ilk bloğu (ilk `---` 'ye kadar) ve son bloğu nedir?

Dosya kalıcıdır. İlk çalıştırmadan Ayşe (`id` 2) kalmıştı ve `CREATE TABLE IF NOT EXISTS` tabloyu silmez. Yeni kayıtlar en büyük `id` + 1 kuralıyla 3 (Ahmet) ve 4 (Ayşe) numaralarını alır. `guncelle` Ahmet'i Akdeniz 35 yapar. İlk blok: 2 Ayşe Demir 40, 3 Ahmet Akdeniz 35, 4 Ayşe Demir 40. `sil` Ahmet'i siler. Son blok: 2 Ayşe Demir 40, 4 Ayşe Demir 40. Aynı kişi artık iki kez kayıtlıdır. Programın ikinci çalıştırması gerçekten şu çıktıyı verdi:

```
2 Ayşe Demir 40
3 Ahmet Akdeniz 35
4 Ayşe Demir 40
---
2 Ayşe Demir 40
4 Ayşe Demir 40
---
```

Bu sonuç bir tasarım eksiğini gösterir. Aynı kişinin iki kez eklenmesi istenmiyorsa tabloya `UNIQUE(ad, soyad)` gibi bir kısıt konur ya da program eklemeden önce kişinin var olup olmadığını sorgular.

Örnek 11.4

Bir giriş ekranı şu kodla denetleniyor. Tabloda üç kişi var. Kullanıcı ad olarak `ahmet`, şifre olarak `' OR '1'='1` yazarsa oluşan SQL nedir, kaç satır döner, giriş başarılı olur mu?

```
String sql = "SELECT * FROM kisiler WHERE ad = '" + ad
           + "' AND sifre = '" + sifre + "'";
ResultSet rs = stmt.executeQuery(sql);
boolean girisBasarili = rs.next();
```

Birleştirme şu metni üretir: `SELECT * FROM kisiler WHERE ad = 'ahmet' AND sifre = '' OR '1'='1'`. SQL'de `AND`, `OR`'dan önce uygulanır. Koşul `(ad = 'ahmet' AND sifre = '') OR '1'='1'` olarak okunur. Sağ taraf her satırda doğrudur; **3 satır** döner. `rs.next()` `true` döndürür ve giriş **şifresiz başarılı olur**. Çözüm `WHERE ad = ? AND sifre = ?` ile `PreparedStatement` kullanmaktır; aynı girdiyle 0 satır döner. Üçü de çalıştırılarak doğrulandı:

```
SELECT * FROM kisiler WHERE ad = 'ahmet' AND sifre = '' OR '1'='1'
giriş: true, satır: 3
PreparedStatement giriş: false
doğru şifre: true
SELECT * FROM kisiler WHERE ad = 'ahmet' --' AND sifre = 'yanlis'
yorumla giriş: true
```

Çıktının son iki satırı başka bir saldırı biçimini gösterir. Ad alanına `ahmet' --` yazılırsa `--` SQL açıklaması başlatır ve şifre koşulu hiç okunmaz; yanlış şifreyle bile giriş başarılı olur. Gerçek sistemlerde şifreler zaten düz metin olarak değil, özetlenerek (*hash*) saklanır.

Örnek 11.5

`havale` metodundan `setAutoCommit(false)`, `commit()` ve `rollback()` satırları çıkarılsaydı program sonunda Ali'nin ve Ece'nin bakiyesi ne olurdu?

Birinci havale aynı biçimde başarılı olur: Ali 700, Ece 800. İkinci havalede yatırma hemen kalıcı olur ve Ece 5800'e çıkar. Çekme `CHECK` yüzünden başarısız olur, geri alacak işlem yoktur. Sonuç: **Ali 700, Ece 5800**. 5000 TL yoktan var olmuştur ve program yine de “Geri alındı” yazar. Program bu biçimde çalıştırılarak doğrulandı.

Sunumdaki dört alıştırmanın çözümleri de JDK 21 ve SQLite sürücüsüyle denendi. **Birinci alıştırma** silme programını `executeUpdate`'in dönüş değerine göre ileti verecek biçimde düzeltir:

```
int sayi = stmt.executeUpdate(sql);
if (sayi > 0) System.out.println("Veri silindi!");
else System.out.println("Kayıt bulunamadı.");
```

Program aynı satırı iki kez silmeye çalıştığında önce "Veri silindi!", sonra "Kayıt bulunamadı." yazdı. **İkinci alıştırmada** `ogrenciler(id, ad, vize, final)` tablosuna Ada (60, 70), Bora (30, 50) ve Cem (80, 40) bir döngüde `PreparedStatement` ile eklenir. Ortalaması (vize yüzde 40, final yüzde 60) 50 ve üstü olanlar büyükten küçüğe listelenir:

```
SELECT ad, vize * 0.4 + final * 0.6 AS ort FROM ogrenciler
WHERE vize * 0.4 + final * 0.6 >= 50 ORDER BY ort DESC
```

Ada'nın ortalaması 60 · 0,4 + 70 · 0,6 = 24 + 42 = 66'dır. Bora'nınki 12 + 30 = 42, Cem'inki 32 + 24 = 56'dır. Bora elenir; çıktı `Ada 66.0` ve `Cem 56.0`'dır. Değer `getDouble("ort")` ile okunur. `WHERE`'de ifade yeniden yazılmıştır, çünkü mantıksal sırada `WHERE`, `SELECT`'ten önce gelir ve `ort` takma adı henüz tanımlı değildir. `ORDER BY` ise takma adı kullanabilir. Katsayılar `0.4` gibi ondalıklı yazıldığı için sonuç `REAL`'dir. `(vize * 4 + final * 6) / 10` gibi bir tam sayı ifadesi ise SQLite'ta tam sayı bölmesi yapar ve kesirli kısmı atar.

Üçüncü alıştırma menülü bir programdır: 1-Ekle, 2-Güncelle, 3-Sil, 4-Listele. Menü `Scanner` ve `switch` ile kurulur. `nextInt()` satır sonunu tamponda bıraktığı için ardından bir `nextLine()` çağrılır; yoksa sonraki ad okuması boş dizgi döner. Güncelleme `UPDATE … WHERE id = ?`, silme `DELETE … WHERE id = ?` ile yapılır. `executeUpdate`'in döndürdüğü sayı 0 ise o `id`'li kullanıcı yoktur ve "Kullanıcı bulunamadı." yazılır. **Dördüncü alıştırma** oyun skorlarını `skorlar(oyuncu, puan)` tablosuna `INSERT INTO skorlar(oyuncu, puan) VALUES(?, ?)` ile kaydeder. En yüksek 10 skor `SELECT oyuncu, puan FROM skorlar ORDER BY puan DESC LIMIT ?` ve `ps.setInt(1, 10)` ile listelenir. `LIMIT`'in değeri de bir yer tutucuyla verilebilir; 12 skorlu bir tabloda sorgu büyükten küçüğe 10 satır döndürdü.

11.11 Sık yapılan hatalar

Bölüm boyunca anılan hatalar burada gerçek iletileriyle toplanır. Tablo 11.25 JDBC ve temel SQL hatalarını, Tablo 11.26 ilişki, gruplama ve işlem hatalarını verir. İletiler SQLite JDBC 3.40.0.0'ın `getMessage()` çıktısından alınmıştır; sürücü her iletinin başına `[SQLITE_ERROR] SQL error or missing database` gibi bir hata kodu ekler, tabloda yalnız parantez içindeki asıl ileti verilir.

Hata	Örnek ve ileti
Sürücü sınıf yolunda değil	Jar eklenmemiş: `No suitable driver found for jdbc:sqlite:sample.db`
`SQLException`'ı ele almamak	`try` ya da `throws` yok (derleme hatası): `unreported exception SQLException; must be caught or declared to be thrown`
Tablo yok ya da adı yanlış	Tablo oluşturulmadan sorgu: `no such table: users`
Metni tırnaksız yazmak	`VALUES(Ahmet, …)` : `no such column: Ahmet`
Kesme işaretli metni birleştirmek	`VALUES('Ali'nin')` : `near "nin": syntax error`
Yanlış `execute` metodu	`executeQuery` ile `INSERT` : `query does not return ResultSet` (satır yine de eklenir)
Sütun sırasını 0'dan başlatmak	`rs.getString(0)` : `column 0 out of bounds [1,3]`
`WHERE`'siz `UPDATE` / `DELETE`	Hata vermez; bütün satırları değiştirir ya da siler (üç satırlı tabloda 3 döner)

Tablo 11.25: JDBC ve temel SQL'de sık yapılan hatalar.

Hata	Örnek ve sonuç
`= NULL` ile karşılaştırmak	`WHERE bolum_id = NULL` hata vermez, 0 satır döner. Doğrusu `IS NULL`.
`ON`'suz birleştirme	`FROM ogrenciler, bolumler` : her satır her satırla eşleşir, 4 × 3 = 12 satır.
Belirsiz sütun adı	İki tabloda da `ad` var, tablo adı yazılmamış: `ambiguous column name: ad`
`WHERE`'de toplama fonksiyonu	Grup koşulu `HAVING`'e yazılır: `misuse of aggregate: SUM()`
Yabancı anahtar denetimini açmamak	SQLite'ta `PRAGMA foreign_keys = ON` olmadan `bolum_id = 9` sessizce kabul edilir.
`wasNull()`'u geç çağırmak	Son okunan sütuna bakar; ilgili `getInt`'in hemen ardından çağrılır.
Gruplanmayan sütunu seçmek	`SELECT musteri, tutar … GROUP BY musteri` : SQLite gruptaki satırlardan birinin tutarını getirir; birçok veri tabanında hatadır.
Otomatik onay açıkken binlerce `INSERT`	Her satır ayrı işlem: çok yavaş. `setAutoCommit(false)`, `addBatch`, `commit`.
Hata yolunda `rollback()`'i unutmak	`finally`'deki `setAutoCommit(true)` yarım işlemi commit eder.

Tablo 11.26: İlişkiler, gruplama ve işlemlerde sık yapılan hatalar.

Birinci tablonun çalışma zamanı satırları tek bir programda yeniden üretildi:

```
tablo yok -> [SQLITE_ERROR] SQL error or missing database (no such table: users)
tırnaksız -> [SQLITE_ERROR] SQL error or missing database (no such column: Ahmet)
kesme işareti -> [SQLITE_ERROR] SQL error or missing database (near "nin": syntax error)
yanlış metot -> query does not return ResultSet
indis 0 -> column 0 out of bounds [1,3]
indis 4 -> column 4 out of bounds [1,3]
WHERE'siz UPDATE -> 3
WHERE'siz DELETE -> 3
kalan 0
tablo duruyor mu -> 1
sınıf: org.sqlite.SQLiteException, kod: 1, SQLState: null
```

Son satır, yakalanan istisnanın kendisini inceler. Sınıfı `org.sqlite.SQLiteException`'dır; bu sınıf `SQLException`'ın bir alt sınıfıdır. `getErrorCode()` SQLite'ın hata kodunu (1, genel SQL hatası) verir. Hataların çoğunda program derlenir ve çalışır; sorun ancak çalışma zamanında, çoğu zaman da sessizce ortaya çıkar. Bu yüzden JDBC kodu yazıldıktan sonra tablonun gerçek içeriği, DB Browser for SQLite ya da bir `SELECT` ile mutlaka denetlenir.

11.12 Özet

- Veri tabanı, kalıcı, düzenli ve sorgulanabilir veri topluluğudur; onu VTYS yönetir. Dosyaya göre hızlı arama, güvenli eş zamanlı erişim, kısıtlarla tutarlılık, alan düzeyinde yetki ve hatadan sonra kurtarma sağlar.
- İlişkisel model veriyi tablolarda tutar: sütun bir özellik, satır bir kayıttır; birincil anahtar her satırı tekil olarak tanımlar. Tablo bir sınıfa, sütun bir alana, satır bir nesneye benzer.
- SQLite gömülü, sunucusuz ve tek dosyalı bir ilişkisel veri tabanıdır. Java'dan kullanmak için sürücü jar'ı sınıf yolunda olmalıdır; adres `jdbc:sqlite:sample.db` biçimindedir.
- Temel SQL komutları CRUD işlemlerini karşılar: `INSERT`, `SELECT`, `UPDATE`, `DELETE`. `WHERE` satırları süzer, `ORDER BY` sıralar, `LIMIT` sınırlar. `WHERE`'siz `UPDATE` ve `DELETE` bütün satırları etkiler.
- `INTEGER PRIMARY KEY`'e değer verilmezse SQLite en büyük `id` + 1 atar; `AUTOINCREMENT` kullanılmış bir numarayı yeniden vermez.
- Normalleştirme her bilgiyi tek yerde tutar ve güncelleme, ekleme, silme anomalilerini önler. Tablolar yabancı anahtarla bağlanır; SQLite'ta denetim `PRAGMA foreign_keys = ON` ile açılır.
- `JOIN … ON` yalnız eşleşen satırları, `LEFT JOIN` soldaki tablonun bütün satırlarını verir. `ON` unutulursa Kartezyen çarpım oluşur.
- `GROUP BY` satırları gruplar; `WHERE` gruplamadan önce satırları, `HAVING` gruplamadan sonra grupları süzer. Mantıksal sıra: `FROM` → `WHERE` → `GROUP BY` → `HAVING` → `SELECT` → `ORDER BY` → `LIMIT`.
- `NULL` bilinmeyen değerdir; `IS NULL` ile denetlenir. `getInt` `NULL` için 0 döndürür; ayırt etmek için hemen ardından `wasNull()` çağrılır.

- İndeks aramayı tarama yerine B-ağacı aramasına çevirir; bedeli her yazmada indeksin de güncellenmesidir.
- JDBC zinciri: `DriverManager` → `Connection` → `Statement` / `PreparedStatement` → `ResultSet`. Bunlar arayüzdür; sürücü onları uygular.
- `execute` tablo işlemleri için `boolean`, `executeUpdate` etkilenen satır sayısını, `executeQuery` bir `ResultSet` döndürür. Sonuç `while (rs.next())` ile okunur; sütun numaraları 1'den başlar.
- Kullanıcı girdisi SQL metnine eklenmez; her zaman `?` ve `setXxx` ile verilir. Aksi hâlde SQL enjeksiyonu oluşur.
- DAO bir tablonun SQL'ini tek sınıfta toplar ve satırları nesne olarak döndürür.
- İşlem: `setAutoCommit(false)`, `commit()`, hata durumunda `rollback()`; ya hepsi ya hiçbiri (ACID). Toplu eklemeler tek işlemde yapılır.
- Kaynaklar try-with-resources ile açılır ve `rs`, `stmt`, `conn` sırasıyla kapanır. `SQLException` kontrol edilen istisnadır.

11.13 Kavram listesi

veri tabanı *database* · ilişkisel model *relational model* · birincil anahtar *primary key* · yabancı anahtar *foreign key* · sorgu *query* · normalleştirme *normalization* · birleştirme (JOIN) *join* · toplama fonksiyonu *aggregate function* · indeks *database index* · sonuç kümesi (ResultSet) *result set* · önceden hazırlanan deyim (PreparedStatement) *prepared statement* · SQL enjeksiyonu *SQL injection* · veri erişim nesnesi (DAO) *data access object* · işlem (transaction) *transaction* · toplu işlem *batch update* · kontrol edilen istisna *checked exception* · try-with-resources

11.14 Kendini Dene

Sorular bölümün üç tablosunu kullanır: `ogrenciler` (Ada 1, Bora 2, Cem 1, Deniz `NULL`), `bolumler` (Bilgisayar, Elektrik, Makine) ve `siparisler` (Ali 120, Ece 80, Ali 50, Can 200, Ali 30, Ece 40). Cevaplar JDK 21 ve SQLite JDBC ile çalıştırılarak doğrulandı.

1. `SELECT COUNT(*) FROM ogrenciler WHERE bolum_id <> 2` kaç döndürür?

   Cevap

   2. Ada ve Cem (1) koşulu sağlar; Bora (2) sağlamaz. Deniz'in `bolum_id`'si `NULL`'dur ve `NULL <> 2` doğru değil, bilinmiyordur; Deniz sayılmaz.

2. `id INTEGER PRIMARY KEY` sütunlu bir tabloya üç satır eklenir (1, 2, 3), 2 numaralı satır silinir ve

yeni bir satır eklenir. Tablodaki numaralar nedir?

Cevap

1, 3, 4. Yeni satır en büyük `id` + 1 = 4 alır. Ortadaki boşluk doldurulmaz; yalnız en büyük numara silinseydi o numara yeniden kullanılırdı.

3. `SELECT musteri, SUM(tutar) FROM siparisler WHERE tutar < 100 GROUP BY musteri HAVING COUNT(*) >= 2` ne döndürür?

Cevap

`WHERE` önce 120 ve 200'lük siparişleri eler. Ali'nin 50 ve 30'u, Ece'nin 80 ve 40'ı kalır; Can'ın hiç siparişi kalmaz. İki grubun da adedi 2'dir. Sonuç: **Ali 80** ve **Ece 120**.

4. `SELECT COUNT(*) FROM bolumler b LEFT JOIN ogrenciler o ON o.bolum_id = b.id` kaç döndürür? `LEFT` kaldırılırsa?

Cevap

Bilgisayar iki, Elektrik bir öğrenciyle eşleşir; Makine eşleşmez ama `LEFT JOIN` onu bir satırla korur: **4**. Düz `JOIN`'de Makine düşer: **3**. Deniz iki sorguda da yoktur, çünkü korunan taraf bölümlerdir.

5. Ad araması `"SELECT ad FROM kisiler WHERE ad = '" + girdi + "'"` ile yapılıyor. Kullanıcı `' OR 1=1 --` yazarsa üç kişilik tabloda kaç satır döner?

Cevap

SQL metni `SELECT ad FROM kisiler WHERE ad = '' OR 1=1 --'` olur. `--` sondaki tırnağı açıklamaya çevirir; koşul her satırda doğrudur. **3** satır döner. `PreparedStatement` ile 0 satır dönerdi.

6. Malzeme tablosunda zımba 40, makas 10, bant 25 var. Bir `Statement`'a sırayla `INSERT … ('kalem', 5)`, `UPDATE malzeme SET stok = stok + 10 WHERE stok < 20` ve `DELETE FROM malzeme WHERE stok > 100` komutları `addBatch` ile eklenip `executeBatch()` çağrılıyor. Dönen dizi nedir?

Cevap

[1, 2, 0]. Ekleme bir satır etkiler. Güncellemede stoğu 20'den az olanlar makas (10) ve yeni eklenen kalemdir (5); ikisi 20 ve 15 olur. Silinecek satır yoktur. `Statement`'ın `addBatch(String)` biçimi farklı SQL metinlerini de kuyruğa alır.

7. `ogrenciler` tablosu `id` sırasıyla okunuyor; her satırda `toplam += rs.getInt("bolum_id")` yapılıyor ve `rs.wasNull()` doğruysa `bos++`. Sonunda `toplam` ve `bos` kaçtır?

   Cevap

   `getInt` Deniz için 0 verir: 1 + 2 + 1 + 0 = **4**. `wasNull()` yalnız Deniz'de doğrudur: `bos` = **1**.

8. `uye(id INTEGER PRIMARY KEY, email TEXT UNIQUE)` boş. `setAutoCommit(false)`, `a@x` ve `b@x` eklenir, `rollback()`, `c@x` eklenir, `commit()`. Tabloda hangi satırlar vardır?

   Cevap

   Tek satır: **(1, c@x)**. `rollback()` ilk iki eklemeyi geri alır; geri alınan eklemeler numara da tüketmez. `c@x` boş tabloya eklenir ve 1 alır.

11.15 İleri okuma

- Oracle, *The Java Tutorials*, "JDBC Database Access": bağlantı kurmayı, sorgu çalıştırmayı, `PreparedStatement`'ı ve işlemleri kısa örneklerle anlatan resmî ve ücretsiz kaynaktır.
- SQLite belgeleri, "Datatypes In SQLite" ve "Appropriate Uses For SQLite": SQLite'ın esnek tip sistemini anlatır; sunucu tabanlı bir veri tabanının hangi durumlarda daha uygun olduğunu da açıklar.
- OWASP, *SQL Injection Prevention Cheat Sheet*: enjeksiyonun türlerini ve önceden hazırlanan deyimlerin neden birincil savunma olduğunu örneklerle anlatır.
- C. S. Horstmann, *Core Java, Volume II: Advanced Features* (12. baskı), "Database Programming" bölümü: JDBC'yi, işlemleri ve sonuç kümelerinin ileri özelliklerini ayrıntılı işler.

11.16 Kaynaklar

1. Oracle, *The Java Tutorials*: JDBC Database Access (Establishing a Connection; Processing SQL Statements with JDBC; Using Prepared Statements; Using Transactions).
2. Oracle, *Java SE 21 API Documentation*: `java.sql` paketi (`DriverManager`, `Connection`, `PreparedStatement`, `ResultSet`).
3. SQLite belgeleri, sqlite.org: *SQL As Understood By SQLite*; *Datatypes In SQLite*; *Appropriate Uses For SQLite*.
4. Xerial, *SQLite JDBC Driver*, github.com/xerial/sqlite-jdbc.
5. P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (Accessing Databases with JDBC).
6. C. S. Horstmann, *Core Java, Volume II: Advanced Features*, 12. baskı, Pearson (Database Programming).
7. OWASP, *SQL Injection Prevention Cheat Sheet*.

EK A

Terim Sözlüğü

Kitapta geçen İngilizce terimler ve her biri için kullanılan tek Türkçe karşılık. Sağdaki sayı, terimin ayrıntılı anlatıldığı bölümdür. Sözlüğün aranabilir hâli ders sayfasındadır.

A

acceptance testing → kabul testi 7
Sistemi kullanıcılarla birlikte gerçek verilerle sınayıp müşterinin sistemi kabul edip etmeyeceğini belirleyen test.

ACID → ACID 11
Bir işlemin dört özelliği: bölünmezlik (atomicity), tutarlılık (consistency), yalıtım (isolation) ve kalıcılık (durability).

adapter class → adaptör sınıfı 9
Bir dinleyici arayüzünün bütün metotlarını boş gövdelerle uygulayan hazır sınıf; ondan türeyen sınıf yalnız ilgilendiği metodu geçersiz kılar.

aggregate function → toplama fonksiyonu 11
Bir satır kümesinden tek değer üreten COUNT, SUM, AVG, MIN, MAX.

amortized cost → amortize maliyet 5
Tek tek bazen pahalı olan çağrıların uzun bir dizideki ortalama maliyeti; ArrayList'e sona ekleme ortalamada sabit iş yapar.

assertion → doğrulama (assert) 7
Beklenen ile gerçek sonucu karşılaştıran ya da bir varsayımı denetleyen ifade.

atomic operation → atomik işlem 10
Bölünmeden, tek adımda tamamlanan işlem (AtomicInteger).

auto-commit → otomatik onay 11
JDBC bağlantısının varsayılan ayarı: her komut kendi başına bir işlemdir ve çalışır çalışmaz kalıcı olur.

AWT (Abstract Window Toolkit) → AWT 9
Java'nın ilk grafik kütüphanesi (java.awt); bugün daha çok Graphics, Color, Font, yerleşim yöneticileri ve olay sınıfları için kullanılır.

B

batch update → toplu işlem 11
Birçok komutu addBatch ile biriktirip executeBatch ile birlikte gönderme.

binary heap → ikili yığın 5
Bir dizide saklanan, her ebeveynin çocuklarından küçük ya da onlara eşit olduğu tam ikili ağaç; PriorityQueue'nun iç yapısı.

black-box / white-box testing → kara kutu / beyaz kutu testi 7
Kodun içine bakmadan yalnız girdi ve beklenen çıktıyla yapılan test ile kodun yapısı bilinerek her dalı ve döngüyü en az bir kez çalıştırmayı amaçlayan test.

boundary value analysis → sınır değer analizi 7
Her denklik sınıfının uç değerlerini ve geçersiz bölgenin ilk değerlerini test girdisi olarak seçme.

breakpoint → kesme noktası 7
Hata ayıklayıcının programı durdurduğu satır.

buffer / buffering → tampon / tamponlama 8
Veriyi bellekte biriktirip toplu okuma-yazma yaparak disk erişimini azaltma.

busy waiting → meşgul bekleme 10
Bir koşulun döngüde sürekli sınanarak beklenmesi; çekirdeği boşuna meşgul eder.

byte stream / character stream → bayt akışı / karakter akışı 8
InputStream/OutputStream ile Reader/Writer sınıf aileleri.

C

capacity → kapasite 5
Dinamik dizideki iç dizinin toplam hücre sayısı; dolu hücre sayısı olan size() ile aynı değildir.

channel (NIO) → kanal 8
NIO'da tamponla çift yönlü okuma-yazma yapan bağlantı.

character encoding → karakter kodlaması 8
Karakterlerin hangi bayt dizileriyle saklanacağını belirleyen kural.

checked exception → kontrol edilen istisna 6
Derleyicinin yakalanmasını ya da bildirilmesini zorunlu tuttuğu istisna.

code coverage → kod kapsamı 7
Testlerin çalıştırdığı kod satırlarının oranı.

collection → koleksiyon 5
Birden çok nesneyi bir arada tutan veri yapısı.

Collections Framework → Koleksiyon Çerçevesi 5
Java'nın arayüz, sınıf ve algoritmalardan oluşan koleksiyon kütüphanesi.

collision → çakışma 5
İki farklı öğenin hash tablosunda aynı kovaya düşmesi; çakışan öğeler kovada bir zincir oluşturur.

comparator → karşılaştırıcı (Comparator) 5
İki nesneyi dışarıdan karşılaştıran, başka bir sıra tanımlayan nesne.

compare-and-set (CAS) → karşılaştır ve değiştir (CAS) 10
Değer hâlâ beklenen ise yenisini yazıp true döndüren, değilse dokunmayıp false döndüren tek atomik işlem.

component → bileşen 9
Pencereye yerleştirilen görsel öğe (JButton, JLabel…).

concurrency / parallelism → eş zamanlılık / paralellik 10
İşlerin zaman içinde iç içe ilerlemesi ile gerçekten aynı anda çalışması.

container → kapsayıcı 9
Başka bileşenleri içeren bileşen (JFrame, JPanel).

content pane → içerik bölmesi 9
Pencerenin başlık çubuğunun altında kalan iç alanı; add ile eklenen bileşenler buraya yerleşir.

context switching → bağlam değiştirme 10
İşlemcinin bir iş parçacığından ötekine geçmesi; çalışanın durumu (program sayacı, yazmaçlar, yığın işaretçisi) saklanır, sıradakininki yüklenir.

continuous integration (CI) → sürekli entegrasyon (CI) 7
Testleri programcıya sormadan otomatik çalıştıran sistemler (GitHub Actions, Jenkins).

critical section → kritik bölge 10
Paylaşılan veriye erişen, aynı anda yalnızca bir iş parçacığının çalıştırması gereken kod parçası.

custom exception → özel istisna sınıfı 6
Exception ya da RuntimeException'dan türetilen, uygulamaya özgü istisna.

D

daemon thread → arka plan iş parçacığı 10
JVM'in kapanmasını engellemeyen yardımcı iş parçacığı.

data access object (DAO) → veri erişim nesnesi (DAO) 11
Bir tablonun SQL işlemlerini tek sınıfta toplayıp nesne döndüren tasarım.

database → veri tabanı 11
Verileri düzenli, kalıcı ve sorgulanabilir biçimde saklayan sistem.

database index → indeks 11
Bir sütunun değerlerini sıralı tutup aramayı hızlandıran ek yapı.

deadlock → kilitlenme 10
İş parçacıklarının birbirinin tuttuğu kilidi sonsuza dek beklemesi.

debugger → hata ayıklayıcı 7
Programı adım adım çalıştırıp değişkenleri izlemeyi sağlayan araç.

debugging → hata ayıklama 7
Bir hatanın nedenini bulup düzeltme süreci.

decorator (wrapper stream) → sarmalayıcı akış 8
Başka bir akışı sarıp ona tampon gibi yetenek ekleyen akış.

defect / failure → kusur / arıza 7
Programcının yanılgısının koda yansıyan yanlış satırı ile bu kod çalışırken ortaya çıkan yanlış davranış (yanlış çıktı, çökme, donma).

deque (double-ended queue) → çift uçlu kuyruk 5
Her iki ucundan ekleme ve çıkarma yapılabilen kuyruk.

doubly linked list → çift yönlü bağlı liste 5
Her düğümün bir değer ile önceki ve sonraki düğümün referansını tuttuğu liste (LinkedList).

dynamic array → dinamik dizi 5
Dolunca daha büyük bir diziye kopyalanarak büyüyen dizi (ArrayList).

E

equivalence class → denklik sınıfı 7
Programda aynı davranışa yol açan girdilerin kümesi; her sınıftan tek bir temsilci seçmek yeterlidir.

error (Error class) → Error 6
Programın genellikle kurtaramayacağı ciddi sorunlar (OutOfMemoryError).

event → olay 9
Kullanıcının tıklama, tuşa basma gibi eylemi ya da sistemden gelen bildirim.

Event Dispatch Thread (EDT) → olay iş parçacığı (EDT) 9
Swing bileşenlerinin kurulduğu ve olayların işlendiği tek iş parçacığı.

event listener → olay dinleyici 9
Bir olay olunca çağrılacak metotları tanımlayan arayüz ya da nesne.

event-driven programming → olay güdümlü programlama 9
Akışı koddaki satır sırasının değil, olayların geliş sırasının belirlediği program.

exception → istisna 6
Program çalışırken normal akışı bozan, nesne olarak fırlatılan olay.

exception propagation → istisna yayılımı 6
Yakalanmayan istisnanın çağrı yığınında çağırana doğru ilerlemesi.

F

file pointer → dosya işaretçisi 8
RandomAccessFile'da okuma ve yazmanın yapılacağı konumu gösteren, her işlemde işlenen bayt sayısı kadar ilerleyen işaretçi.

finally block → finally bloğu 6
İstisna olsa da olmasa da çalışan blok.

foreign key → yabancı anahtar 11
Başka bir tablonun birincil anahtarını gösteren sütun.

G

garbage collector → çöp toplayıcı 7
Hiçbir referansın göstermediği nesnelerin belleğini geri alan JVM bileşeni.

generics → jenerikler 5
Sınıf ve metotların tip parametresiyle yazılması (List<String>).

graphical user interface (GUI) → grafik kullanıcı arayüzü (GUI) 9
Pencere, düğme, menü gibi öğelerle görsel etkileşim sağlayan arayüz.

H

happens-before → önce-olur 10
A işlemi B'den önce-olursa A'nın yazdığı her şeyin B'de kesinlikle görünmesini sağlayan geçişli ilişki.

hashing / hash table → hash / hash tablosu 5
Öğeyi hash koduyla hesaplanan kovaya yerleştirerek ortalama O(1) erişim sağlama.

I

integration test → tümleştirme testi 7
Birlikte çalışan bileşenleri gerçek bağımlılıklarıyla sınayan test.

iterator → yineleyici (Iterator) 5
Koleksiyonun öğelerini sırayla veren, gezinirken silmeye izin veren nesne.

J

JDBC (Java Database Connectivity) → JDBC 11
Java'nın veri tabanlarına bağlanmak için sunduğu standart API (java.sql); belirli bir veri tabanıyla konuşan kodu o veri tabanının JDBC sürücüsü sağlar.

JDBC driver → JDBC sürücüsü 11
Java'nın belirli bir veri tabanıyla konuşmasını sağlayan kütüphane (SQLite için sqlite-jdbc jar'ı).

join → birleştirme (JOIN) 11
İki tablonun satırlarını ON koşuluna göre eşleyen SQL işlemi.

L

layout manager → yerleşim yöneticisi 9
Bileşenlerin kapsayıcıda nasıl dizileceğini belirleyen nesne.

lazy evaluation → tembel değerlendirme 5
Stream ara işlemlerinin öğe işlemeyip yalnız bir tarif kaydetmesi; öğeler uç işlem çağrılınca işlenir.

linked list → bağlı liste 5
Her öğenin bir sonrakine bağlantı tuttuğu liste (LinkedList).

list → liste 5
Sıralı, indisli, tekrara izin veren koleksiyon.

lock → kilit 10
Bir kaynağa erişimi tek iş parçacığına veren nesne (ReentrantLock).

logic error → mantık hatası 6
Program çalışır ama yanlış sonuç üretir.

lost update → kayıp güncelleme 10
İki iş parçacığının aynı eski değeri okuyup üstüne yazması sonucu artışlardan birinin kaybolması.

M

magic number → sihirli sayı 8
Dosyanın türünü gösteren baştaki baytlar; her .class dosyası CA FE BA BE ile başlar.

main thread → ana iş parçacığı 10
JVM'in main() metodunu çalıştırmak için açtığı iş parçacığı.

map → eşleme 5
Anahtar–değer çiftlerini tutan, anahtarı tekil olan yapı.

marker interface → işaret arayüzü 8
Hiç metodu olmayan, tek görevi sınıfı işaretlemek olan arayüz (Serializable).

memory leak → bellek sızıntısı 7
Artık gerekmeyen nesnelere referans kaldığı için belleğin boşalmaması.

monitor → monitör 10
Her nesneye bağlı, synchronized ile alınan kilit ve bekleme mekanizması.

multithreading → çoklu iş parçacığı 10
Bir programın birden çok iş parçacığıyla çalışması.

mutual exclusion → karşılıklı dışlama 10
Kritik bölgede aynı anda yalnızca bir iş parçacığının bulunması.

N

natural ordering (Comparable) → doğal sıra (Comparable) 5
Bir sınıfın compareTo ile tanımladığı varsayılan sıralama.

normalization → normalleştirme 11
Tekrarı önlemek için veriyi her bilgi tek yerde olacak biçimde tablolara bölme.

O

one-to-many / many-to-many relationship → bire çok / çoka çok ilişki 11
Bir kaydın karşı tablodaki birden çok kayıtla eşleştiği ilişki (bir bölüm, çok öğrenci) ile iki yönde de çok kaydın eşleştiği, ara tablo gerektiren ilişki.

P

prepared statement → önceden hazırlanan deyim (PreparedStatement) 11
Parametreleri ? ile ayrı verilen, enjeksiyonu önleyen SQL deyimi.

primary key → birincil anahtar 11
Tablodaki her satırı tekil olarak tanımlayan sütun.

priority queue → öncelikli kuyruk 5
En öncelikli öğeyi önce veren kuyruk.

process → süreç 10
Kendi bellek alanı olan, çalışmakta olan program.

producer–consumer problem → üretici–tüketici problemi 10
Paylaşılan tampona üreten ve tüketen iş parçacıklarının eşgüdümü.

profiler → profil oluşturucu 7
Programın zamanını ve belleğini nerede harcadığını ölçen araç.

property-based testing → özellik tabanlı test 7
Rastgele girdilerle bir özelliğin her zaman sağlandığını sınayan test.

Q

query → sorgu 11
Veri tabanından veri isteyen SQL komutu (SELECT).

queue → kuyruk 5
İlk giren ilk çıkar (FIFO) ilkesiyle çalışan yapı.

R

race condition → yarış durumu 10
Sonucun iş parçacıklarının çalışma sırasına bağlı olması.

random access file → rastgele erişimli dosya 8
İşaretçiyi istenen konuma taşıyarak okunup yazılabilen dosya.

red-black tree → kırmızı-siyah ağaç 5
TreeSet'in öğeleri sıralı saklamak için içte kullandığı ağaç yapısı.

reentrant lock → yeniden girişli kilit 10
Kendisini tutan iş parçacığının aynı kilidi beklemeden yeniden almasına izin veren kilit; bir tutma sayısı saklar.

refactoring → yeniden düzenleme 7
Testler yeşil kalırken kodun temizlenmesi (tekrarları kaldırma, adları düzeltme); TDD döngüsünün son adımı.

regression test → regresyon testi 7
Değişiklikten sonra eski testlerin yeniden çalıştırılması.

relational model → ilişkisel model 11
Veriyi satır ve sütunlardan oluşan tablolarla tutan model.

result set → sonuç kümesi (ResultSet) 11
Bir sorgunun döndürdüğü satırları satır satır okutan nesne.

runtime error → çalışma zamanı hatası 6
Program çalışırken ortaya çıkan, istisnaya yol açan hata.

S

scheduler → çizelgeleyici 10
Hangi iş parçacığının ne zaman ve hangi çekirdekte çalışacağına karar veren JVM ve işletim sistemi bileşeni; programcı bu sırayı denetleyemez.

selector (NIO) → seçici 8
Tek iş parçacığıyla birden çok kanalı izleyen NIO nesnesi.

semaphore → semafor 10
Aynı anda en çok n iş parçacığının erişmesine izin veren sayaç.

serialization → serileştirme 8
Nesneyi bayt dizisine çevirip saklama ve geri okuma.

set → küme 5
Tekrar eden öğeye izin vermeyen koleksiyon.

shuffle → karıştırma 5
Öğelerin sırasını rastgele değiştirme (Fisher–Yates).

spurious wakeup → sahte uyanma 10
wait'teki iş parçacığının hiçbir notify olmadan uyanması; bu yüzden koşul if ile değil while ile denetlenir.

SQL injection → SQL enjeksiyonu 11
Kullanıcı girdisinin SQL komutuna karışıp komutu değiştirmesi saldırısı.

stable sort → kararlı sıralama 5
Eşit sayılan öğelerin kendi aralarındaki sırasını koruyan sıralama (Collections.sort'un kullandığı TimSort).

stack → yığın 5
Son giren ilk çıkar (LIFO) ilkesiyle çalışan yapı.

stack trace → yığın izi 6
İstisnanın tipini, mesajını ve çağrı zincirini gösteren çıktı.

stack unwinding → yığın çözülmesi 6
Uygun catch aranırken çağrı yığınında ilerleyip yol üzerindeki çerçevelerin atılması.

standard input / output → standart girdi / çıktı 8
Programın klavye (System.in) ve ekran (System.out) akışları.

static analysis → statik analiz 7
Kodu çalıştırmadan hata kalıplarını arayan inceleme (SpotBugs).

stream (I/O) → akış 8
Kaynaktan hedefe sırayla akan bayt ya da karakter dizisi.

synchronization → senkronizasyon 10
İş parçacıklarının paylaşılan veriye erişimini düzenleme.

T

test double → test dublörü 7
Birim testinde dış bağımlılığın yerine konan, onu taklit eden basit nesne.

test-driven development (TDD) → test güdümlü geliştirme 7
Önce testi, sonra onu geçiren kodu yazma yöntemi.

text file / binary file → metin dosyası / ikili dosya 8
Baytların karakter olarak okunduğu dosya ile ham veri tutan dosya.

thread → iş parçacığı 10
Bir programın içinde bağımsız ilerleyen yürütme akışı.

thread pool → iş parçacığı havuzu 10
Görevleri hazır bekleyen iş parçacıklarına dağıtan yapı (ExecutorService).

thread safety → iş parçacığı güvenliği 10
Bir sınıfın birden çok iş parçacığından aynı anda doğru kullanılabilmesi.

thread states → iş parçacığı durumları 10
NEW, RUNNABLE, BLOCKED, WAITING, TIMED_WAITING, TERMINATED.

throw / catch → fırlatmak / yakalamak 6
İstisnayı oluşturup göndermek ve onu catch bloğunda ele almak.

transaction → işlem (transaction) 11
Ya hepsi ya hiçbiri ilkesiyle yürütülen SQL komutları grubu.

try-catch block → try-catch bloğu 6
İstisna oluşabilecek kodu try, ele alan kodu catch içine alan yapı.

try-with-resources → try-with-resources 6
Açılan kaynakları blok bitince otomatik kapatan try biçimi.

type safety → tip güvenliği 5
Yanlış tipte değerlerin derleme zamanında yakalanması.

U

unchecked exception → kontrol edilmeyen istisna 6
RuntimeException soyundan gelen, derleyicinin denetlemediği istisna.

unit test → birim testi 7
Bir metodu ya da sınıfı tek başına sınayan otomatik test.

V

virtual thread → sanal iş parçacığı 10
İşletim sistemi yerine JVM'in yönettiği, milyonlarcası açılabilen iş parçacığı (Java 21); bekleme ağırlıklı çok sayıda kısa görev için uygundur.

EK B

Java ile Nesne Yönelimli Programlama İçin Ön Bilgiler

JAVA · ÖN BİLGİLER

Ders, bilgisayarın nasıl çalıştığından başlar ve kısa sürede Java programlarını yazmaya, derlemeye ve çalıştırmaya geçer. Bu sayfadaki 28 konu, dersin üzerine kurulduğu ön bilgileri altı grupta toplar: bilgisayar temelleri, sayı sistemleri, algoritmalar ve akış diyagramları, temel programlama kavramları, Java yazmak için gereken araçlar ve nesnelerin arkasındaki iki düşünme alışkanlığı.

Her kartta kısa bir açıklama, küçük bir örnek, konunun kullanıldığı slaytlara bağlantılar ve bir kontrol sorusu bulunur; sorunun üzerine tıklanınca yanıtı açılır.

Bilgisayar temelleri (3) **Sayı sistemleri ve veri gösterimi (4)**

Algoritmalar ve akış diyagramları (6) **Programlama kavramları (7)**

Araçlar ve kurulum (6) **Nesnelere hazırlık (2)**

Bilgisayar temelleri

Bir program saklanırken, belleğe yüklenirken ve çalışırken makinenin içinde olup bitenler.

CPU

Donanım: işlemci, bellek ve depolama

CPU · RAM · disk · girdi/çıktı

Program diskte bir dosya olarak durur. Çalışması için ana belleğe (RAM) yüklenir; işlemci de bellekteki verileri okuyup yazarak programın komutlarını tek tek yürütür. RAM hızlıdır ama elektrik kesilince içeriğini kaybeder, disk ise dosyaları saklamayı sürdürür.

NEDEN ÖNEMLİ

Bir Java programı çalışırken değişkenler, nesneler ve referanslar bellekte bulunur; program bittikten sonra yalnızca bir dosyaya yazılmış olan kalır.

KULLANILDIĞI YERLER **Bölüm 1.1 · Temel Parçalar** **Bölüm 1.1 · CPU – Bellek – Veri Yolu**

Bölüm 1.1 · Bellek Hiyerarşisi

KONTROL SORUSU

▶ **Bir program int puan = 90; atamasını yapıyor, ardından bilgisayar kapatılıyor. 90 değeri ertesi gün hâlâ yerinde midir?**

ÖRNEK (JAVA)

```
Runtime rt = Runtime.getRuntime();
System.out.println(rt.availableProcessors());          // örneğin 8 çekirdek
System.out.println(rt.maxMemory() / (1024 * 1024) + " MB"); // JVM'in
kullanabileceği bellek
```

/

Dosyalar, klasörler ve yollar

mutlak · göreli · uzantı

Yol (path) bir dosyayı adlandırır: mutlak yol kökten başlar (C:\Users\ali\ders\Merhaba.java), göreli yol ise o anki çalışma dizininden başlar (ders\Merhaba.java). Uzantı dosyanın türünü belirtir: .java kaynak kodu, .class derlenmiş bytecode'u gösterir. Windows bilinen uzantıları varsayılan olarak gizler; bu yüzden bir dosyanın adı fark edilmeden Merhaba.java.txt olabilir.

NEDEN ÖNEMLİ

javac ve java kaynak dosyanın bulunduğu klasörde çalıştırılır; dosya giriş çıkışı da dosyaları göreli ya da mutlak yollarla açar.

KULLANILDIĞI YERLER **Bölüm 2.2 · .java Dosyaları** **Bölüm 8 · java.io.File Sınıfı**

Bölüm 8 · File ile Dosya ve Klasör İşlemleri

KONTROL SORUSU

▶ **Çalışma dizini C:\ders iken bir program new File("notlar.txt") ile dosya açıyor. Bu hangi dosyadır?**

ÖRNEK (YOLLAR)

```
C:\Users\ali\ders\Merhaba.java    mutlak yol (Windows)
/home/ali/ders/Merhaba.java       mutlak yol (Linux, macOS)
ders\Merhaba.java                 C:\Users\ali dizinine göre göreli yol
Merhaba.java                      kaynak kod
Merhaba.class                     javac'ın ürettiği bytecode
```

01

Programlar, derleyiciler ve yorumlayıcılar

kaynak kod · makine kodu · bytecode

İşlemci yalnızca makine kodunu yürütür. Derleyici (compiler) programın tamamını önceden çevirir; yorumlayıcı (interpreter) ise programı adım adım okuyup yürütür. Java ikisini birleştirir: javac kaynak kodu platformdan bağımsız bytecode'a derler, Java Sanal Makinesi (JVM) de bu bytecode'u yorumlar ve program çalışırken sık kullanılan kısımları makine koduna derler.

NEDEN ÖNEMLİ

Derleme zamanı hatası ile çalışma zamanı hatası arasındaki fark ve javac ile java komutlarının görevleri bu iki adımlı süreçten çıkar.

KULLANILDIĞI YERLER **Bölüm 2.1 · Derlenen ve Yorumlanan Diller** **Bölüm 2.2 · Derleme** **Bölüm 2.2 · Soyutlama Düzeyleri**

KONTROL SORUSU

▶ **Windows'ta bir Merhaba.class dosyası üretildi. Bu dosya Linux'ta çalıştırılabilir mi?**

ÖRNEK (İKİ ADIM)

```
Merhaba.java  --javac-->  Merhaba.class  --java (JVM)-->  program çıktısı
(kaynak kod)               (bytecode)
```

Sayı sistemleri ve veri gösterimi

Sayıların ve karakterlerin bit olarak nasıl saklandığı ve bunun doğurduğu şaşırtıcı sonuçlar.

2 İkilik, onluk ve onaltılı sistem

2 tabanı · 10 tabanı · 16 tabanı

2 tabanında her basamak ikinin bir kuvveti değerindedir: $1101_2 = 8 + 4 + 0 + 1 = 13$. Onaltılı sistem dört biti 0–9, A–F arasındaki tek bir basamakta toplar; böylece $1111\ 1111_2 = FF_{16} = 255$ olur. Bir sayıyı art arda 2'ye bölerek ikilik sisteme çevirmek ve basamak değerlerini toplayarak geri dönmek rahatça yapılabilmelidir.

NEDEN ÖNEMLİ

Bit örüntüleri, ikiye tümleyen gösterim, renk değerleri ve karakter kodları bu tabanlarda yazılır; Java da ikilik ve onaltılı sabit değerleri kabul eder.

KULLANILDIĞI YERLER **Bölüm 1.1 · Basamak Değeri** **Bölüm 1.1 · İkiliye Çevirme: İzleme** **Bölüm 1.1 · Java'da Tabanlar**

KONTROL SORUSU

▶ **1011_2 ve $2A_{16}$ onluk sistemde kaçtır?**

ÖRNEK (JAVA)

```
System.out.println(Integer.toBinaryString(13));   // 1101
System.out.println(Integer.toHexString(255));      // ff
System.out.println(0b1101 + " " + 0x1F);           // 13 31
System.out.println(Integer.parseInt("101", 2));    // 5
```

± Bitler, baytlar ve tam sayı aralıkları

byte · int · long · ikiye tümleyen · taşma

n bit 2^n farklı değer tutabilir. Java işaretli tam sayıları ikiye tümleyen (two's complement) biçimde saklar; bu yüzden bir byte (8 bit) −128 ile 127 arasında, bir int (32 bit) ise yaklaşık −2,1 milyar ile 2,1 milyar arasında değer alır. Sonuç tipe sığmadığında hata oluşmaz, değer sessizce başa sarar.

NEDEN ÖNEMLİ

int ile long arasında seçim yapmak, faktöriyellerde, toplamlarda ve çarpmalarda taşmayı (overflow) fark etmek bu sınırları bilmeye bağlıdır.

KULLANILDIĞI YERLER **Bölüm 1.1 · İkiye Tümleyen** **Bölüm 1.1 · İkiye Tümleyen ve Taşma** **Bölüm 2.2 · Tam Sayı Tipleri** **Bölüm 2.2 · Taşma**

KONTROL SORUSU

▶ **8 bitte kaç farklı değer saklanabilir ve Java'daki byte tipinin aralığı nedir?**

ÖRNEK (JAVA)

```
byte b = (byte) 130;
System.out.println(b);                // -126
int max = Integer.MAX_VALUE;          // 2147483647
System.out.println(max + 1);          // -2147483648 (taşma)
System.out.println((long) max + 1);   // 2147483648
```

.1 Kesirli sayılar yaklaşıktır

double · yuvarlama hatası · tam sayı bölmesi

double bir ikilik kesir saklar; bu yüzden 0.1 gibi onluk kesirlerin çoğu ancak yaklaşık olarak tutulur ve toplamlarda küçük hatalar ortaya çıkar. İki double değeri bu nedenle == ile değil, bir tolerans payıyla karşılaştırılır. İki tam sayının bölümü de tam sayıdır: 7 / 2 işleminin sonucu 3, 7 / 2.0 işleminin sonucu ise 3.5'tir.

NEDEN ÖNEMLİ

Bu iki etki hesaba katılmazsa ortalamalar, birim dönüşümleri ve para hesapları sessizce yanlış sonuç verir.

KULLANILDIĞI YERLER **Bölüm 1.1 · Kesirli Sayılar** **Bölüm 2.2 · Ondalıklı Sayı Tipleri** **Bölüm 2.2 · Ondalık Sayıların Hassaslığı** **Bölüm 2.2 · Tam Sayı Bölmesi ve Dönüşüm**

KONTROL SORUSU

▶ **Para tutarları neden çoğu zaman kuruş ya da sent cinsinden tam sayı olarak saklanır?**

ÖRNEK (JAVA)

```
System.out.println(0.1 + 0.2);                          // 0.30000000000000004
System.out.println(0.1 + 0.2 == 0.3);                   // false
System.out.println(Math.abs(0.1 + 0.2 - 0.3) < 1e-9);   // true
System.out.println(7 / 2 + " " + 7 / 2.0);              // 3 3.5
```

A Karakterler: ASCII ve Unicode

char · karakter kodu · UTF-8

Her karakter bir sayı olarak saklanır: ASCII'de 'A' 65, 'a' 97, '0' ise 48'dir. Unicode bu numaralandırmayı bütün yazı sistemlerine genişletir; Türkçe harflerin de kodu vardır (ş 351'dir). Java'daki char 16 bitlik bir Unicode birimi tutar ve aritmetik işlemlerde kullanılabilir. Dosyalar metni çoğunlukla UTF-8 kodlamasıyla saklar.

NEDEN ÖNEMLİ

'7' karakterini 7 sayısına çevirmek gibi karakter aritmetiği, dizgilerin karşılaştırılması ve dosyadaki Türkçe metnin doğru okunması karakter kodlarına dayanır.

KULLANILDIĞI YERLER **Bölüm 1.1 · ASCII Tablosu** **Bölüm 1.1 · Unicode** **Bölüm 2.2 · char ve int** **Bölüm 1.1 · UTF-8**

KONTROL SORUSU

▶ **'a' karakterinin kodu 97'dir. System.out.println('a' + 1); ne yazdırır?**

ÖRNEK (JAVA)

```
char c = 'A';
System.out.println((int) c);          // 65
System.out.println((char) (c + 2));   // C
System.out.println('7' - '0');        // 7
System.out.println((int) 'ş');        // 351
```

Algoritmalar ve akış diyagramları

Kod yazmadan önce çözümü planlamak ve kodu elle izlemek.

→ Algoritma: girdi, işlem, çıktı

sonlu · kesin · sonlanan

Algoritma, verilen girdiyi istenen çıktıya dönüştüren ve her zaman duran, sonlu sayıda kesin adımdan oluşan bir dizidir. Kodlamaya geçmeden önce problem açıkça tanımlanır: ne verilmiştir, ne üretilmelidir ve hangi özel durumlar vardır (sıfır, negatif sayılar, boş bir liste).

NEDEN ÖNEMLİ

Dersteki her alıştırma ve örnek program bir algoritmadan yola çıkar; hataların çoğu Java söz diziminden değil, belirsiz bir problem tanımından kaynaklanır.

KULLANILDIĞI YERLER **Bölüm 1.3 · Algoritma Nedir?** **Bölüm 1.2 · Girdi, İşlem, Çıktı**

Bölüm 1.3 · Sorunu Tanımlama

KONTROL SORUSU

▶ **“x negatif olana kadar x'e 1 eklemeyi sürdür.” x 5'ten başlıyorsa bu bir algoritma mıdır?**

ÖRNEK (SÖZDE KOD)

```
Oku a, b
Eğer a > b ise
    max ← a
Değilse
    max ← b
Yaz max
```

≡ Sözde koddan koda

sözde kod · adım adım çeviri

Sözde kod (pseudocode) bir algoritmanın adımlarını söz dizimine takılmadan, sade ve düzenli bir dille anlatır. Ardından her satırın bir Java karşılığı yazılır: Oku Scanner ile okumaya, ← atamaya, Döngü bir döngüye, Yaz da System.out.println çağrısına dönüşür.

NEDEN ÖNEMLİ

Önce adımları yazmak, çözümü düşünmeyi derleyici hatalarıyla uğraşmaktan ayırır.

KULLANILDIĞI YERLER **Bölüm 1.3 · Sözde Kod** **Bölüm 1.3 · Koda Dönüştürme**

Bölüm 1.3 · ToplamaHesaplama İzleme

KONTROL SORUSU

▶ **Örnekte n = 4 için ne yazdırılır?**

ÖRNEK (SÖZDE KOD VE JAVA)

```
sum ← 0                          int sum = 0;
Döngü i = 1'den n'ye kadar        for (int i = 1; i <= n; i++) {
    sum ← sum + i                    sum = sum + i;
                                 }
Yaz sum                          System.out.println(sum);
```

◇ Akış diyagramları ve üç kontrol yapısı

sıra · seçim · tekrar

Akış diyagramı bir algoritmayı standart şekillerle çizer: başlangıç ve bitiş için oval, girdi ve çıktı için paralelkenar, hesaplama için dikdörtgen, karar için eşkenar dörtgen kullanılır. Her algoritma üç yapıyla kurulabilir: sıra, seçim (if) ve tekrar (döngü).

NEDEN ÖNEMLİ

Bir döngünün ya da iç içe kararların akışı çizildiğinde koşulun nerede sınandığı ve programın döngüden ne zaman çıktığı daha kolay görülür.

KULLANILDIĞI YERLER **Bölüm 1.4 · Temel Semboller** **Bölüm 1.4 · Semboller ve Java**

Bölüm 1.4 · Kontrol Mekanizmaları

KONTROL SORUSU

▶ **n > 0 gibi bir koşul hangi akış diyagramı şekline yazılır ve bu şeklin kaç çıkışı vardır?**

ÖRNEK (ŞEKİLLER VE JAVA)

```
( başla / bitir )    oval            program başlar / biter
/ girdi, çıktı /     paralelkenar    Scanner, System.out.println
[ hesaplama ]        dikdörtgen      sum = sum + i;
< koşul? >           eşkenar dörtgen if (n > 0), while (i <= n)
```

≣ Kodu elle izleme

izleme tablosu · adım adım değişken değerleri

İzleme, bir programı kâğıt üzerinde çalıştırmaktır: bir tablo her adımdan sonra her değişkenin değerini kaydeder. Bir döngünün çıktısını önceden kestirmenin, bir eksik ya da bir fazla adım (off-by-one) hatasını bulmanın ve başkasının kodunu anlamanın en güvenilir yoludur.

NEDEN ÖNEMLİ

Birçok slayt ve sınav sorusu kısa bir kod parçasının çıktısını sorar; izleme tablosu bu soruları bilgisayarsız yanıtlamayı sağlar.

KULLANILDIĞI YERLER **Bölüm 1.3 · ToplamaHesaplama İzleme**

Bölüm 3.1 · Sayı Toplama (for) **Bölüm 1.2 · Döngülü İşlemler**

KONTROL SORUSU

▶ **int x = 5; while (x > 1) x = x / 2; Döngü gövdesi kaç kez çalışır ve sonunda x kaçtır?**

ÖRNEK (İZLEME TABLOSU)

```
int s = 0;
for (int i = 1; i <= 4; i++) s += i;

//  i :  1   2   3   4
//  s :  1   3   6  10     -> i 5 olunca döngü biter
```

log Logaritma ve büyüme hızları

$\log_2 n \cdot n \cdot n \log n \cdot n^2 \cdot 2^n$

$\log_2 n$, 2'nin kaçıncı kuvvetinin n ettiğini söyler: $2^{10} = 1024$ olduğu için $\log_2 1024 = 10$'dur. Bir sayıyı art arda yarıya bölmek onu yaklaşık $\log_2 n$ adımda 1'e indirir; bu yüzden logaritma çok yavaş büyür: n bin kat artınca $\log_2 n$ yalnızca 10 kadar artar. Girdi boyutu n büyürken farklı fonksiyonlar çok farklı hızlarda büyür; küçükten büyüğe sıra 1, log n, n, n log n, n^2, 2^n'dir. n^2 büyümede n iki katına çıkınca iş dört katına çıkar; 2^n büyümede ise n'nin 1 artması bile işi iki katına çıkarır.

NEDEN ÖNEMLİ

Arama ve sıralama algoritmalarının, dizi ve koleksiyon işlemlerinin maliyeti O(1), O(log n), O(n), O(n log n) ve $O(n^2)$ gibi gösterimlerle karşılaştırılır. Düz özyinelemeli Fibonacci'nin üstel büyüyen çağrı sayısı da aynı fikirle açıklanır.

KULLANILDIĞI YERLER **Bölüm 3.3 · Karmaşıklık: O(1), O(n), $O(n^2)$**

Bölüm 1.3 · Karşılaştırma: Doğrusal ve İkili **Bölüm 5 · Map Sınıflarını Karşılaştırma**

KONTROL SORUSU

▶ **Sıralı bir dizide arama alanı her adımda yarıya iniyor. Dizi 1.000.000 öğeliyse en çok kaç adım gerekir?**

ÖRNEK (JAVA VE TABLO)

```
int n = 1000, adim = 0;
while (n > 1) {
    n = n / 2;                 // her turda yarıya
    adim++;
}
System.out.println(adim);     // 9  (log₂ 1000 ≈ 9,97)

//  n       log₂ n   n log₂ n    n²
//  10      ≈ 3,3    ≈ 33        100
//  1000    ≈ 10     ≈ 10.000    1.000.000
```

n!

Faktöriyel, toplam sembolü ve tümevarımsal tanımlar

n! · Σ · C(n, k) · başlangıç değeri ve kural

n! (n faktöriyel) 1'den n'ye kadar sayıların çarpımıdır ve 0! = 1 kabul edilir: 5! = 120. Σ sembolü bir toplamı kısaca yazar: 1'den n'ye kadar i'lerin toplamı n(n + 1)/2'dir. Bir dizi tümevarımsal olarak da tanımlanabilir: bir ya da birkaç başlangıç değeri verilir ve her terim öncekilerden bir kuralla bulunur. n! = n · (n − 1)! ve F(n) = F(n − 1) + F(n − 2) böyle tanımlardır. n öğeden k tanesini seçmenin yollarının sayısı C(n, k) = n! / (k! (n − k)!) kombinasyonudur; Pascal üçgeninde her sayı üstündeki iki sayının toplamıdır: C(n, k) = C(n − 1, k − 1) + C(n − 1, k).

NEDEN ÖNEMLİ

Özyinelemeli bir metot böyle bir tanımı doğrudan koda çevirir: başlangıç değeri taban durum, kural da özyinelemeli adım olur. Faktöriyel, Fibonacci, Catalan ve Bell serileri ile Pascal üçgeni bu gösterimle verilir.

KULLANILDIĞI YERLER **Bölüm 3.5 · Taban Durum ve Özyinelemeli Adım**

Bölüm 3.5 · Catalan Serisi **Bölüm 3.3 · Düzensiz Dizi: Kod İzleme**

KONTROL SORUSU

▶ **a(0) = 3 ve a(n) = a(n − 1) + 2 kuralıyla tanımlanan dizinin a(4) terimi kaçtır?**

ÖRNEK (JAVA'YA ÖN BAKIŞ)

```
static long faktoriyel(int n) {
    if (n == 0) return 1;              // başlangıç: 0! = 1
    return n * faktoriyel(n - 1);      // kural: n! = n · (n − 1)!
}
static long kombinasyon(int n, int k) {
    if (k == 0 || k == n) return 1;    // üçgenin kenarları
    return kombinasyon(n - 1, k - 1) + kombinasyon(n - 1, k);
}
// faktoriyel(5) == 120, kombinasyon(5, 2) == 10
```

Programlama kavramları

Dil ne olursa olsun her programı oluşturan temel yapı taşları.

x

Değişkenler, tipler ve atama

int · double · boolean · char · String

Değişken, bellekte adı olan ve hangi değerleri tutabileceğini tipinin belirlediği bir yerdir. Java tipleri derleme zamanında denetler: bir int "5" değerini tutamaz ve bir değişken okunmadan önce ona değer verilmiş olmalıdır. = işareti matematikteki eşitlik değil, atamadır (sağ tarafın değeri değişkene kopyalanır).

NEDEN ÖNEMLİ

Her program verisini değişkenlerde tutar; ilkel tipler ile referans tipleri ileride atandıklarında ya da metotlara aktarıldıklarında farklı davranır.

KULLANILDIĞI YERLER **Bölüm 2.2 · Değişkenler** **Bölüm 2.2 · Atama İfadeleri** **Bölüm 2.2 · İlkel Veri Tipleri**

KONTROL SORUSU

▶ **int a = 3; int b = a; a = 10; b kaçtır?**

ÖRNEK (JAVA)

```
int age = 20;
double gpa = 3.25;
boolean passed = gpa >= 2.0;
String name = "Ayşe";
age = age + 1;
System.out.println(name + " " + age + " " + passed);   // Ayşe 21 true
```

%

İfadeler ve işleçler

/ · % · işleç önceliği · dizgi birleştirme

Aritmetik işlemler öncelik kurallarına uyar: * / % işlemleri + − işlemlerinden önce yapılır, eşit öncelikli işleçler soldan sağa uygulanır. Kalan işleci % ile tam sayı bölmesi / birlikte bir sayıyı basamaklarına ayırır. + işleminin işlenenlerinden biri String olduğunda + toplama yapmaz, metinleri birleştirir.

NEDEN ÖNEMLİ

Basamak toplamı, tek/çift sınaması, bölünebilirlik ve biçimli çıktı bu işleçlere dayanır.

KULLANILDIĞI YERLER **Bölüm 2.2 · Aritmetik İşleçler** **Bölüm 2.2 · Öncelik Sırası**

Bölüm 3.4 · Birleştirme Sırası: Kod İzleme

KONTROL SORUSU

▶ **n = 4729 için n % 10 ve n / 10 kaçtır?**

ÖRNEK (JAVA)

```
System.out.println(17 / 5);        // 3
System.out.println(17 % 5);        // 2
System.out.println(2 + 3 * 4);     // 14
System.out.println("1" + 2 + 3);   // 123
System.out.println(1 + 2 + "3");   // 33
```

&& Mantıksal ifadeler

&& · || · ! · kısa devre · De Morgan

Koşullar && (ve), || (veya) ve ! (değil) ile birleştirilir. Java bunları soldan sağa değerlendirir ve sonuç belli olur olmaz durur (kısa devre değerlendirme, short-circuit); bu yüzden sonraki bir parça önceki bir denetime güvenebilir. De Morgan yasaları olumsuzlamayı içeri dağıtır: !(a && b) ile !a || !b aynıdır.

NEDEN ÖNEMLİ

Aralık denetimleri, girdi doğrulama ve döngü koşulları boolean ifadelerdir; yanlış yere yazılmış bir && ya da || en sık rastlanan mantık hatalarındandır.

KULLANILDIĞI YERLER **Bölüm 2.2 · Mantıksal İşleçler** **Bölüm 2.2 · Kısa Devre**

Bölüm 3.1 · İç İçe Kararlar (Nesting)

KONTROL SORUSU

▶ **!(x >= 1 && x <= 10) ifadesi ! kullanılmadan nasıl yazılır?**

ÖRNEK (JAVA)

```
int x = 7;
boolean inRange = x >= 1 && x <= 10;      // true
boolean outside = !(x >= 1 && x <= 10);   // false: x < 1 || x > 10 ile aynı
int d = 0;
if (d != 0 && 10 / d > 1)                 // sağ taraf atlanır: sıfıra bölme
olmaz
    System.out.println("büyük");
System.out.println(inRange + " " + outside);   // true false
```

? Kararlar: if, else if, switch

dallanma · iç içe kararlar

if, koşulu doğruysa bir bloğu çalıştırır; else bunun seçeneğini verir. else-if zinciri koşulları sırayla sınar ve yalnızca koşulu sağlanan ilk dalı çalıştırır. switch ise tek bir ifadenin değerine göre bir case seçer.

NEDEN ÖNEMLİ

Not hesaplama, menüler, girdi denetimleri ve dersteki hemen her algoritma karar verir.

KULLANILDIĞI YERLER **Bölüm 3.1 · if-else Yapısı** **Bölüm 3.1 · else-if Zinciri**

Bölüm 3.1 · switch-case Yapısı

KONTROL SORUSU

▶ **int x = 5; if (x > 3) yaz "A"; else if (x > 1) yaz "B"; Ne yazdırılır?**

ÖRNEK (JAVA)

```
int score = 72;
if (score >= 85) System.out.println("A");
else if (score >= 70) System.out.println("B");   // B
else System.out.println("C");
```

↻ Döngüler: for, while, do-while

sayaç · koşul · iç içe döngüler

Döngü, koşulu doğru olduğu sürece bir bloğu tekrarlar. for tekrar sayısının bilindiği durumlara, while her turdan önce sınanan bir koşula, do-while ise gövdesinin en az bir kez çalışması gereken durumlara uygundur. Her döngüde, koşulun sonunda yanlış olmasını sağlayacak biçimde değişen bir değişken bulunmalıdır.

NEDEN ÖNEMLİ

Toplamlar, aramalar, desenler, girdi döngüleri ve dizi işlemleri döngüyle yazılır; sonsuz döngüler ve bir eksik ya da bir fazla dönen döngüler sık yapılan hatalardır.

KULLANILDIĞI YERLER **Bölüm 3.1 · for Döngüsü** **Bölüm 3.1 · while Döngüsü**

Bölüm 3.1 · Döngü Türlerinin Karşılaştırılması **Bölüm 3.1 · İç İçe Döngüler**

KONTROL SORUSU

▶ **for (int i = 10; i > 0; i -= 3) döngüsü kaç kez döner?**

ÖRNEK (JAVA)

```
for (int i = 0; i < 3; i++) System.out.print(i + " ");   // 0 1 2
System.out.println();
```

```
int n = 1234, digits = 0;
while (n > 0) {
    n /= 10;
    digits++;
}
System.out.println(digits);                                  // 4
```

f() Fonksiyonlar: parametreler ve dönüş değerleri

metot · parametre · return · void

Fonksiyon (Java'da metot) bir hesaplamayı bir ad altında toplar: değerleri parametreler aracılığıyla alır ve return ile tek bir sonuç döndürebilir; void bir metot değer döndürmez. Java argümanları değer ile aktarır; bu yüzden bir metot, çağıranın int değişkenini değiştiremez.

NEDEN ÖNEMLİ

Programlar en baştan metotlara bölünür; sınıflar da metotların ve verinin bir araya geldiği yapılardır.

KULLANILDIĞI YERLER **Bölüm 3.2 · Fonksiyon Tanımı** **Bölüm 3.2 · return ve void**

Bölüm 3.2 · Değer ile Aktarım

KONTROL SORUSU

▶ **static void inc(int x) { x++; } tanımından sonra int a = 5; inc(a); çalışırsa a kaç olur?**

ÖRNEK (JAVA)

```
static int square(int x) { return x * x; }
static void greet(String name) { System.out.println("Merhaba " + name); }

// main içinde:
int r = square(5) + square(2);   // 29
greet("Ali");                    // Merhaba Ali
```

[ ] Diziler ve dizgiler: ilk tanışma

0'dan başlayan indis · length · charAt

Dizi, tek bir tipten sabit sayıda değer tutar; değerler 0'dan length − 1'e kadar indislerle numaralanır. String ise konumları yine 0'dan numaralanan bir karakter dizisidir. Bu sınırların dışı okunmaya çalışılırsa program bir istisnayla durur.

NEDEN ÖNEMLİ

Not listeleri, matrisler, metin işleme ve koleksiyon sınıflarının hepsi indisle erişime dayanır.

KULLANILDIĞI YERLER **Bölüm 3.3 · Dizi Tanımlama** **Bölüm 3.3 · Erişim ve length**

Bölüm 3.4 · İndisleme **Bölüm 3.3 · İndis Sınır Dışı Hatası**

KONTROL SORUSU

▶ **int[] a = {4, 8, 15}; a[3] okunursa ne olur?**

ÖRNEK (JAVA)

```
int[] a = {4, 8, 15};
System.out.println(a.length + " " + a[0] + " " + a[a.length - 1]);   // 3 4 15
String s = "Java";
System.out.println(s.length() + " " + s.charAt(1));                  // 4 a
```

Araçlar ve kurulum

Java kurulumu, programların derlenip çalıştırılması ve araçların bildirdiklerinin okunması.

>_

Komut satırının temelleri

cd · dir / ls · mkdir · çalışma dizini

Komut satırında (Windows'ta Komut İstemi ya da PowerShell, macOS ve Linux'ta Terminal) programlar adları yazılarak çalıştırılır. İstem (prompt) o anki çalışma dizinini gösterir; cd bu dizini değiştirir, dir (macOS/Linux'ta ls) dosyaları listeler, mkdir klasör oluşturur. Sekme (Tab) tuşu dosya adlarını tamamlar, yukarı ok tuşu önceki komutları geri getirir.

NEDEN ÖNEMLİ

javac ve java komut satırı programlarıdır; onları elle çalıştırmak, bir IDE'nin Çalıştır düğmesinin arkasında ne yaptığını gösterir.

KULLANILDIĞI YERLER **Bölüm 2.2 · Komut Satırında Derleme ve Çalıştırma**

Bölüm 2.2 · Programlama Araçları

KONTROL SORUSU

▶ **İstem C:\Users\ali\ders> biçiminde. cd .. komutundan sonra çalışma dizini nedir?**

ÖRNEK (WİNDOWS KOMUT SATIRI)

```
C:\Users\ali> mkdir ders          klasör oluşturur
C:\Users\ali> cd ders             klasöre girer
C:\Users\ali\ders> dir            dosyaları listeler (macOS/Linux'ta ls)
C:\Users\ali\ders> cd ..          bir üst düzeye çıkar
C:\Users\ali>
```

JDK

JDK kurulumu

JDK · LTS sürümü · PATH · java -version

Java Geliştirme Kiti (Java Development Kit, JDK) javac derleyicisini, JVM ile birlikte java başlatıcısını ve standart kütüphaneyi içerir. JDK 17 ya da 21 gibi uzun süreli destek (LTS) sürümü (örneğin Adoptium ya da Oracle dağıtımı) iyi bir seçimdir. Komutların her dizinde çalışması için JDK'nın bin klasörü PATH'e eklenmelidir; bunu kurulum programı ya da kullanıcı yapar. java -version ve javac -version komutları kurulumu doğrular.

NEDEN ÖNEMLİ

Çalışan bir JDK olmadan ne komut satırı örnekleri ne de bir IDE Java kodunu derleyebilir.

KULLANILDIĞI YERLER **Bölüm 2.2 · Java Platformu** **Bölüm 2.2 · Programlama Araçları**

KONTROL SORUSU

▶ **Komut satırı 'javac' is not recognized as an internal or external command yanıtını veriyor. Olası neden nedir?**

ÖRNEK (KURULUMU DENETLEME)

```
> java -version
openjdk version "21.0.3" 2024-04-16 LTS
OpenJDK Runtime Environment Temurin-21.0.3+9 (build 21.0.3+9-LTS)
OpenJDK 64-Bit Server VM Temurin-21.0.3+9 (build 21.0.3+9-LTS, mixed mode,
sharing)
> javac -version
javac 21.0.3
```

▶

javac ve java: derleme ve çalıştırma

Merhaba.java · Merhaba.class · main

javac Merhaba.java komutu kaynak dosyayı derler ve hata yoksa Merhaba.class dosyasını yazar. Ardından java Merhaba komutu JVM'i başlatır ve bu sınıfın main metodunu çalıştırır (ad .class uzantısı olmadan verilir). public bir sınıf, büyük-küçük harf dahil tam olarak aynı adı taşıyan bir dosyaya kaydedilmelidir.

NEDEN ÖNEMLİ

Dersteki her program bu biçimdedir: main metodu olan bir sınıf önce derlenir, sonra çalıştırılır.

KULLANILDIĞI YERLER **Bölüm 2.2 · Örnek .java Dosyası** **Bölüm 2.2 · Derleme**

Bölüm 2.2 · Komut Satırında Derleme ve Çalıştırma **Bölüm 2.2 · Bytecode**

KONTROL SORUSU

▶ **Test.java adlı bir dosyada public class Merhaba tanımı var. javac Test.java ne bildirir?**

ÖRNEK (MERHABA.JAVA)

```
public class Merhaba {
    public static void main(String[] args) {
        System.out.println("Merhaba, Java!");
    }
}

> javac Merhaba.java      (Merhaba.class oluşur)
> java Merhaba
Merhaba, Java!
```

IDE

IDE ile çalışma

IntelliJ IDEA · Eclipse · VS Code · hata ayıklayıcı

Tümleşik geliştirme ortamı (integrated development environment, IDE) bir düzenleyiciyi, derleyiciyi ve hata ayıklayıcıyı bir araya getirir: yazarken hataların altını çizer, adları tamamlar, Çalıştır (Run) düğmesiyle de programı derleyip çalıştırır. Yaygın seçenekler IntelliJ IDEA, Eclipse ve Java eklentileriyle Visual Studio Code'dur. Hata ayıklayıcı bir kesme noktasında (breakpoint) durur ve değişkenleri göstererek programı satır satır yürütür.

NEDEN ÖNEMLİ

Birkaç sınıftan oluşan büyük programlar IDE'de yazılır; uzun kodlarda elle izlemenin yerini hata ayıklayıcı alır.

KULLANILDIĞI YERLER **Bölüm 2.2 · Programlama Araçları** **Bölüm 7 · Hata Ayıklayıcı**

Bölüm 7 · Step Into / Over / Out

KONTROL SORUSU

▶ **Hata ayıklayıcı int r = square(5); satırında durmuş. Step Over ile Step Into arasındaki fark nedir?**

ÖRNEK (KULLANIŞLI IDE KISAYOLLARI)

```
IntelliJ IDEA   psvm + Tab   ->  public static void main(String[] args)
IntelliJ IDEA   sout + Tab   ->  System.out.println();
Eclipse         main + Ctrl+Space, sysout + Ctrl+Space
tüm IDE'lerde   satırın yanındaki kenar boşluğuna tıklamak kesme noktası koyar
```

! Hata iletilerini okuma

derleme zamanı hatası · istisna · yığın izi

javac, derleme zamanı hatasını dosya adı, satır numarası ve sorunun altına konan şapka (^) işaretiyle bildirir; bu durumda .class dosyası üretilmez. Çalışma zamanı hatası ise bir istisnadır (exception): program derlenir ama çalışırken durur ve istisnanın tipini, bir iletiyi ve metot çağrılarının yığın izini (stack trace) yazdırır. İşe genellikle ilk iletiden ve onun satır numarasından başlanır.

NEDEN ÖNEMLİ

Hataları hızla düzeltmek, tahmin yürütmek yerine bu iletileri okumaya bağlıdır.

KULLANILDIĞI YERLER **Bölüm 2.1 · Hata Mesajlarını Okumak**

Bölüm 2.1 · Neler Yanlış Gidebilir? **Bölüm 6 · Yığın İzini Okumak**

KONTROL SORUSU

▶ **int x = "5"; bir derleme zamanı hatası mıdır, yoksa çalışma zamanı hatası mı?**

ÖRNEK (GERÇEK İLETİLER)

```
Merhaba.java:3: error: ';' expected
        System.out.println("Merhaba")
                                    ^
1 error

Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Bolme.main(Bolme.java:4)
```

doc API belgelerini okuma

Javadoc · metot imzası · dönüş tipi

Java standart kütüphanesi, pakete ve sınıfa göre düzenlenmiş Javadoc sayfalarında (docs.oracle.com) belgelenir. Bir metodun başlığı neye ihtiyaç duyduğunu ve ne döndürdüğünü söyler: public static int parseInt(String s) başlığında parametre bir String, sonuç bir int'tir; static ise metodun Integer.parseInt("42") biçiminde sınıf üzerinden çağrıldığını gösterir.

NEDEN ÖNEMLİ

String, Math, Scanner, ArrayList ve HashMap gibi sınıflar belgelenmiş metotları aracılığıyla kullanılır.

KULLANILDIĞI YERLER **Bölüm 3.2 · Math Sınıfı** **Bölüm 3.4 · Temel Metotlar**

Bölüm 3.2 · En Sık Kullanılan Metotlar

KONTROL SORUSU

▶ **Math.max metodunun int ve double için ayrı sürümleri vardır. Math.max(3, 7.5) ne döndürür?**

ÖRNEK (İMZALAR VE ÇAĞRILAR)

```
public static int parseInt(String s)      // Integer sınıfında
public int length()                       // String sınıfında
public static double sqrt(double a)       // Math sınıfında

int n = Integer.parseInt("42") + 1;       // 43
int k = "Java".length();                  // 4
double r = Math.sqrt(16);                 // 4.0
```

Nesnelere hazırlık

Nesne yönelimli programlamanın dayandığı iki düşünme alışkanlığı.

is

Sınıflandırma: is-a ve has-a

kategori · parça · ilişki

Gündelik nesneler kategorilere ayrılır (her köpek bir hayvandır) ve parçalardan oluşur (her arabanın bir motoru vardır). Nesne yönelimli programlama ilk ilişkiyi kalıtımla (is-a), ikincisini başka nesnelere başvuran alanlarla (has-a) modeller. Bir cümledeki bu iki ilişkiyi birbirinden ayırmak sınıf tasarımının ilk adımıdır.

NEDEN ÖNEMLİ

extends ile bir alan arasında seçim yapmak ve sınıf hiyerarşileri kurmak bu ayrımdan başlar.

KULLANILDIĞI YERLER **Bölüm 4.3 · is-a ve has-a İlişkileri**

Bölüm 4.1 · Nesneler: Alan ve Davranış **Bölüm 4.1 · Sınıf: Nesnenin Taslağı**

KONTROL SORUSU

▶ **Bir öğrencinin öğrenci numarası vardır; bir öğrenci bir kişidir. Hangisi is-a, hangisi has-a ilişkisidir?**

ÖRNEK (JAVA'YA ÖN BAKIŞ)

```
class Animal { }
class Dog extends Animal { }      // her Dog bir Animal'dır

class Engine { }
class Car {
    Engine engine = new Engine();  // her Car'ın bir Engine'i vardır
}
```

▣ Soyutlama ve ayrıştırma

parçalara ayırma · ayrıntıyı gizleme · yeniden kullanım

Ayrıştırma (decomposition) büyük bir problemi ayrı ayrı çözülebilecek küçük parçalara böler; soyutlama (abstraction) ise bir parçanın yalnızca önemli yönlerini tutar ve içeride nasıl çalıştığını gizler. Math.sqrt metodunu algoritmasını bilmeden kullanmak soyutlamadır; bir programı birkaç küçük metot olarak yazmak ise ayrıştırmadır.

NEDEN ÖNEMLİ

Metotlar, sınıflar, arayüzler ve sarmalama bu iki fikrin araçlarıdır.

KULLANILDIĞI YERLER **Bölüm 3.2 · Ayrıştırma** **Bölüm 3.2 · Soyutlama**

Bölüm 4.5 · Soyutlama **Bölüm 1.2 · Hesaplamanın Yapı Taşları**

KONTROL SORUSU

▶ **Bir program notları okuyor, ortalamalarını hesaplıyor ve bir rapor yazdırıyor. Bu program hangi parçalara ayrılabilir?**

ÖRNEK (JAVA'YA ÖN BAKIŞ)

```
    static int[] readGrades() { ... }
    static double average(int[] g) { ... }
    static void printReport(double avg) { ... }

    // main yalnızca parçaları birleştirir:
    printReport(average(readGrades()));
```

Java örneklerindeki çıktılar, programlar javac --release 8 ile derlenip çalıştırılarak denetlendi; “örneğin” ile işaretlenen çıktılar bilgisayara göre değişir.

EK C

Java ve Nesne Yönelimli Programlamada Önemli İsimler

JAVA · İSİMLER

Dersteki fikirlerin arkasındaki 66 kişi, 47 kartta: Öklid algoritmasından, Babbage'ın Analitik Makinesi'nden, Turing'in evrensel makinesinden ve ENIAC'tan Simula ile Smalltalk'a, James Gosling'in Java'sına ve JVM'e, bilgi saklamadan ve Sözleşmeyle Tasarım'dan Koleksiyon Çerçevesi'ne, java.util.concurrent paketine, ilişkisel modele, JUnit'e ve tasarım desenlerine kadar.

Her kartta kişinin yaşadığı yıllar, ülkesi ve kurumu, en bilinen katkısı ve kısa bir özet yer alır; bağlantılar bu katkının geçtiği slaytlara gider. Kartlar bölüme göre süzülebilir, aranabilir, katkı yılına ya da soyadına göre sıralanabilir.

ÖK

Öklid (Eukleides)

MÖ 300 dolayları · Yunan · İskenderiye

Öklid algoritması · Elemanlar, MÖ 300 dolayları

İskenderiye'de çalışan Yunan matematikçidir; yaşamı hakkında çok az şey bilinir. Geometriyi ve sayılar kuramını tanımlar, aksiyomlar ve kanıtlarla düzenleyen 13 kitaplık *Elemanlar*'ın yazarıdır. Kitabın VII. cildi iki sayının en büyük ortak bölenini (EBOB) bulan yöntemi verir. Öklid'in anlatımında büyük sayıdan küçüğü art arda çıkarılır; bölme ve kalanla yazılan bugünkü biçim aynı yöntemin kısaltmasıdır: kalan 0 olunca son bölen EBOB'dur. Bugün de kullanılan, bilinen en eski algoritmalardan biridir. EBOB(a, b) = EBOB(b, a mod b) bağıntısı tek satırlık bir özyinelemeli metoda dönüşür; kesirleri sadeleştiren bir sınıf da aynı yöntemi gizli bir yardımcı metotta kullanır.

DERSTE **Bölüm 1.2 · Öklid Algoritması** **Bölüm 1.2 · Öklid: İzleme**

Bölüm 3.5 · En Büyük Ortak Bölen **Bölüm 4.1 · Tasarım Örneği: RasyonelSayi**

Öklid algoritması EBOB buyurusal bilgi

HZ

Muhammed bin Musa el-Hârizmî

780 dolayları – 850 dolayları · Bağdat · Beytülhikme (Bilgelik Evi)

Hint rakamlarıyla hesap · 825 dolayları

Bağdat'taki Beytülhikme'de çalışan matematikçi ve gökbilimcidir. 825 dolaylarında yazdığı Hint rakamlarıyla hesap kitabı, onluk konumsal sayı sistemini ve bu rakamlarla yapılan dört işlemin yöntemlerini adım adım anlatır; kitap bu sistemin Orta Doğu'da ve Avrupa'da yayılmasını sağladı. Eserin 12. yüzyıldaki Latince çevirisi *Algoritmi de numero Indorum* adıyla tanındı. "Algoritma" sözcüğü yazarın bu Latinceleşmiş adından türedi; "cebir" sözcüğü de onun *el-Cebr* başlıklı kitabından gelir.

DERSTE **Bölüm 1.3 · Algoritma Nedir?** **Bölüm 1.2 · Algoritma**

algoritma onluk sayı sistemi cebir

BA

Charles Babbage ve Ada Lovelace

1791–1871 · 1815–1852 · İngiliz · Londra

Analitik Makine · Not G, 1843

Babbage 1834'ten başlayarak delikli kartlarla programlanacak genel amaçlı bir makine olan Analitik Makine'yi (Analytical Engine) tasarladı; makine hiçbir zaman tamamlanamadı. Ada Lovelace, 1843'te makine üzerine yazdığı notlarda Bernoulli sayılarını hesaplamak için adım adım bir yöntem verdi (Not G); bu yöntem genellikle yayımlanmış ilk bilgisayar programı sayılır. Lovelace böyle bir makinenin sayılardan başka simgeler üzerinde de çalışabileceğini fark etti. Makinenin birbirinden ayrı "ambar" (store) ve "değirmen" (mill) birimleri ile tekrarlanan işlem dizileri, bugünkü bellek, işlemci ve döngü kavramlarının öncüsüdür.

DERSTE **Bölüm 1.2 · Babbage ve Analitik Makine** **Bölüm 1.2 · Ada Lovelace: İlk Program**

Analitik Makine ilk program döngü

AC

Alonzo Church

1903–1995 · Amerikalı · Princeton University

Lambda hesabı · 1936

1930'ların başında, yalnızca adsız fonksiyonlar tanımlamaya ve bunları uygulamaya dayanan bir hesaplama modeli olan lambda hesabını (lambda calculus) geliştirdi. 1936'da bu modelle Turing'den bağımsız olarak bir karar verilemezlik sonucunu kanıtladı; Church–Turing tezi bu yüzden iki adı birlikte taşır. Fonksiyonel programlama dilleri onun hesabından türemiştir. Java 8 ile gelen `x -> x * x` gibi lambda ifadeleri de adını bu hesaptan alır.

DERSTE **Bölüm 4.5 · Anonim (Lambda) Fonksiyonlar**

Bölüm 2.1 · Aynı Problem: Buyruksal ve Fonksiyonel

lambda hesabı adsız fonksiyon fonksiyonel programlama

AT Alan Turing

1912–1954 · İngiliz · Cambridge, Manchester

Turing makinesi · 1936

"On Computable Numbers" (1936) makalesinde Turing makinesini tanımladı. Tanımı kendisine verilen her makineyi tek bir evrensel makinenin taklit edebileceğini gösterdi ve bir programın durup durmayacağına genel olarak hiçbir algoritmanın karar veremeyeceğini kanıtladı (durma problemi, halting problem). Evrensel makine, programın veri gibi ele alınması fikridir; depolanan programlı bilgisayarlar bu fikri gerçeğe dönüştürdü. Bytecode olarak verilen başka programları çalıştıran bir program olan JVM de aynı ilkeye dayanır. Hiçbir Java aracının her sonsuz döngüyü yakalayamamasının nedeni de durma problemidir.

DERSTE **Bölüm 1.2 · Evrensel Turing Makinesi** **Bölüm 1.2 · Hesaplanabilirlik**

Bölüm 1.2 · Sınırlar Java'da

hesaplanabilirlik durma problemi evrensel makine

FY Ronald A. Fisher ve Frank Yates

1890–1962 · 1902–1994 · İngiliz · Rothamsted Deney İstasyonu

Fisher–Yates karıştırması · 1938

Deney tasarımı üzerine Rothamsted'de birlikte çalışan iki istatistikçidir. 1938'de yayımladıkları *Statistical Tables for Biological, Agricultural and Medical Research* kitabında bir sırayı rastgele karıştırmanın kâğıt kalemle uygulanan bir yolunu verdiler: henüz seçilmemiş öğelerden biri rastgele seçilir, listeden çizilir ve yeni sıranın sonuna yazılır. Yöntem n öğenin n! dizilişinin her birini eşit olasılıkla üretir. Bilgisayar için yerinde takasla çalışan biçimini Richard Durstenfeld 1964'te yayımladı; Donald Knuth'un *The Art of Computer Programming* kitabı onu yaygınlaştırdı, bu yüzden Knuth karıştırması diye de anılır. `Collections.shuffle` bu biçimi kullanır: listenin sonundan başa doğru ilerler ve her konumdaki öğeyi, baştan o konuma kadarki öğelerden rastgele seçilen biriyle takas eder. İş O(n) sürer.

DERSTE **Bölüm 5 · Karıştırma**

karıştırma Collections.shuffle rastgele permütasyon

AB

John V. Atanasoff ve Clifford Berry

1903–1995 · 1918–1963 · Amerikalı · Iowa State College

Atanasoff–Berry Bilgisayarı (ABC) · 1942

Fizikçi Atanasoff 1937'de doğrusal denklem sistemlerini çözecek elektronik bir hesap makinesi tasarladı; yüksek lisans öğrencisi Berry ile makineyi 1939–1942 arasında Iowa State College'da kurdular. ABC sayıları ikilik tabanda tuttu, işlemleri 300'den fazla vakum tüpüyle elektronik olarak yaptı ve ara sonuçları dönen tamburlar üzerindeki kondansatörlerde, düzenli olarak tazelenen bir bellekte sakladı. Programlanamıyordu ve programı belleğinde tutmuyordu: tek bir iş için kurulmuş, sabit programlı bir makineydi. 1973'te bir ABD mahkemesi ENIAC patentini geçersiz sayarken elektronik sayısal bilgisayar fikrinin Atanasoff'un çalışmasından alındığına hükmetti.

DERSTE **Bölüm 1.2 · ABC ve Colossus** **Bölüm 1.2 · Sabit Programlı Bilgisayar**

ABC ikilik aritmetik vakum tüpü

TF

Tommy Flowers

1905–1998 · İngiliz · Posta İdaresi Araştırma İstasyonu, Dollis Hill

Colossus · 1944

İngiliz Posta İdaresi'nde telefon santralleri üzerinde çalışan bir mühendisti. Alman ordusunun Lorenz şifre makinesiyle gönderdiği iletileri çözmek için Colossus'u tasarladı ve ekibiyle Dollis Hill'de kurdu. 1500 vakum tüpü taşıyan Mark 1, Ocak 1944'te Bletchley Park'a teslim edildi ve şubatta çalışmaya başladı; 2400 tüplü Mark 2 onu izledi. Colossus programlanabilir ilk elektronik sayısal bilgisayarlardan biridir, ama programı belleğe yüklenmez, anahtarlar ve fişli kablolarla kurulurdu. Çalışma uzun yıllar devlet sırrı olarak kaldı; Flowers bu yüzden yaptığı işi anlatamadı.

DERSTE **Bölüm 1.2 · ABC ve Colossus**

Colossus Bletchley Park vakum tüpü

ME

John Mauchly ve J. Presper Eckert

1907–1980 · 1919–1995 · Amerikalı · Pennsylvania Üniversitesi, Moore Okulu

ENIAC · 1946

Fizikçi Mauchly ile mühendis Eckert, Pennsylvania Üniversitesi'nin Moore Elektrik Mühendisliği Okulu'nda ENIAC'ı tasarlayıp kurdular. Yaklaşık 18.000 vakum tüpü taşıyan makine Aralık 1945'te ilk işini yaptı ve Şubat 1946'da kamuoyuna tanıtıldı. ENIAC genel amaçlı ilk büyük elektronik bilgisayardı, ama programı kablolar ve anahtarlarla kuruluyordu; yeni bir iş için makinenin yeniden bağlanması gerekiyordu. Bu güçlük, programı veri gibi belleğe koyan EDVAC tasarımına yol açtı; aynı ekibin üzerinde çalıştığı bu tasarımı von Neumann 1945 tarihli raporunda tanımladı. İkisi 1946'da kendi bilgisayar şirketlerini kurdu.

DERSTE **Bölüm 1.2 · ENIAC ve EDVAC** **Bölüm 1.2 · Sabit Programlı Bilgisayar**

ENIAC EDVAC elektronik bilgisayar

GN Herman Goldstine ve John von Neumann

1913–2004 · 1903–1957 · Amerikalı · Macar asıllı Amerikalı · Princeton IAS

Akış diyagramları · depolanan program tasarımı

“Planning and Coding of Problems for an Electronic Computing Instrument” (1947–48) başlıklı rapor dizisinde programları kod yazmadan önce akış diyagramlarıyla (flow diagram) planladılar; bu diyagramlar bugünkü akış diyagramlarının öncüsüdür. Von Neumann'ın 1945 tarihli “First Draft of a Report on the EDVAC” raporu, komutları ve veriyi aynı bellekte tutan bir bilgisayarı tanımladı. JVM'i çalıştıran bilgisayar da dahil olmak üzere neredeyse bütün bilgisayarlar bu tasarımı izler.

DERSTE **Bölüm 1.4 · Algoritma ve Akış Diyagramı** **Bölüm 1.1 · von Neumann Mimarisi** **Bölüm 1.1 · Saklanan Program Kavramı**

akış diyagramı depolanan program von Neumann mimarisi

GH Grace Hopper

1906–1992 · Amerikalı · ABD Donanması, Remington Rand

A-0 derleyicisi · 1952

1952'de simgesel kodu makine koduna çeviren ilk programlardan biri olan A-0'ı yazdı ve programların İngilizceye benzeyen sözcüklerle yazılması gerektiğini savundu; geliştirdiği FLOW-MATIC, COBOL'u (1959) etkiledi. Çeviriyi bir programa yaptırma fikri `javac` her çalıştığında sürer. 1947'de Harvard Mark II'nin kayıt defterine bir “bug” (hata) notuyla birlikte yapıştırılan güve onun ekibiyle anılır ve sözcüğün yaygınlaşmasına katkıda bulundu; mühendisler sözcüğü bundan önce de kullanıyordu.

DERSTE **Bölüm 2.2 · Derleme** **Bölüm 2.1 · Derlenen ve Yorumlanan Diller**

Bölüm 7 · Hata, Kusur, Arıza

derleyici yüksek düzeyli dil hata (bug)

BN

John Backus ve Peter Naur

1924–2007 · 1928–2016 · Amerikalı · Danimarkalı · IBM, Kopenhag

FORTRAN · Backus–Naur Formu

Backus, 1957'de FORTRAN'ı ortaya koyan IBM ekibini yönetti; FORTRAN, kodu eniyileyen (optimizing) derleyicisiyle yaygın olarak kullanılan ilk yüksek düzeyli dildi. 1959'da söz dizimi için biçimsel bir gösterim önerdi; Naur bu gösterimi ALGOL 60 raporunda (1960) benimseyip geliştirdi ve gösterim Backus–Naur Formu adıyla tanındı. Java Language Specification, Java'nın dilbilgisini (grammar) benzer bir gösterimle tanımlar; `javac`'ın verdiği `';' expected` gibi bir ileti de bu dilbilgisinin bir kuralının çiğnendiğini bildirir. İkisi de Turing Ödülü'nü aldı (1977, 2005).

DERSTE **Bölüm 1.2 · İlk Programlar ve Fortran** **Bölüm 2.2 · Temel Söz Dizimi Kuralları**

Bölüm 2.1 · Temel Yapılar ve Söz Dizimi

FORTRAN BNF dilbilgisi söz dizimi

JM

John McCarthy

1927–2011 · Amerikalı · MIT, Stanford University

Lisp · çöp toplama

1958'de MIT'de Lisp'i yarattı; dil üzerine yazdığı 1960 tarihli makalede çöp toplamayı (garbage collection) tanımladı: programın artık ulaşamadığı bellek kendiliğinden geri alınır. Java da aynı mekanizmaya dayanır. Bir nesneyi artık hiçbir referans göstermediğinde JVM'in çöp toplayıcısı onu serbest bırakır; Java'da bir nesneyi elle silen bir işleç yoktur. “Yapay zekâ” (artificial intelligence) terimini de o ortaya attı; 1971'de Turing Ödülü'nü aldı.

DERSTE **Bölüm 4.1 · Referans Atama: Adım Adım** **Bölüm 3.3 · Diziler Bellekte Nerede?**

çöp toplama Lisp referanslar

ED

Edsger W. Dijkstra

1930–2002 · Hollandalı · Eindhoven; Austin, ABD

Semaforlar · yapısal programlama

1965'te P ve V işlemleriyle semaforu (semaphore) tanıttı ve kilitlenmenin (deadlock) klasik örneği olan yemek yiyen filozoflar problemini ortaya koydu; `java.util.concurrent.Semaphore` aynı fikri `acquire()` ve `release()` metotlarıyla sunar. 1968 tarihli "Go To Statement Considered Harmful" mektubunda yapısal kontrol akışını savundu; Java `goto` sözcüğünü anahtar sözcük olarak ayırır ama ona bir anlam vermez. Testin hataların varlığını gösterebileceğini, yokluğunu ise hiçbir zaman gösteremeyeceğini de o vurguladı. 1972 Turing Ödülü.

DERSTE **Bölüm 10 · Semaphore** **Bölüm 10 · Kilitlenme (Deadlock)**

Bölüm 2.2 · Anahtar Sözcükler **Bölüm 7 · Neden Test?**

semafor kilitlenme goto test

TH C. A. R. (Tony) Hoare

1934–2026 · İngiliz · Oxford, Microsoft Research

Null referans · monitörler

1965'te ALGOL W'yu tasarlarken dile null referansı ekledi; 2009'daki bir konuşmasında bu kararı "milyar dolarlık hatası" olarak niteledi. Her `NullPointerException` bu karara dayanır. 1974 tarihli "Monitors: An Operating System Structuring Concept" makalesinde monitörü (monitor) tanımladı: yordamları birer birer çalışan ve çağıranları belirli koşulları bekleyebilen paylaşılan bir nesne. Java her nesneye `synchronized`, `wait()` ve `notify()` aracılığıyla bu türden bir kilit verir. Hızlı sıralamayı (quicksort) da o buldu; 1980'de Turing Ödülü'nü aldı.

DERSTE **Bölüm 4.1 · null ve NullPointerException** **Bölüm 6 · NullPointerException**

Bölüm 10 · synchronized ve Monitör **Bölüm 10 · wait() ve notify()**

null monitör synchronized

BJ Corrado Böhm ve Giuseppe Jacopini

1923–2017 (Böhm) · İtalyan · Roma

Yapısal program teoremi · 1966

1966 tarihli "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules" makalelerinde, akış diyagramıyla yazılmış her programın yalnızca üç yapıyla yeniden yazılabileceğini gösterdiler: sıra, seçim ve tekrar. Java'da bu yapıların karşılıkları art arda yazılan deyimler, `if` / `switch` ve `while` / `for` deyimleridir; bir algoritmayı ifade etmek için atlama deyimine gerek yoktur. Bu sonuç yapısal programlamanın kuramsal temeli oldu.

DERSTE **Bölüm 1.4 · Kontrol Mekanizmaları** **Bölüm 3.1 · Kontrol Yapıları**

Bölüm 1.2 · Algoritmik İşlem Türleri

sıra seçim tekrar yapısal programlama

DN

Ole-Johan Dahl ve Kristen Nygaard

1931–2002 · 1926–2002 · Norveçli · Norwegian Computing Center, Oslo

Simula 67 · sınıflar ve kalıtım

Benzetim (simulation) problemlerinden yola çıkarak Simula'yı tasarladılar. Simula 67 sınıfları, sınıflardan oluşturulan nesneleri, bir sınıftan kalıtım alan alt sınıfları ve alt sınıfın yeniden tanımlayabileceği sanal yordamları (virtual procedure) getirdi. Bunlar nesne yönelimli programlamanın temel fikirleridir: Java'daki `class`, `new`, `extends` ve geçersiz kılınabilen metotlar buradan gelir; C++ da işe Simula'nın sınıflarını C'ye ekleyerek başladı. 2001 Turing Ödülü'nü paylaştılar.

DERSTE **Bölüm 4.1 · Sınıf: Nesnenin Taslağı** **Bölüm 4.3 · Kalıtım Nedir?**

Bölüm 4.4 · Dinamik Bağlama

Simula sınıf kalıtım sanal metot

EC

Edgar F. Codd

1923–2003 · İngiliz · IBM San Jose

İlişkisel model · 1970

1970 tarihli “A Relational Model of Data for Large Shared Data Banks” makalesinde veriyi ilişkiler, yani satırlardan oluşan tablolar olarak saklamayı ve veriye saklama yapısındaki işaretçileri izleyerek değil değerler üzerinden ulaşmayı önerdi. JDBC ile oluşturulup sorgulanan her tablo, onun tanımıyla bir ilişkidir. Bir SQL sorgusu hangi satırların istendiğini söyler, bu satırların nasıl saklandığını ya da nasıl bulunacağını söylemez; bu fikir Codd'un veri bağımsızlığı (data independence) ilkesine dayanır. 1981 Turing Ödülü.

DERSTE **Bölüm 11 · İlişkisel Model** **Bölüm 11 · Dosya mı, Veri Tabanı mı?**

ilişkisel model tablo veri bağımsızlığı

NW

Niklaus Wirth

1934–2024 · İsviçreli · ETH Zürih

Pascal · P-kodu

Pascal'ı (1970) yapısal programlamayı öğretmek için tasarladı. 1970'lerin Pascal-P derleyicileri basit bir sanal yığın makinesi için P-kodu (P-code) üretiyordu; böylece Pascal'ı yeni bir bilgisayara taşımak için küçük bir yorumlayıcı yazmak yetiyordu. Java'nın bytecode'u ve JVM, derlenmiş tek bir programı birçok platformda çalıştırmak için aynı yolu izler. Algorithms + Data Structures = Programs (1976) adlı kitabı programlamaya bakışını özetler. 1984 Turing Ödülü.

DERSTE **Bölüm 2.2 · Bytecode** **Bölüm 2.2 · Platform Bağımsızlık** **Bölüm 2.2 · Derleme**

Pascal P-kodu sanal makine taşınabilirlik

AK Alan Kay

d. 1940 · Amerikalı · Xerox PARC

Smalltalk · nesne yönelimli programlama

1970'lerde Xerox PARC'ta Smalltalk'u geliştiren grubu yönetti. Smalltalk'ta her şey bir nesnedir ve bütün hesaplama nesnelere ileti gönderilerek yapılır. “Nesne yönelimli programlama” (object-oriented programming) teriminin ona ait olduğu kabul edilir. Java'da `araba.hizlan(10)` gibi bir metot çağrısı da aynı düşünceyi yansıtır: nesneye bir ileti gönderilir, nesne de nasıl yanıt vereceğine kendi verisi ve davranışıyla karar verir. 2003 Turing Ödülü.

DERSTE **Bölüm 4.1 · Nesne Yönelimli Programlama**

Bölüm 4.1 · Nesneler: Alan ve Davranış **Bölüm 2.1 · Mantıksal ve Nesne Yönelimli**

Smalltalk ileti gönderme nesne

DP David Parnas

d. 1941 · Amerika doğumlu Kanadalı · Carnegie Mellon, McMaster University

Bilgi saklama · 1972

1972 tarihli “On the Criteria To Be Used in Decomposing Systems into Modules” makalesinde her modülün, değişmesi olası bir tasarım kararını gizlemesi ve dışarıya yalnızca bir arayüz açması gerektiğini savundu. Java'da `public` metotların arkasındaki `private` alanlar bu bilgi saklama (information hiding) ilkesini uygular: bir sınıfın iç gösterimi, onu kullanan kod değişmeden başka bir gösterimle değiştirilebilir.

DERSTE **Bölüm 4.2 · Bilgi Saklama** **Bölüm 4.2 · Gerçeklemeyi Değiştirmek**

Bölüm 4.1 · Erişim Belirleyiciler

bilgi saklama modül sarmalama

RK

Dennis Ritchie ve Brian Kernighan

1941–2011 · d. 1942 · Amerikalı · Kanadalı · Bell Labs

C · The C Programming Language, 1978

Ritchie 1970'lerin başında Bell Labs'te C'yi yarattı; Kernighan ile birlikte yazdıkları kitap (1978), ilk program olarak “hello, world” yazdırmayı ünlü yaptı. Java'nın görünen söz dizimi C'den gelir: süslü parantezler ve noktalı virgüller, `for` döngüleri, `break` unutulunca aşağı düşen `switch`, `++`, `&&` ve `||` işleçleri ve `printf` tarzı biçim dizgileri. Ritchie, Unix için verilen 1983 Turing Ödülü'nü Ken Thompson ile paylaştı.

DERSTE **Bölüm 2.2 · Örnek .java Dosyası** **Bölüm 3.1 · break Unutulursa: Aşağı Düşme** **Bölüm 3.4 · printf Metodu**

C hello, world printf söz dizimi

PB

Per Brinch Hansen

1938–2007 · Danimarkalı · Kopenhag; Syracuse University, ABD

Monitörler · Concurrent Pascal

Monitör fikrini 1972–73'te önerdi, Operating System Principles (1973) adlı kitabında ona bir sınıf gösterimi verdi ve fikri Concurrent Pascal diline (1975) yerleştirdi. “Java's Insecure Parallelism” (SIGPLAN Notices, 1999) yazısında `synchronized` anahtar sözcüğünün bu fikirlerin zayıf bir biçimi olduğunu savundu: senkronize edilmemiş metotların ya da private olmayan alanların birkaç iş parçacığınca aynı anda kullanılmasını hiçbir şey engellemez. Paylaşılan bir sayaçta güncellemelerin kaybolması tam olarak bu türden bir hatadır.

DERSTE **Bölüm 10 · synchronized ve Monitör** **Bölüm 10 · Yarış Durumu** **Bölüm 10 · sayac++ Neden Kaybolur?**

monitör yarış durumu karşılıklı dışlama

CB

Donald Chamberlin ve Raymond Boyce

günümüz · 1946–1974 · Amerikalı · IBM San Jose

SEQUEL / SQL · 1974

IBM'de Codd'un ilişkisel modeli için İngilizceye benzeyen bir sorgu dili olan SEQUEL'i (1974) tasarladılar; bir ticari marka çakışması yüzünden dilin adı SQL olarak değiştirildi ve SQL 1986'da ANSI standardı oldu. `CREATE TABLE`, `INSERT` ve `SELECT … WHERE` gibi komutlar Java'dan

`Statement` ya da `PreparedStatement` ile gönderilir. Boyce–Codd normal biçimine de adını veren Boyce, 1974'te 27 yaşında öldü.

DERSTE **Bölüm 11 · Yaygın SQL Komutları** **Bölüm 11 · SELECT Ayrıntıları**

Bölüm 11 · PreparedStatement

SQL sorgu dili SELECT

TR Trygve Reenskaug

1930–2024 · Norveçli · Xerox PARC; University of Oslo

Model–Görünüm–Denetleyici · 1979

1978–79'da Xerox PARC'ta konuk araştırmacıyken Smalltalk için Model–Görünüm–Denetleyici (Model–View–Controller, MVC) yapısını tanımladı: model veriyi tutar, görünüm onu gösterir, denetleyici de kullanıcının girdisini ele alır. Swing bu ayrımın değiştirilmiş bir biçimini kullanır; bir `JTextField` görünümdür, `getDocument()` ile elde edilen `Document` ise metni saklayan modeldir. MVC aynı zamanda standart mimari desenlerden biridir.

DERSTE **Bölüm 9 · Alanlar ve Yapıcı** **Bölüm 9 · AWT ve Swing**

Bölüm 4.5 · Tasarım Desenleri

MVC Swing GUI mimarisi

GY Jim Gray

1944–2007 · Amerikalı · IBM, Tandem, Microsoft Research

Veri tabanı işlemleri · 1981

Veri tabanı işlemlerinin (transaction) kilitleme ve günlük kaydı (logging) ile nasıl yürütüleceğini ortaya koydu; “The Transaction Concept: Virtues and Limitations” (1981) yazısında işlemleri bölünmez (atomic), tutarlı (consistent) ve kalıcı (durable) olarak tanımladı. Theo Härder ile Andreas Reuter 1983'te yalıtımı (isolation) ekleyip ACID kısaltmasını ortaya attı. JDBC'de `setAutoCommit(false)`, `commit()` ve `rollback()`, bir para aktarımının ya tamamen gerçekleşmesini ya da hiç gerçekleşmemesini sağlar. 1998 Turing Ödülü; 2007'de denizde kayboldu.

DERSTE **Bölüm 11 · İşlem ve ACID** **Bölüm 11 · Havale Örneği**

işlem (transaction) ACID commit/rollback

BS

Bjarne Stroustrup

d. 1950 · Danimarkalı · Bell Labs; Texas A&M, Columbia University

C++ · 1983

1979'da Bell Labs'te Simula'nın sınıflarını C'ye ekleyen “C with Classes” üzerinde çalışmaya başladı; dilin adı 1983'te C++ oldu. Java sınıflar, yapıcılar ve `public`, `private`, `protected` erişim belirleyicileri için C++'ın söz diziminin büyük kısmını aldı; sınıflarda çoklu kalıtımı, işleç aşırı yüklemeyi ve açık işaretçileri ise bilerek dışarıda bıraktı. Java'da bir sınıf bunların yerine tek bir sınıfı genişletir ve birden çok arayüzü uygulayabilir.

DERSTE **Bölüm 4.3 · Çoklu Kalıtım** **Bölüm 4.5 · Birden Fazla Arayüz**

Bölüm 2.1 · Popüler Programlama Dilleri

C++ çoklu kalıtım erişim belirleyiciler

BL

Barbara Liskov

d. 1939 · Amerikalı · MIT

Soyut veri tipleri · yerine koyma ilkesi

1974'te Stephen Zilles ile birlikte, gösterimleriyle değil işlemleriyle tanımlanan soyut veri tiplerini (abstract data type) tanımladı ve bunları CLU diline yerleştirdi. OOPSLA 1987'deki “Data Abstraction and Hierarchy” açılış konuşmasında bugün Liskov yerine koyma ilkesi (Liskov substitution principle) denen kuralı dile getirdi: bir alt tipin nesnesi, üst tipinin nesnesinin beklendiği her yerde kullanılabilmelidir. İlkeyi 1994'te Jeannette Wing ile biçimselleştirdi. Doğru kurulmuş bir is-a ilişkisi de her öğesinde aynı metodu çağıran çok biçimli bir dizi de bu kurala dayanır. 2008 Turing Ödülü.

DERSTE **Bölüm 4.3 · is-a ve has-a İlişkileri** **Bölüm 4.3 · Kalıtımla Tasarım İlkeleri**

Bölüm 4.2 · Arayüz

soyut veri tipi alt tip LSP

BM

Bertrand Meyer

d. 1950 · Fransız · Eiffel Software, ETH Zürih

Sözleşmeyle Tasarım · Eiffel

1980'lerin ortasında Eiffel dilini Sözleşmeyle Tasarım (Design by Contract) fikri üzerine kurdu: her yordam, çağıranın sağlaması gereken bir ön koşul ile kendisinin garanti ettiği bir son koşul belirtir; sınıf düzeyinde de bir sınıf değişmezi tanımlanır. Object-Oriented Software

Construction (1988) adlı kitabında açık–kapalı ilkesini (open–closed principle) de dile getirdi: modüller genişletmeye açık, değiştirmeye kapalı olmalıdır. Java'da yerleşik sözleşmeler olmadığından public metotların ön koşulları genellikle `IllegalArgumentException` fırlatılarak, iç varsayımlar ise `assert` ile denetlenir.

DERSTE **Bölüm 4.2 · Ön ve Son Koşullar** **Bölüm 4.2 · Sözleşme: Tam Örnek**

Bölüm 6 · IllegalArgumentException **Bölüm 4.4 · Neden Çok Biçimlilik?**

Sözleşmeyle Tasarım ön koşul açık–kapalı ilkesi

GB Grady Booch

d. 1955 · Amerikalı · Rational Software, IBM

Booch yöntemi · UML

Object-Oriented Analysis and Design with Applications (1991) adlı kitabı Booch yöntemini ve sınıflarla aralarındaki ilişkileri gösteren çizim kurallarını tanıttı. Rational Software'de 1994'te kendi yöntemini James Rumbaugh'un yöntemiyle birleştirmeye başladı, 1995'te Ivar Jacobson da onlara katıldı. Ortaya çıkan Birleşik Modelleme Dili'ni (Unified Modeling Language, UML) Object Management Group 1997'de standart olarak benimsedi. Ad, nitelikler ve işlemlerden oluşan UML sınıf kutusu ile kalıtımı gösteren içi boş üçgen uçlu ok, sınıf hiyerarşilerini çizmenin olağan yoludur.

DERSTE **Bölüm 4.3 · Hayvan Sınıf Hiyerarşisi** **Bölüm 4.1 · Örnek: Araba Sınıfı**

Bölüm 4.5 · Arayüzü Uygulamak

UML sınıf diyagramı nesne yönelimli tasarım

JR James Rumbaugh

d. 1947 · Amerikalı · General Electric R&D, Rational Software

OMT · UML

General Electric'in araştırma merkezinde, Object-Oriented Modeling and Design (1991) kitabında anlatılan Nesne Modelleme Tekniği'nin (Object Modeling Technique, OMT) geliştirilmesini yönetti. OMT bir sistemi, sınıflardan ve ilişkilerden oluşan bir nesne modeli ile durum diyagramlarından oluşan bir dinamik modelle betimliyordu. 1994'te OMT'yi Booch yöntemiyle UML'de birleştirmek için Rational'a katıldı. Bir iş parçacığının NEW, RUNNABLE, BLOCKED ve TERMINATED gibi durumlardan geçen yaşam döngüsü standart olarak UML durum diyagramlarıyla çizilir.

DERSTE **Bölüm 4.4 · Örnek Sınıf Hiyerarşisi** **Bölüm 10 · İş Parçacığı Durumları**

Bölüm 10 · Durum Diyagramı

OMT UML durum diyagramı

IJ Ivar Jacobson

d. 1939 · İsveçli · Ericsson, Objectory, Rational Software

Kullanım senaryoları · OOSE

Ericsson'da telefon santrali yazılımı üzerinde çalışırken, bir sistemi kullanıcılarının onunla ne yaptığı üzerinden betimleyen kullanım senaryolarını (use case) geliştirdi; bunları OOPSLA 1987'de sundu ve OOSE yöntemini bu senaryoların çevresinde kurdu (kitap, 1992). Şirketi Objectory 1995'te Rational'a katıldı ve kullanım senaryosu diyagramları UML'in parçası oldu. Sınıfları seçmeden önce kullanıcının neye ihtiyaç duyduğundan yola çıkmak ve testleri bu ihtiyaçlardan türetmek aynı düşünce çizgisini izler.

DERSTE **Bölüm 1.3 · Sorunu Tanımlama** **Bölüm 4.2 · Sınıf Tasarımı**

Bölüm 7 · Test Düzeyleri

kullanım senaryosu gereksinimler UML

TP Ken Thompson ve Rob Pike

d. 1943 · d. 1956 · Amerikalı · Kanadalı · Bell Labs

UTF-8 · 1992

Eylül 1992'de Unicode'un UTF-8 kodlamasını tasarladılar: ASCII karakterleri bu kodlamada tek bayt kalır, diğer karakterler ise iki ile dört bayt kaplar. ş gibi bir Türkçe harf UTF-8'de iki bayt tutar; bir UTF-8 dosyası yanlış karakter kümesiyle okunduğunda bu harfler bozuk görünür. Java'da bir `char` 16 bitlik bir UTF-16 birimidir; bu yüzden dosyadan okunan metin çözülür (decode) ve karakter kümesi (charset) açıkça belirtilebilir. Thompson Unix'in de yaratıcılarındandır; ikisi daha sonra Go dilini birlikte tasarladı.

DERSTE **Bölüm 1.1 · UTF-8** **Bölüm 1.1 · Unicode**

Bölüm 8 · Karakter Kodlamasını Belirtmek

UTF-8 Unicode karakter kodlaması

GF

Erich Gamma, Richard Helm, Ralph Johnson ve John Vlissides

Vlissides 1961–2005 · Zürih · Sidney · Urbana-Champaign · IBM Research

Design Patterns · 1994

Yazarları “Dörtlü Çete” (Gang of Four) diye tanınan Design Patterns: Elements of Reusable Object-Oriented Software (1994) kitabı, 23 tasarım desenini (design pattern) yaratımsal, yapısal ve davranışsal gruplarda topladı. Bu desenlerin birçoğu Java kütüphanesinde doğrudan görülür: bir akışı `BufferedReader` ile sarmak Dekoratör (Decorator) desenidir, `Iterator` Yineleyici desenidir, bir `Comparator` da Strateji desenidir. Olay dinleyiciler Gözlemci (Observer) desenini izler; bir algoritmanın adımlarını sabitleyen soyut sınıf ise Şablon Metot (Template Method) desenidir.

DERSTE **Bölüm 8 · Sarmalayıcı (Decorator) Akışlar** **Bölüm 4.5 · Şablon Metot İzleme** **Bölüm 5 · Iterator** **Bölüm 9 · Dinleyici Arayüzü ve Adaptör** **Bölüm 4.5 · Tasarım Desenleri**

tasarım desenleri Dekoratör Strateji Gözlemci

JG

James Gosling

d. 1955 · Kanadalı · Sun Microsystems

Java · 1995

Sun'ın Green projesi 1991'de tüketici cihazları için Gosling'in tasarladığı Oak adlı bir dil üzerinde çalışmaya başladı; adı Java olarak değiştirilen dil Mayıs 1995'te duyuruldu, JDK 1.0 da Ocak 1996'da çıktı. Gosling ilk derleyiciyi ve sanal makineyi yazdı: `javac` kaynak kodu bytecode'a çevirir, JVM de bu bytecode'u her platformda çalıştırır. Bu ilke “bir kez yaz, her yerde çalıştır” (write once, run anywhere) sloganıyla özetlenir. Bill Joy ve Guy Steele ile birlikte The Java Language Specification (1996) kitabını yazdı.

DERSTE **Bölüm 2.2 · Derleme** **Bölüm 2.2 · Komut Satırında Derleme ve Çalıştırma** **Bölüm 2.2 · Java Sanal Makinesi** **Bölüm 2.2 · Bytecode Yığını: Kod İzleme**

Java JVM bytecode javac

LY

Tim Lindholm ve Frank Yellin

günümüz · Amerikalı · Sun Microsystems

The Java Virtual Machine Specification · 1996

The Java Virtual Machine Specification (1996) adlı kitapları `.class` dosya biçimini, bytecode komut kümesini ve bytecode doğrulamasını herhangi bir gerçeklemeden bağımsız olarak tanımladı. Bu belirtim sayesinde bir kez derlenen bir sınıf dosyası HotSpot ya da OpenJ9 gibi farklı JVM'lerde çalışabilir ve `javap -c` her yerde aynı komutları gösterir. Lindholm daha sonra Java bellek modelinin yenilenmesi olan JSR 133'ün belirtim sorumlusu oldu.

DERSTE **Bölüm 2.2 · Bytecode Örneği** **Bölüm 2.2 · Java Sanal Makinesi**

Bölüm 10 · Java Sanal Makinesi (JVM)

JVM belirtimi sınıf dosyası bytecode

KB

Kent Beck ve Erich Gamma

d. 1961 · d. 1961 · Amerikalı · İsviçreli

JUnit · 1997

Beck, Smalltalk için SUnit adlı bir test çerçevesi yazmıştı. OOPSLA 1997 konferansına giderken uçakta Gamma ile birlikte, eşli programlama yaparak ve testleri önce yazarak JUnit'in ilk sürümünü yazdılar. JUnit, `assertEquals` gibi doğrulamalar kullanan otomatik birim testlerini Java geliştirmenin olağan bir parçası yaptı ve xUnit ailesinin doğmasına yol açtı. Beck, Uç Programlama'yı (Extreme Programming, 1999) da ortaya koydu ve test güdümlü geliştirmeyi Test-Driven Development: By Example (2002) kitabında anlattı; Gamma ise daha sonra Eclipse'in Java geliştirme araçlarını yönetti.

DERSTE **Bölüm 7 · JUnit** **Bölüm 7 · İlk JUnit Testi** **Bölüm 7 · Test Güdümlü Geliştirme**

JUnit birim testi TDD

JB

Joshua Bloch

d. 1961 · Amerikalı · Sun Microsystems, Google

Koleksiyon Çerçevesi · Effective Java

Java 1.2 (1998) ile gelen Koleksiyon Çerçevesi'nin (Collections Framework) tasarımını yönetti: `List`, `Set` ve `Map` arayüzleri, `ArrayList` ve `HashMap` gibi gerçeklemeleri ve `Collections` algoritmaları. `java.math` paketini ve `assert` deyimini de o tasarladı. Effective Java (2001; üçüncü baskı 2018) kitabı, `equals` geçersiz kılındığında `hashCode` metodunun da geçersiz kılınması ve kalıtım yerine oluşumun (composition) yeğlenmesi gibi kuralları bir araya getirir.

DERSTE **Bölüm 5 · Çerçevenin Üç Unsuru** **Bölüm 5 · Arayüze Göre Programlama**

Bölüm 4.3 · equals ve hashCode **Bölüm 4.2 · assert ile Değişmezi Denetlemek**

koleksiyonlar equals/hashCode Effective Java

GJ Gilad Bracha, Martin Odersky, David Stoutamire ve Philip Wadler

günümüz · Sun Microsystems · University of South Australia · Bell Labs

GJ (Generic Java) · 1998

OOPSLA 1998'de sundukları GJ tasarımı, tip parametrelerini Java'ya silme (erasure) yoluyla ekledi: derleyici `List<String>` tipini denetler, sonra tip argümanlarını siler ve gereken yerlere açık dönüşümler (cast) ekler; böylece sonuç var olan JVM'lerde çalışır. GJ, JSR 14'ün temeli olarak seçildi ve jenerikler (generics) Java 5 (2004) ile dile girdi. `new T[10]` ve `List<int>` yazılamamasının nedeni silmedir. Odersky'nin GJ derleyicisi JDK 1.3'ten başlayarak `javac`'ın temeli oldu; Odersky daha sonra Scala'yı yarattı, Wadler ise Haskell'in tasarımcılarından biridir.

DERSTE **Bölüm 5 · Jenerik Tipler ve Tip Güvenliği** **Bölüm 4.5 · Neden Jenerikler?**

jenerikler tip silme tip güvenliği

CH Koen Claessen ve John Hughes

günümüz · İsveçli · İngiliz · Chalmers University of Technology

QuickCheck · 2000

QuickCheck aracı (ICFP 2000), Haskell için özellik tabanlı testi (property-based testing) getirdi. Programcı örnek girdileri tek tek sıralamak yerine bir özellik yazar, örneğin bir listeyi iki kez ters çevirmenin özgün listeyi verdiğini; araç da bu özelliği yüzlerce rastgele girdi üzerinde sınar. Sonraki sürümler, testi bozan bir girdiyi küçültüp küçük bir karşı örneğe indirger. Fikir, jqwik ve junit-quickcheck gibi Java kütüphaneleri de dahil olmak üzere birçok dile taşındı.

DERSTE **Bölüm 7 · QuickCheck** **Bölüm 7 · Özellik Testi Hata Buluyor**

özellik tabanlı test rastgele test

RH D. Richard Hipp

d. 1961 · Amerikalı · Charlotte, Kuzey Carolina

SQLite · 2000

2000'de SQLite'ı başlattı: bütün veri tabanını tek bir dosyada tutan, sunucu gerektirmeyen ve kamu malı (public domain) olarak yayımlanan bir veri tabanı motoru. Geliştiricileri onu en yaygın kullanılan veri tabanı motoru olarak tanımlar. Ayrı bir sunucu gerekmediği için bir Java

programının tablo oluşturup SQL çalıştırmasına JDBC sürücüsü ve `jdbc:sqlite:okul.db` gibi bir bağlantı adresi yeter.

DERSTE **Bölüm 11 · SQLite Nedir?** **Bölüm 11 · Sunucu Tabanlı ve Gömülü**

Bölüm 11 · SQLite'a Bağlanma

SQLite gömülü veri tabanı JDBC

TP

Tim Peters

günümüz · Python çekirdek geliştiricisi

TimSort · 2002

2002'de Python'un liste sıralaması için TimSort'u yazdı. Algoritma verideki hazır sıralı parçaları (*run*) bulur, kısa parçaları eklemeli sıralamayla uzatır ve parçaları birleştirme sıralamasıyla birleştirir. Kararlıdır: eşit sayılan öğelerin sırası korunur. En kötü durumda O(n log n), zaten sıralı veride yaklaşık n karşılaştırma sürer. Java SE 7 nesne dizilerinin sıralamasını TimSort'a geçirdi; `Arrays.sort(Object[])` ve listeleri ona dayanarak sıralayan `List.sort` ile `Collections.sort` bu algoritmayı kullanır. İlkel dizileri sıralayan `Arrays.sort(int[])` ise Vladimir Yaroslavskiy, Jon Bentley ve Joshua Bloch'un çift pivotlu hızlı sıralamasını (*Dual-Pivot Quicksort*) kullanır.

DERSTE **Bölüm 5 · Sıralama Metotları**

TimSort kararlı sıralama Collections.sort

DL

Doug Lea

günümüz · Amerikalı · SUNY Oswego

java.util.concurrent · 2004

Concurrent Programming in Java (1997) kitabı iş parçacıkları için tasarım desenlerini bir araya getirdi; ücretsiz dağıttığı eş zamanlılık kütüphanesi de yönettiği JSR 166'ya dönüştü. Sonuç, Java 5'in (2004) `java.util.concurrent` paketidir: `ExecutorService` arkasındaki iş parçacığı havuzları, `ReentrantLock`, `ArrayBlockingQueue` gibi engelleyen kuyruklar, `CountDownLatch`, `Semaphore` ve atomik sınıflar. Java 7'ye fork/join çerçevesini de o ekledi.

DERSTE **Bölüm 10 · ExecutorService** **Bölüm 10 · ReentrantLock Sınıfı**

Bölüm 10 · Hazır Çözüm: ArrayBlockingQueue **Bölüm 10 · Atomik Sınıflar**

java.util.concurrent iş parçacığı havuzu kilitler

WP

William (Bill) Pugh

günümüz · Amerikalı · University of Maryland

Java bellek modeli · FindBugs

2000 dolaylarında Java'nın ilk bellek modelinin kusurlu olduğunu gösterdi; Jeremy Manson ve Sarita Adve ile birlikte, Java 5'te JSR 133 aracılığıyla benimsenen yeni modeli tasarladı. Bu model, bir iş parçacığının başka bir iş parçacığının yazdıklarını ne zaman göreceği konusunda `volatile` ve `synchronized` anahtar sözcüklerinin neyi garanti ettiğini tanımlar. Java bytecode'u için bir statik analiz aracı olan FindBugs'ın da yazarlarındandır (ardılı SpotBugs); atlamalı listeleri (skip list, 1990) de o buldu.

DERSTE **Bölüm 10 · volatile Alanlar** **Bölüm 10 · Görünürlük Sorunu** **Bölüm 7 · Statik Analiz: SpotBugs**

bellek modeli volatile statik analiz

JW

Jeannette Wing

d. 1956 · Amerikalı · Carnegie Mellon, Columbia University

Hesaplamalı düşünme · 2006

Communications of the ACM'de yayımlanan 2006 tarihli “Computational Thinking” yazısında problemi parçalamanın, soyutlamanın ve algoritmik düşünmenin yalnızca programcılar için değil herkes için temel beceriler olduğunu savundu. Bir problemi metotlara ve sınıflara bölmek ve ayrıntıları bir arayüzün arkasına gizlemek, aynı becerilerin Java'daki uygulamasıdır. Barbara Liskov ile birlikte davranışsal alt tiplemenin (behavioural subtyping) biçimsel tanımını da verdi (1994).

DERSTE **Bölüm 1.2 · Hesaplamanın Yapı Taşları** **Bölüm 1.2 · Mantık, Problem Çözme, Yaratıcılık** **Bölüm 3.2 · Ayrıştırma**

hesaplamalı düşünme soyutlama ayrıştırma

BG

Brian Goetz

günümüz · Amerikalı · Oracle

Java Concurrency in Practice · Java 8 lambda ifadeleri

İş parçacığı güvenliği ve görünürlük (visibility) konusunda standart kaynak olan Java Concurrency in Practice (2006; Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes ve Doug Lea ile) kitabının baş yazarıdır. Oracle'da Java Dil Mimarı olarak, Java 8'e (2014) lambda ifadelerini ve akış (stream) kütüphanesini getiren JSR 335'in belirtim sorumlusuydu. Ardından kayıtların (record), switch ifadelerinin, mühürlü sınıfların ve desen eşlemenin (pattern matching) kaynağı olan Project Amber'ı yönetti.

DERSTE **Bölüm 10 · İş Parçacığı Güvenliği** **Bölüm 4.5 · Anonim (Lambda) Fonksiyonlar**

Bölüm 4.1 · Kayıtlar (Records) **Bölüm 4.5 · sealed, non-sealed, permits**

Bölüm 5 · Akış API'si (java.util.stream)

lambda akışlar kayıtlar iş parçacığı güvenliği

MR

Mark Reinhold

günümüz · Amerikalı · Sun Microsystems, Oracle

Modül sistemi · altı aylık sürümler

Java Platform Group'un baş mimarı olarak Java SE 7, 8 ve 9'un belirtim sorumlusuydu. JDK'yı modüllere bölen ve paketlerin üstünde bir düzey olarak `module-info.java` dosyasını getiren Java 9 (2017) modül sistemini, yani Project Jigsaw'ı yönetti. 2017'de, Mart 2018'deki Java 10'dan başlayarak her altı ayda bir yeni Java sürümü çıkarılmasını önerdi; yeni `switch` gibi özelliklerin Java 14 gibi sürüm numaralarına bağlanmasının nedeni budur.

DERSTE **Bölüm 2.2 · Java Platformu** **Bölüm 4.2 · Paketler**

Bölüm 3.1 · Yeni switch (Java 14+)

modüller sürüm takvimi Java SE

Yıllar ve katkılar özgün yayınlara ve ansiklopedik kaynaklara dayanır. “Günümüz” ibaresi, doğum yılı yayımlanmış kaynaklarda verilmeyen yaşayan kişileri gösterir. Öklid ve el-Hârizmî için verilen yıllar yaklaşıktır.