DERS KİTABI

Bilgisayar ve Programlamaya Giriş

Sercan Külcü

Giresun Üniversitesi · Bilgisayar Mühendisliği Bölümü

Bilgisayar ve Programlamaya Giriş

Sercan Külcü
Giresun Üniversitesi, Bilgisayar Mühendisliği Bölümü
Giresun, Türkiye

Bu kitap, şu adreste yayımlanan ders notlarının basılı hâlidir:
`sercankulcu.github.io/teaching/computer-and-introduction-to-programming/`

Sunumlar ve çalışma soruları o sayfadadır. Programlama bölümlerinin kodu `github.com/sercankulcu/computer-programming-java` deposundadır.

© Sercan Külcü. Tüm hakları saklıdır.

Notların 14 Kasım 2026 tarihli hâlinden üretilmiştir.

Önsöz

Bu kitap bilgisayarın ve bilgi teknolojilerinin temellerinden başlar: donanım, işletim sistemleri, İnternet ve ofis uygulamaları. Ardından algoritma ve akış diyagramlarıyla programlamaya geçer ve Java ile değişkenlerden özyinelemeye kadar programlamanın temel yapılarını işler. Kitap, Giresun Üniversitesi Bilgisayar Mühendisliği Bölümü'nde verilen Bilgisayar ve Programlamaya Giriş dersinin notlarından oluşur ve bölümleri dersteki sırayla izler.

Kitap üç kısımdır. Kısımlar sırasıyla şunlardır: **Bilgisayar ve Bilgi Teknolojileri** (Bölüm 1–3), **Algoritmadan Programa** (Bölüm 4–5) ve **Java ile Programlama** (Bölüm 6–12).

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.

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

Sercan Külcü
Giresun

İçindekiler

Dersin Haritası 1

KISIM I · BİLGİSAYAR VE BİLGİ TEKNOLOJİLERİ

1 Bilgisayar Temelleri ve Hesaplamalı Düşünme 5

1.1 Temel Kavramlar 6
1.2 Bilgisayar Donanımı 64
1.3 Bilgisayarlar 121
1.4 İnternet 176
1.5 Hesaplama ve Hesaplamalı Düşünme 243

2 İşletim Sistemleri 287

2.1 Yazılım ve İşletim Sistemleri 288
2.2 Windows İşletim Sistemi 347

3 Ofis Uygulamaları 413

3.1 Microsoft Word 414
3.2 Microsoft Excel 477
3.3 Microsoft PowerPoint 540

KISIM II · ALGORİTMADAN PROGRAMA

4 Algoritmalar ve Programlamaya Giriş 602

4.1 Algoritmalar 603
4.2 Akış Diyagramları 651
4.3 Programlamaya Giriş 713

5 Java'ya Giriş ve İlk Program 787

5.1 Program ve programlama dili 788
5.2 Kaynak kod ve araçlar 790
5.3 Java 792
5.4 İlk Java programı 795
5.5 Kaçış dizileri ve dizgi birleştirme 803
5.6 Java dosyasının yapısı 810
5.7 Derleme ve çalıştırma 811
5.8 Bytecode ve JVM 816
5.9 Dilin yapı taşları 821
5.10 Açıklamalar 825
5.11 Hatalar 828
5.12 Kodlamadan önce 836
5.13 Özet 838
5.14 Kavram listesi 839
5.15 Kendini Dene 840

5.16 İleri okuma 845
5.17 Kaynaklar 846

KISIM III · JAVA İLE PROGRAMLAMA

6 Değişkenler, Veri Tipleri ve İşleçler 848
6.1 Değişkenler 848
6.2 Veri tipleri 858
6.3 Tip dönüşümü 872
6.4 İşleçler 880
6.5 Girdi ve çıktı 897
6.6 Örnekler 906
6.7 Yaygın hatalar 914
6.8 Özet 916
6.9 Kavram listesi 917
6.10 Kendini Dene 918
6.11 İleri okuma 920
6.12 Kaynaklar 920

7 Koşullu İfadeler 921
7.1 Akış yapıları 922
7.2 Koşul ve if 925
7.3 Çok yollu karar 931
7.4 İç içe kararlar 936
7.5 Mantıksal işleçlerle karar 941
7.6 Koşul işleci ?: 947
7.7 Dizgileri karşılaştırmak: == ve equals 949
7.8 switch 951
7.9 Örnekler 958
7.10 Özet 973
7.11 Kavram listesi 974
7.12 Kendini Dene 975
7.13 İleri okuma 977
7.14 Kaynaklar 977

8 Döngüler 978
8.1 Döngü nedir? 979
8.2 for döngüsü 981
8.3 while döngüsü 985
8.4 do-while döngüsü 987
8.5 break ve continue 990
8.6 Döngü hataları ve doğrulama 994
8.7 Rakam kalıbı 999
8.8 Çarpım ve bölen kalıpları 1005
8.9 Girdiyle çalışan döngüler 1011
8.10 İç içe döngüler 1014

8.11 Desenler 1016
8.12 Test tasarımı 1022
8.13 Özet 1023
8.14 Kavram listesi 1024
8.15 Kendini Dene 1024
8.16 İleri okuma 1027
8.17 Kaynaklar 1027

9 Metotlar 1028

9.1 Metot nedir? 1029
9.2 Metot tanımı ve çağrısı 1031
9.3 Çağrı yığını 1036
9.4 Metot tanımlamasının parçaları 1039
9.5 Dönüş değeri 1043
9.6 Parametre aktarımı 1050
9.7 Aşırı yükleme 1054
9.8 Değişken sayıda argüman 1060
9.9 Kapsam 1064
9.10 Hazır metotlar: Math 1071
9.11 Metot tasarımı 1074
9.12 Örnek metotlar 1077
9.13 Özet 1093
9.14 Kavram listesi 1094
9.15 Kendini Dene 1094
9.16 İleri okuma 1097
9.17 Kaynaklar 1097

10 Diziler 1098

10.1 Dizi kavramı 1099
10.2 Dizi bildirme ve oluşturma 1102
10.3 Öğelere erişim 1106
10.4 Döngülerle gezinme 1110
10.5 Temel dizi işlemleri 1113
10.6 Kopyalama, Arrays sınıfı ve metotlar 1122
10.7 Arama ve sıralama 1129
10.8 İki boyutlu diziler 1138
10.9 Dizi mi liste mi? 1146
10.10 Yaygın hatalar 1147
10.11 Özet 1149
10.12 Kavram listesi 1150
10.13 Kendini Dene 1150
10.14 İleri okuma 1156
10.15 Kaynaklar 1156

11 Dizgiler 1157

11.1 Dizgi nedir? 1158

11.2 Değiştirilemezlik 1164
11.3 İndisleme 1168
11.4 Dizgi metotları 1170
11.5 Karşılaştırma 1182
11.6 Bellek: dizgi havuzu 1192
11.7 Scanner ile metin girdisi 1195
11.8 Birleştirme ve StringBuilder 1200
11.9 Uygulamalar 1204
11.10 printf ile biçimleme 1221
11.11 Unicode ve metin uzunluğu 1224
11.12 Özet 1226
11.13 Kavram listesi 1227
11.14 Kendini Dene 1227
11.15 İleri okuma 1229
11.16 Kaynaklar 1230

12 Özyineleme 1231

12.1 Özyineleme nedir? 1232
12.2 Faktöriyel ve çağrı yığını 1235
12.3 Tek çağrılı örnekler 1239
12.4 Üs alma 1245
12.5 Böl ve fethet 1248
12.6 İki çağrılı özyineleme 1253
12.7 Çağrı yığını ve yığın taşması 1257
12.8 Özyineleme mi döngü mü? 1261
12.9 Fibonacci ve bellekleme 1263
12.10 Kuyruk özyinelemesi 1272
12.11 Taşma, uç durumlar ve hata ayıklama 1274
12.12 Özet 1277
12.13 Kavram listesi 1278
12.14 Kendini Dene 1279
12.15 İleri okuma 1281
12.16 Kaynaklar 1281

EKLER

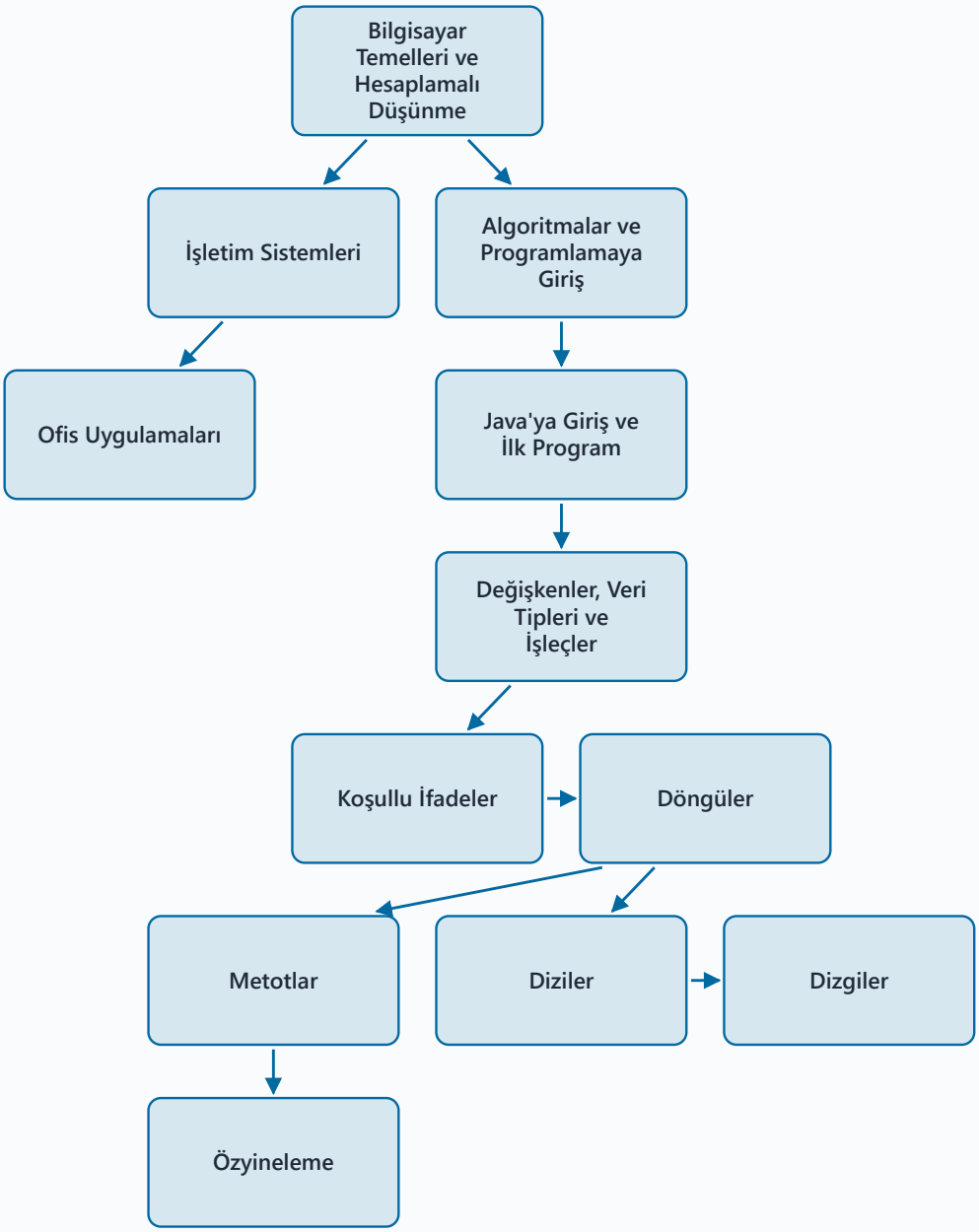
Ek A Terim Sözlüğü 1282

Ek B Bilgisayar ve Programlamaya Giriş İçin Ön Bilgiler 1297

Ek C Bilgisayar ve Programlamada Önemli İsimler 1321

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. Bilgisayarın temellerinden algoritmaya, oradan Java ile programlamaya geçilir.

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
Bilgisayar Temelleri ve Hesaplamalı Düşünme (1)	—	Verinin bitlerle temsili, donanım ve hesaplamalı düşünme; dersin bütün kavramlarının zemini.
İşletim Sistemleri (2)	Bilgisayar Temelleri ve Hesaplamalı Düşünme	Donanımı yöneten yazılım: önyükleme, süreç, bellek ve dosya sistemi.

Konu (bölüm)	Dayandığı konu	Bağ
Ofis Uygulamaları (3)	İşletim Sistemleri	İşletim sistemi üzerinde çalışan uygulamalar: Word, Excel ve PowerPoint.
Algoritmalar ve Programlamaya Giriş (4)	Bilgisayar Temelleri ve Hesaplamalı Düşünme	Hesaplamalı düşünme algoritmaya dönüşür: sözde kod, akış diyagramı, arama ve sıralama.
Java'ya Giriş ve İlk Program (5)	İşletim Sistemleri, Algoritmalar ve Programlamaya Giriş	Algoritma Java koduna dönüşür: kaynak kod, derleyici, bytecode ve JVM.
Değişkenler, Veri Tipleri ve İşleçler (6)	Bilgisayar Temelleri ve Hesaplamalı Düşünme, Java'ya Giriş ve İlk Program	Verinin programdaki karşılığı: ilkel tipler, taşma, tip dönüşümü ve işleçler.
Koşullu İfadeler (7)	Değişkenler, Veri Tipleri ve İşleçler	boolean ifadeler programın akışını seçer: if, else if, switch.
Döngüler (8)	Koşullu İfadeler	Koşul doğru kaldıkça tekrar: for, while, do-while; sayaç ve biriktirici.
Metotlar (9)	Döngüler	Tekrar eden işi adlandırmak: parametre, dönüş değeri, çağrı yığını.
Diziler (10)	Döngüler, Metotlar	Döngüyle gezilen indisli veri; Algoritmalar bölümündeki arama ve sıralamanın Java kodu.
Dizgiler (11)	Metotlar, Diziler	Karakter dizisi: String metotları, değiştirilemezlik ve StringBuilder.
Özyineleme (12)	Algoritmalar ve Programlamaya Giriş, Metotlar	Kendini çağıran metot: taban durumu, çağrı yığını, Hanoi ve ikili arama.

Tablo 1: Haritadaki bağlar.

Haritanın ana omurgası şöyle okunabilir: **bilgisayar → algoritma → program → yapı → yeniden kullanım**. İlk kısım makineyi ve onu yöneten yazılımı tanıtır. Algoritmalar bölümü problem çözme adımlarını dilden bağımsız biçimde yazar; Java bölümleri bu adımları sırayla koda çevirir: önce değişken ve işleç, sonra karar ve döngü, en sonda metot. Diziler ve dizgiler döngü ile metodun birlikte kullanıldığı yerdir; özyineleme ise metot çağrısını problemi küçültmenin aracı yapar.

KISIM I

Bilgisayar ve Bilgi Teknolojileri

Bölüm 1–3

KISIM I · BÖLÜM 1

Bilgisayar Temelleri ve Hesaplamalı Düşünme

Bilgisayar, veriyi kesin talimatlarla işleyerek problem çözen bir makinedir. Bölüm beş sunumdan oluşur. Temel Kavramlar verinin bitlerle nasıl temsil edildiğini ve bilgisayarın temel çalışma döngüsünü anlatır. Bilgisayar Donanımı sunumu işlemciyi, bellek hiyerarşisini, depolamayı, giriş ve çıkış birimlerini tanıtır. Bilgisayarlar ikiye tümleyen, kayan noktalı sayılar, karakter kodlamaları, mantık kapıları ve getir–çöz–yürüt döngüsüyle bilgisayar mimarisini ele alır. İnternet sunumu IP adresi, DNS, HTTP/HTTPS ve güvenli kullanımı; Hesaplama sunumu ise hesaplama kavramını, Turing makinesini ve hesaplamalı düşünmeyi anlatır.

BU BÖLÜMÜN NOTLARI

1.1 Temel Kavramlar

1.2 Bilgisayar Donanımı

1.3 Bilgisayarlar

1.4 İnternet

1.5 Hesaplama ve Hesaplamalı Düşünme

KISIM I · BÖLÜM 1.1

Temel Kavramlar

Bilgisayar, ne iş yaparsa yapsın, aynı basit düzenle çalışır: veriyi alır, işler, sonucu sunar ve gerekirse saklar. Bu düzenin altında daha da basit bir gerçek yatar. Bilgisayarın içindeki her şey, yazı da resim de müzik de, yalnız 0 ve 1'lerden oluşan dizilerle tutulur. Not, önce bu çalışma döngüsünü ve donanımın hangi parçasının hangi aşamada görev aldığını anlatır. Ardından bit ve bayt kavramlarına, ikili sayı sistemine ve onluk ile ikili arasındaki dönüşümlere geçer. Metnin, görüntünün, sesin ve videonun nasıl sayıya çevrildiği, dosyaların neden sıkıştırıldığı, bellek birimleri ve ağ hızının nasıl ölçüldüğü adım adım hesaplarla gösterilir. Mantıksal işlemler ve parite biti, bilgisayarın veriyle karar vermesini ve hata yakalamasını örnekler. Son iki kesim, bilgisayar türlerini ve bilgi teknolojilerinin günlük yaşamdaki yerini ele alır.

ÖĞRENME ÇIKTILARI

Çalışma prensibi
Giriş, işlem, çıkış ve depolama aşamalarını açıklamak; verilen bir işte her aşamayı ve görev alan donanımı belirlemek.

Sayısal temsil
Bit, bayt ve ikili sayı sisteminin rolünü yorumlamak; ikili, onluk ve onaltılık sayılar arasında dönüşüm yapmak; metin, görüntü ve sesin kapladığı yeri hesaplamak.

Veri işleme
Aritmetik, karşılaştırma ve mantıksal işlemlerin kararlarda nasıl birleştiğini göstermek; parite bitiyle hata yakalamanın gücünü ve sınırını açıklamak.

Bilgisayar türleri
Bilgisayarları kullanım amacı ve kapasitesine göre karşılaştırmak; bir ihtiyaç için uygun türü gerekçesiyle seçmek.

Bilgi teknolojileri
Verinin bilgiye dönüşümünü ve veri yaşam döngüsünü açıklamak; bilgi teknolojilerinin günlük yaşamdaki rolünü ve getirdiği sorumlulukları tartışmak.

1.1.1 Çalışma prensibi

Bir hesap makinesi, bir akıllı telefon, bir oyun konsolu ve bir bankanın büyük sunucuları dışarıdan birbirine hiç benzemez. Boyutları, fiyatları ve yaptıkları işler çok farklıdır. Yine de hepsi aynı temel düzenle çalışır. Bu kesim o ortak düzeni anlatır: bilgisayarın ne olduğunu, bir işi hangi aşamalardan geçirerek yaptığını ve her aşamada hangi donanımın görev aldığını.

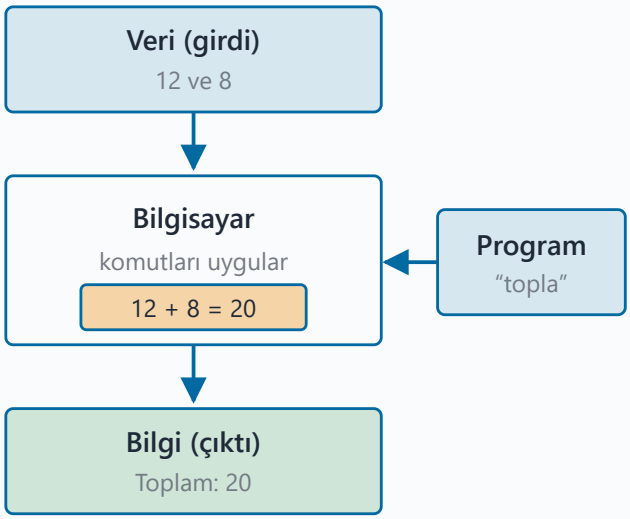
1.1.1.1 Bilgisayar nedir?

Günlük dilde "bilgisayar" denince çoğunlukla masadaki kasa ya da dizüstü cihaz akla gelir. Bilgisayar biliminde kavram daha geniştir. Bir çamaşır makinesinin içindeki küçük denetim kartı da, bir sunucu odasını dolduran dolaplar da bilgisayardır. Onları ortak kılan, belirli bir işi yapan tek amaçlı bir alet olmaları değil, kendilerine verilen komutları izleyerek veriyi işlemeleridir.

Tanım 1.1.1

Bilgisayar (*computer*), aldığı verileri bir programdaki komutlara göre işleyen elektronik bir sistemdir. Bilgisayara giren ham değerlere **veri** (*data*), işlemden sonra elde edilen ve yorumlanabilen sonuca **bilgi** (*information*) denir. Bilgisayara ne yapacağını adım adım söyleyen komutlar dizisine **program** denir.

Tanımdaki üç parça birbirinden ayrı düşünülmelidir. Veri, işlenecek malzemedir: bir sınav notu, bir sıcaklık ölçümü, klavyeden basılan bir harf. Program, malzemeye ne yapılacağını söyleyen tariftir: "bu iki sayıyı topla", "ortalamayı hesapla", "sonucu ekrana yaz". Bilgisayar ise tarifi harfiyen uygulayan makinedir. Aynı makineye farklı bir program yüklenince bambaşka bir iş yapar. Bilgisayarı hesap makinesinden ayıran asıl özellik de budur: ne yapacağı donanımına gömülü değildir, programla belirlenir.



Şekil 1.1.1: Bilgisayarın en yalın modeli. Veri (12 ve 8) bilgisayara girer, "topla" komutunu içeren program bu veriyle çalışır ve bilgi (toplam: 20) çıkar.

Şekil 1.1.1 bu modeli en küçük ölçekte gösterir. 12 ve 8 sayıları tek başına bir şey anlatmaz; nereden geldikleri bilinmez. Program bu iki sayıyı toplamasını söyler ve bilgisayar 20 sonucunu üretir. "Toplam: 20" artık okunup yorumlanabilen bir sonuçtur. Veri ile bilgi arasındaki ayrımın ayrıntılı tanımı Tanım 1.1.21'de verilir.

Tanım, bilgisayarın temel görevlerini de içerir. Bilgisayar kullanıcıdan ya da başka sistemlerden *veri alır*. Bu veriyi programa göre *işler* ve anlamlı bir sonuç üretir. Sonucu kullanıcıya *sunar* ya da daha sonra kullanmak üzere *saklar*. Bu dört eylem, yani almak, işlemek, sunmak ve saklamak, bilgisayarın çalışma döngüsünün aşamalarıdır.

Sık yapılan hata

Bilgisayarın "düşündüğünü" ya da "anladığını" sanmak. Bilgisayar, programdaki komutları çok hızlı ve hatasız uygular, ama komutların amacını bilmez. 12 ile 8'i toplarken bunların sınav notu mu, fiyat mı, sıcaklık mı olduğunu bilmez. Anlamı veren, veriye bağlam ekleyen insandır.

1.1.1.2 Temel çalışma döngüsü

Bilgisayar her işte aynı dört aşamayı kullanır: **giriş** (*input*), **işleme** (*processing*), **çıkış** (*output*) ve **depolama** (*storage*). İngilizce baş harflerinden ötürü bu modele kısaca IPOS modeli de denir. Bir metin yazarken, bir video izlerken ya da bir oyun oynarken bu dört aşama saniyede milyonlarca kez tekrarlanır. Aşamaları tanımanın en iyi yolu, onları tek bir somut iş üzerinde izlemektir.

Tanım 1.1.2

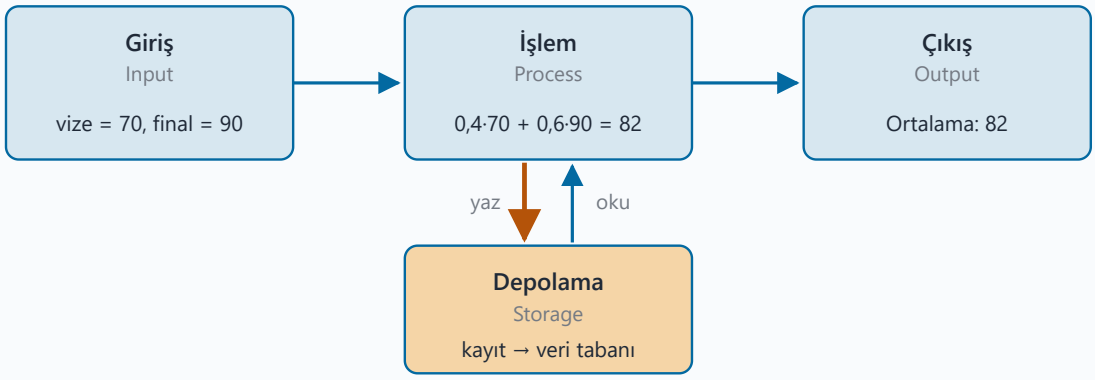
Giriş, verinin dış dünyadan bilgisayara alınmasıdır. **İşlem**, verinin program komutlarına göre hesaplanması, karşılaştırılması ya da dönüştürülmesidir. **Çıkış**, işlenmiş sonucun kullanıcıya ya da başka bir sisteme sunulmasıdır. **Depolama**, verinin ve sonuçların daha sonra yeniden kullanılmak üzere saklanmasıdır.

Örnek 1.1.1

Bir öğrenci bilgi sistemi, dönem ortalamasını *0,4 × vize + 0,6 × final* formülüyle hesaplıyor. Öğretmen bir öğrenci için vize = 70, final = 90 giriyor. Dört aşama şöyle işler:

1. **Giriş.** Öğretmen formdaki kutulara 70 ve 90 yazar. Klavye, basılan her tuşu bilgisayara iletir. Bu aşamada veri yalnızca bilgisayarın içine taşınır; henüz hesap yapılmamıştır.
2. **İşlem.** Merkezî işlem birimi (CPU) formülü uygular. Önce iki çarpma yapılır: *0,4 × 70 = 28* ve *0,6 × 90 = 54*. Sonra iki sonuç toplanır: *28 + 54 = 82*. Bunlar aritmetik işlemlerdir.
3. **Çıkış.** Sonuç ekranda "Ortalama: 82" biçiminde görünür. Monitör burada çıkış birimidir.
4. **Depolama.** Öğrencinin not kaydı veri tabanına yazılır. Böylece bilgisayar kapansa da kayıt kaybolmaz; dönem sonunda transkript hazırlanırken kayıt yeniden okunur ve işlenir.

Sonuç: ekranda "Ortalama: 82" görünür ve kayıt kalıcı olarak saklanır.



Şekil 1.1.2: Örnek 1.1.1'in dört aşaması. Giriş (vize = 70, final = 90) işleme gider, işlem sonucu (82) çıkışta görünür. Depolama ile işlem arasındaki ok iki yönlüdür: kayıt veri tabanına yazılır ve gerektiğinde oradan geri okunur.

Şekil 1.1.2 dört aşamanın birbirine nasıl bağlandığını gösterir. Giriş, işlem ve çıkış tek yönlü bir hat gibidir: veri soldan girer, sağdan sonuç olarak çıkar. Depolama ise bu hattın yanında durur ve işlemle iki yönde konuşur. İşlem, sonucu depolamaya yazabilir; ileride aynı veriye gerek olursa depolamadan

okuyabilir. Ortalama örneğinde dönem sonundaki transkript hesabı, daha önce saklanmış notları okuyarak yapılır. O hesap için depolama, girişin yerini tutar.

Dört aşama her işte aynı adlarla bulunur, ama her işte aynı ağırlıkta değildir. Bir hesap makinesinde depolama neredeyse yoktur; sonuç ekranda görünür ve bir sonraki hesapta kaybolur. Bir arşiv sisteminde ise asıl iş depolamadır. Bir modeli bir işe uygularken önce dört sorunun cevabı aranır: veri nereden geliyor, üzerinde hangi hesap yapılıyor, sonuç nereye gidiyor, ne saklanıyor?

Örnek 1.1.2

Bir ATM'den 500 TL çekilirken aşamalar şöyle ayrılır:

1. **Giriş:** kart okuyucu kartın bilgilerini okur; kullanıcı tuş takımından şifreyi ve 500 TL tutarını girer.
2. **İşlem:** bankanın bilgisayarı şifreyi doğrular, bakiyeyi çekilecek tutarla karşılaştırır (bakiye ≥ 500 mü?) ve yeni bakiyeyi hesaplar (bakiye − 500).
3. **Çıkış:** para çekmecesi banknotları verir, ekran yeni bakiyeyi gösterir, yazıcı fişi basar.
4. **Depolama:** işlem kaydı ve yeni bakiye bankanın veri tabanına yazılır.

İşlem aşamasında iki tür iş vardır: bir karşılaştırma (bakiye yeterli mi?) ve bir aritmetik işlem (çıkarma). İkisi de işlemcinin işidir.

Sık yapılan hata

Ekranda bir sonucun görünmesini depolama sanmak. Ekran çıkıştır; gösterilen sonuç, bilgisayar kapanınca ya da ekran değişince kaybolur. Bir sonucun saklanmış sayılması için kalıcı bir ortama, örneğin diske ya da veri tabanına yazılması gerekir.

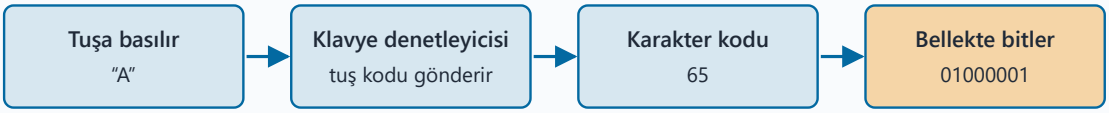
1.1.1.3 Giriş

Giriş, bilgisayarın kullanıcıdan, bir sensörden ya da başka bir sistemden veri almasıdır. Bu işi yapan donanıma **giriş birimi** (*input unit*) denir. En tanıdık giriş birimleri klavye, fare, mikrofon ve dokunmatik ekrandır. Sensörler de giriş birimidir: bir termometre sıcaklığı, bir ivmeölçer telefonun eğimini, bir kamera ışığı ölçer.

Giriş birimi	Algıladığı fiziksel olay	Bilgisayara giden veri
Klavye	Tuşa basılması	Tuşun kodu (karakter ya da komut)
Fare	Hareket ve düğmeye basma	Konum değişimi ve tıklama
Mikrofon	Hava basıncındaki dalgalanma (ses)	Belirli aralıklarla ölçülmüş sayı dizisi
Sensör	Sıcaklık, ışık, ivme gibi bir büyüklük	Ölçülen değer

Tablo 1.1.1: Yaygın giriş birimleri ve bilgisayara ilettikleri veri.

Tablodaki giriş birimlerinin hepsi aynı işi yapar: fiziksel bir olayı sayılara, daha doğrusu bitlere çevirir. Bilgisayarın içinde tuş, ses ya da sıcaklık yoktur; yalnızca sayılar vardır. Giriş biriminin görevi, dış dünyadaki olayı bilgisayarın işleyebileceği bu biçime getirmektir.



Şekil 1.1.3: Klavyede "A" tuşuna basıldığında olanlar. Klavye denetleyicisi tuş kodunu gönderir, bu kod "A" karakterinin 65 numaralı koduna çevrilir ve bellekte 01000001 bitleri olarak saklanır.

Şekil 1.1.3 bu çeviriyi bir tuş vuruşu için gösterir. Tuşa basılınca klavyenin içindeki küçük denetleyici, hangi tuşa basıldığını belirten bir kod gönderir. İşletim sistemi bu kodu, seçili klavye düzenine göre bir karaktere çevirir. Büyük "A" harfinin karakter kodu 65'tir. 65 sayısı bellekte sekiz bitlik 01000001 dizisi olarak tutulur. Bu dizinin neden 65'e karşılık geldiği Örnek 1.1.4'te adım adım hesaplanır.

Sık yapılan hata

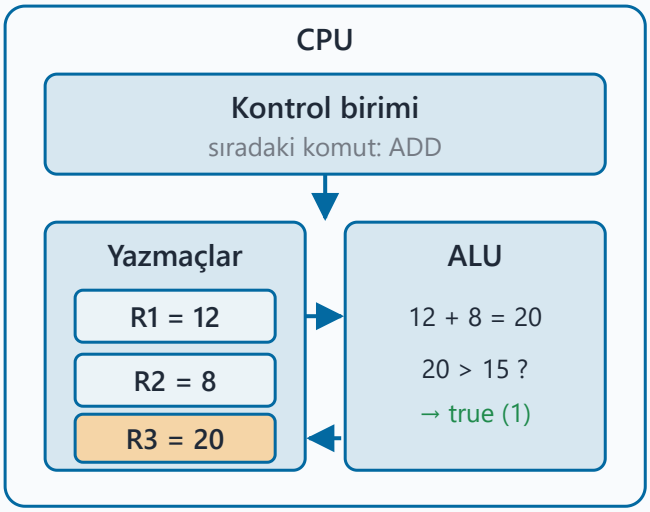
Dokunmatik ekranı yalnızca çıkış birimi saymak. Dokunmatik ekran hem görüntü gösterir (çıkış) hem de dokunulan noktanın konumunu algılar (giriş). Bu yüzden hem giriş hem çıkış birimidir.

1.1.1.4 İşlem

İşlem aşamasının merkezinde **merkezî işlem birimi** (*central processing unit*, CPU) bulunur. Günlük dilde buna kısaca işlemci denir. İşlemci program komutlarını sırayla yürütür. Her komut çok basit bir iştir: iki sayıyı topla, bir sayıyı bellekten oku, iki sayıyı karşılaştır, sonuca göre başka bir komuta atla. Karmaşık programlar, milyonlarca böyle küçük komutun art arda yürütülmesinden oluşur.

Tanım 1.1.3

İşlemcinin üç temel parçası vardır. **Kontrol birimi** (*control unit*), sıradaki komutu getirir, ne istediğini çözer ve öteki parçalara ne yapacaklarını söyler; komutların sırasını o yönetir. **Aritmetik-mantık birimi** (*arithmetic logic unit*, ALU), toplama ve çıkarma gibi hesapları ve karşılaştırmaları yapar. **Yazmaçlar** (*registers*), işlemcinin içindeki çok küçük ve çok hızlı bellek hücreleridir; üzerinde çalışılan sayıları geçici olarak tutar.



Şekil 1.1.4: İşlemcinin içi. Kontrol birimi sıradaki komutu (ADD, yani topla) yönetir. Yazmaçlar R1 = 12 ve R2 = 8 değerlerini tutar. ALU toplamı (20) hesaplayıp R3'e yazar ve ardından 20 > 15 karşılaştırmasını yapar; sonuç doğrudur (1).

Şekil 1.1.4 tek bir toplama komutunun yürütülmesini gösterir. Kontrol birimi sıradaki komutun ADD (topla) olduğunu çözer. Toplanacak sayılar, 12 ve 8, daha önce R1 ve R2 yazmaçlarına yüklenmiştir. ALU bu iki yazmacı okur, *12 + 8 = 20* hesabını yapar ve sonucu R3 yazmacına yazar. Ardından programın bir sonraki komutu bir karşılaştırmadır: 20, 15'ten büyük mü? ALU bu soruyu da cevaplar ve "doğru" anlamına gelen 1'i üretir.

ALU'nun yaptığı işler iki gruba ayrılır. **Aritmetik işlemler** sayılarla hesap yapar: toplama, çıkarma, çarpma, bölme. **Mantıksal işlemler** ise karşılaştırma yapar ve koşulları birleştirir: "20 > 15 mi?", "hem puan yeterli hem devam tamam mı?". Mantıksal işlemlerin sonucu her zaman "doğru" ya da "yanlış"tır. Bilgisayarın "karar vermesi", yani bir duruma göre farklı yollar izlemesi, bu işlemlerle olur. İki tür işlem Şekil 1.1.21'de bir not hesabı üzerinde ayrıntılı olarak gösterilir.

İşlemcide hem veriler hem de komutların kendisi ikili biçimde, yani 0 ve 1'lerle temsil edilir. "ADD" komutu da bellekte bir bit dizisidir; kontrol birimi bu diziyi okuyup hangi işin yapılacağını anlar. Programın da veri gibi bellekte durması, bilgisayara yeni bir iş yaptırmak için yalnızca belleğe yeni bir program yüklemeyi yeterli kılar.

Sık yapılan hata

İşlemcinin hızını yalnızca "kaç GHz" sorusuyla ölçmek. Saat hızı, işlemcinin saniyede kaç temel adım attığını gösterir. Bir komutun kaç adımda bittiği, kaç çekirdek bulunduğu ve verinin bellekten ne kadar çabuk geldiği de hızı belirler. Aynı saat hızındaki iki işlemci aynı programı farklı sürelerde çalıştırabilir.

1.1.1.5 Çıkış ve depolama

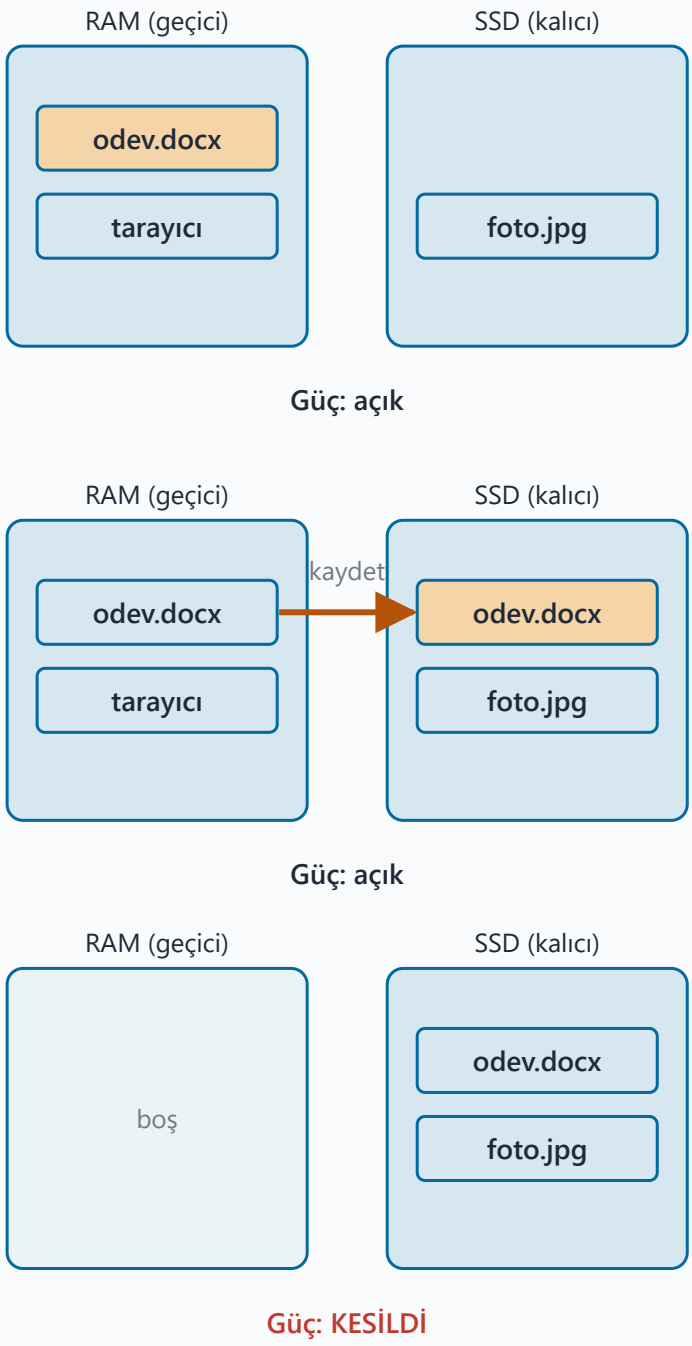
Çıkış birimleri (*output unit*), işlenmiş bilgiyi insanın algılayabileceği ya da başka bir sistemin kullanabileceği biçime getirir. Monitör sonucu görüntü olarak, yazıcı kâğıt üzerinde, hoparlör ses olarak sunar. Giriş birimlerinin yaptığının tersi olur: bitler yeniden ışığa, mürekkebe ya da sese dönüşür. Ortalama örneğindeki "Ortalama: 82" yazısı, hesap sonucunun ekrandaki çıkışıdır.

Depolama iki türlüdür ve bu ayrım günlük bilgisayar kullanımında çok önemlidir.

Tanım 1.1.4

Geçici depolama (*volatile storage*), elektrik kesilince içeriğini kaybeden bellektir. Bilgisayarın ana belleği olan **RAM** (*random access memory*, rastgele erişimli bellek) geçicidir; çok hızlıdır ve açık programlarla belgeleri tutar. **Kalıcı depolama** (*non-volatile storage*), güç olmadan da veriyi koruyan ortamdır: SSD, sabit disk (HDD), USB bellek ve hafıza kartı bu türdendir.

Bir belge açıldığında belgenin kopyası kalıcı depolamadan RAM'e okunur. Üzerinde yapılan her değişiklik önce RAM'deki bu kopyaya işlenir. "Kaydet" komutu verilince RAM'deki güncel hâl kalıcı depolamaya, örneğin SSD'ye yazılır. Bu sıralamanın nedeni hızdır. RAM, SSD'den çok daha hızlıdır. İşlemci her harf için SSD'ye gitseydi bilgisayar çok yavaşlardı. Hızın bedeli ise güvenliktir: RAM'deki her şey güç kesilince silinir.



Şekil 1.1.5: RAM ile SSD'nin görev paylaşımı. (a) Çalışırken açık belge (odev.docx) ve tarayıcı RAM'dedir; SSD'de yalnızca foto.jpg vardır. (b) "Kaydet" komutu belgeyi RAM'den SSD'ye kopyalar. (c) Güç kesilince RAM boşalır; SSD'deki odev.docx ve foto.jpg korunur.

Şekil 1.1.5 bu üç anı sırayla gösterir. (a)'da belge yalnızca RAM'de yaşar; o anda elektrik kesilseydi bütün çalışma kaybolurdu. (b)'de "Kaydet" ile belgenin bir kopyası SSD'ye yazılır. (c)'de güç kesilir: RAM'deki her şey silinir, ama SSD'ye yazılmış iki dosya yerinde durur. Bilgisayar yeniden açılıp belge açıldığında, SSD'deki son kaydedilmiş hâl RAM'e yeniden okunur.

Özellik	RAM (geçici)	SSD, sabit disk (kalıcı)
Güç kesilince	İçerik silinir	İçerik korunur
Hız	Çok hızlı	RAM'e göre yavaş
Tipik kapasite	8–32 GB	256 GB – birkaç TB
Ne tutar?	Çalışan programlar, açık belgeler	Kurulu programlar, kaydedilmiş dosyalar

Tablo 1.1.2: Geçici ve kalıcı depolamanın karşılaştırması.

Örnek 1.1.3

Bir öğrenci saat 14.00'te ödev belgesini kaydediyor. 14.00 ile 14.20 arasında belgeye iki sayfa ekliyor ama kaydetmiyor. 14.20'de elektrik kesiliyor. Bilgisayar yeniden açılıp belge açıldığında ne görülür?

1. 14.00'teki kayıt, belgenin o andaki hâlini SSD'ye yazmıştır.
2. 14.00–14.20 arasındaki değişiklikler yalnızca RAM'deki kopyaya işlenmiştir.
3. Güç kesilince RAM'in içeriği silinir; eklenen iki sayfa kaybolur.
4. Belge yeniden açıldığında SSD'deki son kayıt, yani 14.00'teki hâl, RAM'e okunur.

Sonuç: belge 14.00'teki hâliyle açılır. Bazı programların "otomatik kurtarma" özelliği belgenin kopyasını belirli aralıklarla diske yazar. Bu özellik açıksa son birkaç dakikanın kaybı azalır; ama bu, programın ayrıca yaptığı bir kayıttır, RAM'in kendisi bir şey korumaz.

Sık yapılan hata

"Bilgisayarımın belleği 512 GB" derken diski kastetmek. 512 GB çoğunlukla SSD'nin kapasitesidir; RAM bundan çok daha küçüktür (8, 16 ya da 32 GB gibi). Bellek (RAM) ile depolama (SSD, disk) ayrı donanımlardır ve farklı işler görür.

1.1.2 Sayısal temsil

Önceki kesimde giriş birimlerinin her fiziksel olayı sayılara çevirdiği söylendi. Bu kesim o sayıların bilgisayarın içinde nasıl tutulduğunu anlatır. Cevap şaşırtıcı derecede yalındır: bilgisayar yalnızca iki durumu ayırt eder. Bu iki duruma 0 ve 1 adı verilir. Bütün sayılar, harfler ve görüntüler, bu iki simgeden oluşan uzun dizilerle yazılır.

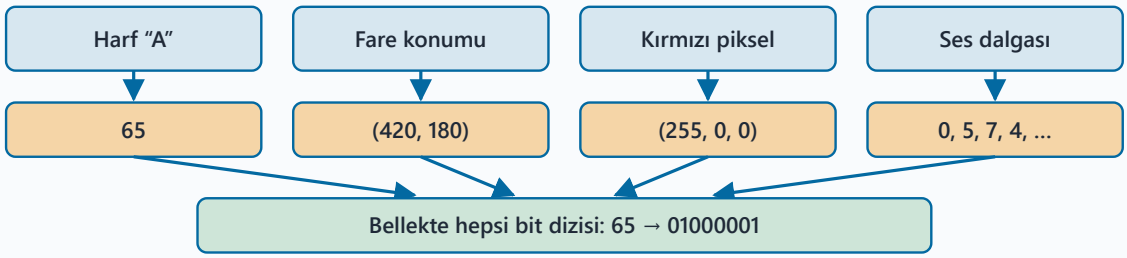
1.1.2.1 Her veri sayıya dönüşür

Bilgisayarın işlediği veri çok çeşitlidir: bir mesajdaki harfler, farenin ekrandaki konumu, bir fotoğrafın renkleri, bir şarkının sesi. İşlemci ise yalnızca sayılarla çalışır. Bu yüzden her veri türü için onu sayılara çeviren bir kural vardır. Kuralların ayrıntısı veri türüne göre değişir, ama düşünce hep aynıdır: veri-

yi parçalara ayır, her parçayı bir sayıyla yaz.

Veri türü	Dönüşüm kuralı	Örnek
Metin	Her karakterin bir kod numarası vardır.	A → 65 → 01000001
Fare hareketi	Ekrandaki konum iki koordinatla yazılır.	(x, y) → (420, 180)
Görüntü	Her piksel kırmızı, yeşil ve mavi bileşenlerle yazılır.	RGB(255, 0, 0): kırmızı
Ses	Ses dalgası belirli aralıklarla ölçülür.	genlik → sayı dizisi

Tablo 1.1.3: Farklı veri türlerinin sayıya dönüşmesi.



Şekil 1.1.6: Dört farklı veri, dört farklı kuralla sayıya dönüşür: "A" harfi 65'e, fare konumu (420, 180)'e, kırmızı bir piksel (255, 0, 0)'a, bir ses dalgası 0, 5, 7, 4, ... dizisine. Bellekte hepsi aynı biçimde, bit dizisi olarak durur.

Şekil 1.1.6 bu dört dönüşümü yan yana gösterir. En alttaki satır önemli bir sonucu vurgular: dönüşümden sonra bellekte bir harfi bir sesten ayıran hiçbir işaret kalmaz. Hepsi bit dizisidir. Bir bit dizisinin harf mi, sayı mı, renk mi olduğunu, onu okuyan program bilir. Bu düşünce Şekil 1.1.12'de yeniden ele alınır.

1.1.2.2 Bit, bayt ve basamak değeri

Tanım 1.1.5

Bit, İngilizce *binary digit* (ikili basamak) ifadesinin kısaltmasıdır ve bilginin en küçük birimidir. Bir bit yalnızca iki değerden birini alır: 0 ya da 1. Sekiz bitlik bir gruba **bayt** (*byte*) denir: 1 bayt = 8 bit.

Gündelik hayatta sayıları **onluk** (*decimal*) sistemle yazarız. Onluk sistemde on rakam vardır (0'dan 9'a) ve her basamağın değeri, sağındakinin on katıdır. 352 sayısında 2 birler basamağında, 5 onlar basamağında, 3 yüzler basamağındadır:

$$352 = 3 \times 100 + 5 \times 10 + 2 \times 1$$

İkili (*binary*) sistem aynı düşünceyi yalnızca iki rakamla, 0 ve 1 ile uygular. Her basamağın değeri sağındakinin *iki* katıdır. En sağdaki basamağın değeri 1'dir; sola doğru 2, 4, 8, 16, 32, 64, 128 diye ikiye katlanarak gider. Bu değerler 2'nin kuvvetleridir: $2^0 = 1$, $2^1 = 2$, $2^2 = 4$ ve böyle sürer. Bir basamağın konumu sağdan 0'dan başlayarak sayılır; k. konumdaki basamağın değeri 2^k'dır.

Tanım 1.1.6

Bir sayı sisteminde her konumun bir **basamak değeri** (*place value*) vardır. Onluk sistemde basamak değerleri 1, 10, 100, …, ikili sistemde 1, 2, 4, 8, … biçimindedir. Bir sayının değeri, her basamaktaki rakamın o basamağın değeriyle çarpılıp sonuçların toplanmasıyla bulunur. Hangi sistemde yazıldığı karışmasın diye sayının sağ altına sistemin tabanı yazılır: 101_2 ikili, 101_{10} onluk bir sayıdır.

101_2 ile 101_{10} çok farklı sayılardır. 101_{10} yüz birdir. 101_2 ise $1 \times 4 + 0 \times 2 + 1 \times 1 = 5$'tir. Taban yazılmadığında sayı onluk kabul edilir; ama ikili ve onluk sayıların bir arada geçtiği hesaplarda tabanı belirtmek karışıklığı önler.

Sık yapılan hata

Basamakları soldan saymak. İkili bir sayıda en küçük değerli basamak (1'ler basamağı) her zaman en *sağdadır*, tıpkı onluk sistemdeki birler basamağı gibi. 1000_2'deki 1, sağdan 3. konumdadır ve değeri $2^3 = 8$'dir; 1 değil.

1.1.2.3 İkiliden onluğa dönüşüm

İkili bir sayıyı onluğa çevirmek için her bitin altına basamak değeri yazılır, 1 olan bitlerin değerleri toplanır. 0 olan bitler toplama bir şey katmaz. Yöntemi, klavyedeki “A” harfinin bellekteki karşılığı olan 01000001 dizisi üzerinde izleyelim.

Örnek 1.1.4

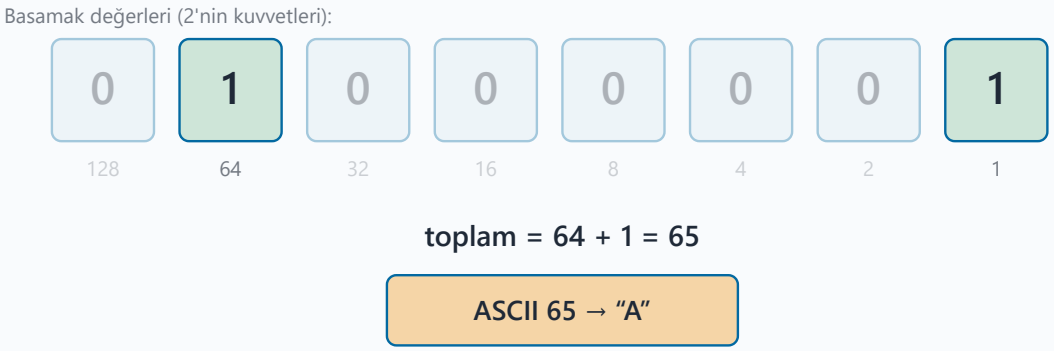
01000001_2 sayısını onluğa çevirelim.

1. **Basamak değerleri.** Sekiz bitin altına, sağdan sola 1, 2, 4, 8, 16, 32, 64, 128 değerleri yazılır.
2. **1 olan bitler.** Soldan ikinci bit (6. konum) 1'dir; değeri 64'tür. En sağdaki bit (0. konum) 1'dir; değeri 1'dir. Öteki altı bit 0'dır.
3. **Toplam.** $64 + 1 = 65$.

Sonuç: $01000001_2 = 65_{10}$. ASCII tablosunda 65 sayısı büyük “A” harfine karşılık gelir.

Konum	7	6	5	4	3	2	1	0
Basamak değeri	128	64	32	16	8	4	2	1
Bit	0	1	0	0	0	0	0	1
Katkı	0	64	0	0	0	0	0	1

Tablo 1.1.4: 01000001_2 için basamak değerleri ve toplama katkısı.



Şekil 1.1.7: 01000001 bit dizisi ve basamak değerleri. Yalnızca 64 ve 1 basamaklarındaki bitler 1'dir; toplam *64 + 1 = 65*'tir ve ASCII'de 65 "A" harfidir.

Şekil 1.1.7 hesabın görsel karşılığıdır. Basamak değerleri ezberlenirse ikiliden onluğa dönüşüm, birkaç sayının toplanmasından ibaret kalır. Sekiz bitin değerleri (128, 64, 32, 16, 8, 4, 2, 1) bilgisayar bilimiyle uğraşan herkesin zamanla ezberlediği bir dizidir.

Örnek 1.1.5

11001010_2 sayısı onluk sistemde kaçtır?

1. Bitler soldan sağa 1, 1, 0, 0, 1, 0, 1, 0; basamak değerleri 128, 64, 32, 16, 8, 4, 2, 1.
2. 1 olan bitlerin değerleri: 128, 64, 8, 2.
3. Toplam: *128 + 64 = 192*; *192 + 8 = 200*; *200 + 2 = 202*.

Sonuç: $11001010_2 = 202_{10}$. Bir denetim yolu: en sağdaki bit 0 olduğu için sayı çifttir; 202 de gerçekten çifttir. En sağdaki bit 1 olsaydı sonuç tek çıkmalıydı.

Sık yapılan hata

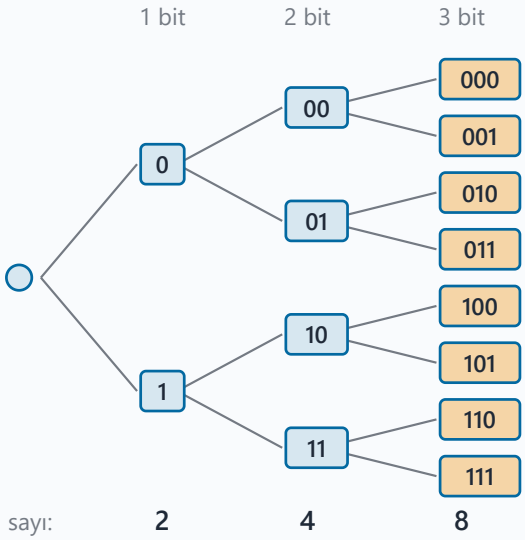
Baştaki sıfırları önemsiz sayıp basamak değerlerini kaydırmak. 01000001'deki baştaki 0, sayının değerini değiştirmez; 1000001_2 de 65'tir. Ama baştaki 0 atılınca sayı yedi bite iner ve en soldaki 1'in değeri 64 olarak kalır, 128 olmaz. Basamak değerleri her zaman *sağdan* sayılarak yazılırsa bu karışıklık olmaz.

1.1.2.4 n bit kaç farklı değer gösterir?

Tek bir bit iki farklı değer gösterir: 0 ve 1. İki bitle dört farklı dizi yazılabilir: 00, 01, 10, 11. Üç bitle sekiz dizi vardır: 000'dan 111'e. Her yeni bit, var olan her dizinin önüne bir kez 0, bir kez 1 konarak eklenir; bu yüzden dizi sayısı her adımda iki katına çıkar.

> **Teorem 1.1.1 (n bitin değer sayısı)**
>
> n bitle tam olarak 2^n farklı bit dizisi yazılabilir. Bu diziler işaretsiz tam sayı olarak okunursa 0'dan $2^n - 1$'e kadar olan sayıları gösterir.

Teoremin gerekçesi bir sayma işlemidir. Her bit için iki seçenek vardır ve bitler birbirinden bağımsız seçilir. İlk bit için 2, ikinci bit için yine 2 seçenek, toplamda $2 \times 2 = 4$ dizi eder. n bit için seçenekler n kez çarpılır: $2 \times 2 \times \ldots \times 2 = 2^n$. En küçük dizi bütün bitleri 0 olan dizidir ve 0'ı gösterir. En büyük dizi bütün bitleri 1 olandır ve değeri $2^n - 1$'dir. Bunun nedeni 0'ın da sayılmasıdır: 2^n değer 0'dan başlar ve $2^n - 1$'de biter.



Şekil 1.1.8: Her yeni bit kod sayısını ikiye katlar. 1 bitle 2 (0, 1), 2 bitle 4 (00–11), 3 bitle 8 (000–111) farklı kod yazılır.

Bit sayısı	Farklı değer	İşaretsiz değer aralığı
1 bit	$2^1 = 2$	0–1
2 bit	$2^2 = 4$	0–3
8 bit	$2^8 = 256$	0–255
10 bit	$2^{10} = 1.024$	0–1.023
16 bit	$2^{16} = 65.536$	0–65.535

Bit sayısı	Farklı değer	İşaretsiz değer aralığı
32 bit	2^{32} = 4.294.967.296	0–4.294.967.295

Tablo 1.1.5: Bit sayısı, farklı değer sayısı ve işaretsiz değer aralığı.

Tablo 1.1.5 bu büyümenin ne kadar hızlı olduğunu gösterir. 8 bit 256 değer verir; bu yüzden bir bayt 0 ile 255 arasında bir sayı tutar. 10 bit 1.024 değer verir ve bu sayı, ilerideki bellek birimlerinde (1 KB = 1.024 bayt) yeniden karşımıza çıkar. 32 bitle dört milyarı aşan sayıda farklı değer yazılabilir. Bir bit eklemek değer sayısını iki katına çıkarır; on bit eklemek ise yaklaşık bin katına çıkarır, çünkü $2^{10} = 1.024$.

Sık yapılan hata

"8 bit 256'ya kadar sayar" demek. 8 bit 256 *farklı* değer gösterir, ama bu değerler 0'dan başladığı için en büyüğü 255'tir. 256 sayısını yazmak için 9 bit gerekir: 100000000_2.

1.1.2.5 Kaç bit gerekir?

Önceki teoremin tersi de sık sorulur: N farklı şeye ayrı bir kod verilecekse en az kaç bit gerekir? Cevap, $2^n \geq N$ koşulunu sağlayan en küçük n'dir. Bu n, 1 bitten başlanıp kod sayısı N'ye ulaşana kadar bit eklenerek bulunur. Sunumdaki algoritma bu düşünceyi dört satırla yazar:

```
N = 58   (29 büyük + 29 küçük harf)
n = 1; kapasite = 2
kapasite < N iken: n = n + 1; kapasite = kapasite × 2
kapasite ≥ N → n bit yeter
```

Algoritmada `kapasite`, `n` bitle yazılabilecek kod sayısını tutar. Başlangıçta 1 bit ve 2 kod vardır. Kapasite `N`'den küçük kaldıkça bir bit eklenir ve kapasite iki katına çıkar. Kapasite `N`'ye ulaştığı ya da onu geçtiği anda döngü durur ve o anki `n` cevaptır.

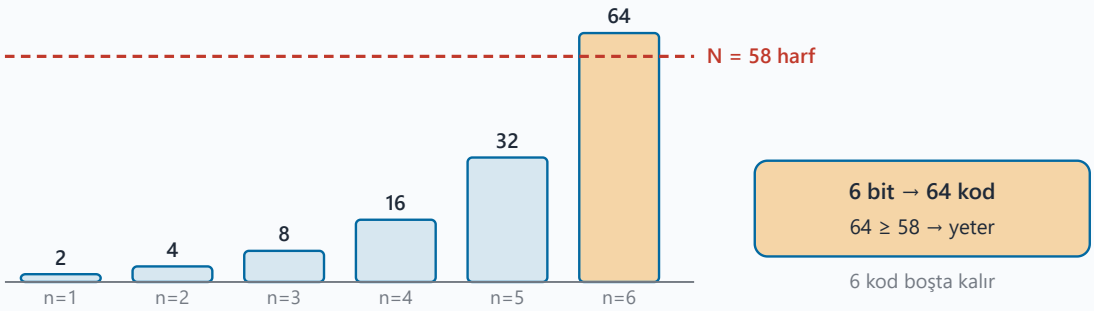
Örnek 1.1.6

Türk alfabesinde 29 harf vardır. Büyük ve küçük harfler ayrı kodlanırsa $N = 58$ farklı kod gerekir. En az kaç bit yeter?

n	kapasite = 2^n	kapasite < 58?	Karar
1	2	evet	yetmez, bit ekle
2	4	evet	yetmez, bit ekle
3	8	evet	yetmez, bit ekle
4	16	evet	yetmez, bit ekle

n	kapasite = 2^n	kapasite < 58?	Karar
5	32	evet	yetmez, bit ekle
6	64	hayır	yeter, döngü durur

Sonuç: 6 bit gerekir. 64 kodun 58'i kullanılır, *64 − 58 = 6* kod boşta kalır. 5 bit yetmez, çünkü 32 kodla 58 harfin ancak 32'si kodlanabilir.



Şekil 1.1.9: 58 harf için gereken bit sayısı. Çubuklar $n = 1, \ldots, 6$ bitle yazılabilen kod sayısını (2, 4, 8, 16, 32, 64) gösterir. Yalnızca 6 bitlik çubuk 58 çizgisini geçer; 6 kod boşta kalır.

Şekil 1.1.9 aynı sonucu çubuklarla gösterir. Çubukların her biri bir öncekinin iki katıdır ve 58 çizgisini ilk geçen çubuk 6 bitliktir. Gerçek bir metin yalnızca harflerden oluşmaz: on rakam, boşluk, nokta, virgül, soru işareti gibi noktalama işaretleri de kodlanmalıdır. Bunlar eklenince 64 kod yetmez ve 7 bit (128 kod) gerekir. İngilizce metinler için 1960'larda tanımlanan **ASCII** (*American Standard Code for Information Interchange*) kod tablosu bu yüzden 7 bitliktir. Tablonun 128 kodunun içine büyük ve küçük İngiliz harfleri, rakamlar, noktalama işaretleri ve bazı denetim karakterleri sığar.

Bu hesabın matematikteki karşılığı logaritmadır: gereken bit sayısı, $\log_2 N$ sayısının yukarı yuvarlanmışıdır. $\log_2 58 \approx 5{,}86$ olduğundan 6 bit gerekir. Logaritmayla rahat olmayan biri için 2'nin kuvvetlerini sırayla yazmak aynı sonucu verir ve hata yapma olasılığı daha düşüktür.

Sık yapılan hata

Kodlanacak değer sayısını en büyük değerle karıştırmak. 0'dan 100'e kadar sayılar kodlanacaksa değer sayısı 100 değil 101'dir. $2^6 = 64$ yetmez, $2^7 = 128 \geq 101$ olduğu için 7 bit gerekir. Aynı soru 0'dan 127'ye kadar sayılar için sorulsaydı da cevap 7 bit olurdu; 0'dan 128'e kadar sayılar için ise 8 bit gerekirdi.

1.1.2.6 Onluktan ikiliye dönüşüm

Ters yöndeki dönüşüm için en yaygın yöntem **2'ye bölme yöntemidir**. Sayı art arda 2'ye bölünür ve her bölmede kalan not edilir. Kalan her zaman 0 ya da 1'dir: sayı çiftse 0, tekse 1. Bölüm 0 olunca bölme biter. Kalanlar *son bölmeden ilk bölmeye doğru*, yani ters sırayla okununca ikili sayı elde edilir.

Yöntem şu algoritmayla yazılabilir:

```
sayı ← N
bitler ← ""
sayı > 0 olduğu sürece tekrarla:
    kalan ← sayı mod 2
    bitler ← kalan + bitler      (kalan sola eklenir)
    sayı ← sayı div 2
yaz(bitler)
```

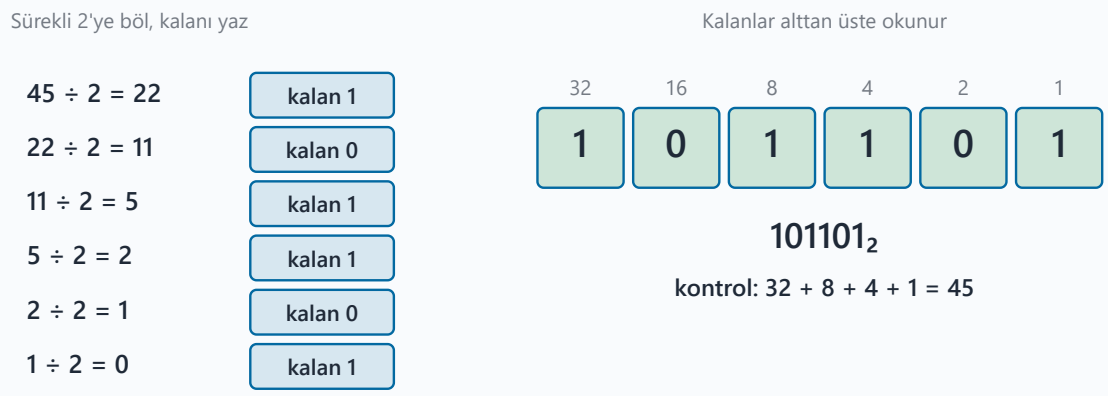
Algoritmada `mod` bölümden kalanı, `div` tam sayı bölümünü verir: *45 mod 2 = 1*, *45 div 2 = 22*. Her yeni kalan, şimdiye kadar biriken bitlerin *soluna* eklenir. İlk bölmenin kalanı böylece en sağda, son bölmenin kalanı en solda kalır.

Örnek 1.1.7

45 sayısını ikiliye çevirelim.

Adım	Bölme	Bölüm	Kalan	Biriken bitler
1	45 ÷ 2	22	1	1
2	22 ÷ 2	11	0	01
3	11 ÷ 2	5	1	101
4	5 ÷ 2	2	1	1101
5	2 ÷ 2	1	0	01101
6	1 ÷ 2	0	1	101101

Bölüm 6. adımda 0 olduğu için bölme biter. Sonuç: $45_{10} = 101101_2$. Denetim için basamak değerleri toplanır: 101101_2'de 1 olan basamaklar 32, 8, 4 ve 1'dir; *32 + 8 + 4 + 1 = 45*.



Şekil 1.1.10: 45'in 2'ye bölme yöntemiyle ikiliye çevrilmesi. Soldaki altı bölmenin kalanları, sağdaki basamak kutularına alttan üste yerleşir: ilk kalan 1'ler basamağına, son kalan 32'ler basamağına. Sonuç 101101_2'dir.

Yöntemin neden çalıştığı da basit bir gözlemle açıklanır. Bir sayının 2'ye bölümünden kalan, ikili yazılışının en sağdaki bitidir: çift sayıların son biti 0, tek sayılarınki 1'dir. Sayıyı 2'ye bölüp kalanı atmak ise ikili yazılıştaki en sağ biti silmek demektir. $45 = 101101_2$'dir. 2'ye bölününce $22 = 10110_2$ elde edilir; son bit gitmiştir. Bölme sürdükçe her adımda bir bit daha okunur ve bitler sağdan sola doğru tek tek ortaya çıkar.

Örnek 1.1.8

156 sayısını iki yolla ikiliye çevirelim.

1. yol: 2'ye bölme. 156 ÷ 2 = 78, kalan 0. 78 ÷ 2 = 39, kalan 0. 39 ÷ 2 = 19, kalan 1. 19 ÷ 2 = 9, kalan 1. 9 ÷ 2 = 4, kalan 1. 4 ÷ 2 = 2, kalan 0. 2 ÷ 2 = 1, kalan 0. 1 ÷ 2 = 0, kalan 1. Kalanlar sondan başa okununca 10011100_2 bulunur.

2. yol: büyük basamak değerlerinden çıkarma. 156'ya sığan en büyük 2 kuvveti 128'dir: *156 − 128 = 28*. 28'e sığan en büyük kuvvet 16'dır: *28 − 16 = 12*. Sonra 8: *12 − 8 = 4*. Sonra 4: *4 − 4 = 0*. Kullanılan değerler 128, 16, 8 ve 4'tür; bu basamaklara 1, ötekilere 0 yazılır: $1001\,1100_2$.

Sonuç: iki yol da $156_{10} = 10011100_2$ verir.

İkinci yol, basamak değerlerini iyi bilen biri için çoğunlukla daha hızlıdır. İlk yol ise hiç düşünmeden uygulanabilen mekanik bir tariftir ve bir programın da izleyebileceği biçimdedir. İki yolun sonucu her zaman aynıdır; biri ötekinin denetimi olarak kullanılabilir.

Sık yapılan hata

Kalanları ilk bölmeden son bölmeye doğru okumak. 45 için kalanlar sırasıyla 1, 0, 1, 1, 0, 1'dir ve bu sırayla yazılırsa 101101 çıkar. Bu yanlış okuma 45'te tesadüfen doğru sonuç verir, çünkü dizi tersten de aynıdır. 156 için kalanlar 0, 0, 1, 1, 1, 0, 0, 1'dir. Bu sırayla yazılan 00111001_2 = 57'dir ve yanlıştır. Doğru okuma sondan başadır: 10011100_2 = 156. Her dönüşümün sonunda basamak değerleriyle geri dönüp denetim yapmak bu hatayı hemen yakalar.

1.1.2.7 Onaltılık gösterim

Uzun bit dizilerini okumak ve yazmak insan için zordur. 32 bitlik bir sayıda tek bir biti yanlış yazmak çok kolaydır. Bu yüzden bit dizileri çoğunlukla **onaltılık** (*hexadecimal*) sistemde kısaltılarak yazılır. Onaltılık sistemde 16 rakam vardır: 0–9 ve onların ardından 10–15 değerleri için A, B, C, D, E, F harfleri.

Onaltılığın kullanışlı olmasının nedeni, $16 = 2^4$ olmasıdır. Dört bit tam olarak 16 farklı değer gösterir; yani her onaltılık rakam tam dört bite karşılık gelir. Bir bit dizisini onaltılığa çevirmek için hesap yapmak gerekmez: dizi sağdan başlayarak dörderli gruplara ayrılır ve her grup tek bir onaltılık rakamla yazılır.

İkili	Onluk	Onaltılık	İkili	Onluk	Onaltılık
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

Tablo 1.1.6: Dört bitlik grupların onluk ve onaltılık karşılıkları.

Örnek 1.1.9

1. **45 → onaltılık.** 45 = 101101_2'dir. Sekiz bite tamamlanır: 0010 1101. 0010 = 2, 1101 = D. Sonuç $2D_{16}$. Denetim: $2 \times 16 + 13 = 45$.
2. **255 → onaltılık.** 255 = 1111 1111_2; iki grup da 1111 = F'dir. Sonuç FF_{16}. Bir baytın alabileceği en büyük değer iki onaltılık rakamla FF diye yazılır.
3. **Bir rengin kodu.** Kırmızı = 59, yeşil = 130, mavi = 246 olan rengin bileşenleri onaltılığa çevrilir. 59 = 0011 1011_2 = $3B_{16}$. 130 = 1000 0010_2 = 82_{16}. 246 = 1111 0110_2 = $F6_{16}$. Bileşenler yan yana yazılınca renk kodu 3B82F6 olur. Web sayfalarında renkler bu altı ha-

neli onaltılık kodlarla yazılır.

Onaltılık gösterim yeni bir veri türü değildir; aynı bit dizisinin daha kısa bir yazılışıdır. Bellek adresleri, renk kodları ve Unicode karakter numaraları (örneğin "A" için U+0041) bu yüzden onaltılık yazılır. Aynı mantıkla üç bitlik gruplar da tek rakamla yazılabilir; buna **sekizli** (*octal*) gösterim denir: 101 $101_2 = 55_8$. Sayı tabanı dönüştürücü simülatöründe aynı sayının dört tabandaki yazılışları yan yana görülebilir.

Sık yapılan hata

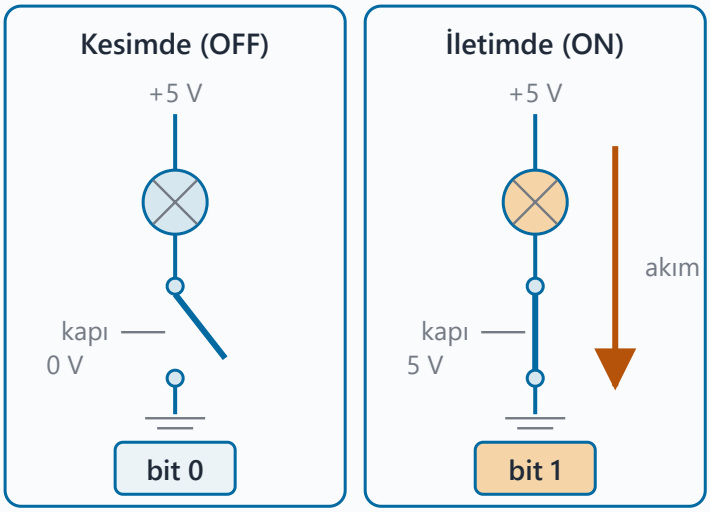
Bitleri dörderli gruplara *soldan* başlayarak ayırmak. 101101_2 soldan gruplanırsa 1011 01 olur ve yanlış sonuç çıkar. Gruplama her zaman sağdan yapılır; soldaki eksik grup başına 0 eklenerek tamamlanır: 10 1101 → 0010 1101 → 2D.

1.1.2.8 Transistörler

Bilgisayarın neden ikili sistemi kullandığı sorusunun cevabı donanımdadır. İşlemciler ve bellekler, **transistör** (*transistor*) adı verilen çok küçük elektronik parçalardan yapılır. Transistör, elektrik akımını denetleyen bir anahtar gibi çalışır. Bu anahtarı açan ya da kapatan mekanik bir kol değil, bir elektrik gerilimidir.

Tanım 1.1.7

Transistör, bir ucuna (kapı) uygulanan gerilime göre iki ucu arasından akım geçirip geçirmediği değişen yarı iletken bir devre elemanıdır. Kapıya yeterli gerilim uygulanınca transistör **iletime** geçer ve akım akar; gerilim yoksa **kesime** girer ve akım akmaz. Bilgisayarlarda bu iki durum 1 ve 0 olarak yorumlanır.



Şekil 1.1.11: Transistör bir anahtar gibi çalışır. Solda kapı gerilimi 0 V'tur: transistör kesimdedir, akım akmaz, lamba sönüktür; bu durum bit 0'dır. Sağda kapı gerilimi 5 V'tur: transistör iletimdedir, akım akar, lamba yanar; bu durum bit 1'dir.

Şekil 1.1.11 bu iki durumu bir lamba devresiyle gösterir. Bilgisayarın neden onluk değil ikili çalıştığı da buradan anlaşılır. Bir anahtarın yalnızca iki durumunu, açık ve kapalıyı, güvenilir biçimde ayırt etmek kolaydır. On farklı gerilim düzeyini birbirinden ayırmak ise çok daha zordur; küçük bir elektrik gürültüsü 6'yı 7'ye çevirebilir. İki durumlu sistemde gerilim biraz oynasa da "yüksek" ile "düşük" arasındaki fark büyük olduğu için okuma bozulmaz.

Tek bir transistör tek bir bit tutar ya da tek bir bit üzerinde çok basit bir iş yapar. Gücün kaynağı sayıdır. Transistörler birleştirilerek mantık kapıları, kapılar birleştirilerek toplayıcılar ve bellek hücreleri, onlar da birleştirilerek işlemciler kurulur. Günümüzün bir telefon işlemcisinde on milyarlarca transistör bulunur. Her biri saniyede milyarlarca kez açılıp kapanabilir. Milyarlarca basit anahtarın düzenli birlikteliği, video oynatmak ya da bir yapay zekâ modelini çalıştırmak gibi karmaşık işleri mümkün kılar.

Sık yapılan hata

0'ı "hiçbir şey" ya da "bilgi yok" sanmak. 0, 1 kadar anlamlı bir değerdir: transistörün kesimde olduğunu, yani belirli bir durumu bildirir. 01000001'deki altı sıfır, sayının 65 olması için en az iki 1 kadar gereklidir.

1.1.3 Veri temsili

Bit ve bayt, bilgisayarın bütün verileri yazdığı ortak alfabedir. Bu kesim, farklı veri türlerinin bu alfabeyle nasıl yazıldığını ve ne kadar yer kapladığını ele alır: sayılar ve metin, görüntüler, ses ve video. Ardından dosyaları küçültmenin iki yolu, kayıpsız ve kayıplı sıkıştırma anlatılır. Kesim, veri miktarını

ölçen birimlerle ve hız ile boyut arasındaki bit–bayt farkıyla biter.

1.1.3.1 Sayılar ve metin

Tam sayılar doğrudan ikili karşılıklarıyla saklanır. 13 sayısı bir baytta 00001101 olarak durur: *8 + 4 + 1 = 13*. Eksi sayılar ve ondalıklı sayılar için ayrı ikili kodlama kuralları vardır. Örneğin 3,75 gibi ondalıklı bir sayı üç parçaya bölünerek saklanır: işaret, üs ve anlamlı basamaklar. Bu biçime **kayan noktalı sayı** (*floating-point number*) denir. Bu kuralların ayrıntısı bu notun kapsamı dışındadır. Bilinmesi gereken, her sayı türü için önceden kararlaştırılmış bir kuralın bulunduğudur.

Karakterler, yani harfler, rakamlar ve işaretler, bir kod tablosuyla sayıya çevrilir. Tabloda her karaktere bir numara verilmiştir. Bilgisayar bir harfi saklarken aslında o harfin numarasını saklar; ekrana yazarken de numaraya karşılık gelen harf şeklini çizer.

Tanım 1.1.8

Karakter kümesi (*character set*), karakterlerle sayısal kodları eşleştiren tablodur. **ASCII**, 128 karakterlik 7 bitlik bir tablodur ve İngiliz alfabesini, rakamları ve noktalama işaretlerini kapsar. **Unicode**, dünyadaki hemen bütün yazı sistemlerini, matematik simgelerini ve emojileri kapsayan, bir milyondan fazla kod yeri tanımlayan evrensel karakter standardıdır. Unicode'un ilk 128 kodu ASCII ile aynıdır.

ASCII tablosunun düzeni bazı küçük hesapları kolaylaştırır. Büyük harfler 65'ten (A) başlar ve alfabe sırasıyla ardışıktır: B = 66, C = 67, …, Z = 90. Küçük harfler 97'den (a) başlar. Her küçük harfin kodu, büyüğünün kodundan tam 32 fazladır: a = 97 = 65 + 32. Rakam karakterleri 48'den başlar: “0” = 48, “1” = 49, …, “9” = 57. Burada önemli bir ayrım vardır. “7” *karakterinin* kodu 55'tir; 7 *sayısı* ise ikili olarak 00000111 diye saklanır. Klavyeden yazılan “7”, sayı olarak kullanılmadan önce dönüştürülmelidir.

Örnek 1.1.10

“Kedi” kelimesi ASCII ile kaç bayt tutar ve bellekte hangi bitlerle saklanır?

Karakter	Kod (onluk)	Bitler	Hesap
K	75	01001011	64 + 8 + 2 + 1
e	101	01100101	64 + 32 + 4 + 1
d	100	01100100	64 + 32 + 4
i	105	01101001	64 + 32 + 8 + 1

Sonuç: dört karakter, her biri bir bayt, toplam 4 bayt (32 bit). K'nin kodu A'nın kodundan 10 fazladır, çünkü K alfabede A'dan 10 harf sonra gelir. Küçük e'nin kodu ise büyük E'nin (69) kodundan 32 fazladır: *69 + 32 = 101*.

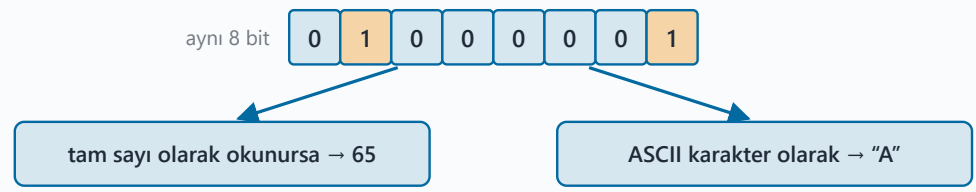
ASCII yalnızca İngiliz alfabesini kapsar; “ğ”, “ş”, “ı” gibi Türkçe harfler ASCII'de yoktur. Unicode bu sorunu bütün diller için çözer. Unicode'da her karakterin bir **kod noktası** (*code point*) vardır. Kod noktası “U+” ile başlayan onaltılık bir sayıyla yazılır: “A” = U+0041 (onluk 65), “Ğ” = U+011E (onluk 286), “€” = U+20AC, Çince “中” = U+4E2D. Kod noktalarının bellekte kaç baytla saklanacağını ise ayrı bir **kodlama** (*encoding*) kuralı belirler. Web'de en yaygın kodlama UTF-8'dir. UTF-8, ASCII karakterlerini tek baytla, Türkçe harfleri iki baytla, “€” ve “中” gibi karakterleri üç baytla yazar.

Karakter	Kod noktası	Onluk	UTF-8 baytları (onaltılık)	Bayt sayısı
A	U+0041	65	41	1
Ğ	U+011E	286	C4 9E	2
€	U+20AC	8.364	E2 82 AC	3
中	U+4E2D	20.013	E4 B8 AD	3

Tablo 1.1.7: Bazı karakterlerin Unicode kod noktaları ve UTF-8'deki bayt sayıları.

Bu ayrımın pratik bir sonucu vardır. “Kedi” UTF-8'de de 4 bayttır. “Kağıt” ise beş karakter olduğu hâlde 7 bayttır: “K”, “a” ve “t” birer, “ğ” ve “ı” ikişer bayt tutar. Karakter sayısı ile bayt sayısı, ASCII dışı harfler kullanıldığında farklılaşır.

Kodlama düşüncesinin en önemli sonucu şudur: bir bit dizisinin anlamını, dizinin kendisi değil, onu okurken kullanılan kural belirler.



Şekil 1.1.12: Aynı sekiz bit, 01000001, iki farklı kuralla okunur. Tam sayı olarak okunursa 65, ASCII karakteri olarak okunursa “A” anlamına gelir.

Şekil 1.1.12 bu durumu tek bir bayt üzerinde gösterir. Bellekteki 01000001 bitleri bir hesap programına göre 65 sayısıdır, bir metin düzenleyiciye göre “A” harfidir, bir resim programına göre bir pikselin koyu gri tonu olabilir. Bir dosya yanlış kodlamayla açıldığında Türkçe harflerin yerine “Ä±” ya da “ÅŸ” gibi anlamsız işaretler görülmesinin nedeni budur: baytlar doğrudur, ama yanlış kuralla okunmuştur.

Sık yapılan hata

“Bir karakter her zaman bir bayttır” demek. Bu yalnızca ASCII için doğrudur. UTF-8'de bir karakter 1 ile 4 bayt arasında yer tutar; Türkçe metinler bu yüzden karakter sayılarından biraz daha fazla bayt kaplar.

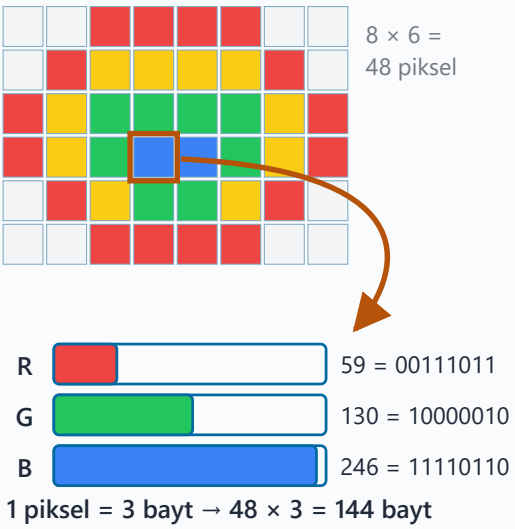
1.1.3.2 Görüntüler

Ekranda görülen bir fotoğraf, yakından bakılınca küçük renkli karelerden oluşur. Her kareye **piksel** (*pixel*, "picture element" yani resim öğesi kısaltması) denir. Bir piksel, görüntünün tek renkle boyanmış en küçük parçasıdır. Görüntü bir piksel tablosu olarak saklanır: her satırda belirli sayıda piksel, her pikselde bir renk.

Tanım 1.1.9

Çözünürlük (*resolution*), bir görüntünün yatay ve dikey piksel sayısıdır ve "genişlik × yükseklik" diye yazılır: 1920 × 1080 gibi. **RGB** renk modelinde her pikselin rengi kırmızı (*red*), yeşil (*green*) ve mavi (*blue*) üç bileşenin karışımıyla tanımlanır. Bileşenlerin her biri 0 ile 255 arasında bir sayıdır ve bir bayt yer tutar.

RGB modeli, ekranın ışığı nasıl ürettiğine dayanır. Ekrandaki her piksel aslında yan yana duran üç küçük ışık kaynağıdır: bir kırmızı, bir yeşil, bir mavi. Bileşenin değeri o ışığın parlaklığını belirler. 0 ışığın tamamen kapalı, 255 tam parlak olduğunu gösterir. Üç ışık da tam parlak yanınca göz beyaz görür: RGB(255, 255, 255). Üçü de kapalıysa siyah görülür: RGB(0, 0, 0). Yalnız kırmızı tam parlaksa renk saf kırmızıdır: RGB(255, 0, 0). Kırmızı ve yeşil birlikte yanınca sarı elde edilir: RGB(255, 255, 0). Üç bileşen eşitse renk bir gri tonudur; değer büyüdükçe gri açılır.



Şekil 1.1.13: 8 × 6 = 48 piksellik bir görüntü. Seçili mavi pikselin bileşenleri R = 59 (00111011), G = 130 (10000010), B = 246 (11110110)'dır. Her piksel 3 bayt tuttuğu için görüntü *48 × 3 = 144* bayttır.

Şekil 1.1.13 çok küçük bir görüntü üzerinde bu düzeni gösterir. Görüntünün 8 sütunu ve 6 satırı vardır; toplam 48 piksel eder. Seçili mavi pikselde mavi bileşen en büyük değeri (246) taşır, kırmızı en küçüğü (59). Her bileşen bir bayt, yani sekiz bittir. Bir piksel üç bayt ettiğine göre bütün görüntü *48 × 3 = 144* bayt yer tutar. Gerçek bir fotoğrafta da hesap aynıdır; yalnızca piksel sayısı milyonlarca olur.

Sık yapılan hata

RGB'yi boya karıştırmaya benzetmek. Boyalar karıştıkça renk koyulaşır; kırmızı, yeşil ve mavi boya karışımı kahverengimsi bir koyu renk verir. Ekranda ise ışık karışır ve ışık eklendikçe renk açılır; üç ışığın tam karışımı beyazdır.

1.1.3.3 Renk derinliği ve görüntü boyutu

Bir pikselin rengini yazmak için kaç bit kullanıldığına **renk derinliği** ya da **bit derinliği** (*color depth, bit depth*) denir. Bit derinliği, gösterilebilecek farklı renk sayısını belirler. Teorem 1.1.1'e göre b bitlik bir piksel 2^b farklı renk alabilir.

Bit / piksel	Farklı renk	Tipik kullanım
1	$2^1 = 2$	Siyah-beyaz çizim, faks, belge tarama
8	$2^8 = 256$	Gri tonlama ya da 256 renkli paletli görüntü (GIF)
24	$2^{24} = 16.777.216$	Fotoğraf: R, G ve B için 8'er bit
32	16,7 milyon renk + saydamlık	Arayüz grafikleri, PNG (alfa kanalı)

Tablo 1.1.8: Bit derinliği, farklı renk sayısı ve tipik kullanım.

24 bit, insan gözünün ayırt edebileceğinden fazla renk verir; fotoğraflar için standart derinlik budur. 32 bitte renk yine 24 bittir; kalan 8 bit **alfa kanalı** denen saydamlık bilgisini tutar. Alfa 0 ise piksel tamamen saydam, 255 ise tamamen örtücüdür. Bir uygulama simgesinin köşelerinin arka plan rengini göstermesi bu kanal sayesindedir.

Bit derinliği ve çözünürlük birlikte, sıkıştırılmamış bir görüntünün boyutunu belirler:

> boyut (bayt) = genişlik × yükseklik × bit derinliği ÷ 8

Formüldeki ÷ 8, bit cinsinden toplamı bayta çevirir. Piksel başına bayt sayısı tam sayıysa (24 bit = 3 bayt gibi) hesap daha da kısalır: boyut = piksel sayısı × piksel başına bayt.

Örnek 1.1.11

800 × 600 piksellik bir görüntünün sıkıştırılmamış boyutunu üç farklı bit derinliği için hesaplayalım. (1 KB = 1.024 bayt, 1 MB = 1.048.576 bayt alınmıştır.)

1. **Piksel sayısı:** *800 × 600 = 480.000* piksel.
2. **24 bit (3 bayt/piksel):** *480.000 × 3 = 1.440.000* bayt. MB'ye çevirmek için 1.048.576'ya bölünür: ≈ 1,37 MB.
3. **8 bit (1 bayt/piksel):** *480.000 × 1 = 480.000* bayt; *480.000 ÷ 1.024 ≈ 468,75* KB.
4. **1 bit:** *480.000 ÷ 8 = 60.000* bayt; *60.000 ÷ 1.024 ≈ 58,6* KB.

Sonuç: aynı görüntü 24 bitle 1,37 MB, 8 bitle yaklaşık 469 KB, 1 bitle yaklaşık 58,6 KB tutar. 24 bitlik sürüm 1 bitlik sürümün tam 24 katıdır, çünkü piksel sayısı aynıdır ve yalnızca piksel başına bit sayısı değişmiştir.

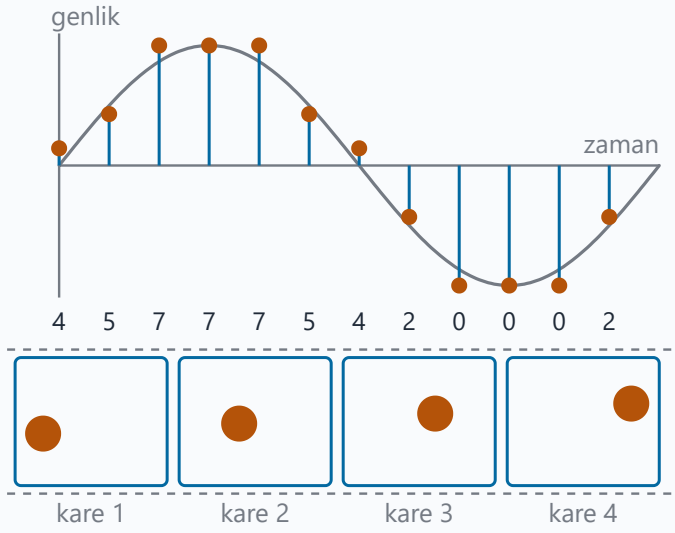
Renk derinliği arttıkça renk geçişleri yumuşar, ama dosya büyür. 8 bitlik paletli bir görüntüde yalnızca 256 renk kullanılabildiği için gökyüzü gibi tonların yavaşça değiştiği bölgeler basamak basamak görünür. 24 bitte aynı geçiş göze kesintisiz gelir. Çözünürlüğün etkisi ise daha da büyüktür: genişlik ve yükseklik ikişer katına çıkarsa piksel sayısı, dolayısıyla boyut, dört katına çıkar.

Sık yapılan hata

Bit derinliğini bayt sanıp 8'e bölmeyi unutmak. 800 × 600 × 24 = 11.520.000 bulunur ve "11,5 milyon bayt" denirse sonuç sekiz kat büyük çıkar. 11.520.000 bit cinsindendir; bayta çevirmek için 8'e bölünür: 1.440.000 bayt.

1.1.3.4 Ses ve video

Ses, havadaki basınç dalgalanmalarıdır. Mikrofon bu dalgalanmayı, zamanla sürekli değişen bir elektrik gerilimine çevirir. Bilgisayar sürekli bir dalgayı olduğu gibi saklayamaz; sonsuz sayıda noktası vardır. Bunun yerine dalga eşit aralıklarla ölçülür. Bu işleme **örnekleme** (*sampling*), her ölçüme bir **örnek** (*sample*) denir. Her ölçüm de belirli düzeylerden en yakınına yuvarlanarak bir tam sayıya çevrilir.



Şekil 1.1.14: Üstte bir ses dalgası 12 noktada örneklenmiş; her örnek 0 ile 7 arasında bir sayıya yuvarlanmıştır: 4, 5, 7, 7, 7, 5, 4, 2, 0, 0, 0, 2. Altta bir videonun dört ardışık karesi; top her karede biraz ilerler.

Şekil 1.1.14'ün üst kısmında dalga, eşit aralıklı 12 noktada ölçülmüştür. Dalganın tepesine yakın örnekler 7, çukuruna yakın örnekler 0 değerini almıştır. Bilgisayar dalganın kendisini değil, bu 12 sayıyı saklar. Ses yeniden çalınırken sayılar sırayla gerilime çevrilir ve hoparlör bu gerilimle titreşir.

Video, art arda gösterilen durağan görüntülerden oluşur. Her görüntüye **kare** (*frame*) denir. Kareler yeterince hızlı gösterilince göz onları ayrı ayrı göremez ve hareket algılar. Saniyede gösterilen kare sayısına **fps** (*frames per second*) denir. Sinema geleneksel olarak 24 fps, televizyon ve internet videoları çoğunlukla 30 fps kullanır; oyunlarda 60 fps ve üstü yaygındır. Videonun sesi, görüntü karelerinden ayrı bir kanalda saklanır ve oynatılırken karelerle eş zamanlı çalınır.

Örnek 1.1.12

1920 × 1080 çözünürlükte, 24 bit renkli ve 30 fps olan bir video sıkıştırılmadan saklansaydı bir dakikası ne kadar yer tutardı? (Ses yok sayılır; 1 GiB = 2^{30} = 1.073.741.824 bayt.)

1. **Bir kare:** *1920 × 1080 = 2.073.600* piksel; × 3 bayt = 6.220.800 bayt.
2. **Bir saniye:** 30 kare × 6.220.800 = 186.624.000 bayt, yani saniyede yaklaşık 178 MB.
3. **Bir dakika:** 60 saniye × 186.624.000 = 11.197.440.000 bayt.
4. **GiB'ye çevirme:** 11.197.440.000 ÷ 1.073.741.824 ≈ 10,43 GiB.

Sonuç: sıkıştırmasız bir dakikalık Full HD video yaklaşık 10,4 GiB tutar. Bir dakikası 100 MB civarında olan sıradan bir video dosyası bundan yüz kat küçüktür; fark sıkıştırmadan gelir.

1.1.3.5 Sesi sayısallaştırma

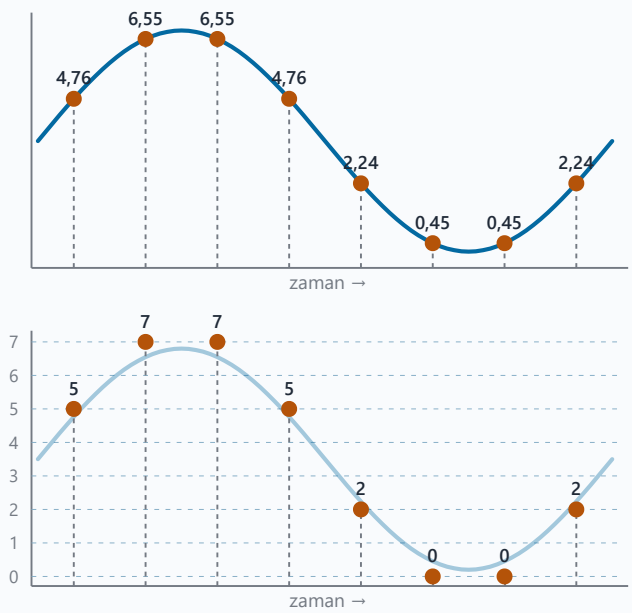
Analog bir ses sinyali üç adımda sayısal veriye dönüşür: örnekleme, nicemleme ve kodlama. Sunumdaki algoritma bu adımları ve boyut formülünü şöyle sıralar:

```
analog sinyal: sürekli dalga
örnekle: eşit aralıklarla ölç
nicemle: en yakın düzeye yuvarla (3 bit → 0…7)
kodla: her düzeyi 3 bitle yaz
boyut = örnekleme hızı × bit derinliği × kanal × süre
kayıplı sıkıştır (MP3)
```

Tanım 1.1.10

Örnekleme, sinyalin eşit zaman aralıklarıyla ölçülmesidir; saniyedeki ölçüm sayısına **örnekleme hızı** (*sampling rate*) denir ve hertz (Hz) ile ölçülür. **Nicemleme** (*quantization*), her ölçümün önceden belirlenmiş sonlu sayıdaki düzeyden en yakınına yuvarlanmasıdır; ölçüm ile düzey arasındaki farka **nicemleme hatası** denir. **Kodlama**, her düzeyin numarasının ikili sayıyla yazılmasıdır. Bir örneği yazmak için kullanılan bit sayısına bit derinliği denir.

Adımları sunumdaki küçük örnek üzerinde izleyelim. Sinyal eşit aralıklarla sekiz kez ölçülmüştür. Ölçümler ondalıklıdır: 4,76; 6,55; 6,55; 4,76; 2,24; 0,45; 0,45; 2,24. İki ölçüm arasında sinyalin nasıl değiştiği kaydedilmez; örnekleme bu bilgiyi kaçınılmaz olarak kaybeder.



Şekil 1.1.15: Sesin sayısallaştırılmasının ilk iki adımı. Üstte örnekleme: sinyal eşit aralıklarla sekiz kez ölçülür (4,76; 6,55; …). Altta nicemleme: 3 bit yalnızca 8 düzey (0…7) tanır ve her ölçüm en yakın düzeye yuvarlanır; ölçümler 5, 7, 7, 5, 2, 0, 0, 2 olur.

Nicemleme için 3 bit kullanılsın. Teorem 1.1.1'e göre 3 bit $2^3 = 8$ düzey tanır: 0, 1, …, 7. Her ölçüm en yakın tam sayıya yuvarlanır (Şekil 1.1.15, alt).

Örnek	Ölçüm	Düzey	Nicemleme hatası	3 bitlik kod
1	4,76	5	0,24	101
2	6,55	7	0,45	111
3	6,55	7	0,45	111
4	4,76	5	0,24	101
5	2,24	2	0,24	010
6	0,45	0	0,45	000
7	0,45	0	0,45	000
8	2,24	2	0,24	010

Tablo 1.1.9: Sekiz ölçümün nicemlenmesi ve 3 bitle kodlanması.

Tablo 1.1.9 üç adımın sonucunu birlikte verir. Nicemleme hatası en çok 0,45 düzeydir. En yakın düzeye yuvarlamada hata hiçbir zaman yarım düzeyi geçemez, çünkü her ölçüm iki düzeyin arasında durur ve yakın olana en çok 0,5 uzaktadır. Kodlama adımında her düzey 3 bitlik ikili sayıyla yazılır: 5 → 101, 7 → 111, 2 → 010, 0 → 000. Sekiz örnek *8 × 3 = 24* bit tutar.

Bu küçük örnek, gerçek seste iki noktada büyür. Birincisi, örnekleme hızıdır. Saniyede sekiz ölçüm insan sesini anlaşılır kılmaya yetmez; gerçek ses kayıtları saniyede binlerce, CD kalitesinde 44.100 örnek alır. İkincisi, bit derinliğidir. 3 bitlik 8 düzey kaba bir merdiven gibidir; CD 16 bit kullanır ve $2^{16} = 65.536$ düzey tanır. Bu kadar ince bir merdivende nicemleme hatası kulağın duyamayacağı kadar küçülür.

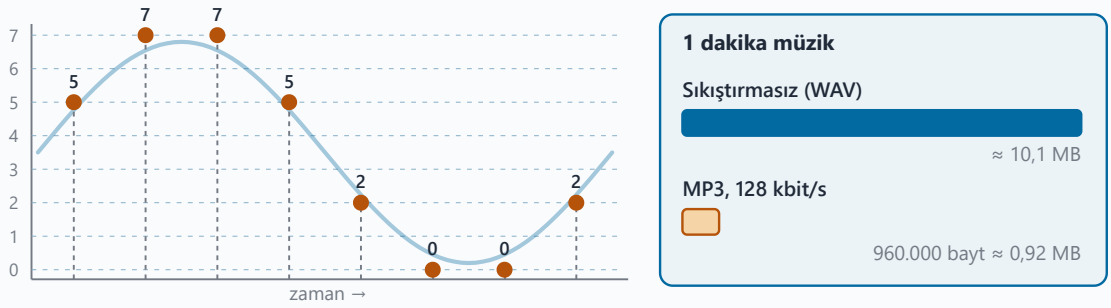
> boyut (bit) = örnekleme hızı × bit derinliği × kanal sayısı × süre (s)

Örnek 1.1.13

CD kalitesinde (44.100 Hz, 16 bit, 2 kanal: sol ve sağ) bir dakikalık ses kaç bayt tutar? Aynı müzik 128 kbit/s MP3 olarak kaydedilirse ne kadar yer kaplar?

1. **Saniyedeki bit sayısı:** *44.100 × 16 × 2 = 1.411.200* bit/s.
2. **Bir dakika:** *1.411.200 × 60 = 84.672.000* bit.
3. **Bayta çevirme:** *84.672.000 ÷ 8 = 10.584.000* bayt. 1 MB = 1.048.576 bayt alınırsa ≈ 10,1 MB.
4. **MP3:** 128 kbit/s = 128.000 bit/s. Bir dakikada *128.000 × 60 = 7.680.000* bit, yani *7.680.000 ÷ 8 = 960.000* bayt ≈ 0,92 MB.
5. **Oran:** *10.584.000 ÷ 960.000 ≈ 11*.

Sonuç: CD kalitesinde bir dakika ≈ 10,1 MB, aynı dakikanın 128 kbit/s MP3'ü ≈ 0,92 MB'dir; MP3 yaklaşık 11 kat küçüktür.



Şekil 1.1.16: Bir dakikalık müziğin sıkıştırmasız (WAV, ≈ 10,1 MB) ve MP3 (128 kbit/s, 960.000 bayt ≈ 0,92 MB) boyutları.

MP3 bu küçülmeyi kayıplı sıkıştırmayla sağlar: kulağın zor fark ettiği ayrıntıları, örneğin yüksek bir sesin hemen ardından gelen çok kısık sesleri, atar. Atılan ayrıntı bir daha geri gelmez.

Sık yapılan hata

Kanal sayısını unutmak. Stereo (iki kanallı) bir kayıtta sol ve sağ hoparlör için ayrı örnekler saklanır; boyut mono kaydın iki katıdır. CD hesabında 2 çarpanı atlanırsa sonuç ≈ 5 MB çıkar ve yanlıştır.

1.1.3.6 Kayıpsız ve kayıplı sıkıştırma

Sıkıştırmasız görüntü, ses ve video çok yer kaplar. Bu yüzden dosyalar çoğunlukla **sıkıştırılarak** (*compression*) saklanır. Sıkıştırma, aynı veriyi daha az bitle yazmanın yoludur. İki temel türü vardır ve aralarındaki fark, açılan dosyanın özgün dosyayla aynı olup olmadığıdır.

Tanım 1.1.11

Kayıpsız sıkıştırma (*lossless compression*), açıldığında özgün veriyi bit bit aynen geri veren sıkıştırmadır; verideki tekrarları kısaltarak çalışır. **Kayıplı sıkıştırma** (*lossy compression*), insanın göz ve kulağının zor fark ettiği ayrıntıları kalıcı olarak atar; açıldığında özgüne yakın ama ondan farklı bir kopya verir.

	Kayıpsız	Kayıplı
Açılınca	Özgün veri bit bit aynen geri gelir	Özgüne yakın bir kopya elde edilir
Yöntem	Tekrarları kısaltır (RLE, ZIP)	Göz ve kulağın zor fark ettiği ayrıntıyı atar
Örnek biçimler	ZIP, PNG, FLAC	JPEG, MP3, MP4 (H.264)
Tipik küçülme	Çoğunlukla 2–3 kat	10 kat ve daha fazlası
Uygun veri	Metin, program, tablo, tıbbi görüntü	Fotoğraf, müzik, video

Tablo 1.1.10: Kayıpsız ve kayıplı sıkıştırmanın karşılaştırması.

Hangi türün seçileceğini verinin doğası belirler. Bir programın tek bir biti bile değişirse program çalışmaz ya da yanlış çalışır. Bir metin dosyasında tek bir bitin değişmesi bir harfi başka bir harfe çevirir. Bir banka tablosunda 1.000 TL'nin 1.008 TL'ye dönüşmesi kabul edilemez. Bu veriler yalnızca kayıpsız sıkıştırılır. Tıbbi görüntülerde de küçük bir ayrıntı tanı için önemli olabileceğinden kayıpsız biçimler tercih edilir. Fotoğraf, müzik ve videoda ise birkaç pikselin rengindeki ya da çok kısık bir sesteki küçük değişiklik fark edilmez. Bu veriler kayıplı sıkıştırılarak on kat ve daha fazla küçültülür.

Kayıplı sıkıştırmanın gizli bir maliyeti vardır. Kayıplı bir dosya açılıp düzenlenip yeniden kaydedildiğinde sıkıştırma yeniden yapılır ve her seferinde biraz daha ayrıntı atılır. Aynı JPEG fotoğrafın defalarca düzenlenip kaydedilmesi, görüntüde giderek artan bulanıklık ve kare kare bozulmalar yaratır.

Atılan ayrıntı geri gelmez. Bu yüzden üzerinde çalışılan özgün dosyalar kayıpsız biçimde saklanır ve kayıplı sürüm yalnızca paylaşım için üretilir.

Sık yapılan hata

Bir JPEG dosyasını PNG olarak kaydetmenin kaybolan kaliteyi geri getireceğini sanmak. PNG kayıpsızdır, ama yalnızca kendisine verilen veriyi korur. JPEG sıkıştırmasında atılmış ayrıntı artık hiçbir yerde yoktur; PNG ancak bozulmuş hâli bozulmadan saklar.

1.1.3.7 Çalışma uzunluğu kodlaması (RLE)

Kayıpsız sıkıştırmanın en basit örneği **çalışma uzunluğu kodlamasıdır** (*run-length encoding*, RLE). Düşünce şudur: aynı değer art arda birçok kez tekrarlanıyorsa, değeri tek tek yazmak yerine "kaç kez tekrarlandığı" ile "değerin kendisi" yazılır. Art arda gelen aynı değerlerden oluşan bloğa **koşu** (*run*) denir. Örneğin "AAAAABBB" dizisi (5, A)(3, B) diye yazılır.

Tanım 1.1.12

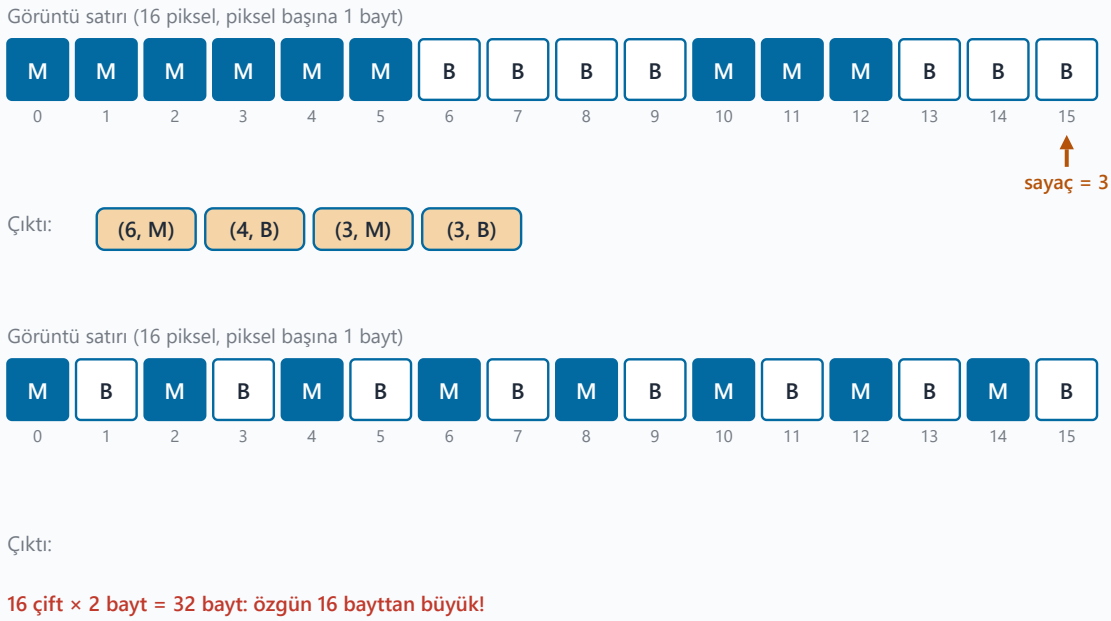
RLE, bir diziyi soldan sağa okuyup her koşuyu (uzunluk, değer) çiftiyle yazan kayıpsız sıkıştırma yöntemidir. Okuma sırasında bir sayaç ve geçerli değer tutulur. Gelen öğe geçerli değerle aynıysa sayaç bir artar. Farklıysa biten koşu (sayaç, geçerli değer) olarak çıktıya yazılır, sayaç 1'e döner ve yeni değer geçerli değer olur. Dizi bitince açık kalan son koşu da yazılır.

Örnek 1.1.14

Bir görüntünün 16 piksellik bir satırı iki renkten oluşuyor: M (mavi) ve B (beyaz). Satır, piksel numaraları 0'dan 15'e olmak üzere MMMMMMBBBBMMMBBB'dir. Her piksel bir bayt tutar; satır 16 bayttır.

Piksel	Renk	Olay	Sayaç	Çıktı
0	M	ilk piksel, geçerli renk M	1	—
1–5	M	aynı renk, sayaç her pikselde artar	6	—
6	B	renk değişti, koşu biter	1	(6, M)
7–9	B	aynı renk	4	(6, M)
10	M	renk değişti	1	(6, M) (4, B)
11–12	M	aynı renk	3	(6, M) (4, B)
13	B	renk değişti	1	(6, M) (4, B) (3, M)
14–15	B	aynı renk	3	(6, M) (4, B) (3, M)
(son)	—	satır bitti, açık koşu yazılır	—	(6, M) (4, B) (3, M) (3, B)

Sonuç: dört çift elde edilir. Her çift 2 bayttır (bir bayt uzunluk, bir bayt renk): *4 × 2 = 8* bayt. Özgün satır 16 bayttı; RLE onu yarıya indirmiştir. Denetim için uzunluklar toplanır: *6 + 4 + 3 + 3 = 16* piksel.



Şekil 1.1.17: RLE'nin iyi ve kötü durumu. Üstte 16 piksellik satır dört koşudan oluşur ve (6, M) (4, B) (3, M) (3, B) diye 8 bayta iner. Altta renk her pikselde değişir: her koşu 1 uzunluktadır, 16 çift 32 bayt tutar ve çıktı özgün 16 bayttan büyüktür.

Çözme işlemi kodlamanın tersidir. Her çift okunur ve değer, uzunluk kadar yazılır: (6, M) → MMMMMM, (4, B) → BBBB, (3, M) → MMM, (3, B) → BBB. Elde edilen satır özgün satırla birebir aynıdır; hiçbir bilgi kaybolmamıştır. RLE bu yüzden kayıpsızdır.

Şekil 1.1.17'nin alt kısmı yöntemin sınırını gösterir. Renk her pikselde değişirse hiçbir koşu birden uzun olmaz. 16 pikselin her biri için (1, renk) çifti yazılır: *16 × 2 = 32* bayt. "Sıkıştırılmış" veri özgünün iki katına çıkmıştır. RLE ancak uzun tekrarların bol olduğu verilerde işe yarar: siyah-beyaz taranmış belgeler, düz renkli çizimler, simgeler. Fotoğraflarda komşu pikseller nadiren tam aynı renktir; bu yüzden fotoğraflar için başka yöntemler kullanılır.

Sık yapılan hata

Dizinin sonundaki koşuyu yazmayı unutmak. RLE bir koşuyu ancak renk değiştiğinde yazar. Son koşudan sonra renk değişmez, dizi biter. Bu son adım atlanırsa örnekteki çıktı (6, M) (4, B) (3, M)'de kalır ve son üç beyaz piksel kaybolur; çözülen satır 13 pikseldir.

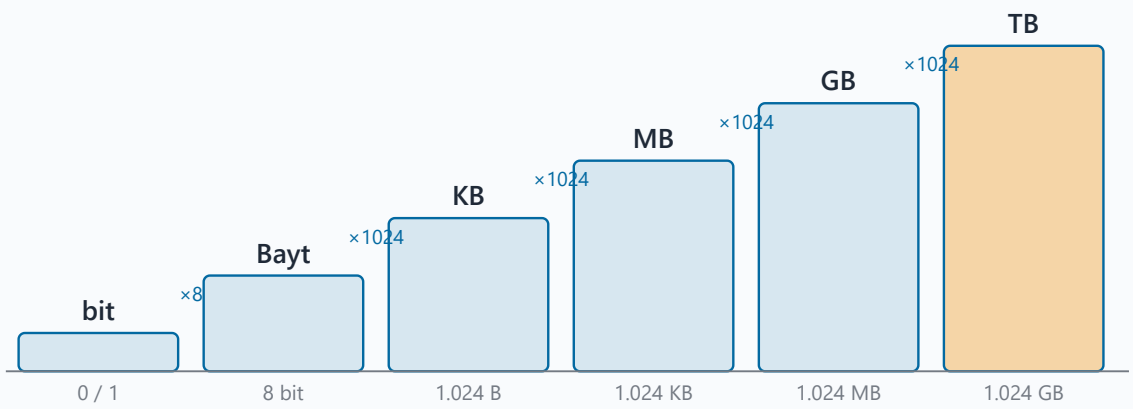
1.1.3.8 Bellek ölçü birimleri

Önceki hesaplarda bayt sayıları milyonlara, milyarlara ulaştı. Bu kadar büyük sayıları rahat yazmak için daha büyük birimler kullanılır. Birimler bitten başlayıp her basamakta büyüyen bir merdiven oluşturur.

Birim	Açılım	İkili karşılığı	Bayt olarak
bit	*binary digit*	0 ya da 1	—
bayt (B)	*byte*	8 bit	1
KB	*kilobyte*	1.024 bayt	2^{10} = 1.024
MB	*megabyte*	1.024 KB	2^{20} = 1.048.576
GB	*gigabyte*	1.024 MB	2^{30} = 1.073.741.824
TB	*terabyte*	1.024 GB	$2^{40} \approx 1{,}1 \times 10^{12}$
PB	*petabyte*	1.024 TB	$2^{50} \approx 1{,}13 \times 10^{15}$

Tablo 1.1.11: Bellek ölçü birimleri ve ikili karşılıkları.

Tablonun ilk iki satırı farklıdır: bit ile bayt arasında 8 kat vardır. Bayttan sonraki her basamak ise öncekinin 1.024 katıdır. 1.024 sayısı rastgele seçilmemiştir; 2^{10}'dur ve 1.000'e çok yakındır. Bellekler ikili adreslerle çalıştığı için kapasiteleri doğal olarak 2'nin kuvvetleri olur. “Kilo” ön eki bu yüzden bellek dünyasında 1.000 değil 1.024 anlamında kullanılmaya başlanmıştır.



Şekil 1.1.18: Bellek birimleri merdiveni. Bitten bayta 8 kat, sonraki her basamakta 1.024 kat büyüme vardır. Çubuk yükseklikleri ölçekli değildir; gerçekte her basamak bir öncekinin yaklaşık bin katıdır.

Birimlerin büyüklüğünü sezmek için tanıdık örnekler yardımcı olur. Tek bir karakter 1 bayttır. Düz metinden oluşan yarım sayfa birkaç KB tutar. Bir telefon fotoğrafı yaklaşık 3 MB, HD kalitesinde bir film yaklaşık 4 GB'dir. Dizüstü bilgisayarların diskleri çoğunlukla 512 GB ile 1 TB arasındadır. Büyük bir bulut hizmetinin veri merkezleri ise petabaytlarca veri saklar.

Örnek 1.1.15

1 KB = 1.024 B, 1 MB = 1.024 KB, 1 GB = 1.024 MB alınarak:

1. **3 KB kaç bittir?** *3 × 1.024 = 3.072* bayt; *3.072 × 8 = 24.576* bit.
2. **128 GB'lık bir telefona 4 MB'lık fotoğraflardan kaç tane sığar?** *128 × 1.024 = 131.072* MB; *131.072 ÷ 4 = 32.768* fotoğraf. (İşletim sistemi ve uygulamaların kapladığı yer bu hesapta yok sayılmıştır.)
3. **5.000.000 bayt kaç MB'dir?** *5.000.000 ÷ 1.048.576 ≈ 4,77* MB.

Büyük birimden küçüğe geçerken çarpılır, küçükten büyüğe geçerken bölünür. Bayt ile bit arasında çarpan 8, öteki her basamakta 1.024'tür.

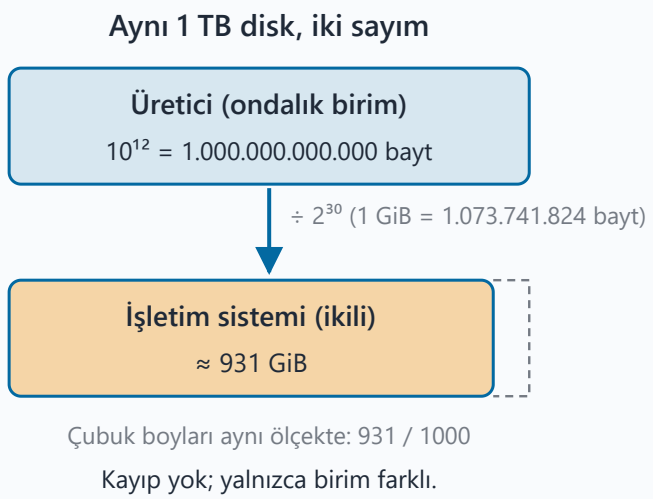
1.024 ile 1.000 arasındaki küçük fark, büyük birimlerde büyür ve bir karışıklığa yol açar. Uluslararası Elektroteknik Komisyonu (IEC), iki anlamı ayırmak için ikili birimlere ayrı adlar vermiştir. 1.024 bayta **KiB** (kibibayt), 2^{20} bayta **MiB** (mebibayt), 2^{30} bayta **GiB** (gibibayt) denir. Bu tanımda KB, MB, GB ön ekleri ondalık anlamlarını korur: 1 GB = 10^9 = 1.000.000.000 bayt. Disk üreticileri kapasiteyi bu ondalık tanımla yazar. Bazı işletim sistemleri ise boyutu ikili birimle hesaplar ama ekranda yine "GB" diye gösterir.

Örnek 1.1.16

Kutusunda "1 TB" yazan bir disk, işletim sisteminde neden yaklaşık 931 GB görünür?

1. Üretici ondalık tanımı kullanır: 1 TB = 10^{12} = 1.000.000.000.000 bayt.
2. İşletim sistemi bayt sayısını 1 GiB = 2^{30} = 1.073.741.824 bayta böler: *1.000.000.000.000 ÷ 1.073.741.824 ≈ 931,3*.
3. Ekranda "931 GB" yazar; doğru adı "931 GiB"dir.

Sonuç: diskte eksik bayt yoktur. Bayt sayısı iki sayımda da aynıdır; yalnızca birimin tanımı farklıdır. Fark birim büyüdükçe artar: KB'de %2,4, GB'de %7,4, TB'de yaklaşık %10'dur.



Şekil 1.1.19: Aynı 1 TB disk, iki sayım. Üretici 10^{12} bayt yazar; işletim sistemi bunu 2^{30} bayta bölüp ≈ 931 GiB gösterir. Çubuk boyları aynı ölçektedir: 931 / 1000.

Sık yapılan hata

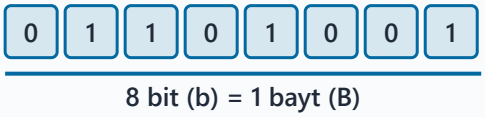
1 TB ile 931 GB arasındaki farkı "biçimlendirmede kaybolan yer" ya da "bozuk sektör" sanmak. Biçimlendirmenin ve dosya sisteminin kapladığı yer çok küçüktür. Farkın neredeyse tamamı birim tanımından gelir; iki sayı aynı bayt miktarını farklı birimlerle yazar.

1.1.3.9 Bit mi bayt mı? Hız ve boyut

Birim kısaltmalarında büyük ve küçük harf aynı şey değildir. Küçük **b** biti, büyük **B** baytı gösterir. 100 Mb (megabit) ile 100 MB (megabayt) arasında tam 8 kat fark vardır. Bu ayrım en çok internet hızı ile dosya boyutu birlikte konuşulurken önem kazanır, çünkü ikisi geleneksel olarak farklı birimlerle ölçülür.

Tanım 1.1.13

Veri aktarım hızı, saniyede aktarılan *bit* sayısıyla ölçülür: bit/s, kbit/s, Mbit/s (kısaca Mbps) gibi. Hız birimlerindeki ön ekler ondalıktır: 1 Mbps = 10^6 = 1.000.000 bit/s. **Dosya ve disk boyutu** ise *bayt* ile ölçülür: KB, MB, GB.



Şekil 1.1.20: 8 bit (b) = 1 bayt (B).

Bir dosyanın ne kadar sürede ineceği hesaplanırken iki birim aynı dile çevrilmelidir. Ya hız bayta çevrilir (8'e bölünür) ya da boyut bite çevrilir (8 ile çarpılır). İki yol da aynı sonucu verir.

Örnek 1.1.17

100 Mbps hızındaki bir bağlantıyla 3 GB'lık (3×10^9 bayt) bir dosya en iyi durumda kaç saniyede iner?

Adım	Hesap	Sonuç
Hız bayta çevrilir	100.000.000 ÷ 8	12.500.000 bayt/s = 12,5 MB/s
Dosya bite çevrilir	3×10^9 bayt × 8	24×10^9 bit
Süre	$24 \times 10^9 \div 10^8$	240 s = 4 dakika

Aynı sonuç birinci satırdan da bulunur: *3.000.000.000 ÷ 12.500.000 = 240* s. “100 Mbps ile saniyede 100 MB iner” diye düşünülseydi süre 30 saniye bulunurdu; bu, gerçeğin sekizde biridir.

240 saniye en iyi durumdur. Gerçekte süre daha uzundur. Ağda gönderilen her veri parçasına adres ve denetim bilgisi eklenir; bu **protokol ek yükü**, hattın bir kısmını kullanır. Aynı hattı paylaşan başka cihazlar da hızı böler. Sunucunun kendisi de sınırlı hızda veri gönderebilir. Bu yüzden 100 Mbps'lik bir bağlantıda indirme hızı çoğunlukla 10–11 MB/s civarında görülür.

Sık yapılan hata

İndirme programının gösterdiği “11 MB/s” değerini internet paketindeki “100 Mbps” ile karşılaştırıp hızın düşük olduğunu sanmak. 11 MB/s = 88 Mbps'tir; ek yük ve paylaşım düşünülünce 100 Mbps'lik bir hat için iyi bir değerdir.

1.1.4 Veri işleme

Bilgisayar veriyi yalnızca saklamaz, onunla hesap yapar ve kararlar verir. Bu kesim işlemcinin yaptığı temel işlem türlerini bir arada ele alır. Ardından aynı işlemlerin, verinin iletim sırasında bozulup bozulmadığını anlamak için nasıl kullanıldığını parite biti örneğiyle gösterir.

1.1.4.1 Aritmetik ve mantıksal işlemler

Programların yaptığı işlerin çoğu dört türlü işlemin birleşimidir.

Tür	Ne yapar?	İşleçler	Sonucun türü
Aritmetik	Sayılarla hesap yapar	+ − × ÷ %	Sayı
Karşılaştırma	İki değerin ilişkisini sınar	> < ≥ ≤ == !=	Doğru / yanlış

Tür	Ne yapar?	İşleçler	Sonucun türü
Mantıksal	Koşulları birleştirir ya da tersine çevirir	AND, OR, NOT	Doğru / yanlış
Karar	Sonuca göre programın izleyeceği yolu seçer	eğer … ise	Bir yol

Tablo 1.1.12: Temel işlem türleri.

Aritmetik işleçlerden % işareti yüzde değil, **bölümden kalan** anlamına gelir: *17 % 5 = 2*, çünkü 17'nin içinde üç tane 5 vardır ve 2 artar. Bu işlem, onluktan ikiliye dönüşümde kullanılan "mod" işleminin aynısıdır. Karşılaştırma işleçlerinde == "eşit mi?" ve != "eşit değil mi?" anlamındadır. Tek = işareti programlama dillerinde atama için kullanıldığından eşitlik sorusu iki eşittir işaretiyle yazılır.

Mantıksal işlemlerin girdisi ve çıktısı doğru ya da yanlıştır. Bilgisayarda doğru 1, yanlış 0 ile gösterilir. Üç temel mantıksal işlem vardır.

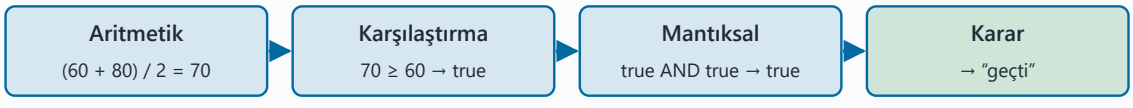
Tanım 1.1.14

AND (ve): iki koşulun *ikisi de* doğruysa doğru, en az biri yanlışsa yanlış verir. **OR** (veya): koşullardan *en az biri* doğruysa doğru, ikisi de yanlışsa yanlış verir. **NOT** (değil): tek bir koşulu tersine çevirir; doğruyu yanlış, yanlışı doğru yapar.

A	B	A AND B	A OR B	NOT A
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

Tablo 1.1.13: AND, OR ve NOT işlemlerinin doğruluk tablosu (1 = doğru, 0 = yanlış).

Tablo 1.1.13 dört satırla bütün durumları kapsar, çünkü iki koşulun alabileceği yalnızca dört değer bileşimi vardır. AND sütununda yalnızca son satır 1'dir: "ve" bağlacı iki tarafın da sağlanmasını ister. OR sütununda yalnızca ilk satır 0'dır: "veya" bağlacı tek bir tarafın sağlanmasıyla yetinir. Günlük dildeki "ve" ile "veya" da bu anlamlarda kullanılır. "Sınavı geçen ve devamı tam olan öğrenci" iki koşulu birden ister. "Kimliği ya da pasaportu olan yolcu" ise ikisinden birini yeterli bulur.



Örnek: vize 60, final 80, devam_ok = true

Şekil 1.1.21: Bir not hesabında dört işlem türü. Vize 60, final 80 ve devam_ok = doğru için: aritmetik (60 + 80) / 2 = 70; karşılaştırma 70 ≥ 60 → doğru; mantıksal doğru AND doğru → doğru; karar: "geçti".

Örnek 1.1.18

Bir derste kural şöyledir: ortalama = (vize + final) / 2; ortalaması 60 ve üstü olan *ve* devam koşulunu sağlayan öğrenci geçer. İki öğrenciyi izleyelim.

Adım	1. öğrenci	2. öğrenci
Veri	vize 60, final 80, devam_ok = doğru	vize 40, final 90, devam_ok = yanlış
Aritmetik	(60 + 80) / 2 = 70	(40 + 90) / 2 = 65
Karşılaştırma	70 ≥ 60 → 1	65 ≥ 60 → 1
Mantıksal	1 AND 1 → 1	1 AND 0 → 0
Karar	"geçti"	"kaldı"

İkinci öğrencinin ortalaması yeterlidir, ama devam koşulu sağlanmadığı için AND işlemi 0 verir. Kural "veya" ile yazılsaydı (1 OR 0 = 1) bu öğrenci de geçerdi. Bağlacın seçimi kuralın anlamını değiştirir.

Şekil 1.1.21 ve Örnek 1.1.18, dört işlem türünün bir zincir hâlinde nasıl çalıştığını gösterir. Aritmetik işlem sayıyı üretir. Karşılaştırma bu sayıyı doğru/yanlış değerine çevirir. Mantıksal işlem birden fazla doğru/yanlış değerini birleştirir. Karar adımı da son değere göre yolu seçer. İlk üç iş ALU'da yapılır. Karar adımında ise kontrol birimi, sonuca göre programın bir sonraki komutunu seçer.

Sık yapılan hata

"Büyük" ile "büyük ya da eşit"i karıştırmak. 60 puanın geçer sayılması isteniyorsa koşul "puan > 60" değil "puan ≥ 60" olmalıdır. "puan > 60" yazılırsa tam 60 alan öğrenci kalır. Sınır değerlerin koşulun hangi tarafında kaldığı her zaman ayrıca denetlenmelidir.

1.1.4.2 Parite biti ile hata yakalama

Veri bir kablodan ya da kablosuz bağlantıdan geçerken elektriksel gürültü bazı bitleri bozabilir. 1 olan bit 0'a ya da 0 olan bit 1'e dönebilir. Alıcı, kendisine ulaşan bitlerin doğru olup olmadığını nereden bilecektir? En basit çözüm, veriye denetim amaçlı fazladan bir bit eklemektir.

Tanım 1.1.15

Parite biti (*parity bit*), veri bitlerine eklenen ve toplam 1 sayısını önceden kararlaştırılmış bir kurala uyduran denetim bitidir. **Çift parite** kuralında parite biti, veri bitleri ile parite bitindeki 1'lerin toplam sayısı çift olacak biçimde seçilir. Veri bitlerindeki 1 sayısı çiftse parite biti 0, tekse 1'dir. Alıcı bütün bitlerdeki 1'leri sayar; sayı tek çıkarsa hata vardır.

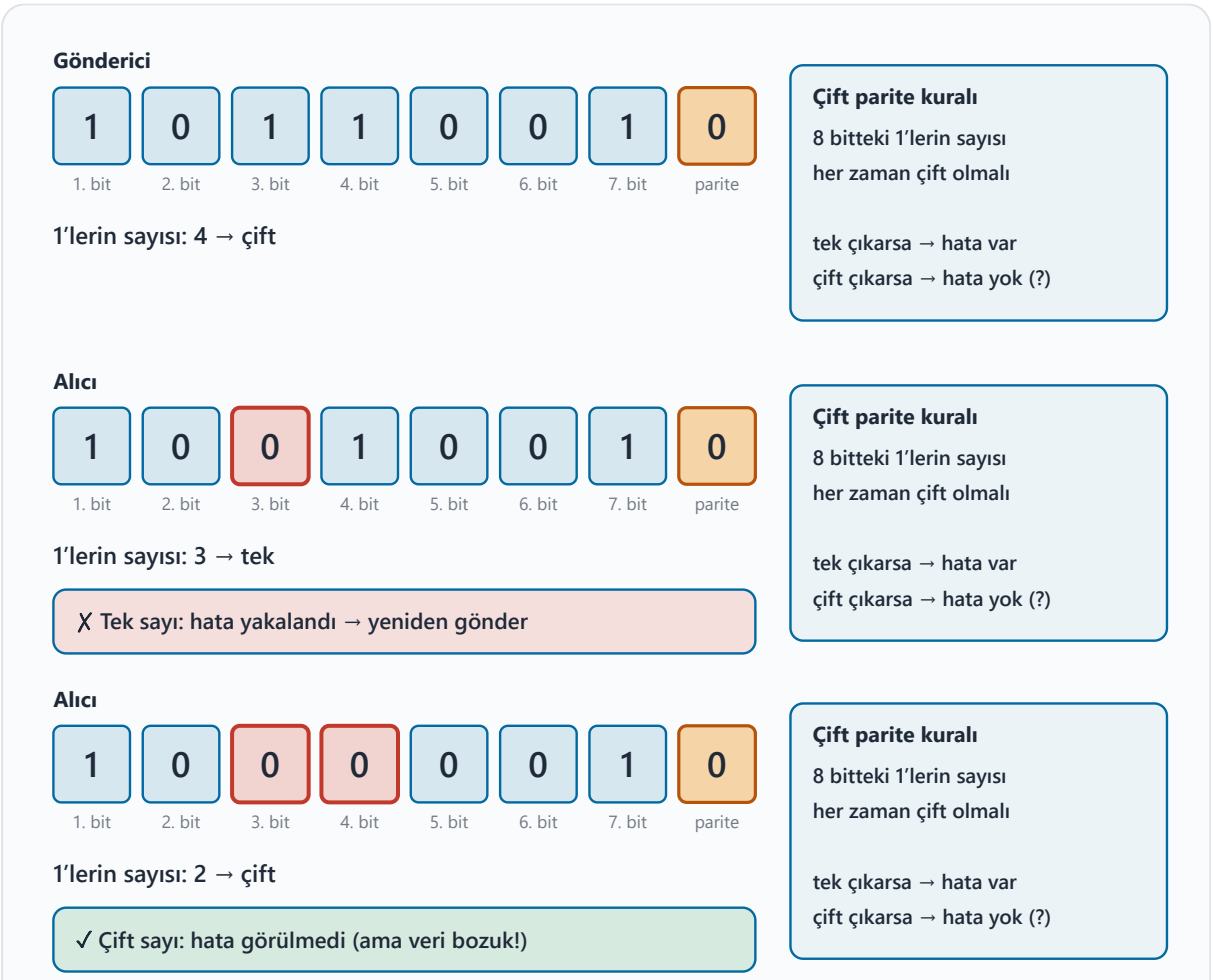
Kuralın bir benzeri de **tek parite**dir: orada toplam 1 sayısı tek olmalıdır. İki kuralın gücü aynıdır; gönderici ile alıcının aynı kuralı kullanması yeter. Aşağıdaki örnek çift pariteyi kullanır.

Örnek 1.1.19

ASCII'de "Y" harfinin kodu 89'dur; yedi bitle 1011001 diye yazılır. Bu harf çift pariteyle gönderiliyor.

1. **Gönderici 1'leri sayar.** 1011001'de 1'ler 1., 3., 4. ve 7. bitlerdedir: dört tane. Dört çift olduğu için parite biti 0'dır. Gönderilen sekiz bit: 10110010.
2. **İletimde bir bit bozulur.** Gürültü 3. biti 1'den 0'a çevirir. Alıcıya 10010010 ulaşır.
3. **Alıcı 1'leri sayar.** 10010010'da üç tane 1 vardır. Üç tek bir sayıdır; çift parite kuralı bozulmuştur. Alıcı hatayı fark eder ve veriyi yeniden ister.

Sonuç: tek bitlik hata yakalanır. Ama alıcı yalnızca bir hata *olduğunu* bilir; hangi bitin bozulduğunu bilemez. Bu yüzden hatayı kendisi düzeltemez, verinin yeniden gönderilmesini ister.



Şekil 1.1.22: Çift parite ile hata denetimi. Üstte gönderici: yedi veri bitinde dört 1 var, parite biti 0. Ortada alıcı: 3. bit bozulmuş, 1 sayısı 3 (tek), hata yakalandı. Altta iki bit birden bozulmuş: 1 sayısı 2 (çift), alıcı hata görmez

ama veri bozuktur.

Şekil 1.1.22'nin son satırı parite bitinin sınırını gösterir. 3. ve 4. bitler birlikte bozulursa alıcıya 10000010 ulaşır. Bu dizide iki tane 1 vardır; iki çift bir sayıdır. Alıcı kuralın sağlandığını görür ve bozuk veriyi doğru sanır. Genel kural şudur: bir bitin bozulması 1 sayısını bir artırır ya da bir azaltır, yani tek/çift durumunu değiştirir. İki bitin bozulması durumu iki kez değiştirir ve başa döndürür. Parite biti bu yüzden yalnızca *tek sayıda* bit hatasını yakalar; çift sayıda hata gözden kaçar.

Bozulan bit	Alınan dizi	1 sayısı	Alıcının kararı	Gerçek durum
Yok	10110010	4 (çift)	Hata yok	Doğru
3. bit	10010010	3 (tek)	Hata var	Yakalandı
3. ve 4. bit	10000010	2 (çift)	Hata yok	Gözden kaçtı
1., 3. ve 4. bit	00000010	1 (tek)	Hata var	Yakalandı

Tablo 1.1.14: Bozulan bit sayısına göre parite denetiminin sonucu (gönderilen 10110010, dört tane 1).

Parite biti çok ucuz bir denetimdir: her yedi ya da sekiz bit için tek bir fazladan bit. Hataların seyrek ve tek tek geldiği ortamlarda işe yarar. Gürültünün bitleri öbek hâlinde bozduğu ortamlarda ise daha güçlü yöntemler kullanılır. Bu yöntemler daha fazla denetim biti ekleyerek birden çok hatayı yakalar, bazıları hatanın yerini bulup düzeltir de. Bellek modüllerinde kullanılan hata düzelten kodlar (*error-correcting code*, ECC) ve internet paketlerindeki sağlama toplamları bu türdendir.

Sık yapılan hata

Parite bitini hesaplarken veri bitlerinin *değerini* (89) ya da bit *sayısını* (7) kullanmak. Parite yalnızca 1 olan bitlerin sayısına bakar. 1011001'in değeri olan 89 tektir, ama içindeki 1 sayısı dört, yani çifttir; parite biti 0'dır.

1.1.5 Bilgisayar türleri

Bütün bilgisayarlar aynı çalışma döngüsünü kullanır, ama farklı işler için farklı biçimlerde üretilir. Bir hava durumu modelini çalıştıran makine ile bir çamaşır makinesinin denetim kartı, aynı ilkelerle çalışan iki çok farklı bilgisayardır. Bu kesim başlıca bilgisayar türlerini, her birinin hangi ihtiyaca cevap verdiğini ve birbirlerinden nerede ayrıldıklarını anlatır.

1.1.5.1 Kapasite ve kullanım amacına göre sınıflandırma

Bilgisayarlar en çok iki ölçüte göre sınıflandırılır: kapasite (ne kadar hesap gücü, bellek ve depolama sunduğu) ve kullanım amacı (kime, hangi iş için hizmet ettiği). Tablo 1.1.15 başlıca türleri bu iki ölçüte göre özetler.

Tür	Öncelikli amaç	Tipik örnek
Süper bilgisayar	Yoğun bilimsel hesaplama	İklim modeli, fizik benzetimi
Ana bilgisayar	Çok sayıda güvenilir işlem	Bankacılık, sigorta
Sunucu	Ağ üzerinden hizmet	Web sitesi, e-posta, veri tabanı
Kişisel bilgisayar	Bireysel kullanım	Masaüstü, dizüstü
Gömülü sistem	Belirli bir cihaz işlevi	Klima, otomobil, çamaşır makinesi
İnce istemci	Merkezî sisteme erişim	Kütüphane, banka şubesi terminali
Mobil cihaz	Taşınabilir kullanım	Telefon, tablet
Kuantum bilgisayar	Özel problem sınıfları	Kimya benzetimi, araştırma

Tablo 1.1.15: Başlıca bilgisayar türleri ve tipik kullanımları.

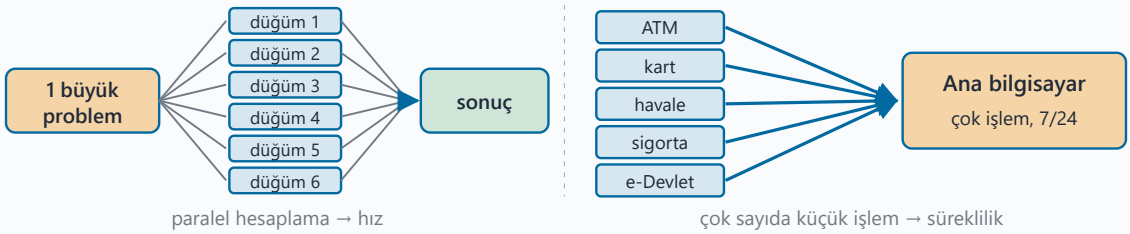
Bu sınıflandırmanın kesin sınırları yoktur. Güçlü bir masaüstü bilgisayar, üzerine bir web sitesi kurulunca sunucu görevi görür. Bir akıllı telefon hem mobil cihaz hem kişisel bilgisayardır. Bir otomobilde onlarca gömülü bilgisayar bulunur ve bunlar bir ağ üzerinden birbirleriyle konuşur. Bir cihazın hangi türe girdiği, donanımından çok o anda nasıl kullanıldığına bağlıdır.

1.1.5.2 Süper ve ana bilgisayarlar

Kapasite ölçeğinin en üstünde iki tür vardır. İkisi de çok güçlü ve pahalıdır, ama farklı bir şeyi en iyi yapmak için tasarlanmıştır.

Tanım 1.1.16

Süper bilgisayar (*supercomputer*), çok yüksek hesaplama gücü gerektiren tek bir büyük problemi, binlerce işlemciye bölüştürerek paralel çözen bilgisayardır. **Ana bilgisayar** (*mainframe*), çok sayıda kullanıcıdan gelen küçük işlemleri eş zamanlı, kesintisiz ve güvenilir biçimde yürütmek için tasarlanmış bilgisayardır.



Şekil 1.1.23: Solda süper bilgisayar: tek büyük problem altı düğüme bölünür, düğümler paralel hesaplar, parçalar birleşip sonuç olur; amaç hızdır. Sağda ana bilgisayar: ATM, kart, havale, sigorta ve e-Devlet işlemleri gibi çok sayıda küçük iş 7/24 işlenir; amaç sürekliliktir.

Şekil 1.1.23 iki türün farkını iş yüküyle gösterir. Bir süper bilgisayara bir iklim modeli verildiğinde dünya atmosferi milyonlarca küçük hücreye bölünür. Her işlemci birkaç hücrenin sıcaklığını, basıncını ve rüzgârını hesaplar ve sonuçlarını komşularıyla paylaşır. İş, tek bir bilgisayarda aylar sürecekken birkaç saatte biter. Süper bilgisayarlar fizik benzetimlerinde, ilaç ve malzeme araştırmalarında, büyük yapay zekâ modellerinin eğitiminde de kullanılır. Güçleri saniyede yapılan kayan noktalı işlem sayısıyla (FLOPS, *floating-point operations per second*) ölçülür.

Ana bilgisayarın işi farklıdır. Bir bankada her saniye binlerce kart ödemesi, havale ve bakiye sorgusu gelir. Bu işlemlerin her biri tek başına çok küçüktür; zor olan, hepsini aynı anda, hiç hata yapmadan ve hiç durmadan işlemektir. Bir ödeme sisteminin bir saatlik kesintisi büyük kayıplara yol açar. Ana bilgisayarlar bu yüzden yedekli donanımla kurulur: bir parça bozulduğunda sistem durmadan çalışmayı sürdürür ve parça çalışırken değiştirilebilir.

Öncelik	Süper bilgisayar	Ana bilgisayar
Temel amaç	Hesaplama hızı	İşlem hacmi ve süreklilik
Tipik iş yükü	Büyük bilimsel hesap	Çok sayıda kurumsal işlem
Kullanıcı	Az sayıda araştırmacı	Binlerce eş zamanlı kullanıcı
Başarı ölçüsü	Saniyedeki işlem (FLOPS)	Saniyedeki hareket sayısı, kesintisiz çalışma süresi

Tablo 1.1.16: Süper bilgisayar ile ana bilgisayarın karşılaştırması.

Sık yapılan hata

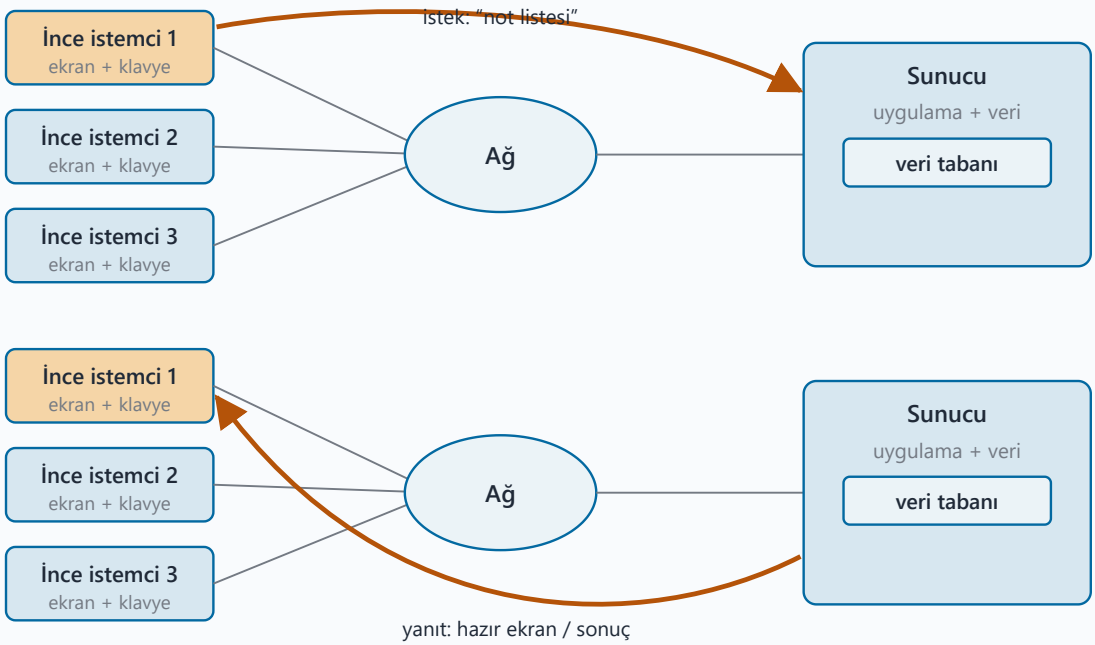
Ana bilgisayarı “eski, yavaş süper bilgisayar” sanmak. İki tür aynı ölçeğin farklı noktaları değildir; farklı amaçlara göre tasarlanmıştır. Bir bankanın işi için süper bilgisayar, bir iklim modeli için ana bilgisayar uygun değildir.

1.1.5.3 Sunucular ve ince istemciler

Tanım 1.1.17

Sunucu (*server*), ağ üzerinden başka bilgisayarlara web, e-posta, dosya, veri tabanı ya da uygulama hizmeti veren bilgisayardır. Hizmet alan bilgisayara **istemci** (*client*) denir. **İnce istemci** (*thin client*), işlemin çoğunu sunucuya bırakan, kendi üzerinde yalnızca görüntüleme ve girdi alma işini yapan basit bir istemcidir.

Sunucu ile istemci arasındaki konuşma bir **istek–yanıt** (*request–response*) düzeniyle yürür. İstemci bir şey ister; sunucu isteği işler ve sonucu geri gönderir. Bir web sayfası açıldığında da, bir e-posta indirildiğinde de bu düzen işler.



Şekil 1.1.24: Üç ince istemci ağ üzerinden aynı sunucuya bağlıdır. Üstte istek: istemci 1 not listesini ister ve istek ağ üzerinden sunucuya gider. Altta yanıt: sunucu veri tabanını sorgulayıp sonucu hazırlar, hazır ekran istemciye döner.

Örnek 1.1.20

Bir üniversite kütüphanesindeki ince istemciden not listesi açılıyor. Şekil 1.1.24 üç adımı gösterir:

1. **İstek.** İstemci 1, "not listesi" isteğini ağ üzerinden sunucuya gönderir. İstemcinin yaptığı yalnızca ekrandaki tıklamayı bir isteğe çevirmektir.
2. **İşlem.** Asıl iş sunucuda yapılır: veri tabanı sorgulanır, notlar bulunur, ortalamalar hesaplanır ve sonuç bir ekran olarak hazırlanır.
3. **Yanıt.** Hazır ekran istemciye döner. İnce istemci onu yalnızca gösterir ve kullanıcının bir sonraki tıklamasını bekler.

Çalışma döngüsü burada iki bilgisayar arasında bölüşülmüştür: giriş ve çıkış istemcide, işlem ve depolama sunucudadır.

İnce istemcinin üstünlüğü yönetim kolaylığıdır. Yüz ince istemcili bir kütüphanede programlar yalnızca sunucuya kurulur ve yalnızca orada güncellenir. İstemcilerde veri saklanmadığı için biri bozulduğunda ya da çalındığında veri kaybolmaz; yerine yenisi takılır. Yerel donanım ucuz ve basit olabilir. Zayıf yanı ise bağımlılıktır: ağ ya da sunucu çökerse yüz istemcinin hepsi birden işe yaramaz hâle gelir.

Sık yapılan hata

Sunucuyu yalnızca büyük, özel bir makine sanmak. “Sunucu” bir donanım türünden çok bir roldür. Evdeki bir dizüstü, üzerinde bir dosya paylaşım programı çalışırken öteki bilgisayarlar için sunucudur. Veri merkezlerindeki sunucular ise bu rolü sürekli üstlenmek için güçlü, yedekli ve dolaplara sığacak biçimde üretilmiştir.

1.1.5.4 Kişisel ve mobil cihazlar

Kişisel bilgisayar (*personal computer*), tek bir kişinin kullanımı için tasarlanmış genel amaçlı bilgisayardır. Kişisel ve mobil cihazlar, aynı genel amaçlı bilgisayarın farklı kullanım koşullarına uyarlanmış biçimleridir.

Cihaz	Öne çıkan özellik	Bedeli
Masaüstü	Sabit kullanım, genişletilebilir donanım, yüksek performans	Taşınamaz, daha çok enerji harcar
Dizüstü	Taşınabilir yapı, batarya	Donanım yükseltmesi sınırlı
Tablet	Dokunmatik ekran, hafif kullanım	Uzun metin yazmak ve yoğun iş zor
Telefon	Mobil bağlantı, sensörler (GPS, ivmeölçer, kamera), uygulamalar	Küçük ekran, sınırlı batarya

Tablo 1.1.17: Kişisel ve mobil cihazların karşılaştırması.

Masaüstü bilgisayarda kasanın içi açılabilir; ekran kartı, bellek ya da disk değiştirilip yükseltilebilir. Bol enerji ve geniş soğutma alanı sayesinde aynı fiyata en yüksek performansı verir. Dizüstü bilgisayar aynı parçaları küçük bir gövdeye ve bataryaya sığdırır; performansın bir kısmı taşınabilirlik ve pil ömrü için feda edilir. Tablet ve telefon bu yönde daha da ileri gider: enerji tüketimi o kadar düşük tutulur ki işlemci soğutma fanı olmadan çalışır.

Telefonu öteki kişisel cihazlardan ayıran asıl özellik sensörleridir. Konum alıcısı, ivmeölçer, jiroskop, kamera, mikrofon ve parmak izi okuyucusu, telefonu dünyayı sürekli ölçen bir giriş cihazına çevirir. Harita uygulamasının yönü göstermesi, ekranın telefon yan çevrilince dönmesi ve adım sayarın çalışması bu sensörler sayesindedir.

Cihaz seçiminde performans tek ölçüt değildir. Taşınabilirlik, enerji tüketimi, ekran boyutu ve asıl kullanım amacı birlikte düşünülür. Uzun raporlar yazacak ve programlama yapacak bir öğrenci için iyi bir klavye ve geniş ekranlı bir dizüstü, en hızlı telefondan daha uygundur.

1.1.5.5 Gömülü bilgisayarlar

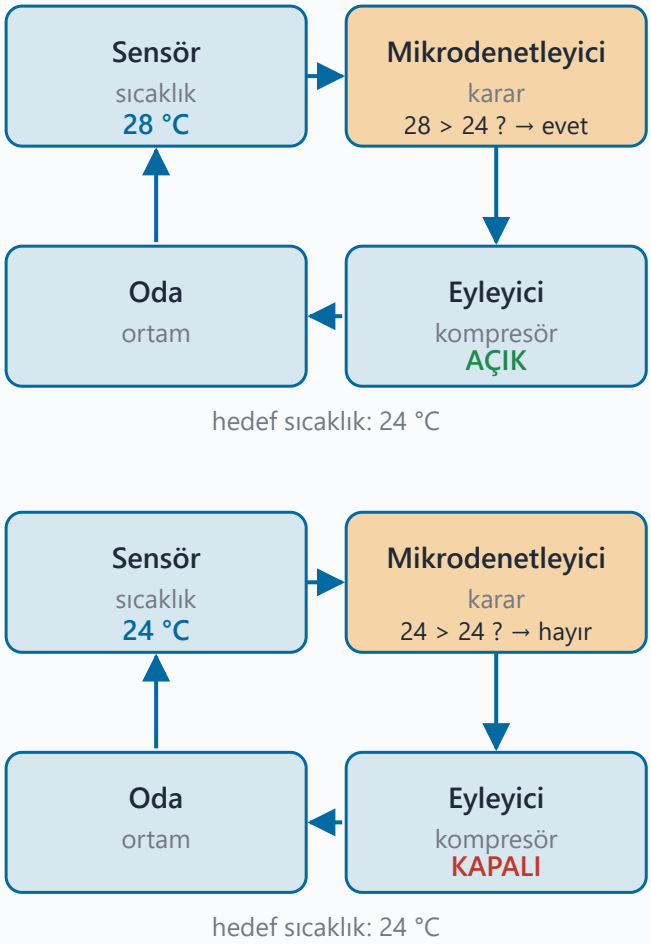
Tanım 1.1.18

Gömülü sistem (*embedded system*), daha büyük bir cihazın içine yerleştirilmiş ve o cihazın belirli bir görevini yerine getiren bilgisayardır. Çoğunlukla işlemci, bellek ve giriş–çıkış birimlerini tek bir yongada birleştiren bir **mikrodenetleyici** (*microcontroller*) üzerine kurulur.

Gömülü bilgisayarlar sayıca en yaygın bilgisayar türüdür. Bir otomobilde motoru, frenleri ve hava yastıklarını denetleyen onlarca gömülü bilgisayar vardır. Çamaşır makinesi, mikrodalga fırın, klima, akıllı saat, tıbbi ölçüm cihazı ve trafik lambası birer gömülü bilgisayar içerir. Bu bilgisayarların ortak özellikleri şunlardır:

- Tek bir işi yaparlar ve programları cihaz üretilirken yüklenir.
- Düşük güç tüketirler ve sınırlı bellek, sınırlı hesap gücüyle çalışırlar.
- Güvenilirlik ve **gerçek zamanlı** tepki önemlidir: hava yastığı denetimi çarpışmayı milisaniyeler içinde algılayıp karar vermelidir.
- Kullanıcı, cihazın içinde bir bilgisayar olduğunu çoğunlukla fark etmez.

Gömülü bir bilgisayar, çalışma döngüsünü bir **denetim döngüsü** biçiminde sürekli tekrarlar: ölç, karar ver, uygula, yeniden ölç. Klima örneğinde giriş birimi bir sıcaklık sensörü, çıkış birimi ise kompresörü açıp kapatan bir **eyleyicidir** (*actuator*).



Şekil 1.1.25: Klimanın denetim döngüsü; hedef sıcaklık 24 °C. (a) Sensör 28 °C ölçer; 28 > 24 doğru olduğu için mikrodenetleyici kompresörü açık tutar. (b) Sensör 24 °C ölçer; 24 > 24 yanlış olduğu için kompresör kapanır.

Örnek 1.1.21

Klimanın hedef sıcaklığı 24 °C'dir ve kural şudur: ölçülen sıcaklık hedeften büyükse kompresör açılır, değilse kapanır. Sensör art arda 28, 26 ve 24 °C ölçüyor.

Tur	Ölçüm (giriş)	Koşul (işlem)	Kompresör (çıkış)	Oda
1	28 °C	28 > 24 → doğru	AÇIK	soğur
2	26 °C	26 > 24 → doğru	AÇIK	soğur
3	24 °C	24 > 24 → yanlış	KAPALI	soğuma durur

Üçüncü turda 24, 24'ten büyük değildir, ona eşittir; “büyüktür” koşulu yanlış olur ve kompresör kapanır. Döngü burada bitmez. Sensör ölçmeyi sürdürür; oda ısınıp sıcaklık yeniden 24'ü geçince kompresör tekrar açılır.

Bu basit kuralın pratikte bir kusuru vardır. Sıcaklık 24,0 ile 24,1 arasında gidip gelirse kompresör çok sık açılıp kapanır ve yıpranır. Gerçek klimalarda bu yüzden iki eşik kullanılır: örneğin sıcaklık 25'i geçince açılır, 23'ün altına inince kapanır. Arada kalan bölgede kompresör önceki durumunu korur. Bu küçük değişiklik, gömülü sistemlerde yazılımın donanımın ömrünü doğrudan etkilediğini gösterir.

Sık yapılan hata

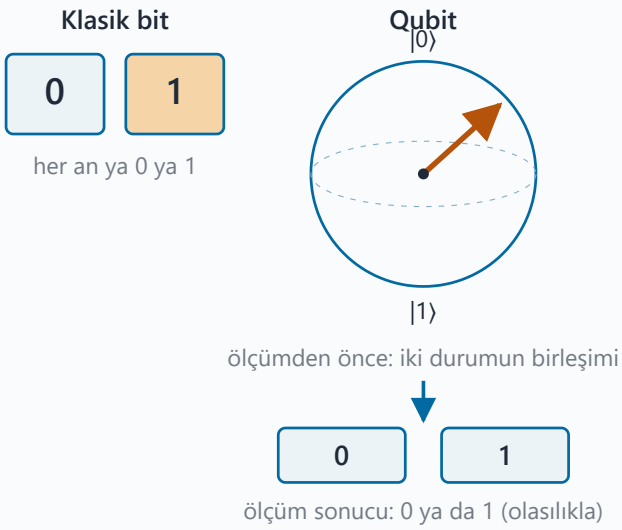
Gömülü bilgisayarın döngüsünün bir kez çalışıp bittiğini sanmak. Bir hesap programı sonucu verip durur; gömülü bir denetim programı ise cihaz açık olduğu sürece hiç durmadan ölçmeye ve karar vermeye devam eder.

1.1.5.6 Kuantum bilgisayarlar

Şimdiye kadar anlatılan bütün bilgisayarlar, bilgiyi 0 ya da 1 olan bitlerle işler. Kuantum bilgisayarlar ise kuantum fiziğinin kurallarına dayanan farklı bir bilgi birimi kullanır.

Tanım 1.1.19

Qubit (*quantum bit*, kuantum biti), ölçülmeden önce 0 ve 1 durumlarının bir birleşiminde bulunabilen bilgi birimidir. Qubit ölçüldüğünde 0 ya da 1 sonucunu verir; hangisinin çıkacağı, birleşimdeki ağırlıklara bağlı bir olasılıkla belirlenir. **Kuantum bilgisayar** (*quantum computer*), qubitlerle hesap yapan bilgisayardır.



Şekil 1.1.26: Klasik bit her an ya 0 ya 1'dir. Qubit ölçümden önce |0⟩ ve |1⟩ durumlarının birleşimindedir; ölçülünce olasılıkla 0 ya da 1 verir.

Kuantum bilgisayarlar belirli problem sınıflarında klasik bilgisayarlardan çok farklı hesaplama yöntemleri sunar. Moleküllerin ve malzemelerin davranışını benzetmek, bazı optimizasyon problemleri ve bazı şifreleme yöntemlerinin çözümü bu sınıflar arasında incelenir. Özellikle büyük sayıları asal çarpanlarına ayırma problemi, bugün yaygın kullanılan bazı şifreleme sistemlerinin güvenliğinin dayandığı problem olduğu için yakından izlenir.

Bununla birlikte kuantum bilgisayarlar günlük kullanımda klasik bilgisayarların yerini almış değildir ve yakın gelecekte de genel amaçlı kullanımda onların yerine geçmesi beklenmez. Qubitler çevreden gelen en küçük etkiyle bozulur; bu yüzden çoğu kuantum bilgisayar mutlak sıfıra yakın sıcaklıklarda soğutularak çalıştırılır. Kelime işlemek, internette gezinmek ya da bir tabloyu hesaplamak gibi işlerde kuantum bilgisayarın hiçbir üstünlüğü yoktur. Kuantum bilgisayar, klasik bilgisayarın daha hızlı bir sürümü değil, belirli problemler için tasarlanmış farklı bir araçtır.

Sık yapılan hata

Qubitin "aynı anda bütün cevapları deneyip doğrusunu seçtiğini" sanmak. Ölçüm her zaman tek bir sonuç verir. Kuantum algoritmaları, yanlış cevapların olasılığını azaltıp doğru cevabın olasılığını artıracak biçimde dikkatle tasarlanır; bu yalnızca belirli problemlerde mümkündür.

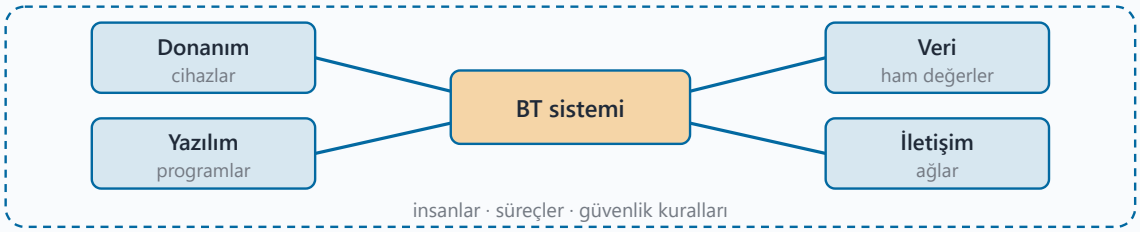
1.1.6 Bilgi teknolojileri

Önceki kesimler bilgisayarın içine baktı: bitlere, işlemlere, donanım türlerine. Bu kesim dışarıya, bilgisayarların insanlar ve kurumlarla birlikte oluşturduğu sistemlere bakar. Veriyi bilgiye dönüştüren bu sistemlerin bütününe bilgi teknolojileri denir.

1.1.6.1 Bilgi teknolojisi nedir?

Tanım 1.1.20

Bilgi teknolojisi (*information technology*, BT), veriyi toplamak, işlemek, saklamak ve iletmek için kullanılan sistemlerin bütünüdür. Bir BT sisteminin dört teknik bileşeni vardır: **donanım** (fiziksel cihazlar), **yazılım** (programlar ve işletim sistemleri), **veri** (işlenen ham değerler) ve **iletişim** (ağlar ve veri aktarımı).



Şekil 1.1.27: Bir BT sistemi. Ortada dört teknik bileşen (donanım, yazılım, veri, iletişim); hepsinin çevresinde insanlar, süreçler ve güvenlik kuralları.

Şekil 1.1.27 BT sisteminin yalnızca bilgisayardan oluşmadığını vurgular. Bir hastanenin hasta kayıt sistemi düşünülsün. Donanım sunucular, bilgisayarlar ve yazıcılardır. Yazılım kayıt programı ve veri tabanıdır. Veri hasta bilgileri, tahliller, reçetelerdir. İletişim, binalar arasındaki ağdır. Ama sistem, kayıtları giren sekreterler, onları okuyan doktorlar ve hangi bilgiyi kimin görebileceğini belirleyen kurallar olmadan çalışmaz. Sistemin dış halkasındaki insanlar, süreçler ve güvenlik kuralları, teknik bileşenler kadar önemlidir. En iyi donanım bile, kullanıcı şifresini bir kâğıda yazıp ekrana yapıştırıyorsa güvenli değildir.

1.1.6.2 Veri ve bilgi arasındaki fark

Veri ve bilgi günlük dilde çoğunlukla aynı anlamda kullanılır. Bilgisayar biliminde ise aralarında açık bir fark vardır ve bu fark bütün BT sistemlerinin varlık nedenini açıklar.

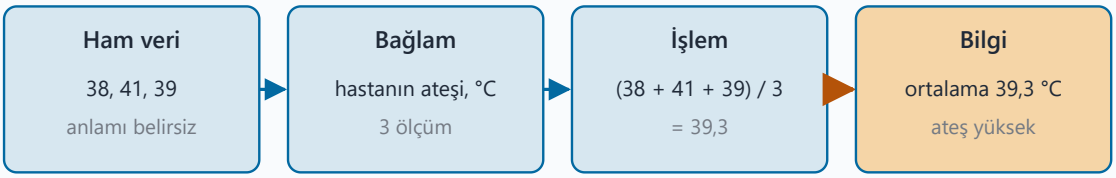
Tanım 1.1.21

Veri, ham ölçüm, sayı, metin ya da kayıttır; bağlam olmadan anlamı belirsiz kalabilir. **Bilgi**, verinin düzenlenmesi, işlenmesi ve bir bağlam içinde yorumlanmasıyla ortaya çıkan, karar vermeye yarayan anlamlı sonuçtur.

Örnek 1.1.22

Bir kâğıtta yalnızca şu üç sayı yazıyor: 38, 41, 39. Bu sayılar nasıl bilgiye dönüşür?

1. **Ham veri.** 38, 41, 39 yalnızca sayılardır. Ne ölçüldüğü bilinmedikçe anlamları yoktur: yaş, sınav puanı ya da hız olabilirler.
2. **Bağlam.** Bunların bir hastanın gün içindeki üç ateş ölçümü olduğu, birimin °C olduğu eklenir. Artık sayıların neyi gösterdiği bellidir.
3. **İşlem.** Ortalama hesaplanır: *(38 + 41 + 39) / 3 = 118 / 3 ≈ 39,3*.
4. **Bilgi.** “Hastanın ortalama ateşi 39,3 °C'dir; ateş yüksektir.” Bu cümleye dayanarak karar verilebilir: hastaya ateş düşürücü verilir.



Şekil 1.1.28: Veriden bilgiye dört adım: ham veri (38, 41, 39; anlamı belirsiz), bağlam (hastanın ateşi, °C, 3 ölçüm), işlem (ortalama 39,3) ve bilgi (ortalama 39,3 °C, ateş yüksek).

Şekil 1.1.28 dönüşümün adımlarını tek bir çizgide toplar. Bilgisayar bu adımlardan yalnızca işlem kısmını yapar: toplar ve böler. Sayıların ateş ölçümü olduğunu, 39,3 °C'nin "yüksek" sayılması gerektiğini bilen insandır. Bağlamı ekleyen ve sonucu yorumlayan insan olmadan bilgisayarın çıktısı yine yalnızca bir sayıdır. Aynı 39,3 sayısı bir oda sıcaklığı olsaydı yorum bambaşka olurdu.

Sık yapılan hata

Ne kadar çok veri toplanırsa o kadar çok bilgi elde edileceğini sanmak. Bağlamı belirsiz, düzensiz ya da hatalı veri, ne kadar çok olursa olsun bilgi vermez; yanlış yorumlara bile yol açabilir. Bilgi, verinin miktarından değil, doğru düzenlenip yorumlanmasından doğar.

1.1.6.3 Günlük yaşamda bilgi teknolojileri

Bilgi teknolojileri bugün hemen her alanda gündelik işlerin bir parçasıdır. Birkaç örnek, aynı çalışma döngüsünün farklı alanlarda nasıl göründüğünü gösterir.

- **Eğitim:** çevrim içi dersler, sayısal ders kaynakları ve öğrenme platformları. Bir öğrenci bilgi sistemi notları toplar (giriş), ortalamaları hesaplar (işlem), transkripti gösterir (çıkış) ve kayıtları yıllarca saklar (depolama).
- **Sağlık:** elektronik hasta kayıtları, görüntüleme sistemleri (röntgen, MR) ve hasta takibi. Bir hastanın tahlil sonuçları farklı hastanelerden aynı kayda erişilebilir.
- **Finans:** bankacılık, ödeme sistemleri ve işlem güvenliği. Bir kart ödemesi saniyeler içinde birkaç ülkedeki bilgisayarlardan geçerek onaylanır.
- **İş ve eğlence:** kurumsal iletişim, iş süreçlerinin otomasyonu, oyunlar ve medya hizmetleri. Bir müzik uygulaması dinlenen şarkılardan yola çıkarak yeni şarkılar önerir.

Bu alanların hepsinde BT'nin katkısı benzerdir: veriyi daha hızlı toplamak, daha doğru işlemek, daha uzun süre saklamak ve gereken kişiye daha çabuk ulaştırmak.

1.1.6.4 Veri yaşam döngüsü

Veri bir BT sisteminde rastgele dolaşmaz; belirli aşamalardan geçer. Bu aşamaların bütününe **veri yaşam döngüsü** (*data life cycle*) denir.

Tanım 1.1.22

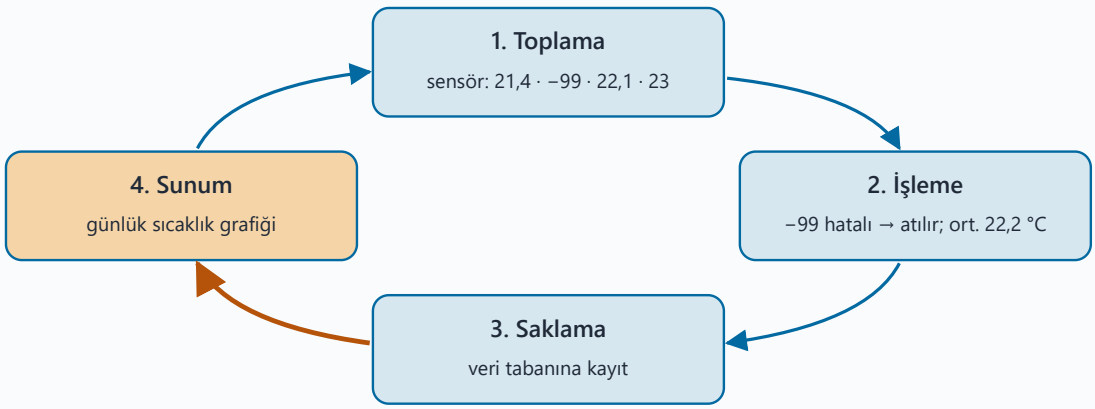
Veri yaşam döngüsünün dört aşaması vardır. **Toplama**: veri insanlardan, makinelerden ya da sensörlerden alınır. **İşleme**: hatalı veri ayıklanır, hesaplar yapılır. **Saklama**: veri dosyada, veri tabanında ya da bulutta korunur. **Sunum**: sonuç rapor, tablo, grafik ya da uygulama ekranıyla paylaşılır. Sunulan sonuca göre verilen kararlar yeni veri toplanmasını başlatır ve döngü yeniden döner.

Örnek 1.1.23

Bir hava istasyonunun sensörü dört sıcaklık ölçümü gönderiyor: 21,4 · −99 · 22,1 · 23 (°C). Sensör, ölçüm yapamadığında −99 değerini "arıza" kodu olarak gönderir.

1. **Toplama.** Dört değer sensörden alınır.
2. **İşleme.** −99 gerçek bir sıcaklık değil, arıza kodudur; atılır. Kalan üç ölçümün ortalaması alınır: *(21,4 + 22,1 + 23) / 3 = 66,5 / 3 ≈ 22,17*, yani yaklaşık 22,2 °C.
3. **Saklama.** Ölçümler ve ortalama, zaman bilgisiyle birlikte veri tabanına kaydedilir.
4. **Sunum.** Günlük sıcaklık grafiği bir web sayfasında yayımlanır. Grafiği izleyen bir çiftçi sulama saatini buna göre ayarlar; bu karar yeni ölçümlerin toplanmasıyla sınanır.

−99 atılmasaydı ortalama *(21,4 − 99 + 22,1 + 23) / 4 = −32,5 / 4 ≈ −8,1* °C çıkardı. Bu sayı yaz ortasında hiçbir anlam taşımaz. İşleme aşamasındaki ayıklama, verinin bilgiye dönüşmesi için şarttır.



Şekil 1.1.29: Hava istasyonu örneğinde veri yaşam döngüsü: toplama (sensör: 21,4 · −99 · 22,1 · 23), işleme (−99 atılır; ortalama 22,2 °C), saklama (veri tabanına kayıt) ve sunum (günlük sıcaklık grafiği).

Şekil 1.1.29 döngünün kapalı bir halka olduğunu gösterir. Veri yaşam döngüsü, bilgisayarın giriş–işlem–çıkış–depolama döngüsüne çok benzer; ama ölçeği farklıdır. Bilgisayarın döngüsü tek bir makinede saniyenin binde biri içinde döner. Veri yaşam döngüsü ise bir kurumun bütün sistemlerine yayı-

lır ve günler, aylar, hatta yıllar sürebilir.

> **Sık yapılan hata**
>
> Hatalı veriyi "yok saymak" yerine sıfır yazmak. −99 atılacağına 0 olarak hesaba katılırsa ortalama *(21,4 + 0 + 22,1 + 23) / 4 ≈ 16,6* °C çıkar ve yine yanlıştır. Eksik ya da hatalı bir ölçüm ortalamaya hiç girmemelidir; bölünen sayı da ölçüm sayısı değil, geçerli ölçüm sayısıdır.

1.1.6.5 Avantajlar ve sorumluluklar

Bilgi teknolojileri önemli kolaylıklar getirir:

- **Hız ve otomasyon:** veriler insan eliyle yapılabilecek olandan milyonlarca kat hızlı işlenir; tekrarlanan görevler, örneğin bütün öğrencilerin ortalamalarının hesaplanması, kendiliğinden yapılır.
- **Erişim ve iletişim:** bilgiye farklı yerlerden ulaşmak ve onu paylaşmak kolaylaşır; bir doktor başka bir şehirdeki hastanın tahlillerini görebilir.

Aynı kolaylıklar yeni sorumluluklar da doğurur:

- **Mahremiyet ve güvenlik:** kişisel veriler yetkisiz erişime ve kayba karşı korunmalıdır. Bir sağlık kaydının ya da banka bilgisinin sızması, kişiye uzun süreli zarar verir. Türkiye'de kişisel verilerin işlenmesi 6698 sayılı Kişisel Verilerin Korunması Kanunu (KVKK) ile düzenlenir.
- **Doğruluk ve etik:** yanlış veri ve önyargılı sistemler hatalı sonuç üretebilir. Geçmişteki önyargılı kararlardan öğrenen bir yazılım, aynı önyargıyı hızla ve büyük ölçekte tekrarlayabilir.

Güvenlik bu yüzden yalnızca teknik bir konu değildir. En güçlü şifreleme bile, kullanıcı şifresini bir sahte e-postaya yazıyorsa koruma sağlamaz. Bir BT sisteminin güvenliği; donanım ve yazılım kadar, onu kullanan insanların bilgisine, kurumun kurallarına ve bu kuralların ne kadar uygulandığına da bağlıdır. Teknolojik kolaylık, veriyi doğru, güvenli ve sorumlu kullanma yükümlülüğünü ortadan kaldırmaz.

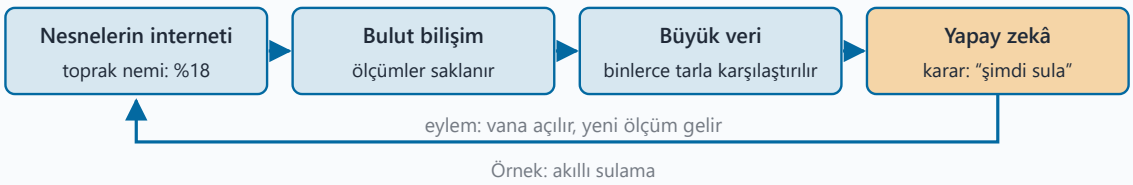
1.1.6.6 Bilgi teknolojilerinin geleceği

Bugün bilgi teknolojilerinin gelişimini dört alan belirgin biçimde yönlendirir.

Alan	Kısa tanım
Yapay zekâ	Veriden örüntü öğrenen karar destek ve üretim sistemleri
Nesnelerin interneti	Sensörlü cihazların ağ üzerinden veri paylaşması
Bulut bilişim	Hesaplama ve depolama kaynaklarının ağ üzerinden sağlanması
Büyük veri analitiği	Yüksek hacimli ve çeşitli verilerden anlamlı sonuç çıkarılması

Tablo 1.1.18: Bilgi teknolojilerinde öne çıkan dört alan.

Bu dört alan çoğunlukla tek bir sistemde birlikte çalışır. Şekil 1.1.30 bunu akıllı bir sulama sistemiyle örnekler.



Şekil 1.1.30: Akıllı sulama: nesnelerin interneti sensörü toprak nemini ölçer (%18), bulut ölçümleri saklar, büyük veri analitiği binlerce tarlayı karşılaştırır, yapay zekâ "şimdi sula" kararını verir; vana açılır ve yeni ölçüm gelir.

Tarladaki sensör, nesnelerin internetinin bir parçasıdır: toprak nemini ölçer ve ağ üzerinden gönderir. Ölçümler bulutta, yani çiftçinin kendi bilgisayarında değil, ağ üzerinden kiralanan sunucularda saklanır. Büyük veri analitiği, aynı bölgedeki binlerce tarlanın nem, hava ve ürün verilerini karşılaştırır. Yapay zekâ bu karşılaştırmadan öğrendiklerine dayanarak "şimdi sula" kararını verir ve vana açılır. Sulamadan sonra yeni ölçümler gelir ve döngü yeniden başlar. Bu sistem de özünde veri yaşam döngüsünün ve bilgisayarın temel çalışma döngüsünün büyük ölçekli bir uygulamasıdır.

Yeni teknolojiler yeni soruları da beraberinde getirir. Veri merkezlerinin harcadığı enerji giderek artıyor. Teknolojiye erişimi olmayanlar, hizmetlerin sayısallaşmasıyla geride kalabilir. Daha çok cihazın ağa bağlanması saldırı yüzeyini genişletir. Yapay zekânın verdiği kararların hesabını kimin vereceği hâlâ tartışılır. Bilgi teknolojilerinin geleceği yalnızca teknik ilerlemeyle değil; enerji, erişilebilirlik, güvenlik ve etik sorularına verilecek cevaplarla da şekillenecektir.

1.1.7 Özet

- **Bilgisayar**, aldığı verileri program komutlarına göre işleyen elektronik bir sistemdir. Her işte aynı dört aşamayı kullanır: **giriş**, **işlem**, **çıkış** ve **depolama**. Depolama işlemle iki yönlü konuşur: sonuç yazılır, gerektiğinde geri okunur.
- Giriş birimleri fiziksel olayları bitlere çevirir. İşlemcide **kontrol birimi** komut sırasını yönetir, **ALU** hesap ve karşılaştırma yapar, **yazmaçlar** üzerinde çalışılan değerleri tutar.
- **RAM** hızlı ama geçicidir; güç kesilince içeriği silinir. **SSD** ve sabit disk kalıcıdır. "Kaydet", belgeyi RAM'den kalıcı depolamaya kopyalar.
- Bilgisayarda bütün veri türleri **bit** dizileriyle temsil edilir; 8 bit 1 **bayt** eder. Bir bit dizisinin anlamını, onu okuyan kural (kodlama) belirler.
- n bit 2^n farklı değer gösterir (işaretsiz $0 \ldots 2^n - 1$). N değer için $2^n \geq N$ olan en küçük n gerekir: 58 harf için 6 bit.

- İkiliden onluğa dönüşümde 1 olan bitlerin **basamak değerleri** toplanır. Onluktan ikiliye dönüşümde sayı art arda 2'ye bölünür ve kalanlar sondan başa okunur: $45 = 101101_2$. **Onaltılık** rakamların her biri dört bite karşılık gelir: $45 = 2D_{16}$.
- Transistör iki durumlu bir anahtardır; iletim 1, kesim 0 olarak yorumlanır. İki durumu ayırt etmek güvenilir olduğu için bilgisayarlar ikili çalışır.
- Metin **ASCII** ya da **Unicode** kodlarıyla, görüntü **RGB** bileşenli piksellerle saklanır. Sıkıştırılmamış görüntü boyutu = genişlik × yükseklik × bit derinliği ÷ 8 bayttır.
- Ses **örnekleme**, **nicemleme** ve **kodlama** ile sayıya dönüşür; boyut = örnekleme hızı × bit derinliği × kanal × süredir. CD kalitesinde bir dakika ≈ 10,1 MB'dir. Video, saniyede belirli sayıda (fps) gösterilen karelerden oluşur.
- Sıkıştırma **kayıpsız** (ZIP, PNG; özgün veri aynen geri gelir) ya da **kayıplı** (JPEG, MP3; ayrıntı atılır) olabilir. **RLE** art arda tekrarları (uzunluk, değer) çiftleriyle yazar; uzun tekrarlarda küçültür, sık değişen veride büyütür.
- Bayttan sonraki her birim öncekinin 1.024 katıdır (KiB, MiB, GiB); disk üreticileri ise ondalık birim kullanır, bu yüzden 1 TB ≈ 931 GiB görünür. Ağ hızı **bit/s**, dosya boyutu **bayt** ile ölçülür; 100 Mbps = 12,5 MB/s'dir.
- **AND** iki koşulun ikisini de, **OR** en az birini ister; **NOT** koşulu tersine çevirir. Aritmetik, karşılaştırma, mantıksal işlem ve karar bir zincir hâlinde çalışır.
- **Parite biti** tek sayıda bit hatasını yakalar ama yerini bulamaz; çift sayıda hata gözden kaçar.
- Bilgisayar türlerini kullanım amacı, kapasite ve çalışma ortamı ayırır: süper bilgisayar hız, ana bilgisayar süreklilik, sunucu ağ hizmeti, gömülü sistem tek bir cihaz görevi için tasarlanır. Kuantum bilgisayar belirli problemler için farklı bir araçtır.
- **Bilgi teknolojileri** donanım, yazılım, veri ve iletişimle birlikte insanlardan, süreçlerden ve güvenlik kurallarından oluşur. Bilgisayar veriyi işler; bağlamı ekleyip **bilgiye** dönüştüren insandır. Veri; toplama, işleme, saklama ve sunum aşamalarından oluşan bir **yaşam döngüsünden** geçer.

Soru	Kural	Örnek
n bit kaç değer?	2^n	8 bit → 256
N değer için kaç bit?	$2^n \geq N$ olan en küçük n	58 → 6 bit
İkiliden onluğa	1 olan bitlerin basamak değerlerinin toplamı	$01000001_2 = 64 + 1 = 65$
Onluktan ikiliye	Art arda 2'ye bölme; kalanlar sondan başa okunur	$45 = 101101_2$
İkiliden onaltılığa	Sağdan dörderli gruplama	$0010\ 1101_2 = 2D_{16}$
Görüntü boyutu	genişlik × yükseklik × bit derinliği ÷ 8	800 × 600, 24 bit → 1.440.000 bayt
Ses boyutu	örnekleme hızı × bit derinliği × kanal × süre ÷ 8	CD, 1 dk → 10.584.000 bayt
İndirme süresi	boyut (bit) ÷ hız (bit/s)	3 GB, 100 Mbps → 240 s

Tablo 1.1.19: Notta kullanılan hesap kuralları.

1.1.8 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

bilgisayar *computer* · veri *data* · bilgi *information* · giriş birimi *input unit* · çıkış birimi *output unit* · depolama *storage* · merkezî işlem birimi (CPU) *central processing unit* · kontrol birimi *control unit* · aritmetik-mantık birimi (ALU) *arithmetic logic unit* · yazmaç *register* · RAM (rastgele erişimli bellek) *random access memory* · geçici / kalıcı *volatile / non-volatile* · bit *bit* · bayt *byte* · ikili *binary* · onluk *decimal* · onaltılık *hexadecimal* · basamak / basamak değeri *digit / place value* · transistör *transistor* · karakter kümesi *character set* · ASCII *ASCII* · Unicode *Unicode* · kodlama *encoding* · piksel *pixel* · çözünürlük *resolution* · renk derinliği *color depth* · örnekleme / örnek *sampling / sample* · örnekleme hızı *sampling rate* · nicemleme *quantization* · kare *frame* · kayıpsız sıkıştırma *lossless compression* · kayıplı sıkıştırma *lossy compression* · çalışma uzunluğu kodlaması (RLE) *run-length encoding* · KB / MB / GB / TB *kilobyte / megabyte / gigabyte / terabyte* · KiB / MiB / GiB *kibibyte / mebibyte / gibibyte* · aritmetik işlem *arithmetic operation* · karşılaştırma *comparison* · mantıksal işlem *logical operation* · parite biti *parity bit* · süper bilgisayar *supercomputer* · ana bilgisayar *mainframe* · sunucu *server* · istemci *client* · ince istemci *thin client* · istek / yanıt *request / response* · kişisel bilgisayar *personal computer* · mobil cihaz *mobile device* · gömülü sistem *embedded system* · mikrodenetleyici *microcontroller* · sensör *sensor* · kuantum bilgisayar *quantum computer* · qubit (kuantum biti) *qubit* · bilgi teknolojisi (BT) *information technology* · veri yaşam döngüsü *data life cycle* · mahremiyet *privacy* · yapay zekâ *artificial intelligence* · nesnelerin interneti *Internet of Things* · bulut bilişim *cloud computing* · büyük veri *big data*

1.1.9 Kendini Dene

1. 10110110_2 sayısı onluk sistemde kaçtır? Onaltılık yazılışı nedir?

Cevap

1 olan bitlerin basamak değerleri 128, 32, 16, 4 ve 2'dir: *128 + 32 + 16 + 4 + 2 = 182*. Dörderli gruplar 1011 = B ve 0110 = 6'dır; onaltılık yazılış $B6_{16}$'dır. Denetim: *11 × 16 + 6 = 182*.

2. 100 sayısı 2'ye bölme yöntemiyle ikiliye çevrilince hangi dizi elde edilir? Sonuç basamak değerleriyle nasıl denetlenir?

Cevap

100 ÷ 2 = 50 kalan 0; 50 ÷ 2 = 25 kalan 0; 25 ÷ 2 = 12 kalan 1; 12 ÷ 2 = 6 kalan 0; 6 ÷ 2 = 3 kalan 0; 3 ÷ 2 = 1 kalan 1; 1 ÷ 2 = 0 kalan 1. Kalanlar sondan başa okunur: 1100100_2. Denetim: *64 + 32 + 4 = 100*.

3. Bir fakültede 0 ile 99 arasında numaralanmış 100 laboratuvar dolabı var. Her dolabın numarasını ikili kodlamak için en az kaç bit gerekir? Kaç kod boşta kalır?

Cevap

Kodlanacak değer sayısı 100'dür. $2^6 = 64 < 100$ olduğu için 6 bit yetmez; $2^7 = 128 \geq 100$ olduğu için 7 bit gerekir. *128 − 100 = 28* kod boşta kalır.

4. 640 × 480 piksellik, 24 bit renkli bir görüntü sıkıştırılmadan kaç bayt tutar? Kaç KB eder (1 KB = 1.024 bayt)? Aynı görüntü 8 bitlik gri tonlamaya çevrilirse boyut ne olur?

Cevap

Piksel sayısı *640 × 480 = 307.200*. 24 bit = 3 bayt olduğundan *307.200 × 3 = 921.600* bayt; *921.600 ÷ 1.024 = 900* KB. 8 bitte piksel başına 1 bayt kalır: 307.200 bayt = 300 KB, yani üçte biri.

5. 44.100 Hz örnekleme hızı ve 16 bit derinlikle yapılmış 30 saniyelik *mono* (tek kanallı) bir ses kaydı kaç bayt tutar?

Cevap

44.100 × 16 × 1 × 30 = 21.168.000 bit; ÷ 8 = 2.646.000 bayt. 1 MB = 1.048.576 bayt alınırsa ≈ 2,52 MB. Kayıt stereo olsaydı boyut iki katına, 5.292.000 bayta çıkardı.

6. BBBMMMMMMMBB satırı (B = beyaz, M = mavi; her piksel 1 bayt) RLE ile nasıl kodlanır? Kaç bayt kazanılır?

Cevap

Koşular üç beyaz, yedi mavi ve iki beyazdır: (3, B) (7, M) (2, B). Üç çift × 2 bayt = 6 bayt; özgün satır 12 bayttır. 6 bayt kazanılır ve satır yarıya iner. Uzunlukların toplamı *3 + 7 + 2 = 12*'dir.

7. Yedi veri biti 1101000, çift pariteyle gönderiliyor. Parite biti nedir? Alıcıya 11000001 ulaşırsa alıcı

ne karar verir?

Cevap

1101000'de üç tane 1 vardır; toplamı çift yapmak için parite biti 1'dir ve gönderilen dizi 11010001'dir. Alınan 11000001'de üç tane 1 vardır; sayı tektir, alıcı hata olduğunu anlar ve veriyi yeniden ister. (Gerçekte 4. bit 1'den 0'a dönmüştür; alıcı bunun yerini bilemez.)

8. 50 Mbps hızındaki bir bağlantıyla 1,5 GB'lık ($1{,}5 \times 10^9$ bayt) bir dosya en iyi durumda kaç saniyede iner?

Cevap

Dosya bite çevrilir: $1{,}5 \times 10^9 \times 8 = 12 \times 10^9$ bit. Hız 50×10^6 bit/s'dir. Süre $12 \times 10^9 \div (50 \times 10^6) = 240$ s = 4 dakika. Hız bayta çevrilerek de bulunur: 50 Mbps = 6,25 MB/s; 1.500 MB ÷ 6,25 = 240 s.

9. Kutusunda "500 GB" yazan bir disk işletim sisteminde yaklaşık kaç GiB görünür?

Cevap

Üretici ondalık birim kullanır: 500×10^9 bayt. İşletim sistemi bunu 2^{30} = 1.073.741.824 bayta böler: ≈ 465,7 GiB. Bayt kaybı yoktur; fark birim tanımından gelir.

10. Bir seranın havalandırması (sıcaklık > 28) AND (nem < 60) koşuluyla açılıyor. Sıcaklık 30 °C, nem %70 iken koşulun değeri nedir? AND yerine OR yazılsaydı ne olurdu?

Cevap

30 > 28 doğrudur (1); 70 < 60 yanlıştır (0). 1 AND 0 = 0: havalandırma açılmaz. OR ile 1 OR 0 = 1 olur ve havalandırma açılır. Aynı veriyle bağlacın seçimi kararı değiştirir.

1.1.10 İleri okuma

- J. G. Brookshear ve D. Brylow, *Computer Science: An Overview* (13. baskı, Pearson, 2019), Bölüm 1: bitlerin depolanması, ikili sistem, metin, görüntü ve sesin temsili, veri sıkıştırma ve iletim hataları bu nottakine çok yakın bir sırayla ve bol örnekle anlatılır.
- W. Stallings, *Computer Organization and Architecture* (11. baskı, Pearson, 2019), Bölüm 1–2: bilgisayarın temel yapısı, işlemcinin parçaları ve bilgisayarların tarihsel gelişimi daha ayrıntılı bir donanım bakışıyla verilir.
- C. Petzold, *Code: The Hidden Language of Computer Hardware and Software* (2. baskı, Microsoft Press, 2022): mors alfabesinden ve ampullerden başlayarak ikili sayıları, mantık kapılarını ve işlemciyi teknik bilgi gerektirmeden kurar.

1.1.11 Kaynaklar

1. J. G. Brookshear, D. Brylow, *Computer Science: An Overview*, Pearson.
2. W. Stallings, *Computer Organization and Architecture*, Pearson.
3. The Unicode Consortium, *The Unicode Standard*.
4. NIST, *Glossary of Key Information Security Terms* (NISTIR 7298).
5. IEC 80000-13, *Quantities and units — Part 13: Information science and technology* (ikili ön ekler: kibi, mebi, gibi).
6. 6698 sayılı Kişisel Verilerin Korunması Kanunu, Resmî Gazete, 7 Nisan 2016.

KISIM I · BÖLÜM 1.2

Bilgisayar Donanımı

Bir bilgisayarda yazılan her program, sonunda elle tutulur parçaların üzerinde çalışır. İşlemci komutları yürütür, bellek o anda kullanılan veriyi tutar, depolama birimi veriyi kalıcı saklar; klavye ve ekran ise kullanıcıyla makine arasındaki köprüyü kurar. Bu not, bu parçaların her birinin ne iş yaptığını, birbirleriyle nasıl haberleştiğini ve bir bilgisayar seçerken hangi sayılara bakılması gerektiğini anlatır. Konuların çoğu küçük hesaplarla desteklenir. Bunlar arasında saat hızından bir turun süresi, önbelleğin kazandırdığı zaman, bir ekranın piksel yoğunluğu, bir dosyanın kaç saniyede kopyalanacağı, güç kaynağının kaç watt olması gerektiği vardır. Bu hesaplar, teknik belirtimlerdeki sayıların ne anlama geldiğini görmeyi kolaylaştırır.

ÖĞRENME ÇIKTILARI

Bileşenler
Temel donanım parçalarının (işlemci, anakart, RAM, ROM, ekran kartı, güç kaynağı, kasa) görevlerini açıklamak.

Karşılaştırma
RAM, ROM, HDD ve SSD arasındaki farkları bellek hiyerarşisi içinde ayırt etmek; önbelleğin neden hız kazandırdığını isabet ve ıska sayarak göstermek.

Aygıt seçimi
Giriş ve çıkış birimlerini kullanım amacına göre seçmek; çözünürlük, piksel yoğunluğu, yenileme hızı ve depolama hızı gibi değerleri hesaplayıp yorumlamak.

Uyumluluk
Bileşenlerin bağlantı, güç, fiziksel boyut ve ısı açısından birbirine uyup uymadığını değerlendirmek.

1.2.1 Donanım nedir?

Bilgisayar denince çoğu kişinin aklına önce ekran, klavye ve kasa gelir. Bunlar bilgisayarın görünen yüzüdür. Kasanın içinde ise işlemci, bellek modülleri, depolama birimi, güç kaynağı ve bütün bunları birbirine bağlayan anakart bulunur. Bu kesim, bu parçaların genel resmini çizer. Her parça ilerleyen kesimlerde ayrıca ele alınır.

1.2.1.1 Donanım ve yazılım

Bir bilgisayar sistemi iki katmandan oluşur. Birinci katman, dokunulabilen, takılıp çıkarılabilen, bozulunca değiştirilen fiziksel parçalardır. İkinci katman ise bu parçalara ne yapacaklarını söyleyen komutlardır. Birinci katmana donanım, ikinci katmana yazılım denir.

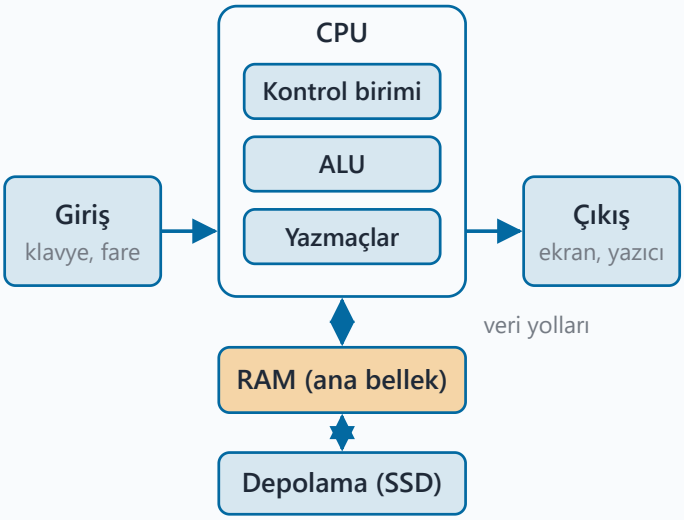
Tanım 1.2.1

Donanım (*hardware*), bilgisayar sisteminin fiziksel parçalarının tamamıdır: işlemci, bellek, depolama birimleri, anakart, güç kaynağı, kasa, giriş ve çıkış birimleri ile bunları bağlayan kablolar. **Yazılım** ise bu parçaların hangi işlemleri hangi sırayla yapacağını belirleyen programlardır.

Donanım ile yazılım birbirini tamamlar. Yazılım olmadan donanım hiçbir iş yapmaz; en hızlı işlemci bile kendisine verilen komut olmadan bekler. Donanım olmadan da yazılım çalışamaz, çünkü komutları yürütecek, veriyi tutacak ve sonucu gösterecek bir fiziksel ortam gerekir. Bir müzik çalarla bir şarkı arasındaki ilişkiye benzer: çalar donanımdır, şarkının notaları ise yazılımdır.

Bir bilgisayarın yaptığı her iş beş temel adımdan oluşur. Şekil 1.2.1 bu adımları ve her adımı üstlenen donanım parçasını gösterir:

- **Giriş birimleri** veriyi sisteme alır. Klavyede basılan bir tuş, farenin hareketi, mikrofona gelen ses birer giriştir.
- **CPU** (merkezî işlem birimi) komutları işler: toplar, karşılaştırır, karar verir.
- **RAM** çalışan programı ve o anda kullanılan veriyi tutar. İşlemci veriyi buradan okur, sonucu buraya yazar.
- **Depolama** birimi (SSD ya da HDD) veriyi kalıcı saklar. Bilgisayar kapatıldığında da içerik korunur.
- **Çıkış birimleri** sonucu kullanıcıya sunar: ekranda bir görüntü, hoparlörden bir ses, yazıcıdan bir sayfa.



Şekil 1.2.1: Bir bilgisayarda verinin akışı. Giriş birimlerinden gelen veri CPU'ya ulaşır. CPU'nun içinde kontrol birimi, aritmetik-mantık birimi (ALU) ve yazmaçlar bulunur. CPU, RAM ile sürekli veri alışverişi yapar; RAM de depolama birimiyle iki yönlü çalışır. Sonuç çıkış birimlerine gönderilir. Parçalar arasındaki bağlantılara veri yolu denir.

Şekil 1.2.1'deki çift yönlü oklar önemlidir. CPU ile RAM arasındaki trafik iki yönlüdür: işlemci RAM'den hem komut hem veri okur, işlem sonuçlarını da RAM'e yazar. RAM ile depolama arasında da veri iki yönde akar. Bir program açılırken dosyası depolamadan RAM'e kopyalanır; bir belge kaydedilirken değişiklikler RAM'den depolamaya yazılır. Bu akış, 1.2.9.1'de bir belgenin açılıp kaydedilmesi üzerinde adım adım izlenir.

Örnek 1.2.1

Bir hesap makinesi programında `12 + 30` yazılıp Enter tuşuna basılıyor. Beş adım şöyle işler:

1. **Giriş.** Klavye her tuş basışını bir sayı koduna çevirip bilgisayara gönderir: önce `1`, `2`, sonra `+`, `3`, `0` ve Enter.
2. **Bellek.** Hesap makinesi programı zaten RAM'dedir. Basılan tuşlar da programın RAM'deki değişkenlerine yazılır.
3. **İşlem.** Enter gelince CPU, programın toplama komutunu yürütür. 12 ve 30 sayılarını RAM'den yazmaçlara alır, ALU ile toplar ve 42 sonucunu bulur.
4. **Çıkış.** Program 42'yi ekrana yazdırmak için ekran kartına gönderir; monitör sonucu gösterir.
5. **Depolama.** Bu örnekte kalıcı kayıt gerekmez. Program geçmiş hesapları bir dosyada tutuyorsa, sonuç ayrıca SSD'ye yazılır.

Görüldüğü gibi, en basit işlem bile birkaç donanım parçasının birlikte çalışmasını gerektirir.

Sık yapılan hata

"Bellek" ile "depolama"yı aynı şey sanmak. Günlük dilde ikisine de "hafıza" denebilir, ama görevleri farklıdır. RAM, programın *çalışırken* kullandığı geçici alandır ve elektrik kesilince boşalır. Depolama (SSD, HDD) ise dosyaların *kalıcı* durduğu yerdir. "Bilgisayarın 16 GB belleği, 512 GB depolaması var" cümlesinde iki ayrı parçadan söz edilir.

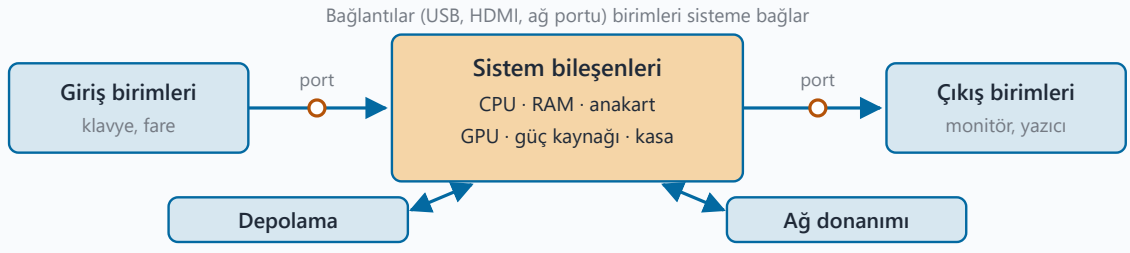
1.2.1.2 Donanım grupları

Bilgisayar donanımı, görevlerine göre altı grupta toplanabilir. Bu gruplama, bir parçanın sistemde hangi rolü üstlendiğini hızla anlamaya yarar.

Grup	Örnekler	Görevi
Sistem bileşenleri	CPU, anakart, RAM, GPU, güç kaynağı, kasa	komutları işler, parçaları bağlar ve besler
Giriş birimleri	klavye, fare, mikrofon, kamera, sensörler	veriyi sisteme alır
Çıkış birimleri	monitör, yazıcı, hoparlör, projektör	sonucu kullanıcıya sunar
Depolama	HDD, SSD, USB bellek, hafıza kartı, optik disk	veriyi kalıcı saklar
Ağ donanımı	ağ kartı, switch, yönlendirici, erişim noktası	bilgisayarları birbirine bağlar
Bağlantılar	USB, HDMI, DisplayPort, ses ve ağ portları	birimleri sisteme fiziksel olarak bağlar

Tablo 1.2.1: Donanım grupları ve örnekler.

Ağ donanımındaki bazı adlar kısaca açıklanmalıdır. **Ağ kartı** (*network interface card*, NIC) bilgisayarın kablolu ya da kablosuz ağa bağlanmasını sağlar. **Switch** (anahtar), aynı yerel ağdaki aygıtları birbirine bağlayan kutudur. **Yönlendirici** (*router*) farklı ağlar arasında, örneğin ev ağı ile internet arasında, trafiği yönlendirir. **Erişim noktası** (*access point*) ise kablosuz aygıtların kablolu ağa katılmasını sağlar.



Şekil 1.2.2: Donanım gruplarının ilişkisi. Giriş birimleri sistem bileşenlerine veri verir; sistem bileşenleri sonucu çıkış birimlerine gönderir. Depolama ve ağ donanımı sistemle iki yönlü çalışır. Birimler sisteme portlar üzerinden bağlanır.

Şekil 1.2.2 gruplar arasındaki veri akışını özetler. Giriş ve çıkış birimleri tek yönlüdür: biri veriyi içeri alır, öteki dışarı verir. Depolama ve ağ donanımı ise hem okur hem yazar. Bir SSD'den dosya okunur, aynı SSD'ye dosya yazılır. Ağ kartı da hem veri alır hem veri gönderir. Bu ayrım, 1.2.8.1'deki giriş-çıkış sınıflandırmasının temelidir.

1.2.2 Merkezî işlem birimi

İşlemci, bilgisayarın komutları yürüten parçasıdır. Bir programın her satırı, sonunda işlemcinin anladığı çok basit komutlara dönüşür: iki sayıyı topla, bir değeri bellekten oku, iki değeri karşılaştır, sonuca göre başka bir komuta atla. İşlemcinin gücü, bu basit komutları saniyede milyarlarca kez yürütebilmesinden gelir.

1.2.2.1 İşlemcinin görevi: getir, çöz, yürüt

Tanım 1.2.2

Merkezî işlem birimi (*central processing unit*, CPU) ya da kısaca **işlemci**, program komutlarını bellekten getiren, ne anlama geldiklerini çözen ve yürüten donanım birimidir. Bu üç adımlık döngüye **getir–çöz–yürüt** döngüsü denir.

Döngünün adımları şöyledir:

1. **Getir.** İşlemci, sıradaki komutu bellekten alır. Hangi komutun sırada olduğunu özel bir yazmaç tutar.
2. **Çöz.** Komutun ne istediği belirlenir: toplama mı, karşılaştırma mı, bellekten okuma mı? Hangi değerler kullanılacak?
3. **Yürüt.** İstenen işlem yapılır ve sonuç bir yazmaca ya da belleğe yazılır. Ardından döngü bir sonraki komutla yeniden başlar.

Bu döngüyü işlemcinin içindeki üç parça yürütür:

- **Aritmetik-mantık birimi** (*arithmetic logic unit*, ALU) aritmetik işlemleri (toplama, çıkarma) ve mantıksal işlemleri (karşılaştırma, “ve”, “veya”) yapar.
- **Kontrol birimi** komutların sırasını ve veri akışını yönetir. Hangi verinin ALU'ya gideceğine, sonucun nereye yazılacağına o karar verir.
- **Yazmaçlar** (*registers*) işlemcinin içindeki çok küçük ve çok hızlı saklama hücreleridir. ALU'nun o anda üzerinde çalıştığı değerler yazmaçlarda durur.



Şekil 1.2.3: Dört çekirdekli bir işlemci yongası. Her çekirdeğin kendi kontrol birimi, ALU'su, yazmaçları ve L1/L2 önbelleği vardır. L3 önbellek bütün çekirdeklerce ortak kullanılır.

Günümüz işlemcileri tek yongada birden çok **çekirdek** barındırır (Şekil 1.2.3). Her çekirdek, getir–çöz–yürüt döngüsünü kendi başına işletebilen eksiksiz bir işlem birimidir. Şekildeki **önbellek** (L1, L2, L3) ise sık kullanılan veriyi işlemcinin hemen yanında tutan hızlı bellektir; 1.2.4'te ayrıntılı ele alınır.

İşlemcinin performansını tek bir sayı belirlemez. Dört etken birlikte rol oynar: **çekirdek sayısı**, **saat hızı** (GHz), **önbellek** büyüklüğü ve işlemcinin **mimarisi**, yani iç tasarımı. Aynı saat hızındaki iki işlemciden yeni nesil olanı, her turda daha fazla iş yaptığı için belirgin biçimde hızlı olabilir.

Örnek 1.2.2

Bellekte `x = 5` ve `y = 7` değerleri var. `z = x + y` satırı işlemcide kabaca dört basit komuta dönüşür. Her komut bir getir–çöz–yürüt döngüsüyle işler:

Komut	Çöz: ne isteniyor?	Yürüt: sonuç
1	`x` 'i bellekten yazmaç R1'e oku	R1 = 5
2	`y` 'yi bellekten yazmaç R2'ye oku	R2 = 7
3	R1 ile R2'yi topla, R3'e yaz (ALU)	R3 = 12
4	R3'ü belleğe, `z` 'nin yerine yaz	z = 12

Toplamayı ALU yapar; okuma ve yazma komutlarında kontrol birimi veriyi bellek ile yazmaçlar arasında taşır. Tek satırlık bir toplama için dört döngü gerekir. Bu adımlar Basit CPU simülatöründe adım adım izlenebilir.

1.2.2.2 Saat hızı

İşlemcinin içindeki bütün işler bir ritimle ilerler. Bu ritmi bir **saat** sinyali verir: sinyal düzenli aralıklarla yükselip alçalır ve her yükselişte işlemcinin devreleri bir adım ilerler. Bir orkestrada şefin vuruşları gibi düşünülebilir; bütün çalgılar aynı vuruşa göre çalar.

Tanım 1.2.3

Saat hızı (*clock speed*), saat sinyalinin bir saniyede kaç tur attığıdır ve hertz (Hz) ile ölçülür. 1 GHz (gigahertz) saniyede 10^9, yani 1 milyar turdur. Bir turun süresine **saat turu** (*clock cycle*) denir:

> bir turun süresi = 1 ÷ saat hızı

Bu formülün nereden geldiği kolayca görülür. Saniyede 3 milyar tur atılıyorsa, 1 saniye 3 milyar eşit parçaya bölünmüştür; her parça 1/3.000.000.000 saniyedir. Bu çok küçük süreler için **nanosaniye** (ns) kullanılır: 1 ns = 10^{-9} s, yani saniyenin milyarda biridir.

Örnek 1.2.3

3 GHz ve 3,2 GHz işlemcilerde bir saat turu kaç ns sürer? 3,2 GHz işlemci her komutu ortalama 4 turda bitirseydi saniyede kaç komut yürütürdü?

1. **3 GHz.** 3 GHz = 3×10^9 tur/s. Bir tur: $1 \div (3 \times 10^9)$ s = $0{,}333\ldots \times 10^{-9}$ s ≈ **0,33 ns**.
2. **3,2 GHz.** $1 \div (3{,}2 \times 10^9)$ s = $0{,}3125 \times 10^{-9}$ s = **0,3125 ns**.
3. **Komut sayısı.** Saniyede $3{,}2 \times 10^9$ tur atılır ve her komut 4 tur sürer: $3{,}2 \times 10^9 \div 4 = 0{,}8 \times 10^9$ = **800 milyon komut/s**.

Bir turun ne kadar kısa olduğunu bir karşılaştırma gösterir: ışık bile 0,33 ns'de yalnızca 10 cm kadar yol alır.

Saat hızı yüksek olan işlemci her zaman daha hızlı değildir. Bir işlemcinin hızını, saat hızı ile her turda yapılan iş miktarının çarpımı belirler. Yeni mimariler bir turda birden çok komutu aynı anda ilerletebilir. Bu yüzden 3 GHz'lik yeni bir işlemci, 4 GHz'lik eski bir işlemciden hızlı olabilir. Üstelik saat hızı sabit değildir: işlemci boştayken enerji tasarrufu için yavaşlar, yük gelince ve sıcaklık izin verdikçe hızlanır.

Sık yapılan hata

Farklı nesil ya da farklı üreticinin işlemcilerini yalnızca GHz değerine bakarak karşılaştırmak. GHz, aynı ailedeki iki işlemciyi kıyaslarken anlamlıdır. Farklı mimarilerde bir turda yapılan iş değiştiği için karşılaştırma yanıltıcı olur; gerçek iş yükünde yapılan ölçümler daha güvenilir-

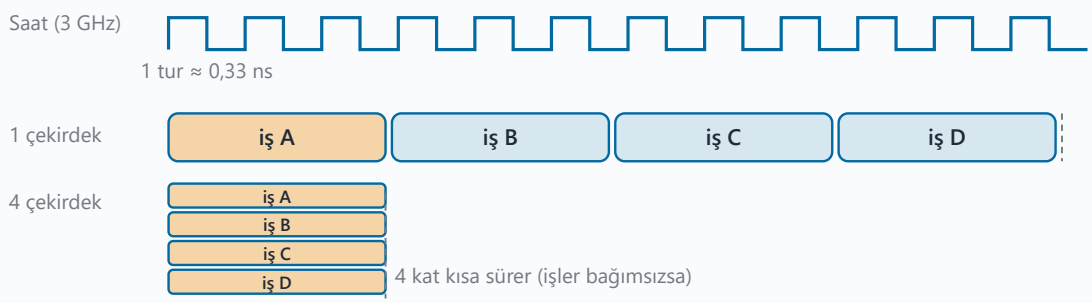
dir.

1.2.2.3 Çekirdekler

Tanım 1.2.4

Çekirdek (*core*), işlemcinin içinde bağımsız bir komut akışını yürütebilen birimdir. Çok çekirdekli bir işlemci, aynı anda birden çok komut akışını işletebilir.

Bir programın yürüttüğü komut dizisine **iş parçacığı** (*thread*) denir. Tek iş parçacıklı bir program, komutlarını tek bir sıra hâlinde yürütür; bu sırayı bölüp farklı çekirdeklere dağıtamaz. Birden çok iş parçacığına bölünmüş bir program ise çekirdekleri aynı anda kullanabilir. Bir video dönüştürme programı, filmin farklı dakikalarını farklı çekirdeklere verebilir. Bir metin düzenleyicinin harf yazma işlemi ise bölünecek kadar büyük değildir.



Şekil 1.2.4: Saat sinyali ve çekirdekler. 3 GHz'de bir tur yaklaşık 0,33 ns sürer. Dört bağımsız iş (A, B, C, D) tek çekirdekte sırayla, dört çekirdekte aynı anda yürür; işler bağımsızsa toplam süre dörtte bire iner.

Şekil 1.2.4 çok çekirdeğin ne zaman yararlı olduğunu gösterir: işler birbirinden bağımsız olmalıdır. Bir işin sonucu öbürüne girdi olarak gerekiyorsa, ikinci iş birinciyi beklemek zorundadır ve iki çekirdek birlikte çalışamaz. Aynı anda çalışan farklı programlar (tarayıcı, müzik çalar, virüs tarayıcı) ise doğal olarak bağımsızdır; işletim sistemi bunları farklı çekirdeklere dağıtır.

Örnek 1.2.4

Her biri tek çekirdekte 5 saniye süren 7 bağımsız iş var. İşler bölünemez ve bir çekirdek aynı anda tek iş yürütür. 3 çekirdekli bir işlemcide toplam süre nedir?

1. **İlk 5 saniye.** Üç çekirdeğe üç iş verilir (1, 2, 3). Dört iş bekler.
2. **5–10 saniye.** İlk üç iş biter, sıradaki üç iş (4, 5, 6) başlar. Bir iş bekler.
3. **10–15 saniye.** Yalnızca 7. iş kalmıştır. Bir çekirdek çalışır, ikisi boş bekler.

Toplam süre **15 s**'dir. Tek çekirdekte 7 × 5 = 35 s sürerdi. Hızlanma 35 ÷ 15 ≈ 2,3 kattır, 3 kat değildir; çünkü son turda iki çekirdek boş kalır. Genel kural: tur sayısı, iş sayısının çekirdek sayısına bölümünün yukarı yuvarlanmasıdır (7 ÷ 3 = 2,33 → 3 tur).

Sık yapılan hata

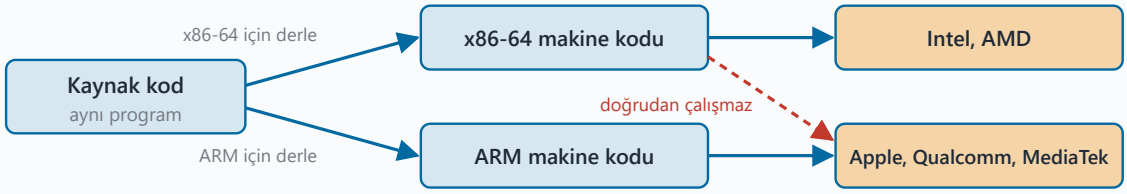
"8 çekirdek = 8 kat hız" sanmak. Tek iş parçacıklı bir program, 8 çekirdekli işlemcide de tek çekirdek kullanır ve hızlanmaz. Çekirdek sayısı ancak iş bölünebiliyorsa ya da aynı anda çok sayıda program çalışıyorsa fark yaratır.

1.2.2.4 İşlemci mimarileri ve komut kümeleri

Her işlemci ailesi belirli bir komut listesini anlar. Bir işlemcinin anladığı komutların tamamına **komut kümesi** (*instruction set*) denir. Komut kümesi, işlemci ile yazılım arasındaki ortak dildir: aynı komut kümesini anlayan iki işlemci, aynı makine kodunu çalıştırabilir. Günümüzde iki komut kümesi öne çıkar:

- **x86-64:** Masaüstü, dizüstü ve sunucu bilgisayarlarda yaygındır. Intel ve AMD işlemcileri bu komut kümesini kullanır.
- **ARM:** Telefonlarda, tabletlerde ve enerji verimliliğinin önemli olduğu sistemlerde yaygındır; giderek dizüstü bilgisayarlarda da kullanılır. Apple, Qualcomm ve MediaTek gibi şirketler ARM tabanlı işlemciler tasarlar.

Üçüncü bir seçenek olarak **RISC-V** de anılmalıdır. RISC-V açık bir komut kümesidir: kullanımı için lisans ücreti ödenmez. Bu yüzden özellikle gömülü sistemlerde ve araştırmalarda giderek yaygınlaşıyor.



Şekil 1.2.5: Aynı kaynak kod iki ayrı komut kümesi için ayrı ayrı derlenir. x86-64 makine kodu Intel ve AMD işlemcilerde, ARM makine kodu Apple, Qualcomm ve MediaTek işlemcilerde çalışır. Bir komut kümesi için üretilen kod öbüründe doğrudan çalışmaz.

Şekil 1.2.5, programcının yazdığı kaynak kod ile işlemcinin çalıştırdığı makine kodu arasındaki farkı gösterir. Kaynak kod insanların okuduğu biçimdir. **Derleyici** bu kodu belirli bir komut kümesinin makine koduna çevirir. Aynı program iki mimaride çalışacaksa iki kez derlenir. Bir telefon uygulamasının bilgisayarda doğrudan çalışmamasının nedenlerinden biri budur.

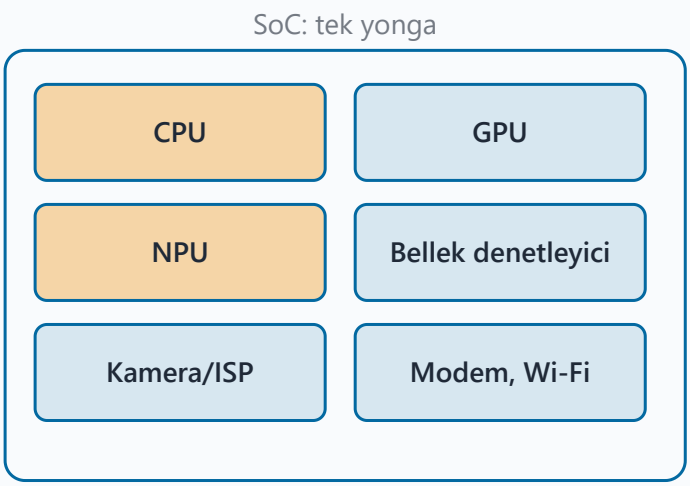
Marka ya da model adı tek başına performansı belirlemez. İki işlemci aynı üreticiden ve aynı komut kümesinden olsa bile nesilleri, güç sınırları ve çekirdek yapıları farklı olabilir. Doğru karşılaştırma, kullanılacak iş yükü üzerinden yapılır: ofis programları, oyun ya da video işleme farklı işlemci özelliklerinden yararlanır.

1.2.2.5 SoC, mobil işlemci ve NPU

Masaüstü bir bilgisayarda işlemci, ekran kartı ve diğer denetleyiciler çoğunlukla ayrı parçalardır. Telefonda ise bu kadar yer yoktur ve pil ömrü çok önemlidir. Çözüm, bütün birimleri tek bir yongada toplamaktır.

Tanım 1.2.5

Yonga üstü sistem (*System-on-Chip*, SoC), CPU, GPU, bellek denetleyici ve diğer birimlerin tek bir yonga üzerinde toplandığı tasarımdır. **Sinirsel işlem birimi** (*neural processing unit*, NPU) ise yapay zekâ hesaplarını düşük güçle hızlandırmak için tasarlanmış birimdir.



Şekil 1.2.6: Bir SoC'nin bileşenleri: CPU, GPU, NPU, bellek denetleyici, kamera görüntü işlemcisi (ISP) ve modem/Wi-Fi tek yongadadır.

Birimlerin tek yongada olması iki kazanç sağlar. Birincisi yer tasarrufudur: telefonun ve ince dizüstülerin içine sığacak kadar küçük bir parça elde edilir. İkincisi güç tasarrufudur: birimler arasındaki veri kısa yoldan gider, ayrı yongalar arasındaki uzun bağlantılar için harcanan enerji azalır. RAM çoğu SoC'de ayrı bir yonga olarak durur, ama SoC'ye çok yakın, hatta aynı paketin içine yerleştirilir. Ayrıntı üreticiden üreticiye değişir.

NPU'nun hızlandırdığı işler gündelik hayatta sık karşılaşılan işlerdir: sesli komutları tanıma, fotoğraftaki gürültüyü azaltma, yüz tanıma, anlık çeviri. Bu işler aynı türden çok sayıda küçük çarpma ve toplama içerir. CPU bunları da yapabilir, ama NPU aynı işi çok daha az enerjiyle bitirir.

Mobil işlemcilerin enerji verimliliği bazen yalnızca ARM komut kümesine bağlanır. Oysa verimlilik farkı yalnız komut kümesinden değil, tasarımdan da gelir: çekirdeklerin yapısı, üretim teknolojisi ve güç yönetimi de sonucu belirler. Bu yüzden ARM tabanlı yongalar dizüstü bilgisayarlarda da kullanılmaya başlanmıştır.

1.2.3 Anakart, RAM ve ROM

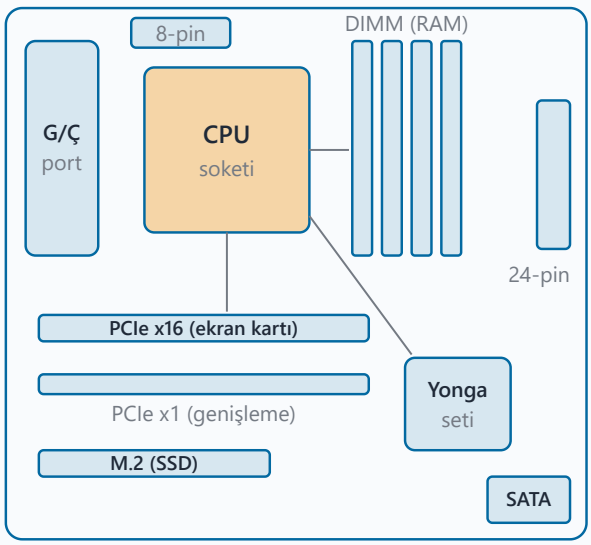
İşlemci tek başına çalışamaz. Komutlarını ve verisini bir bellekten okur, elektriğini güç kaynağından alır, sonuçlarını ekran kartına ve diğer birimlere gönderir. Bütün bu bağlantıları anakart sağlar. Bu kesimde anakart ile iki bellek türü ele alınır: çalışan programları tutan RAM ve bilgisayarın açılışını yöneten ürün yazılımının durduğu ROM.

1.2.3.1 Anakart

Tanım 1.2.6

Anakart (*motherboard*), bilgisayarın bileşenlerini fiziksel ve elektriksel olarak birleştiren ana devre kartıdır. Bileşenler, anakart üzerindeki **veri yolları** aracılığıyla birbiriyle haberleşir.

Anakart, bir şehrin yol ağına benzetilebilir. İşlemci şehrin merkezidir; bellek, depolama ve ekran kartı farklı semtlerdir. Yollar ne kadar geniş ve düzgünse trafik o kadar hızlı akar. Anakartın üzerinde her bileşen için ayrı bir yuva ya da bağlantı noktası bulunur (Şekil 1.2.7).



Şekil 1.2.7: Anakartın yerleşimi: CPU soketi, RAM için DIMM yuvaları, ekran kartı için PCIe x16 yuvası, genişleme kartları için PCIe x1 yuvası, SSD için M.2 yuvası, SATA bağlantıları, yonga seti, güç girişleri (24-pin ve 8-pin) ve arka paneldeki G/Ç portları.

Parça	Görevi
CPU soketi	İşlemcinin takıldığı yuvadır. Desteklenen işlemci ailesini belirler.
DIMM yuvaları	RAM modüllerini barındırır.
PCIe yuvaları	Ekran kartı ve diğer genişleme kartları takılır.
M.2 ve SATA	Depolama aygıtları (SSD, HDD) bağlanır.
Yonga seti	Portlar ve yuvalar arasındaki veri trafiğini yönetir.
24-pin ve 8-pin	Güç kaynağından anakarta ve işlemciye elektrik taşır.

Tablo 1.2.2: Anakart üzerindeki başlıca bağlantı noktaları.

Tablodaki kısaltmalar bilgisayar parçalarının belirtimlerinde sık geçer. **DIMM** (*dual inline memory module*), masaüstü bilgisayarlardaki RAM modülünün adıdır. **PCIe** (*PCI Express*), ekran kartı ve hızlı SSD gibi aygıtların bağlandığı yüksek hızlı veri yoludur; "x16", "x4" gibi ekler kaç **şerit** (*lane*) kullanıldığını gösterir. Şerit, veri yolunun paralel kanallarından biridir; şerit sayısı arttıkça aynı anda taşınan veri artar. **M.2**, küçük SSD'lerin doğrudan anakarta takıldığı yuvanın biçimidir. **SATA** ise sabit disklerin ve eski tip SSD'lerin kabloyla bağlandığı arayüzdür.

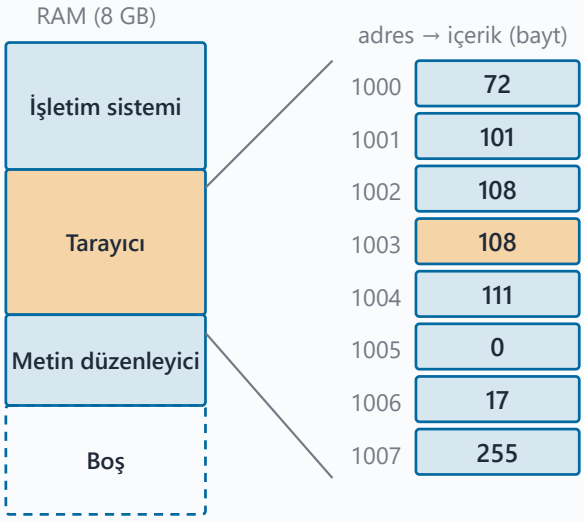
Anakartın bir başka görevi uyumluluğu belirlemektir. CPU soketi hangi işlemcinin takılabileceğini, DIMM yuvaları hangi RAM neslinin kullanılabileceğini, yonga seti ise kaç tane hızlı bağlantının aynı anda desteklendiğini belirler. Bu nedenle bir bilgisayar parça parça toplanırken ilk karar çoğunlukla işlemci ile anakart arasında verilir. Uyumluluk konusu 1.2.10.1'de toplu olarak ele alınır.

1.2.3.2 RAM

Tanım 1.2.7

RAM (*random access memory*, rastgele erişimli bellek), çalışan programların ve kullandıkları verinin tutulduğu ana bellektir. **Geçicidir** (*volatile*): elektrik kesilince içeriği kaybolur. "Rastgele erişim", her bellek hücresine sırayı beklemeden, doğrudan adresiyle ulaşılabilmesi demektir.

RAM'in yapısını anlamak için onu çok uzun bir dolap sırası gibi düşünmek yararlıdır. Her dolabın bir numarası, yani **adresi** vardır ve her dolapta tam olarak bir **bayt** durur. Bir bayt 8 bitten oluşur ve 0 ile 255 arasında bir tam sayı tutabilir. İşlemci bir veriyi okumak istediğinde adresi söyler, bellek o adresteki baytı verir. 1000 numaralı adrese ulaşmak için 0'dan 999'a kadar olan adresleri gezmek gerekmez; her adrese aynı sürede ulaşılır.



Şekil 1.2.8: 8 GB RAM'in bir anı. İşletim sistemi, tarayıcı ve metin düzenleyici bellekte ayrı bölgeler kaplar; bir kısım boştur. Yakından bakıldığında her baytın bir adresi vardır: 1000 adresinde 72, 1001'de 101, 1002 ve 1003'te 108, 1004'te 111 durur.

Şekil 1.2.8'deki sayılar rastgele seçilmemiştir. Metinler bellekte sayı olarak saklanır; her karaktere bir sayı karşılık gelir. Yaygın kullanılan ASCII kodlamasında 72 “H”, 101 “e”, 108 “l”, 111 “o” harfidir. Demek ki 1000–1004 adresleri arasında “Hello” sözcüğü durur. Bellek yalnızca sayıları bilir; bu sayıların harf mi, renk mi, ses örneği mi olduğuna onları kullanan program karar verir.

Örnek 1.2.5

1000 adresindeki 72 değeri bellekte 8 bit olarak nasıl durur? Aynı değer onaltılık tabanda nasıl yazılır?

1. **İkili taban.** 8 bitin basamak değerleri soldan sağa 128, 64, 32, 16, 8, 4, 2, 1'dir. 72'yi bu değerlerin toplamı olarak yazalım: 72 = 64 + 8. 64 ve 8 basamakları 1, ötekiler 0 olur: **01001000**.
2. **Denetim.** 0·128 + 1·64 + 0·32 + 0·16 + 1·8 + 0·4 + 0·2 + 0·1 = 64 + 8 = 72.
3. **Onaltılık taban.** 8 bit, dörder bitlik iki gruba ayrılır: 0100 ve 1000. 0100 = 4, 1000 = 8 olduğu için 72 onaltılık tabanda **48** olarak yazılır (programlamada çoğunlukla `0x48`).
4. **En büyük değer.** Sekiz bitin hepsi 1 olduğunda 11111111 = 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = **255** elde edilir; onaltılık karşılığı `0xFF` 'tir. Şekil 1.2.8'deki 1007 adresindeki 255 bu yüzden bir baytın alabileceği en büyük değerdir.

RAM'in dört özelliği bir bilgisayarın günlük kullanımını doğrudan etkiler:

- **Geçicilik:** Güç kesilince içerik kaybolur. Kaydedilmemiş bir belgenin elektrik kesintisinde yok olmasının nedeni budur.

- **Hızlı erişim:** CPU her adrese doğrudan ulaşır. RAM'e erişim yaklaşık 50–100 ns sürer; bu, depolama birimlerinden binlerce kat hızlıdır.
- **Kapasite:** RAM'in büyüklüğü, aynı anda rahatça çalışabilecek program miktarını belirler. Günümüz bilgisayarlarında 8–64 GB yaygındır.
- **Uyumluluk:** RAM modülünün nesli (DDR4, DDR5), hızı ve anakartın desteği birbirine uymalıdır.

RAM yetmediğinde ne olur? İşletim sistemi, RAM'de yer kalmayınca bir süredir kullanılmayan verinin bir kısmını diske yazar ve yeri boşaltır. Bu diske yazılan alana **takas alanı** (*swap space*) ya da Windows'ta sayfa dosyası denir. Gerekince veri yeniden diskten okunur. Disk RAM'den çok yavaş olduğu için bilgisayar belirgin biçimde yavaşlar. Diskte boş yer bol olsa bile durum değişmez: yavaşlığın nedeni yer azlığı değil, diskin hızıdır.

Sık yapılan hata

Yavaşlayan bir bilgisayarda yalnızca diskteki boş alana bakmak. Depolama birimi neredeyse boş olsa bile RAM yetersizse sistem diski geçici bellek olarak kullanır ve programlar yavaşlar. Bu durumda çözüm disk değil, daha fazla RAM ya da aynı anda daha az program çalıştırmaktır.

1.2.3.3 ROM ve ürün yazılımı

Bilgisayar açıldığında RAM boştur. İşletim sistemi diskte durur, ama diski okuyacak, ekranı başlatacak, klavyeyi tanıyacak bir yazılım henüz çalışmaz. Bu “ilk adım” sorununu, anakart üzerindeki kalıcı bir bellekte duran küçük bir yazılım çözer.

Tanım 1.2.8

ROM (*read-only memory*, salt okunur bellek), elektrik kesilince içeriği kaybolmayan bellektir. **Ürün yazılımı** (*firmware*) ise donanımın temel davranışını yöneten, düşük düzeyli yazılımdır. Bilgisayarlarda ürün yazılımının adı **BIOS** ya da onun yerini alan daha yeni standartla **UEFI**'dir.

ROM adı “salt okunur” anlamına gelse de günümüzde ürün yazılımı çoğunlukla yeniden yazılabilir **flash bellekte** tutulur. Flash bellek de güç kesilince içeriğini korur, ama gerektiğinde yeniden yazılabilir. Böylece ürün yazılımı güncellenebilir.

Bilgisayarın açılışı kabaca şu sırayla ilerler:

1. Güç düğmesine basılır; işlemci, anakarttaki flash yongasında duran UEFI'yi çalıştırmaya başlar.
2. UEFI donanımı sınar ve başlatır: RAM, ekran kartı, klavye, depolama birimleri tanınır. Bu sınamaya **açılışta öz sınama** (*power-on self-test*, POST) denir.
3. UEFI, ayarlarda belirtilen **önyükleme sırasına** göre diskleri tarar ve işletim sistemini başlatacak programı (önyükleyiciyi) bulur.

4. Önyükleyici işletim sistemini diskten RAM'e yükler ve denetimi ona devreder.

Ürün yazılımı güncellemesi, yeni bir işlemciyi tanımak, bir uyumluluk sorununu gidermek ya da bir güvenlik açığını kapatmak için yapılabilir. Ancak dikkat gerektirir: güncelleme sırasında elektrik kesilirse flash bellekteki yazılım yarım kalabilir ve bilgisayar açılmayabilir. Bu yüzden güncelleme yalnızca gerekli olduğunda ve kesintisiz güç altında yapılır.

Sık yapılan hata

ROM ile RAM'i adları benzediği için karıştırmak. RAM geçicidir ve çalışan programları tutar; boyutu gigabaytlarla ölçülür. ROM (flash) kalıcıdır ve yalnızca açılış yazılımını tutar; boyutu birkaç on megabayttır. Bilgisayar açıldığında ilk çalışan yazılım RAM'de değil, bu kalıcı bellekte durur.

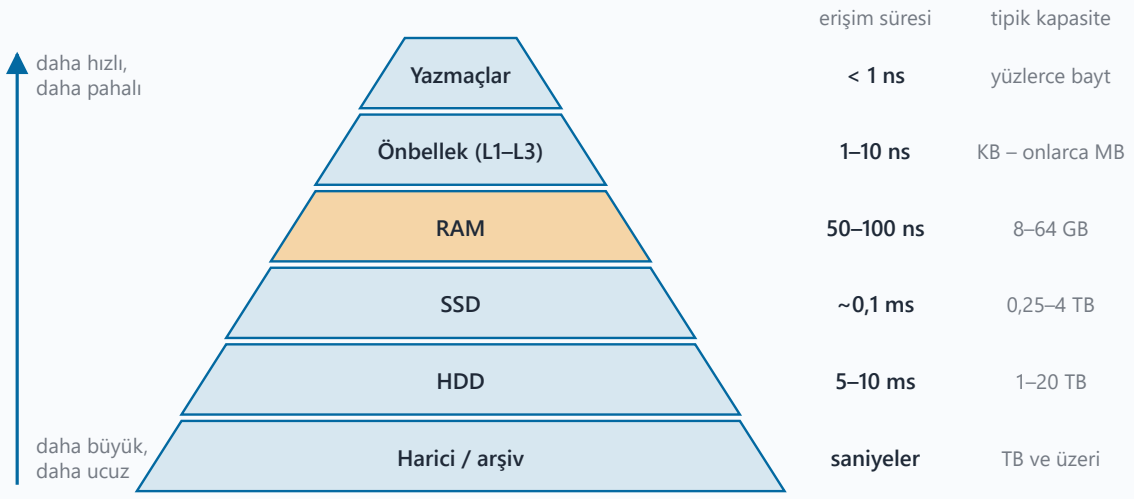
1.2.4 Bellek hiyerarşisi ve önbellek

İdeal bir bellek hem çok hızlı, hem çok büyük, hem de ucuz olurdu. Böyle bir bellek yoktur. Hızlı bellek teknolojileri pahalıdır ve küçük kalır; büyük ve ucuz bellekler ise yavaştır. Bilgisayarlar bu sorunu birkaç farklı belleği katmanlar hâlinde bir arada kullanarak çözer.

1.2.4.1 Bellek hiyerarşisi

Tanım 1.2.9

Bellek hiyerarşisi (*memory hierarchy*), bilgisayardaki bellek türlerinin hızlarına ve kapasitelerine göre katmanlar hâlinde sıralanmasıdır. Üst katmanlar (yazmaçlar, önbellek) hızlı, küçük ve pahalıdır; alt katmanlar (RAM, SSD, HDD) yavaş, büyük ve ucuzdur.



Şekil 1.2.9: Bellek hiyerarşisi piramidi. Yukarı çıktıkça erişim hızlanır, kapasite azalır ve bayt başına maliyet artar. Değerler yaklaşıktır.

Katman	Erişim süresi	Tipik kapasite	Kalıcı mı?
Yazmaçlar	< 1 ns	yüzlerce bayt	hayır
Önbellek (L1–L3)	1–10 ns	KB – onlarca MB	hayır
RAM	50–100 ns	8–64 GB	hayır
SSD	≈ 0,1 ms	0,25–4 TB	evet
HDD	5–10 ms	1–20 TB	evet
Harici / arşiv	saniyeler	TB ve üzeri	evet

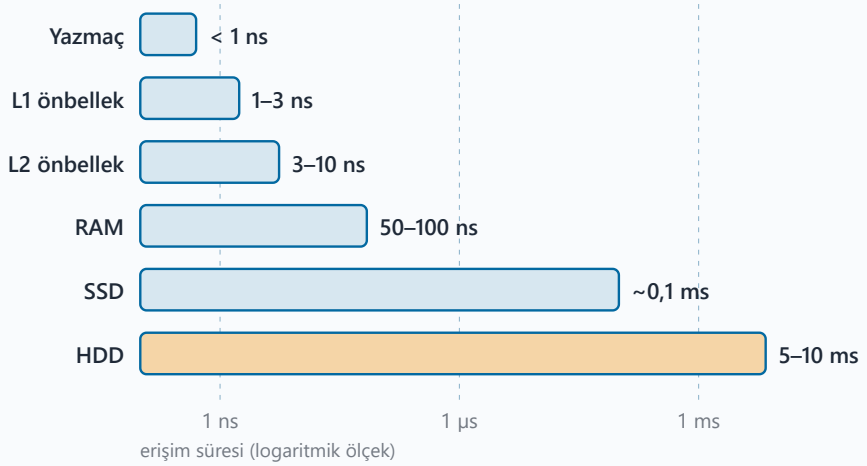
Tablo 1.2.3: Bellek katmanlarının yaklaşık erişim süreleri ve tipik kapasiteleri.

Hiyerarşinin işe yaramasının nedeni, programların belleği düzensiz kullanmamasıdır. Bir program kısa bir süre içinde çoğunlukla aynı küçük veri kümesiyle çalışır: bir döngü aynı değişkenleri tekrar tekrar okur, bir dizi baştan sona sırayla gezilir. Bu küçük küme üst katmanlarda tutulursa işlemcinin isteklerinin büyük çoğunluğu hızlı katmandan karşılanır. Yavaş katmanlara yalnızca nadiren inilir.

Tablonun son sütunu da önemli bir ayrımı gösterir. RAM ve üzerindeki katmanlar geçicidir, SSD ve altındakiler kalıcıdır. Bir belge üzerinde çalışılırken değişiklikler RAM'de durur; ancak “Kaydet” komutuyla kalıcı katmana yazılır.

1.2.4.2 Katmanlar arasındaki hız farkı

Erişim süreleri arasındaki fark o kadar büyüktür ki, bu süreleri aynı çizgi üzerinde göstermek için özel bir ölçek gerekir. Şekil 1.2.10 **logaritmik ölçek** kullanır: eksen üzerindeki her ızgara çizgisi bir öncekinin 1000 katıdır (1 ns, 1 µs, 1 ms). Böylece nanosaniyelerle milisaniyeler aynı şekle sığar.



Şekil 1.2.10: Bellek katmanlarının erişim süreleri, logaritmik ölçekte. Yazmaç 1 ns'den kısa, L1 önbellek 1–3 ns, L2 önbellek 3–10 ns, RAM 50–100 ns, SSD yaklaşık 0,1 ms, HDD 5–10 ms sürer.

Birimler arasındaki ilişkiyi hatırlamak hesapları kolaylaştırır: 1 ms (milisaniye) = 1000 µs (mikrosaniye) = 1.000.000 ns. Yani 0,1 ms, 100.000 ns'dir.

Örnek 1.2.6

SSD, RAM'den kaç kat yavaştır? HDD, SSD'den ve RAM'den kaç kat yavaştır?

1. **SSD ile RAM.** SSD ≈ 0,1 ms = 100.000 ns. RAM 50–100 ns'dir. 100.000 ÷ 100 = 1000, 100.000 ÷ 50 = 2000. SSD, RAM'den yaklaşık **1000–2000 kat** yavaştır.
2. **HDD ile SSD.** HDD 5–10 ms'dir. 5 ÷ 0,1 = 50, 10 ÷ 0,1 = 100. HDD, SSD'den **50–100 kat** yavaştır.
3. **HDD ile RAM.** 5 ms = 5.000.000 ns ve 10 ms = 10.000.000 ns. En küçük oran 5.000.000 ÷ 100 = 50.000, en büyük oran 10.000.000 ÷ 50 = 200.000'dir. HDD, RAM'den **50.000–200.000 kat** yavaştır.

Bu farkı gündelik bir ölçeğe taşımak sezgi kazandırır. RAM'den bir okuma 1 saniye sürseydi, SSD'den bir okuma yaklaşık 17–33 dakika, HDD'den bir okuma ise 14 saatle 2,3 gün arasında sürerdi.

Bu oranlar, işlemcinin neden veriyi diskten değil RAM'den, mümkünse de önbellekten okumak istediğini açıklar. RAM bile işlemcinin bir saat turuna göre yavaştır: 3 GHz'de bir tur 0,33 ns'dir, RAM'den gelecek bir değeri beklemek ise 150–300 tur kadar sürer. Önbellek bu beklemeyi azaltmak için vardır.

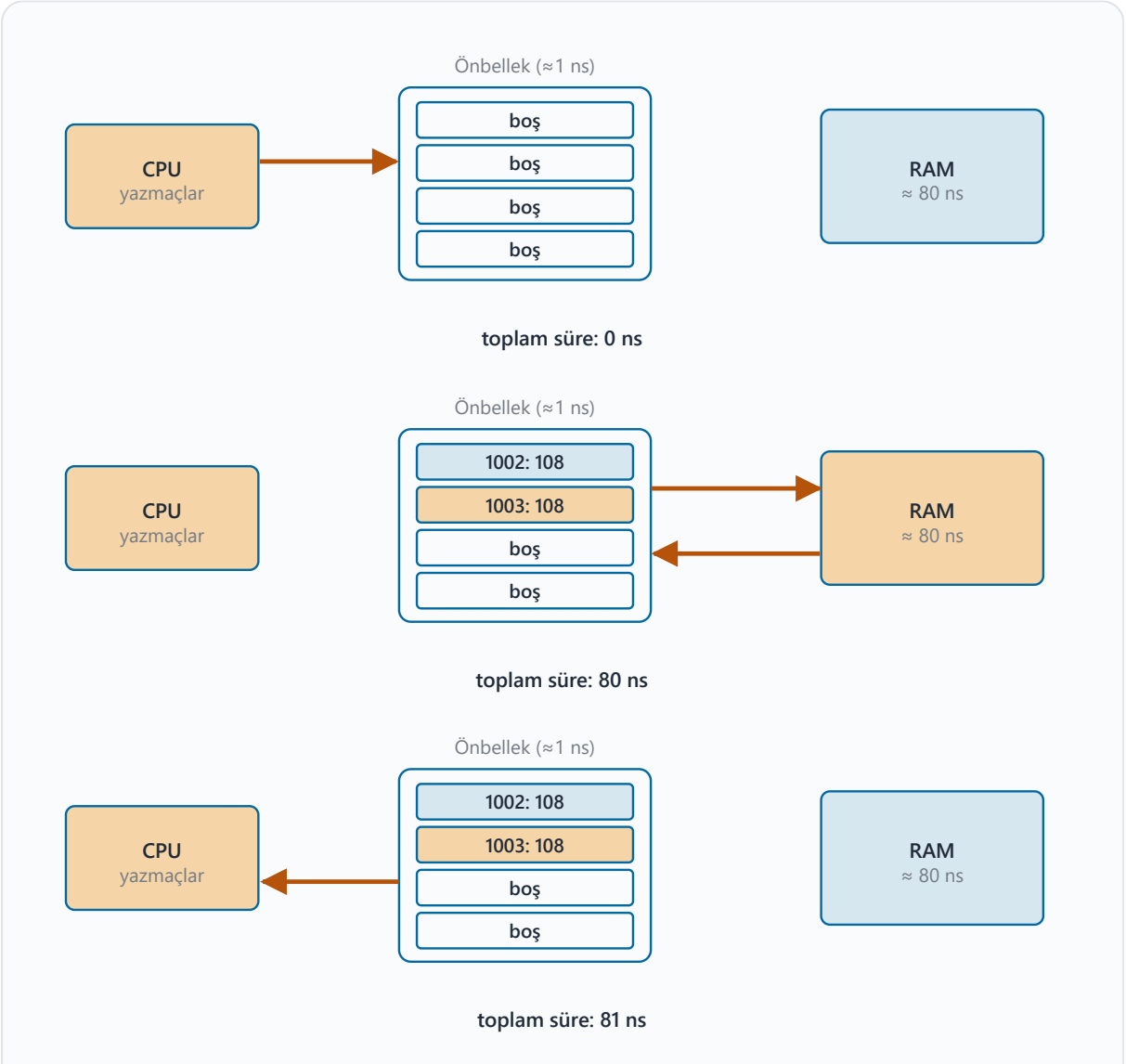
1.2.4.3 Önbellek: isabet ve ıska

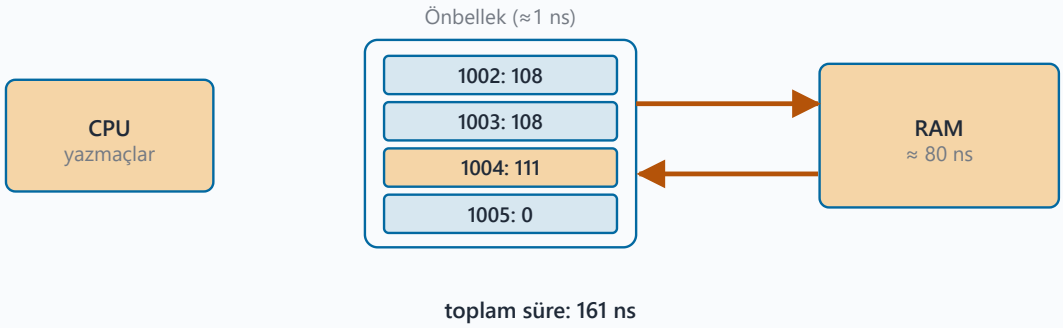
Tanım 1.2.10

Önbellek (*cache*), sık kullanılan verinin kopyasını işlemcinin yanında tutan küçük ve hızlı bellektir. İşlemcinin istediği veri önbellekte bulunursa buna **isabet** (*hit*), bulunamazsa **ıska** (*miss*) denir. Iskada veri RAM'den getirilir ve bir kopyası önbelleğe yazılır.

Önbelleğin çalışma kuralı basittir. İşlemci bir adresi okumak istediğinde önce önbelleğe bakar. Veri oradaysa yaklaşık 1 ns'de alınır. Değilse RAM'e gidilir; bu yaklaşık 80 ns sürer. RAM'den gelen veri yalnız işlemciye verilmez, önbelleğe de kopyalanır. Üstelik istenen baytın yanındaki komşu baytlar da onunla birlikte getirilir, çünkü sıradaki isteğin büyük olasılıkla komşu adrese olacağı beklenir.

Şekil 1.2.11 bu kuralı, Şekil 1.2.8'deki bellek içeriği üzerinde dört anda gösterir. Bu örnekte önbellek iki adreslik bloklar hâlinde doldurulur: 1002–1003, 1004–1005 gibi.





Şekil 1.2.11: Önbellekte isabet ve ıska, yukarıdan aşağıya dört an. (1) Önbellek boştur; CPU 1003 adresini ister. (2) Iska: 1003 RAM'den 80 ns'de getirilir, komşusu 1002 ile birlikte önbelleğe yazılır. (3) 1003 yeniden istenir: isabet, değer 108 yaklaşık 1 ns'de gelir; toplam 81 ns. (4) 1004 istenir: önbellekte yoktur, ıska; 1004–1005 bloğu RAM'den getirilir, toplam 161 ns olur.

İstek	Önbellekte var mı?	Sonuç	Süre	Toplam	İstekten sonra önbellek
1003 (ilk)	hayır	ıska	80 ns	80 ns	1002, 1003
1003 (tekrar)	evet	isabet	1 ns	81 ns	1002, 1003
1004	hayır	ıska	80 ns	161 ns	1002, 1003, 1004, 1005

Tablo 1.2.4: Şekil 1.2.11'deki üç isteğin izlemesi (isabet ≈ 1 ns, ıska ≈ 80 ns).

Üç istek önbellekle 161 ns sürmüştür. Önbellek olmasaydı her istek RAM'e gider ve toplam 3 × 80 = 240 ns olurdu. Kazanç ikinci istekten gelir: aynı adres kısa süre içinde yeniden istendiği için RAM'e gitmeye gerek kalmamıştır.

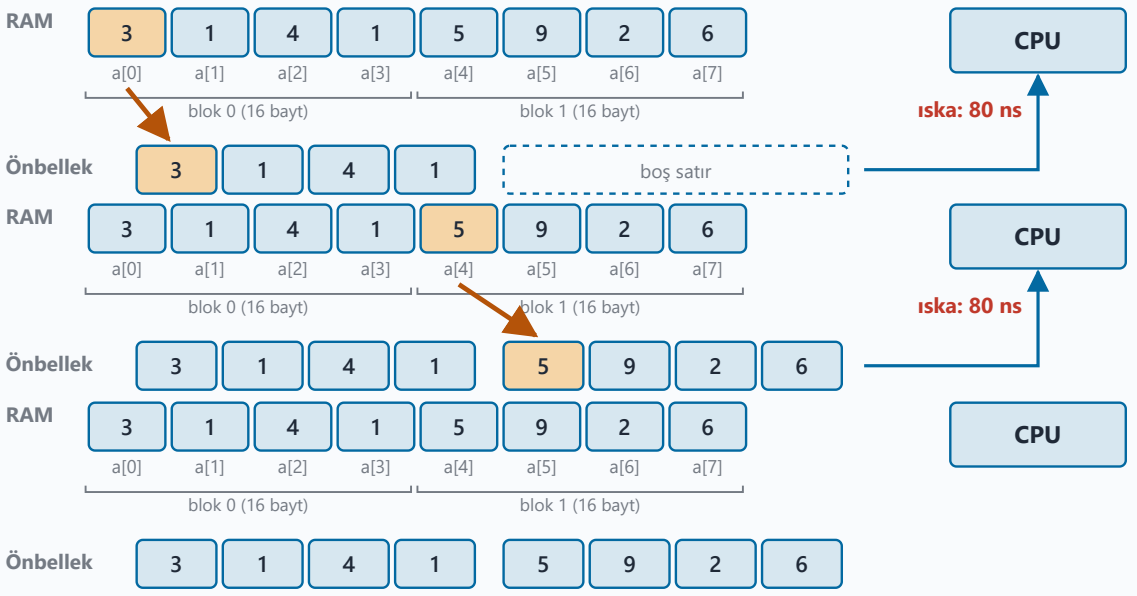
Gerçek önbellekler bu örnekten çok daha büyüktür, ama sınırlıdır. Önbellek dolduğunda yeni bir blok için yer açmak gerekir. Çoğu önbellek, uzun süredir kullanılmayan bloğu çıkarır. Bu küçük örneklerde yer sorunu yok sayılmıştır.

1.2.4.4 Dizi toplamında önbellek: uzamsal yerellik

Önbelleğin komşu baytları da getirmesi, programlardaki çok yaygın bir davranışa dayanır. Bir dizi (aynı türden değerlerin bellekte art arda dizildiği yapı) baştan sona gezildiğinde, her eleman bir öncekinin hemen yanındadır. Bu davranışın ne kadar zaman kazandırdığını sekiz elemanlı bir dizinin toplamı üzerinde sayalım. Program şudur:

```
toplam = 0
her i = 0, 1, …, 7 için
    toplam = toplam + a[i]
// 4 baytlık tamsayı, 16 baytlık satır
```

Hesap için şu varsayımlar kullanılır: bir tam sayı 4 bayt yer kaplar; önbelleğe RAM'den her seferinde 16 baytlık bir blok getirilir. Bu bloğa **önbellek satırı** (*cache line*) denir. 16 ÷ 4 = 4 olduğu için bir satıra 4 eleman sığar: a[0]…a[3] birinci blokta, a[4]…a[7] ikinci bloktadır. İsabet yaklaşık 1 ns, ıska yaklaşık 80 ns sürer. Dizinin değerleri 3, 1, 4, 1, 5, 9, 2, 6'dır.



Şekil 1.2.12: Dizi toplamında önbellek. Üstte a[0] istenmiş ve ıska olmuştur; bütün blok 0 (a[0]…a[3]) önbelleğe gelmiştir. Ortada a[4] istenmiş, ikinci ıskada blok 1 (a[4]…a[7]) gelmiştir. Altta döngü bitmiştir: iki blok da önbellektedir.

`i`	a[i]	Blok	Sonuç	toplam	isabet	ıska	Süre
0	3	0	ıska	3	0	1	80 ns
1	1	0	isabet	4	1	1	81 ns
2	4	0	isabet	8	2	1	82 ns
3	1	0	isabet	9	3	1	83 ns
4	5	1	ıska	14	3	2	163 ns
5	9	1	isabet	23	4	2	164 ns
6	2	1	isabet	25	5	2	165 ns
7	6	1	isabet	31	6	2	166 ns

Tablo 1.2.5: Dizi toplamının izlemesi. Her satır bir döngü adımından sonraki durumu gösterir.

Tablo 1.2.5 sonucu açıkça verir: toplam 31'dir; 8 erişimin 6'sı isabet, 2'si ıskadır. Süre 2 × 80 + 6 × 1 = **166 ns**'dir. Önbellek olmasaydı her erişim RAM'e gider ve 8 × 80 = **640 ns** sürerdi. Önbellek bu döngüyü yaklaşık 3,9 kat hızlandırmıştır. Erişim başına ortalama süre de 166 ÷ 8 ≈ 20,8 ns'ye inmiştir.

Tanım 1.2.11

Uzamsal yerellik (*spatial locality*), bir adrese erişen programın kısa süre içinde ona yakın adreslere de erişme eğilimidir. Önbelleğin istenen bayt yerine bütün bir satırı getirmesi bu eğilimden yararlanır.

Örnek 1.2.7

Aynı dizi, 16 elemanlı olsaydı ve iki farklı biçimde gezilseydi süreler ne olurdu? (a) Elemanlar sırayla okunuyor. (b) Yalnızca a[0], a[4], a[8], a[12] okunuyor.

1. **(a) Sırayla.** 16 eleman 4 bloğa yayılır. Her bloğun ilk elemanında ıska, kalan üçünde isabet olur: 4 ıska, 12 isabet. Süre 4 × 80 + 12 × 1 = 332 ns. Önbelleksiz 16 × 80 = 1280 ns olurdu.
2. **(b) Dörder atlayarak.** Okunan her eleman farklı bir bloktadır ve her blok yalnızca bir kez kullanılır. 4 erişimin 4'ü de ıskadır: 4 × 80 = 320 ns. Okunan eleman sayısı dörtte bire inmiştir, ama süre neredeyse hiç kısalmamıştır.

Atlayarak gezmede önbelleğin getirdiği komşu elemanlar hiç kullanılmaz. Erişim başına süre (a)'da 332 ÷ 16 ≈ 20,8 ns iken (b)'de 80 ns'dir.

Sık yapılan hata

Okunan eleman sayısı azaldıkça sürenin aynı oranda kısalacağını sanmak. Önbellekli bir sistemde süreyi asıl belirleyen, RAM'e kaç kez gidildiğidir. Dağınık adreslere yapılan az sayıda erişim, art arda adreslere yapılan çok sayıda erişimden daha uzun sürebilir.

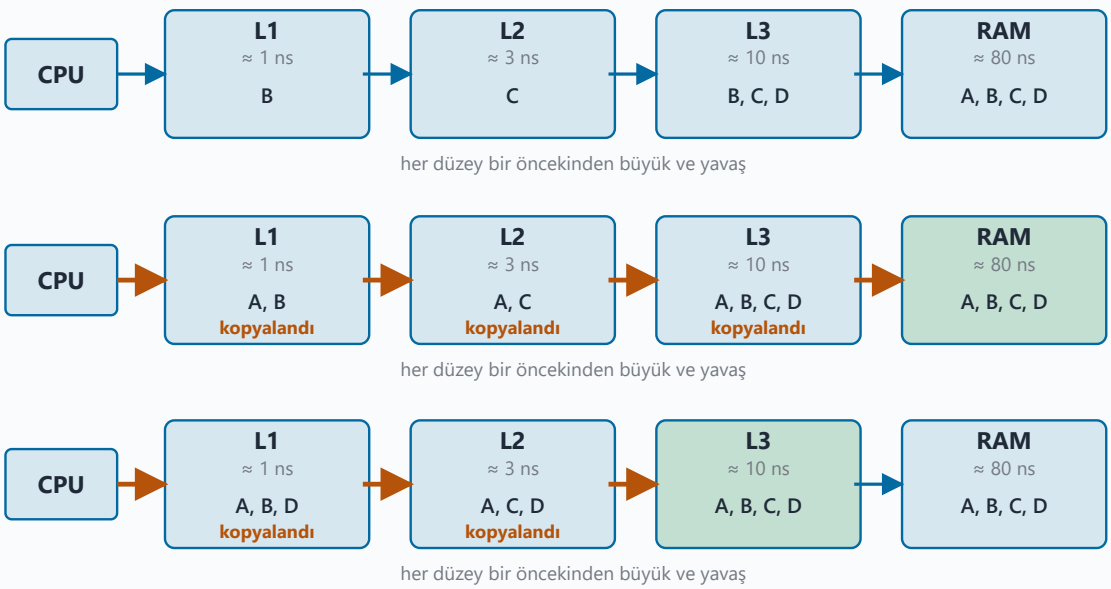
1.2.4.5 Çok düzeyli önbellek: zamansal yerellik

Gerçek işlemcilerde tek bir önbellek yerine birkaç düzey bulunur. **L1** en küçük ve en hızlı düzeydir ve her çekirdeğin içindedir. **L2** daha büyük ama biraz daha yavaştır. **L3** en büyük ve en yavaş önbellek düzeyidir; çoğunlukla bütün çekirdeklerce paylaşılır (Şekil 1.2.3). İşlemci bir veriyi en hızlı düzeyden başlayarak arar:

```
L1'de var mı?  Evet → oku, bitir
L2'de var mı?  Evet → L1'e kopyala, oku
L3'te var mı?  Evet → L2 ve L1'e kopyala, oku
RAM'den getir → L3, L2, L1'e kopyala, oku
```

Arama her düzeyde biraz zaman harcar; veri bulunamazsa bir alt düzeye geçilir. Bu yüzden bir ıskanın maliyeti, aranan bütün düzeylerin sürelerinin toplamıdır. Aşağıdaki örnekte L1 ≈ 1 ns, L2 ≈ 3 ns, L3 ≈ 10 ns, RAM ≈ 80 ns kabul edilir. Başlangıçta L1'de B, L2'de C, L3'te B, C ve D vardır; A yalnızca

RAM'dedir. İşlemci sırayla A, A ve D adreslerini okuyacaktır.



Şekil 1.2.13: Çok düzeyli önbellekte okuma. Üstte başlangıç durumu. Ortada A, RAM'den getirildikten sonraki durum: A bütün düzeylere kopyalanmıştır. Altta D, L3'te bulunduktan sonraki durum: D, L2 ve L1'e kopyalanmıştır.

İstek	Arama yolu	Süre	Sonra L1 / L2 / L3
A	L1 yok (1) → L2 yok (+3) → L3 yok (+10) → RAM (+80)	94 ns	A, B / A, C / A, B, C, D
A	L1'de var	1 ns	değişmez
D	L1 yok (1) → L2 yok (+3) → L3'te var (+10)	14 ns	A, B, D / A, C, D / A, B, C, D

Tablo 1.2.6: Üç okumanın izlemesi.

Üç okuma toplam 94 + 1 + 14 = **109 ns** sürmüştür. Üçü de doğrudan RAM'den okunsaydı 3 × 80 = **240 ns** olurdu. Tablodaki iki ayrıntı önemlidir:

- İlk A okuması, önbelleksiz okumadan (80 ns) daha uzundur: 94 ns. Iskanın bedeli, boşuna aranan düzeylerin süresidir. Önbellek her tek okumayı hızlandırmaz; kazanç, aynı verinin tekrar kullanılmasından gelir.
- D, L3'te bulunmuştur: 14 ns. RAM'e gitmek 94 ns sürerdi; 94 ÷ 14 ≈ 6,7, yani yaklaşık 7 kat daha hızlıdır.

Tanım 1.2.12

Zamansal yerellik (*temporal locality*), bir veriye erişen programın aynı veriye kısa süre içinde yeniden erişme eğilimidir. İkinci A okumasının 1 ns sürmesi bu eğilimden kaynaklanır.

Uzamsal ve zamansal yerellik birlikte, önbelleğin neden bu kadar etkili olduğunu açıklar. Bir döngünün sayaç değişkeni her turda yeniden okunur (zamansal yerellik); döngünün gezdiği dizi ise ardışık adreslerde durur (uzamsal yerellik). Tipik programlarda işlemcinin isteklerinin büyük çoğunluğu önbellekten karşılanır.

Örnek 1.2.8

Bir programın bellek isteklerinin %95'i önbellekte isabet ediyor. İsabet 1 ns, ıska ise önbelleğe bakma (1 ns) ile RAM'den getirme (80 ns) toplamı olarak 81 ns sürüyor. Bir isteğin ortalama süresi nedir?

1. 100 istekten 95'i isabet: 95 × 1 = 95 ns.
2. Kalan 5'i ıska: 5 × 81 = 405 ns.
3. Toplam 95 + 405 = 500 ns; istek başına ortalama 500 ÷ 100 = **5 ns**.

Önbelleksiz her istek 80 ns sürerdi. %5'lik ıska oranı bile süreyi belirler: 5 ns'nin 4 ns'den fazlası ıskalardan gelir. İsabet oranındaki küçük bir artış bu yüzden büyük kazanç sağlar.

1.2.5 Ekran kartı, bağlantılar ve ses

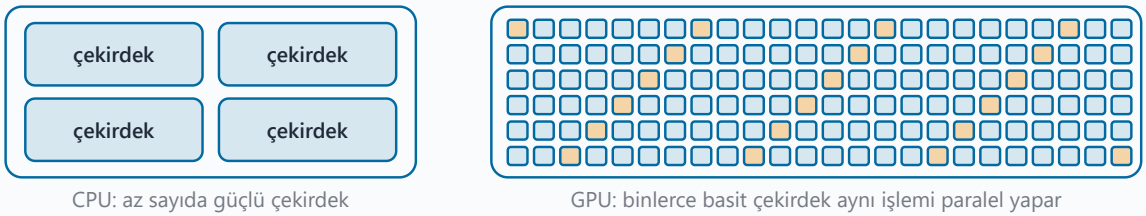
İşlemci sonucu hesaplar, ama bu sonucun ekranda bir görüntüye, hoparlörde bir sese dönüşmesi için başka birimler gerekir. Bu kesim görüntüyü üreten grafik işlem birimini, birimleri bilgisayara bağlayan portları ve sesi sayıya, sayıyı sese çeviren ses kartını ele alır.

1.2.5.1 Grafik işlem birimi

Tanım 1.2.13

Grafik işlem birimi (*graphics processing unit*, GPU), çok sayıda benzer işlemi aynı anda yürütmek üzere tasarlanmış işlemcidir. Adını, milyonlarca pikselin rengini hesaplama işinden alır. GPU'yu taşıyan kart **ekran kartı** olarak da anılır.

Bir ekranda görüntü oluşturmak, aynı hesabı çok sayıda piksel için tekrarlamak demektir. 1920 × 1080 çözünürlüklü bir ekranda 1920 × 1080 = 2.073.600 piksel vardır. Saniyede 60 kare gösterilecekse her saniye 124 milyondan fazla piksel rengi hesaplanmalıdır. Bu hesapların her biri basittir ve birbirinden bağımsızdır. Böyle bir iş için birkaç güçlü çekirdek yerine binlerce basit çekirdek daha uygundur.



Şekil 1.2.14: CPU az sayıda güçlü çekirdekten oluşur; GPU'da ise binlerce basit çekirdek aynı işlemi paralel yapar.

Şekil 1.2.14 iki işlemci türünün farkını özetler. CPU'nun çekirdekleri karmaşık ve birbirinden farklı işleri hızla yürütmeye uygundur: bir karar ver, bir dosya aç, bir hesabı bitirip başka bir işe geç. GPU'nun çekirdekleri ise tek başlarına zayıftır, ama binlercesi aynı komutu farklı veriler üzerinde aynı anda yürütür. Bir sınıftaki herkesin aynı anda kendi sorusunu çözmesine benzer: tek bir öğretmenin bütün soruları sırayla çözmesinden çok daha hızlıdır.

GPU'lar iki biçimde bulunur:

- **Entegre GPU** işlemciyle aynı pakettedir ve sistem belleğini (RAM'i) paylaşır. Az güç harcar, ek maliyet getirmez; ofis işleri, video izleme ve hafif oyunlar için yeterlidir.
- **Bağımsız GPU** ayrı bir karttır. Kendi işlemcisi ve kendi **video belleği** (*video memory*, VRAM) vardır. Daha güçlüdür, ama daha çok güç harcar ve daha çok ısı üretir.

GPU'ların paralel hesap gücü yalnız grafikle sınırlı kalmamıştır. Video işleme, bilimsel hesaplama ve özellikle yapay zekâ modellerinin eğitimi de aynı türden çok sayıda bağımsız çarpma ve toplama içerir. Bir GPU seçilirken iş yükü, VRAM miktarı, güç tüketimi, soğutma ve kullanılacak yazılımın o GPU'yu destekleyip desteklemediği birlikte değerlendirilir.

> **Sık yapılan hata**
>
> GPU'nun CPU'nun yerini tutabileceğini düşünmek. GPU yalnızca bölünebilen, aynı türden çok sayıda işte hızlıdır. İşletim sistemini çalıştırmak, bir programın adım adım kararlarını yürütmek gibi işler CPU'da kalır. İki işlemci birbirini tamamlar.

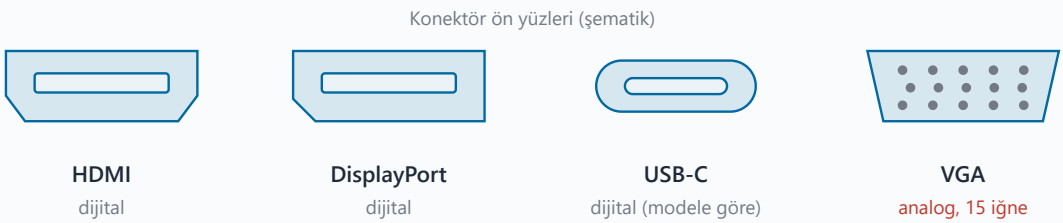
1.2.5.2 Görüntü bağlantıları

Ekran kartının ürettiği görüntü monitöre bir kablo ile taşınır. Kablonun ve portun türü, taşınabilecek çözünürlüğü, yenileme hızını ve sesin de aynı kablodan gidip gitmeyeceğini belirler.

Bağlantı	Taşıdığı sinyal	Yaygın kullanım
HDMI	dijital görüntü ve ses	monitör, televizyon, projektör
DisplayPort	dijital görüntü ve ses	bilgisayar monitörleri
USB-C	modele göre görüntü, veri ve güç	dizüstü ve mobil aygıtlar

Bağlantı	Taşıdığı sinyal	Yaygın kullanım
VGA	analog görüntü	eski ekranlar ve projektörler

Tablo 1.2.7: Yaygın görüntü bağlantıları.



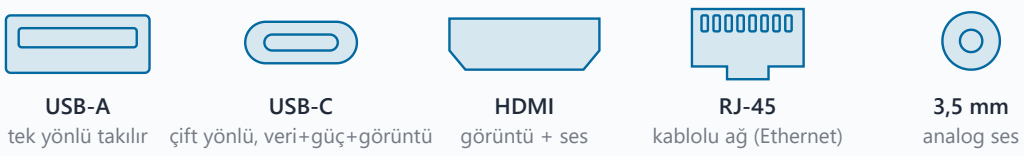
Şekil 1.2.15: Görüntü konektörlerinin ön yüzleri (şematik). HDMI'nin alt köşeleri kesik bir yamuktur; DisplayPort'un yalnız bir köşesi kesiktir; USB-C oval ve simetriktir; VGA ise 15 iğneli, analog bir konektördür.

Analog ile dijital sinyal arasındaki fark burada önemlidir. **Dijital** bağlantı, her pikselin rengini sayı olarak taşır; alıcı bu sayıları olduğu gibi okur. **Analog** bağlantı (VGA) ise renkleri sürekli değişen gerilimlerle taşır. Kablo uzadıkça ya da çevrede elektriksel gürültü arttıkça bu gerilimler bozulur ve görüntüde bulanıklık, gölge görülür. Bu yüzden VGA günümüzde yalnızca eski aygıtlarla uyum için kullanılır.

Şekil 1.2.15'teki biçimler, konektörlerin neden yanlış porta takılamadığını da açıklar. Kesik köşeler, konektörün yalnızca tek bir yönde takılmasını sağlar. USB-C ise iki yönlü simetrik olduğu için ters takılma sorunu yoktur.

1.2.5.3 USB ve diğer portlar

USB (*Universal Serial Bus*, evrensel seri veri yolu), klavyeden harici diske kadar pek çok aygıtın bağlandığı ortak arayüzdür. USB'yi anlamanın anahtarı, iki ayrı kavramı birbirinden ayırmaktır: **konektör biçimi** ve **sürüm**.



Şekil 1.2.16: Yaygın portlar. USB-A tek yönlü takılır; USB-C çift yönlüdür ve veri, güç ve görüntü taşıyabilir; HDMI görüntü ve ses taşır; RJ-45 kablolu ağ (Ethernet) portudur; 3,5 mm jak analog ses taşır.

Konektör biçimi, fişin şeklidir: A tipi (geniş, dikdörtgen) ya da C tipi (küçük, oval). Sürüm ise verinin hangi hızda taşınacağını belirleyen standarttır. Tablo 1.2.8 başlıca sürümleri ve en yüksek hızlarını gösterir.

USB sürümü	En yüksek hız	Tipik kullanım
USB 2.0	480 Mbit/s	klavye, fare
USB 3.2 Gen 1 (eski adı USB 3.0)	5 Gbit/s	harici disk
USB 3.2 Gen 2	10 Gbit/s	hızlı harici SSD
USB4	20 veya 40 Gbit/s	USB-C; ekran, yerleştirme istasyonu
Thunderbolt 4	40 Gbit/s	USB-C; ekran, yerleştirme istasyonu

Tablo 1.2.8: USB sürümleri ve en yüksek teorik hızları.

Tablodaki hızlar **bit** cinsindendir. Dosya boyutları ise çoğunlukla **bayt** ile verilir. Birimlerde küçük "b" biti, büyük "B" baytı gösterir: 480 Mbit/s ile 480 MB/s çok farklı hızlardır. Bir bayt 8 bit olduğu için bit cinsinden hız 8'e bölünerek bayt cinsine çevrilir.

Örnek 1.2.9

5 GB'lık (5000 MB) bir dosya en yüksek teorik hızla kopyalanıyor. USB 2.0 ile ve USB 3.2 Gen 2 ile kaç saniye sürer?

1. **USB 2.0.** 480 Mbit/s ÷ 8 = 60 MB/s. Süre = boyut ÷ hız = 5000 ÷ 60 ≈ **83,3 s**.
2. **USB 3.2 Gen 2.** 10 Gbit/s = 10.000 Mbit/s; ÷ 8 = 1250 MB/s. Süre = 5000 ÷ 1250 = **4 s**.

Aynı dosya için fark yaklaşık 21 kattır. Gerçek hızlar teorik değerlerden düşüktür, çünkü protokolün kendi denetim verileri, diskin hızı ve kablonun kalitesi de sonucu etkiler.

Sık yapılan hata

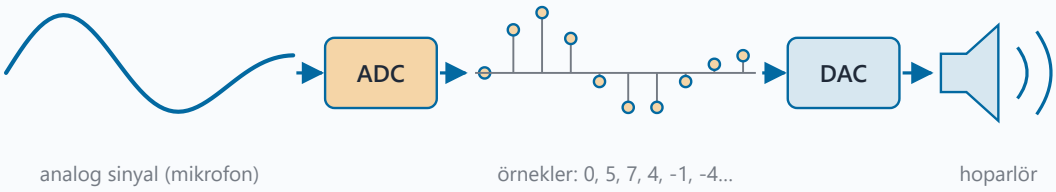
Her USB-C portunun aynı hızda olduğunu ve görüntü verebildiğini sanmak. USB-C yalnızca konektör biçimidir. Aynı dizüstündeki bir USB-C portu USB 2.0 hızında (480 Mbit/s) çalışırken öbürü USB4 (40 Gbit/s) olabilir. Görüntü çıkışı ve şarj desteği de porta göre değişir; bu bilgiler aygıtın belirtimlerinde ya da portun yanındaki simgelerde yer alır.

1.2.5.4 Ses kartı

Ses, havadaki basınç dalgalarıdır. Mikrofon bu dalgaları sürekli değişen bir elektrik gerilimine çevirir; buna **analog sinyal** denir. Bilgisayar ise yalnızca sayılarla çalışır. Ses kartının görevi bu iki dünya arasında çeviri yapmaktır.

Tanım 1.2.14

ADC (analogdan dijitale dönüştürücü), analog sinyali düzenli aralıklarla ölçüp her ölçümü bir sayıya çevirir. Her ölçüme **örnek**, saniyedeki ölçüm sayısına **örnekleme hızı** denir. **DAC** (dijitalden analoğa dönüştürücü) ise sayıları yeniden hoparlörü süren elektrik sinyaline dönüştürür.



Şekil 1.2.17: Mikrofondan gelen analog sinyal ADC'de örneklenip sayılara çevrilir (0, 5, 7, 4, −1, −4, ...). Çalarken DAC bu sayıları yeniden sinyale çevirir ve hoparlör sesi üretir.

Örnekleme hızı ne kadar yüksekse ses o kadar doğru kaydedilir. Müzik CD'lerinde örnekleme hızı saniyede 44.100 örnektir (44,1 kHz). Her örnek 16 bit, yani 2 bayt ile saklanır. Stereo kayıtta sol ve sağ için iki ayrı kanal vardır.

Örnek 1.2.10

Bir ses kaydı tek kanallı (mono), 48.000 örnek/s ve örnek başına 16 bit ile yapılıyor. 30 saniyelik sıkıştırılmamış kayıt kaç bayt tutar? Kaç MiB eder?

1. **Bir saniye.** 48.000 örnek × 2 bayt × 1 kanal = 96.000 bayt.
2. **30 saniye.** 96.000 × 30 = **2.880.000 bayt**.
3. **MiB'ye çevirme.** 1 MiB = 1024 × 1024 = 1.048.576 bayttır. 2.880.000 ÷ 1.048.576 ≈ **2,75 MiB**.

Aynı kayıt stereo olsaydı boyut iki katına çıkardı. MP3 gibi sıkıştırma biçimleri bu boyutu, kulağın fark etmediği ayrıntıları atarak on kat kadar küçültür.

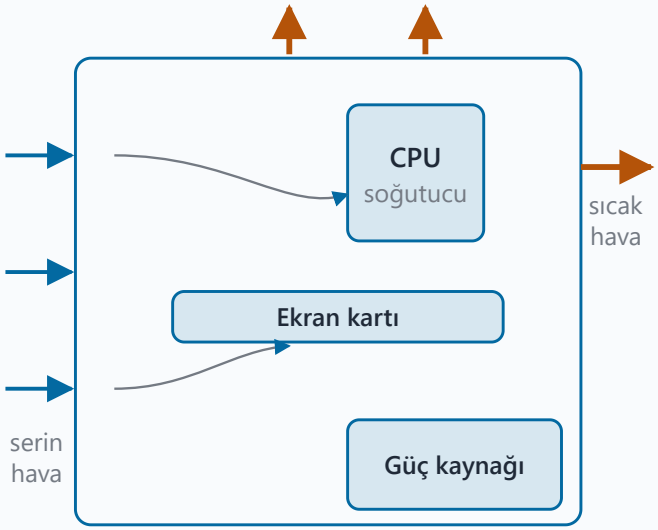
Çoğu anakart, günlük kullanım için yeterli bir ses birimi içerir: kulaklık ve mikrofon girişleri anakartın arka panelinde ya da kasanın önündedir. Kayıt kalitesinin önemli olduğu durumlarda, birden çok mikrofon bağlamak gerektiğinde ya da profesyonel müzik çalışmasında **harici ses kartı** kullanılır. Harici kartlar daha kaliteli ADC ve DAC içerir; bilgisayarın içindeki elektriksel gürültüden de uzakta durur.

1.2.6 Kasa, soğutma ve güç kaynağı

Bir bilgisayarın bileşenleri ne kadar güçlü olursa olsun, düzgün beslenmez ve soğutulmazsa verimli çalışmaz. Kasa ve güç kaynağı göz ardı edilmeye en yatkın parçalardır, ama sistemin kararlılığını doğrudan etkiler.

1.2.6.1 Kasa ve soğutma

Bilgisayarın içindeki her bileşen, harcadığı elektrik enerjisinin neredeyse tamamını ısıya çevirir. 125 W çeken bir işlemci, 125 W'lık küçük bir ısıtıcı gibi davranır. Bu ısı uzaklaştırılmazsa sıcaklık hızla yükselir.



Şekil 1.2.18: Kasa içinde hava akışı. Serin hava önden girer, işlemci soğutucusunun ve ekran kartının üzerinden geçer; ısınan hava arkadan ve üstten çıkar.

Kasanın dört görevi vardır:

- **Koruma:** Bileşenleri darbeden, tozdan ve dokunmadan kaynaklanabilecek statik elektrikten korur.
- **Hava akışı:** Serin hava önden girer, sıcak hava arkadan ya da üstten çıkar (Şekil 1.2.18). Sıcak hava yükseldiği için üstten çıkış doğal akışı destekler.
- **Form faktörü:** Kasanın boyutu ve iç düzeni, hangi anakart ve güç kaynağı boyutlarının sığacağını belirler. Yaygın anakart boyutları ATX ve daha küçük mATX'tir.
- **Kablo düzeni:** Kabloların toplanması hava akışını engellemez ve bakımı kolaylaştırır.

İşlemcinin üstüne bir **soğutucu** yerleştirilir. Soğutucu, ısıyı işlemciden alıp geniş yüzeyli metal kanatçıklara yayar; bir fan da bu kanatçıkların arasından hava geçirir. Daha güçlü sistemlerde sıvılı soğutma kullanılır: ısı bir sıvıya aktarılır ve sıvı, fanların bulunduğu bir radyatörde soğutulur.

Tanım 1.2.15

Isıl kısma (*thermal throttling*), işlemcinin sıcaklığı güvenli sınıra yaklaştığında saat hızını kendiliğinden düşürmesidir. Böylece üretilen ısı azalır ve yonga korunur, ama performans da düşer.

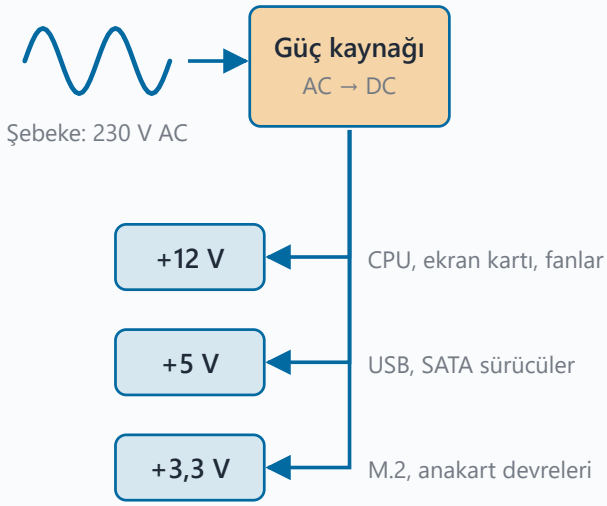
Isıl kısma bir korunma düzeneğidir; işlemcinin bozulmasını önler. Ancak sık yaşanıyorsa soğutmanın yetersiz olduğunu gösterir. Toz dolmuş bir dizüstü bilgisayarın zamanla yavaşlamasının yaygın nedenlerinden biri budur. Yüksek sıcaklık uzun vadede bileşenlerin ömrünü de kısaltabilir.

1.2.6.2 Güç kaynağı

Tanım 1.2.16

Güç kaynağı (*power supply unit*, PSU), şebekedeki **alternatif akımı** (AC) bilgisayar bileşenlerinin kullandığı **doğru akım** (DC) gerilimlerine dönüştüren birimdir.

Prizden gelen elektrik, Türkiye'de 230 V ve 50 Hz'lik alternatif akımdır: gerilimin yönü saniyede 50 kez ileri geri değişir. Bilgisayarın devreleri ise yönü değişmeyen, sabit ve düşük gerilimler ister. Güç kaynağı bu dönüşümü yapar ve farklı bileşenler için farklı gerilimler üretir (Şekil 1.2.19).



Şekil 1.2.19: Güç kaynağı, şebekedeki 230 V AC'yi üç DC gerilime çevirir: +12 V işlemciyi, ekran kartını ve fanları; +5 V USB ve SATA sürücüleri; +3,3 V ise M.2 aygıtlarını ve anakart devrelerini besler.

Bir güç kaynağı seçilirken dört ölçüte bakılır:

- **Kapasite (W):** Güç kaynağının verebileceği en yüksek güçtür. Bileşenlerin toplam ihtiyacını güvenli bir payla karşılamalıdır.
- **Verimlilik:** Şebekeden çekilen gücün ne kadarının bileşenlere ulaştığıdır. Geri kalanı ısıya dönüşür.
- **Bağlantılar:** Anakart (24-pin), işlemci (8-pin), ekran kartı ve depolama kabloları sisteme uygun olmalıdır.
- **Koruma:** Aşırı akım, aşırı gerilim ve kısa devre durumunda güç kaynağı kendini ve bileşenleri korumalıdır.

Örnek 1.2.11

Bir sistemde işlemci 125 W, ekran kartı 220 W, diğer bileşenler 75 W çekiyor. %30 güvenlik payı bırakılacak. Piyasada 450, 550, 650 ve 750 W'lık modeller var. Hangisi seçilir?

1. **Toplam ihtiyaç.** 125 + 220 + 75 = 420 W.
2. **Pay eklenmiş ihtiyaç.** %30 pay, ihtiyacı 1,3 ile çarpmak demektir: 420 × 1,3 = 546 W.
3. **Model seçimi.** 546 W'tan küçük olmayan en küçük model **550 W**'tır.

Pay bırakılmasının iki nedeni vardır. Bileşenler kısa anlarda belirtilen değerin üstünde güç çekebilir. Ayrıca güç kaynakları sınırlarına çok yakın çalıştıklarında daha çok ısınır ve verimleri düşer.

Örnek 1.2.12

%90 verimli bir güç kaynağı bileşenlere 500 W veriyor. Şebekeden ne kadar güç çeker? Ne kadarı ısıya dönüşür?

1. Verim = verilen güç ÷ çekilen güç olduğu için çekilen güç = 500 ÷ 0,90 ≈ **555,6 W**'tır.
2. Isıya dönüşen kısım = 555,6 − 500 ≈ **55,6 W**'tır.

Aynı yükte %80 verimli bir güç kaynağı 500 ÷ 0,80 = 625 W çekerdi; ısıya dönüşen kısım 125 W olurdu. Verimli güç kaynağı hem elektrik faturasını hem kasa içindeki ısıyı azaltır.

Sık yapılan hata

Güç kaynağının watt değerini, bilgisayarın sürekli çektiği güç sanmak. 650 W'lık bir güç kaynağı, sistem yalnızca 200 W istiyorsa 200 W civarında çeker. Watt değeri bir üst sınırdır, sürekli tüketim değildir.

1.2.7 Giriş birimleri

Bir bilgisayarın dış dünyadan bilgi alması giriş birimleriyle olur. Klavye ve fare en tanıdık örneklerdir; ama bir telefonun konum algılayıcısı, bir akıllı saatin nabız ölçeri ya da bir fabrikadaki sıcaklık sensörü de birer giriş birimidir.

1.2.7.1 Giriş birimlerinin türleri

Tanım 1.2.17

Giriş birimi (*input device*), kullanıcıdan ya da çevreden gelen bilgiyi bilgisayarın işleyebileceği sayısal veriye dönüştürüp sisteme ileten aygıttır.

Aygıt	Aldığı bilgi
Klavye	metin, sayı ve komut
Fare, dokunmatik yüzey	işaretleme, seçme ve hareket
Mikrofon	ses
Kamera, tarayıcı	görüntü ve belge
Oyun denetleyicisi	eksen, düğme ve hareket bilgisi
Sensör	sıcaklık, konum, ışık ya da hareket ölçümü

Tablo 1.2.9: Başlıca giriş birimleri ve aldıkları bilgi.

Bütün giriş birimlerinde ortak bir adım vardır: fiziksel bir büyüklük sayıya çevrilir. Klavyede bir tuşun basılması bir tuş koduna, mikrofondaki ses örneklere (1.2.5.4), kameradaki ışık her piksel için üç renk değerine dönüşür. Bilgisayar bu sayıları işler; onların bir zamanlar ses ya da ışık olduğunu bilmez.

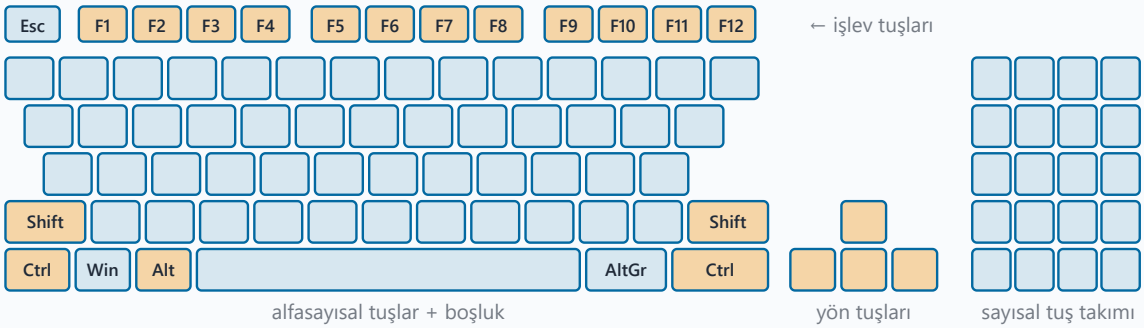
"Tarayıcı" sözcüğüne dikkat etmek gerekir. Burada kâğıt belgeyi görüntüye çeviren aygıt (*scanner*) kastedilir. İnternette gezinmeye yarayan program da Türkçede tarayıcı olarak anılır, ama o bir yazılımdır, giriş birimi değildir.

1.2.7.2 Klavye

Klavye, metin girişinin temel aracıdır. Tuşları görevlerine göre bölgelere ayrılır (Şekil 1.2.20):

- **Alfasayısal tuşlar** harfleri, rakamları ve noktalama işaretlerini girer.
- **İşlev tuşları** (F1–F12) uygulamaya göre özel görevler üstlenir; örneğin pek çok programda F1 yardımı açar.
- **Denetim tuşları** (Ctrl, Alt, Shift, Windows tuşu) tek başına bir şey yazmaz; başka tuşlarla birlikte basılınca **kısayol** oluşturur.

- **Yön tuşları** imleci hareket ettirir; **sayısal tuş takımı** çok sayıda rakam girmeyi kolaylaştırır.



Şekil 1.2.20: Klavyenin bölgeleri: üstte işlev tuşları, ortada alfasayısal bölüm ve denetim tuşları (Shift, Ctrl, Win, Alt, AltGr), sağda yön tuşları ve sayısal tuş takımı.

Metin düzenlerken en sık kullanılan tuşlar Enter, Backspace ve Delete'tir. Enter bir girişi onaylar ya da yeni satıra geçer. **Backspace**, imlecin (*cursor*) *solundaki* karakteri, **Delete** ise imlecin *sağındaki* karakteri siler. Seçili bir metin varsa ikisi de seçimin tamamını siler. Sık kullanılan kısayollar şunlardır:

Kısayol	İşlev
Ctrl + C	Kopyala
Ctrl + X	Kes
Ctrl + V	Yapıştır
Ctrl + Z	Geri al
Alt + Tab	Açık uygulamalar arasında geçiş

Tablo 1.2.10: Yaygın klavye kısayolları.

Kopyala ve kes komutları seçili metni **pano** adı verilen geçici bir alana koyar. Yapıştır, panodaki içeriği imlecin bulunduğu yere ekler; pano bu sırada değişmez. Bu yüzden aynı içerik art arda birçok kez yapıştırılabilir.

Örnek 1.2.13

Bir metin kutusunda “Ders” yazıyor ve imleç metnin sonunda. Pano boş. Sırayla şu işlemler yapılıyor: bütün metin seçilir ve Ctrl + C'ye basılır; seçim kaldırılıp imleç sona alınır; Ctrl + V; Backspace; Ctrl + Z. Her adımdan sonra metin ve pano nedir? (İmlecin yeri `|` ile gösterilmiştir.)

Adım	İşlem	Metin	Pano
0	başlangıç	`Ders|`	boş
1	tümünü seç, Ctrl + C	`Ders` (seçili)	`Ders`
2	seçimi kaldır, imleç sona	`Ders|`	`Ders`

Adım	İşlem	Metin	Pano
3	Ctrl + V	`DersDers|`	`Ders`
4	Backspace	`DersDer|`	`Ders`
5	Ctrl + Z	`DersDers|`	`Ders`

Ctrl + C metni değiştirmez, yalnızca panoyu doldurur. Backspace imlecin solundaki “s” harfini siler. Ctrl + Z son işlemi, yani silmeyi geri alır. Pano baştan sona “Ders” olarak kalır.

Sık yapılan hata

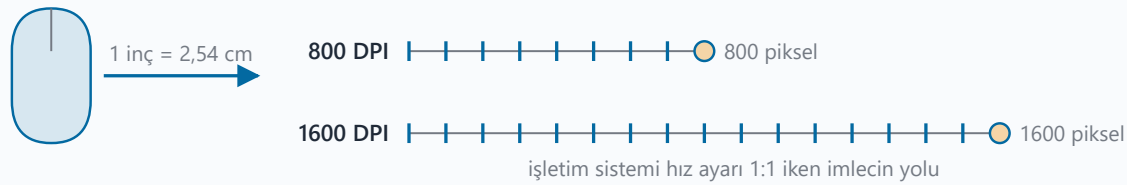
Delete ile Backspace'in aynı karakteri sildiğini sanmak. İmleç metnin sonundayken Delete hiçbir şey silmez, çünkü imlecin sağında karakter yoktur. Backspace ise sondaki karakteri siler.

1.2.7.3 Fare ve işaretleme aygıtları

Fare, ekrandaki imleci yönlendirmeye yarayan bir **işaretleme aygıtıdır** (*pointing device*). Altındaki optik algılayıcı yüzeyin fotoğrafını saniyede binlerce kez çeker, ardışık görüntüleri karşılaştırarak ne kadar ve hangi yöne kaydığını hesaplar. Farenin teknik belirtimlerinde iki sayı öne çıkar.

Tanım 1.2.18

DPI (*dots per inch*), fare 1 inç (2,54 cm) hareket ettiğinde bilgisayara bildirdiği adım sayısıdır. **Yoklama hızı** (*polling rate*) ise farenin konumunu saniyede kaç kez gönderdiğidir ve Hz ile ölçülür.



Şekil 1.2.21: Fare 1 inç hareket ettiğinde 800 DPI'lık fare 800, 1600 DPI'lık fare 1600 adım bildirir. İşletim sisteminin hız ayarı 1:1 iken imleç de o kadar piksel ilerler.

Örnek 1.2.14

1600 DPI'lık bir fare masada 3 inç (yaklaşık 7,6 cm) kaydırılıyor. Hız ayarı 1:1'dir. İmleç kaç piksel ilerler? Farenin yoklama hızı 500 Hz ise konum kaç ms'de bir gönderilir?

1. **Adım sayısı.** 1600 adım/inç × 3 inç = **4800 adım**. Hız ayarı 1:1 olduğu için imleç 4800 piksel ilerler; bu, 1920 piksel genişliğindeki bir ekranın iki katından fazladır.
2. **Yoklama aralığı.** Saniyede 500 kez gönderim: 1000 ms ÷ 500 = **2 ms**'de bir. 1000 Hz'lik farede bu süre 1 ms'ye iner.

Yüksek DPI her zaman daha iyi değildir. Çok yüksek DPI'da en küçük el titremesi bile imleci birkaç piksel oynatır; hassas iş yapmak zorlaşır. Uygun değer ekranın çözünürlüğüne ve kullanım alışkanlığına göre seçilir. Farenin tuşları ve tekerleği de seçme, bağlam menüsünü açma, sayfayı kaydırma gibi işler görür.

Fareye seçenek olan işaretleme aygıtları da vardır: dizüstü bilgisayarlardaki **dokunmatik yüzey** (*touchpad*), çizim için **kalem**, elle döndürülen bir topla çalışan **iz topu** ve doğrudan ekrana dokunulan **dokunmatik ekran**. Dokunmatik ekran, bir sonraki kesimde görüleceği gibi, aynı zamanda bir çıkış birimidir.

1.2.8 Çıkış birimleri

Çıkış birimleri, bilgisayarın ürettiği sayısal sonucu insanın algılayabileceği bir biçime çevirir: ışığa, sese, kâğıt üzerindeki mürekkebe ya da titreşime.

1.2.8.1 Çıkış birimleri ve sınıflandırma

Tanım 1.2.19

Çıkış birimi (*output device*), bilgisayarın ürettiği veriyi kullanıcıya görüntü, ses, baskı ya da hareket olarak sunan aygıttır.

Aygıt	Ürettiği çıktı
Monitör	metin, görüntü ve video
Yazıcı	dijital belgenin kâğıt üzerindeki kopyası
Hoparlör, kulaklık	ses
Projektör	büyük bir yüzeye yansıtılan görüntü
Dokunsal geri bildirim	titreşim ya da kuvvet
Çok işlevli aygıt	giriş ve çıkış bir arada (ör. tarayıcılı yazıcı)

Tablo 1.2.11: Başlıca çıkış birimleri.

Bir aygıtın giriş mi çıkış mı olduğunu belirleyen tek ölçüt **veri akışının yönüdür**. Veri bilgisayara doğru akıyorsa aygıt giriş, kullanıcıya doğru akıyorsa çıkış birimidir. Bazı aygıtlarda veri iki yönde de akar (Şekil 1.2.22).



Şekil 1.2.22: Giriş, çıkış ve hem giriş hem çıkış birimleri. Dokunmatik ekran, mikrofonlu kulaklık, çok işlevli yazıcı, titreşimli oyun kolu, ağ kartı ve depolama birimleri iki yönlü çalışır.

Örnek 1.2.15

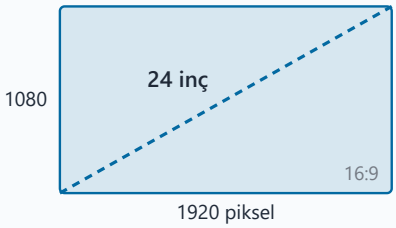
Web kamerası, akıllı saat ekranı (dokunmatik), yazıcı, harici SSD ve oyun direksiyonu (kuvvet geri bildirimli) hangi gruba girer?

- **Web kamerası:** görüntüyü bilgisayara verir → giriş.
- **Akıllı saat ekranı:** dokunuşu alır, görüntüyü gösterir → ikisi.
- **Yazıcı:** belgeyi kâğıda aktarır → çıkış (yalnız baskı yapan bir yazıcı için).
- **Harici SSD:** hem okunur hem yazılır → ikisi.
- **Oyun direksiyonu:** dönüş açısını gönderir, yoldaki sarsıntıyı direksiyona kuvvet olarak geri verir → ikisi.

1.2.8.2 Monitör: çözünürlük, boyut ve en-boy oranı

Monitör, ekran yüzeyini milyonlarca küçük noktaya bölerek görüntü oluşturur. Bu noktalardan her birine **piksel** denir. Bir monitörün belirtimlerinde dört temel değer bulunur:

- **Çözünürlük** (*resolution*): yatay ve dikey piksel sayısıdır; örneğin 1920 × 1080.
- **Ekran boyutu**: köşegen uzunluğudur ve inç ile ölçülür; örneğin 24 inç.
- **Piksel yoğunluğu**: bir ince kaç piksel düştüğüdür (1.2.8.3).
- **En-boy oranı** (*aspect ratio*): ekranın genişliğinin yüksekliğine oranıdır; örneğin 16:9.



Köşegendeki piksel sayısı:

$\sqrt{(1920^2 + 1080^2)} \approx 2203$ piksel

Piksel yoğunluğu:

$2203 \div 24 \approx 92$ PPI

Aynı 1920 × 1080, 15,6 inç dizüstünde ≈ 141 PPI

Şekil 1.2.23: 24 inç, 16:9 bir monitör: 1920 piksel genişlik, 1080 piksel yükseklik. Köşegendeki piksel sayısı $\sqrt{(1920^2 + 1080^2)} \approx 2203$'tür; piksel yoğunluğu $2203 \div 24 \approx 92$ PPI olur. Aynı çözünürlük 15,6 inçlik bir dizüstünde yaklaşık 141 PPI verir.

1920 × 1080 çözünürlüğün en-boy oranı 16:9'dur, çünkü $1920 \div 1080 = 16 \div 9 \approx 1{,}78$'dir. Ekran boyutu köşegen olarak verildiği için genişlik ve yükseklik doğrudan yazılmaz; ama en-boy oranından hesaplanabilir.

Örnek 1.2.16

24 inç, 16:9 bir monitörün ekranı kaç cm genişlik ve yüksekliktedir?

1. **Oranın köşegeni.** Genişlik 16 birim, yükseklik 9 birim alınırsa Pisagor teoremine göre köşegen $\sqrt{(16^2 + 9^2)} = \sqrt{(256 + 81)} = \sqrt{337} \approx 18{,}36$ birimdir.
2. **Bir birim kaç inç?** Köşegen 24 inç olduğuna göre 1 birim = $24 \div 18{,}36 \approx 1{,}307$ inçtir.
3. **Genişlik ve yükseklik.** Genişlik $16 \times 1{,}307 \approx 20{,}92$ inç, yükseklik $9 \times 1{,}307 \approx 11{,}77$ inçtir.
4. **Santimetreye çevirme.** 1 inç = 2,54 cm: genişlik $\approx 20{,}92 \times 2{,}54 \approx$ **53,1 cm**, yükseklik $\approx 11{,}77 \times 2{,}54 \approx$ **29,9 cm**.

1.2.8.3 Piksel yoğunluğu

Aynı çözünürlük, küçük bir ekranda daha keskin görünür. Bunun nedeni, aynı sayıda pikselin daha küçük bir alana sığdırılmasıdır. Pikseller küçüldükçe göz onları tek tek ayırt edemez ve kenarlar daha düzgün görünür (Şekil 1.2.24).



Şekil 1.2.24: Aynı daire 8 × 8 = 64 ve 24 × 24 = 576 piksellik ızgaralarda. Piksel sayısı arttıkça kenarlar daha düzgün görünür. Her piksel kırmızı, yeşil ve mavi üç alt pikselden oluşur.

Şeklin altındaki üç renkli çubuk önemli bir ayrıntıyı gösterir: bir piksel aslında üç küçük ışık kaynağından oluşur. Kırmızı, yeşil ve mavi alt piksellerin parlaklıkları ayarlanarak her renk elde edilir. Üçü de tam parlaksa beyaz, üçü de sönükse siyah görünür.

Tanım 1.2.20

Piksel yoğunluğu (*pixel density*), ekranın bir inçlik uzunluğuna düşen piksel sayısıdır ve **PPI** (*pixels per inch*, inç başına piksel) ile ölçülür. Köşegen üzerinden hesaplanır:

> PPI = $\sqrt{(\text{yatay piksel}^2 + \text{dikey piksel}^2)} \div$ köşegen (inç)

Formülün mantığı şudur: köşegen uzunluğu inç olarak bilinir; köşegen boyunca kaç piksel olduğu ise Pisagor teoremiyle bulunur. İkincisi birinciye bölünerek bir inçe düşen piksel sayısı elde edilir.

Örnek 1.2.17

27 inç ve 3840 × 2160 (4K) bir monitörün PPI değeri nedir? Aynı boyuttaki 2560 × 1440 bir monitörden ne kadar yoğundur?

1. **4K köşegen pikseli.** $3840^2 = 14.745.600$ ve $2160^2 = 4.665.600$. Toplam 19.411.200; karekökü ≈ 4405,8.
2. **4K PPI.** 4405,8 ÷ 27 ≈ **163,2 PPI**.
3. **2560 × 1440 köşegen pikseli.** $2560^2 + 1440^2 = 6.553.600 + 2.073.600 = 8.627.200$; karekökü ≈ 2937,2.
4. **2560 × 1440 PPI.** 2937,2 ÷ 27 ≈ **108,8 PPI**.

Aynı boyutta 4K ekran yaklaşık 1,5 kat daha yoğundur. Piksel sayısındaki fark daha büyüktür: 3840 × 2160 = 8.294.400 piksel, 2560 × 1440 = 3.686.400 pikselin 2,25 katıdır. Bu yüzden 4K ekran ekran kartını da daha çok yorar.

Sık yapılan hata

Çözünürlüğü tek başına keskinlik ölçüsü saymak. 32 inçlik 4K bir monitör yaklaşık 138 PPI'dır; 14 inçlik 1920 × 1080 bir dizüstü ise yaklaşık 157 PPI'dır. Çözünürlüğü daha düşük olan dizüstü ekranı daha keskin görünür. Keskinliği belirleyen, çözünürlük ile boyutun birlikte verdiği piksel yoğunluğudur; izleme mesafesi de rol oynar.

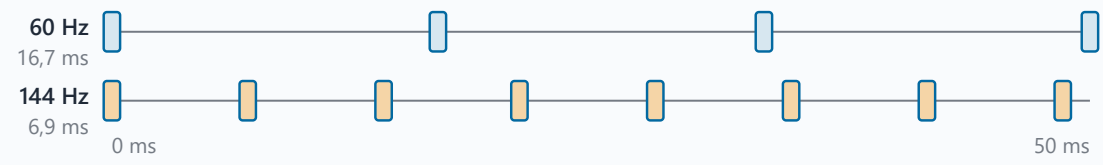
1.2.8.4 Yenileme hızı ve tepki süresi

Ekrandaki hareket aslında art arda gösterilen durağan karelerden oluşur. Kareler yeterince hızlı değişirse göz bunları kesintisiz bir hareket olarak algılar. Monitörün hareketi ne kadar iyi gösterdiğini birkaç özellik belirler:

Özellik	Ne ifade eder?
Yenileme hızı	Ekranın saniyede kaç kez yenilendiğidir; Hz ile ölçülür.
Tepki süresi	Bir pikselin bir renkten ötekine geçme süresidir.
Parlaklık	Ekranın ışık çıkışıdır (cd/m^2).
Renk doğruluğu	Renklerin hedef değerlere ne kadar yakın gösterildiğidir.
Görüş açısı	Yandan bakıldığında görüntünün ne kadar korunduğudur.

Tablo 1.2.12: Monitörün hareket ve görüntü özellikleri.

Yenileme hızı (*refresh rate*) 60 Hz olan bir ekran, görüntüsünü saniyede 60 kez yeniler. Bir karenin ekranda kalma süresi yenileme hızının tersidir: 1000 ms ÷ 60 ≈ 16,7 ms. Yenileme hızı arttıkça bu süre kısalır ve hareket daha akıcı görünür (Şekil 1.2.25).



Şekil 1.2.25: 50 ms içinde 60 Hz ekran 3, 144 Hz ekran 7 yeni kare gösterir. Bir karenin süresi 60 Hz'de 16,7 ms, 144 Hz'de 6,9 ms'dir.

Örnek 1.2.18

60, 144 ve 240 Hz ekranlarda bir kare kaç ms ekranda kalır? 50 ms içinde kaçar yeni kare gösterilir?

Yenileme hızı	Kare süresi (1000 ÷ Hz)	50 ms'deki kare (50 ÷ süre)
60 Hz	≈ 16,7 ms	3
144 Hz	≈ 6,9 ms	7,2 → 7 tam kare
240 Hz	≈ 4,2 ms	12

Ekranın yenileme hızı yüksek olsa bile ekran kartı o kadar kare üretemiyorsa kazanç görülmez. 144 Hz'lik bir monitörün yararlı olması için oyunun ya da uygulamanın saniyede 144 kareye yakın üretebilmesi gerekir.

Tepki süresi (*response time*) ise başka bir şeyi ölçer: bir pikselin, kendisine yeni renk söylendikten sonra o renge ne kadar sürede ulaştığını. Tepki süresi uzun bir ekranda hızlı hareket eden nesnelerin arkasında silik bir iz kalır. İki özellik birbirini tamamlar: yenileme hızı yeni karelerin ne sıklıkla geldiğini, tepki süresi her karenin ne kadar temiz çizildiğini belirler.

Sık yapılan hata

Yenileme hızı ile tepki süresini aynı şey sanmak. 144 Hz'lik bir ekranda kare süresi 6,9 ms'dir; ama aynı ekranın tepki süresi 1 ms de olabilir, 5 ms de. Biri saniyedeki kare sayısını, öbürü bir pikselin renk değiştirme hızını ölçer.

1.2.8.5 Panel teknolojileri: LCD ve OLED

Monitörün ışığı nasıl ürettiği, görüntünün kontrastını, renklerini ve görüş açısını belirler. Bugün iki ana teknoloji kullanılır: LCD ve OLED.



Şekil 1.2.26: LCD ve OLED katmanları. LCD'de en altta hep açık bir arka ışık (LED) vardır; ışık polarizörden, sıvı kristalden ve renk filtresinden geçer. Siyah için sıvı kristal ışığı engellemeye çalışır, ama arka ışık açık kalır. OLED'de her piksel kendi ışığını üretir; siyah piksel tamamen kapalıdır.

LCD (sıvı kristal ekran) panelinde ışığı bir **arka ışık** (*backlight*) üretir. Önündeki **sıvı kristal** katmanı, her alt piksel için geçen ışık miktarını ayarlayan küçük bir perde gibi çalışır. Renk filtresi de geçen ışığa kırmızı, yeşil ya da mavi rengini verir. Siyah göstermek için perde kapanır; ama perde ışığı tamamen engelleyemez ve bir miktar ışık sızar. Bu yüzden karanlık bir odada LCD'nin siyahı koyu gri görünür.

OLED (organik ışık yayan diyot) panelinde arka ışık yoktur. Her piksel kendi ışığını üretir. Siyah bir piksel tamamen sönüktür; hiç ışık vermez. Bu yüzden OLED'in kontrastı çok yüksektir ve siyah gerçekten siyahtır.

LCD panellerin kendi içinde de türleri vardır. IPS, VA ve TN birer LCD türüdür; aralarındaki fark sıvı kristallerin diziliş biçimindedir:

Panel	Güçlü yanı	Zayıf yanı
IPS (LCD)	geniş görüş açısı, dengeli renkler	siyah düzeyi orta
VA (LCD)	yüksek kontrast, güçlü siyah	görüş açısı IPS'ten dar
TN (LCD)	düşük tepki süresi	görüş açısı ve renkler sınırlı
OLED	her piksel kendi ışığını üretir, çok yüksek kontrast	genellikle daha pahalı

Tablo 1.2.13: Panel teknolojileri.

Panel seçimi kullanım amacına göre yapılır. Ofis işleri ve tasarım için renk doğruluğu ve görüş açısı önemlidir; IPS bu yüzden yaygındır. Karanlık ortamda film izlemek için kontrast öne çıkar; VA ve OLED burada güçlüdür. Rekabetçi oyunlarda tepki süresi ve yenileme hızı belirleyicidir. Bütçe de her seçimde ayrı bir sınırdır.

1.2.8.6 Yazıcı, ses ve projektör

Yazıcılar dijital bir belgeyi kâğıda aktarır. Günümüzde iki teknoloji yaygındır (Şekil 1.2.27):

- **Mürekkep püskürtmeli yazıcı**, kafasındaki küçük deliklerden kâğıda mikroskobik mürekkep damlaları püskürtür. Renkli belge ve fotoğraf baskısında yaygındır.
- **Lazer yazıcı**, bir lazer ışınıyla dönen bir tambur üzerinde elektrik yükü deseni oluşturur. Toz hâlindeki boya (toner) bu desene yapışır, kâğıda aktarılır ve ısıyla kâğıda sabitlenir. Yüksek hacimli metin baskısında hızlı ve ekonomik olabilir.



Şekil 1.2.27: Mürekkep püskürtmeli yazıcı kâğıda damlacık püskürtür. Lazer yazıcıda lazer tambur üzerinde bir yük deseni oluşturur; toner bu desene yapışır, kâğıda aktarılır ve ısıyla sabitlenir.

Hoparlör ve kulaklık, ses kartının DAC'sinden gelen analog sinyali ya da USB, Bluetooth gibi dijital bir bağlantıdan gelen veriyi sese çevirir. Dijital bağlantıda DAC hoparlörün ya da kulaklığın içindedir.

Projektör, görüntüyü bir perdeye ya da duvara yansıtır. Seçiminde üç değer öne çıkar: parlaklık, çözünürlük ve yansıtma mesafesi. Aydınlık bir sınıfta görüntünün seçilebilmesi için parlaklık yüksek olmalıdır. Yansıtma mesafesi ise belirli bir görüntü boyutu için projektörün perdeden ne kadar uzağa konması gerektiğini gösterir.

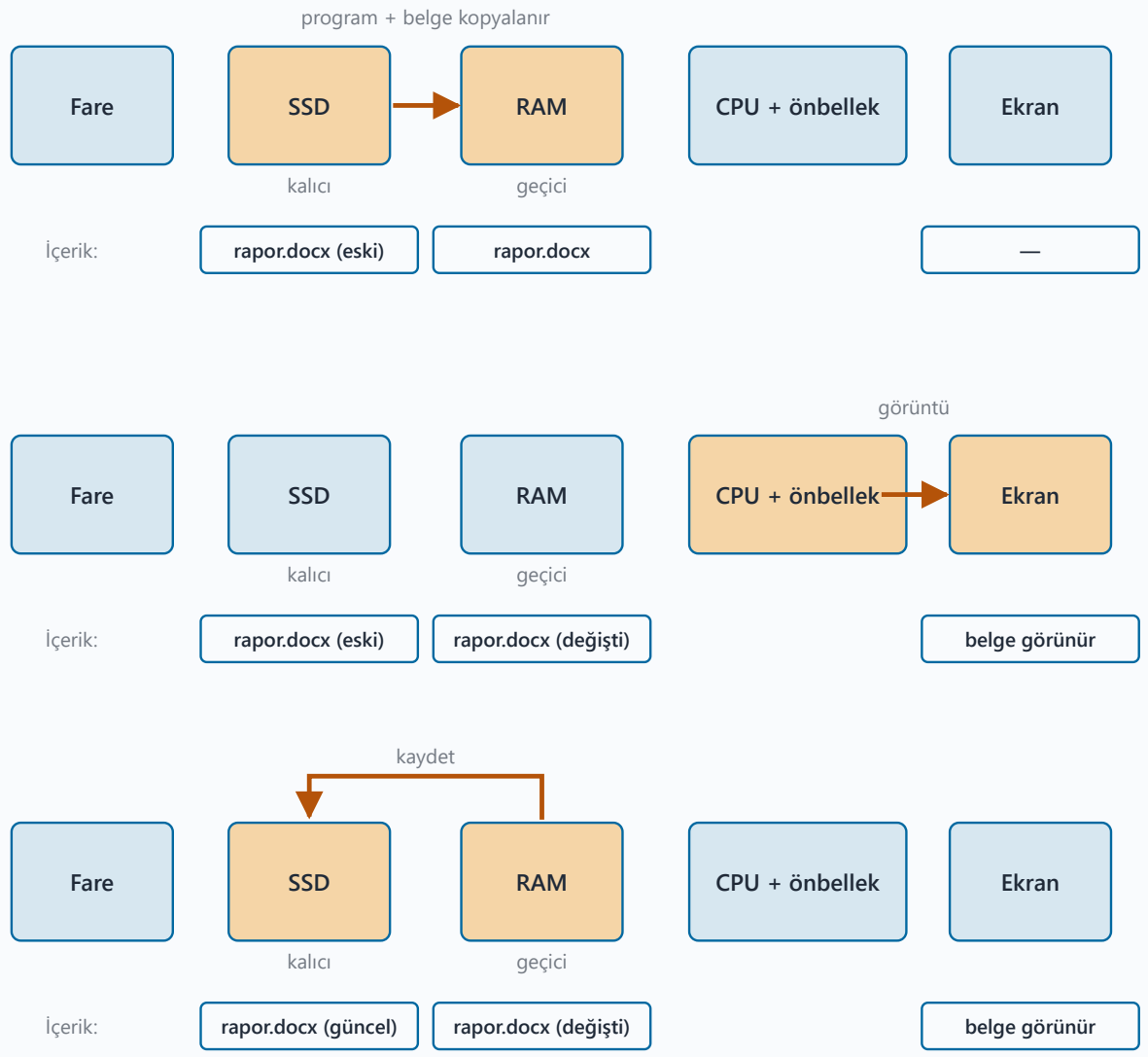
1.2.9 Depolama

RAM'deki her şey bilgisayar kapanınca kaybolur. Belgelerin, fotoğrafların, programların ve işletim sisteminin kalıcı olarak saklanması için ayrı bir birim gerekir. Bu kesim depolama birimlerini, aralarındaki hız farklarını ve bu farkların günlük kullanımdaki karşılığını ele alır.

> **Tanım 1.2.21**
>
> **Depolama birimi** (*storage*), veriyi elektrik kesildiğinde de koruyan, yani **kalıcı** (*non-volatile*) bellektir. Başlıca türleri sabit disk (HDD), katı hâl sürücüsü (SSD), USB bellek, hafıza kartı ve optik disktir.

1.2.9.1 Bir belgenin bileşenler arasındaki yolculuğu

Depolamanın sistemdeki yerini görmenin en iyi yolu, gündelik bir işi izlemektir. `rapor.docx` adlı bir belge açılıp düzenleniyor ve kaydediliyor. Veri bu sırada fare, SSD, RAM, işlemci ve ekran arasında dolaşır (Şekil 1.2.28).



Şekil 1.2.28: Bir belgenin açılıp kaydedilmesi, üç an. Üstte program ve belge SSD'den RAM'e kopyalanır. Ortada belge düzenlenmiştir: RAM'deki kopya değişmiş, SSD'deki kopya hâlâ eskidir; sonuç ekranda görünür. Altta "Kaydet" ile değişiklikler RAM'den SSD'ye yazılır ve SSD'deki kopya güncellenir.

Adım	Ne olur?	SSD'deki belge	RAM'deki belge
1	Kullanıcı `rapor.docx`'a çift tıklar; fare bu olayı bilgisayara iletir (giriş).	eski	—
2	İşletim sistemi program ve belge dosyalarını SSD'de bulur.	eski	—
3	Program ve belge SSD'den RAM'e kopyalanır.	eski	eski (kopya)
4	CPU komutları RAM'den önbelleğe ve yazmaçlara getirip işler (getir–çöz–yürüt).	eski	düzenleniyor
5	Sonuç ekran kartı üzerinden monitöre gider; kullanıcı belgeyi görür ve düzenler (çıkış).	eski	değişti

Adım	Ne olur?	SSD'deki belge	RAM'deki belge
6	“Kaydet” ile değişiklikler RAM'den SSD'ye yazılır.	güncel	değişti

Tablo 1.2.14: Belge açma, düzenleme ve kaydetme adımları.

Tablonun 3. adımı önemli bir gerçeği gösterir: **CPU programı doğrudan SSD'den çalıştırmaz**. Program önce RAM'e kopyalanır; işlemci komutları RAM'den, önbellek üzerinden getirir. SSD işlemcinin beklemeye dayanamayacağı kadar yavaştır (1.2.4.2). Okuma yolu bu yüzden SSD → RAM → önbellek → yazmaçlar sırasını izler; kaydetme ise RAM → SSD yönündedir.

4. ve 5. adımlarda belgenin iki farklı sürümü vardır: RAM'deki düzenlenmiş kopya ve SSD'deki eski kopya. Elektrik bu sırada kesilirse RAM'deki değişiklikler kaybolur, SSD'deki eski sürüm kalır. Pek çok programın belirli aralıklarla otomatik kayıt yapmasının nedeni budur.

Sık yapılan hata

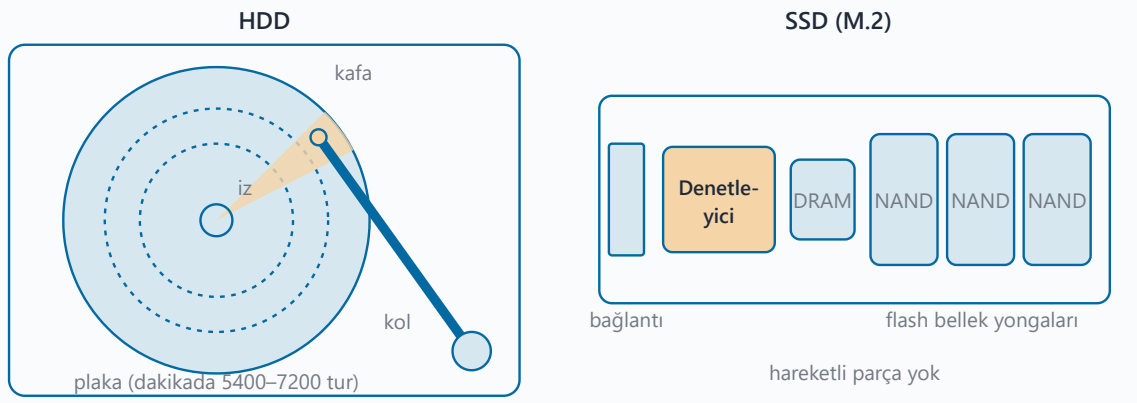
Ekranda görünen belgenin diskte de aynı hâliyle durduğunu sanmak. Ekrandaki görüntü RAM'deki kopyayı yansıtır. Kaydedilene kadar diskteki dosya eski hâlindedir.

1.2.9.2 HDD ve SSD

Bilgisayarlardaki iki ana depolama teknolojisi, veriyi tamamen farklı yollarla saklar.

Tanım 1.2.22

Sabit disk (*hard disk drive*, HDD), veriyi dönen manyetik **plakalar** üzerinde saklayan depolama birimidir. Veri, bir kolun ucundaki **okuma-yazma kafası** ile okunur ve yazılır. **SSD** (*solid-state drive*, katı hâl sürücüsü) ise veriyi hareketli parçası olmayan **flash bellek** yongalarında saklar.



Şekil 1.2.29: Solda HDD'nin iç yapısı: dakikada 5400–7200 tur dönen plaka, plakanın üzerindeki izler, kol ve kafa. Sağda bir M.2 SSD: denetleyici, önbellek olarak kullanılan DRAM ve NAND flash bellek yongaları. SSD'de hareketli parça yoktur.

HDD'de verinin okunması mekanik bir iştir. Kafa doğru **ize** (plakadaki eş merkezli halkalardan birine) gitmeli, sonra okunacak **sektör** (izin küçük bir parçası) dönerek kafanın altına gelmelidir. Bu iki hareket milisaniyeler sürer. SSD'de ise **denetleyici** veriyi birden çok flash yongasından aynı anda okur; erişim mikrosaniyeler düzeyindedir.

Özellik	HDD	SSD
Teknoloji	dönen manyetik disk	flash bellek
Hız	daha düşük	daha yüksek
Darbe dayanımı	mekanik parça nedeniyle daha hassas	hareketli parça yok
Kapasite başına maliyet	genellikle daha düşük	genellikle daha yüksek
Yaygın kullanım	arşiv ve yüksek kapasite	işletim sistemi ve uygulamalar

Tablo 1.2.15: HDD ve SSD karşılaştırması.

Bu karşılaştırmadan yaygın bir düzen çıkar. İşletim sistemi ve programlar hızlı açılsın diye SSD'ye konur. Fotoğraf arşivi ve yedekler gibi büyük ama seyrek kullanılan veriler ise daha ucuz olan HDD'de tutulur. Dizüstü bilgisayarlarda ise darbe dayanımı, düşük güç tüketimi ve sessizlik nedeniyle neredeyse yalnızca SSD kullanılır.

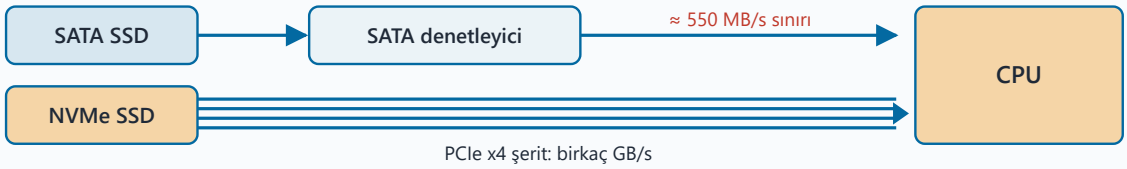
1.2.9.3 SATA ve NVMe

SSD'nin hızı yalnızca flash yongalarına bağlı değildir; SSD'nin bilgisayara hangi yoldan bağlandığı da belirleyicidir. İki arayüz yaygındır.

Özellik	SATA SSD	NVMe SSD
Arayüz	SATA (sabit diskler için tasarlanmış eski arayüz)	PCIe üzerinden NVMe protokolü

Özellik	SATA SSD	NVMe SSD
Tipik sıralı okuma	yaklaşık 550 MB/s (SATA III sınırı)	birkaç GB/s (PCIe sürümüne ve şerit sayısına bağlı)
Paralellik	tek komut kuyruğu	çok sayıda paralel kuyruk; düşük gecikme
Biçim	2,5 inç kasa ya da M.2	çoğunlukla M.2 (bazı sunucularda U.2)

Tablo 1.2.16: SATA SSD ve NVMe SSD.



Şekil 1.2.30: SATA SSD'nin verisi SATA denetleyicisinden geçer ve yaklaşık 550 MB/s ile sınırlanır. NVMe SSD ise dört PCIe şeridiyle doğrudan işlemciye bağlanır ve saniyede birkaç GB aktarabilir.

SATA, dönen disklerin hızına göre tasarlanmıştı. SSD'ler bu sınırı kısa sürede aştı: SATA III'ün kuramsal sınırı 6 Gbit/s'dir; veri kodlaması için ayrılan bitler düşüldüğünde bu yaklaşık 600 MB/s eder. SATA SSD'lerin pratikte 550 MB/s civarında kalmasının nedeni budur. NVMe (*Non-Volatile Memory Express*) ise doğrudan flash bellek için tasarlanmış bir protokoldür ve PCIe şeritlerini kullanır.

Örnek 1.2.19

Bir PCIe şeridi, PCIe 3.0'da saniyede yaklaşık 0,985 GB taşır ve her yeni nesilde bu değer iki katına çıkar. Dört şeritli (x4) bir NVMe SSD'nin kuramsal sınırı her nesilde ne olur?

1. **PCIe 3.0 x4.** 4 × 0,985 ≈ **3,94 GB/s**.
2. **PCIe 4.0 x4.** Şerit başına 2 × 0,985 ≈ 1,97 GB/s; 4 × 1,97 ≈ **7,88 GB/s**.
3. **PCIe 5.0 x4.** Şerit başına 2 × 1,97 ≈ 3,94 GB/s; 4 × 3,94 ≈ **15,75 GB/s**.

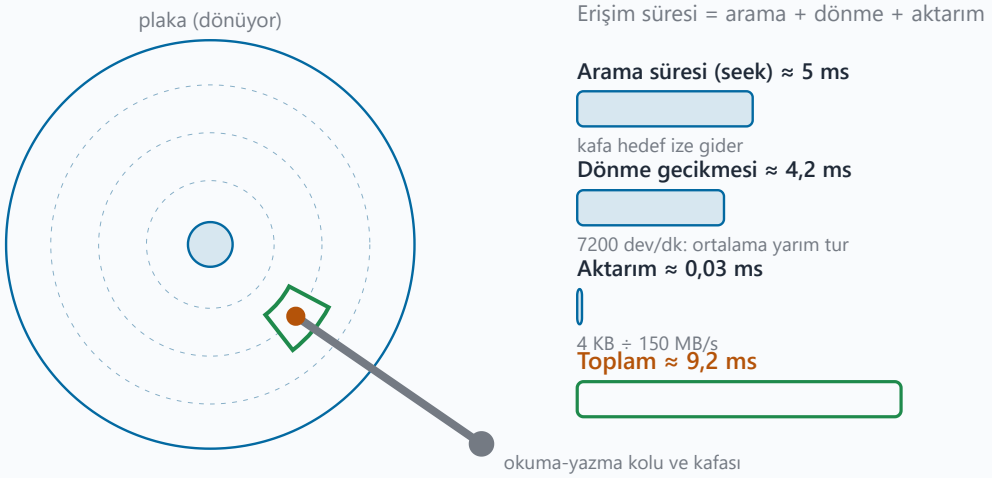
SATA'nın 0,6 GB/s'lik sınırıyla karşılaştırıldığında PCIe 4.0 x4 bağlantısı yaklaşık 13 kat daha geniştir. Gerçek SSD'ler bu sınırlara yaklaşabilir, ama aşamaz.

Sık yapılan hata

M.2'yi bir protokol sanmak ve her M.2 SSD'nin NVMe hızında çalışacağını düşünmek. M.2 yalnızca bir biçimdir, yani kartın şekli ve yuvası. Bir M.2 yuvası SATA, NVMe ya da her ikisini destekleyebilir; hangisinin desteklendiği anakart kılavuzunda yazar. M.2 biçimindeki bir SATA SSD, 550 MB/s sınırında kalır.

1.2.9.4 HDD'de bir sektörün okunması

HDD'nin neden yavaş olduğunu görmek için tek bir sektörün okunmasını adım adım izleyelim. Plaka dakikada 7200 tur döner. Okunacak sektör ile okuma-yazma kafası başlangıçta farklı yerlerdedir. Erişim süresi üç parçadan oluşur (Şekil 1.2.31).



Şekil 1.2.31: HDD'de bir sektörün okunması. Kafa önce hedef ize gider (arama, ≈ 5 ms); sonra sektörün dönüp kafanın altına gelmesi beklenir (dönme gecikmesi, ≈ 4,2 ms); son olarak veri okunur (aktarım, ≈ 0,03 ms). Toplam ≈ 9,2 ms.

Tanım 1.2.23

HDD'de bir sektörün **erişim süresi** üç parçanın toplamıdır:

> erişim süresi = arama süresi + dönme gecikmesi + aktarım süresi

Arama süresi (*seek time*), kolun kafayı hedef ize götürme süresidir. **Dönme gecikmesi** (*rotational latency*), sektörün dönüp kafanın altına gelmesi için beklenen süredir; ortalama yarım tur kabul edilir. **Aktarım süresi**, sektör kafanın altından geçerken verinin okunma süresidir.

Örnek 1.2.20

7200 dev/dk dönen bir HDD'de ortalama arama süresi 5 ms, aktarım hızı 150 MB/s'dir. 4 KB'lık bir sektörün ortalama erişim süresi nedir? (1 MB = 1000 KB alınır.)

1. **Arama.** Kol kafayı hedef ize götürür: **5 ms**.
2. **Bir tam tur.** 1 dakika = 60.000 ms. 60.000 ÷ 7200 ≈ 8,33 ms'de bir tur atılır.

3. **Dönme gecikmesi.** Sektör ortalama yarım tur uzaktadır: 8,33 ÷ 2 ≈ **4,17 ms** (yaklaşık 4,2 ms).
4. **Aktarım.** 4 KB = 0,004 MB. 0,004 ÷ 150 ≈ 0,0000267 s ≈ **0,03 ms**.
5. **Toplam.** 5 + 4,17 + 0,03 ≈ **9,2 ms**.

Toplamın 9,17 ms'si mekanik beklemedir; asıl okuma 0,03 ms'dir. Bekleme, okumanın 300 katından fazladır.

Bu hesap, HDD'nin büyük ve tek parça dosyalarda neden kabul edilebilir, çok sayıda küçük dosyada ise neden çok yavaş olduğunu açıklar. Büyük bir dosya plakada art arda duran sektörlerde saklanır: kafa bir kez konumlanır ve veri sürekli okunur. 100 küçük dosyanın her biri ise ayrı bir konumdadır. Her dosya için arama ve dönme beklemesi yeniden yaşanır: 100 × 9,2 ms = 920 ms, yani neredeyse bir saniye. Aynı iş, erişimi yaklaşık 0,1 ms olan bir SSD'de 100 × 0,1 = 10 ms sürer.

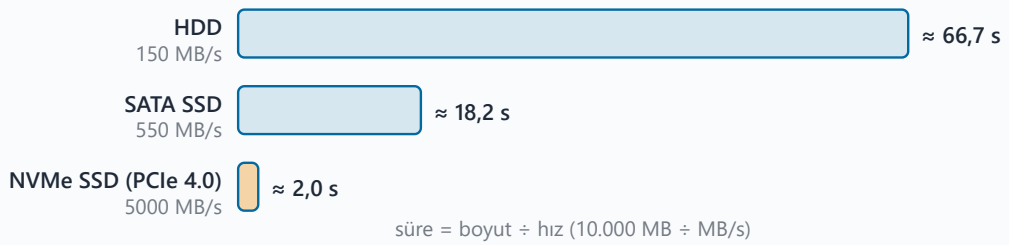
Sık yapılan hata

Dönme gecikmesi için tam tur süresini almak. Sektör kafanın hemen önünde de olabilir, tam bir tur ötede de. Ortalama durumda yarım tur beklenir; bu yüzden tur süresi 2'ye bölünür.

1.2.9.5 Kopyalama süresi ve kapasite birimleri

Büyük bir dosyanın kopyalanma süresi, depolama biriminin **sıralı okuma** (*sequential read*) hızıyla tahmin edilebilir. Sıralı okuma, verinin art arda duran bloklardan sürekli okunmasıdır; büyük dosyalarda bu hız belirleyicidir.

> süre = dosya boyutu ÷ aktarım hızı



Şekil 1.2.32: 10 GB'lık (10.000 MB) bir dosyanın kopyalanma süresi: HDD (150 MB/s) ≈ 66,7 s, SATA SSD (550 MB/s) ≈ 18,2 s, NVMe SSD (PCIe 4.0, 5000 MB/s) ≈ 2,0 s.

Örnek 1.2.21

10 GB'lık bir video dosyası üç farklı birimden okunuyor. Süreler nedir?

1. **Birim çevirme.** Üreticilerin kullandığı tabanla 10 GB = 10.000 MB'dir.
2. **HDD.** 10.000 ÷ 150 ≈ **66,7 s**.
3. **SATA SSD.** 10.000 ÷ 550 ≈ **18,2 s**.
4. **NVMe SSD.** 10.000 ÷ 5000 = **2,0 s**.

Bu değerler büyük ve tek parça dosyalar için geçerlidir. Çok sayıda küçük dosyada HDD, kafa hareketi nedeniyle çok daha fazla yavaşlar (1.2.9.4). Kopyalamada hedef birimin yazma hızı da önemlidir: veri en yavaş birimin hızında akar.

Kapasite birimlerinde iki ayrı gelenek vardır ve bu fark sık sık karışıklığa yol açar. Disk üreticileri SI önekleriyle onluk tabanı kullanır: 1 GB = 1000 MB = 10^9 bayt. İşletim sistemleri ise kapasiteyi çoğunlukla 1024 tabanıyla hesaplar: 1 GiB (gibibayt) = 1024 MiB = 2^{30} = 1.073.741.824 bayt. Sorun, bazı işletim sistemlerinin bu ikinci değeri "GiB" yerine "GB" diye göstermesidir.

Örnek 1.2.22

Kutusunda "1 TB" yazan bir disk, işletim sisteminde kaç GB görünür?

1. **Üreticinin değeri.** 1 TB = 10^{12} = 1.000.000.000.000 bayt.
2. **1024 tabanına çevirme.** 1 GiB = 2^{30} = 1.073.741.824 bayt. 10^{12} ÷ 1.073.741.824 ≈ **931,3 GiB**.

İşletim sistemi bu değeri "931 GB" diye gösterirse disk yaklaşık %7 "küçük" görünür. Bayt sayısı aynıdır; değişen yalnızca birimdir. Aynı hesapla 500 GB'lık bir disk ≈ 465,7 GiB, 256 GB'lık bir SSD ≈ 238,4 GiB görünür.

1.2.9.6 Taşınabilir ve arşiv depolama

Bilgisayarın içindeki diskin yanında, veriyi taşımak, yedeklemek ya da paylaşmak için başka depolama seçenekleri de vardır.

Seçenek	Kullanım ve özellik
USB bellek	Kolay taşınır; kaybolma ve fiziksel hasar riski vardır.
SD kart	Kamera ve mobil aygıtlarda yaygındır.
Optik disk	Yazılım dağıtımında ve belirli arşiv senaryolarında kullanılır.
Harici disk	Yedekleme ve veri taşıma için yüksek kapasite sunar.
Ağ depolama	Yerel ağdaki birden çok kullanıcıya merkezî dosya erişimi sağlar.
Bulut depolama	İnternet üzerinden eşitleme ve paylaşım sağlar.

Tablo 1.2.17: Taşınabilir ve arşiv depolama seçenekleri.



Şekil 1.2.33: Bilgisayar USB bellek, SD kart, harici disk ve optik diske doğrudan (USB ya da kart okuyucuyla) bağlanır; ağ depolamaya yerel ağ, bulut depolamaya internet üzerinden ulaşır.

Bu seçeneklerin en önemli kullanım alanı **yedeklemedir** (*backup*). Bir dosyanın tek kopyası bulunuyorsa, diskin bozulması, bilgisayarın çalınması ya da yanlışlıkla silme o dosyayı kalıcı olarak yok eder. Yaygın bir öneri, önemli verinin en az üç kopyasını, en az iki farklı ortamda tutmak ve bu kopyalardan birini başka bir yerde (örneğin bulutta) saklamaktır.

1.2.10 Sistem tasarımı

Tek tek iyi parçalar seçmek, iyi bir bilgisayar elde etmeye yetmez. Parçaların birbirine uyması, birbirini sınırlamaması ve kullanım amacına hizmet etmesi gerekir. Bu kesim, bir sistemi bütün olarak değerlendirmenin ölçütlerini toplar.

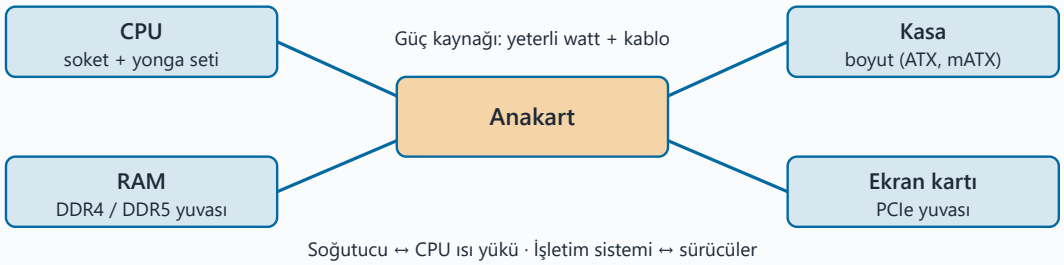
1.2.10.1 Bileşen uyumluluğu

Tanım 1.2.24

Uyumluluk (*compatibility*), bileşenlerin fiziksel, elektriksel, mantıksal ve yazılım açısından birlikte sorunsuz çalışabilmesidir.

Tür	Kontrol edilen
Fiziksel	Kasa, anakart ve kart boyutları birbirine uymalıdır.
Elektriksel	Güç kaynağının kapasitesi ve bağlantıları yeterli olmalıdır.
Mantıksal	İşlemci soketi, bellek nesli ve yonga seti desteklenmelidir.
Bağlantı	Port, kablo ve protokol sürümleri uyumlu olmalıdır.
Isıl	Soğutma sistemi işlemci ve GPU'nun ısı yükünü karşılamalıdır.
Yazılım	İşletim sistemi ve sürücü desteği bulunmalıdır.

Tablo 1.2.18: Uyumluluk türleri.



Şekil 1.2.34: Uyumluluk kontrolleri anakartın çevresinde toplanır: CPU ile soket ve yonga seti, RAM ile DDR4/DDR5 yuvası, kasa ile anakart boyutu (ATX, mATX), ekran kartı ile PCIe yuvası. Ayrıca güç kaynağı yeterli watt ve kablo sağlamalı, soğutucu işlemcinin ısı yükünü karşılamalı, işletim sisteminin sürücüleri bulunmalıdır.

Uyumluluk kontrolünün merkezinde anakart vardır (Şekil 1.2.34). Bir bilgisayar parça parça toplanırken kontroller kabaca belli bir sırayla yapılır. Önce işlemci ve ona uyan soketli anakart seçilir. Ardından anakartın desteklediği RAM nesli belirlenir. Sonra anakartın ve ekran kartının kasaya sığıp sığmadığına bakılır. En son bütün sistemin güç ve soğutma ihtiyacı hesaplanır.

Örnek 1.2.23

Bir öğrenci ATX boyutunda, DDR5 destekli bir anakart seçmiş. Anakartın M.2 yuvası yalnızca SATA destekliyor. Öğrencinin seçtiği öteki parçalar iki DDR4 RAM modülü, bir NVMe SSD ve mATX boyutunda bir kasadır. Hangi sorunlar vardır?

1. **RAM.** DDR4 ve DDR5 modülleri fiziksel olarak farklıdır; DDR4 modül DDR5 yuvasına takılmaz. RAM, DDR5 olarak değiştirilmelidir.
2. **SSD.** M.2 yuvası yalnızca SATA destekliyorsa NVMe SSD bu yuvada çalışmaz. Ya SATA M.2 SSD seçilmeli ya da NVMe destekli bir yuva kullanılmalıdır.
3. **Kasa.** mATX kasa, daha büyük olan ATX anakartı almaz. Kasa ATX olmalıdır; tersi (ATX kasaya mATX anakart) ise sorunsuzdur.

Üç sorunun hiçbiri tek tek parçaların kalitesiyle ilgili değildir; hepsi uyumluluk sorunudur.

1.2.10.2 DDR5 ve güncel standartlar

Bilgisayar donanımı birkaç yılda bir yeni standartlarla güncellenir. Bir parçanın hangi standardı desteklediği, hem hızını hem de diğer parçalarla uyumunu belirler.

Standart	Ne değişti?	Dikkat edilecek nokta
DDR4 → DDR5	Bant genişliği ve hız aralığı artar; DDR5 modülleri daha yüksek hızlarda başlar. Güç yönetimi modülün üzerindedir; her modülde iki bağımsız alt kanal vardır.	Fiziksel olarak farklıdır: DDR4 modülü DDR5 yuvasına (ve tersi) takılmaz.

Standart	Ne değişti?	Dikkat edilecek nokta
PCIe 3 → 4 → 5	Şerit başına bant genişliği her nesilde yaklaşık iki katına çıkar.	Yuva geriye uyumludur; hız, yuva ile aygıttan yavaş olanına göre belirlenir.
USB 3.2 → USB4	USB-C konektörü, 20/40 Gbit/s ve Thunderbolt uyumu; USB4 Sürüm 2.0 daha da yüksek hız tanımlar.	Her USB-C portu bu hızları desteklemez.
Wi-Fi 6 → 7	6 GHz bandı (6E ile gelmişti), 320 MHz'e kadar kanal genişliği, çoklu bağlantı (MLO).	Yönlendirici ve aygıtın ikisi de desteklemelidir.

Tablo 1.2.19: Güncel standartlardaki değişimler.

Tablodaki terimlerin bazıları açıklanmalıdır. **Bant genişliği** (*bandwidth*), bir bağlantının birim zamanda taşıyabildiği veri miktarıdır. **Geriye uyumluluk** (*backward compatible*), yeni bir standardın eski aygıtlarla da çalışabilmesidir. Wi-Fi'deki **kanal genişliği**, kablosuz yayının kullandığı frekans aralığıdır; geniş kanal, daha çok verinin aynı anda taşınması demektir. MLO (*Multi-Link Operation*) ise bir aygıtın birden çok frekans bandını aynı anda kullanmasıdır.

Örnek 1.2.24

PCIe 4.0 x4 bir NVMe SSD, PCIe 3.0 x4 bir M.2 yuvasına takılıyor. SSD çalışır mı? Kuramsal en yüksek hız ne olur?

1. **Çalışma.** PCIe geriye uyumludur; SSD çalışır.
2. **Hız.** Hız, yuva ile aygıttan yavaş olanına göre belirlenir: PCIe 3.0. Örnek 1.2.19'daki hesapla 3.0 x4 sınırı ≈ 3,94 GB/s'dir. SSD'nin kendi sınırı 7 GB/s olsa bile bu yuvada en çok yaklaşık 3,9 GB/s'ye ulaşır.

Sık yapılan hata

DDR nesillerinin de PCIe gibi geriye uyumlu olduğunu sanmak. PCIe kartları farklı nesillerin yuvalarında çalışır, ama DDR4 ve DDR5 modüllerinin çentikleri farklı yerdedir ve birbirinin yuvasına fiziksel olarak takılamaz.

1.2.10.3 Kullanım amacına göre seçim

Her kullanıcı için “en iyi” bilgisayar yoktur; kullanım amacına en uygun bilgisayar vardır. Bütçe sınırlı olduğunda para, işin en çok ihtiyaç duyduğu bileşenlere ayrılır.

Kullanım	Öncelikli bileşenler
Ofis ve eğitim	dengeli CPU, yeterli RAM, SSD, ergonomik ekran
Oyun	GPU, CPU, soğutma, yüksek yenileme hızlı ekran
İçerik üretimi	çok çekirdekli CPU, GPU, RAM, renk doğruluğu, hızlı depolama

Kullanım	Öncelikli bileşenler
Veri analizi	RAM kapasitesi, CPU/GPU, hızlı depolama
Taşınabilir kullanım	batarya, ağırlık, ekran, bağlantılar, enerji verimliliği

Tablo 1.2.20: Kullanım amacına göre öncelikli bileşenler.

Tablonun mantığı, önceki kesimlerde anlatılanlara dayanır. Ofis işleri tek iş parçacıklı ve hafiftir; çok çekirdek ya da güçlü GPU fazla fark yaratmaz, ama SSD programların hızlı açılmasını sağlar. Oyunlarda her kare GPU'da çizildiği için ekran kartı belirleyicidir; yüksek yenileme hızlı bir ekran da ancak GPU yeterince kare üretebiliyorsa işe yarar. Video kurgusu gibi içerik üretimi işleri bölünebilir; çok çekirdekten ve hızlı depolamadan doğrudan yararlanır. Büyük veri kümelerini işleyen programlar ise veriyi RAM'de tutmak ister; RAM yetmezse sistem diski kullanır ve belirgin biçimde yavaşlar.

Örnek 1.2.25

Aynı bütçeyle iki seçenek var. (A) Güçlü bir bağımsız GPU, 8 GB RAM ve 1 TB HDD. (B) Entegre GPU, 16 GB RAM ve 512 GB NVMe SSD. Ders notu yazan, internette araştırma yapan ve birkaç programı aynı anda açık tutan bir öğrenci için hangisi uygundur?

1. **İş yükü.** Metin düzenleme, tarayıcı ve ofis programları GPU'ya pek yük bindirmez; entegre GPU yeterlidir.
2. **RAM.** Çok sayıda tarayıcı sekmesi ve programın aynı anda açık olması RAM ister. 8 GB'ta sistem takas alanını kullanmaya başlayabilir; 16 GB daha rahattır.
3. **Depolama.** İşletim sistemi ve programların açılışı depolama hızına bağlıdır. NVMe SSD, HDD'den çok daha hızlıdır (1.2.9.5).

Bu kullanım için **B** daha uygundur. A seçeneği oyun için güçlüdür, ama bu öğrencinin günlük işlerinde yavaş disk ve az RAM nedeniyle daha yavaş hissettirir.

1.2.11 Özet

- **Donanım**, bilgisayarın fiziksel parçalarıdır; **yazılım** bu parçalara ne yapacaklarını söyler. Her iş giriş, işlem, bellek, depolama ve çıkış adımlarından oluşur.
- **CPU** komutları **getir–çöz–yürüt** döngüsüyle işler. ALU hesap yapar, kontrol birimi akışı yönetir, yazmaçlar o anki değerleri tutar. Bir saat turunun süresi 1 ÷ saat hızıdır (3 GHz'de ≈ 0,33 ns). Performansı saat hızı, çekirdek sayısı, önbellek ve mimari birlikte belirler; “8 çekirdek = 8 kat hız” değildir.
- **x86-64** masaüstü ve sunucularda, **ARM** mobil aygıtlarda yaygındır; **RISC-V** açık bir komut kümesidir. **SoC**, CPU, GPU, NPU ve denetleyicileri tek yongada toplar.

- **Anakart** bileşenleri bağlar ve uyumluluğu belirler (soket, DIMM, PCIe, M.2, SATA, yonga seti). **RAM** geçici ve hızlıdır; her baytın bir adresi vardır. **ROM/flash**'taki **UEFI**, açılışta donanımı başlatır ve işletim sistemini yükler.
- **Bellek hiyerarşisinde** yukarı çıktıkça hız artar, kapasite azalır: yazmaç < 1 ns, önbellek 1–10 ns, RAM 50–100 ns, SSD ≈ 0,1 ms, HDD 5–10 ms. SSD, RAM'den 1000–2000 kat; HDD, SSD'den 50–100 kat yavaştır.
- **Önbellekte** veri bulunursa **isabet** (≈ 1 ns), bulunamazsa **ıska** (≈ 80 ns) olur; ıskada bütün önbellek satırı getirilir. Ardışık erişim **uzamsal**, aynı veriye tekrar erişim **zamansal yerellikten** yararlanır. 8 elemanlı dizi toplamı önbellekle 166 ns, önbelleksiz 640 ns sürer.
- **GPU** binlerce basit çekirdekle paralel çalışır; entegre GPU RAM'i paylaşır, bağımsız GPU'nun kendi VRAM'i vardır. HDMI ve DisplayPort dijital, VGA analog görüntü taşır. USB'de konektör biçimi ile sürüm (hız) ayrıdır; bit cinsinden hız 8'e bölünerek bayta çevrilir.
- **Ses kartında** ADC analog sesi örnekleyip sayıya, DAC sayıyı sinyale çevirir. Kayıt boyutu = örnekleme hızı × örnek boyutu × kanal × süredir.
- **Kasa** hava akışını sağlar; aşırı ısınan işlemci **ısıl kısma** ile yavaşlar. **Güç kaynağı** AC'yi DC'ye çevirir; kapasitesi toplam ihtiyacın üstünde güvenli payla seçilir (420 W × 1,3 = 546 W → 550 W).
- Bir aygıtın **giriş** ya da **çıkış** olduğunu veri akışının yönü belirler; dokunmatik ekran, depolama ve ağ kartı iki yönlüdür.
- **PPI** = $\sqrt{(\text{yatay}^2 + \text{dikey}^2)} \div$ köşegen; keskinliği çözünürlük ile boyut birlikte belirler. Kare süresi = 1000 ÷ Hz ms'dir; yenileme hızı ile tepki süresi farklı şeyleri ölçer. LCD arka ışık kullanır, OLED'de her piksel kendi ışığını üretir.
- CPU programı SSD'den değil RAM'den çalıştırır; kaydedilmemiş değişiklikler RAM'dedir. **HDD** erişimi = arama + dönme gecikmesi (yarım tur) + aktarımdır ve milisaniyeler sürer. **NVMe**, PCIe üzerinden SATA'nın ≈ 550 MB/s sınırını aşar; M.2 bir biçimdir, protokol değildir. 1 TB = 10^{12} bayt ≈ 931,3 GiB'dir.
- Sistem tasarımında fiziksel, elektriksel, mantıksal, bağlantı, ısıl ve yazılım **uyumluluğu** kontrol edilir. PCIe geriye uyumludur, DDR4 ve DDR5 değildir. Bileşenler kullanım amacına göre önceliklendirilir.

Hesap	Formül	Örnek
Saat turu	1 ÷ saat hızı	3,2 GHz → 0,3125 ns
Önbellekli süre	ıska × 80 + isabet × 1 (ns)	2 ıska, 6 isabet → 166 ns
Bit → bayt	Mbit/s ÷ 8 = MB/s	480 Mbit/s → 60 MB/s
Ses kaydı	örnek/s × bayt × kanal × s	48.000 × 2 × 1 × 30 = 2.880.000 bayt
Güç kaynağı	toplam × 1,3, üstteki model	420 W → 546 W → 550 W
Piksel yoğunluğu	$\sqrt{(y^2 + d^2)} \div$ köşegen	1920 × 1080, 24 inç → ≈ 91,8 PPI
Kare süresi	1000 ÷ Hz (ms)	144 Hz → ≈ 6,9 ms
HDD erişimi	arama + yarım tur + aktarım	5 + 4,17 + 0,03 ≈ 9,2 ms
Kopyalama	boyut ÷ hız	10.000 MB ÷ 550 MB/s ≈ 18,2 s

Hesap	Formül	Örnek
GB → GiB	bayt ÷ 2^{30}	1 TB → ≈ 931,3 GiB

Tablo 1.2.21: Notta kullanılan başlıca hesaplar.

1.2.12 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

donanım *hardware* · merkezî işlem birimi (CPU) *central processing unit* · çekirdek *core* · saat hızı *clock speed* · saat turu *clock cycle* · iş parçacığı *thread* · komut kümesi *instruction set* · SoC (yonga üstü sistem) *System-on-Chip* · sinirsel işlem birimi (NPU) *neural processing unit* · anakart *motherboard* · yonga seti *chipset* · yuva *slot* · şerit *lane (PCIe)* · RAM (rastgele erişimli bellek) *random access memory* · geçici / kalıcı *volatile / non-volatile* · takas alanı *swap space* · ROM *read-only memory* · ürün yazılımı *firmware* · BIOS / UEFI *BIOS / UEFI* · açılışta öz sınama (POST) *power-on self-test* · bellek hiyerarşisi *memory hierarchy* · önbellek *cache* · isabet / ıska *cache hit / miss* · önbellek satırı (blok) *cache line* · uzamsal yerellik *spatial locality* · zamansal yerellik *temporal locality* · grafik işlem birimi (GPU) *graphics processing unit* · video belleği (VRAM) *video memory* · ADC / DAC *ADC / DAC* · ısıl kısma *thermal throttling* · güç kaynağı (PSU) *power supply unit* · giriş birimi / çıkış birimi *input / output device* · işaretleme aygıtı *pointing device* · yoklama hızı *polling rate* · çözünürlük *resolution* · piksel yoğunluğu (PPI) *pixel density* · en-boy oranı *aspect ratio* · yenileme hızı *refresh rate* · tepki süresi *response time* · arka ışık *backlight* · sabit disk (HDD) *hard disk drive* · SSD *solid-state drive* · plaka *platter* · arama süresi *seek time* · dönme gecikmesi *rotational latency* · sıralı okuma *sequential read* · yedekleme *backup* · uyumluluk *compatibility* · bant genişliği *bandwidth* · geriye uyumlu *backward compatible*

1.2.13 Kendini Dene

1. 2,5 GHz bir işlemcide bir saat turu kaç ns sürer? Her komut ortalama 5 tur sürüyorsa işlemci sani-

yede kaç komut yürütür?

Cevap

Bir tur $1 \div (2{,}5 \times 10^9)$ s = $0{,}4 \times 10^{-9}$ s = 0,4 ns'dir. Saniyede $2{,}5 \times 10^9$ tur atılır ve her komut 5 tur sürer: $2{,}5 \times 10^9 \div 5$ = 500 milyon komut/s.

2. Her biri tek çekirdekte 3 saniye süren 10 bağımsız ve bölünemez iş, 4 çekirdekli bir işlemcide çalıştırılıyor. Toplam süre ve tek çekirdeğe göre hızlanma nedir?

Cevap

Tur sayısı $10 \div 4 = 2{,}5$'in yukarı yuvarlanmasıdır: 3 tur. İlk iki turda 4'er, son turda 2 iş çalışır. Toplam süre $3 \times 3 = 9$ s'dir. Tek çekirdekte $10 \times 3 = 30$ s sürerdi; hızlanma $30 \div 9 \approx 3{,}3$ kattır, 4 kat değildir.

3. 12 elemanlı bir tam sayı dizisi (eleman 4 bayt) sırayla toplanıyor. Önbellek satırı 32 bayt, önbellek başta boş; isabet 1 ns, ıska 80 ns. Kaç ıska olur ve toplam süre nedir?

Cevap

Bir satıra $32 \div 4 = 8$ eleman sığar. a[0]…a[7] ilk blokta, a[8]…a[11] ikinci bloktadır. a[0] ve a[8]'de ıska, kalan 10 erişimde isabet olur: $2 \times 80 + 10 \times 1 = 170$ ns. Önbelleksiz $12 \times 80 = 960$ ns sürerdi.

4. L1 ≈ 1 ns, L2 ≈ 3 ns, L3 ≈ 10 ns, RAM ≈ 80 ns. Başlangıçta L1'de A, L2'de B, L3'te B ve C var. İşlemci sırayla C, B, C okuyor; bulunan veri üst düzeylere kopyalanıyor. Her okuma ne kadar sürer?

Cevap

C: L1'de yok (1), L2'de yok (+3), L3'te var (+10) → 14 ns; C, L2 ve L1'e kopyalanır. B: L1'de yok (1), L2'de var (+3) → 4 ns; B, L1'e kopyalanır. C: artık L1'de → 1 ns. Toplam 19 ns; hepsi RAM'den okunsaydı 240 ns olurdu.

5. Stereo, 48.000 örnek/s ve örnek başına 24 bit (3 bayt) ile 1 dakikalık sıkıştırılmamış kayıt yapılıyor. Kayıt kaç bayt ve yaklaşık kaç MiB tutar?

Cevap

Saniyede $48.000 \times 3 \times 2 = 288.000$ bayt; 60 saniyede $288.000 \times 60 = 17.280.000$ bayt. $17.280.000 \div 1.048.576 \approx 16{,}5$ MiB.

6. 15,6 inçlik, 2560 × 1440 çözünürlüklü bir dizüstü ekranının PPI değeri nedir? Bu ekran 165 Hz ise

bir kare kaç ms ekranda kalır?

Cevap

$\sqrt{(2560^2 + 1440^2)} = \sqrt{8.627.200} \approx 2937{,}2$; $2937{,}2 \div 15{,}6 \approx 188{,}3$ PPI. Kare süresi $1000 \div 165 \approx 6{,}1$ ms'dir.

7. 5400 dev/dk dönen bir HDD'nin ortalama arama süresi 12 ms, aktarım hızı 120 MB/s'dir. 4 KB'lık bir sektörün ortalama erişim süresi nedir? (1 MB = 1000 KB)

Cevap

Bir tur $60.000 \div 5400 \approx 11{,}11$ ms; dönme gecikmesi yarım tur ≈ 5,56 ms. Aktarım $0{,}004 \text{ MB} \div 120 \text{ MB/s} \approx 0{,}033$ ms. Toplam $12 + 5{,}56 + 0{,}03 \approx 17{,}6$ ms.

8. Kutusunda “2 TB” yazan bir disk, 1024 tabanıyla hesap yapan bir işletim sisteminde yaklaşık kaç GiB görünür?

Cevap

$2 \text{ TB} = 2 \times 10^{12}$ bayt. $2 \times 10^{12} \div 2^{30} \approx 1862{,}6$ GiB. Bayt sayısı aynıdır; fark yalnızca birimden gelir.

9. Bir sistemde işlemci 65 W, ekran kartı 160 W, diğer bileşenler 50 W çekiyor. %30 pay bırakılacak; 300, 400, 450 ve 550 W'lık modeller var. Hangisi seçilir?

Cevap

Toplam $65 + 160 + 50 = 275$ W; payla $275 \times 1{,}3 = 357{,}5$ W. Bunun altında kalmayan en küçük model 400 W'tır.

10. Bir program dosyası SSD'de duruyor. Program açılıp bir hesap yapıldığında veri hangi sırayla hangi birimlerden geçer? Sonuç kaydedilince ne olur?

Cevap

Program SSD'den RAM'e kopyalanır; CPU komutları ve veriyi RAM'den önbelleğe, oradan yazmaçlara getirip işler: SSD → RAM → önbellek → yazmaçlar. Sonuç ekran kartı üzerinden monitöre gider. Kaydedildiğinde veri RAM'den SSD'ye yazılır; kaydedilene kadar değişiklikler yalnızca RAM'dedir.

1.2.14 İleri okuma

- C. Petzold, *Code: The Hidden Language of Computer Hardware and Software* (2. baskı, 2022): ikili sayılardan ve mantık kapılarından başlayarak bir işlemcinin nasıl kurulduğunu, ön bilgi gerektirmeden ve adım adım anlatır.
- D. A. Patterson ve J. L. Hennessy, *Computer Organization and Design RISC-V Edition* (2. baskı, 2020), Bölüm 1 ve 5: performansın saat hızı ve komut sayısıyla nasıl hesaplandığını, bellek hiyerarşisini ve önbellek tasarımını ayrıntılı işler.
- R. E. Bryant ve D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective* (3. baskı, 2015), Bölüm 6: depolama teknolojilerini, HDD erişim süresinin hesabını ve yerelliğin program hızına etkisini programcı gözüyle anlatır.

1.2.15 Kaynaklar

1. D. A. Patterson, J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware/Software Interface*, 2. baskı, Morgan Kaufmann, 2020.
2. W. Stallings, *Computer Organization and Architecture*, 11. baskı, Pearson, 2019.
3. R. E. Bryant, D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3. baskı, Pearson, 2015.
4. C. Petzold, *Code: The Hidden Language of Computer Hardware and Software*, 2. baskı, Microsoft Press, 2022.
5. PCI-SIG (PCI Express), USB-IF (USB), VESA (DisplayPort) ve JEDEC (DDR bellek) teknik standartları.

KISIM I · BÖLÜM 1.3

Bilgisayarlar

Bilgisayar yalnız iki durumu ayırt eden milyarlarca küçük anahtardan oluşur. Buna karşın aynı makine hesap yapar, metin yazar, resim gösterir, müzik çalar. Bu not, aradaki boşluğun nasıl kapandığını anlatır. Önce sayıların bit dizileriyle nasıl yazıldığı ele alınır: tabanlar, negatif sayılar, taşma ve kayan noktalı sayılar. Ardından harflerin ASCII, Unicode ve UTF-8 ile nasıl sayıya çevrildiği gösterilir. Not, mantık kapılarından kurulan toplayıcılarla devam eder ve işlemcinin bellekteki bir programı getir–çöz–yürüt döngüsüyle adım adım nasıl çalıştırdığını izler. Son kesimler bilgisayar mimarisinin temel tasarım seçimlerini ve bir Java satırının makine komutlarına kadar inen yolunu konu alır.

ÖĞRENME ÇIKTILARI

Sayı temsili
Sayıları ikili, sekizli, onluk ve onaltılık tabanlar arasında çevirmek; sınırlı bit sayısının sonuçlarını (negatif sayılar, taşma, kayan nokta) açıklamak.

Metin temsili
ASCII, Unicode ve dizgi kavramlarını ayırt etmek; bir karakterin UTF-8 baytlarını elle bulmak.

Program yürütme
Depolanan program kavramını ve getir–çöz–yürüt döngüsünü açıklamak; küçük bir makine programını adım adım izlemek.

Mimari
CPU, bellek, yazmaç ve veri yolu ilişkisini yorumlamak; von Neumann ve Harvard mimarilerini, RISC ve CISC yaklaşımlarını karşılaştırmak.

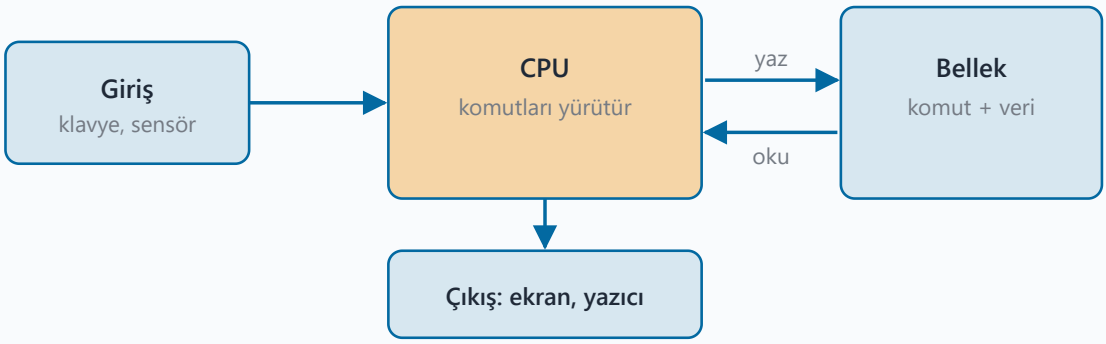
1.3.1 Bilgisayar ve bilgi

Bir hesap makinesi yalnız hesap yapar; bir müzik çalar yalnız müzik çalar. Bilgisayar ise aynı donanımla bunların hepsini ve çok daha fazlasını yapabilir. Bu esnekliğin iki kaynağı vardır. Birincisi, her türlü bilginin sayılara, sayıların da bitlere çevrilebilmesidir. İkincisi, makinenin ne yapacağını söyleyen komutların da birer sayı olarak bellekte durmasıdır. Bu kesim, bilgisayarın ana parçalarını ve bilginin bitlere dönüşme fikrini tanıtır.

1.3.1.1 Bilgisayarın temel parçaları

Bir masaüstü bilgisayar, bir telefon ya da bir çamaşır makinesinin denetim kartı ilk bakışta çok farklı görünür. Yine de üçünde de aynı üç temel parça bulunur.

- **Merkezî işlem birimi** (CPU, *central processing unit*), kısaca işlemci, komutları yürütür. Toplama, karşılaştırma, bir değeri bir yerden başka bir yere taşıma gibi işleri o yapar.
- **Bellek** (*memory*), hem işlenecek verileri hem de işlemcinin yürüteceği komutları saklar. Bu notta "bellek" sözcüğü, aksi söylenmedikçe ana belleği (RAM) anlatır.
- **Giriş ve çıkış birimleri** (G/Ç), bilgisayarın dış dünyayla veri alışverişini sağlar. Klavye, fare, mikrofon ve sıcaklık sensörü giriş birimidir; ekran, yazıcı ve hoparlör çıkış birimidir.



Şekil 1.3.1: Bilgisayarın temel parçaları. Giriş birimleri CPU'ya veri verir. CPU belleğe değer yazar ve bellekten değer okur. Sonuç çıkış birimine gider.

Şekil 1.3.1, parçalar arasındaki veri akışını gösterir. Klavyeden basılan bir tuş, giriş birimi üzerinden CPU'ya ulaşır. CPU bu değeri belleğe yazabilir, daha sonra bellekten geri okuyabilir, üzerinde işlem yapabilir. Ortaya çıkan sonuç ekrana gönderilir. CPU ile bellek arasındaki iki ok, bu ikisinin sürekli konuştuğunu anlatır: işlemci çalıştığı sürece bellekten komut ve veri okur, belleğe sonuç yazar. CPU'nun kendi içindeki bölümler ve bellekle bağlantısı 1.3.7'de ayrıntılı işlenir.

> **Sık yapılan hata**
>
> Belleği, dosyaların kalıcı olarak saklandığı disk ile karıştırmak. Ana bellek (RAM) hızlıdır, ama elektrik kesilince içeriği silinir. Disk ya da SSD daha yavaştır, ama içeriğini korur. Bir program çalışmadan önce diskten belleğe yüklenir; işlemci yalnız bellekteki komutları yürütebilir.

1.3.1.2 Bilgiden bitlere

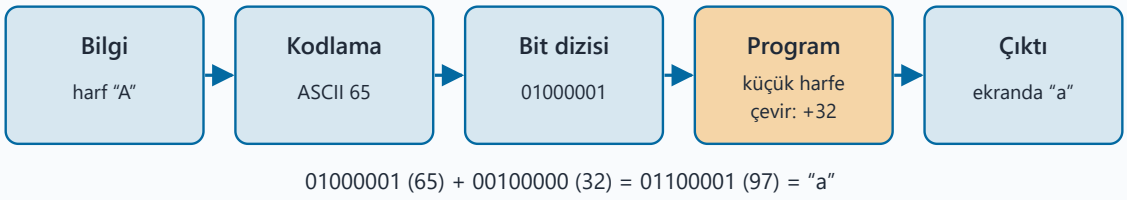
Bilgisayarın içindeki her devre, temelde iki durumu ayırt eder: gerilim var ya da yok, anahtar açık ya da kapalı. Bu iki durum 1 ve 0 rakamlarıyla gösterilir. Tek bir 0 ya da 1'e **bit** (*binary digit*, ikili basamak) denir. Sekiz bitlik gruba **bayt** (*byte*) denir.

Tek bit yalnız iki durum anlatabilir. Bitler yan yana gelince olasılıklar katlanarak artar. İki bitle 00, 01, 10, 11 olmak üzere 4 farklı dizi yazılır; üç bitle 8, sekiz bitle $2^8 = 256$ farklı dizi. Genel kural şudur: n bitle 2^n farklı bit dizisi yazılabilir. Sayılar, harfler, resimdeki noktaların renkleri ve programın komutları bu dizilerden biriyle eşleştirilerek saklanır.

Tanım 1.3.1

Kodlama (*encoding*), bir bilgi türünün öğelerini bit dizileriyle eşleştiren kurallar bütünüdür. **Veri tipi** (*data type*) ise bellekteki bir bit dizisinin hangi kodlamayla okunacağını belirler.

Bu tanımın önemli bir sonucu vardır: bit dizisi tek başına bir anlam taşımaz. `01000001` dizisi tam sayı olarak okunursa 65'tir; ASCII karakteri olarak okunursa "A" harfidir. Anlamı, diziyi okuyan programın seçtiği veri tipi verir.



Şekil 1.3.2: "A" harfinin yolculuğu. Harf ASCII ile 65 sayısına, 65 de `01000001` bit dizisine çevrilir. Program bu sayıya 32 ekler ve sonuç `01100001` = 97 olur; ekranda "a" görünür.

Şekil 1.3.2 bütün zinciri bir arada gösterir. Klavyede "A" tuşuna basılır. Kodlama kuralı bu harfi 65 sayısıyla eşleştirir. Bellekte 65, sekiz bitlik `01000001` dizisi olarak durur. Bir program bu sayıya 32 eklerse `01100001`, yani 97 elde edilir. Ekran bu sayıyı yine aynı kodlamayla okur ve "a" harfini çizer. Bilgisayar harfi "küçültmeyi" bilmez; yalnız bir sayıya 32 ekler. Anlamı kodlama ve veri tipi kurar. Bu küçük örnek 1.3.4.2'de bit işlemleriyle yeniden ele alınır.

1.3.2 Tam sayıların temsili

Bilgisayarın en temel işi sayılarla uğraşmaktır. Bu kesim, tam sayıların bitlerle nasıl yazıldığını anlatır. Önce gündelik onluk sistemin nasıl çalıştığı hatırlanır; aynı fikir ikili sisteme taşınır. Ardından negatif sayılar ve sınırlı bit sayısının doğurduğu taşma sorunu ele alınır.

1.3.2.1 Konumsal sayı sistemleri ve taban

Onluk sistemde 345 yazıldığında üç rakamın her biri farklı bir değer taşır. 5 birler basamağında, 4 onlar basamağında, 3 yüzler basamağındadır:

$$345 = 3 \cdot 100 + 4 \cdot 10 + 5 \cdot 1 = 3 \cdot 10^2 + 4 \cdot 10^1 + 5 \cdot 10^0$$

Rakamın değeri, bulunduğu konuma göre değişir. Bu yüzden böyle sistemlere **konumsal sayı sistemi** denir. Her basamağın ağırlığı, bir sağındakinin 10 katıdır. Buradaki 10 sayısına sistemin **tabanı** (*base, radix*) denir.

Tanım 1.3.2

Tabanı b olan konumsal sistemde 0'dan $b - 1$'e kadar b tane rakam kullanılır. Sağdan sola 0, 1, 2, … diye numaralanan basamaklardan i. basamağın ağırlığı b^i'dir. Sayının değeri, her rakamın kendi ağırlığıyla çarpımlarının toplamıdır.

Bilgisayar biliminde dört taban sık kullanılır. Tablo 1.3.1, hepsinde aynı sayıyı, 13'ü gösterir. Hangi tabanda yazıldığı karışmasın diye sayının sağ altına taban küçük rakamlarla eklenir: 1101_2, 15_8, 13_{10}, D_{16}.

Sistem	Taban	Rakamlar	Örnek	Değeri
İkili	2	0–1	1101_2	$1 \cdot 8 + 1 \cdot 4 + 0 \cdot 2 + 1 \cdot 1 = 13$
Sekizli	8	0–7	15_8	$1 \cdot 8 + 5 \cdot 1 = 13$
Onluk	10	0–9	13_{10}	$1 \cdot 10 + 3 \cdot 1 = 13$
Onaltılık	16	0–9, A–F	D_{16}	$13 \cdot 1 = 13$

Tablo 1.3.1: Sık kullanılan sayı sistemleri. Dört örnek de aynı sayıyı (on üçü) gösterir.

Onaltılık sistemde on altı rakam gerekir, ama gündelik yazıda yalnız on rakam vardır. Eksik altı rakam için A–F harfleri kullanılır: A = 10, B = 11, C = 12, D = 13, E = 14, F = 15. Programlama dillerinde taban çoğu zaman bir önekle belirtilir. Java'da `0b1101` ikili, `0xD` onaltılık bir sabittir.

1.3.2.2 Onluktan ikiliye, ikiliden onluğa

İkiliden onluğa çevirmek, tanımı doğrudan uygulamaktır. İkili sistemde basamak ağırlıkları sağdan sola 1, 2, 4, 8, 16, 32, 64, 128, … diye ikişer katına çıkar. Hangi basamakta 1 varsa o ağırlık toplama eklenir. Örneğin 101101_2'de 1'ler 32, 8, 4 ve 1 basamaklarındadır: 32 + 8 + 4 + 1 = 45.

Onluktan ikiliye çevirmenin en düzenli yolu, sayıyı art arda 2'ye bölmektir. Her bölmenin kalanı (0 ya da 1) bir bittir. Bölüm 0 olunca durulur. Kalanlar sondan başa doğru okunur, çünkü ilk kalan en sağdaki, yani birler basamağıdır.

Örnek 1.3.1

45 sayısını ikili tabana çevirelim.

Bölünen	÷ 2 bölüm	Kalan (bit)	Basamak ağırlığı
45	22	1	1
22	11	0	2
11	5	1	4
5	2	1	8
2	1	0	16
1	0	1	32

Kalanlar alttan üste okunur: $45_{10} = 101101_2$. Denetim: 32 + 8 + 4 + 1 = 45.

Küçük sayılarda ikinci bir yol daha pratiktir: sayıdan sığan en büyük ikinin kuvvetini çıkarmak. 45'e sığan en büyük kuvvet 32'dir; kalan 13'tür. 13'e 8 sığar, kalan 5; 5'e 4 sığar, kalan 1; 1'e 1 sığar. Kullanılan kuvvetler 32, 8, 4, 1'dir ve bunların basamaklarına 1, ötekilere 0 yazılır: 101101.

Java'nın hazır metotları aynı dönüşümleri yapar. Aşağıdaki kod 45'i üç tabanda yazar, sonra ikili bir yazıyı yeniden sayıya çevirir. Son satır, `0b` ve `0x` önekleriyle yazılmış iki sabitin toplamını yazdırır; ikisi de 45 olduğu için sonuç 90'dır.

```
int n = 45;
System.out.println(Integer.toBinaryString(n));
System.out.println(Integer.toOctalString(n));
System.out.println(Integer.toHexString(n));
System.out.println(Integer.parseInt("101101", 2));
System.out.println(0b101101 + 0x2D);
```

Çıktı:

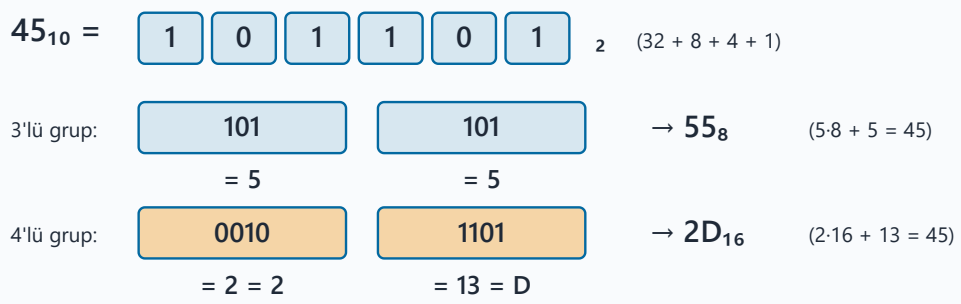
```
101101
55
2d
45
90
```

Java onaltılık rakamları küçük harfle yazdığı için 45 “2d” olarak görünür. $2D_{16}$ ile $2d_{16}$ aynı sayıdır.

1.3.2.3 Sekizli ve onaltılık: bitleri gruplamak

İkili yazım bilgisayar için doğaldır, ama insan için uzun ve okunaksızdır. 32 bitlik bir sayıyı otuz iki rakamla yazmak ve okumak hataya açıktır. Sekizli ve onaltılık tabanlar, uzun ikili sayıları kısa yazmanın yoludur. Bu iki tabanın seçilmesinin nedeni basittir: $8 = 2^3$ ve $16 = 2^4$'tür.

Bu eşitlikler sayesinde dönüşüm için hesap yapmaya gerek kalmaz. Bitler sağdan başlayarak gruplanır: sekizli için üçerli, onaltılık için dörderli. Her grup tek bir rakama çevrilir. En soldaki grup eksik kalırsa soluna 0 eklenir; baştaki sıfırlar değeri değiştirmez.



Şekil 1.3.3: $45_{10} = 101101_2$'nin bit gruplamasıyla sekizli ve onaltılığa çevrilmesi. Üçerli gruplar 101 ve 101'dir: 55_8. Dörderli gruplar 0010 ve 1101'dir: $2D_{16}$.

Şekil 1.3.3'te 101101 önce üçerli gruplanır: 101 | 101. Her grup 5'tir ve sonuç 55_8 olur. Denetim: $5 \cdot 8 + 5 = 45$. Dörderli gruplamada sağdaki grup 1101'dir; soldaki yalnız iki bit kalır (10) ve başına iki 0 eklenerek 0010 yapılır. 0010 = 2 ve 1101 = 13 = D olduğundan sonuç $2D_{16}$'dır. Denetim: $2 \cdot 16 + 13 = 45$.

Ters yön de aynı kolaylıktadır. Onaltılık bir sayıyı ikiliye çevirmek için her rakam dört bitle yazılır. Örneğin $A7_{16}$'da A = 1010, 7 = 0111'dir; sonuç 10100111_2'dir. Onluk değeri $10 \cdot 16 + 7 = 167$'dir. Bu kolaylık yüzünden bellek adresleri, renk kodları ve karakter kodları çoğunlukla onaltılık yazılır. Bir bayt her zaman tam iki onaltılık rakama karşılık gelir.

Örnek 1.3.2

156 sayısını ikili, sekizli ve onaltılık tabanda yazalım.

1. **İkili:** 156'ya sığan kuvvetler 128, 16, 8 ve 4'tür (128 + 16 + 8 + 4 = 156). Bu basamaklara 1 yazılır: 10011100_2.
2. **Sekizli:** Sağdan üçerli gruplar 100, 011 ve 10'dur. Son grup 010 yapılır. Gruplar 2, 3, 4 olduğundan sonuç 234_8'dir. Denetim: $2 \cdot 64 + 3 \cdot 8 + 4 = 156$.
3. **Onaltılık:** Dörderli gruplar 1001 ve 1100'dür, yani 9 ve 12 (C). Sonuç $9C_{16}$'dır. Denetim: $9 \cdot 16 + 12 = 156$.

Sık yapılan hata

Tabanı yazılmamış bir sayıyı onluk okumak. “1101” bin yüz bir de olabilir, on üç de. Tabanı belirsiz bırakmamak için alt simge (1101_2) ya da önek (`0b1101` , `0x1101`) kullanılır. Benzer biçimde 10_{16} on değil 16'dır; onaltılıkta on, A ile yazılır.

1.3.2.4 Sınırlı bit sayısı ve işaretsiz sayılar

Kâğıt üzerinde bir sayının basamak sayısı sınırsızdır; gerektiği kadar basamak eklenir. Bilgisayarda durum farklıdır. İşlemcinin içindeki küçük ve hızlı saklama alanlarına **yazmaç** (*register*) denir ve her yazmacın bit sayısı sabittir: 8, 16, 32 ya da 64. Bellekteki her değişken de belli sayıda bit kaplar. Java'da `byte` 8, `short` 16, `int` 32, `long` 64 bittir.

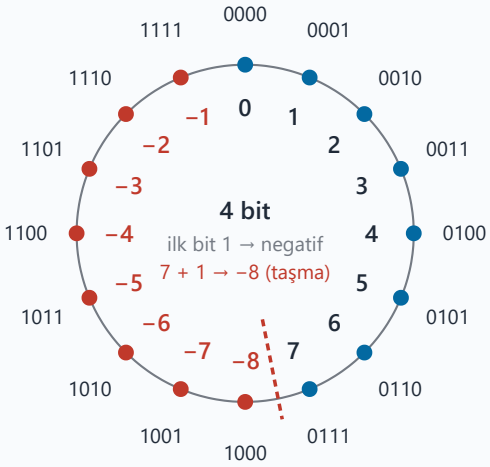
Bitlerin hepsi sayının büyüklüğü için kullanılırsa **işaretsiz** (*unsigned*) sayı elde edilir. n bitle 0'dan $2^n - 1$'e kadar sayılar yazılabilir. 8 bitte en büyük sayı $11111111_2 = 255$'tir. Negatif sayılar ise bu yöntemle yazılamaz. Onları yazmak için bitlerden birine "işaret" görevi vermek gerekir.

1.3.2.5 Negatif sayılar ve ikiye tümleyen

Negatif sayıları yazmanın akla ilk gelen yolu, en soldaki biti işaret için ayırmaktır: 0 artı, 1 eksi. Bu yönteme işaret-büyüklük gösterimi denir ve iki kusuru vardır. Birincisi, sıfırın iki ayrı yazımı oluşur: 00000000 (+0) ve 10000000 (−0). İkincisi, toplama devresi işaretlere bakıp farklı durumlar için farklı işler yapmak zorunda kalır. Modern bilgisayarlar bu yüzden negatif tam sayıları çoğunlukla **ikiye tümleyen** (*two's complement*) gösterimle saklar.

> **Tanım 1.3.3**
>
> n bitlik **ikiye tümleyen** gösterimde en soldaki bitin ağırlığı $+2^{n-1}$ yerine -2^{n-1}'dir; öteki bitlerin ağırlıkları değişmez. Bu yüzden en soldaki bit 1 ise sayı negatiftir ve bu bite **işaret biti** denir. Gösterilebilen aralık -2^{n-1} ile $2^{n-1} - 1$ arasıdır; 8 bitte −128 … 127.

Tanım ilk okumada soyut görünebilir. Bir örnekle somutlaşır. 8 bitte 11111011 dizisinin değeri, en soldaki bitin ağırlığı −128 alınarak bulunur: −128 + 64 + 32 + 16 + 8 + 2 + 1 = −5. Aynı dizi işaretsiz okunsaydı 128 + 64 + 32 + 16 + 8 + 2 + 1 = 251 olurdu. İki okuma arasındaki fark tam 256'dır, yani 2^8. Bu ilişki her negatif sayı için geçerlidir: 8 bitte $-k$'nın bit dizisi, işaretsiz $256 - k$'nın bit dizisiyle aynıdır.



Şekil 1.3.4: 4 bitlik ikiye tümleyen çemberi. 0000'dan 0111'e kadar diziler 0 … 7 pozitif sayılarıdır; 1000'dan 1111'e kadar diziler −8 … −1 negatif sayılarıdır. İlk biti 1 olan her dizi negatiftir. 7'ye 1 eklenince çemberde −8'e geçilir: bu bir taşmadır.

Şekil 1.3.4, 4 bitlik bütün dizileri bir saat kadranı gibi dizer. Saat yönünde ilerlemek 1 eklemek demektir. 0000'dan başlanıp 1 eklendikçe 1, 2, …, 7'ye ulaşılır. 0111'e bir daha 1 eklenince 1000 elde edilir ve bu dizi −8'dir. Kadran böylece bir "uçurum" gösterir: en büyük pozitif sayının hemen ardından en küçük negatif sayı gelir. Saat yönünün tersine gidildiğinde ise 0000'dan 1111'e, yani −1'e geçilir. −1'in bütün bitlerinin 1 olması bu yüzden şaşırtıcı değildir.

İkiye tümleyenin üç önemli üstünlüğü vardır.

- **Tek sıfır.** Sıfırın yalnız bir yazımı vardır (0000); artı sıfır ve eksi sıfır ayrımı oluşmaz.
- **Ortak aritmetik.** Toplama devresi, pozitif ve negatif sayılarda aynı kuralları kullanır. Sayının işaretine bakmaz, bitleri sıradan ikili toplamayla toplar.
- **Kolay olumsuzlama.** Bir sayının negatifi iki basit adımla bulunur (1.3.2.6).

Bit	İşaretli aralık	İşaretsiz aralık	Java tipi
8	−128 … 127	0 … 255	`byte`
16	−32.768 … 32.767	0 … 65.535	`short`
32	−2.147.483.648 … 2.147.483.647	0 … 4.294.967.295	`int`
64	yaklaşık $\pm 9{,}22 \cdot 10^{18}$	0 … yaklaşık $1{,}84 \cdot 10^{19}$	`long`

Tablo 1.3.2: Bit sayısına göre işaretli (ikiye tümleyen) ve işaretsiz aralıklar. Java'daki karşılığı olan tipler son sütundadır.

Aralığın simetrik olmadığına dikkat edilmelidir. 8 bitte −128 yazılabilir, ama +128 yazılamaz. Bunun nedeni, 256 dizinin birinin sıfıra ayrılmasıdır. Geriye kalan 255 dizi, 128 negatif ve 127 pozitif sayı arasında paylaşılır.

1.3.2.6 −5 nasıl yazılır?

Bir sayının ikiye tümleyen gösterimdeki negatifini bulmak için iki adım yeterlidir:

1. Bütün bitler ters çevrilir (0 ↔ 1). Ortaya çıkan diziye **birler tümleyeni** (*one's complement*) denir.
2. Sonuca 1 eklenir.

Örnek 1.3.3

8 bitte −5'i yazalım ve sonucu denetleyelim.

1. **+5'i yaz:** 5 = 4 + 1 olduğundan +5 = 00000101.
2. **Bitleri ters çevir:** 00000101 → 11111010. Bu dizi birler tümleyenidir ve henüz −5 değildir.
3. **1 ekle:** 11111010 + 1 = 11111011. En soldaki 1, sayının negatif olduğunu gösterir.
4. **Denetle:** 5 + (−5) = 0 olmalıdır. 00000101 + 11111011 = 100000000 bulunur. Sonuç 9 bittir; 9. bit 8 bitlik yazmaca sığmaz ve atılır. Kalan 00000000 = 0'dır.



Şekil 1.3.5: +5 ile −5'in 8 bitlik toplamı. Toplam 9 bit çıkar. Taşan 9. bit (elde) atılır, 8 bitte kalan 00000000 = 0'dır.

Şekil 1.3.5'teki denetim, ikiye tümleyenin asıl gücünü gösterir. Toplama devresi −5'in negatif olduğunu bilmez. Bitleri her zamanki gibi toplar, taşan biti atar ve doğru sonucu bulur. Negatif sayılar için ayrı bir devre gerekmez.

Aynı iki adım ters yönde de çalışır. Negatif bir dizinin büyüklüğünü bulmak için yine bitler çevrilir ve 1 eklenir. 11111011 → 00000100 → 00000101 = 5; demek ki dizi −5'tir. Java'da tilde (`~`) bütün bitleri çeviren işlemdir; `~5 + 1` bu yüzden −5'tir:

```
byte x = -5;
System.out.println(Integer.toBinaryString(x & 0xFF));
System.out.println(~5 + 1);
System.out.println(Byte.MIN_VALUE + " " + Byte.MAX_VALUE);
```

Çıktı:

```
11111011
-5
-128 127
```

İlk satırdaki `x & 0xFF`, `byte` değerinin yalnız son 8 bitini almak içindir. Böylece −5'in bellekteki dizisi 11111011 olarak görünür. Son satır, `byte` tipinin sınırlarını Tablo 1.3.2'deki gibi verir.

Sık yapılan hata

İkiye tümleyende yalnız bitleri çevirip 1 eklemeyi unutmak. 11111010 birler tümleyenidir ve ikiye tümleyen gösterimde −6'dır, −5 değil. Bitleri çevirdikten sonra mutlaka 1 eklenir: 11111011 = −5.

1.3.2.7 Çıkarma aslında toplamadır: 5 − 9

İkiye tümleyenin bir başka sonucu, işlemcide ayrı bir çıkarma devresine gerek kalmamasıdır. Çıkarma şu eşitlikle toplamaya çevrilir:

> a − b = a + (−b)

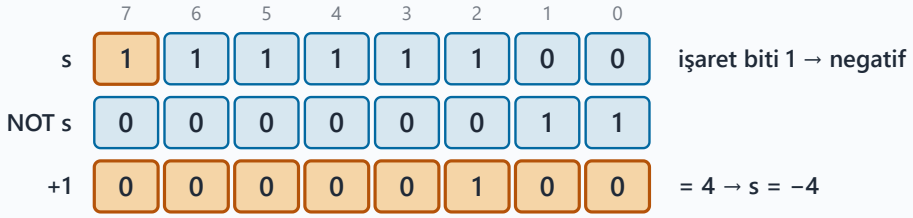
Önce *b*'nin ikiye tümleyeni bulunur, sonra *a* ile toplanır. Sunumdaki adımlar aşağıdaki sözde kodla yazılmıştır. Her satırın sağında, noktalı virgülden sonra açıklaması vardır.

```
a ← 00000101     ; +5
b ← 00001001     ; +9
ters ← NOT b     ; bitleri çevir
eksi ← ters + 1  ; −9
s ← a + eksi     ; 5 + (−9)
yaz(s)           ; işaretli oku
```

Satır	Değişken	Bitler	Açıklama
1	`a`	`00000101`	+5
2	`b`	`00001001`	+9 = 8 + 1
3	`ters`	`11110110`	bütün bitler çevrildi
4	`eksi`	`11110111`	1 eklendi: −9 (işaretsiz okunursa 247 = 256 − 9)
5	`s`	`11111100`	5 + 247 = 252; 256'ya ulaşmadığı için elde çıkmaz
6	yazılan	−4	işaret biti 1, büyüklük: çevir + 1 → 00000100 = 4

Tablo 1.3.3: 5 − 9 işleminin izlemesi. Her satır, sözde kodun bir satırı yürütüldükten sonraki durumu gösterir.

Tablonun son satırı sonucun nasıl okunduğunu gösterir. `s`'nin en soldaki biti 1 olduğu için sayı negatiftir. Büyüklüğünü bulmak için bitler yine çevrilir (00000011) ve 1 eklenir (00000100 = 4). Demek ki *s* = *−4*'tür ve 5 − 9 = −4 doğrudur.



Şekil 1.3.6: 5 − 9 sonucunun okunması. `s` = 11111100'ün işaret biti 1'dir, sayı negatiftir. Bitler çevrilip 1 eklenince 00000100 = 4 bulunur; *s* = *−4*.

Burada dikkat çekici bir nokta vardır: işlemci aynı 11111100 dizisini üretir ve onu nasıl yorumlayacağını bilmez. İşaretsiz yorumda bu dizi 252'dir; işaretli yorumda −4'tür. Hangisinin doğru olduğuna program, değişkenin veri tipine bakarak karar verir.

1.3.2.8 Taşma: 8 bitte 200 + 100

Bit sayısının sınırlı olması, bazı işlemlerin sonucunun yazmaca sığmaması demektir. 8 bitlik işaretsiz bir yazmaç en fazla 255 tutar. 200 ile 100 toplanırsa ne olur?

Örnek 1.3.4

1. **İşlenenler:** 200 = 128 + 64 + 8 → 11001000; 100 = 64 + 32 + 4 → 01100100.
2. **Gerçek toplam:** 300 = 256 + 32 + 8 + 4 → 100101100_2. Sonuç 9 bit gerektirir.
3. **Yazmaçta kalan:** Yazmaç yalnız 8 bit tutar. Ağırlığı 256 olan 9. bit kaybolur ve geriye 00101100 = 44 kalır.
4. **Kısa yol:** 300 − 256 = 44.



Şekil 1.3.7: 8 bitte 200 + 100. Gerçek toplam 300 = 100101100_2 9 bit gerektirir. 9. bit (256) yazmaca sığmaz; 8 bitte kalan 00101100 = 44'tür.

Tanım 1.3.4

Taşma (*overflow*), bir aritmetik işlemin gerçek sonucunun, sonucu tutacak yazmacın ya da değişkenin aralığına sığmamasıdır. Taşma olunca sığmayan yüksek bitler atılır ve yazmaçta yanlış bir değer kalır.

Taşma işaretli sayılarda da olur. 8 bitte 127 + 1 hesaplanırsa 01111111 + 1 = 10000000 bulunur ve bu dizi −128'dir. Şekil 1.3.4'teki kadranda 7'den −8'e geçiş, aynı olayın 4 bitlik karşılığıdır.

Taşma programlarda sessiz bir hatadır. Java tam sayı taşmasında hata vermez; sonuç sessizce yanlış çıkar:

```
byte b = (byte) (100 + 50);
System.out.println(b);
int buyuk = Integer.MAX_VALUE;
System.out.println(buyuk);
System.out.println(buyuk + 1);
```

Çıktı:

```
-106
2147483647
-2147483648
```

İlk satırda 100 + 50 = 150 hesaplanır ve sonuç `(byte)` ile 8 bite sıkıştırılır. 150, `byte` aralığına sığmaz; 8 bitlik dizisi 10010110'dır ve bu dizi işaretli okunduğunda 150 − 256 = −106'dır. Sonraki satırlarda `int`'in en büyük değerine 1 eklenir ve en küçük değer elde edilir. Bu yüzden bir değişkenin tipi seçilirken alabileceği en büyük değer düşünülmelidir. Örneğin bir ülkenin nüfusu `short`'a sığmaz; dünya nüfusu `int`'e bile sığmaz (yaklaşık 8 milyar > 2,1 milyar).

1.3.2.9 Elde ile işaretli taşmanın farkı

Toplama devresi iki ayrı uyarı üretebilir ve ikisi sık karıştırılır. Birincisi, en soldaki basamaktan dışarı taşan **elde** (*carry*), yani 9. bittir. İkincisi **işaretli taşmadır** (*signed overflow*). İkisi farklı yorumların sorunudur.

- **Elde**, sayılar *işaretsiz* yorumlandığında sonucun 8 bite sığmadığını gösterir. İşaretli toplamada elde atılır ve sonuç yine doğru olabilir.
- **İşaretli taşma**, aynı işaretli iki sayının toplamı ters işaretli çıkınca oluşur. İki pozitifin toplamı negatif ya da iki negatifin toplamı pozitif görünüyorsa sonuç yanlıştır. Farklı işaretli iki sayının toplamı hiçbir zaman taşmaz, çünkü toplamın büyüklüğü iki sayıdan büyüğünü aşamaz.

İşlem	İkili toplam	Elde	İşaretli taşma	Sonuç
5 + (−9)	`00000101 + 11110111 = 11111100`	0	yok	−4 ✓
(−1) + (−1)	`11111111 + 11111111 = 1 11111110`	1	yok	−2 ✓
100 + 50	`01100100 + 00110010 = 10010110`	0	var	−106 ✗
(−100) + (−50)	`10011100 + 11001110 = 1 01101010`	1	var	+106 ✗

Tablo 1.3.4: 8 bitte dört toplama. Elde ve işaretli taşma birbirinden bağımsız oluşur.

Tablo 1.3.4, dört olası durumun hepsini gösterir. İlk satırda ne elde ne taşma vardır. İkinci satırda elde vardır, ama sonuç doğrudur: −1 + (−1) = −2. Elde yalnız işaretsiz yorumda sorundur; işaretsiz okunursa bu işlem 255 + 255 = 510'dur ve 8 bite sığmaz. Üçüncü satırda iki pozitif sayının toplamı negatif çıkmıştır; elde yoktur ama işaretli taşma vardır. Dördüncü satırda iki negatifin toplamı pozitif çıkmıştır; hem elde hem taşma vardır.

Örnek 1.3.5

8 bitte 120 + 10 ve (−3) + (−5) işlemlerinde elde ve işaretli taşma var mıdır?

1. **120 + 10:** 01111000 + 00001010 = 10000010. Toplam 8 bite sığdığı için elde 0'dır. Ama iki pozitif sayının toplamının işaret biti 1 çıkmıştır: işaretli taşma vardır. Dizi işaretli okunursa −126'dır; doğru sonuç olan 130, 8 bitlik işaretli aralığa sığmaz.
2. **(−3) + (−5):** −3 = 11111101, −5 = 11111011. Toplam 1 11111000'dir; elde 1'dir ve atılır. Kalan 11111000 negatiftir ve büyüklüğü 00001000 = 8'dir: sonuç −8, doğrudur. İki negatifin toplamı negatif çıktığı için işaretli taşma yoktur.

Sık yapılan hata

Elde ile işaretli taşmayı aynı şey sanmak. Elde işaretsiz yorumun, taşma işaretli yorumun sorunudur. Biri olurken öteki olmayabilir: (−1) + (−1)'de elde vardır ama sonuç doğrudur; 100 + 50'de elde yoktur ama işaretli sonuç yanlıştır.

1.3.3 Kayan noktalı sayılar

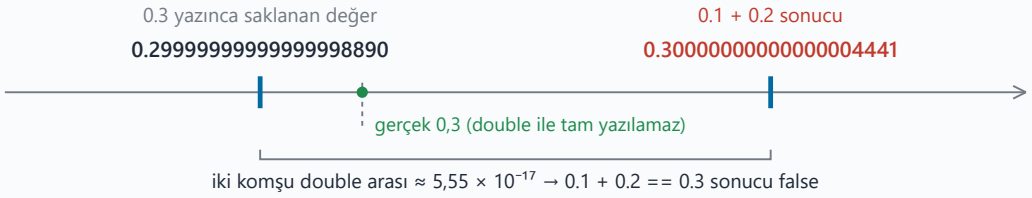
Tam sayılar sayma işlerinde yeterlidir, ama mühendislik hesaplarının çoğu kesirli değerlerle yapılır: bir direncin üzerindeki gerilim 3,3 V, bir sinyalin periyodu 0,000016 s, bir yıldızın uzaklığı $4 \cdot 10^{16}$ m olabilir. Bu kesim, böyle sayıların sınırlı bitle nasıl saklandığını ve bunun bedelini anlatır.

1.3.3.1 Neden kayan nokta?

Bilimsel gösterimde çok büyük ve çok küçük sayılar aynı kalıpla yazılır: $4 \cdot 10^{16}$, $1{,}6 \cdot 10^{-5}$. Sayı iki parçaya ayrılır: anlamlı basamaklar ve virgülün yerini söyleyen üs. Üs değiştikçe virgül sola ya da sağa "kayar". Bilgisayardaki **kayan nokta** (*floating point*) gösterimi aynı fikri 10 yerine 2 tabanıyla uygular.

Bu gösterimin dört özelliği vardır.

- **Geniş aralık.** Çok büyük ve çok küçük değerler aynı sayıda bitle saklanır. 32 bitlik `float` yaklaşık $1{,}2 \cdot 10^{-38}$ ile $3{,}4 \cdot 10^{38}$ arasındaki değerleri tutabilir.
- **Sınırlı hassasiyet.** Anlamlı basamak sayısı sabittir. `float` yaklaşık 7, `double` yaklaşık 15–16 onluk basamak taşır. Her ondalık değer sonlu bitlerle tam temsil edilemez.
- **Yuvarlama.** Saklanamayan basamaklar yuvarlanır; hesaplama sonuçlarında küçük farklar oluşabilir.
- **Karşılaştırma.** Bu küçük farklar yüzünden kayan noktalı sonuçlar tam eşitlikle değil, bir tolerans değeriyle karşılaştırılır.



Şekil 1.3.8: 0,3 çevresinde sayı doğrusu. Programda 0,3 yazıldığında saklanan `double` değeri 0,29999999999999998890'dır. 0,1 + 0,2'nin sonucu bir sonraki `double` 'dır: 0,30000000000000004441. Gerçek 0,3 bu ikisinin arasındadır ve `double` ile tam yazılamaz. İki komşu `double` arasındaki uzaklık yaklaşık $5{,}55 \cdot 10^{-17}$'dir.

Şekil 1.3.8, sınırlı hassasiyetin ne demek olduğunu somut olarak gösterir. Sayı doğrusundaki her nokta bir `double` ile yazılamaz; yalnız belirli noktalar yazılabilir ve bunların arasında boşluklar vardır. 0,3 bu noktalardan biri değildir. Program 0,3 yazınca ona en yakın nokta saklanır. 0,1 ve 0,2 de tam saklanamaz; yuvarlanmış hâlleri toplanınca sonuç, 0,3'ün yuvarlanmış hâli yerine bir sonraki noktaya düşer. Bu yüzden Java'da `0.1 + 0.2` ekrana `0.30000000000000004` olarak yazılır ve `0.1 + 0.2 == 0.3` karşılaştırması `false` verir.

1.3.3.2 Ondalık kesri ikiliye çevirmek

İkili sistemde virgülden sonraki basamakların ağırlıkları 1/2, 1/4, 1/8, 1/16, … diye yarıya iner. Örneğin $0{,}101_2$ = 1/2 + 1/8 = 0,625'tir. Ondalık bir kesri ikiliye çevirmek için tam sayılardaki bölme yönteminin tersi kullanılır.

Tanım 1.3.5

Kesri ikiliye çevirme yöntemi: Kesir 2 ile çarpılır. Çarpımın tam kısmı (0 ya da 1) virgülden sonraki bir sonraki bittir. Tam kısım atılır ve kalan kesirle devam edilir. Kesir 0 olunca açılım biter. Daha önce görülmüş bir kesir yeniden çıkarsa açılım döngüye girmiştir ve hiç bitmez.

Örnek 1.3.6

0,375 sayısını ikiliye çevirelim.

1. 0,375 × 2 = 0,75 → tam kısım 0, ilk bit 0.
2. 0,75 × 2 = 1,5 → tam kısım 1, ikinci bit 1; kesir 0,5 ile devam edilir.
3. 0,5 × 2 = 1,0 → tam kısım 1, üçüncü bit 1; kesir 0 oldu, açılım bitti.

Sonuç 0,375 = $0{,}011_2$'dir. Denetim: 1/4 + 1/8 = 0,25 + 0,125 = 0,375.

0,375'in ikili açılımı sonlu olduğu için bu sayı bilgisayarda tam saklanır. Bunun nedeni, 0,375'in 3/8 olması ve paydasının 2'nin bir kuvveti olmasıdır. Genel kural şudur: sadeleştirilmiş kesrin paydası yalnız 2'nin kuvvetlerinden oluşuyorsa ikili açılım sonludur; aksi hâlde sonsuza dek tekrarlar. 0,5, 0,25, 0,75 ve 0,375 tam saklanır. 0,1 = 1/10, 0,2 = 1/5 ve 0,3 = 3/10 ise paydalarında 5 çarpanı bulunduğu için tam saklanamaz.

1.3.3.3 0,1 neden tam saklanamaz?

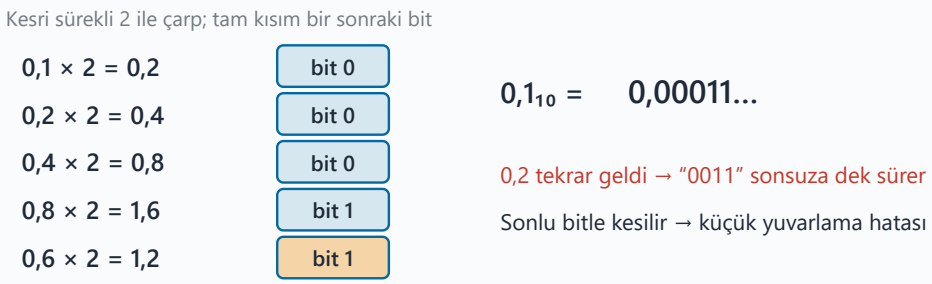
Aynı yöntem 0,1'e uygulandığında ne olduğunu adım adım görelim.

Adım	İşlem	Bit	Kalan kesir
1	0,1 × 2 = 0,2	0	0,2
2	0,2 × 2 = 0,4	0	0,4
3	0,4 × 2 = 0,8	0	0,8
4	0,8 × 2 = 1,6	1	0,6
5	0,6 × 2 = 1,2	1	0,2 (2. adımdaki kesir)

Tablo 1.3.5: 0,1'in ikili açılımı. Beşinci adımdan sonra kesir yeniden 0,2 olur ve açılım döngüye girer.

Beşinci adımın sonunda kesir yine 0,2 olmuştur. 0,2 ikinci adımda zaten işlenmişti; bundan sonra 2.–5. adımlar aynen tekrarlanır ve 0011 bitleri sonsuza dek üretilir:

> $0{,}1_{10} = 0{,}0\ 0011\ 0011\ 0011\ \ldots_2$



Şekil 1.3.9: 0,1'in ikiliye çevrilmesi. Beş çarpmadan sonra 0,2 kesri yeniden görülür; "0011" deseni sonsuza dek sürer. Sonlu bitle kesilince küçük bir yuvarlama hatası oluşur.

Bilgisayar sonsuz bit saklayamaz. Açılım belli bir yerde kesilir ve son bit yuvarlanır. Saklanan sayı 0,1'e çok yakındır ama ona eşit değildir. Aynı durum, onluk sistemde 1/3'ün 0,3333… diye yazılmasına benzer: kaç basamak yazılırsa yazılsın 1/3'e tam ulaşılmaz. Onluk sistemde sorun çıkarmayan 0,1, ikili sistemde 1/3 gibi davranır.

Küçük yuvarlama hataları art arda yapılan işlemlerde birikebilir. Aşağıdaki kod 0,1'i on kez toplar:

```
double toplam = 0.0;
for (int i = 0; i < 10; i++) {
    toplam = toplam + 0.1;
}
System.out.println(toplam);
System.out.println(toplam == 1.0);
System.out.println(Math.abs(toplam - 1.0) < 1e-9);
```

Çıktı:

```
0.9999999999999999
false
true
```

Matematikte 10 · 0,1 = 1'dir, ama döngü 0,9999999999999999 üretir. Bu yüzden ikinci satırdaki `==` karşılaştırması `false` verir. Son satır doğru yöntemi gösterir: iki sayının farkının mutlak değeri, küçük bir eşikten (burada 10^{-9}) küçükse sayılar "eşit" kabul edilir.

Sık yapılan hata

Kayan noktalı sonuçları `==` ile karşılaştırmak. 0,1 + 0,2 ≠ 0,3 olduğu için `if (x == 0.3)` beklenmedik biçimde atlanabilir. Doğrusu, `Math.abs(a - b) < eşik` biçiminde bir toleransla karşılaştırmaktır. Eşik, hesaplanan büyüklüklerin ölçeğine göre seçilir.

1.3.3.4 IEEE 754 biçimi

Kayan noktalı sayıların bitlere nasıl yerleşeceği, bütün işlemcilerin uyduğu **IEEE 754** standardıyla belirlenir. Standart, sayıyı ikili bilimsel gösterime getirir:

$$\text{değer} = (-1)^{\text{işaret}} \times 1{,}\text{kesir} \times 2^{(\text{üs} - \text{sapma})}$$

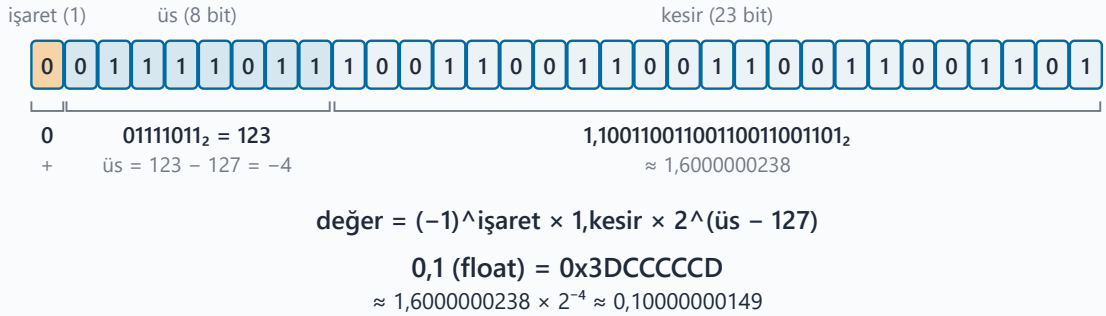
Bit dizisi üç alana bölünür: **işaret**, **üs** ve **kesir** (*sign, exponent, fraction*). Tablo 1.3.6, Java'daki iki kayan noktalı tipin alanlarını gösterir.

Tip	Toplam	İşaret	Üs	Kesir	Sapma	Onluk basamak
`float`	32 bit	1	8	23	127	yaklaşık 7
`double`	64 bit	1	11	52	1023	yaklaşık 15–16

Tablo 1.3.6: IEEE 754 biçimleri. Sapma, üs alanına yazılmadan önce gerçek üsse eklenen sabittir.

Alanların görevleri şöyledir.

- **İşaret biti** 0 ise sayı pozitif, 1 ise negatiftir. Kayan noktalı sayılarda ikiye tümleyen kullanılmaz; büyüklük her zaman pozitif saklanır.
- **Üs alanına** gerçek üs değil, ona **sapma** (*bias*) eklenmiş hâli yazılır. `float`'ta sapma 127'dir. Böylece negatif üsler de işaretsiz bir sayı olarak saklanır: üs −4 ise alana 123 yazılır.
- **Kesir alanı**, virgülden sonraki bitleri tutar. Normal sayılarda virgülün solunda her zaman tek bir 1 bulunur. Bu 1 hiç değişmediği için saklanmaz; standart bir bit kazanmış olur.



Şekil 1.3.10: 0,1'in 32 bitlik `float` gösterimi. İşaret 0; üs alanı $01111011_2 = 123$, yani gerçek üs 123 − 127 = −4; kesir 1001 1001 … 1101. Değer $1{,}6000000238 \times 2^{-4} \approx 0{,}10000000149$'dur ve onaltılık yazımı

3DCCCCCD'dir.

Şekil 1.3.10, 0,1'in `float` olarak nasıl saklandığını gösterir. 0,1 = 1,1001 1001 1001…$_2$ × 2^{-4}'tür. Kesirdeki 1001 deseni sonsuza dek tekrarlar. 23 bite sığmayınca son bit yuvarlanır; bu yüzden kesir alanının sonunda 1100 yerine 1101 görülür. Saklanan değer tam olarak 0,100000001490116119384765625'tir: 0,1'e çok yakın, ama eşit değil. Java bu değeri `BigDecimal` ile bütün basamaklarıyla gösterebilir:

```
System.out.println(Integer.toHexString(Float.floatToIntBits(-6.25f)));
System.out.println(Integer.toHexString(Float.floatToIntBits(0.1f)));
System.out.println(new java.math.BigDecimal(0.1f));
```

Çıktı:

```
c0c80000
3dcccccd
0.100000001490116119384765625
```

`Float.floatToIntBits` metodu, bir `float`'ın 32 bitini olduğu gibi bir `int` içine kopyalar; `Integer.toHexString` de bu bitleri onaltılık yazar. İlk satırdaki c0c80000, bir sonraki kesimde elle bulunacak.

1.3.3.5 −6,25'i float olarak kodlama

IEEE 754'ü anlamanın en iyi yolu, bir sayıyı elle kodlamaktır. Sunumdaki adımlar aşağıdaki sözde kodla özetlenmiştir:

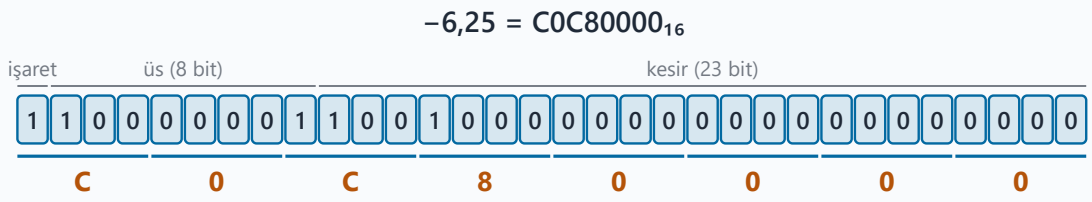
```
x ← -6,25
işaret ← 1          ; negatif
ikili ← 110,01      ; 6 = 110, 0,25 = 0,01
norm ← 1,1001 × 2²  ; virgül 2 sola
üs ← 2 + 127 = 129  ; 10000001
kesir ← 1001 0000…  ; baştaki 1 yazılmaz
yaz(onaltılık(bitler))
```

Örnek 1.3.7

−6,25 sayısının 32 bitlik `float` karşılığını bulalım.

1. **İşaret:** Sayı negatif olduğundan işaret biti 1'dir. Geri kalan işlemler 6,25 üzerinde yapılır.

2. **İkiliye çevirme:** Tam kısım 6 = 110_2. Kesir 0,25 için 0,25 × 2 = 0,5 → 0; 0,5 × 2 = 1,0 → 1 ve kesir 0'a iner: $0{,}01_2$. Demek ki 6,25 = $110{,}01_2$'dir. 0,1'in aksine açılım sonludur ve sayı tam saklanır.
3. **Normalleştirme:** Virgül, solunda tek bir 1 kalacak kadar kaydırılır. 110,01'de virgül 2 basamak sola gider: $1{,}1001 \times 2^2$. Gerçek üs 2'dir.
4. **Üs alanı:** Sapma eklenir: 2 + 127 = 129 = 10000001_2.
5. **Kesir alanı:** Baştaki 1 yazılmaz; yalnız virgülden sonraki 1001 yazılır. Kalan 19 bit 0 ile doldurulur: 1001 0000 0000 0000 0000 000.
6. **Bitleri birleştirme:** 1 | 10000001 | 10010000000000000000000. 32 bit dörderli gruplanır: 1100 0000 1100 1000 0000 0000 0000 0000 → $C0C80000_{16}$.



Şekil 1.3.11: −6,25'in 32 biti. İşaret 1, üs 10000001 (129), kesir 1001 ve on dokuz 0. Dörderli gruplar C, 0, C, 8, 0, 0, 0, 0'dır: −6,25 = $C0C80000_{16}$.

Elle bulunan C0C80000, Java'nın `Float.floatToIntBits(-6.25f)` sonucuyla (c0c80000) aynıdır. Ters yönde okuma da aynı adımların tersidir. Örneğin 41280000_{16}'ın bitleri 0 | 10000010 | 0101000...'dır. İşaret 0, gerçek üs 130 − 127 = 3, sayı $1{,}0101_2 \times 2^3 = 1010{,}1_2$ = 10,5'tir.

Sık yapılan hata

Üs alanına gerçek üssü yazmak ya da kesir alanına baştaki 1'i de koymak. −6,25 örneğinde üs alanına 2 (00000010) değil 129 (10000001) yazılır. Kesir alanı 11001 ile değil 1001 ile başlar; virgülün solundaki 1 saklanmaz.

1.3.3.6 Kayan noktalı sayılarla çalışırken

Kayan noktalı sayıların sınırlarını bilmek, gündelik programlamada üç pratik kurala dönüşür.

- **Eşitlik yerine tolerans.** Hesaplanmış iki değer `==` ile karşılaştırılmaz; farklarının mutlak değeri küçük bir eşikle karşılaştırılır.
- **Hassasiyet için `double`.** `float`'ın yaklaşık 7 basamağı çoğu mühendislik hesabı için azdır. Java'da kesirli sabitler varsayılan olarak `double`'dır ve genellikle o tercih edilir.
- **Para için tam sayı.** Kuruş hatası kabul edilmeyen hesaplarda tutarlar kuruş cinsinden tam sayı olarak ya da onluk tabanda tam hesap yapan bir tiple (Java'da `BigDecimal`) tutulur.

Tam sayılarla kayan noktalı sayıların farkı burada netleşir. İkili gösterim, aralığa sığan her tam sayı için kesindir: 13, 13 olarak saklanır. Kayan nokta ise bazı ondalık değerler için yaklaşıktır. Taşma tam sayıların, yuvarlama kayan noktalı sayıların tipik sorunudur; ikisinin de kaynağı sınırlı bit sayısıdır.

1.3.4 Metnin temsili

Bilgisayar harfleri doğrudan saklayamaz; yalnız sayıları saklar. Metni saklamak için her karaktere bir numara verilir ve bu numara bitlerle yazılır. Hangi karaktere hangi numaranın verileceğini bir kodlama tablosu belirler. Bu kesim, en eski ve en basit tablo olan ASCII'den başlar, bütün dünya yazılarını kapsayan Unicode'a ve onu baytlara yazan UTF-8'e ulaşır.

1.3.4.1 ASCII

ASCII (*American Standard Code for Information Interchange*), temel Latin harflerini, rakamları, noktalama işaretlerini ve kontrol karakterlerini sayısal kodlarla eşleştirir. 1960'larda tanımlanmış 7 bitlik bir koddur ve $2^7 = 128$ karakter (0–127) içerir. Bir bayt içinde saklandığında en yüksek bit 0'dır.

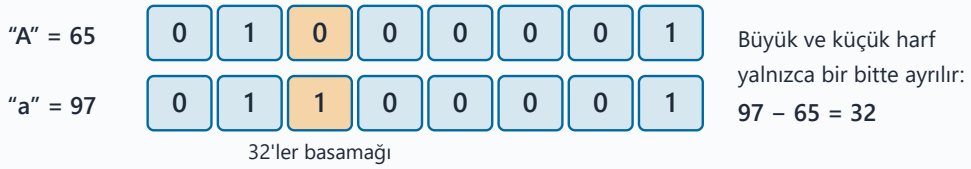
Karakter	Onluk	Onaltılık	İkili (8 bit)
“A”	65	41	`01000001`
“Z”	90	5A	`01011010`
“a”	97	61	`01100001`
“z”	122	7A	`01111010`
“0”	48	30	`00110000`
“7”	55	37	`00110111`
boşluk	32	20	`00100000`
satır sonu (LF)	10	0A	`00001010`

Tablo 1.3.7: Bazı ASCII kodları. Büyük harfler 65'ten, küçük harfler 97'den, rakamlar 48'den başlar ve ardışıktır.

Tablodaki son satır bir **kontrol karakteridir** (*control character*). Kontrol karakterleri ekranda bir şekil olarak görünmez; metnin nasıl düzenleneceğini söyler. Satır sonu (LF, *line feed*), imlecin alt satıra geçmesi gerektiğini bildirir. ASCII'nin ilk 32 kodu (0–31) ve 127 bu türdendir.

ASCII tablosunun düzeni rastgele değildir; iki özelliği hesap yapmayı kolaylaştırır.

- **Harfler ve rakamlar ardışıktır.** “B” = 66, “C” = 67 diye gider. Bir rakam karakterinin sayısal değeri, kodundan “0”ın kodu çıkarılarak bulunur: “7” − “0” = 55 − 48 = 7.
- **Büyük ve küçük harf arasında hep 32 fark vardır.** “a” − “A” = 97 − 65 = 32. $32 = 2^5$ olduğu için iki harf yalnız tek bir bitte ayrılır.



Şekil 1.3.12: "A" = 65 = 01000001 ile "a" = 97 = 01100001 yalnız 32'ler basamağındaki bitte farklıdır.

Sık yapılan hata

Rakam karakteri ile sayıyı karıştırmak. Klavyeden girilen "7" bir karakterdir ve kodu 55'tir. Sayısal değeri 7 değil 55 olarak hesaba girerse sonuç yanlış çıkar. Doğrusu, "7" − "0" = 7 dönüşümünü yapmaktır.

ASCII'nin büyük eksiği kapsamıdır. 128 karakter İngilizce için yeterlidir, ama "ç, ğ, ı, ö, ş, ü" gibi Türkçe harfler, Yunan ve Arap alfabeleri, Çince karakterler ve matematik simgeleri tabloda yoktur. 1980'ler ve 90'larda her dil için ayrı 8 bitlik tablolar yapıldı; aynı bayt bir tabloda "ğ", ötekinde başka bir harf demekti. Türkçe metinlerin başka bir bilgisayarda bozuk görünmesinin nedeni çoğunlukla budur. Unicode bu karmaşayı çözmek için geliştirildi (1.3.4.3).

1.3.4.2 Bit işlemleri: harfin büyüklüğünü değiştirme

Büyük ve küçük harfin tek bir bitte ayrılması, **bit işlemlerini** (*bitwise operation*) tanıtmak için iyi bir örnektir. Bit işlemleri, iki sayının aynı konumdaki bitlerini tek tek işler. Bir sayının yalnız bazı bitlerini değiştirmek için **bit maskesi** (*bit mask*) kullanılır: maskede 1 olan konumlar işleme girer, 0 olanlar korunur.

"G" = 71 ile "g" = 103 arasındaki fark 32 = 2^5'tir. Demek ki iki harf 5. bitte ayrılır (bitler sağdan ve 0'dan başlanarak numaralanır). Maske olarak yalnız bu biti 1 olan 00100000 seçilir. Sunumdaki sözde kod şöyledir:

```
c ← 'G'            ; 71 = 01000111
m ← 00100000       ; maske: 5. bit (32)
k ← c OR m         ; biti 1 yap
b ← k AND (NOT m)  ; biti 0 yap
t ← b XOR m        ; biti ters çevir
s ← c << 1         ; sola kaydır
```

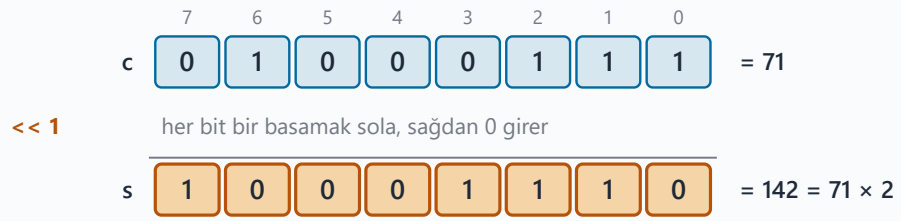
İşlem	Bitler	Değer	Kural
`c = "G"`	`01000111`	71 = "G"	başlangıç
`c OR m`	`01100111`	103 = "g"	maskede 1 olan bit 1 olur
`k AND (NOT m)`	`01000111`	71 = "G"	NOT m = 11011111; maskedeki bit 0'a çekilir

İşlem	Bitler	Değer	Kural
`b XOR m`	`01100111`	103 = "g"	maskedeki bit ters çevrilir
`c << 1`	`10001110`	142 = 71 × 2	bitler bir basamak sola kayar, sağdan 0 girer
`c >> 1`	`00100011`	35 = 71 ÷ 2 (tam kısım)	bitler bir basamak sağa kayar

Tablo 1.3.8: "G" harfi üzerinde bit işlemleri. Maske `m` = 00100000 (32).

Her işlemin ne işe yaradığı tablonun son sütununda görülür. **OR** bir biti kesinlikle 1 yapar: harf zaten küçükse değişmez, büyükse küçülür. **AND** ters maskeyle bir biti kesinlikle 0 yapar: harf büyüğe döner. **XOR** biti ters çevirir: büyük harf küçülür, küçük harf büyür. Aynı maskeyle ikinci kez XOR yapılırsa eski değer geri gelir; bu özellik basit şifreleme yöntemlerinde de kullanılır.

Bit kaydırma (*bit shift*) ise bütün bitleri birlikte yerinden oynatır. Sola bir kaydırmada her bit bir basamak yükselir ve ağırlığı iki katına çıkar; değer 2 ile çarpılmış olur. Sağa kaydırma 2'ye böler ve kesirli kısım atılır: 71 >> 1 = 35.



Şekil 1.3.13: "G" = 71'in bir basamak sola kaydırılması. Her bit bir basamak yükselir, sağdan 0 girer: 10001110 = 142 = 71 × 2.

Java'da bit işlemleri `|` (OR), `&` (AND), `~` (NOT), `^` (XOR), `<<` ve `>>` işleçleriyle yazılır. `char` tipi Java'da sayısal bir koddur; bu yüzden harflerle doğrudan bit işlemi yapılabilir:

```
char c = 'G';
System.out.println((int) c);
System.out.println((char) (c | 32));
System.out.println((char) ('g' & ~32));
System.out.println((char) (c ^ 32));
System.out.println(c << 1);
System.out.println(c >> 1);
```

Çıktı:

```
71
g
G
g
142
35
```

Bit işleminin sonucu `int`'tir. Ekranda harf görmek için sonuç `(char)` ile yeniden karaktere çevrilir. Son iki satırda bu çevirme yapılmadığı için sayılar yazılır. Aynı bitlerin bir yerde harf, bir yerde sayı olarak görünmesi, 1.3.1.2'deki ilkenin, yani anlamı veri tipinin verdiğinin bir başka örneğidir.

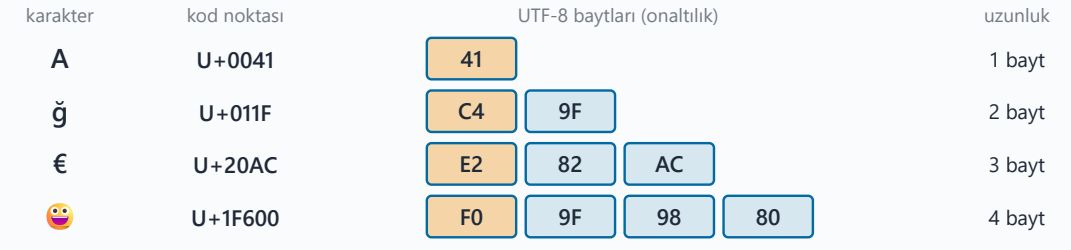
Sık yapılan hata

Harf büyüklüğünü değiştirmek için hep 32 eklemek ya da çıkarmak. "G" + 32 doğru sonucu verir, ama "g" + 32 = 135 tanımsız bir koddur. OR ile küçültme ve AND ile büyütme, harf zaten istenen hâldeyse onu değiştirmez; bu yüzden daha güvenlidir. Bu yöntem yalnız ASCII'deki İngilizce harflerde geçerlidir; Türkçedeki "ı/I" ve "i/İ" çiftleri bu kurala uymaz.

1.3.4.3 Unicode ve kod noktası

Unicode, dünyadaki yazı sistemlerinin karakterlerine tek ve evrensel bir numara verir. Bu numaraya **kod noktası** (*code point*) denir ve "U+" önekiyle onaltılık yazılır. "A"nın kod noktası U+0041'dir; bu, onluk 65 eder ve ASCII koduyla aynıdır. Unicode ilk 128 kod noktasını bilerek ASCII ile aynı tutar. "ğ" U+011F, "€" U+20AC, gülen yüz emojisi U+1F600'dür. Unicode bugün Türkçe harfleri, farklı alfabeleri, matematik simgelerini ve emojiyi kapsayan yüz binlerce karakter içerir.

Kod noktası yalnız bir numaradır. Numaranın dosyada ya da bellekte hangi baytlarla yazılacağı ayrı bir karardır. Aynı kod noktası farklı Unicode dönüşüm biçimleriyle saklanabilir: UTF-8, UTF-16 ve UTF-32 bunların en bilinenleridir. Bir metin dosyası açılırken hangi biçimle yazıldığı bilinmezse karakterler bozuk görünür.



Şekil 1.3.14: Dört karakterin kod noktası ve UTF-8 baytları. "A" (U+0041) 1 bayt: 41. "ğ" (U+011F) 2 bayt: C4 9F. "€" (U+20AC) 3 bayt: E2 82 AC. Gülen yüz (U+1F600) 4 bayt: F0 9F 98 80.

1.3.4.4 UTF-8 kodlama: “ğ”

UTF-8, Unicode kod noktalarını değişken uzunlukta bayt dizileriyle yazar. Küçük kod noktaları az, büyükleri çok bayt kaplar. Hangi baytın hangi görevde olduğu, baytın başındaki bitlerden anlaşılır. Tablo 1.3.9, kuralları özetler.

Kod noktası aralığı	Bit sayısı	Bayt şablonu	Örnek
U+0000 – U+007F	7	`0xxxxxxx`	“A” → 41
U+0080 – U+07FF	11	`110xxxxx 10xxxxxx`	“ğ” → C4 9F
U+0800 – U+FFFF	16	`1110xxxx 10xxxxxx 10xxxxxx`	“€” → E2 82 AC
U+10000 – U+10FFFF	21	`11110xxx 10xxxxxx 10xxxxxx 10xxxxxx`	emoji → F0 9F 98 80

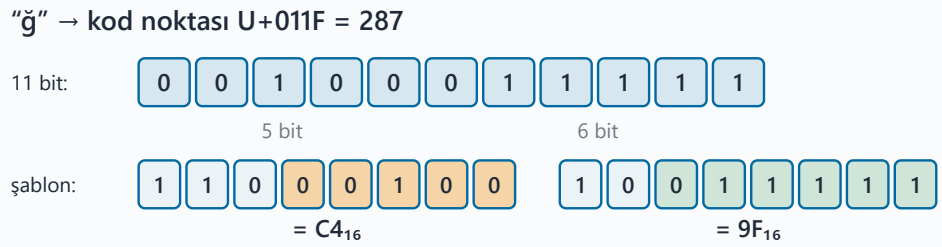
Tablo 1.3.9: UTF-8 şablonları. `x` konumlarına kod noktasının bitleri soldan sağa yerleştirilir.

Şablonların başındaki bitler birer etikettir. 0 ile başlayan bayt tek başına bir ASCII karakteridir. 110 “iki baytlık karakterin ilk baytı”, 1110 “üç baytlık karakterin ilk baytı”, 11110 “dört baytlık karakterin ilk baytı” demektir. 10 ile başlayan bayt ise bir **devam baytıdır**. Bu etiketler sayesinde metnin ortasından okumaya başlayan bir program bile karakter sınırlarını bulabilir.

Örnek 1.3.8

“ğ” harfinin UTF-8 baytlarını bulalım.

1. **Kod noktası:** Unicode, “ğ”ya U+011F'yi atar. $011F_{16} = 1 \cdot 256 + 1 \cdot 16 + 15 = 287$.
2. **İkili yazım:** $287 = 256 + 16 + 8 + 4 + 2 + 1 = 100011111_2$. Sayı 7 bite sığmaz (ASCII değildir), ama 11 bite sığar. Demek ki 2 bayt gerekir. 11 bite tamamlamak için başa iki 0 eklenir: 00100011111.
3. **Bölme:** İki baytlık şablon `110xxxxx 10xxxxxx` 'tir ve 5 + 6 = 11 boş yer vardır. 11 bit soldan 5 + 6 olarak bölünür: 00100 | 011111.
4. **Yerleştirme:** İlk bayt 110 + 00100 = 11000100 = $C4_{16}$. İkinci bayt 10 + 011111 = 10011111 = $9F_{16}$.
5. **Sonuç:** “ğ” = C4 9F, 2 bayt.



Şekil 1.3.15: "ğ" = U+011F = 287'nin 11 biti, 5 + 6 olarak `110xxxxx 10xxxxxx` şablonuna yerleştirilir: 11000100 10011111 = C4 9F.

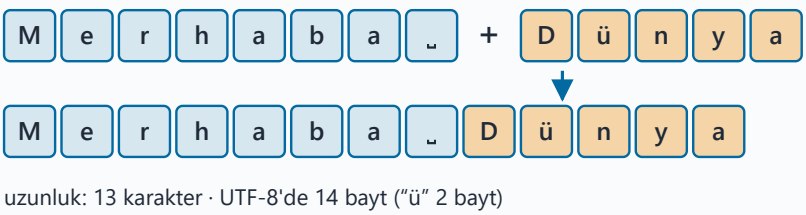
Aynı yöntem "ç" için de çalışır. "ç"nin kod noktası U+00E7 = 231'dir; 11 bitle 00011100111 yazılır. 5 + 6 bölmesiyle 00011 | 100111 elde edilir ve baytlar 11000011 10100111 = C3 A7 olur. Türkçeye özgü harflerin hepsi U+0080 – U+07FF aralığındadır; bu yüzden UTF-8'de her biri 2 bayt tutar. ASCII karakterleri 1, "€" 3, emoji 4 bayt tutar.

UTF-8'in en büyük üstünlüğü ASCII ile uyumudur. ASCII karakterleri tek baytla ve ASCII'deki kodlarıyla yazıldığı için eski ASCII metinler aynı zamanda geçerli UTF-8'dir. Web sayfalarının büyük çoğunluğu bugün UTF-8 ile yazılır. Bir UTF-8 dosyası yanlışlıkla eski bir 8 bitlik tabloyla okunursa "ğ"nin iki baytı iki ayrı karakter gibi görünür; ekranda "ÄŸ" gibi anlamsız harflerin çıkmasının nedeni budur.

1.3.4.5 Dizgiler

Dizgi (*string*), sıralı karakterlerden oluşan veri tipidir. Bir ad, bir cümle ya da bir dosyanın bütün içeriği birer dizgidir. Dizgiler üzerinde yapılan temel işlemler şunlardır:

- **Uzunluk:** karakter ya da kod birimi sayısı.
- **Birleştirme:** iki dizgiyi uç uca ekleme.
- **Arama:** bir karakteri ya da alt dizgiyi bulma.
- **Karşılaştırma:** içerik eşitliğini ve alfabetik sıralama ilişkisini denetleme.



Şekil 1.3.16: "Merhaba " ile "Dünya"nın birleştirilmesi. Sonuç 13 karakterlik bir dizgidir; UTF-8'de 14 bayt tutar, çünkü "ü" 2 bayttır.

Şekil 1.3.16, önemli bir ayrımı gösterir: ekranda görünen karakter sayısı ile bellekteki bayt sayısı her zaman aynı değildir. “Merhaba Dünya” 13 karakterdir, ama UTF-8'de 14 bayt tutar. Belleği ayırırken, bir dosyanın boyutunu hesaplarken ya da bir veritabanı alanının uzunluğunu seçerken hangisinin sayıldığı bilinmelidir. Bellekteki saklama biriminin adı **kod birimidir** (*code unit*): UTF-8'de kod birimi 1 bayt, Java'nın kullandığı UTF-16'da 2 bayttır.

```
String s = "çağ";
System.out.println(s.length());
byte[] b = s.getBytes(java.nio.charset.StandardCharsets.UTF_8);
System.out.println(b.length);
for (byte x : b) {
    System.out.printf("%02X ", x);
}
System.out.println();
```

Çıktı:

```
3
5
C3 A7 61 C4 9F
```

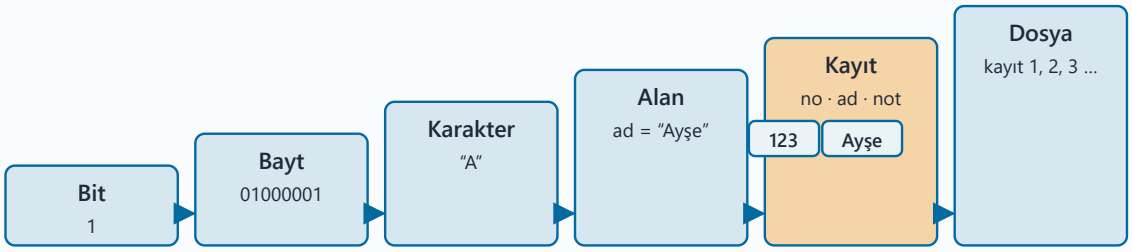
Java'da `length()` metodu UTF-16 kod birimlerini sayar; “çağ” için 3 verir. `getBytes` ise dizgiyi UTF-8 baytlarına çevirir: “ç” C3 A7, “a” 61, “ğ” C4 9F. Toplam 5 bayttır. Bir emoji UTF-16'da iki kod birimi tuttuğu için onun `length()`'i 2 çıkar; “karakter sayısı” kavramı Unicode'da göründüğünden daha inceliklidir.

> **Sık yapılan hata**
>
> Karakter sayısını bayt sayısı sanmak. UTF-8'de “çağ” 3 karakter, 5 bayttır (ç 2 + a 1 + ğ 2). 10 baytlık bir alana 10 Türkçe harf sığmayabilir.

1.3.5 Yapılandırılmış bilgi ve sık yapılan hatalar

1.3.5.1 Bitten dosyaya

Bitler tek başına anlam taşımaz; daha büyük yapılar içinde düzenlenerek anlam kazanır. Bu yapılar katman katman kurulur.



Şekil 1.3.17: Bit, bayt, karakter, alan, kayıt ve dosya. Her düzey bir öncekinden oluşur: sekiz bit bir bayt, bir ya da birkaç bayt bir karakter, karakterler bir alan, alanlar bir kayıt, kayıtlar bir dosya oluşturur.

- **Bit ve bayt:** en küçük birimler. Sekiz bit bir bayt eder.
- **Karakter:** ASCII'de bir, UTF-8'de bir ile dört arası bayt.
- **Alan** (*field*): anlamlı tek bir bilgi; örneğin bir ad ("Ayşe") ya da bir not (85).
- **Kayıt** (*record*): birbiriyle ilgili alanların topluluğu. Bir öğrenci kaydı numara, ad ve not alanlarından oluşabilir.
- **Dosya:** kayıtların ve başka verilerin diskte saklanan dizisi.

Bir dosyanın içindeki baytların nasıl okunacağını **dosya biçimi** (*file format*) tanımlar: alanların sırası, her alanın kaç bayt olduğu ve hangi kodlamayla yazıldığı. Aynı baytlar bir program için öğrenci listesi, başka bir program için anlamsız sayılardır. Bir resim dosyasını metin düzenleyiciyle açınca karmakarışık karakterler görülmesinin nedeni budur: baytlar yanlış biçimle yorumlanmıştır.

1.3.5.2 Veri temsilinde sık yapılan hatalar

Veri temsiliyle ilgili kesimlerde dağınık olarak geçen hatalar, toplu bir başvuru için Tablo 1.3.10'da bir araya getirilmiştir.

Sık yapılan hata	Doğrusu
Tabanı yazmadan okumak: "1101 bin yüz birdir"	1101_2 = 13. Taban alt simgeyle ya da `0b` / `0x` önekiyle belirtilir.
10_{16}'yı on sanmak	10_{16} = 16; onaltılıkta on, A ile yazılır.
İkiye tümleyende yalnız bitleri çevirmek	11111010 birler tümleyenidir (−5 değil); 1 eklenince 11111011 = −5.
Kayan noktalı sonuçları `==` ile karşılaştırmak	0,1 + 0,2 ≠ 0,3; $\lvert a - b \rvert$ küçük bir toleranstan küçük mü diye bakılır.
Rakam karakteri ile sayıyı karıştırmak	"7" karakterinin kodu 55'tir; sayısal değeri "7" − "0" = 7.
Karakter sayısını bayt sayısı sanmak	UTF-8'de "çağ" 3 karakter, 5 bayttır (ç 2 + a 1 + ğ 2).
Elde ile işaretli taşmayı karıştırmak	Elde işaretsiz yorumun, taşma işaretli yorumun sorunudur; biri olurken öteki olmayabilir.

Tablo 1.3.10: Veri temsilinde sık yapılan hatalar ve doğruları.

1.3.6 Mantık kapıları ve toplayıcılar

Önceki kesimlerde toplama hep kâğıt üzerinde yapıldı. İşlemci ise toplamayı elektronik devrelerle yapar. Bu devrelerin yapı taşı mantık kapılarıdır. Bu kesim, birkaç kapının bir araya gelerek iki biti, sonra da çok bitli sayıları nasıl topladığını gösterir.

1.3.6.1 Mantık kapıları ve doğruluk tabloları

Tanım 1.3.6

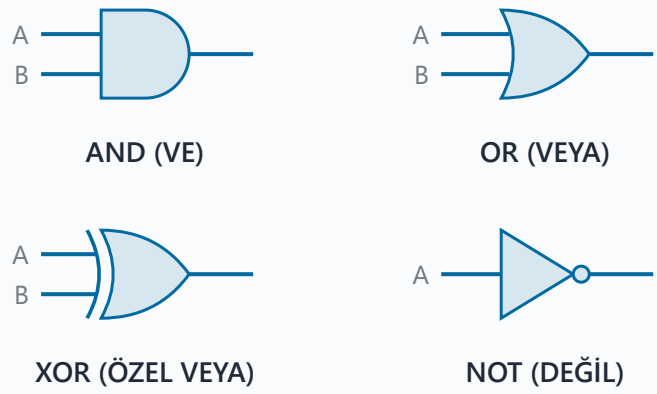
Mantık kapısı (*logic gate*), bir ya da iki bitlik girdiden tek bitlik bir çıktı üreten temel devredir. Bir kapının davranışı, bütün girdi bileşimleri için çıktıyı listeleyen **doğruluk tablosuyla** (*truth table*) tam olarak tanımlanır.

Dört temel kapı vardır:

- **AND** (VE): iki girdi de 1 ise 1 verir.
- **OR** (VEYA): girdilerden en az biri 1 ise 1 verir.
- **XOR** (ÖZEL VEYA): girdiler farklıysa 1 verir.
- **NOT** (DEĞİL): tek girdisini tersine çevirir.

A	B	AND	OR	XOR	NOT A
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Tablo 1.3.11: Dört temel kapının doğruluk tablosu.



Şekil 1.3.18: AND, OR, XOR ve NOT kapılarının devre simgeleri. Girdiler solda, çıktı sağdadır. NOT'un ucundaki küçük daire "tersine çevirme" anlamına gelir.

OR ile XOR arasındaki tek fark son satırdadır: iki girdi de 1 olunca OR 1, XOR 0 verir. XOR'un bu davranışı ikili toplamada sonuç basamağını üretir, çünkü 1 + 1'in basamağı 0'dır. AND ise 1 + 1'de oluşan eldeyi verir.

Kapılar birkaç transistörden kurulur. Transistör, bir gerilimle açılıp kapanan küçük bir elektronik anahtardır. Bugünkü bir işlemcide milyarlarca transistör bulunur ve işlemcideki bütün işlemler bu temel kapıların birleşiminden oluşur. Kapılar birbirine bağlanarak daha karmaşık mantık ifadeleri hesaplanır. Örneğin *X = NOT (A AND B) OR C* ifadesinde önce parantez, sonra NOT, en son OR hesaplanır. A = 1, B = 1, C = 0 için A AND B = 1, NOT 1 = 0 ve 0 OR 0 = 0 bulunur. C = 1 olsaydı OR'da tek bir 1 yeteceği için sonuç A ve B ne olursa olsun 1 olurdu.

1.3.6.2 Yarım toplayıcı

İki tek bitlik sayının toplamı en çok $1 + 1 = 2 = 10_2$ olabilir; sonuç iki bitle yazılır. Sağdaki bit sonuç basamağı (SUM), soldaki bit eldedir (CARRY). Tablo 1.3.12, dört durumu listeler.

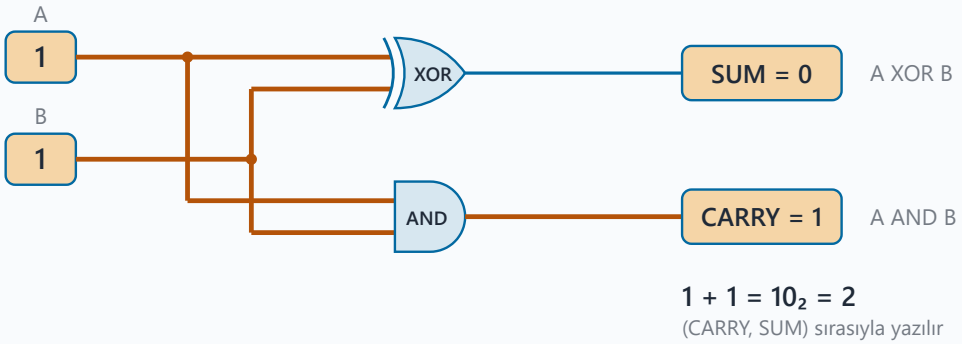
A	B	A + B	CARRY	SUM
0	0	$0 = 00_2$	0	0
0	1	$1 = 01_2$	0	1
1	0	$1 = 01_2$	0	1
1	1	$2 = 10_2$	1	0

Tablo 1.3.12: Yarım toplayıcının doğruluk tablosu. SUM sütunu XOR'un, CARRY sütunu AND'in tablosuyla aynıdır.

Tabloya dikkatle bakınca SUM sütununun XOR'un, CARRY sütununun AND'in tablosuyla aynı olduğu görülür. Demek ki iki biti toplamak için iki kapı yeterlidir.

Tanım 1.3.7

Yarım toplayıcı (*half adder*), iki tek bitlik girdiyi (A, B) toplayan devredir: *SUM = A XOR B* sonuç basamağını, *CARRY = A AND B* eldeyi verir.



Şekil 1.3.19: Yarım toplayıcı, 1 + 1 durumunda. A ve B her iki kapıya da gider. XOR(1, 1) = 0 sonuç basamağını, AND(1, 1) = 1 eldeyi verir: (CARRY, SUM) = 10_2 = 2.

Şekil 1.3.19, devrenin en ilginç durumunu gösterir. 1 + 1 için XOR 0, AND 1 verir. Sonuç (CARRY, SUM) sırasıyla 10_2 = 2'dir. Onluk sistemdeki 9 + 1 = 10 gibi, basamak taşar ve elde bir üst basamağa gider. Yarım toplayıcının adı da buradan gelir: gelen bir eldeyi alacak girişi yoktur, toplama işinin yalnız yarısını yapar.

1.3.6.3 Tam toplayıcı

Çok bitli sayılar toplanırken en sağdaki sütun dışında her sütuna, sağındaki sütundan bir elde gelebilir. Bu yüzden her sütunda üç bit toplanır: A, B ve gelen elde (Cin). Üç bitin toplamı en çok 3 = 11_2'dir; yine iki bitle yazılır.

Tanım 1.3.8

Tam toplayıcı (*full adder*), üç tek bitlik girdiyi (A, B, Cin) toplayıp sonuç basamağını (SUM) ve çıkan eldeyi (Cout) üreten devredir. İki yarım toplayıcı ve bir OR kapısıyla kurulur:

> t = A XOR B, SUM = t XOR Cin, Cout = (A AND B) OR (t AND Cin)

İlk yarım toplayıcı A ile B'yi toplar ve ara sonuç *t*'yi üretir. İkinci yarım toplayıcı *t*'ye gelen eldeyi ekler. Elde iki yarım toplayıcının birinden çıkabilir; ikisinden birden çıkamaz. OR kapısı bu iki olası eldeyi birleştirir.

Örnek 1.3.9

A = 1, B = 1, Cin = 1 için tam toplayıcının çıktılarını bulalım.

1. **İlk yarım toplayıcı:** t = 1 XOR 1 = 0; ilk elde 1 AND 1 = 1.
2. **İkinci yarım toplayıcı:** SUM = 0 XOR 1 = 1; ikinci elde 0 AND 1 = 0.
3. **OR:** Cout = 1 OR 0 = 1.
4. **Denetim:** (Cout, SUM) = 11_2 = 3 = 1 + 1 + 1.

1.3.6.4 Dört bitlik toplayıcı: 0110 + 0111

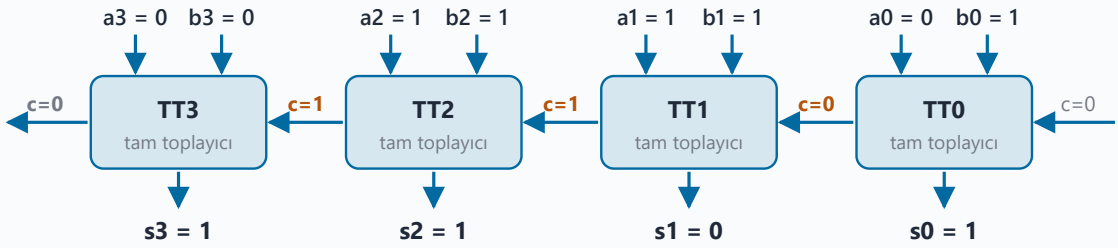
Dört tam toplayıcı yan yana konur ve her birinin elde çıkışı solundakinin elde girişine bağlanır. Böylece dört bitlik sayıları toplayan bir devre elde edilir. En sağdaki toplayıcının gelen eldesi 0'dır. Sunumdaki sözde kod, devrenin sağdan sola yaptığı işi anlatır:

```
a ← 0110  ; 6
b ← 0111  ; 7
c ← 0     ; ilk elde
i = 0, 1, 2, 3 için: ; sağdan sola
  t ← a[i] XOR b[i]
  s[i] ← t XOR c
  c ← (a[i] AND b[i]) OR (t AND c)
yaz(c, s)
```

i	a[i]	b[i]	gelen c	t	s[i]	çıkan c
0	0	1	0	1	1	0
1	1	1	0	0	0	1
2	1	1	1	0	1	1
3	0	0	1	0	1	0

Tablo 1.3.13: 0110 + 0111 (6 + 7) için tam toplayıcı zincirinin izlemesi. Her satırda `c` gelen elde, son sütun çıkan eldedir.

Tablo satır satır şöyle okunur. *i = 0*'da 0 + 1 + 0 toplanır: *t = 1*, $s_0 = 1$; iki giriş birden 1 olmadığı için elde 0'dır. *i = 1*'de 1 + 1 + 0 = 2 = 10_2'dir: basamak 0, elde 1. Bu, onluktaki 5 + 5 = 10'un ikili karşılığıdır. *i = 2*'de gelen elde de eklenir: 1 + 1 + 1 = 3 = 11_2, basamak 1, elde yine 1. *i = 3*'te 0 + 0 + 1 = 1'dir; basamak 1, elde 0. Son elde 0 olduğu için sonuç 4 bite sığmıştır.



Şekil 1.3.20: Dört tam toplayıcının (TT0–TT3) zinciri, 0110 + 0111 için. Elde sağdan sola ilerler: TT0'dan 0, TT1'den 1, TT2'den 1, TT3'ten 0 çıkar. Sonuç 1101_2 = 13 = 6 + 7.

Sonuç 1101_2 = 13 = 6 + 7'dir. Bu düzende elde, sağdan sola bir dalga gibi ilerler; devrenin adı da buradan gelir: **dalgalı elde toplayıcı** (*ripple-carry adder*). Her tam toplayıcı, sağındakinin eldesini beklemek zorundadır. Bit sayısı arttıkça bekleme süresi, yani gecikme de büyür: 64 bitlik bir dalgalı elde toplayıcıda en soldaki basamak, 63 toplayıcının sırayla bitirmesini bekler. Gerçek işlemciler bu yüzden eldeyi önceden hesaplayan daha karmaşık toplayıcılar kullanır; ama hepsi aynı kapılardan kurulur.

Bu devre, 1.3.2'deki kavramlarla doğrudan bağlantılıdır. Son tam toplayıcıdan çıkan elde, 1.3.2.9'daki elde bitidir. Örneğin 1011 + 0111 (11 + 7) toplanırsa her sütunda elde 1 çıkar; 4 bitte kalan 0010 = 2'dir ve son elde 1, sonucun 4 bite sığmadığını bildirir. İkiye tümleyen sayesinde aynı devre çıkarma da yapar: $a - b$ için b'nin bitleri NOT kapılarıyla çevrilir ve en sağdaki gelen elde 1 yapılır. Böylece "bitleri çevir, 1 ekle" kuralı donanımda tek bir devreyle uygulanmış olur.

> **Sık yapılan hata**
>
> Çok bitli toplamayı yarım toplayıcılarla kurmaya çalışmak. Yarım toplayıcının gelen elde girişi yoktur; yalnız en sağdaki sütunda kullanılabilir. Öteki sütunlarda üç bit toplandığı için tam toplayıcı gerekir.

1.3.7 Depolanan program ve CPU

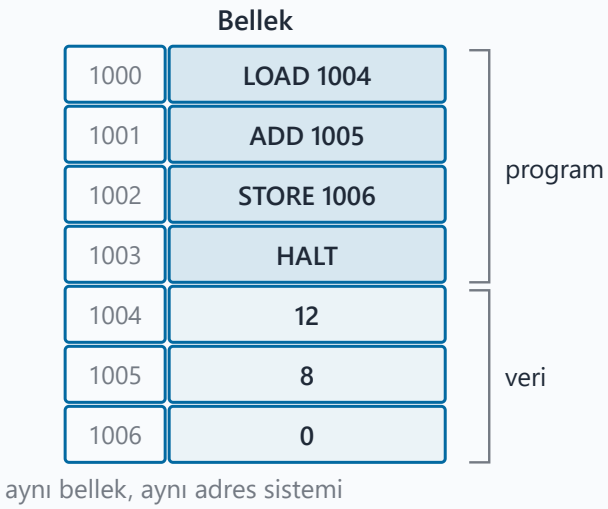
Bir toplayıcı yalnız toplar. Bilgisayarı genel amaçlı yapan şey, hangi işlemin hangi sırayla yapılacağını söyleyen bir programın varlığıdır. Bu kesim, programın bellekte nasıl durduğunu ve işlemcinin onu nasıl adım adım yürüttüğünü anlatır.

1.3.7.1 Depolanan program kavramı

İlk elektronik hesap makinelerinden bazıları, yeni bir hesap için kabloları elle yeniden bağlamayı gerektiriyordu. Bir programı değiştirmek günler sürebiliyordu. 1940'ların ortasında ortaya konan bir fikir bu durumu kökten değiştirdi: komutlar da birer sayı olarak verilerle aynı belleğe konabilirdi.

Tanım 1.3.9

Depolanan program (*stored program*) yaklaşımında program komutları ve işlenecek veriler aynı bellekte sayısal kodlar olarak saklanır. İşlemci komutları bellekten sırayla okur ve yürütür.



Şekil 1.3.21: Aynı bellekte program ve veri. 1000–1003 adreslerinde komutlar (LOAD 1004, ADD 1005, STORE 1006, HALT), 1004–1006 adreslerinde veriler (12, 8, 0) durur. İkisi aynı adres sistemini kullanır.

Bu yaklaşımın üç önemli sonucu vardır.

- **Esneklik:** Aynı donanım farklı programlarla farklı görevler yapar. Bir bilgisayarda hesap tablosu, oyun ve tarayıcı aynı işlemcide çalışır.
- **Güncelleme:** Davranış, donanımı değiştirmeden yazılımla değiştirilebilir. Bir telefonun güncellenmesi, belleğe yeni komutların yazılmasıdır.
- **Genel amaçlılık:** Tek bir sistem hesaplama, iletişim ve medya görevlerini birlikte yürütebilir. Hatta bir program, başka bir programı veri olarak okuyabilir, üretebilir ya da çevirebilir; derleyiciler tam olarak bunu yapar.

1.3.7.2 Getir–çöz–yürüt döngüsü

İşlemci, bellekteki programı hiç durmadan tekrarlanan üç aşamalı bir döngüyle yürütür. Bu döngüyü anlatmadan önce iki yazmacı tanımak gerekir.

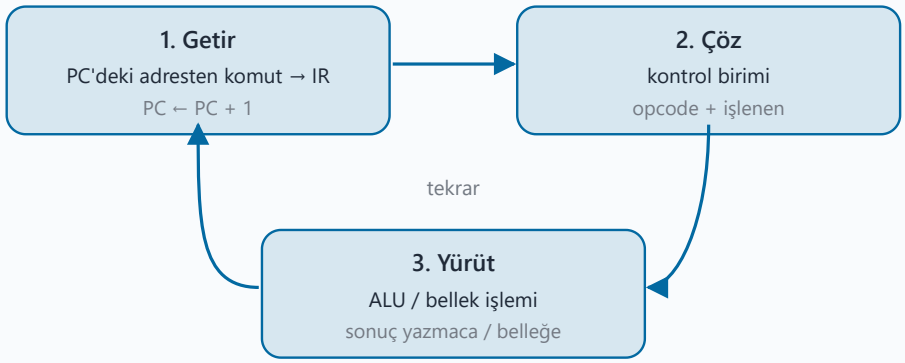
- **Program sayacı** (PC, *program counter*): sıradaki komutun bellekteki adresini tutar.
- **Komut yazmacı** (IR, *instruction register*): bellekten getirilen ve o an yürütülen komutu tutar.

Tanım 1.3.10

Getir–çöz–yürüt döngüsü (*fetch-decode-execute cycle*) üç aşamadan oluşur:

1. **Getir:** PC'deki adresten komut okunur ve IR'ye konur; PC bir artırılır.
2. **Çöz:** Kontrol birimi IR'deki komutu ayırır: işlem kodu (*opcode*) ne yapılacağını, işlenen (*operand*) neyle yapılacağını söyler.
3. **Yürüt:** ALU ya da başka bir birim işlemi yapar; sonuç bir yazmaca ya da belleğe yazılır.

Ardından döngü yeniden getir aşamasıyla başlar.

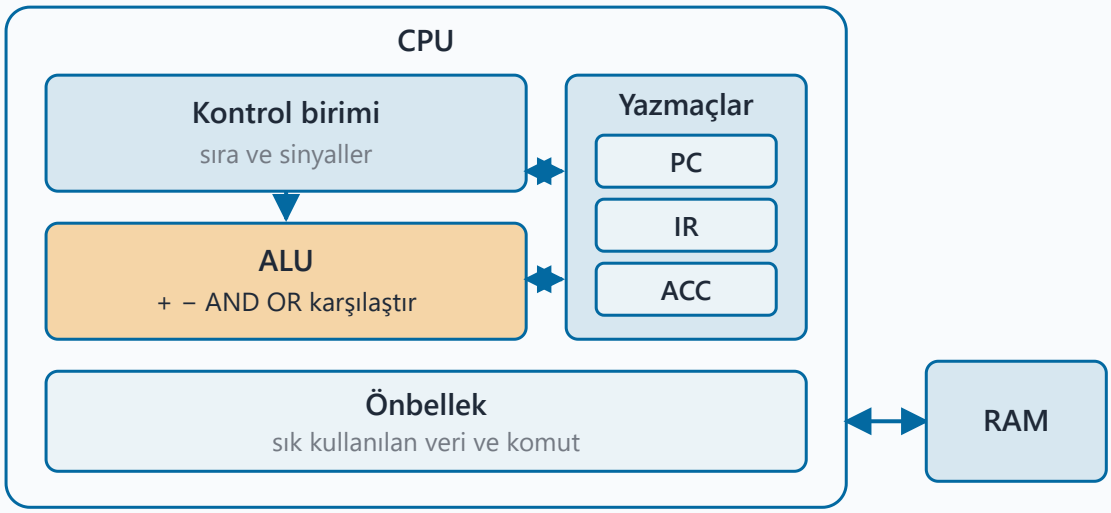


Şekil 1.3.22: Getir, çöz ve yürüt aşamaları bir döngü oluşturur. Getirde PC'deki adresten komut IR'ye gelir ve PC bir artar. Çözde kontrol birimi opcode ile işleneni ayırır. Yürütte ALU ya da bellek işlemi yapılır ve sonuç yazmaca ya da belleğe yazılır.

PC'nin getir aşamasında, yani komut daha yürütülmeden artırılmasının bir nedeni vardır. Komutların büyük çoğunluğu bir sonraki komuta geçmeyi gerektirir; PC'yi önceden artırmak bunu otomatik yapar. Dallanma komutları ise yürütme aşamasında PC'nin üzerine yeni bir adres yazarak bu sırayı değiştirir (1.3.7.8).

Döngü anlatılırken bellekten okuma diye söz edilse de komut ve veri çoğu zaman RAM'den değil, işlemcinin içindeki **önbellekten** (*cache*) gelir. Yazmaçlar ve önbellek, RAM'den çok daha hızlıdır. Bu ayrıntı döngünün mantığını değiştirmez, yalnız hızını artırır.

1.3.7.3 CPU'nun temel bölümleri



Şekil 1.3.23: CPU'nun içi. Kontrol birimi sırayı ve sinyalleri yönetir; ALU toplama, çıkarma, AND, OR ve karşılaştırma yapar; yazmaçlar (PC, IR, ACC) ara değerleri tutar; önbellek sık kullanılan veri ve komutları RAM'den getirip yakında saklar.

Şekil 1.3.23, bir işlemcinin dört ana bölümünü gösterir.

- **Kontrol birimi** (*control unit*): komutların sırasını ve veri hareketini yönetir. Çözme aşamasını yapar ve öteki birimlere ne zaman ne yapacaklarını bildiren sinyaller gönderir.
- **Aritmetik-mantık birimi** (ALU, *arithmetic logic unit*): aritmetik ve mantıksal işlemleri yapar. 1.3.6.4'teki toplayıcı ALU'nun bir parçasıdır.
- **Yazmaçlar:** işlem sırasında kullanılan küçük ve hızlı saklama alanlarıdır. PC ve IR'nin yanında, bu nottaki örnek işlemcide hesap sonuçlarının biriktiği bir **akümülatör** (ACC, *accumulator*) vardır.
- **Önbellek:** sık kullanılan veri ve komutları işlemciye yakın tutar.

1.3.7.4 Veri yolları: bellekten okuma

CPU ile bellek ve G/Ç birimleri, **yol** (*bus*) adı verilen paralel tel demetleriyle bağlıdır. Üç ayrı yol vardır ve her birinin görevi farklıdır.

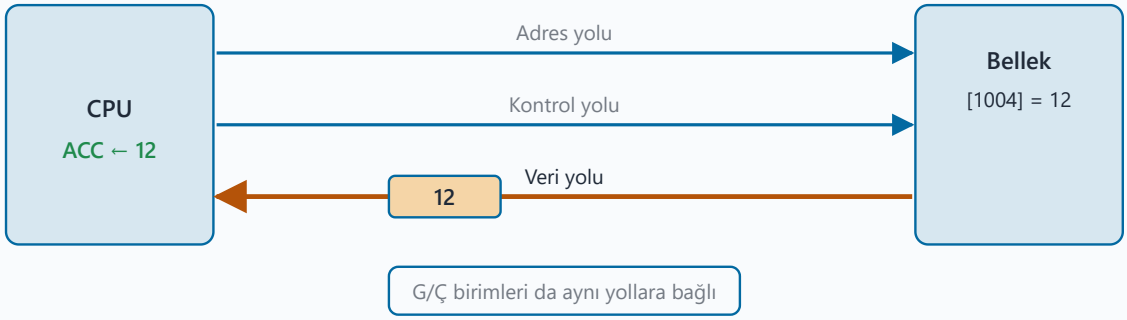
Yol	Görev	Yön
Adres yolu	İşlem yapılacak bellek ya da aygıt konumunu belirtir.	tek yönlü: CPU'dan çıkar
Kontrol yolu	Okuma, yazma ve zamanlama sinyallerini taşır.	çoğunlukla CPU'dan çıkar
Veri yolu	CPU, bellek ve aygıtlar arasında veriyi taşır.	çift yönlü

Tablo 1.3.14: CPU'yu bellek ve aygıtlara bağlayan üç yol.

Örnek 1.3.10

CPU, 1004 adresindeki değeri okuyacaktır; hücrede 12 vardır. Üç yol sırayla şöyle kullanılır:

1. **Adres yolu:** CPU hangi hücreyi istediğini gönderir: 1004.
2. **Kontrol yolu:** CPU OKU sinyalini gönderir. Yazma olsaydı YAZ sinyali giderdi.
3. **Veri yolu:** Bellek hücrenin içeriğini, yani 12'yi, CPU'ya gönderir. Değer ACC'ye yüklenir.



Şekil 1.3.24: Bellekten okumanın son adımı. Adres yolundan 1004 ve kontrol yolundan OKU sinyali gitmiştir; bellek, hücrenin içeriği olan 12'yi veri yoluyla CPU'ya gönderir.

Yazma işleminde adres ve kontrol yolları yine CPU'dan belleğe doğru kullanılır, ama veri bu kez CPU'dan belleğe akar. Veri yolunun çift yönlü olması bundandır. Adres yolunun genişliği, gösterilebilecek adres sayısını belirler: 16 bitlik bir adres yolu 2^{16} = 65.536 farklı adres gösterebilir.

Sık yapılan hata

Adresin de veri yoluyla gittiğini sanmak. “1004” sayısı belleğe adres yoluyla gider; veri yolu yalnız hücrenin içeriğini (12) taşır. Kontrol yolu ise ne yapılacağını (OKU ya da YAZ) bildirir.

1.3.7.5 Bellek adresleri

Bellek, numaralanmış hücrelerden oluşan uzun bir dizi gibi düşünülebilir. Her hücrenin bir **adresi** vardır ve komutlar veriye bu adresle ulaşır. Bu notta kullanılan küçük örnek bellek Tablo 1.3.15'te görülür.

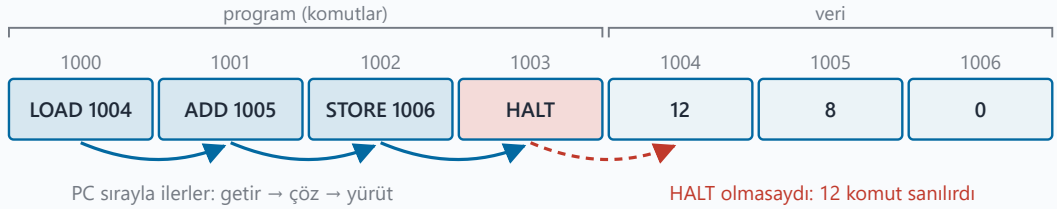
Adres	İçerik	Anlamı
1000	`LOAD 1004`	ACC ← bellek[1004]
1001	`ADD 1005`	ACC ← ACC + bellek[1005]
1002	`STORE 1006`	bellek[1006] ← ACC
1003	`HALT`	dur

Adres	İçerik	Anlamı
1004	12	veri
1005	8	veri
1006	0	sonuç yeri

Tablo 1.3.15: Örnek bellek içeriği. İlk dört hücre komut, son üç hücre veridir.

Tablodaki “bellek[1004]” yazımı, “1004 adresindeki hücrenin içeriği” demektir. Adres ile içerik farklı şeylerdir: 1004 adresinde 12 vardır. Bir ev adresi ile o evde oturan kişi arasındaki fark gibidir.

Bellekte komutlar ile veriler arasında fiziksel bir ayrım yoktur. Komutlar da bellekte birer sayı olarak durur; aynı bit dizisi komut ya da veri olarak yorumlanabilir. Bir hücrenin hangisi olduğunu, PC'nin o hücreye gelip gelmediği belirler. HALT komutu bu yüzden gereklidir. HALT olmasaydı CPU 1003'ten sonra 1004'e geçer ve oradaki 12'yi de bir komut sanıp çalıştırmaya çalışırdı.



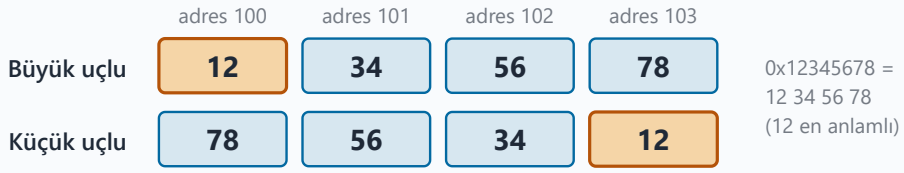
Şekil 1.3.25: PC 1000'den başlayıp 1001, 1002 ve 1003'teki komutları sırayla getirir. 1003'teki HALT CPU'yu durdurur; HALT olmasaydı 1004'teki 12 komut sanılırdı.

1.3.7.6 Çok baytlı değerler ve bayt sırası

Gerçek bilgisayarlarda bellek **bayt adreslidir**: her adres 1 bayt tutar. 32 bitlik bir `int` ise 4 bayttır ve 4 ardışık adrese yayılır. Bu durumda bir soru doğar: dört baytın hangisi en küçük adrese yazılacak? Bu sıraya **bayt sırası** (*endianness*) denir.

0x12345678 değerinde en anlamlı bayt 12, en az anlamlı bayt 78'dir; tıpkı 1234 sayısında 1'in binler, 4'ün birler basamağında olması gibi. İki yaygın düzen vardır.

- **Büyük uçlu** (*big-endian*): en anlamlı bayt en küçük adrese yazılır. Adres 100'den 103'e 12 34 56 78 okunur; sayı kâğıtta yazıldığı sırayla dizilir. Ağ protokolleri bu sırayı kullanır; bu yüzden ona **ağ bayt sırası** da denir.
- **Küçük uçlu** (*little-endian*): en az anlamlı bayt en küçük adrese yazılır. Adres 100'den 103'e 78 56 34 12 okunur. x86-64 ve yaygın ARM sistemleri bu sırayla çalışır.



Şekil 1.3.26: 0x12345678'in 100–103 adreslerine yerleşimi. Büyük uçlu düzende en anlamlı bayt (12) 100 adresinde, küçük uçlu düzende 103 adresindedir.

İki düzen de doğru çalışır; sorun yalnız iki sistem arasında ham ikili veri taşındığında çıkar. Küçük uçlu bir makinenin yazdığı 78 56 34 12 baytları, büyük uçlu bir makinede tek seferde okunursa 0x78563412 olarak görülür. Dosya biçimleri ve ağ protokolleri bu yüzden bayt sırasını açıkça belirtir. Java'nın `ByteBuffer` sınıfıyla sıra seçilebilir:

```
java.nio.ByteBuffer bb = java.nio.ByteBuffer.allocate(4);
bb.order(java.nio.ByteOrder.LITTLE_ENDIAN);
bb.putInt(0x12345678);
for (byte x : bb.array()) {
    System.out.printf("%02X ", x);
}
System.out.println();
```

Çıktı:

```
78 56 34 12
```

Tek baytlık değerlerde, örneğin bir ASCII karakterinde, sıra sorunu yoktur. UTF-8 de bayt bayt okunduğu için bayt sırasından etkilenmez; “ğ” her makinede C4 9F sırasıyla yazılır.

1.3.7.7 Programı adım adım çalıştırma

Şimdiye kadar tanıtılan parçalar, Tablo 1.3.15'teki küçük programın yürütülmesinde bir araya gelir. Program iki sayıyı toplar ve sonucu belleğe yazar. Her satırın sağında, noktalı virgülden sonra komutun anlamı yazılıdır:

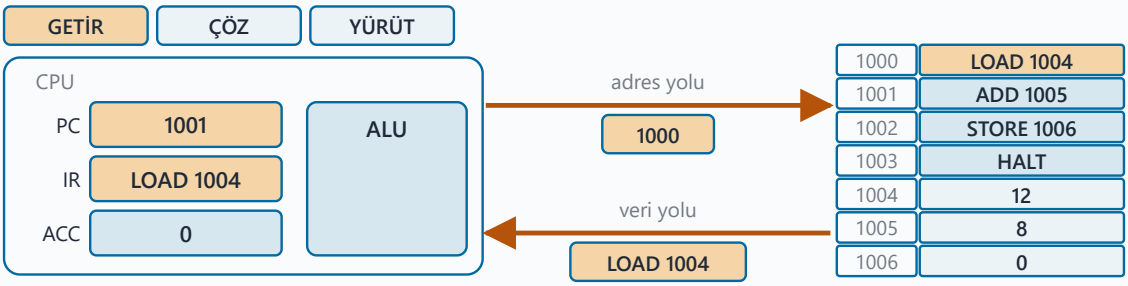
```
1000: LOAD  1004 ; ACC ← bellek[1004]
1001: ADD   1005 ; ACC ← ACC + bellek[1005]
1002: STORE 1006 ; bellek[1006] ← ACC
1003: HALT       ; dur
1004: 12         ; veri
1005: 8          ; veri
1006: 0          ; sonuç
```

Bu örnek işlemcide hesaplar tek bir yazmaçta, ACC'de yapılır. Komutların anlamları şöyledir: `LOAD a` bellekteki *a* adresinin içeriğini ACC'ye yükler; `ADD a` o içeriği ACC'ye ekler; `STORE a` ACC'yi *a* adresine yazar; `HALT` işlemciyi durdurur. Başlangıçta PC = 1000, yani ilk komutun adresi, ve ACC = 0'dır.

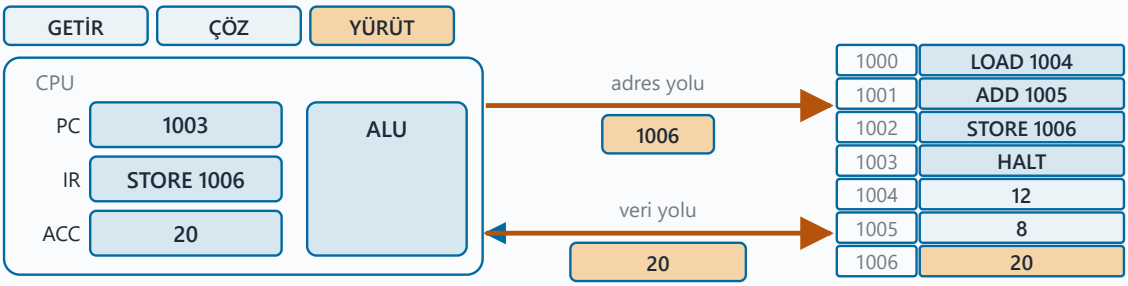
Adım	Aşama	Ne olur?	PC	IR	ACC	bellek[1006]
0	başlangıç	—	1000	—	0	0
1	getir	1000 adres yoluna çıkar, LOAD 1004 veri yoluyla IR'ye gelir	1001	LOAD 1004	0	0
2	çöz	opcode LOAD, adres 1004	1001	LOAD 1004	0	0
3	yürüt	1004 okunur, 12 ACC'ye yüklenir	1001	LOAD 1004	12	0
4	getir	ADD 1005 IR'ye gelir	1002	ADD 1005	12	0
5	çöz	opcode ADD, adres 1005	1002	ADD 1005	12	0
6	yürüt	1005'teki 8 okunur, ALU 12 + 8 = 20 hesaplar	1002	ADD 1005	20	0
7	getir	STORE 1006 IR'ye gelir	1003	STORE 1006	20	0
8	çöz	opcode STORE, adres 1006	1003	STORE 1006	20	0
9	yürüt	ACC'deki 20 veri yoluyla 1006'ya yazılır	1003	STORE 1006	20	20
10	getir	HALT IR'ye gelir	1004	HALT	20	20
11	çöz	opcode HALT → CPU durur	1004	HALT	20	20

Tablo 1.3.16: Programın getir–çöz–yürüt izlemesi. Her satır, adım bittikten sonraki yazmaç ve bellek durumunu gösterir.

İzlemede birkaç düzenli örüntü göze çarpar. PC yalnız getir aşamalarında değişir ve her seferinde bir artar. IR, her getirde yeni komutu alır ve o komut bitene kadar değişmez. ACC yalnız LOAD ve ADD'in yürütme aşamalarında değişir; bellek ise yalnız STORE'un yürütmesinde değişir. Program bittiğinde bellek[1006] = 20'dir. Dört komut, dört getir–çöz–yürüt turuyla yürütülmüştür; HALT'ın yürütülecek bir işi olmadığı için onun turu çözme aşamasında biter.



Şekil 1.3.27: İlk getir aşaması (adım 1). PC'deki 1000 adres yoluna çıkar, LOAD 1004 komutu veri yoluyla IR'ye gelir ve PC 1001'e artar.



Şekil 1.3.28: STORE'un yürütme aşaması (adım 9). ACC'deki 20 veri yoluyla 1006 adresine yazılır.

Örnek 1.3.11

Programda STORE ile ADD'in yeri değiştirilirse (1001: STORE 1006, 1002: ADD 1005) sonuç ne olur?

1. **LOAD 1004:** ACC = 12.
2. **STORE 1006:** o anki ACC yazılır; bellek[1006] = 12.
3. **ADD 1005:** ACC = 12 + 8 = 20; ama bu değer belleğe hiç yazılmaz.
4. **HALT:** durulur. Sonuç: ACC = 20, bellek[1006] = 12; yine 4 komut getirilmiştir.

CPU komutları yazıldıkları sırayla yürütür; sıra değişince sonuç değişir.

1.3.7.8 Dallanma: döngüde PC nasıl değişir?

Önceki program komutları baştan sona bir kez yürüttü. Gerçek programlar ise karar verir ve işleri tekrarlar. Bunun için **dallanma** (*branch*) komutları kullanılır. Dallanma komutu, yürütme aşamasında PC'nin üzerine yeni bir adres yazar; böylece bir sonraki getir başka bir yerden yapılır. Örnek işlemcide iki dallanma komutu vardır:

- `JUMP a` : koşulsuz dallanma; PC ← *a*.

- `JUMPZ a`: koşullu dallanma; ACC = 0 ise PC ← *a*, değilse PC olduğu gibi kalır ve sıradaki komuta geçilir.

Aşağıdaki program, `sayaç` değişkenini 2'den geriye sayarak `toplam`'a 2 + 1 ekler. 1010, 1011 ve 1012 adresleri sırasıyla `sayaç`, `toplam` ve sabit 1'i tutar:

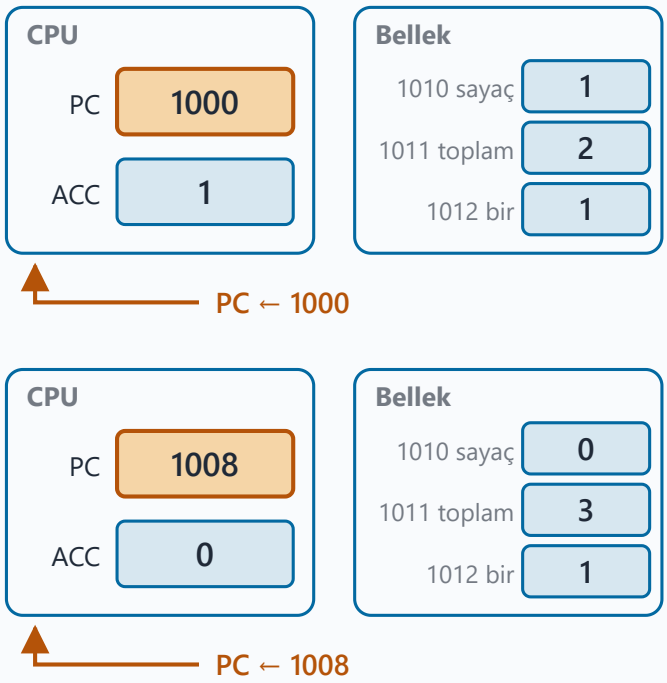
```
1000: LOAD  1010  ; ACC ← sayaç
1001: JUMPZ 1008  ; ACC = 0 ise atla
1002: ADD   1011  ; ACC ← ACC + toplam
1003: STORE 1011  ; toplam ← ACC
1004: LOAD  1010
1005: SUB   1012  ; ACC ← ACC − 1
1006: STORE 1010  ; sayaç ← ACC
1007: JUMP  1000  ; PC ← 1000
1008: HALT
1010: 2   1011: 0   1012: 1
```

Program Java'daki şu döngünün makine düzeyindeki karşılığıdır: `while (sayac != 0) { toplam = toplam + sayac; sayac = sayac - 1; }`. 1000–1001 satırları döngü koşulunu, 1002–1006 satırları döngü gövdesini, 1007 satırı başa dönüşü yapar.

Komut no	Yürütülen	PC	ACC	sayaç	toplam	Açıklama
—	başlangıç	1000	0	2	0	
1	1000: LOAD 1010	1001	2	2	0	ACC ← sayaç
2	1001: JUMPZ 1008	1002	2	2	0	ACC ≠ 0, atlama yok
3–4	ADD 1011, STORE 1011	1004	2	2	2	toplam = 2 + 0
5–7	LOAD, SUB, STORE 1010	1007	1	1	2	sayaç = 2 − 1
8	1007: JUMP 1000	1000	1	1	2	PC'nin 1008 değeri 1000 ile değişir
9–16	ikinci tur (aynı 8 komut)	1000	0	0	3	toplam = 1 + 2, sayaç = 0
17	1000: LOAD 1010	1001	0	0	3	ACC ← 0
18	1001: JUMPZ 1008	1008	0	0	3	ACC = 0, atlanır
19	1008: HALT	1009	0	0	3	CPU durur

Tablo 1.3.17: Dallanma programının izlemesi. Her satır, komut yürütüldükten sonraki durumu gösterir. PC sütunu, bir sonraki getirde kullanılacak adrestir.

İzlemenin kritik anları iki dallanma komutudur. 8. komutta `JUMP 1000` yürütülür. Getir aşamasında PC zaten 1008'e artmıştı; yürütme bu değerin üzerine 1000 yazar ve döngü başa döner. 18. komutta ise `JUMPZ` bu kez atlar, çünkü ACC'ye yüklenen sayaç 0'dır. PC 1008 olur ve döngü gövdesi atlanır. Toplam 19 komut getirilmiştir: iki tam tur için 8 + 8 ve çıkış için 3 (LOAD, JUMPZ, HALT).



Şekil 1.3.29: Dallanmanın iki anı. (a) İlk turun sonunda `JUMP 1000` : PC 1000 olur, sayaç 1, toplam 2'dir. (b) Son turda `JUMPZ` atlar: ACC = 0 olduğu için PC 1008'e geçer.

Komut sayısının genel formülü de izlemeden çıkarılabilir. Sayaç k'dan başlarsa döngü k tur döner ve her tur 8 komuttur; çıkışta 3 komut daha yürütülür. Toplam $8k + 3$ komuttur: $k = 2$ için 19, $k = 5$ için 43. `toplam` değişkeninin son değeri ise $k + (k - 1) + \ldots + 1$ olur; $k = 5$ için 15. Bu sayım, bir programın süresinin girdinin büyüklüğüyle nasıl arttığını gösteren ilk örnektir.

Sık yapılan hata

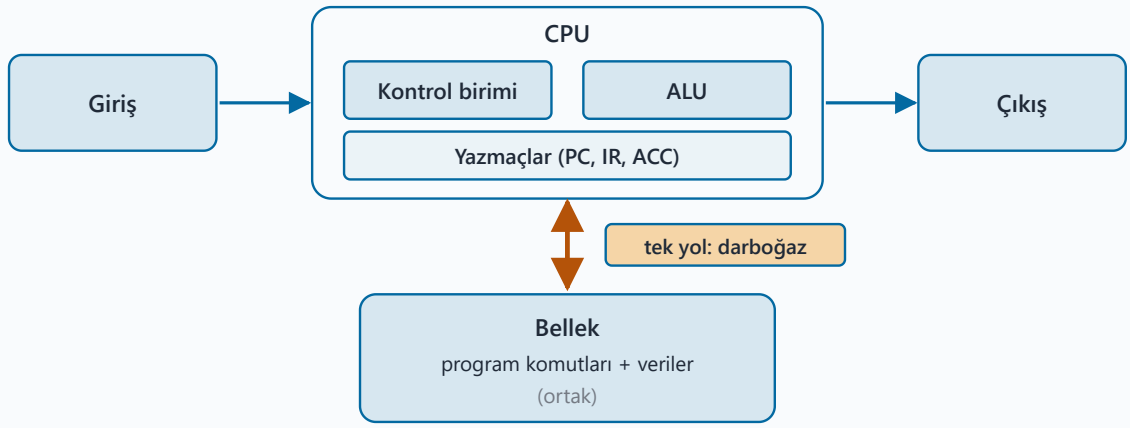
Dallanma komutunun PC'yi getir aşamasında değiştirdiğini sanmak. Getirde PC her komutta olduğu gibi bir artar (1007 → 1008). Atlama, yürütme aşamasında bu yeni değerin üzerine yazılarak yapılır. Koşullu dallanmada koşul tutmazsa PC'ye dokunulmaz ve artırılmış değer geçerli kalır.

1.3.8 Bilgisayar mimarisi

Bir bilgisayarın parçalarının nasıl düzenlendiğine ve programcının gördüğü komutların nasıl tasarlandığına **mimari** (*architecture*) denir. Bu kesim, bugünkü bilgisayarların temelindeki düzeni, onun bir alternatifini, komutların bitlere nasıl yazıldığını ve performansı belirleyen etkenleri ele alır.

1.3.8.1 von Neumann mimarisi

Bugünkü genel amaçlı bilgisayarlar, depolanan program fikrini kullanan bir düzene dayanır. Bu düzen, John von Neumann'ın 1945'te yazdığı bir rapordan sonra **von Neumann mimarisi** (*von Neumann architecture*) adıyla anılır.



Şekil 1.3.30: von Neumann mimarisi. CPU (kontrol birimi, ALU, yazmaçlar) ile program komutlarının ve verilerin birlikte tutulduğu ortak bellek tek bir yolla bağlıdır. Giriş ve çıkış birimleri CPU'ya bağlanır.

Mimarinin üç belirleyici özelliği vardır.

- Program komutları ve veriler **ortak bellekte** tutulur.
- CPU komutları **sırayla** işler; dallanma komutları bu sırayı değiştirebilir.
- Komut ve veri **aynı yoldan** gelir.

Son özellik bir bedel doğurur. CPU bir komutu getirirken aynı yoldan veri okuyamaz; ikisi sırayla taşınır. Üstelik işlemciler yıllar içinde bellekten çok daha hızlı gelişmiştir. Sonuçta CPU–bellek aktarımı performansı sınırlar. Bu duruma **von Neumann darboğazı** (*von Neumann bottleneck*) denir. Önbellek, bu darboğazı hafifletmek için vardır.

1.3.8.2 von Neumann ve Harvard

Darboğaza karşı farklı bir yol, komutları ve verileri ayrı belleklerde tutmaktır. Bu düzene **Harvard mimarisi** (*Harvard architecture*) denir; adı, bu düzenle kurulan Harvard Mark I bilgisayarından gelir.



Şekil 1.3.31: Solda von Neumann: tek bellek ve tek yol. Sağda Harvard: ayrı komut ve veri bellekleri, ayrı yollar. CPU komut belleğinden yalnız okur; veri belleğinden okur ve ona yazar.

Özellik	von Neumann	Harvard
Bellek	Komut ve veri ortak bellekte	Komut ve veri ayrı belleklerde
Yol	Tek yol: komut ve veri sırayla taşınır	İki yol: komut ve veri aynı anda okunur
Esneklik	Program, veri gibi yüklenip değiştirilebilir	Program belleği çoğunlukla sabittir (flash)
Yaygın kullanım	Genel amaçlı bilgisayarlar	Mikrodenetleyiciler, sinyal işlemciler

Tablo 1.3.18: von Neumann ve Harvard mimarilerinin karşılaştırması.

Harvard düzeninin üstünlüğü hızdır: bir komut getirilirken önceki komutun verisi aynı anda okunabilir. Bedeli ise esnekliktir. Bir mikrodenetleyicide program, flash belleğe bir kez yazılır ve cihaz çalışırken değişmez; bir çamaşır makinesi için bu yeterlidir. Genel amaçlı bir bilgisayarın ise her gün yeni programlar yüklemesi gerekir.

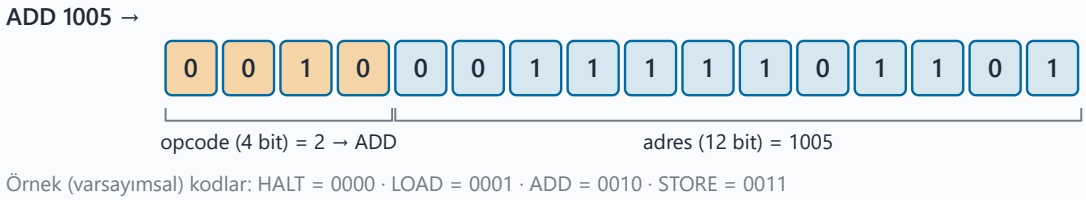
Günümüz işlemcileri iki düzeni birleştirir. Ana bellek ortaktır; programlar ve veriler aynı RAM'e yüklenir. Çekirdeğe en yakın birinci düzey (L1) önbellek ise komut ve veri için ayrıdır. Bu karma düzene **değiştirilmiş Harvard** mimarisi denir. Programcı ortak belleği görür, işlemci ise içeride iki ayrı yoldan yararlanır.

1.3.8.3 Komutların kodlanması

Depolanan program fikrinin bir gereği olarak komutlar da bitlerle yazılır. Bir makine komutu genellikle iki bilgi taşır: yapılacak işlem ve kullanılacak değer ya da adres.

- **Opcode** (işlem kodu): işlemin türünü tanımlar; örneğin ADD.
- **İşlenen** (*operand*): işlemde kullanılacak yazmaç, değer ya da adres; örneğin 1005 ya da R1, R2.
- **Komut biçimi** (*instruction format*): bit alanlarının sayısını ve yerini belirler. Her mimarinin kendi biçimi vardır.
- **Komut kümesi** (*instruction set*): işlemcinin doğrudan anlayabildiği komutların tamamıdır.

Bu notun örnek işlemcisi için varsayımsal bir biçim kullanılır: komut 16 bittir, ilk 4 bit opcode, kalan 12 bit adrestir. Opcode'lar HALT = 0000, LOAD = 0001, ADD = 0010, STORE = 0011'dir.



Şekil 1.3.32: 16 bitlik komut sözcüğü. ADD 1005 için ilk 4 bit opcode 0010 (ADD), kalan 12 bit adres 1005 = 001111101101'dir.

Örnek 1.3.12

Tablo 1.3.15'teki programın dört komutunu bu biçimle kodlayalım.

1. **Adresler:** 1004 = $3EC_{16}$ = 0011 1110 1100; 1005 = $3ED_{16}$ = 0011 1110 1101; 1006 = $3EE_{16}$ = 0011 1110 1110. (Denetim: $3 \cdot 256 + 14 \cdot 16 + 12 = 1004$.)
2. **LOAD 1004:** 0001 0011 1110 1100 = $13EC_{16}$.
3. **ADD 1005:** 0010 0011 1110 1101 = $23ED_{16}$.
4. **STORE 1006:** 0011 0011 1110 1110 = $33EE_{16}$.
5. **HALT:** 0000 0000 0000 0000 = 0000_{16}; adres alanı kullanılmaz.

12 bitlik adres alanıyla 2^{12} = 4.096 farklı adres (0–4095) gösterilebilir. Daha büyük bir bellek için adres alanı genişletilmelidir.

Bu kodlama, 1.3.7.5'teki gözlemi somutlaştırır. 1004 adresindeki 12 sayısı 16 bitle 0000 0000 0000 1100 olarak yazılır. Bu dizinin ilk dört biti 0000'dır, yani HALT'ın opcode'u. Aynı bit dizisi veri olarak okunursa 12, komut olarak okunursa bir HALT komutudur.

1.3.8.4 Komut kümesi tasarımı: RISC ve CISC

Komut kümesinin nasıl tasarlanacağı konusunda iki ana yaklaşım vardır.

Özellik	RISC	CISC
Komut kümesi	Az sayıda, basit komut	Çok sayıda, karmaşık komut
Komut uzunluğu	Sabit (ör. 32 bit)	Değişken (x86-64'te 1–15 bayt)
Bellek erişimi	Yalnız LOAD / STORE komutları	İşlem komutları doğrudan bellek işleneni alabilir
Örnek	ARM, RISC-V	x86-64

Tablo 1.3.19: RISC ve CISC yaklaşımlarının karşılaştırması.

RISC (*reduced instruction set computer*, azaltılmış komut kümeli bilgisayar) yaklaşımında komutlar basit ve eşit uzunluktadır. Belleğe yalnız LOAD ve STORE erişir; hesaplar yalnız yazmaçlar üzerinde yapılır. **CISC** (*complex instruction set computer*, karmaşık komut kümeli bilgisayar) yaklaşımında ise

tek bir komut bellekten okuyup hesaplayıp belleğe yazabilir. Fark, bellekteki `a = a + b` işlemi üzerinde görülür.

RISC:

```
LOAD  R1, a
LOAD  R2, b
ADD   R1, R1, R2
STORE R1, a
```

CISC (x86-64):

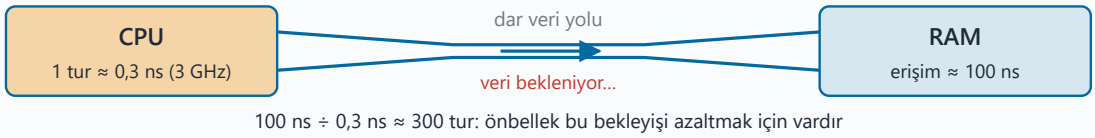
```
MOV EAX, [b]
ADD [a], EAX
```

RISC'te dört komut gerekir: iki okuma, bir toplama, bir yazma. x86-64'te iş iki komutla biter: ilk komut *b*'yi EAX yazmacına alır, ikincisi doğrudan bellekteki *a*'ya ekler. Köşeli parantez "bu adresteki bellek içeriği" demektir. Ama komut sayısının azalması işin azaldığı anlamına gelmez. İkinci komut, işlemcinin içinde yine okuma, toplama ve yazma adımlarına bölünür. Bugün x86-64 işlemciler de karmaşık komutları içeride RISC benzeri küçük adımlara çevirerek yürütür. Telefonların büyük çoğunluğu ARM, kişisel bilgisayarların çoğu x86-64 kullanır; son yıllarda ARM tabanlı dizüstü bilgisayarlar da yaygınlaşmıştır.

1.3.8.5 Mimari sınırlar

Bilgisayar ne kadar hızlı olursa olsun bazı sınırlarla çalışır. Bu sınırların çoğu, bu notta ayrı ayrı karşılaşılan olguların genel adlarıdır.

- **Sonlu bellek:** Aynı anda saklanabilecek veri miktarı sınırlıdır.
- **Sonlu hız:** Her komut zaman ve enerji tüketir.
- **Temsil hataları:** Sınırlı bit sayısı taşma (1.3.2.8) ve yuvarlama (1.3.3.3) üretebilir.
- **Veri darboğazı:** CPU, bellekteki veriyi beklerken boş kalabilir.
- **Paralellik sınırı:** Her problem eşit ölçüde paralelleştirilemez. Bir işin bir adımı öncekinin sonucuna bağlıysa, dalgalı elde toplayıcıdaki gibi, adımlar aynı anda yapılamaz.
- **Güvenlik:** Program ve veri aynı sistem kaynaklarını paylaştığı için programların birbirinden yalıtılması gerekir. Depolanan program fikrinin gücü, yani verinin komut gibi çalıştırılabilmesi, kötü amaçlı yazılımların da kapısıdır.



Şekil 1.3.33: Veri darboğazı. 3 GHz'lik bir CPU'da bir saat turu yaklaşık 0,3 ns sürer; RAM erişimi ise yaklaşık 100 ns'dir. CPU bir veriyi beklerken yaklaşık 300 tur boşa geçebilir.

Şekil 1.3.33'teki sayılar darboğazın boyutunu gösterir. 3 GHz'lik bir işlemci saniyede $3 \cdot 10^9$ tur atar; bir tur $1 / (3 \cdot 10^9)$ s ≈ 0,33 ns sürer. 100 ns'lik bir RAM erişimi bu yüzden yaklaşık $100 \div 0{,}33 \approx 300$ tura denk gelir. Önbellek olmasaydı işlemci zamanının büyük bölümünü bekleyerek geçirirdi.

1.3.8.6 Performans: komut sayısı, CPI ve saat hızı

Bir işlemcinin ne kadar hızlı olduğu sorusu tek bir sayıyla cevaplanamaz. Bir programın CPU'da geçirdiği süre üç etkenin çarpımıdır:

> CPU süresi = komut sayısı × CPI ÷ saat hızı

- **Komut sayısı:** programın yürüttüğü makine komutlarının sayısı. Algoritmaya, derleyiciye ve komut kümesine bağlıdır.
- **CPI** (*cycles per instruction*, komut başına saat turu): bir komutun ortalama kaç saat turu sürdüğü. İşlemcinin iç tasarımına ve bellek beklemelerine (önbellek ıskaları) bağlıdır.
- **Saat hızı:** işlemcinin saniyede attığı tur sayısı; 3 GHz, saniyede $3 \cdot 10^9$ tur demektir. Üretim teknolojisine, güç ve ısı sınırlarına bağlıdır.

Durum	Saat hızı	CPI	Süre
İşlemci A	3 GHz	1,5	$2 \cdot 10^9 \times 1{,}5 \div (3 \cdot 10^9) = 1{,}0$ s
İşlemci B	4 GHz	2,5	$2 \cdot 10^9 \times 2{,}5 \div (4 \cdot 10^9) = 1{,}25$ s
A, daha iyi derleyiciyle ($1{,}6 \cdot 10^9$ komut)	3 GHz	1,5	$1{,}6 \cdot 10^9 \times 1{,}5 \div (3 \cdot 10^9) = 0{,}8$ s

Tablo 1.3.20: $2 \cdot 10^9$ komutluk bir programın üç durumdaki CPU süresi.

Örnek 1.3.13

A ve B işlemcilerinin süresini adım adım hesaplayalım.

1. **Toplam tur (A):** $2 \cdot 10^9$ komut × 1,5 tur = $3 \cdot 10^9$ tur.
2. **Süre (A):** $3 \cdot 10^9$ tur ÷ ($3 \cdot 10^9$ tur/s) = 1,0 s.
3. **Toplam tur (B):** $2 \cdot 10^9 \times 2{,}5 = 5 \cdot 10^9$ tur.
4. **Süre (B):** $5 \cdot 10^9 \div (4 \cdot 10^9) = 1{,}25$ s.

5. **Karşılaştırma:** 1,25 ÷ 1,0 = 1,25. B'nin saat hızı daha yüksek olduğu hâlde A 1,25 kat hızlıdır.

Tablonun üçüncü satırı, donanım değişmeden de programın hızlanabileceğini gösterir. Daha iyi bir derleyici aynı programı $1{,}6 \cdot 10^9$ komuta indirirse A'nın süresi 0,8 s'ye düşer. Demek ki tek bir sayı, örneğin yalnız saat hızı, performansı belirlemez. Bir işlemciyi değerlendirmenin en güvenilir yolu, gerçek programları üzerinde çalıştırıp süreyi ölçmektir.

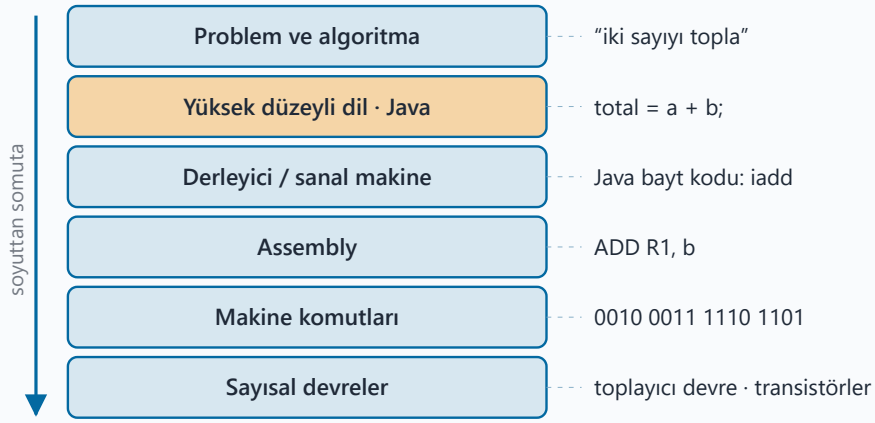
Sık yapılan hata

GHz değeri yüksek olan işlemcinin her zaman daha hızlı olduğunu sanmak. Saat hızı yalnız denklemin bir çarpanıdır. Bir tur daha kısa sürse de komut başına daha çok tur gerekiyorsa (yüksek CPI) toplam süre uzayabilir.

1.3.9 Programlama soyutlama düzeyleri

1.3.9.1 Problemden devreye

Bir programcı Java'da `total = a + b;` yazarken transistörleri, yazmaçları ya da opcode'ları düşünmez. Bunu mümkün kılan, bilgisayar sistemlerinin katmanlar hâlinde kurulmasıdır. Her katman, altındakinin ayrıntısını gizler ve üstündekine daha kullanışlı bir görünüm sunar. Ayrıntıyı gizleyip yalnız gerekeni göstermeye **soyutlama** (*abstraction*) denir.



Şekil 1.3.34: Soyutlama düzeyleri, yukarıdan aşağıya soyuttan somuta: problem ve algoritma ("iki sayıyı topla"), yüksek düzeyli dil (`total = a + b;`), derleyici ya da sanal makine (Java bayt kodu `iadd`), assembly (`ADD R1, b`), makine komutları (0010 0011 1110 1101) ve sayısal devreler (toplayıcı, transistörler). Örnek satırlar gösterim amaçlıdır.

Şekil 1.3.34, aynı "iki sayıyı topla" fikrinin katmanlar boyunca aldığı biçimleri gösterir.

- **Problem ve algoritma:** İnsan dilinde ne yapılacağı ve hangi adımlarla yapılacağı.
- **Yüksek düzeyli dil:** Java, Python, C gibi diller. Okunabilir ve **taşınabilirdir**: aynı Java kodu farklı işlemcilerde çalışır.
- **Derleyici ya da sanal makine:** Yüksek düzeyli kodu alt düzeye çevirir. Java derleyicisi kodu önce bayt koduna (ara koda) çevirir; Java sanal makinesi bu kodu çalıştığı işlemcinin komutlarına dönüştürür.
- **Assembly:** Makine komutlarının insanın okuyabileceği adlarla (LOAD, ADD) yazılmış hâlidir. Her assembly komutu genellikle tek bir makine komutuna karşılık gelir.
- **Makine komutları:** İşlemcinin doğrudan yürüttüğü bit dizileri. Assembly ve makine kodu belirli bir işlemci mimarisine özgüdür: ARM için yazılmış makine kodu x86-64'te çalışmaz.
- **Sayısal devreler:** Komutu gerçekten yürüten kapılar, toplayıcılar ve transistörler.

Soyutlamanın değeri, her katmanda çalışan kişinin yalnız kendi katmanını ve hemen altındakinin sunduğu görünümü bilmesinin yetmesidir. Bir uygulama programcısı toplayıcının iç yapısını bilmeden doğru program yazabilir. Yine de alt katmanları tanımak, taşma ve yuvarlama gibi "beklenmedik" davranışların nedenini anlamayı sağlar.

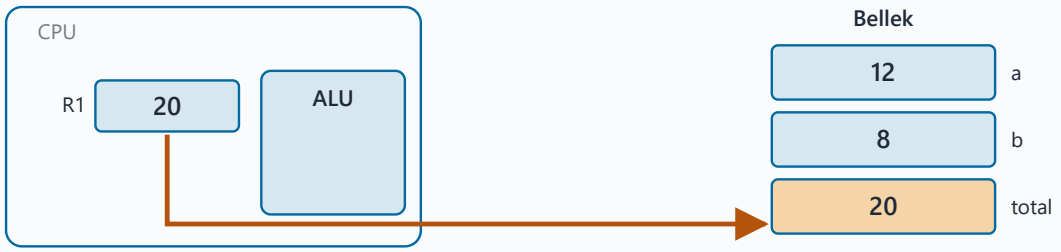
1.3.9.2 Kod izleme: total = a + b

Tek bir Java satırının işlemcide neye dönüştüğü, yazmaçlı bir assembly diliyle gösterilebilir. Bellekte `a = 12` ve `b = 8` vardır:

```
// Java: int total = a + b;
LOAD  R1, a      ; R1 ← a
ADD   R1, b      ; R1 ← R1 + b
STORE R1, total  ; total ← R1
```

Komut	Ne olur?	R1	a	b	total
başlangıç	—	0	12	8	0
`LOAD R1, a`	`a` 'nın değeri bellekten yazmaca taşınır	12	12	8	0
`ADD R1, b`	`b` okunur, ALU 12 + 8 = 20 hesaplar, sonuç R1'e yazılır	20	12	8	0
`STORE R1, total`	R1'deki 20, `total` değişkeninin yerine yazılır	20	12	8	20

Tablo 1.3.21: `total = a + b` satırının assembly izlemesi.



Şekil 1.3.35: STORE adımı. R1'deki 20 belleğe, `total` değişkeninin yerine yazılır; Java satırı tamamlanmıştır.

Tek bir Java satırı üç makine komutuna dönüşmüştür. Önce değer bellekten yazmaca yüklenir, çünkü ALU yalnız yazmaçlarla çalışır. Sonra hesap yapılır, en son sonuç belleğe geri yazılır. Bu yükle–hesapla–yaz örüntüsü, RISC tarzı işlemcilerde neredeyse her atama satırında görülür. Aynı şablonla `int fark = a - b;` satırı da `LOAD R1, a`, `SUB R1, b`, `STORE R1, fark` komutlarına dönüşür. Çıkarmada işlenenlerin sırası önemlidir: önce `a` yüklenir, sonra ondan `b` çıkarılır.

Bu küçük örnek, notun başındaki bütün kavramları bir araya getirir. `a` ve `b` bellekte ikiye tümleyen bit dizileridir; ADD komutu 16 bitlik bir sayı olarak bellekte durur; işlemci onu getir–çöz–yürüt döngüsüyle alır; toplamayı tam toplayıcı zinciri yapar; sonuç veri yoluyla belleğe yazılır. Programcının yazdığı tek satır, bu katmanların hepsinden geçerek çalışır.

1.3.10 Özet

- Bilgisayar bütün verileri **bit dizileriyle** temsil eder. n bitle 2^n farklı dizi yazılır. Bir dizinin anlamını **kodlama** ve **veri tipi** belirler; aynı 01000001 hem 65 hem "A"dır.
- Konumsal sistemde basamak ağırlıkları tabanın kuvvetleridir. Onluktan ikiliye art arda 2'ye bölünerek geçilir. $8 = 2^3$ ve $16 = 2^4$ olduğu için sekizli ve onaltılığa bitler sağdan üçerli ve dörderli gruplanarak çevrilir.
- Negatif tam sayılar **ikiye tümleyen** ile saklanır: bitler çevrilir, 1 eklenir. n bitlik aralık $-2^{n-1} \ldots 2^{n-1} - 1$'dir; 8 bitte −128 … 127. Çıkarma, $a + (-b)$ olarak toplama devresiyle yapılır.
- Sonuç yazmaca sığmazsa **taşma** olur ve yüksek bitler atılır: 8 bitte 200 + 100 = 44. **Elde** işaretsiz, **işaretli taşma** işaretli yorumun sorunudur; aynı işaretli iki sayının toplamı ters işaretli çıkarsa işaretli taşma vardır.
- **Kayan noktalı** sayılar IEEE 754'e göre işaret, üs (sapmalı) ve kesir alanlarıyla saklanır: `float` 1 + 8 + 23, `double` 1 + 11 + 52 bit. Paydası 2'nin kuvveti olmayan kesirler (0,1 gibi) tam saklanamaz; kayan noktalı sonuçlar toleransla karşılaştırılır. İkili gösterim tam sayılar için kesin, kayan nokta bazı ondalıklar için yaklaşıktır.
- **ASCII** 7 bitlik bir koddur (128 karakter). **Unicode** her karaktere bir **kod noktası** verir; **UTF-8** bu numarayı 1–4 baytla yazar: ASCII 1, Türkçe harfler 2, € 3, emoji 4 bayt. Karakter sayısı ile bayt sayısı aynı olmayabilir.

- Bit işlemleri (AND, OR, XOR, NOT, kaydırma) maskelerle tek tek bitleri değiştirir; sola kaydırma 2 ile çarpar, sağa kaydırma 2'ye böler.
- **Mantık kapıları** toplayıcı gibi devrelerin yapı taşıdır. Yarım toplayıcı SUM = A XOR B, CARRY = A AND B üretir; tam toplayıcı gelen eldeyi de toplar. Çok bitli toplama, tam toplayıcı zinciriyle yapılır.
- **Depolanan program** yaklaşımında komutlar ve veriler aynı bellekte sayılar olarak tutulur. CPU, komutları PC ve IR yazmaçlarının yardımıyla **getir–çöz–yürüt döngüsünde** yürütür. Adres, kontrol ve veri yolları CPU'yu belleğe bağlar. Dallanma komutları yürütme aşamasında PC'yi değiştirerek döngü ve karar yapılarını kurar.
- Bellek bayt adreslidir; çok baytlı değerler **büyük uçlu** ya da **küçük uçlu** sırayla yazılır.
- **von Neumann** mimarisinde komut ve veri ortak bellekte ve ortak yoldadır; bu, von Neumann darboğazını doğurur. **Harvard** mimarisi bellekleri ayırır; günümüz işlemcileri ikisini birleştirir. RISC az ve basit, CISC çok ve karmaşık komutla çalışır.
- CPU süresi = komut sayısı × CPI ÷ saat hızı; tek bir sayı performansı belirlemez.
- Yüksek düzeyli diller donanım ayrıntılarını **soyutlar**: tek bir Java satırı birkaç makine komutuna dönüşür.

Hesap	Yöntem	Örnek
Onluk → ikili	art arda 2'ye böl, kalanları sondan oku	$45 = 101101_2$
İkili → sekizli / onaltılık	sağdan 3'erli / 4'erli grupla	$101101 = 55_8 = 2D_{16}$
Negatif (ikiye tümleyen)	bitleri çevir, 1 ekle	$-5 = 11111011$
Taşma (8 bit işaretsiz)	sonuçtan 256 çıkar	$200 + 100 \rightarrow 44$
Kesir → ikili	art arda 2 ile çarp, tam kısımları al	$0{,}375 = 0{,}011_2$
IEEE 754 `float`	işaret, üs + 127, baştaki 1'siz kesir	$-6{,}25 = \mathrm{C0C80000}_{16}$
UTF-8 (2 bayt)	11 bit → `110xxxxx 10xxxxxx`	"ğ" = C4 9F
CPU süresi	komut sayısı × CPI ÷ saat hızı	$2 \cdot 10^9 \times 1{,}5 \div 3$ GHz = 1 s

Tablo 1.3.22: Notta çözülen hesap türleri ve kısa yolları.

1.3.11 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

bit / bayt *bit / byte* · kodlama *encoding* · veri tipi *data type* · taban *base (radix)* · ikili / sekizli / onluk / onaltılık *binary / octal / decimal / hexadecimal* · işaretli / işaretsiz *signed / unsigned* · birler tümleyeni *one's complement* · ikiye tümleyen *two's complement* · taşma *overflow* · işaretli taşma *signed overflow* · elde *carry* · kayan nokta *floating point* · IEEE 754 · işaret / üs / kesir *sign / exponent / fraction* · sapma *bias* · hassasiyet *precision* · yuvarlama *rounding* · ASCII · kontrol karakteri *control character* · bit işlemi *bitwise operation* · bit maskesi *bit mask*

bit kaydırma *bit shift* · Unicode · kod noktası *code point* · UTF-8 · dizgi *string* · kod birimi *code unit* · alan / kayıt / dosya biçimi *field / record / file format* · mantık kapısı *logic gate* · yarım toplayıcı *half adder* · tam toplayıcı *full adder* · dalgalı elde toplayıcı *ripple-carry adder* · depolanan program *stored program* · getir–çöz–yürüt döngüsü *fetch-decode-execute cycle* · program sayacı (PC) *program counter* · komut yazmacı (IR) *instruction register* · akümülatör (ACC) *accumulator* · yazmaç *register* · kontrol birimi *control unit* · önbellek *cache* · veri yolu / adres yolu / kontrol yolu *data / address / control bus* · bellek adresi *memory address* · bayt sırası *endianness* · dallanma *branch* · von Neumann mimarisi *von Neumann architecture* · von Neumann darboğazı *von Neumann bottleneck* · Harvard mimarisi *Harvard architecture* · opcode (işlem kodu) *opcode* · komut biçimi *instruction format* · RISC / CISC · CPI (komut başına saat turu) *cycles per instruction* · soyutlama *abstraction* · assembly *assembly language* · makine komutu *machine instruction*

1.3.12 Kendini Dene

1. 156 sayısı ikili, sekizli ve onaltılık tabanda nasıl yazılır?

Cevap

156 = 128 + 16 + 8 + 4 olduğundan 10011100_2'dir. Üçerli gruplar 010 | 011 | 100 → 234_8. Dörderli gruplar 1001 | 1100 → $9C_{16}$. Denetim: $9 \cdot 16 + 12 = 156$.

2. 8 bit ikiye tümleyen gösterimde −20 nasıl yazılır? 10110100 dizisi hangi sayıdır?

Cevap

+20 = 00010100; bitler çevrilir: 11101011; 1 eklenir: 11101100 = −20. 10110100'ün işaret biti 1'dir, sayı negatiftir. Bitler çevrilip 1 eklenince 01001100 = 76 bulunur; dizi −76'dır. Denetim: −128 + 32 + 16 + 4 = −76.

3. 8 bitte 90 + (−30) ve 70 + 80 işlemlerinde elde ve işaretli taşma var mıdır? Sonuçlar doğru mudur?

Cevap

90 + (−30): 01011010 + 11100010 = 1 00111100. Elde 1'dir ve atılır; kalan 00111100 = 60 doğrudur. İşaretleri farklı iki sayının toplamı taşmaz. 70 + 80: 01000110 + 01010000 = 10010110. Elde 0'dır, ama iki pozitifin toplamı negatif görünür: işaretli taşma vardır. Dizi −106 okunur; doğru sonuç 150, işaretli 8 bitlik aralığa sığmaz.

4. 0,625 ve 0,2 sayılarının ikili açılımları nedir? Hangisi `double` ile tam saklanır?

Cevap

0,625 × 2 = 1,25 → 1; 0,25 × 2 = 0,5 → 0; 0,5 × 2 = 1,0 → 1 ve kesir 0'dır: $0{,}101_2$, tam saklanır (5/8, payda 2'nin kuvveti). 0,2 × 2 = 0,4 → 0; 0,8 → 0; 1,6 → 1; 1,2 → 1; sonra kesir yine 0,2 olur: $0{,}0011\ 0011\ldots_2$, sonsuza dek tekrarlar ve tam saklanamaz.

5. 0,75 sayısının 32 bitlik `float` karşılığı onaltılık olarak nedir?

Cevap

$0{,}75 = 0{,}11_2 = 1{,}1 \times 2^{-1}$. İşaret 0; üs −1 + 127 = 126 = 01111110; kesir 1 ve yirmi iki 0. Bitler: 0 01111110 1000…0 → 0011 1111 0100 0000 … → $3F400000_{16}$.

6. "ü" (U+00FC) harfinin UTF-8 baytları nedir? "Gün" kelimesi kaç karakter, kaç bayttır?

Cevap

$00FC_{16} = 252 = 11111100_2$; 7 bite sığmaz, 11 bitle 00011111100 yazılır. 5 + 6 bölmesi: 00011 | 111100. Baytlar 110 00011 = C3 ve 10 111100 = BC: "ü" = C3 BC. "Gün" 3 karakterdir; G ve n 1'er, ü 2 bayt olduğundan 4 bayttır.

7. Dört tam toplayıcılı zincirde 0101 + 0011 toplanıyor. Her basamakta çıkan elde nedir, sonuç ve son elde ne olur?

Cevap

i = 0: $1 + 1 + 0 = 10_2$, basamak 0, elde 1. *i = 1*: $0 + 1 + 1 = 10_2$, basamak 0, elde 1. *i = 2*: $1 + 0 + 1 = 10_2$, basamak 0, elde 1. *i = 3*: 0 + 0 + 1 = 1, basamak 1, elde 0. Sonuç $1000_2 = 8 = 5 + 3$; son elde 0 olduğu için 4 bite sığmıştır.

8. Dallanma programında 1010 adresindeki sayaç 2 yerine 3 olsaydı program bitince `toplam` ne olurdu? Kaç komut getirilirdi?

Cevap

Döngü 3 tur döner ve toplam = 3 + 2 + 1 = 6 olur. Her tur 8 komuttur; çıkışta LOAD, JUMPZ ve HALT getirilir: 8 · 3 + 3 = 27 komut.

9. $3 \cdot 10^9$ komutluk bir program, CPI'si 2 olan 2 GHz'lik bir işlemcide ve CPI'si 1,2 olan 3 GHz'lik bir

işlemcide kaç saniye sürer? İkincisi kaç kat hızlıdır?

Cevap

Birinci: $3 \cdot 10^9 \times 2 \div (2 \cdot 10^9) = 3$ s. İkinci: $3 \cdot 10^9 \times 1{,}2 \div (3 \cdot 10^9) = 1{,}2$ s. $3 \div 1{,}2 = 2{,}5$; ikinci işlemci 2,5 kat hızlıdır.

10. 32 bitlik 0x0A0B0C0D değeri, küçük uçlu bir makinede 200–203 adreslerine nasıl yazılır? Büyük uçlu bir makine bu dört baytı tek seferde okursa hangi değeri görür?

Cevap

Küçük uçlu düzende en az anlamlı bayt en küçük adrese gider: 200: 0D, 201: 0C, 202: 0B, 203: 0A. Büyük uçlu makine ilk baytı en anlamlı sayar ve 0x0D0C0B0A görür.

1.3.13 İleri okuma

- C. Petzold, *Code: The Hidden Language of Computer Hardware and Software* (2. baskı, Microsoft Press, 2022): Mors alfabesinden ve ampullü devrelerden başlayarak kapıları, toplayıcıları, belleği ve basit bir işlemciyi matematik bilgisi gerektirmeden adım adım kurar.
- N. Nisan ve S. Schocken, *The Elements of Computing Systems* (2. baskı, MIT Press, 2021): NAND kapısından başlayıp ALU, CPU, assembler ve derleyiciye kadar bütün katmanları kurdurarak bu nottaki soyutlama düzeylerini uygulamalı olarak işler.
- R. E. Bryant ve D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective* (3. baskı, 2016), Bölüm 2: tam sayıların ikiye tümleyen gösterimini, taşmayı ve IEEE 754'ü programcı gözüyle ve çok sayıda örnekle anlatır.
- D. Goldberg, "What Every Computer Scientist Should Know About Floating-Point Arithmetic", *ACM Computing Surveys* 23(1), 1991: kayan noktalı hesaptaki yuvarlama hatalarının nedenlerini ve sonuçlarını ayrıntılı ele alan klasik makale.

1.3.14 Kaynaklar

1. D. A. Patterson, J. L. Hennessy, *Computer Organization and Design: The Hardware/Software Interface*, Morgan Kaufmann.
2. R. E. Bryant, D. R. O'Hallaron, *Computer Systems: A Programmer's Perspective*, 3. baskı, Pearson, 2016.
3. The Unicode Consortium, *The Unicode Standard*, https://www.unicode.org/standard/standard.html.
4. IEEE, *IEEE Standard for Floating-Point Arithmetic* (IEEE Std 754-2019), 2019.
5. C. Petzold, *Code: The Hidden Language of Computer Hardware and Software*, 2. baskı, Microsoft Press, 2022.
6. N. Nisan, S. Schocken, *The Elements of Computing Systems: Building a Modern Computer from First Principles*, 2. baskı, MIT Press, 2021.

7. D. Goldberg, “What Every Computer Scientist Should Know About Floating-Point Arithmetic”, *ACM Computing Surveys*, 23(1), 5–48, 1991.

KISIM I · BÖLÜM 1.4

İnternet

Bir web sayfasını açmak, bir e-posta göndermek ya da görüntülü görüşme yapmak bugün birkaç saniyelik işlerdir. Bu işlerin arkasında birbirine bağlı milyonlarca ağ, ortak kurallarla konuşan cihazlar ve dünyanın öbür ucundaki sunucular vardır. Not, bu düzeni baştan sona izler. Önce bilgisayar ağının ne olduğunu, internete nasıl bağlanıldığını, bant genişliği ile gecikme arasındaki farkı ve ağ cihazlarını anlatır. Ardından verinin katmanlar hâlinde paketlenmesini, paketlerin yönlendiricilerden geçişini, IP adreslerini, NAT'ı, DNS'i, URL'yi, portları ve TCP bağlantısını adım adım ele alır. Şifreleme ve HTTPS'ten sonra Web'in, arama motorlarının, bulut hizmetlerinin ve e-postanın nasıl çalıştığına geçer. Son kesim güvenliğe ayrılmıştır: bilgi güvenliğinin üç hedefi, kötü amaçlı yazılımlar, oltalama, parola gücü, çok faktörlü doğrulama, VPN, güvenlik duvarı ve yedekleme.

ÖĞRENME ÇIKTILARI

Ağ kavramları

İnternet, istemci, sunucu ve protokol arasındaki ilişkiyi açıklamak; ağ cihazlarının görevlerini ayırt etmek; bant genişliği ve gecikmeyle basit süre hesapları yapmak.

Adresleme

IP adresi, alan adı, DNS, URL ve port kavramlarını birbirinden ayırmak; bir IPv4 adresini ikili sayıya çevirmek; paketin katmanlardan, yönlendiricilerden ve NAT'tan geçişini izlemek.

Web kullanımı

Bir web isteğinin yolculuğunu adım adım anlatmak; arama motorlarının ters dizinle nasıl çalıştığını açıklamak; arama operatörlerini ve kaynak değerlendirme ölçütlerini kullanmak.

Güvenlik

Tehditleri tanımak; oltalama iletisinin işaretlerini görmek; parola gücünü hesapla karşılaştırmak; çok faktörlü doğrulama, güvenlik duvarı ve yedekleme gibi önlemleri doğru yerde kullanmak.

1.4.1 Ağların temeli

İnternet tek bir makine ya da tek bir kablo değildir. Birbirinden bağımsız kurulmuş çok sayıda ağın, ortak kurallara uyarak birbirine bağlanmasıyla oluşur. Bu yüzden internetin nasıl çalıştığını anlamanın yolu, önce küçük bir ağın nasıl çalıştığını anlamaktan geçer. Bu kesim temel kavramları kurar: ağ, internet, erişim hattı, bant genişliği, gecikme, istemci, sunucu, protokol ve ağ cihazları. Kesimin sonunda kablosuz ağlar ve güvenlikleri ele alınır.

1.4.1.1 Bilgisayar ağı ve internet

Bir evde iki dizüstü bilgisayar, bir telefon ve bir yazıcı olduğunu düşünelim. Dizüstülerden biri belgeyi yazıcıya göndermek, öteki ise telefondaki fotoğrafları almak istiyor. Bu cihazların veri alışverişi yapabilmesi için bir şekilde birbirine bağlı olması gerekir. Bağlantı bir kablo da olabilir, bir radyo dalgası da. Bağlantının biçimi ne olursa olsun ortaya çıkan yapıya ağ denir.

Tanım 1.4.1

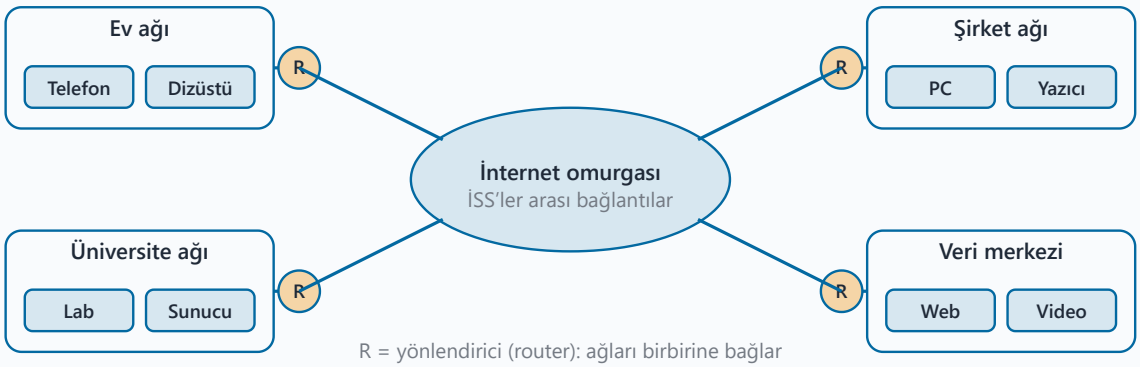
Bilgisayar ağı (*computer network*), cihazların veri ve kaynak paylaşmak için birbirine bağlanmasıyla oluşan yapıdır. Paylaşılan kaynak bir dosya, bir yazıcı, bir internet bağlantısı ya da bir hizmet olabilir.

Tek bir evin ağı küçüktür; bir üniversitenin ağı yüzlerce bina ve binlerce cihaz içerir; bir şirketin ağı farklı şehirlerdeki şubeleri birbirine bağlar. Bu ağların her biri kendi sahibince kurulur ve yönetilir. İnternet, bu ayrı ayrı ağları tek bir dev ağ gibi çalıştırır.

Tanım 1.4.2

İnternet (*Internet*), farklı büyüklükteki ağları ortak kurallarla birleştiren küresel **ağlar ağıdır**. Bu ortak kuralların adı **TCP/IP**'dir: aynı kuralları uygulayan her ağ, internetin bir parçası olabilir.

Şekil 1.4.1'de dört ayrı ağ görülüyor: bir ev ağı, bir üniversite ağı, bir şirket ağı ve bir veri merkezi. Her ağın kendi cihazları vardır ve her ağ bir **yönlendirici** (*router*) üzerinden dışarıya bağlanır. Yönlendiriciler de internet servis sağlayıcılarının birbirine bağlı büyük hatlarından oluşan **internet omurgasına** bağlanır. Ev ağındaki telefon, veri merkezindeki video sunucusuna ulaşmak için en az iki yönlendiriciden ve omurgadan geçer.



Şekil 1.4.1: Ağlar ağı. Ev, üniversite ve şirket ağları ile veri merkezi, kendi yönlendiricileri (R) üzerinden internet omurgasına bağlanır. Omurga, internet servis sağlayıcıları arasındaki yüksek kapasiteli bağlantılardan oluşur.

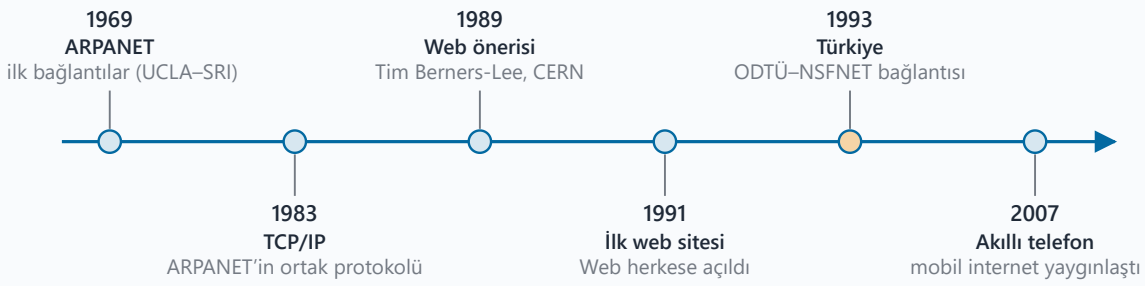
Bu yapının önemli bir sonucu vardır: internetin bir merkezi yoktur. Bir ağ çöktüğünde ötekiler çalışmayı sürdürür; bir hat koptuğunda trafik çoğu zaman başka bir hattan dolaşır. İnternet de zaten bu dayanıklılık düşünülerek tasarlanmıştır.

> **Sık yapılan hata**
>
> İnterneti tek bir şirketin işlettiği tek bir ağ sanmak. İnternet, binlerce bağımsız kuruluşun işlettiği ağlardan oluşur. Bu kuruluşları bir arada tutan şey ortak bir sahip değil, herkesin uyduğu ortak protokollerdir.

1.4.1.2 İnternetin gelişimi

İnternetin kökeni 1960'ların sonuna uzanır. ABD Savunma Bakanlığına bağlı bir araştırma ajansının desteklediği **ARPANET** ağı, ilk bağlantılarını 1969'da kurdu. Bu bağlantılar Kaliforniya Üniversitesi Los Angeles (UCLA) ile Stanford Araştırma Enstitüsündeki (SRI) bilgisayarlar arasındaydı. Sonraki yıllarda ağa yeni üniversiteler ve araştırma merkezleri katıldı. Her yeni ağın kendi kurallarıyla çalışması, ağlar arasında konuşmayı zorlaştırıyordu. Bu sorun 1 Ocak 1983'te çözüldü: ARPANET ortak protokol olarak **TCP/IP**'ye geçti. Bugünkü internet bu ortak dilin üzerinde büyüdü.

1989'da CERN'de çalışan Tim Berners-Lee, belgeleri birbirine bağlantılarla bağlayan bir bilgi sistemi önerdi. Bu öneri **Web**'in (*World Wide Web*) başlangıcıdır. İlk web sitesi 1991'de yayına girdi ve Web herkesin kullanımına açıldı. 2007'de dokunmatik akıllı telefonlar yaygınlaşmaya başladı ve internet cebe girdi. Bugün mobil erişim masaüstü erişimini geride bırakmıştır.



Şekil 1.4.2: İnternetin gelişiminde altı dönüm noktası: ARPANET'in ilk bağlantıları (1969), TCP/IP'ye geçiş (1983), Web önerisi (1989), ilk web sitesi (1991), Türkiye'nin internete bağlanması (1993) ve akıllı telefon (2007).

Tarihçede iki kavramın ayrı tarihleri olduğuna dikkat etmek gerekir. İnternet 1969'dan beri gelişen bir **altyapıdır**; Web ise 1991'de bu altyapının üzerinde başlayan bir **hizmettir**. İkisi arasındaki fark 1.4.5.1'de ayrıntılı ele alınır.

Türkiye'de internet. Türkiye'nin ilk sürekli internet bağlantısı 12 Nisan 1993'te Orta Doğu Teknik Üniversitesi (ODTÜ) ile ABD'deki akademik omurga NSFNET arasında kuruldu. Hattın hızı 64 Kbps, yani saniyede 64 bin bitti. 1996'da TÜBİTAK bünyesinde Ulusal Akademik Ağ ve Bilgi Merkezi (ULAKBİM) kuruldu. Üniversiteleri birbirine ve internete bağlayan ulusal akademik ağ ULAKNET ise 1997'de işletmeye girdi. Bugün fiber, mobil ve uydu bağlantıları farklı erişim seçenekleri sunar ve evlere kadar gelen fiber hatlarda hız 1 Gbps'e ulaşır.



Şekil 1.4.3: Türkiye'de internetin zaman çizelgesi: 1993'te ODTÜ–NSFNET arasındaki 64 Kbps'lik hat, 1996'da ULAKBİM'in kuruluşu ve 1997'de ULAKNET, günümüzde fiber, mobil ve uydu erişimi.

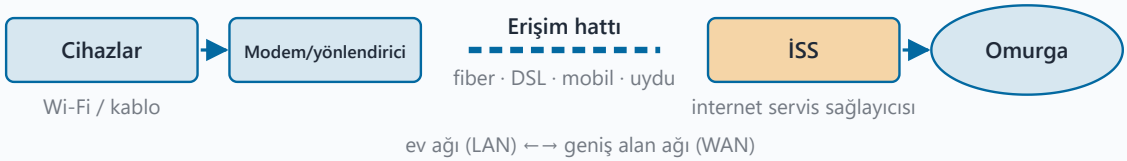
1.4.1.3 İnternete bağlanma

Bir evin ya da bir kurumun ağı internete kendi başına bağlanamaz; araya bir hizmet sağlayıcı girer. Bu sağlayıcı, kullanıcının ağını kendi ağına, kendi ağını da internet omurgasına bağlar.

Tanım 1.4.3

İnternet servis sağlayıcısı (*Internet service provider*, kısaca İSS ya da ISP), kullanıcıyı internet omurgasına bağlayan kuruluştur. Kullanıcının ağı ile İSS arasındaki bağlantıya **erişim hattı** denir.

Şekil 1.4.4 evden internete giden yolu gösterir. Evdeki cihazlar Wi-Fi ya da kablo ile modem/yönlendiriciye bağlanır. Bu kutu erişim hattı üzerinden İSS'ye, İSS de omurgaya bağlanır. Şeklin altındaki yazı iki ağ türünü de adlandırır: evin içi bir yerel ağdır (LAN), dışarısı ise geniş alan ağıdır (WAN). Bu iki kavram 1.4.1.7'de tanımlanır.



Şekil 1.4.4: Evden internete: cihazlar, modem/yönlendirici, erişim hattı (fiber, DSL, mobil ya da uydu), internet servis sağlayıcısı ve omurga.

Erişim hattı farklı teknolojilerle kurulabilir. Tablo 1.4.1 dört yaygın seçeneği karşılaştırır.

Tür	Nasıl çalışır?	Güçlü yanı	Zayıf yanı
Fiber	Veri, cam liflerin içinde ışık darbeleriyle taşınır.	Yüksek kapasite ve düşük gecikme.	Kablonun binaya kadar döşenmesi gerekir.
DSL	Mevcut bakır telefon hattı kullanılır.	Telefon hattı olan hemen her yerde vardır.	Hız, santrale olan uzaklık arttıkça düşer.
Mobil veri	Telefon ya da modem, baz istasyonlarıyla radyo dalgası üzerinden konuşur.	Taşınabilir; kablo gerektirmez.	Hız kapsama alanına ve kalabalığa bağlıdır; çoğu zaman kotalıdır.
Kablosuz ve uydu	Sabit kablosuz verici ya da yörüngedeki uydu kullanılır.	Kablo çekmenin zor olduğu kırsal ve dağlık bölgelerde seçenektir.	Yüksek yörüngeli uyduda gecikme çok yüksektir.

Tablo 1.4.1: İnternet erişim hattı türleri.

Evlerde modem, yönlendirici, kablosuz erişim noktası ve küçük bir switch çoğu zaman tek bir kutuda birleşir. Bu kutu halk arasında “modem” diye anılır; ama içinde aslında birkaç ayrı cihaz çalışır. Bu cihazların görevleri 1.4.1.6'da ayrı ayrı anlatılır.

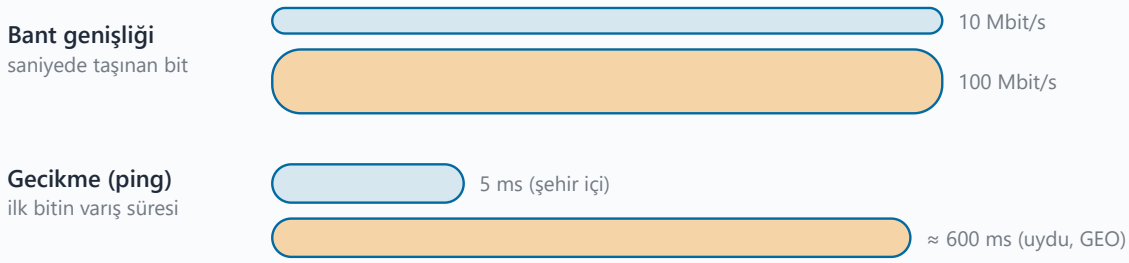
1.4.1.4 Bant genişliği ve gecikme

Bir bağlantının “hızlı” olduğunu söylerken aslında iki ayrı özellikten söz ediyor olabiliriz. Birincisi, bağlantının saniyede ne kadar veri taşıyabildiğidir. İkincisi, gönderilen ilk bitin karşıya ne kadar sürede ulaştığıdır. Bu iki özellik birbirinden bağımsızdır ve farklı işlerde belirleyici olur.

Tanım 1.4.4

Bant genişliği (*bandwidth*), bir bağlantının birim zamanda taşıyabildiği en yüksek veri miktarıdır ve saniyedeki bit sayısıyla ölçülür (bit/s, Mbit/s, Gbit/s). **Gecikme** (*latency*) ise bir bitin kaynaktan hedefe ulaşması için geçen süredir ve milisaniyeyle (ms) ölçülür. Gidiş-dönüş gecikmesi, günlük dilde **ping** değeri olarak anılır.

İki kavramı bir su borusuyla düşünmek kolaylık sağlar (Şekil 1.4.5). Bant genişliği borunun *genişliğidir*: geniş boru saniyede daha çok su akıtır. Gecikme ise borunun *uzunluğudur*: uzun boruda musluğu açınca suyun öbür uçtan çıkması daha uzun sürer. Çok geniş ama çok uzun bir boru da olabilir; dar ama kısa bir boru da.



Şekil 1.4.5: Bant genişliği borunun genişliği, gecikme ise borunun uzunluğu gibidir. 10 Mbit/s ile 100 Mbit/s arasındaki fark genişlikte, şehir içi 5 ms ile yer eşzamanlı (GEO) uydudaki yaklaşık 600 ms arasındaki fark uzunlukta görülür.

Bit ve bayt. Hesaplarda en çok karıştırılan nokta birimlerdir. Hat hızları *bit* cinsinden verilir (Mbit/s ya da Mbps). Dosya boyutları ise *bayt* cinsinden verilir (MB, GB). Bir bayt 8 bittir. Bu yüzden indirme süresi hesaplanırken iki büyüklük önce aynı birime çevrilir. Bu notta 1 GB = 1000 MB ve 1 MB = 1000 KB alınır.

> indirme süresi (s) = dosya boyutu (MB) × 8 ÷ hat hızı (Mbit/s)

Bu formül “en iyi durumu” verir: hattın tam hızda çalıştığı, başka kimsenin hattı kullanmadığı ve protokol başlıklarının ek yük getirmediği varsayılır. Gerçek süre biraz daha uzundur.

Örnek 1.4.1

100 Mbit/s'lik bir hatta 500 MB'lık bir dosya en iyi durumda kaç saniyede iner?

1. Hat hızını bayta çevirelim: 100 Mbit/s ÷ 8 = 12,5 MB/s. Hat saniyede 12,5 megabayt taşır.
2. Dosya boyutunu bu hıza bölelim: 500 MB ÷ 12,5 MB/s = 40 s.
3. Aynı sonuca öteki yoldan da ulaşılır: 500 MB = 500 × 8 = 4000 Mbit; 4000 ÷ 100 = 40 s.

Dosya en iyi durumda **40 saniyede** iner. Aynı yöntemle 200 Mbit/s'lik bir hatta 1,5 GB'lık dosya için 1,5 GB = 1500 MB = 12.000 Mbit ve 12.000 ÷ 200 = 60 s bulunur.

Sık yapılan hata

"100 megabitlik internet" ile "saniyede 100 megabayt" aynı şey sanılır. 100 Mbit/s'lik hat saniyede en çok 12,5 MB taşır. Hesapta bit ile bayt karıştırılırsa sonuç tam 8 kat yanlış çıkar.

Gecikme nereden gelir? Sinyal ışık hızından hızlı gidemez. Fiberde ışık, camın içinde boşluktakinden yavaş yol alır; hızı yaklaşık saniyede 200.000 km'dir. Bu yüzden 800 km uzaklıktaki bir sunucuya fiberle gidip gelmek en az 2 × 800 ÷ 200.000 s = 8 ms sürer. Başka hiçbir bekleme olmasa bile bu süre kısalmaz. Yönlendiricilerde bekleme ve işleme süreleri buna eklenir.

Örnek 1.4.2

Yer eşzamanlı (GEO) uydular ekvatorun yaklaşık 35.786 km üzerinde döner. Uydu üzerinden kurulan bir bağlantıda gidiş-dönüş gecikmesinin alt sınırı nedir? (Işık hızı ≈ 300.000 km/s.)

1. Bir yönde sinyal yerden uyduya çıkar ve uydudan yere iner: 2 × 35.786 = 71.572 km.
2. Bu yolun süresi: 71.572 ÷ 300.000 ≈ 0,239 s, yani yaklaşık 239 ms.
3. İstek gider, yanıt döner: gidiş-dönüşte yol dört kez katedilir ve süre yaklaşık 477 ms olur.

Yalnız yolun kendisi yaklaşık yarım saniye tutar. Yer istasyonlarındaki işleme ve kara bağlantıları da eklenince gerçekte ölçülen **≈ 600 ms** değerine ulaşılır. Bu gecikme, hattın bant genişliği ne kadar yüksek olursa olsun ortadan kalkmaz.

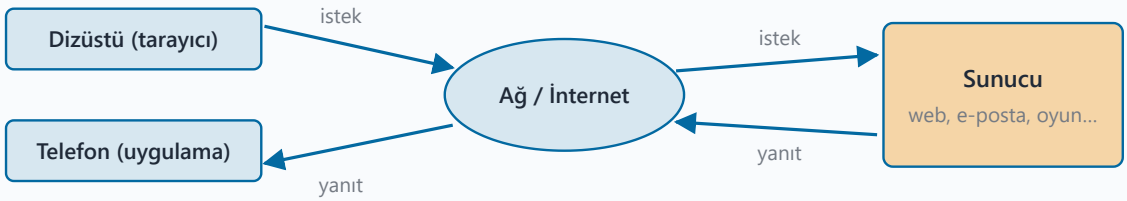
Hangisi önemli? Cevap işin türüne bağlıdır. Büyük bir dosya indirirken belirleyici olan bant genişliğidir: ilk bitin yarım saniye geç gelmesi, birkaç dakikalık indirmede fark edilmez. Oyunda, görüntülü görüşmede ve uzaktan masaüstü bağlantısında ise belirleyici olan düşük gecikmedir. Görüntülü görüşme birkaç Mbit/s ile yetinir, ama 600 ms gecikmede konuşmacıların sözleri üst üste biner. Örneğin 100 Mbit/s'lik bir uydu hattı ile 30 Mbit/s'lik bir fiber hattı karşılaştıralım. 1 GB'lık dosya uydu hattında yaklaşık 80 s'de, fiber hatta yaklaşık 267 s'de iner. Görüntülü görüşmede ise yaklaşık 10 ms gecikmeli fiber hat açıkça daha iyidir.

1.4.1.5 İstemci, sunucu ve protokol

İnternetteki iletişimin büyük bölümü aynı kalıbı izler. Bir taraf bir şey ister, öteki taraf o isteği karşılar. Örneğin bir öğrencinin tarayıcısı ders sayfasını ister; üniversitenin web sunucusu sayfayı gönderir. Telefondaki hava durumu uygulaması tahmini ister; hava durumu hizmetinin sunucusu tahmini gönderir.

Tanım 1.4.5

İstemci (*client*), bir hizmet isteyen cihaz ya da programdır. **Sunucu** (*server*), istekleri karşılayan ve kaynak sunan sistemdir. **Protokol** (*protocol*) ise iletişimin biçimini ve sırasını belirleyen kurallar bütünüdür: kim önce konuşur, ileti hangi biçimde yazılır, bir hata olursa ne yapılır.



Şekil 1.4.6: İstemciler (tarayıcı çalışan dizüstü, uygulama çalışan telefon) ağ üzerinden sunucuya istek gönderir; sunucu her isteğe bir yanıt döndürür.

"Sunucu" sözcüğü iki anlamda kullanılır. Bazen bir donanımı, yani veri merkezindeki güçlü bir bilgisayarı anlatır. Bazen de bir programı, yani istekleri dinleyip yanıtlayan yazılımı anlatır. Aynı makinede bir web sunucusu, bir e-posta sunucusu ve bir veri tabanı sunucusu birlikte çalışabilir. Benzer biçimde bir bilgisayar bir işte istemci, başka bir işte sunucu olabilir.

1.4.1.6 Ağ cihazları

Bir ağ, cihazların yanı sıra bu cihazları birbirine ve dış dünyaya bağlayan özel donanımlardan oluşur. Şekil 1.4.7 tipik bir ev ağındaki sıralamayı gösterir: internetten gelen hat önce modeme, oradan güvenlik duvarına ve yönlendiriciye ulaşır. Yönlendiricinin arkasında kablolu cihazlar için bir switch, kablosuz cihazlar için bir erişim noktası bulunur.



Şekil 1.4.7: Ev ağı şeması: internet, modem, güvenlik duvarı, yönlendirici; yönlendiriciden switch aracılığıyla PC ve yazıcıya, erişim noktasından Wi-Fi ile telefona. Her cihazda bir ağ kartı (NIC) vardır.

Cihaz	Görevi	Neye bakarak çalışır?
Ağ kartı (NIC)	Cihazın ağa bağlanmasını sağlar; bitleri kabloya ya da radyo dalgasına verir.	Her kartın üretimde atanan bir MAC adresi vardır.

Cihaz	Görevi	Neye bakarak çalışır?
Switch	Yerel ağdaki çerçeveleri yalnız hedef cihaza iletir.	Çerçevedeki hedef MAC adresine.
Yönlendirici	Farklı ağlar arasında paket yönlendirir.	Paketteki hedef IP adresine ve yönlendirme tablosuna.
Modem	Erişim hattındaki sinyali (ışık, elektrik, radyo) ağ verisine dönüştürür; tersini de yapar.	Hattın fiziksel sinyaline.
Erişim noktası	Kablosuz cihazları yerel ağa bağlar.	Wi-Fi standardına ve kanal ayarlarına.
Güvenlik duvarı	Ağ trafiğini kurallara göre denetler; izin verir ya da engeller.	Çoğunlukla IP adresi ve port numarasına.

Tablo 1.4.2: Ağ cihazları ve görevleri.

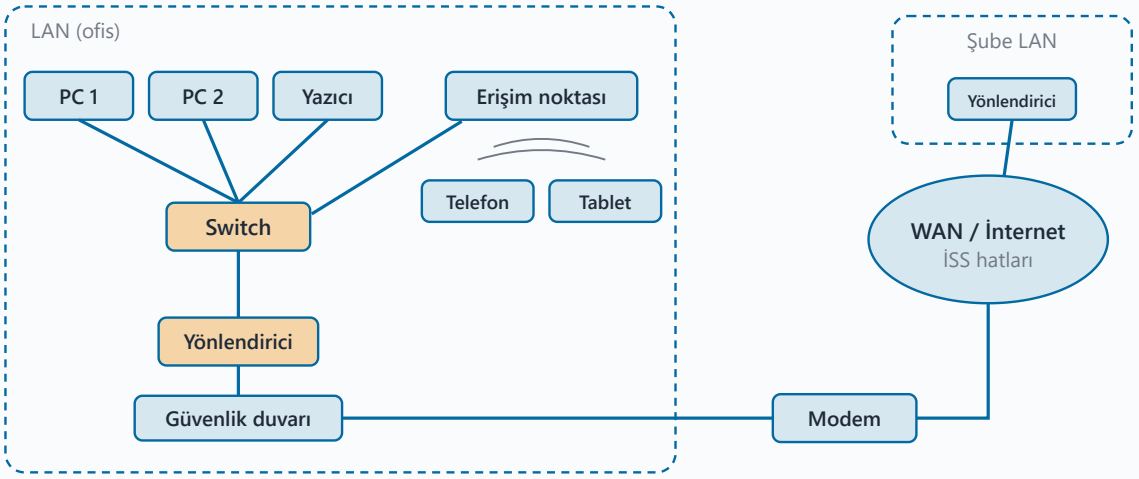
Tablodaki iki cihaz sık karıştırılır: switch ve yönlendirici. İkisinin farkı, hangi sınırın içinde çalıştıklarıdır. **Switch** tek bir yerel ağın içinde çalışır. Ofisteki PC'nin yazıcıya gönderdiği veri switch'e gelir; switch hangi portunun ucunda yazıcının durduğunu bilir ve veriyi yalnız o porta verir. **Yönlendirici** ise ağlar arasındaki sınırda durur. Aynı PC bir web sitesine bağlanmak istediğinde veri yerel ağın dışına çıkmalıdır; bu işi yönlendirici yapar ve veriyi bir sonraki ağa doğru yollar.

Sık yapılan hata

Evdeki kutuya “modem” denildiği için bütün işi modemin yaptığını düşünmek. Modem yalnız erişim hattının sinyalini çevirir. Adres dağıtma, ağlar arasında yönlendirme ve NAT yönlendiricinin; kablosuz bağlantı erişim noktasının; trafiği süzme güvenlik duvarının işidir. Hepsi aynı kutuda olabilir, ama görevleri ayrıdır.

1.4.1.7 Tipik bir ağ ve coğrafi kapsam

Cihazları bir araya getirince tipik bir kurum ağı ortaya çıkar (Şekil 1.4.8). Ofisteki iki PC ve yazıcı kabloyla switch'e bağlıdır. Telefon ve tablet erişim noktası üzerinden kablosuz bağlanır; erişim noktası da switch'e bağlıdır. Switch yönlendiriciye, yönlendirici güvenlik duvarına, güvenlik duvarı da modem üzerinden İSS hattına bağlanır. Aynı kurumun başka şehirdeki şubesi de kendi yönlendiricisiyle internete bağlıdır. İki ofis birbirine internet üzerinden ulaşır.

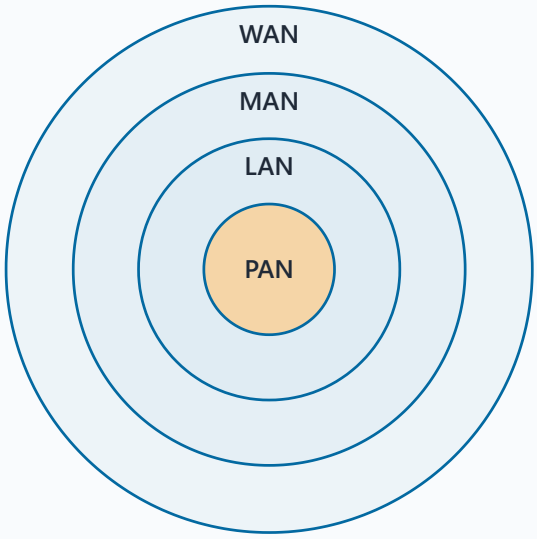


Şekil 1.4.8: LAN'dan WAN'a tipik bir ağ. Kesikli çerçevenin içi ofisin yerel ağıdır (LAN). Switch aynı yerel ağdaki cihazları, yönlendirici ise farklı ağları birbirine bağlar. Şube ağı ile ofis ağı, İSS hatlarından oluşan geniş alan ağı (WAN) üzerinden haberleşir.

Ağlar coğrafi kapsamlarına göre de adlandırılır. Kapsam büyüdükçe ağın kurulumu ve işletilmesi başka kuruluşlara (İSS'ler, operatörler) devredilir.

Tür	Açılımı	Kapsam	Örnek
PAN	kişisel alan ağı (*personal area network*)	Birkaç metre	Telefon ile kablosuz kulaklık
LAN	yerel ağ (*local area network*)	Oda, bina ya da kampüs	Ofis ağı, okul laboratuvarı
MAN	metropol alan ağı (*metropolitan area network*)	Şehir ölçeği	Belediye ağı
WAN	geniş alan ağı (*wide area network*)	Ülke ya da kıta	Kurumun şubeleri; internet en büyük WAN'dır

Tablo 1.4.3: Ağların coğrafi kapsamı.



Şekil 1.4.9: Ağların kapsamı iç içe halkalar gibidir: PAN en içte, onu LAN, MAN ve WAN izler.

1.4.1.8 Wi-Fi standartları ve bantları

Kablosuz yerel ağlar, IEEE (Elektrik ve Elektronik Mühendisleri Enstitüsü) 802.11 standartlar ailesine göre çalışır. Standartların teknik adları akılda kalmadığı için üreticiler 2018'den beri nesil numarası kullanır: Wi-Fi 5, Wi-Fi 6 gibi. Cep telefonu ağları ise farklı bir kuruluşun (3GPP) standartlarına dayanır.

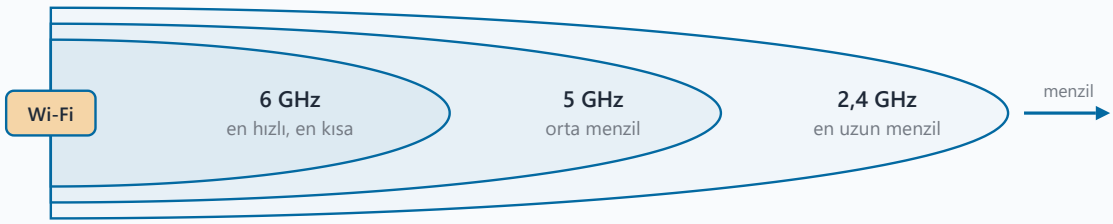
Ad	Standart	Bantlar	Başlıca fark
Wi-Fi 5	802.11ac	5 GHz	Geniş kanallar; 802.11n'e göre belirgin hız artışı.
Wi-Fi 6	802.11ax	2,4 ve 5 GHz	Kalabalık ortamda daha verimli (OFDMA); çok cihazlı ağlarda daha iyi.
Wi-Fi 6E	802.11ax	2,4, 5 ve 6 GHz	Wi-Fi 6'ya 6 GHz bandı eklenir; girişim daha azdır.
Wi-Fi 7	802.11be	2,4, 5 ve 6 GHz	Daha geniş kanallar (320 MHz); birden çok bantta eş zamanlı bağlantı (MLO).
4G (LTE) → 5G	3GPP	Operatör bantları	5G'de tipik olarak daha yüksek hız ve daha düşük gecikme; gerçek değer operatöre ve kapsamaya bağlıdır.

Tablo 1.4.4: Wi-Fi nesilleri ve mobil ağlar.

Tablodaki iki kısaltma kısa bir açıklama ister. **OFDMA** (dikgen frekans bölmeli çoklu erişim), bir kanalı küçük alt parçalara bölüp aynı anda birden çok cihaza dağıtır. Kalabalık bir sınıftaki onlarca telefon böylece sırayla beklemez; her biri aynı anda küçük bir pay alır. **MLO** (çok bağlantılı çalışma) ise bir cihazın aynı anda birden çok bantta bağlantı kurmasıdır; bantlardan biri sıkışınca veri ötekinden akar.

Wi-Fi standartları için verilen hızlar kuramsal en yüksek değerlerdir. Gerçek hız mesafeye, aradaki engellere, kanal genişliğine ve hem yönlendiricinin hem cihazın hangi standardı desteklediğine bağlıdır. Wi-Fi 7 destekli bir yönlendirici, Wi-Fi 5 destekli bir dizüstüyle ancak Wi-Fi 5 hızında konuşur.

Bantlar. Wi-Fi üç frekans bandında çalışabilir: 2,4 GHz, 5 GHz ve 6 GHz. Her bandın güçlü ve zayıf yanları vardır (Şekil 1.4.10).



Şekil 1.4.10: Wi-Fi bantlarının menzili. 6 GHz en hızlı ama en kısa menzilli banttır; 5 GHz orta menzillidir; 2,4 GHz en uzağa ulaşır.

- **2,4 GHz:** Menzil uzundur ve duvarlardan daha iyi geçer; hız tipik olarak düşüktür. Mikrodalga fırın, Bluetooth ve komşu ağlar aynı bandı kullandığı için kalabalıktır.
- **5 GHz:** Orta menzil, daha yüksek hız. Duvarlardan 2,4 GHz'e göre daha çok zayıflar.
- **6 GHz:** Kısa menzil, en yüksek hız potansiyeli ve az kalabalık (Wi-Fi 6E ve 7). Engellerden en çok etkilenen banttır.

Genel kural şudur: frekans yükseldikçe hız potansiyeli artar, menzil ve duvar geçişi azalır. Bu ödünleşimin fiziksel bir nedeni vardır. Elektromanyetik dalganın dalga boyu, ışık hızının frekansa bölümüdür: $\lambda = c / f$. Frekans arttıkça dalga boyu kısalır ve kısa dalgalar duvar, beton ve su gibi engellerde daha çok enerji kaybeder.

Örnek 1.4.3

Üç Wi-Fi bandının dalga boyları nedir? ($c \approx 3 \times 10^8$ m/s.)

1. 2,4 GHz: $\lambda = 3 \times 10^8 \div 2{,}4 \times 10^9 = 0{,}125$ m, yani yaklaşık 12,5 cm.
2. 5 GHz: $3 \times 10^8 \div 5 \times 10^9 = 0{,}06$ m, yani 6 cm.
3. 6 GHz: $3 \times 10^8 \div 6 \times 10^9 = 0{,}05$ m, yani 5 cm.

2,4 GHz'in dalgası 6 GHz'inkinin iki buçuk katı uzunluktadır. Bu fark, iki duvar arkadaki bir odada 2,4 GHz'in neden daha kararlı bağlantı verdiğini açıklar.

Pratik sonuç basittir: modemden uzak bir odada 2,4 GHz, yakın mesafede ise 5 ya da 6 GHz çoğu zaman daha uygundur. Birçok yönlendirici cihazı kendiliğinden en uygun banda yönlendirir.

1.4.1.9 Wi-Fi güvenliği

Kablolu bir ağa bağlanmak için kabloya fiziksel olarak erişmek gerekir. Kablosuz ağda ise sinyal duvarların ötesine, sokağa ve komşu dairelere yayılır. Bu yüzden kablosuz ağın trafiği şifrelenmeli ve ağa yalnız parolayı bilenler katılabilmelidir. Bu işi Wi-Fi güvenlik protokolleri yapar.

Protokol	Durum	Açıklama
WEP	Güvensiz	Çok eskidir; dakikalar içinde kırılabilir. Artık kullanılmamalıdır.
WPA / TKIP	Eski	Önerilmez; yalnız eski cihazlarla uyum için bulunur.
WPA2 (AES)	Yaygın	Güçlü ve uzun parola şarttır; zayıf parola tahmin edilebilir.
WPA3	Güncel	Zayıf parolaya karşı daha dayanıklıdır; 6 GHz bandında WPA3 (ya da parolasız ağlar için Enhanced Open) zorunludur. Cihaz desteği gerekir.

Tablo 1.4.5: Wi-Fi güvenlik protokolleri.



Şekil 1.4.11: Wi-Fi güvenlik protokollerinin sırası: soldan sağa eskiden yeniye ve zayıftan güçlüye WEP, WPA, WPA2, WPA3.

WPA2 ile WPA3 arasındaki önemli bir fark, parola tahminine karşı davranıştır. WPA2'de saldırgan, ağa bağlanan bir cihazın kısa el sıkışma trafiğini kaydedip parolayı kendi bilgisayarında, ağa hiç dokunmadan deneyebilir. Parola kısa ya da sözlükte geçen bir sözcükse kısa sürede bulunur. WPA3'te her tahmin için ağla yeniden konuşmak gerekir; bu yüzden çevrim dışı tahmin yapılamaz.

Açık (parolasız) ağlar. Kafe, havalimanı ya da otel ağlarında trafik şifresiz olabilir. Aynı ağdaki biri trafiği dinleyebilir ya da aynı adla sahte bir erişim noktası kurup kullanıcıları kendi cihazına çekebilir. Bu yüzden böyle ağlarda bankacılık ya da e-Devlet gibi hassas işlemler yalnız HTTPS ile, gerekirse VPN ile yapılmalıdır. HTTPS 1.4.4.2'de, VPN 1.4.7.8'de ele alınır.

Sık yapılan hata

Ağın adını gizlemenin ya da yalnız tanıdık cihazların MAC adreslerine izin vermenin ağı güvenli kıldığını düşünmek. Gizli ağ adı da MAC adresleri de havada açıkça görünür ve kolayca taklit edilir. Kablosuz ağı koruyan şey güncel bir protokol (WPA2 ya da WPA3) ve uzun, tahmin edilemez bir paroladır.

1.4.2 Protokoller, katmanlar ve paketler

İki bilgisayarın birbirini anlaması için ortak kurallara uyması gerekir. Ancak "bir web sayfasını getir" gibi tek bir iş bile çok sayıda alt işten oluşur: sayfa istenir, istek parçalara bölünür, parçalar numaralanır, her parça adreslenir, yol bulunur, bitler kabloya verilir, kaybolan parça yeniden gönderilir. Bu işlerin hepsini tek bir protokole yüklemek yerine her alt iş ayrı bir protokole verilir ve protokoller katmanlar hâlinde üst üste dizilir. Bu kesim önce temel protokolleri tanıtır, sonra katman modelini, paketlere bölmeyi ve paketlerin yönlendiricilerden geçişini adım adım izler.

1.4.2.1 Temel iletişim protokolleri

Ders boyunca adı sık geçecek altı protokol ailesi Tablo 1.4.6'da özetlenmiştir. Her birinin ayrıntısı notun ilerleyen kesimlerinde gelir; burada amaç, hangi işin hangi protokole düştüğünü baştan görmektir.

Protokol	Açılımı	Görevi
IP	internet protokolü (*Internet Protocol*)	Paketlerin ağlar arasında adreslenmesi ve yönlendirilmesi.
TCP	iletim denetim protokolü (*Transmission Control Protocol*)	Verinin sıralı ve güvenilir iletimi.
UDP	kullanıcı datagram protokolü (*User Datagram Protocol*)	Düşük gecikmeli, bağlantısız iletim.
DNS	alan adı sistemi (*Domain Name System*)	Alan adını IP adresine dönüştürme.
HTTP / HTTPS	köprü metni aktarım protokolü (*Hypertext Transfer Protocol*), güvenli sürümü	Web kaynaklarının aktarımı; HTTPS iletişimi şifreler.
SMTP / IMAP	basit posta aktarım protokolü / internet ileti erişim protokolü	E-posta gönderme (SMTP) ve okuma (IMAP).

Tablo 1.4.6: Temel iletişim protokolleri.

TCP ile UDP arasındaki seçim bir ödünleşimdir. TCP her paketin ulaştığını denetler, kaybolanı yeniden gönderir ve veriyi doğru sırayla teslim eder. Bunun bedeli ek mesajlar ve beklemedir. UDP ise paketi gönderir ve arkasına bakmaz: kayıp olursa telafi etmez. Bir dosya ya da web sayfası indirilirken tek bir baytın bile eksik olması kabul edilemez; bu yüzden TCP kullanılır. Canlı görüntülü görüşmede ise yarım saniye önce kaybolmuş bir görüntü karesini yeniden göndermenin anlamı yoktur; o anda yeni kare gelmiştir. Bu tür işlerde UDP tercih edilir. DNS sorguları da kısa oldukları için çoğunlukla UDP ile taşınır.

1.4.2.2 Protokoller katman katman çalışır

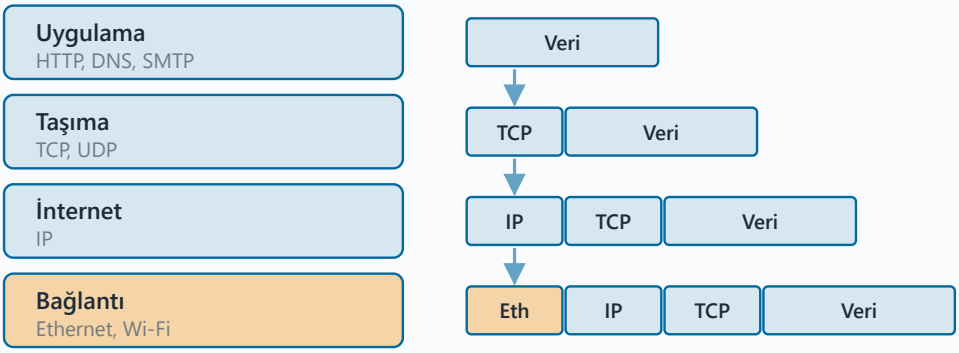
Bir mektubu postayla göndermeyi düşünelim. Mektubu yazan kişi yalnız içerikle ilgilenir. Zarfı kapatıp adresi yazan, mektubun içeriğini bilmez. Postane zarfı torbaya koyar ve torbanın üzerine hangi şehre gideceğini yazar. Kamyon şoförü yalnız torbaları taşır. Her aşama kendi işini yapar, bir üstündeki aşamanın içine bakmaz ve bir altındaki aşamanın nasıl çalıştığıyla uğraşmaz. Ağ protokolleri de aynı ilkeyle düzenlenir.

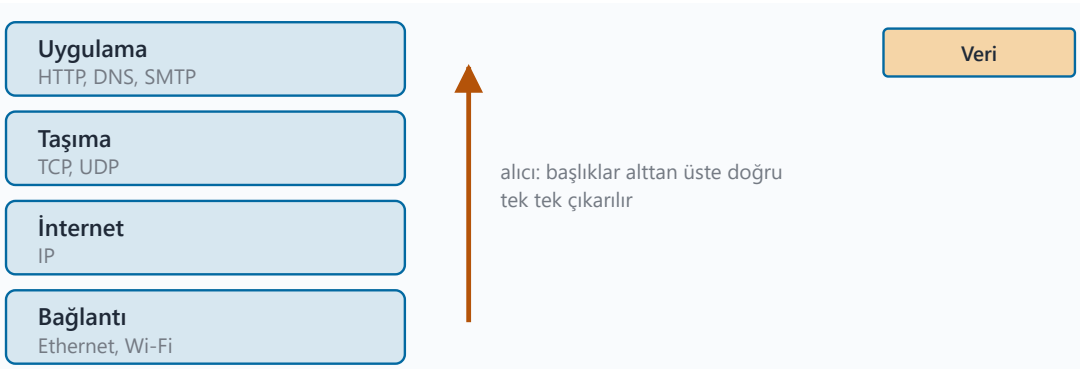
Tanım 1.4.6

İnternet protokolleri dört **katmanda** (*layer*) toplanır. **Uygulama katmanı** ne söyleneceğini belirler (HTTP, DNS, SMTP). **Taşıma katmanı** veriyi parçalara böler ve uçtan uca iletimi yönetir (TCP, UDP). **İnternet katmanı** paketleri adresler ve ağlar arasında yönlendirir (IP). **Bağlantı katmanı** bitleri aynı ağdaki bir sonraki cihaza fiziksel olarak iletir (Ethernet, Wi-Fi). Her katman, aşağıya gönderdiği veriye kendi **başlığını** (*header*) ekler; bu işleme **kapsülleme** denir.

Gönderen tarafta veri yukarıdan aşağıya iner ve her katmanda bir başlık kazanır. Şekil 1.4.12'nin üst kısmı bu birikimi gösterir:

1. **Uygulama katmanı.** Tarayıcı bir HTTP isteği üretir; örneğin "`GET /duyurular`". Bu istek, alt katmanlar açısından yalnızca "veridir".
2. **Taşıma katmanı.** Veri gerekirse parçalara bölünür ve her parçanın başına bir **TCP başlığı** eklenir. Başlıkta port numarası (verinin hangi uygulamaya gideceği) ve sıra numarası (parçanın bütün içindeki yeri) yazar. Kaybolan bir parça bu numara sayesinde yeniden istenebilir.
3. **İnternet katmanı.** TCP parçasının başına bir **IP başlığı** eklenir: kaynak ve hedef IP adresi. Yönlendiriciler yalnız bu başlığa bakarak yol seçer; içerideki TCP başlığını ya da HTTP isteğini okumaları gerekmez.
4. **Bağlantı katmanı.** En dışa çerçeve başlığı eklenir. Bu başlıkta yerel ağdaki bir sonraki cihazın **MAC adresi** bulunur. Bitler sonra kabloya ya da radyo dalgasına verilir.





Şekil 1.4.12: Kapsülleme. Üstte gönderen taraf: veri her katmanda bir başlık kazanır ve sonunda Eth | IP | TCP | Veri biçiminde gönderilir. Altta alıcı taraf: başlıklar alttan üste doğru tek tek çıkarılır ve veri uygulamaya ulaşır.

Alıcı tarafta işlem tersine döner. Bağlantı katmanı çerçeve başlığını okur ve çıkarır. İnternet katmanı IP başlığına bakar; hedef adres bu makineyse başlığı çıkarıp içeriği yukarı verir. Taşıma katmanı parçaları sıra numarasına göre dizer ve port numarasına bakarak verinin hangi uygulamaya ait olduğunu bulur. En sonunda web sunucusu, tarayıcının ürettiği isteğin aynısını alır.

Katman	Örnek protokoller	Başlıkta ne var?	Bu başlığı kim okur?
Uygulama	HTTP, DNS, SMTP	İsteğin kendisi (yöntem, kaynak yolu)	Karşı taraftaki uygulama
Taşıma	TCP, UDP	Kaynak ve hedef port, sıra numarası	Uçlardaki işletim sistemleri
İnternet	IP	Kaynak ve hedef IP adresi, TTL	Yoldaki her yönlendirici
Bağlantı	Ethernet, Wi-Fi	Kaynak ve hedef MAC adresi	Aynı yerel ağdaki bir sonraki cihaz

Tablo 1.4.7: Katmanların başlıkları ve kullananlar.

Katmanlı yapının en büyük kazancı bağımsızlıktır. Tarayıcı, verinin fiberle mi Wi-Fi ile mi gittiğini bilmek zorunda değildir. Yönlendirici, paketin içinde bir web isteği mi yoksa bir e-posta mı olduğunu bilmek zorunda değildir. Bir katmanda yeni bir teknoloji çıktığında (örneğin Wi-Fi 7) öteki katmanlarda hiçbir şey değişmez.

Kapsüllemenin bir bedeli de vardır: her başlık birkaç düzine bayt yer kaplar. Asıl veri küçükse başlıkların payı büyür.

Örnek 1.4.4

Tipik başlık boyları şöyledir: TCP 20 bayt, IPv4 20 bayt, Ethernet 18 bayt (14 bayt başlık ve 4 bayt hata denetimi). 100 baytlık bir HTTP isteği ve 1460 baytlık bir veri parçası için kablodaki toplam boy ve verimlilik nedir?

1. Başlıkların toplamı: 20 + 20 + 18 = 58 bayt.
2. 100 baytlık istek: kablodaki boy 100 + 58 = 158 bayt. Verimlilik 100 ÷ 158 ≈ %63,3.
3. 1460 baytlık parça: 1460 + 58 = 1518 bayt. Verimlilik 1460 ÷ 1518 ≈ %96,2.

Küçük iletilerde kablodaki baytların üçte birinden fazlası başlıktır. Büyük dosya aktarımında ise başlıkların payı %4'ün altına iner. 1518 bayt, Ethernet çerçevesinin olağan en büyük boyudur; bu yüzden TCP verisi çoğu zaman 1460 baytlık parçalarla taşınır.

Sık yapılan hata

Yönlendiricinin paketin içindeki web sayfasını okuyup ona göre yol seçtiğini düşünmek. Yönlendirici yalnız IP başlığına bakar. Paketin içinde ne olduğu yol seçimini etkilemez; HTTPS kullanıldığında içerik zaten şifrelidir ve yönlendirici onu okuyamaz.

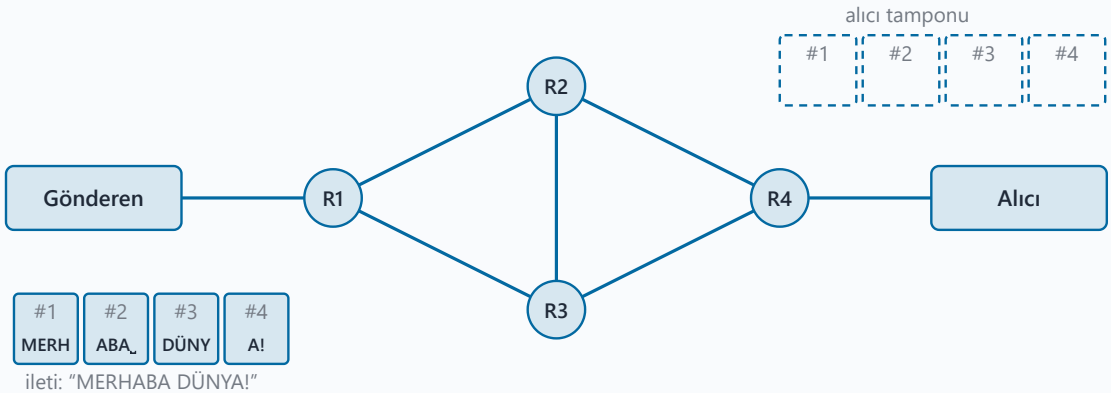
1.4.2.3 Veri paketlere bölünür

Büyük bir veri tek parça hâlinde gönderilmez. Bunun iki nedeni vardır. Birincisi, uzun bir aktarım hattı bir süre tek başına işgal ederdi; başka kimse araya giremezdi. İkincisi, yolda tek bir bit bozulduğunda bütün verinin yeniden gönderilmesi gerekirdi. Veri küçük parçalara bölündüğünde hat birçok kullanıcı arasında paylaşılır ve bozulan yalnız tek bir parça yeniden gönderilir.

Tanım 1.4.7

Paket (*packet*), ağda bağımsız olarak taşınan veri parçasıdır. Her paket kendi başlığını taşır: kaynak adresi, hedef adresi ve sıra numarası. TCP'nin taşıdığı parçaya **bölüt** (*segment*) de denir. Verinin paketlere bölünerek iletildiği bu düzene **paket anahtarlama** adı verilir.

Örnekte "MERHABA DÜNYA!" iletisi gönderiliyor. İleti boşluk ve ünlemle birlikte 14 karakterdir ve dörder karakterlik parçalara bölünür: MERH, ABA␣, DÜNY, A! (␣ boşluğu gösterir). Gönderen ile alıcı arasında dört yönlendirici vardır: R1, R2, R3 ve R4. R1'den R4'e iki yol gider: biri R2 üzerinden, öteki R3 üzerinden (Şekil 1.4.13).



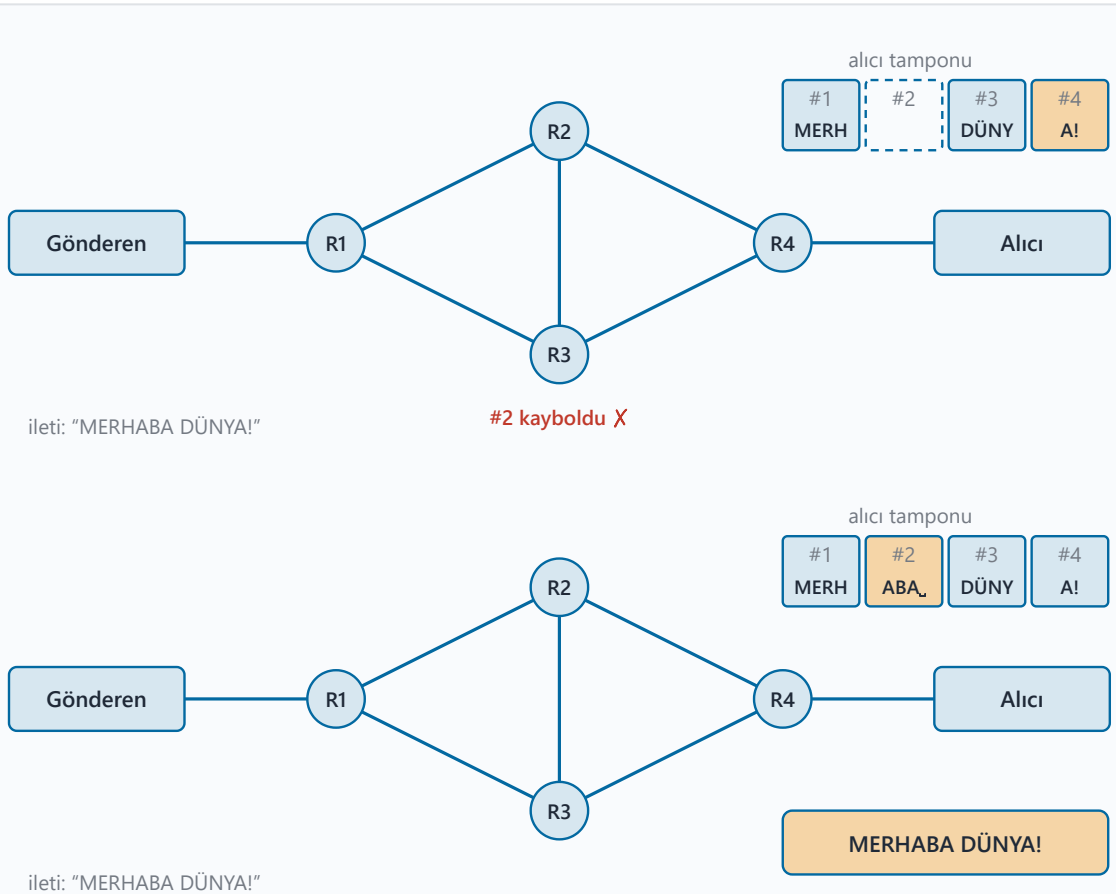
Şekil 1.4.13: "MERHABA DÜNYA!" iletisi dört pakete bölünmüştür. Sağ üstte alıcının tamponu boştur; paketler

geldikçe sıra numaralarına göre bu tampondaki yerlerine yerleşir.

Paketlerin yolculuğu Tablo 1.4.8'de adım adım izlenmiştir.

Adım	Olay	Alıcının tamponu (#1 #2 #3 #4)
1	İleti dört pakete bölünür; her başlıkta sıra numarası, kaynak ve hedef adresi vardır.	— — — —
2	#1 ve #3, R1 → R2 → R4 yolundan gider.	— — — —
3	#2 ve #4, R1 → R3 → R4 yolundan gider. Yönlendiriciler her paket için o anki en uygun yolu seçer.	MERH — DÜNY —
4	Paketler sırasız gelir: önce #1 ve #3, sonra #4. #2 yoğun bir yönlendiricide kaybolur.	MERH — DÜNY A!
5	TCP eksik sıra numarasını fark eder; #2 yeniden gönderilir (bu kez de R3 yolundan).	MERH — DÜNY A!
6	#2 gelir; alıcı paketleri sıra numarasına göre dizer ve iletiyi birleştirir.	MERH ABA_ DÜNY A!

Tablo 1.4.8: “MERHABA DÜNYA!” iletisinin paketlerinin izlemesi.



Şekil 1.4.14: Üstte 4. adım: #1, #3 ve #4 gelmiştir, #2 yolda kaybolmuştur. Altta son adım: yeniden gönderilen #2 de gelmiş ve ileti “MERHABA DÜNYA!” olarak birleştirilmiştir.

Örnekten çıkan üç sonuç önemlidir. Birincisi, aynı iletinin paketleri farklı yollardan gidebilir. İkincisi, paketler gönderildikleri sırayla gelmek zorunda değildir: #4, #2'den önce gelmiştir. Üçüncüsü, kayıp bir istisna değil, ağın olağan bir durumudur: yoğun bir yönlendiricinin tamponu dolduğunda yeni gelen paketler atılır. Bütün bu düzensizliğe rağmen uygulama iletiyi eksiksiz ve doğru sırayla alır, çünkü sıralama ve yeniden gönderme işini TCP üstlenir.

Sık yapılan hata

Paketlerin bir boru içindeki su gibi tek bir yoldan ve sırayla aktığını düşünmek. IP her paketi ayrı ayrı yönlendirir; yol ve varış sırası paketten pakete değişebilir. Sıralı teslim IP'nin değil, TCP'nin sağladığı bir hizmettir.

1.4.2.4 Paketin yönlendiricilerden geçişi

Bir paket, hedefine ulaşana kadar birçok yönlendiriciden geçer. Her yönlendirici, paketi bir sonraki yönlendiriciye teslim eder; bu tek adıma **atlama** (*hop*) denir. Hiçbir yönlendirici yolun tamamını bilmez. Her biri yalnız "bu hedefe giden paketi hangi komşuma vermeliyim?" sorusunu yanıtlar. Yanıt, yönlendiricinin **yönlendirme tablosunda** yazılıdır.

Tanım 1.4.8

Yönlendirme tablosu (*routing table*), her satırında bir adres bloğu ve o bloğa giden paketlerin gönderileceği sonraki atlamayı tutan tablodur. Adres bloğu `203.0.113.0/24` biçiminde yazılır: eğik çizgiden sonraki sayı, adresin baştan kaç bitinin bloğu belirlediğini gösterir. Bu bitlere **önek** (*prefix*) denir. `0.0.0.0/0` satırı önek uzunluğu sıfır olan, yani her adresi kapsayan **varsayılan yoldur** (*default route*).

Önek ne demektir? Bir IPv4 adresi 32 bittir (1.4.3.1). `/24`, adresin ilk 24 bitinin, yani ilk üç sayının sabit olduğunu söyler. `203.0.113.0/24` bloğu 203.0.113.0'dan 203.0.113.255'e kadar $2^8 = 256$ adresi kapsar. `203.0.0.0/16` ise yalnız ilk iki sayıyı sabitler ve $2^{16} = 65.536$ adresi kapsar. Önek ne kadar uzunsa blok o kadar küçük, yani o kadar *özeldir*.

Tanım 1.4.9

En uzun önek eşleşmesi (*longest prefix match*): Hedef adresi kapsayan birden çok satır varsa yönlendirici, öneki en uzun olan, yani en özel satırı seçer.

Her yönlendiricinin bir paketi aldığında izlediği algoritma kısadır:

```
paketi al, hedef IP'ye bak
TTL = TTL − 1;  0 olursa at, kaynağa bildir
tablodan hedefi kapsayan satırları bul
en uzun önekli (en özel) satırı seç
paketi o satırın sonraki atlamasına gönder
```

İkinci satırdaki **TTL** (*time to live*, yaşam süresi) IP başlığındaki bir sayaçtır. Gönderen onu bir başlangıç değerine (örneğin 64) ayarlar; her yönlendirici paketi iletmeden önce TTL'yi 1 azaltır. TTL sıfıra inerse paket atılır ve gönderene "süre aşıldı" iletisi döner. Bu kural, yanlış ayarlanmış yönlendiriciler arasında bir döngüye giren paketin ağda sonsuza dek dolaşmasını önler.

Bir örnek izleyelim. Evdeki dizüstü (192.168.1.10), **203.0.113.10** adresli bir sunucuya paket gönderiyor. Yol üzerinde dört yönlendirici var: evdeki R0, İSS'nin R1'i, R2 ve sunucunun ağına bağlı R3. R1'in ve R2'nin başka komşuları da var (R4, R5, R6), ama bu paket için kullanılmıyorlar.

Yönlendirici	Tablo satırları (önek → sonraki)	Hedefi kapsayan satırlar	Seçilen
R0 (ev)	192.168.1.0/24 → yerel ağ 0.0.0.0/0 → R1 (İSS)	yalnız /0	0.0.0.0/0 → R1
R1 (İSS)	203.0.113.0/24 → R2 198.51.100.0/24 → R4 0.0.0.0/0 → R5	/24 ve /0	203.0.113.0/24 → R2
R2	203.0.0.0/16 → R6 203.0.113.0/24 → R3	/16 ve /24	203.0.113.0/24 → R3
R3	203.0.113.0/24 → doğrudan bağlı	/24	sunucuya teslim

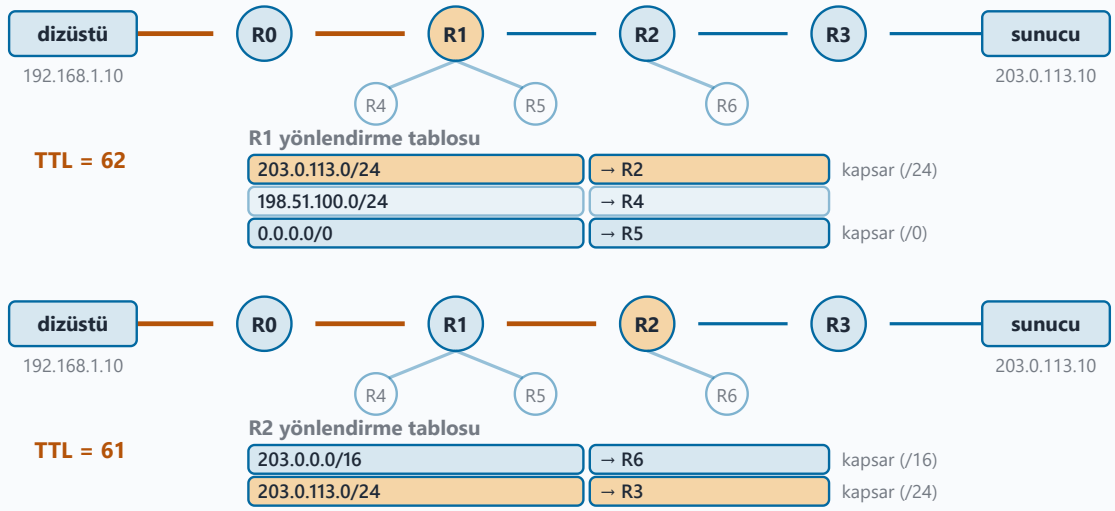
Tablo 1.4.9: Yönlendiricilerin tabloları ve 203.0.113.10 hedefi için seçimler.

Adım	Konum	TTL	Seçilen satır	Sonraki
1	dizüstü	64	—	R0
2	R0	63	0.0.0.0/0	R1 (İSS)
3	R1	62	203.0.113.0/24	R2
4	R2	61	203.0.113.0/24	R3
5	R3	60	203.0.113.0/24	doğrudan bağlı
6	sunucu	60	—	—

Tablo 1.4.10: Paketin yolculuğunun izlemesi.

Adımları tek tek açıklayalım. Dizüstü, hedefin kendi yerel ağında (192.168.1.0/24) olmadığını görür ve paketi **varsayılan ağ geçidine** (*default gateway*), yani ev yönlendiricisi R0'a verir. R0 TTL'yi 63'e indirir. Hedef, R0'ın tablosunda yalnız varsayılan yola uyar; paket İSS'nin R1'ine gider. NAT kullanılıyorsa paketin kaynak adresi de burada değişir (1.4.3.2).

R1'de TTL 62 olur ve iki satır hedefi kapsar: 203.0.113.0/24 ve 0.0.0.0/0. En uzun önek kazanır; /24 daha özeldir ve paket R2'ye gider. R4 ve R5 yolları kullanılmaz. R2'de TTL 61 olur. Burada da iki satır hedefi kapsar: 203.0.0.0/16 ve 203.0.113.0/24. Yine /24 daha uzundur ve paket R3'e gider. /16 seçilseydi paket yanlış yöne, R6'ya giderdi. R3'te TTL 60'tır; hedef ağ R3'e doğrudan bağlıdır ve paket sunucuya teslim edilir.



Şekil 1.4.15: Üstte R1'in kararı: TTL 62, iki satır hedefi kapsar ve en uzun önekli 203.0.113.0/24 satırı R2'yi gösterir. Altta R2'nin kararı: TTL 61, /16 ve /24 satırlarından /24 seçilir ve paket R3'e gider.

Örnek 1.4.5

203.0.113.10 adresinin 203.0.0.0/16 ve 203.0.113.0/24 bloklarına girdiğini bit düzeyinde gösterelim.

1. Adresi ikiliye çevirelim (yöntem 1.4.3.1'de): 203 = 11001011, 0 = 00000000, 113 = 01110001, 10 = 00001010. Adres: `11001011.00000000.01110001.00001010`.
2. 203.0.0.0/16 bloğunun ilk 16 biti: `11001011.00000000`. Adresin ilk 16 biti aynıdır: adres bu bloktadır.
3. 203.0.113.0/24 bloğunun ilk 24 biti: `11001011.00000000.01110001`. Adresin ilk 24 biti de aynıdır: adres bu bloktadır da.
4. İki eşleşme vardır; 24 bitlik eşleşme 16 bitlik eşleşmeden uzundur. Seçilen satır 203.0.113.0/24'tür.

Aynı karşılaştırmayı 203.0.200.5 adresi için yaparsak üçüncü sayı 200 = 11001000 olduğundan /24 bloğu tutmaz; yalnız /16 eşleşir ve R2 bu paketi R6'ya gönderir.

traceroute. TTL mekanizmasının yan ürünü, yoldaki yönlendiricileri keşfetmeyi sağlayan bir araçtır. Windows'ta `tracert`, Linux ve macOS'te `traceroute` adıyla çalışır. Araç önce TTL'si 1 olan bir paket gönderir: ilk yönlendirici TTL'yi sıfıra indirir, paketi atar ve "süre aşıldı" iletisi döndürür. İletinin geldiği adres, yoldaki ilk yönlendiricidir. Sonra TTL 2, 3, … diye artırılır ve yoldaki yönlendiriciler sırayla listelenir. Her satırda o yönlendiriciye gidiş-dönüş süresi de yazar.

Sık yapılan hata

Yönlendirme tablosunun yukarıdan aşağı okunup ilk tutan satırın seçildiğini sanmak. Yönlendiricide satırların sırası önemli değildir: hedefi kapsayan bütün satırlar arasından öneki en uzun olan seçilir. (Güvenlik duvarı kurallarında ise durum farklıdır; orada ilk eşleşen kural uygulanır: 1.4.7.9.)

1.4.3 Adresleme: IP, NAT, DNS, URL ve port

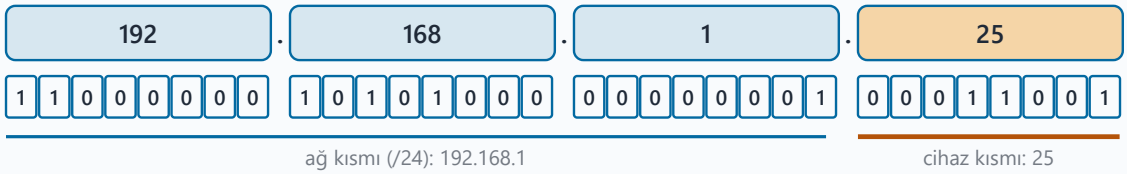
Bir paketin doğru yere ulaşması için hedefin kesin bir adresi olmalıdır. İnternette bu iş birkaç katmanlı bir adres sistemiyle yapılır. IP adresi hangi cihaza gidileceğini, port numarası o cihazdaki hangi programa gidileceğini söyler. İnsanlar sayılardan oluşan adresleri akılda tutamadığı için alan adları kullanılır; DNS bu adları IP adresine çevirir. URL ise bütün bu bilgileri, istenen kaynağın yoluyla birlikte tek bir satırda toplar. Bu kesim bu adresleri sırayla ele alır.

1.4.3.1 IP adresi

Tanım 1.4.10

IP adresi (*IP address*), bir ağ arayüzünü tanımlayan ve paketlerin hedefe yönlendirilmesini sağlayan sayısal adrestir. **IPv4** adresi 32 bittir ve noktayla ayrılmış dört sayı olarak yazılır; her sayı 0 ile 255 arasındadır. Bu yazıma **noktalı onluk gösterim** (*dotted decimal notation*), her sayıya da **sekizli** (*octet*) denir. **IPv6** adresi 128 bittir ve iki nokta üst üsteyle ayrılmış onaltılık gruplar hâlinde yazılır.

Tanımda "cihaz" değil "ağ arayüzü" denmesi bilinçlidir. Bir dizüstünün hem kablolu hem kablosuz ağ kartı varsa iki ayrı IP adresi olabilir. Bir yönlendiricinin de bağlı olduğu her ağ için ayrı bir adresi vardır.



Şekil 1.4.16: 192.168.1.25 adresinin dört sekizlisi ve her birinin 8 bitlik ikili karşılığı. /24 ağında ilk üç sekizli ağı (192.168.1), son sekizli cihazı (25) belirtir.

Neden 0–255? Her sekizli 8 bittir. 8 bitle $2^8 = 256$ farklı değer yazılabilir: en küçüğü 00000000 = 0, en büyüğü 11111111 = 255. Dört sekizli toplam $4 \times 8 = 32$ bit eder ve $2^{32} = 4.294.967.296$ farklı adres verir. Bu sayı ilk bakışta büyük görünür, ama dünya nüfusu 8 milyarı aşmıştır ve kişi başına bir adres bile düşmez. Üstelik bir kişinin telefonu, dizüstüsü, televizyonu ve akıllı saati ayrı adresler ister. IPv4 adresleri bu yüzden tükenmiştir; NAT (1.4.3.2) ve IPv6 bu sorunun iki çözümüdür.

Onluktan ikiliye çevirme. Bir sekizliyi ikiliye çevirmenin en kolay yolu, bit ağırlıklarını büyükten küçüğe denemektir. Sekiz bitin ağırlıkları soldan sağa 128, 64, 32, 16, 8, 4, 2, 1'dir. Her adımda sayı ağırlıktan büyük ya da ona eşitse o bit 1 yazılır ve ağırlık sayıdan çıkarılır; değilse bit 0 yazılır ve sayı değişmez.

Örnek 1.4.6

192.168.1.25 adresindeki 168 ve 25 sekizlilerini ikiliye çevirelim.

Ağırlık	168 için: sayı ≥ ağırlık?	Bit	Kalan	25 için: sayı ≥ ağırlık?	Bit	Kalan
128	168 ≥ 128 evet	1	40	25 ≥ 128 hayır	0	25
64	40 ≥ 64 hayır	0	40	25 ≥ 64 hayır	0	25
32	40 ≥ 32 evet	1	8	25 ≥ 32 hayır	0	25
16	8 ≥ 16 hayır	0	8	25 ≥ 16 evet	1	9
8	8 ≥ 8 evet	1	0	9 ≥ 8 evet	1	1
4	0 ≥ 4 hayır	0	0	1 ≥ 4 hayır	0	1
2	0 ≥ 2 hayır	0	0	1 ≥ 2 hayır	0	1
1	0 ≥ 1 hayır	0	0	1 ≥ 1 evet	1	0

168 = `10101000` (denetim: 128 + 32 + 8 = 168) ve 25 = `00011001` (denetim: 16 + 8 + 1 = 25). 192 = 128 + 64 olduğundan `11000000`, 1 ise `00000001`'dir. Adresin tamamı: `11000000.10101000.00000001.00011001`.

Her sekizli tam 8 bitle yazılır; baştaki sıfırlar atılmaz. 25'in ikili karşılığı “11001” diye değil, “00011001” diye yazılır. Aksi hâlde 32 bitlik adresin hangi bitinin hangi sekizliye ait olduğu kaybolur ve önek karşılaştırması yanlış çıkar.

IPv6. 128 bitlik IPv6 adresi $2^{128} \approx 3{,}4 \times 10^{38}$ farklı değer alabilir; bu, IPv4 adres sayısının yaklaşık 7,9 × 10^{28} katıdır. Adres sekiz gruba bölünür; her grup 16 bittir ve dört onaltılık rakamla yazılır (her onaltılık rakam 4 bittir). Uzun adresleri kısaltmak için iki kural vardır: bir grubun baştaki sıfırları yazılmayabilir ve art arda gelen sıfır grupları bir kez `::` ile gösterilebilir.

Örnek 1.4.7

`2001:db8::25` adresinin açık yazımı nedir?

1. Görünen gruplar üç tanedir: `2001`, `db8` ve `25`. Toplam grup sayısı sekiz olmalıdır; `::` eksik 8 − 3 = 5 sıfır grubunun yerini tutar.
2. Her grup dört rakama tamamlanır: `db8` → `0db8`, `25` → `0025`.
3. Açık yazım: `2001:0db8:0000:0000:0000:0000:0000:0025`.

`::` bir adreste yalnız bir kez kullanılabilir; iki kez kullanılsaydı hangi tarafta kaç sıfır grubu olduğu belirsiz kalırdı.

Sık yapılan hata

Bir sekizliye 256 ya da daha büyük bir sayı yazmak. 192.168.1.256 geçerli bir IPv4 adresi değildir; 8 bit en çok 255'e kadar sayabilir. Benzer biçimde IPv6 adresinde “g” ya da “z” gibi harfler bulunamaz: onaltılık rakamlar yalnız 0–9 ve a–f'dir.

1.4.3.2 Yerel ve genel IP adresi: NAT

Evdeki bir dizüstünün ağ ayarlarına bakıldığında çoğu zaman 192.168.1.10 gibi bir adres görülür. Aynı dizüstüyle “IP adresim ne?” diye soran bir web sitesine girildiğinde ise bambaşka bir adres çıkar, örneğin 203.0.113.7. İki adresin farklı olması bir hata değildir; aradaki dönüşümü yönlendirici yapar.

Tanım 1.4.11

Yerel IP adresi (*private IP address*), yalnız yerel ağ içinde geçerli olan ve internette yönlendirilmeyen adrestir. Bu iş için üç blok ayrılmıştır: 10.0.0.0/8, 172.16.0.0/12 ve 192.168.0.0/16. **Genel IP adresi** (*public IP address*) ise internette tek olan ve doğrudan ulaşılabilen adrestir. **NAT** (*network address translation*, ağ adresi çevirisi), yönlendiricinin yerel adresleri kendi genel adresiyle değiştirmesidir. Port numaralarını da değiştiren yaygın biçimine **NAPT** denir.

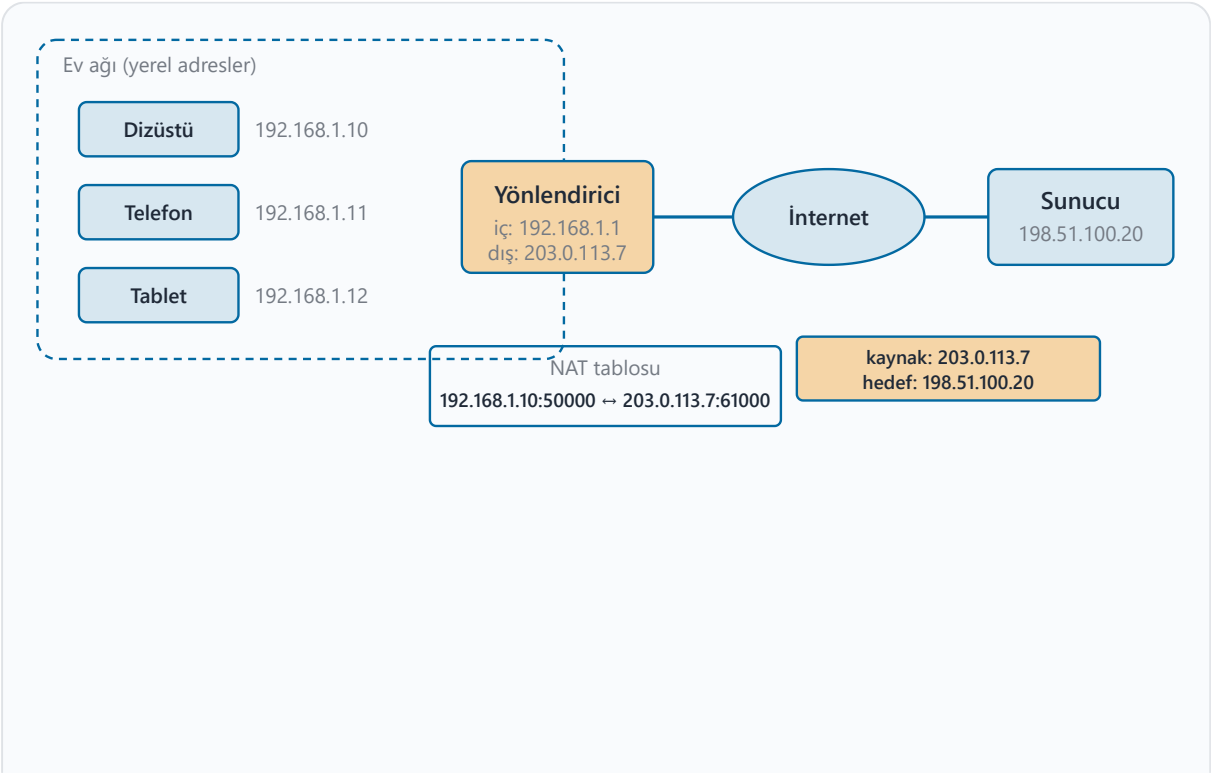
Yerel adres blokları her evde, her okulda yeniden kullanılır. Giresun'daki bir evde de İstanbul'daki bir evde de 192.168.1.10 adresli bir cihaz olabilir. Bu adresler ağın içinde benzersizdir ama dünyada benzersiz değildir. Bu yüzden internete çıkan her paketin kaynak adresi, yönlendiricinin genel adresiyle değiştirilmek zorundadır.

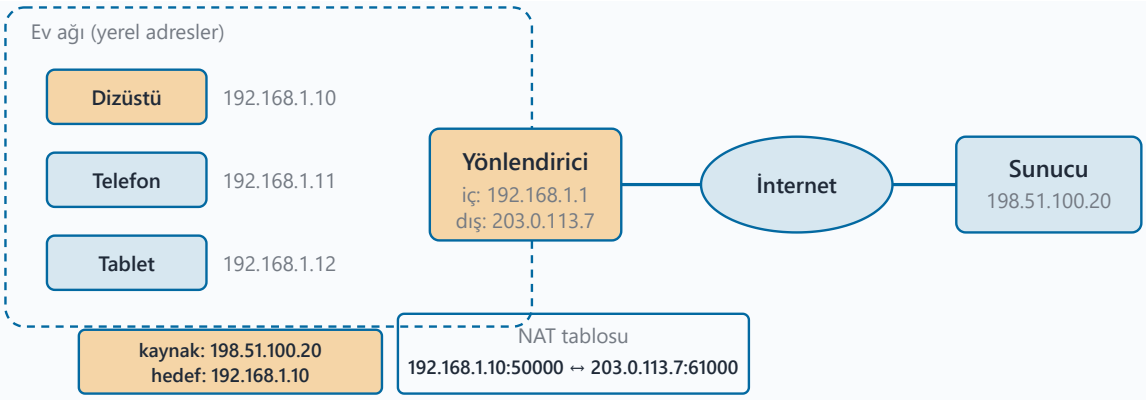
Örnekteki evde üç cihaz var: dizüstü (192.168.1.10), telefon (192.168.1.11) ve tablet (192.168.1.12). Yönlendiricinin iç adresi 192.168.1.1, dış (genel) adresi 203.0.113.7'dir. Dizüstü 198.51.100.20 adresli bir sunucuya bağlanıyor.

Adım	Konum	Kaynak	Hedef	NAT tablosu
1	Ev ağı (gidiş)	192.168.1.10:50000	198.51.100.20	boş
2	Yönlendiriciden çıkış	203.0.113.7:61000	198.51.100.20	192.168.1.10:50000 ↔ 203.0.113.7:61000
3	İnternet (dönüş)	198.51.100.20	203.0.113.7:61000	aynı
4	Ev ağı (dönüş)	198.51.100.20	192.168.1.10:50000	aynı

Tablo 1.4.11: NAT ile adreslerin değişimi (dizüstü 192.168.1.10, genel adres 203.0.113.7, sunucu 198.51.100.20).

1. Dizüstü paketi gönderir. Kaynak adresi yerel IP'dir (192.168.1.10) ve bu özel adres internette yönlendirilmez.
2. Yönlendirici kaynak adresini kendi genel adresiyle (203.0.113.7), kaynak portunu da kendi seçtiği bir portla (61000) değiştirir. Yanıtın kime ait olduğunu sonra bulabilmek için bu eşlemeyi **NAT tablosuna** yazar.
3. Sunucu yalnız 203.0.113.7 adresini görür ve yanıtı bu adrese, 61000 numaralı porta gönderir. Evdeki bütün cihazlar dışarıdan tek bir adres gibi görünür.
4. Yönlendirici yanıtın hedef portuna (61000) bakar; NAT tablosuna göre bu bağlantı dizüstüne aittir. Hedefi 192.168.1.10:50000'e çevirip paketi iletir.





Şekil 1.4.17: Üstte gidiş: yönlendirici kaynağı 203.0.113.7 yapar ve eşlemeyi NAT tablosuna yazar. Altta dönüş: tablodaki eşlemeye bakılarak hedef 192.168.1.10'a çevrilir.

Port numarası neden değiştirilir? Evde telefon ile dizüstü aynı anda aynı sunucuya bağlanabilir ve ikisi tesadüfen aynı kaynak portunu seçmiş olabilir. Yönlendirici her bağlantıya farklı bir dış port verirse dönen yanıtları karıştırmadan ayırabilir. Port numarası 16 bittir ve 2^{16} = 65.536 değer alır; bu yüzden tek bir genel adresin arkasında aynı anda on binlerce bağlantı yürüyebilir.

Örnek 1.4.8

Şu adreslerden hangileri yerel (özel) adrestir? 10.5.5.5 · 172.31.255.1 · 172.32.0.1 · 203.0.113.7 · 192.168.1.10.

1. 10.5.5.5: ilk sekizli 10'dur; 10.0.0.0/8 bloğundadır. **Özel.**
2. 172.16.0.0/12 bloğu 172.16.0.0'dan 172.31.255.255'e kadar uzanır (/12, ilk sekizlinin tamamını ve ikinci sekizlinin ilk 4 bitini sabitler; ikinci sekizli 16 ile 31 arasında değişebilir). 172.31.255.1 bu aralıktadır: **özel.**
3. 172.32.0.1'in ikinci sekizlisi 32'dir; aralığın dışındadır: **genel.**
4. 203.0.113.7 üç bloğun hiçbirinde değildir: **özel değildir.**
5. 192.168.1.10, 192.168.0.0/16 bloğundadır: **özel.**

Not: 203.0.113.0/24 ve 198.51.100.0/24 blokları belgelerde ve örneklerde kullanılmak üzere ayrılmıştır; gerçek bir cihaza verilmez. Bu nottaki örnek adreslerin bu bloklardan seçilmesi bu yüzdendir.

Sık yapılan hata

Bilgisayarın ağ ayarlarında görülen 192.168.x.x adresinin internette görünen adres olduğunu düşünmek. Bu adres yalnız ev ağında geçerlidir. Siteler yönlendiricinin genel adresini görür; aynı evdeki herkes dışarıya aynı genel adresle çıkar.

1.4.3.3 Alan adı ve DNS

Bir web sitesine 203.0.113.10 gibi bir sayıyla bağlanmak mümkündür, ama bu sayıları akılda tutmak zordur. Üstelik bir kurum sunucusunu değiştirdiğinde IP adresi de değişebilir. Bu iki sorunu çözmek için insanların okuyabileceği adlar kullanılır.

Tanım 1.4.12

Alan adı (*domain name*), insanlar için okunabilir bir addır; örneğin `www.ornek.edu.tr`. **DNS** (*Domain Name System*, alan adı sistemi), alan adını bağlantı kurulabilecek IP adresine çeviren, dünyaya dağılmış bir ad veri tabanıdır. Bu çeviri işlemine **DNS çözümlemesi** (*DNS resolution*) denir.

> `www.ornek.edu.tr` → DNS sorgusu → `203.0.113.10` → web sunucusu

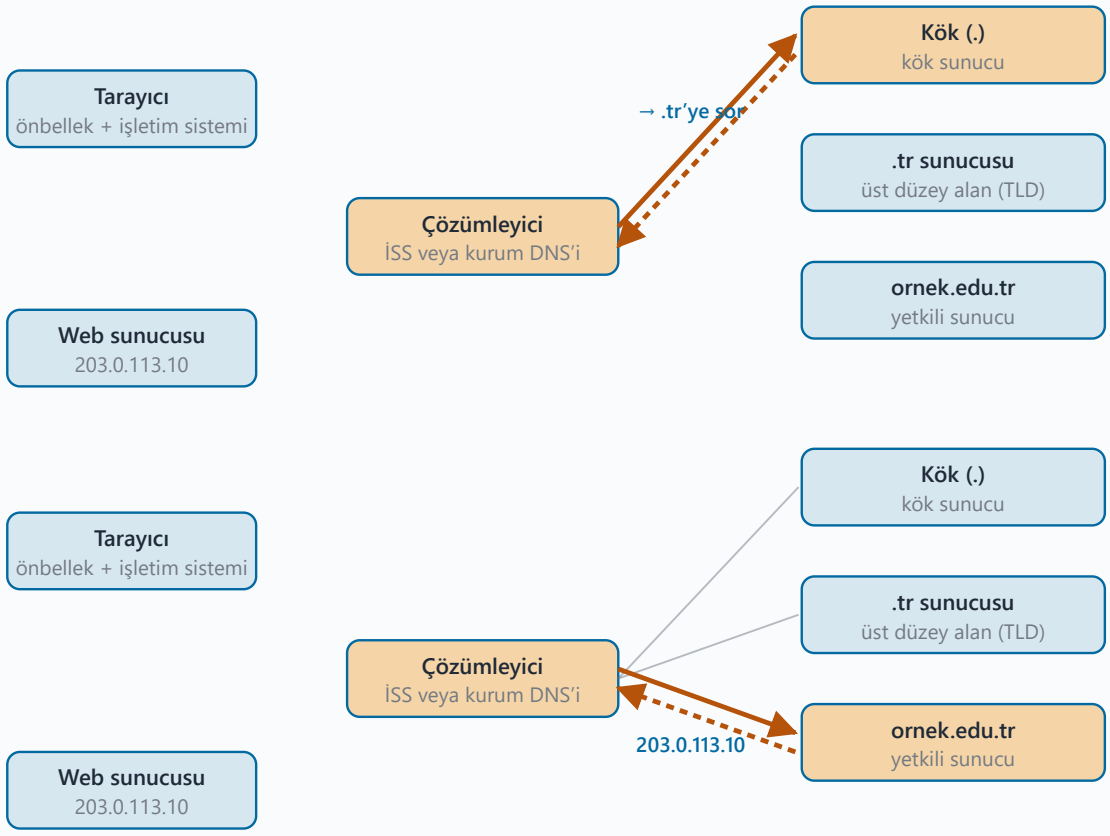
DNS'i telefon rehberine benzetmek yaygındır: kişinin adı bilinir, rehber numarasını verir. Ama DNS tek bir rehber değildir. Dünyadaki bütün alan adlarını tek bir sunucuda tutmak hem çok büyük bir yük olurdu hem de o sunucu çöktüğünde bütün internet dururdu. DNS bu yüzden hiyerarşik ve dağıtık kurulmuştur: her sunucu adın yalnız bir kısmından sorumludur.

1.4.3.4 DNS çözümlemesi adım adım

Çözümlemede dört tür sunucu rol alır. **DNS çözümleyicisi** (*DNS resolver*), kullanıcı adına bütün soruları soran ve sonucu getiren sunucudur; çoğunlukla İSS'nin ya da kurumun sunucusudur. **Kök sunucu** (*root server*), hiyerarşinin en tepesidir ve üst düzey alanların (.tr, .com, .org…) sunucularını bilir. **Üst düzey alan sunucusu** (örneğin .tr sunucusu), kendi altındaki alanların yetkili sunucularını bilir. **Yetkili sunucu** (*authoritative server*) ise belirli bir alanın kayıtlarını tutan ve son yanıtı veren sunucudur.

Bir kullanıcının `www.ornek.edu.tr` adresini ilk kez açtığını, hiçbir önbellekte kayıt olmadığını varsayalım.

1. Kullanıcı adres çubuğuna `www.ornek.edu.tr` yazar. Bağlanmak için bu adın IP adresi gerekir.
2. Tarayıcı ve işletim sistemi önce kendi önbelleklerine bakar: kayıt yoktur.
3. Soru, ağ ayarlarındaki DNS çözümleyicisine gönderilir.
4. Çözümleyici en tepeden başlar. Kök sunucu adı bilmez, ama ".tr sunucularına sor" der.
5. .tr sunucusu da tam adresi bilmez: "ornek.edu.tr'nin yetkili sunucusuna sor" der.
6. Yetkili sunucu kaydı bilir ve yanıtlar: `203.0.113.10`.
7. Çözümleyici yanıtı tarayıcıya iletir ve kaydı belli bir süre önbelleğinde saklar; aynı soru tekrar gelirse hemen yanıtlar.
8. Tarayıcı artık IP adresini bildiği için 203.0.113.10 adresindeki web sunucusuna bağlanır.



Şekil 1.4.18: DNS çözümlemesinin iki adımı. Üstte çözümleyici kök sunucuya sorar ve "tr'ye sor" yönlendirmesini alır. Altta yetkili sunucu kaydı bilir ve 203.0.113.10 yanıtını döndürür.

Dikkat edilirse kök ve .tr sunucuları asıl yanıtı vermez; yalnız bir sonraki adımda kime sorulacağını söyler. Bu yanıtlara **yönlendirme** denir. Çözümleyici ad hiyerarşisinde yukarıdan aşağı iner ve her adımda biraz daha özel bir sunucuya ulaşır. Kullanıcının bilgisayarı ise bu yolculuğun hiçbir adımını görmez: tek bir soru sorar ve tek bir yanıt alır.

1.4.3.5 Çözümleyicinin önbelleği

Her ad için kök sunucudan başlayıp üç ayrı sunucuya sormak hem yavaş hem de kök sunucular için ağır bir yük olurdu. Çözümleyiciler bu yüzden öğrendikleri her bilgiyi bir süre **önbellekte** (*cache*) tutar. Yalnız son yanıt değil, ara yönlendirmeler de saklanır: ".tr sunucusu şudur", "ornek.edu.tr'nin yetkili sunucusu şudur". Her kaydın ne kadar saklanacağını, kaydı veren yetkili sunucu bir **TTL** değeriyle belirler.

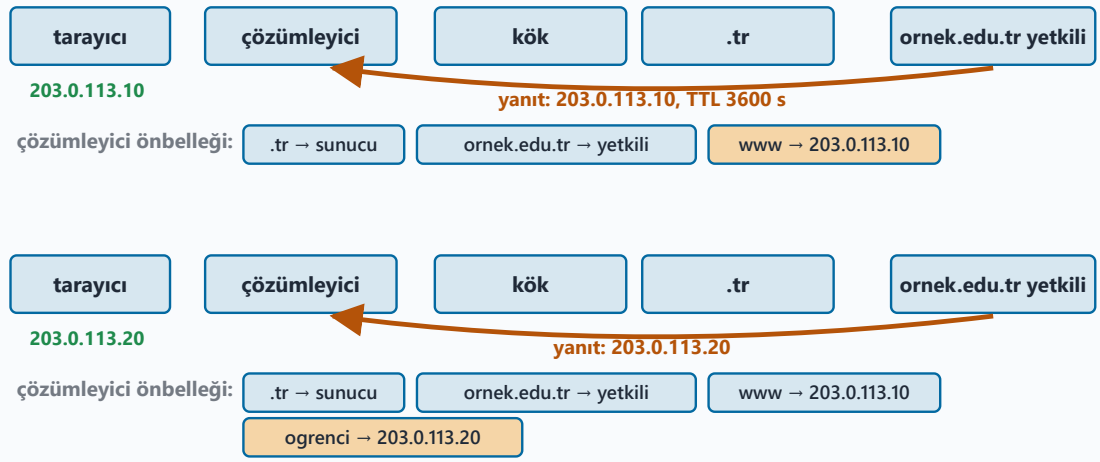
Çözümleyicinin bir soru geldiğinde izlediği algoritma şöyledir:

```
önbellekte ad var mı?  Evet → yanıtla, bitir
sunucu = en yakın bilinen (yoksa kök)
sunucuya adı sor
yönlendirme: önbelleğe yaz, ona sor
IP gelirse önbelleğe yaz (TTL), yanıtla
```

İkinci satır önemlidir: çözümleyici her zaman kökten başlamaz, önbelleğinde bildiği en yakın sunucudan başlar. Tablo 1.4.12 yeni açılmış, önbelleği boş bir çözümleyiciye arka arkaya gelen üç soruyu izler.

Soru	Önbellekte ne bulundu?	Sorulan sunucular	Önbelleğe eklenenler	Yanıt	Sorgu
`www.ornek.edu.tr`	hiçbir şey	kök → .tr → yetkili	.tr → sunucu; ornek.edu.tr → yetkili; www → 203.0.113.10 (TTL 1 saat)	203.0.113.10	3
`ogrenci.ornek.edu.tr` (aynı dakika)	ad yok, ama yetkili sunucu biliniyor	yetkili	ogrenci → 203.0.113.20	203.0.113.20	1
`www.ornek.edu.tr` (başka kullanıcı)	kayıt var	—	—	203.0.113.10	0

Tablo 1.4.12: Çözümleyici önbelleğinin izlemesi. Son sütun, o soru için dış sunuculara gönderilen sorgu sayısıdır.



Şekil 1.4.19: Üstte ilk sorunun sonu: yetkili sunucu 203.0.113.10 yanıtını TTL 3600 s ile verir; önbellekte üç kayıt vardır. Altta ikinci soru: kök ve .tr atlanır, doğrudan yetkili sunucuya sorulur ve ogrenci kaydı önbelleğe eklenir.

İlk soru üç sorgu gerektirir. İkinci soru tek sorguyla biter, çünkü ornek.edu.tr'nin yetkili sunucusu artık bilinir; kök ve .tr adımları atlanır. Üçüncü soru hiç dış sorgu gerektirmez ve yanıt hemen döner. Gerçek çözümleyiciler milyonlarca kullanıcıya hizmet verdiği için popüler adların çoğu her zaman önbellekte bulunur.

Önbelleğin bir yan etkisi de vardır. TTL 3600 s, yani 1 saattir; kayıt bu süre dolmadan silinmez. Kurum bu arada sunucusunu taşıyıp IP adresini değiştirirse, eski kaydı önbelleğinde tutan çözümleyiciler bir saate kadar eski adresi vermeyi sürdürür. Adres değişikliğinin herkese hemen yansımaması bundandır. Kurumlar planlı bir taşımadan önce TTL'yi geçici olarak kısaltır.

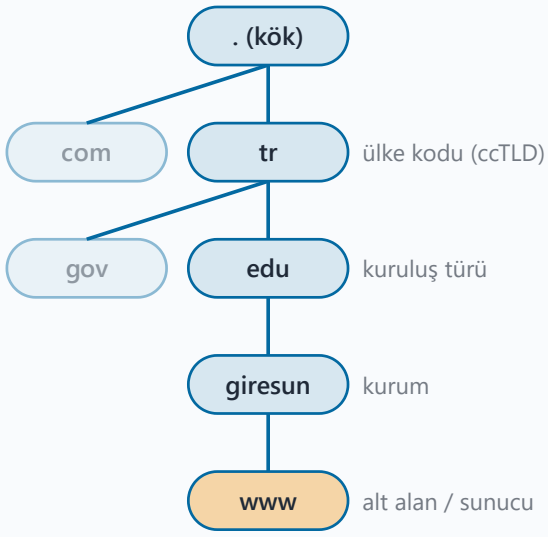
Sık yapılan hata

DNS'teki TTL ile IP başlığındaki TTL'yi aynı şey sanmak. IP paketindeki TTL bir atlama sayacıdır ve her yönlendiricide 1 azalır (1.4.2.4). DNS kaydındaki TTL ise saniye cinsinden bir süredir ve kaydın önbellekte ne kadar tutulacağını söyler. Adları aynıdır, çünkü ikisi de "yaşam süresi" anlamına gelir; ama ölçtükleri şey farklıdır.

1.4.3.6 Alan adının bölümleri

Bir alan adı noktalarla ayrılmış etiketlerden oluşur ve **sağdan sola**, genelden özele doğru okunur. `www.giresun.edu.tr` adını ele alalım:

- **tr:** ülke kodlu **üst düzey alan** (*top-level domain*, TLD). Türkiye'yi gösterir; .de Almanya'yı, .jp Japonya'yı gösterir. .com, .org gibi ülkeye bağlı olmayan genel üst düzey alanlar da vardır.
- **edu:** kuruluş türü. Türkiye'de .edu.tr eğitim kurumlarına, .gov.tr kamu kurumlarına, .com.tr şirketlere ayrılmıştır.
- **giresun:** kurumun alan adı. Kurum bu adı kaydettirmiştir ve altındaki adları kendisi yönetir.
- **www: alt alan adı** (*subdomain*) ya da sunucu adı. Kurum aynı alan altında obs, mail, kutuphane gibi başka alt alanlar da açabilir.



Şekil 1.4.20: DNS ad ağacı. Kökün (.) altında üst düzey alanlar (tr, com), onların altında kuruluş türleri (edu, gov), daha altta kurum (giresun) ve en altta sunucu adı (www) bulunur.

DNS sorgusu bu ağacı kökten aşağı izler: önce kök, sonra tr, sonra edu.tr ve giresun.edu.tr. Ad sağdan sola okunduğu için çözümleme de sağdan sola ilerler. Bu düzen sorumluluğu da dağıtır. Kök sunucular yalnız üst düzey alanları, .tr sunucuları yalnız .tr altındaki alanları bilir. giresun.edu.tr'nin içindeki adları ise yalnız üniversitenin yetkili sunucusu tutar.

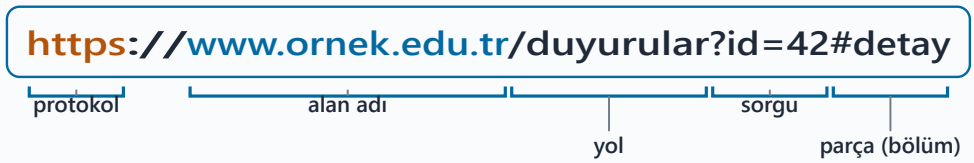
Sık yapılan hata

Alan adını soldan sağa okumak. `giresun.edu.tr.example.com` adresi Giresun Üniversitesine değil, `example.com` alanının sahibine aittir; çünkü ad sağdan sola okunur ve kurumu belirleyen kısım en sağdaki kayıtlı alandır. Oltalama saldırıları bu yanılgıdan yararlanır (1.4.7.4).

1.4.3.7 URL'nin yapısı

Tanım 1.4.13

URL (*Uniform Resource Locator*, tek biçimli kaynak konumlayıcı), internetteki bir kaynağın tam adresidir. Hangi protokolle, hangi sunucuya bağlanılacağını ve o sunucuda hangi kaynağın istendiğini tek satırda söyler.



Şekil 1.4.21: `https://www.ornek.edu.tr/duyurular?id=42#detay` adresinin bölümleri: protokol, alan adı, yol, sorgu ve parça.

Bölüm	Örnekte	Anlamı
Protokol	`https`	Hangi kurallarla iletişim kurulacağı (şifreli HTTP).
Alan adı	`www.ornek.edu.tr`	Sunucunun adı; DNS ile IP adresine çevrilir.
Yol	`/duyurular`	Sunucudaki kaynağın yolu.
Sorgu	`?id=42`	Sunucuya giden ek bilgi, `ad=değer` biçiminde; birden çok çift `&` ile ayrılır.
Parça	`#detay`	Sayfa içindeki bölüm; sunucuya gönderilmez, tarayıcı o bölüme kayar.

Tablo 1.4.13: URL'nin bölümleri.

URL'de bunlardan başka bir bölüm daha olabilir: **port**. Alan adından sonra iki nokta üst üsteyle yazılır, örneğin `http://203.0.113.10:8080`. Port yazılmazsa protokolün **varsayılan portu** kullanılır: HTTP için 80, HTTPS için 443. Bu örnekte alan adı yerine doğrudan IP adresi yazıldığına da dikkat

edelim; bu durumda DNS'e hiç başvurulmaz.

Örnek 1.4.9

`http://203.0.113.10:8080/rapor?yil=2025&ay=3#ozet` adresini bölümlerine ayıralım. Hangi parça sunucuya gitmez? Hangi porta bağlanılır?

1. Protokol: `http`. Şifresizdir.
2. Sunucu: `203.0.113.10`. Bir IP adresidir; DNS sorgusu gerekmez.
3. Port: `8080`. Açıkça yazıldığı için HTTP'nin varsayılan portu 80 kullanılmaz.
4. Yol: `/rapor`.
5. Sorgu: `?yil=2025&ay=3`. İki `ad=değer` çifti vardır: yil = 2025 ve ay = 3.
6. Parça: `#ozet`. Sunucuya gönderilmez; tarayıcı sayfa yüklendikten sonra "ozet" kimlikli bölüme kayar.

Sunucuya giden istek yalnız `/rapor?yil=2025&ay=3` kısmını içerir ve 203.0.113.10 adresinin 8080 numaralı portuna gönderilir.

Sık yapılan hata

Parçanın (`#…`) sunucuya gittiğini düşünmek ya da sorguyu (`?…`) parçayla karıştırmak. Sorgu sunucunun yanıtını değiştirebilir; parça ise yalnız tarayıcıda işlenir. Kişisel ya da gizli bilgi URL'nin hiçbir bölümüne yazılmamalıdır, çünkü URL'ler tarayıcı geçmişinde, sunucu kayıtlarında ve paylaşılan bağlantılarda açıkça görünür.

1.4.3.8 Port

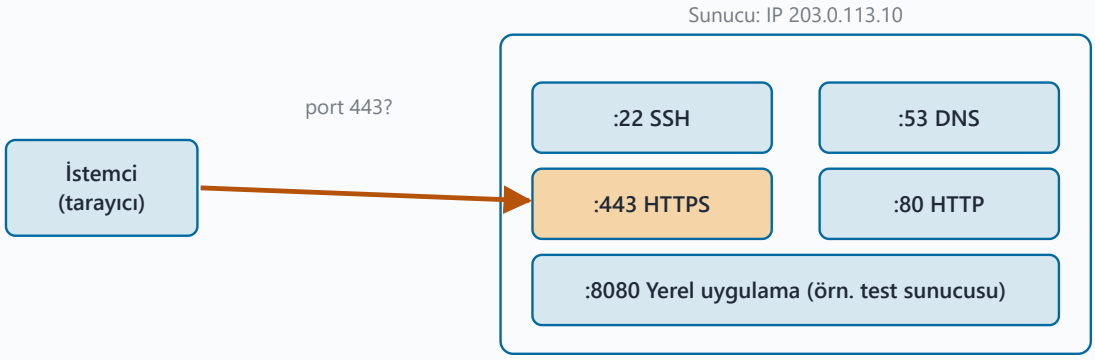
Bir sunucuda aynı anda birçok hizmet çalışabilir: web sunucusu, e-posta sunucusu, uzaktan bağlantı hizmeti, DNS hizmeti. IP adresi paketin hangi bilgisayara gideceğini söyler, ama o bilgisayardaki hangi hizmete gideceğini söylemez. Bu işi port numarası yapar.

Tanım 1.4.14

Port, bir cihazdaki belirli bir hizmeti ya da bağlantı ucunu tanımlayan 16 bitlik sayıdır (0–65.535). IP adresi bir binaya, port numarası o binadaki bir daireye benzer. Sık kullanılan portlar: **80** HTTP, **443** HTTPS, **22** SSH (şifreli uzaktan bağlantı), **53** DNS.

1. Tarayıcı `https://ornek.edu.tr` adresini açar. DNS IP adresini verir; HTTPS için varsayılan port 443'tür. Pakete hedef olarak IP:port, yani 203.0.113.10:443 yazılır.

2. Paket yönlendiriciler aracılığıyla önce IP adresine, yani binaya ulaşır. IP yalnız hangi bilgisayara gideceğini belirler.
3. Sunucunun işletim sistemi port numarasına, yani daireye bakar. Paketi o portu dinleyen hizmete, burada HTTPS web sunucusuna teslim eder. 80 numaralı porta gelen paket başka, 22 numaralı porta gelen paket yine başka bir hizmete gider.
4. URL'ye port yazılırsa varsayılan port yerine o kullanılır: `http://203.0.113.10:8080` adresine giden paket, 8080'i dinleyen başka bir uygulamaya (örneğin bir test sunucusuna) ulaşır.



Şekil 1.4.22: 203.0.113.10 adresli sunucuda beş hizmet farklı portlarda çalışır. 443 numaralı porta gelen paket HTTPS web sunucusuna teslim edilir.

Bağlantının istemci tarafında da bir port vardır. Tarayıcı bir bağlantı açtığında işletim sistemi ona geçici bir port verir (örneğin 50000). Sunucunun yanıtı bu porta gelir. Bu sayede aynı bilgisayarda açık olan on sekme, aynı sunucuya ayrı ayrı bağlanabilir: her sekmenin bağlantısı farklı bir istemci portu kullanır. NAT'ın port değiştirmesi de (1.4.3.2) bu istemci portları üzerinde yapılır.

Sık yapılan hata

Ağ portunu bilgisayarın arkasındaki fiziksel girişlerle (USB, HDMI ya da switch'in kablo girişi) karıştırmak. Ağ portu fiziksel bir delik değil, işletim sisteminin bağlantıları ayırt etmek için kullandığı bir sayıdır.

1.4.4 TCP bağlantısı ve şifreli iletişim

IP paketleri yalnızca “elden geldiğince” taşır: paket kaybolabilir, sırası değişebilir, kopyası gelebilir. Bir web sayfasının ya da bir banka işleminin ise eksiksiz ve doğru sırayla ulaşması gerekir. Ayrıca yolda veriyi okuyabilecek kişilere karşı içeriğin gizli kalması gerekir. Bu kesim bu iki ihtiyacın nasıl karşılandığını anlatır: önce TCP'nin güvenilir bağlantısı, sonra HTTPS'in şifrelemesi.

1.4.4.1 TCP: üçlü el sıkışma, veri ve kapanış

TCP, iki uç arasında veri göndermeden önce bir **bağlantı** kurar. Bağlantı kurulurken iki taraf birbirine başlangıç sıra numaralarını bildirir. Veri aktarılırken her bayt bir numara alır ve alıcı, hangi bayta kadar aldığını **onay** (*acknowledgment*, ACK) ile bildirir. İş bitince bağlantı iki yönde de düzenli biçimde kapatılır.

Tanım 1.4.15

Üçlü el sıkışma (*three-way handshake*), TCP bağlantısının üç bölütle kurulmasıdır: istemci **SYN** (eş zamanla) gönderir, sunucu **SYN + ACK** ile yanıt verir, istemci **ACK** ile onaylar. **Sıra numarası** (*sequence number*, seq) bir bölütteki ilk baytın numarasıdır. **Onay numarası** (ack) ise "bu numaraya kadar her şeyi aldım, sıradaki beklediğim bayt budur" anlamına gelir.

Örnekte istemci, 443 numaralı portta dinleyen bir sunucuya bağlanıyor. Bütün süreç kısaca şöyledir:

```
istemci → SYN
sunucu → SYN + ACK
istemci → ACK        // bağlantı kuruldu
istemci → veri (HTTP isteği)
sunucu → ACK
istemci → FIN;  sunucu → ACK
sunucu → FIN;  istemci → ACK  // kapandı
```

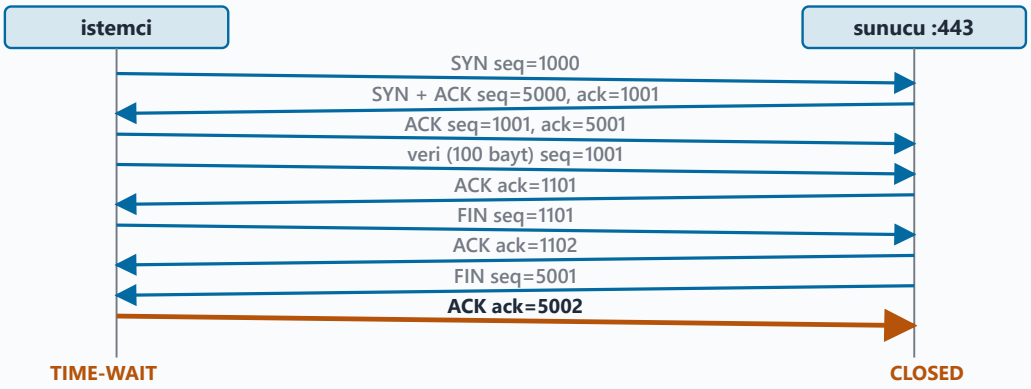
Sayılarla izleyelim. İstemcinin başlangıç sıra numarası 1000, sunucununki 5000 olsun. Gerçekte bu sayılar her bağlantıda rastgele seçilir; böylece eski bir bağlantıdan kalan bölütler yenisiyle karışmaz ve dışarıdan sahte bölüt sokmak zorlaşır.

#	Yön	Bölüt	seq	ack	İstemci durumu	Sunucu durumu
1	istemci → sunucu	SYN	1000	—	SYN-SENT	SYN-RECEIVED
2	sunucu → istemci	SYN + ACK	5000	1001	SYN-SENT	SYN-RECEIVED
3	istemci → sunucu	ACK	1001	5001	ESTABLISHED	ESTABLISHED
4	istemci → sunucu	veri (100 bayt)	1001	—	ESTABLISHED	ESTABLISHED
5	sunucu → istemci	ACK	—	1101	ESTABLISHED	ESTABLISHED
6	istemci → sunucu	FIN	1101	—	FIN-WAIT	CLOSE-WAIT
7	sunucu → istemci	ACK	—	1102	FIN-WAIT	CLOSE-WAIT
8	sunucu → istemci	FIN	5001	—	TIME-WAIT	LAST-ACK
9	istemci → sunucu	ACK	—	5002	TIME-WAIT	CLOSED

Tablo 1.4.14: TCP bağlantısının izlemesi (istemcinin başlangıç sıra numarası 1000, sunucununki 5000).

Satırları sırayla açıklayalım:

1. **SYN.** İstemci bağlantı ister ve başlangıç sıra numarasını (1000) bildirir.
2. **SYN + ACK.** Sunucu kabul eder: kendi sıra numarası 5000'dir ve ack = 1001'dir, yani "1000'i aldım, 1001'i bekliyorum". SYN bölütü veri taşımasa da bir sıra numarası harcar; ack'in 1001 olması bundandır.
3. **ACK.** İstemci, sunucunun SYN'ini ack = 5001 ile onaylar. Üç bölütle bağlantı kurulmuştur, ama henüz hiç veri gönderilmemiştir.
4. **Veri.** İstemci 100 baytlık HTTP isteğini yollar. İlk baytın numarası 1001'dir; baytlar 1001'den 1100'e kadar numaralanır. HTTPS'te bu kanal üzerinden önce TLS anlaşması da yapılır (1.4.4.4).
5. **ACK.** Sunucu ack = 1101 ile 100 baytın hepsini aldığını bildirir: 1001 + 100 = 1101. Onay gelmeseydi istemci bir süre bekleyip aynı baytları yeniden gönderirdi.
6. **FIN.** İş bitmiştir; istemci FIN ile kendi yönünü kapatır.
7. **ACK.** Sunucu FIN'i ack = 1102 ile onaylar. FIN de SYN gibi bir sıra numarası harcar. Sunucunun göndereceği veri kaldıysa hâlâ yollayabilir; bağlantı şimdilik yalnız bir yönde kapanmıştır.
8. **FIN.** Sunucu da kendi yönünü kapatır (seq = 5001).
9. **ACK.** İstemci son onayı (ack = 5002) yollar. Bağlantı iki yönde de kapanmıştır.



Şekil 1.4.23: TCP bağlantısının bütün bölütleri zaman çizgisinde. Yukarıdan aşağı: üçlü el sıkışma, 100 baytlık veri ve onayı, dört bölütlük kapanış. Alttaki yazılar iki tarafın son durumunu gösterir.

Kurmak 3, kapatmak 4 bölüt sürer. Kapanışın dört bölüt sürmesinin nedeni, iki yönün birbirinden bağımsız kapatılmasıdır: bir taraf "benim söyleyeceklerim bitti" dediğinde öteki taraf hâlâ veri göndermeyi sürdürebilir. UDP'de bu adımların hiçbiri yoktur; UDP bağlantı kurmadan ilk pakette doğrudan veriyi gönderir.

Örnek 1.4.10

Başka bir bağlantıda istemcinin başlangıç sıra numarası 300, sunucununki 9000'dir. İstemci 250 baytlık bir istek, sunucu da 1200 baytlık bir yanıt gönderiyor. Bölütlerdeki seq ve ack değerleri nedir?

1. SYN: seq = 300.

2. SYN + ACK: seq = 9000, ack = 300 + 1 = 301.
3. ACK: seq = 301, ack = 9000 + 1 = 9001.
4. İstek: seq = 301; baytlar 301'den 301 + 250 − 1 = 550'ye kadar. Sunucunun onayı ack = 551'dir.
5. Yanıt: seq = 9001; baytlar 9001'den 9001 + 1200 − 1 = 10.200'e kadar. İstemcinin onayı ack = 10.201'dir.

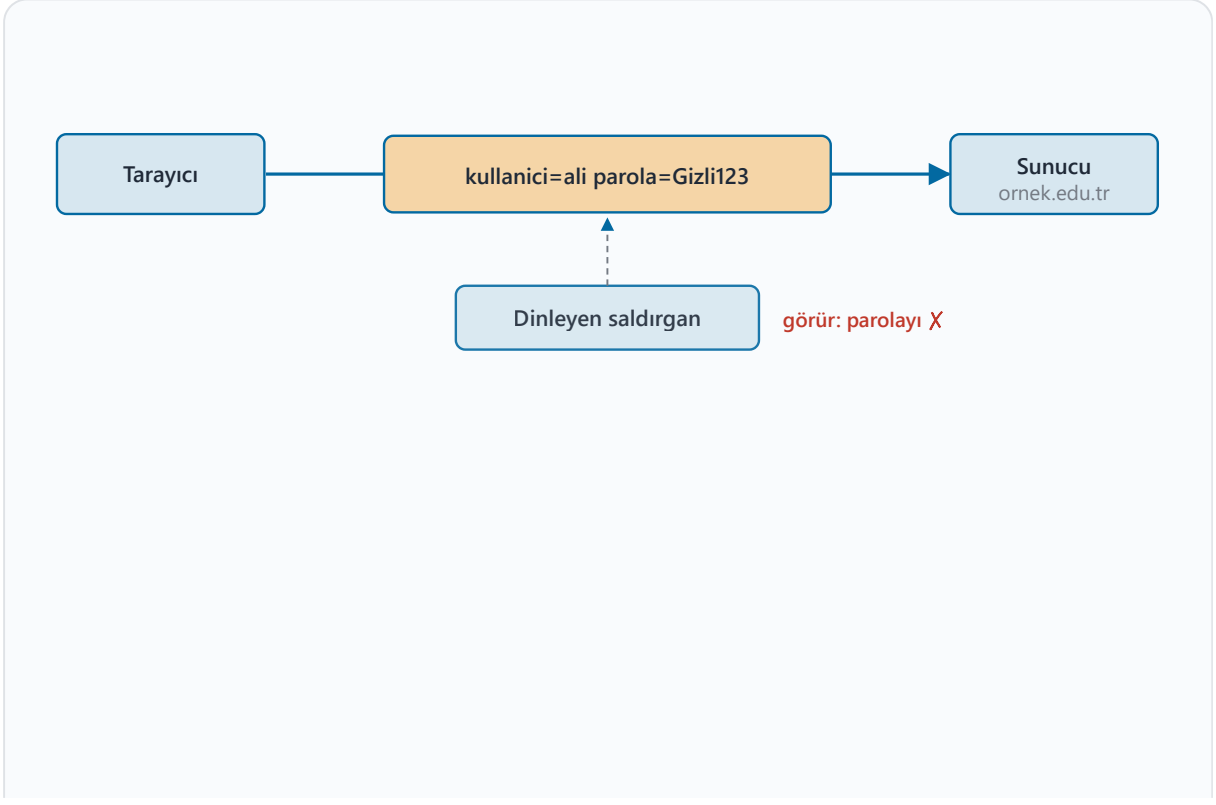
Kural her seferinde aynıdır: onay numarası = alınan son baytın numarası + 1. SYN ve FIN birer numara harcar.

Sık yapılan hata

Onay numarasını alınan son baytın numarası sanmak. ack = 1101, "1101'e kadar aldım" değil, "1100'e kadar aldım, sıradaki beklediğim bayt 1101" demektir. Bir başka yaygın hata da SYN'in sıra numarası harcadığını unutup ilk veri baytını 1000 olarak yazmaktır.

1.4.4.2 HTTP ve HTTPS

HTTP, tarayıcı ile web sunucusu arasındaki konuşmanın protokolüdür. Tarayıcı "şu kaynağı ver" der, sunucu kaynağı gönderir. Bu konuşma düz metin hâlindedir: yoldaki herhangi bir cihaz paketleri okuyabilir. HTTPS ise aynı konuşmayı **TLS** (*Transport Layer Security*, taşıma katmanı güvenliği) ile şifreler.





Şekil 1.4.24: Üstte HTTP: aynı ağdaki dinleyen saldırgan kullanıcı adını ve parolayı düz metin olarak görür. Altta HTTPS: aynı saldırgan yalnız anlamsız baytlar görür; tarayıcı ayrıca sunucunun ornek.edu.tr adına verilmiş sertifikasını denetler.

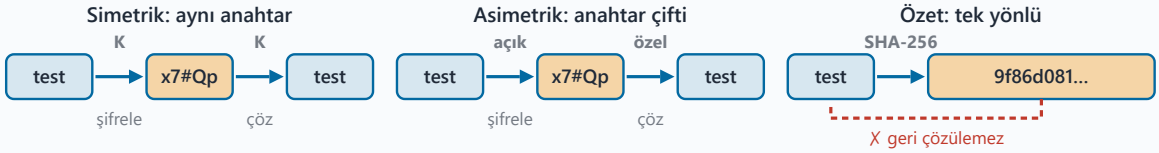
Farkı somut bir durumla görelim. Bir öğrenci kafedeki açık Wi-Fi ağından bir siteye giriş yapıyor. Site HTTP kullanıyorsa, aynı ağdaki biri paketleri yakalayıp `kullanici=ali parola=Gizli123` gibi bir satırı açıkça okuyabilir. Site HTTPS kullanıyorsa aynı paketlerde `7f3a 9c1e b24d 0e88 …` gibi anlamsız baytlar görünür. Tarayıcının adres çubuğundaki "bağlantı güvenli" göstergesi (eski sürümlerde kilit simgesi) bağlantının şifreli olduğunu belirtir.

HTTPS'in ikinci işi kimlik denetimidir. Şifreleme tek başına yetmez: kullanıcı farkında olmadan saldırganın sunucusuna bağlanmışsa, veri şifreli bile olsa saldırgana gider. Bu yüzden sunucu, güvenilir bir kuruluşun imzaladığı bir **sertifika** (*certificate*) gösterir. Sertifika, "bu açık anahtar gerçekten ornek.edu.tr alan adına aittir" der. Tarayıcı imzayı denetler; sertifika başka bir ada verilmişse ya da süresi dolmuşsa uyarı gösterir.

> **Sık yapılan hata**
>
> Kilit simgesini "bu site güvenilir" diye okumak. Kilit iki şeyi garanti eder: trafik şifrelidir ve sertifika adres çubuğundaki alan adına verilmiştir. Sitenin dürüst olduğunu garanti etmez. Oltalama siteleri de kendi alan adları için kolayca sertifika alıp HTTPS kullanır. Asıl bakılması gereken şey alan adının kendisidir.

1.4.4.3 Şifreleme temelleri

Şifreleme (*encryption*), okunabilir bir metni (**düz metin**) bir anahtar yardımıyla anlamsız görünen bir metne dönüştürmektir. Ters işleme **çözme** (*decryption*) denir. Anahtarı bilmeyen biri şifreli metni görse bile içeriği okuyamaz. Kullanılan anahtarın türüne göre iki yaklaşım vardır; bunlara bir de şifrelemeyle sık karıştırılan özet işlemi eklenir.



Şekil 1.4.25: Soldan sağa: simetrik şifrelemede aynı anahtar (K) şifreler ve çözer; asimetrik şifrelemede açık anahtar şifreler, özel anahtar çözer; özet (SHA-256) tek yönlüdür ve geri çözülemez.

Tanım 1.4.16

Simetrik şifreleme (*symmetric encryption*): şifreleme ve çözme için aynı anahtar kullanılır. Hızlıdır; sorun, anahtarın karşı tarafa güvenle ulaştırılmasıdır. Örnek: AES.

Asimetrik şifreleme (*asymmetric encryption*): her tarafın bir anahtar çifti vardır. **Açık anahtar** (*public key*) herkese verilir ve şifrelemede ya da imza doğrulamada kullanılır. **Özel anahtar** (*private key*) gizli kalır ve çözmede ya da imzalamada kullanılır. Daha yavaştır, ama anahtar dağıtım sorununu çözer. Örnek: RSA, eliptik eğri.

Özet (*hash*): veriden sabit uzunlukta bir "parmak izi" üretir. Şifreleme değildir; geri çözülemez. Parola saklama ve bütünlük denetiminde kullanılır. Örnek: SHA-256.

Simetrik şifrelemenin mantığı. Bilinen en eski örnek Sezar şifresidir: her harf alfabede sabit sayıda öteye kaydırılır. 29 harfli Türk alfabesinde kaydırma 3 ise A → Ç, Ğ → İ olur ve "AĞ" sözcüğü "Çİ" biçiminde yazılır. Alıcı aynı anahtarı (3) bilir ve harfleri 3 geri kaydırarak çözer. Bu şifre bugün birkaç saniyede kırılır, çünkü yalnız 28 farklı anahtar vardır ve hepsi denenebilir. AES gibi gerçek algoritmalar 128 ya da 256 bitlik anahtar kullanır; 2^{128} anahtarı denemek akla gelebilecek her bilgisayar gücünün ötesindedir. Ama ilke aynıdır: iki tarafta da aynı gizli anahtar bulunmalıdır.

Sorun da buradadır. Tarayıcı ile hiç tanımadığı bir sunucu, internet üzerinden ortak bir gizli anahtarı nasıl belirleyecek? Anahtarı açıkça gönderirlerse yoldaki dinleyici de öğrenir. Asimetrik şifreleme bu sorunu çözer.

Örnek 1.4.11

Asimetrik şifrelemenin nasıl çalıştığını küçük sayılarla, oyuncak bir RSA örneğiyle görelim. Burada *a mod n*, a'nın n'ye bölümünden kalandır.

1. Sunucu iki asal sayı seçer: p = 5, q = 11. Çarpımları n = 55'tir.
2. (p − 1)(q − 1) = 4 · 10 = 40 hesaplanır. Açık üs e = 3 seçilir; özel üs d, e · d'nin 40'a bölümünden kalan 1 olacak biçimde bulunur: d = 27, çünkü 3 · 27 = 81 = 2 · 40 + 1.
3. Açık anahtar (n = 55, e = 3) herkese duyurulur. Özel anahtar d = 27 yalnız sunucuda kalır.
4. Tarayıcı 7 sayısını göndermek istiyor. Açık anahtarla şifreler: $7^3 = 343$ ve 343 mod 55 = 13 (çünkü 6 · 55 = 330, 343 − 330 = 13). Yola 13 çıkar.

5. Sunucu özel anahtarla çözer: $13^{27} \bmod 55 = 7$. Asıl sayı geri gelmiştir.

Dinleyici 13'ü ve açık anahtarı (55, 3) görür; d'yi bulmak için 55'i asal çarpanlarına ayırması gerekir. $55 = 5 \cdot 11$ hemen bulunur. Gerçek RSA anahtarlarında ise n yüzlerce basamaklıdır (2048 bit). Böyle bir sayıyı çarpanlarına ayırmanın bilinen hızlı bir yolu yoktur. Güvenlik bu zorluğa dayanır.

Özet işlevleri. SHA-256, girdisi ne kadar uzun olursa olsun 256 bitlik (64 onaltılık rakamlık) bir özet üretir. `test` sözcüğünün SHA-256 özeti `9f86d081884c7d65…` ile başlar. Girdideki en küçük değişiklik özeti tamamen değiştirir: `Test` (büyük T ile) `532eaabd9574880d…`, sonuna bir boşluk eklenmiş `test␣` ise `231aa78c52a74a57…` özetini verir. Özetten girdiyi geri hesaplamanın bir yolu yoktur.

Bu özellikler iki işe yarar. **Bütünlük denetimi:** bir dosya indirilirken sitenin yayımladığı özet ile indirilen dosyanın özeti karşılaştırılır; tek bir bit bile değiştiyse özetler tutmaz. **Parola saklama:** iyi tasarlanmış sistemler kullanıcının parolasını değil, parolanın özetini saklar. Giriş sırasında yazılan parolanın özeti hesaplanıp saklananla karşılaştırılır. Veri tabanı çalınsa bile parolalar doğrudan okunamaz.

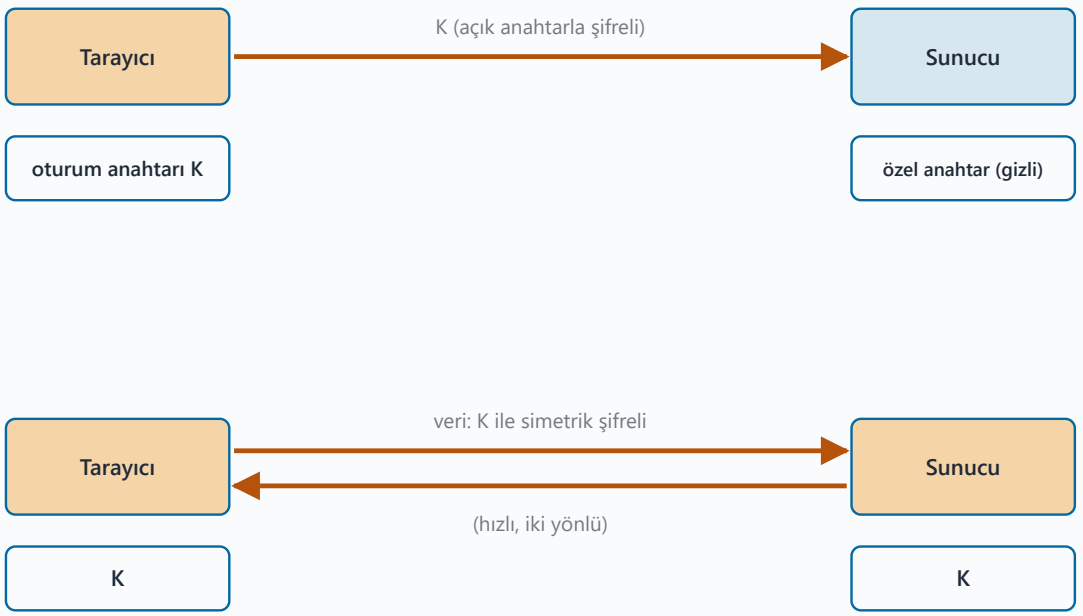
Sık yapılan hata

"Parolalar SHA-256 ile şifrelenmiş" demek. Özet bir şifreleme değildir: anahtar kullanmaz ve geri çözülemez. "Şifrelenmiş" veri doğru anahtarla geri açılır; "özetlenmiş" veri hiçbir anahtarla geri açılmaz. İki kavram ayrı amaçlara hizmet eder.

1.4.4.4 HTTPS'te yöntemler birlikte çalışır

Simetrik şifreleme hızlıdır ama anahtar dağıtımı sorunludur. Asimetrik şifreleme anahtar dağıtımını çözer ama yavaştır. HTTPS ikisinin güçlü yanlarını birleştirir: asimetrik yöntemi yalnız bir kez, ortak bir oturum anahtarı üzerinde anlaşmak için kullanır; asıl veriyi ise bu anahtarla simetrik olarak şifreler.

1. Sunucu, tarayıcıya sertifikasını ve açık anahtarını gönderir. Özel anahtar sunucuda gizli kalır. Tarayıcı sertifikayı denetler.
2. Tarayıcı rastgele bir **oturum anahtarı** (K) üretir ve sunucunun açık anahtarıyla şifreleyerek yollar. Bu asimetrik adımdır; yolda dinleyen biri K'yi çözemez.
3. Sunucu K'yi kendi özel anahtarıyla çözer. Artık iki tarafta da aynı gizli anahtar vardır ve asimetrik şifreleme işini bitirmiştir.
4. Asıl veri (sayfa, form, parola) K ile simetrik olarak şifrelenir ve iki yönde akar.



Şekil 1.4.26: Üstte asimetrik adım: tarayıcı oturum anahtarı K'yi sunucunun açık anahtarıyla şifreleyip gönderir. Altta simetrik adım: iki tarafta da K vardır ve veri K ile iki yönde şifreli akar.

Bu anlatım ilkeyi gösteren sade bir modeldir. Güncel TLS 1.3 sürümü K'yi doğrudan şifreleyip göndermez; iki taraf Diffie–Hellman türü bir **anahtar anlaşması** (*key agreement*) ile, anahtarı hiç yola çıkarmadan ortak bir sır hesaplar. Sonuç aynıdır: iki tarafta aynı gizli anahtar oluşur, dinleyici ise onu öğrenemez. Sertifika ve özel anahtar bu süreçte sunucunun kimliğini kanıtlamak için kullanılır. Asimetrik işlemler simetrik işlemlerden yüzlerce kat yavaş olduğu için bu bölüşüm, şifrelemenin maliyetini neredeyse hissedilmez kılar.

1.4.5 Web

Gündelik dilde “internete girmek” çoğu zaman bir web tarayıcısı açmak anlamına gelir. Oysa Web, internetin üzerinde çalışan hizmetlerden yalnız biridir. Bu kesim önce iki kavramı birbirinden ayırır, sonra bir web sayfasının hangi parçalardan oluştuğunu ve tarayıcıya nasıl ulaştığını adım adım izler. Kesimin ikinci yarısı arama motorlarına ve bulunan bilginin değerlendirilmesine ayrılmıştır.

1.4.5.1 İnternet ve Web

Tanım 1.4.17

Web (*World Wide Web*), birbirine bağlantılarla bağlanmış sayfalardan ve web uygulamalarından oluşan, HTTP/HTTPS ile erişilen bir hizmettir. İnternet altyapıdır; Web bu altyapının üzerindeki hizmetlerden biridir.



Şekil 1.4.27: İnternet altyapısı (kablolar, fiber, radyo bağlantıları, yönlendiriciler ve TCP/IP protokolleri) üzerinde çalışan hizmetler: Web (HTTP/HTTPS), e-posta (SMTP, IMAP), dosya aktarımı (FTP, SFTP), görüntülü arama (RTP) ve çevrim içi oyun (UDP).

Bir benzetmeyle: internet bir ülkenin karayolu ağıdır; Web ise bu yollarda çalışan otobüs firmalarından biridir. Aynı yolları kargo şirketleri, ambulanslar ve özel araçlar da kullanır. E-posta, dosya aktarımı, görüntülü arama ve çevrim içi oyunlar Web'i değil, doğrudan internet altyapısını kullanır. Telefondaki bir mesajlaşma uygulaması da hiç tarayıcı açmadan internet üzerinden çalışır.

Sık yapılan hata

"İnternet" ile "Web"i eş anlamlı kullanmak. İnternet 1969'dan beri gelişen ağ altyapısıdır; Web 1991'de bu altyapının üzerinde başlamıştır. Web olmadan da e-posta gönderilebilir; internet olmadan ise Web çalışamaz.

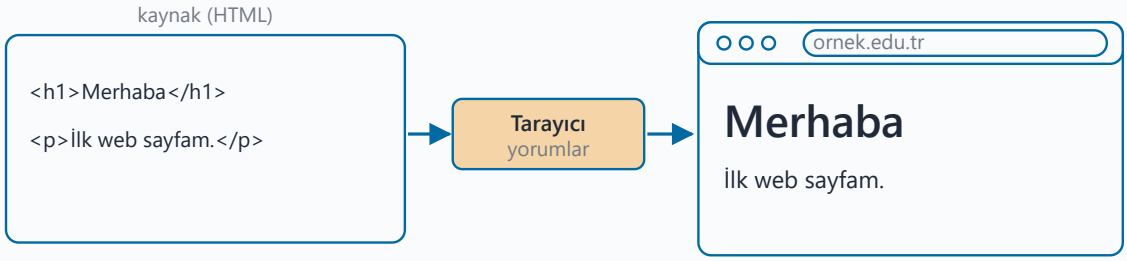
1.4.5.2 HTML, CSS, JavaScript ve tarayıcı

Bir web sayfası tek bir dosya değil, birlikte çalışan üç dilde yazılmış kaynakların toplamıdır. Her dilin ayrı bir görevi vardır:

- **HTML** (köprü metni işaretleme dili) sayfanın **yapısını** tanımlar: başlık nerede, paragraf nerede, hangi metin bir bağlantıdır.
- **CSS** (basamaklı stil sayfaları) **görünümü** düzenler: renkler, yazı tipleri, boşluklar, sayfa düzeni.
- **JavaScript etkileşim ve davranış** ekler: bir düğmeye tıklanınca ne olacağı, bir formun gönderilmeden önce denetlenmesi, sayfanın yenilenmeden güncellenmesi.

Web tarayıcısı (*web browser*) bu kaynakları sunucudan alır, yorumlar ve ekranda gösterir. Aşağıdaki iki satırlık HTML kaynağı, tarayıcıda büyük bir "Merhaba" başlığı ve altında bir paragraf olarak görünür (Şekil 1.4.28).

```
<h1>Merhaba</h1>
<p>İlk web sayfam.</p>
```



Şekil 1.4.28: Soldaki HTML kaynağı tarayıcıdan geçer ve sağdaki pencerede biçimlendirilmiş sayfa olarak görünür.

`<h1>` ve `</h1>` birer **etikettir**: aradaki metnin birinci düzey başlık olduğunu söyler. `<p>` ise bir paragrafı işaretler. HTML bir programlama dili değil, bir **işaretleme dilidir**: hesap yapmaz, döngü kurmaz; yalnız içeriğin hangi parçasının ne olduğunu belirtir. Başlığın hangi büyüklükte ve hangi renkte görüneceğine CSS, düğmeye basılınca ne olacağına JavaScript karar verir.

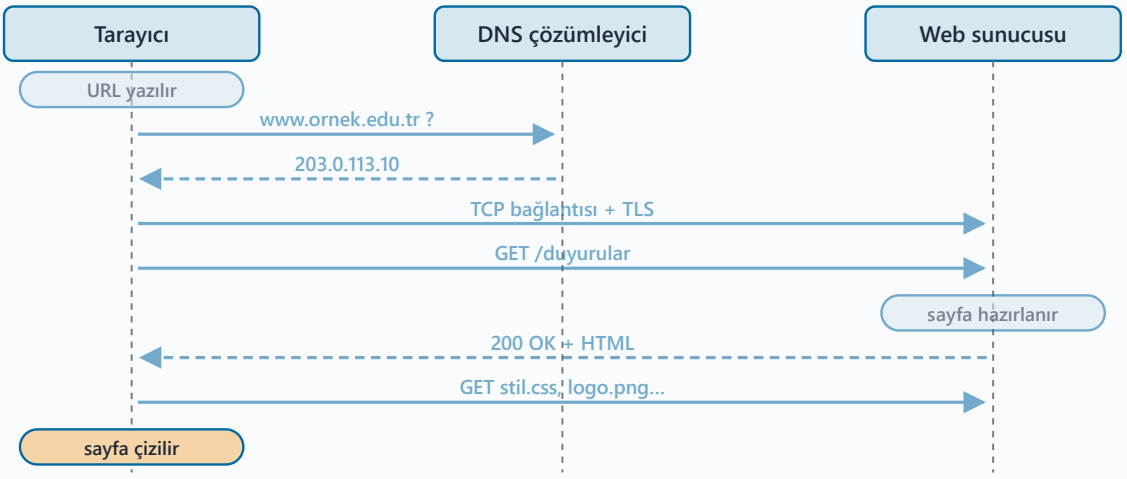
1.4.5.3 Bir web isteğinin yolculuğu

Bu noktaya kadar anlatılan kavramların hepsi, tek bir web sayfasının açılmasında birlikte rol alır. Kullanıcının adres çubuğuna `https://www.ornek.edu.tr/duyurular` yazıp Enter'a bastığını düşünelim.

Adım	Ne olur?	İlgili kavram
1	Tarayıcı URL'yi parçalarına ayırır: protokol https, port 443, alan adı www.ornek.edu.tr, yol /duyurular.	URL, varsayılan port
2	Tarayıcı alan adının IP adresini sorar.	DNS çözümlemesi
3	Yanıt gelir: sunucu 203.0.113.10 adresindedir.	DNS, önbellek
4	Tarayıcı sunucuyla TCP bağlantısı kurar; HTTPS olduğu için TLS ile şifreleme anahtarları üzerinde anlaşılır.	Üçlü el sıkışma, TLS
5	Şifreli kanal üzerinden HTTP isteği gönderilir: "/duyurular kaynağını ver" (`GET /duyurular`).	HTTP isteği
6	Sunucu isteği işler: bir dosyayı okur ya da veri tabanından duyuruları çekip HTML üretir.	Sunucu
7	Yanıt döner: durum kodu 200 OK (başarılı) ve sayfanın HTML içeriği. Kaynak bulunamasaydı 404 dönerdi.	HTTP yanıtı, durum kodu

Adım	Ne olur?	İlgili kavram
8	Tarayıcı HTML'i okurken gereken CSS, JavaScript ve resim dosyalarını ayrı isteklerle alır.	Ek istekler
9	Tarayıcı HTML yapısını, CSS görünümünü ve JavaScript davranışını birleştirip sayfayı ekrana çizer.	Tarayıcı

Tablo 1.4.15: Bir web isteğinin yolculuğu ve her adımda kullanılan kavram.



Şekil 1.4.29: Web isteğinin zaman çizgisi: tarayıcı, DNS çözümleyicisi ve web sunucusu arasındaki iletiler yukarıdan aşağı sırayla; en altta sayfa çizilir.

Görünürde tek bir tıklama, arka planda en az dört ayrı konuşma başlatır: DNS sorgusu, TCP el sıkışması, TLS anlaşması ve HTTP isteği. Her biri en az bir gidiş-dönüş sürer. Sunucu 25 ms uzaktaysa (gidiş-dönüş), ilk bayt gelmeden önce yalnız bu konuşmalar için 100 ms kadar beklenir. Gecikmenin, bant genişliğinden bağımsız olarak sayfa açılış süresini neden bu kadar etkilediği buradan anlaşılır.

Durum kodları. Sunucunun her yanıtı üç basamaklı bir **durum kodu** (*status code*) taşır. İlk basamak sonucun türünü söyler: 2 ile başlayanlar başarı, 3 ile başlayanlar yönlendirme, 4 ile başlayanlar istemci kaynaklı hata, 5 ile başlayanlar sunucu kaynaklı hatadır.

Kod	Anlamı	Ne zaman görülür?
200 OK	Başarılı	İstenen kaynak gönderildi.
301	Kalıcı olarak taşındı	Sayfanın adresi değişti; tarayıcı yeni adrese gider.
403	Yasak	Kaynak var, ama bu kullanıcının erişim izni yok.
404	Bulunamadı	Yol yanlış yazıldı ya da sayfa kaldırıldı.
500	Sunucu iç hatası	Sunucudaki program çalışırken bir hata oluştu.

Tablo 1.4.16: Sık görülen HTTP durum kodları.

Bu yolculuğu adım adım izlemek için Bir Web İsteğinin Yolculuğu simülatörü kullanılabilir.

1.4.5.4 Arama motoru: tara, dizinle, sırala

Bir arama motoruna sorgu yazıldığında Web o anda gezilmez. Milyarlarca sayfayı her sorguda tek tek okumak saatler sürerdi. Yanıt, önceden hazırlanmış bir **dizinden** (*index*) gelir. Arama motorunun işi dört aşamada özetlenebilir: tara, dizinle, sırala, sonuçları göster.

1. **Tara. Web robotu** (*web crawler*) bir başlangıç sayfası listesinden yola çıkar. Her sayfayı indirir, içindeki bağlantıları bulur ve bu bağlantıları ziyaret edilecekler kuyruğuna ekler. Tüm Web taranmaz; yalnız bağlantılarla ulaşılabilen ve site sahibinin taranmasına izin verdiği sayfalar taranır.
2. **Dizinle.** Taranan sayfalardaki sözcükler **ters dizine** yazılır.
3. **Sırala.** Sorgudaki sözcükleri içeren aday sayfalar bulunur ve ilgi, güncellik, gelen bağlantılar, yükleme hızı gibi ölçütlerle puanlanır.
4. **Sonuç.** Puanı en yüksek sayfalar başlık, adres ve kısa bir özetle listelenir. Bütün bu işlem genellikle saniyenin kesirlerinde tamamlanır.

Tanım 1.4.18

Ters dizin (*inverted index*), her sözcük için o sözcüğün geçtiği sayfaların listesini tutan tablodur: sözcük → sayfalar. Adının "ters" olması, sayfa → sözcükler biçimindeki doğal düzenin tersine çevrilmiş olmasındandır. Birden çok sözcüklü bir sorgunun sonucu, sözcüklerin sayfa listelerinin **kesişimidir**.

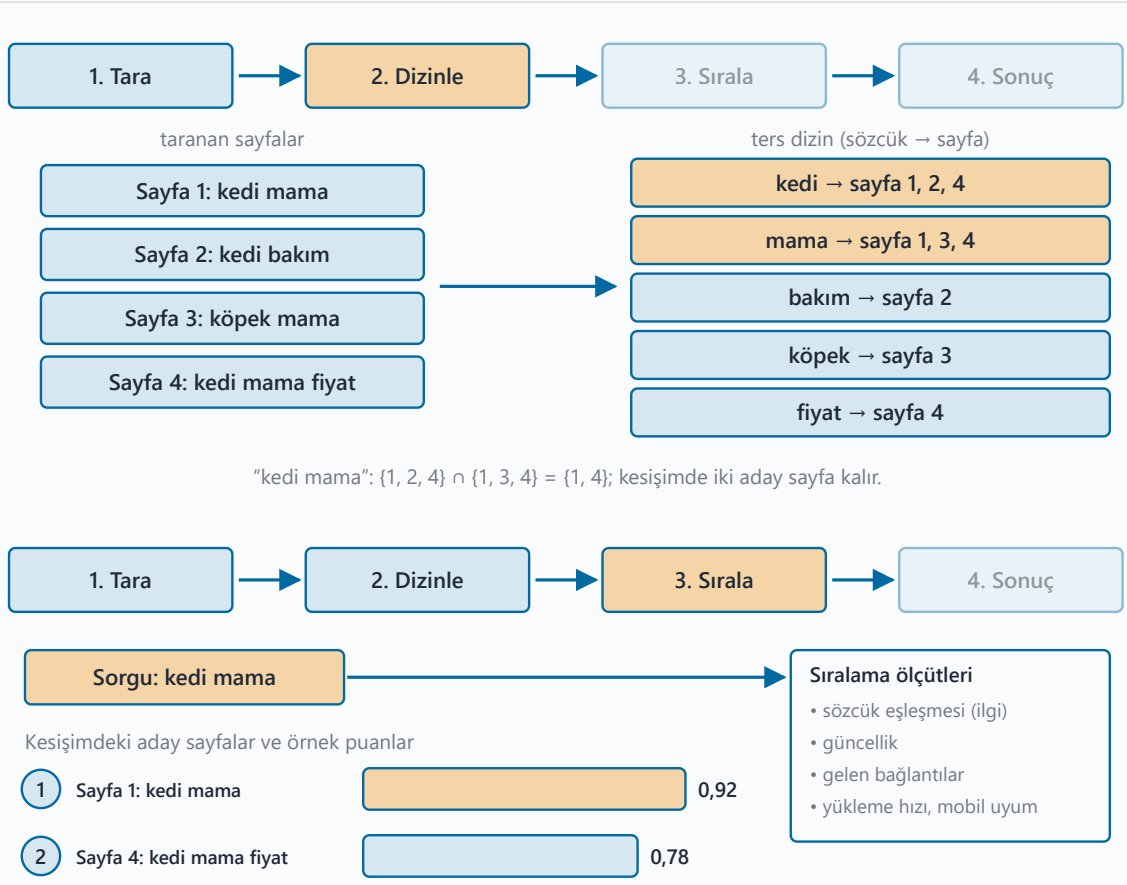
Ters dizinin nasıl işlediğini kitapların sonundaki dizin iyi anlatır. Bir ders kitabında "yönlendirici" sözcüğünün geçtiği yerleri bulmak için kitabı baştan sona okumak gerekmez; sondaki dizinde "yönlendirici: 12, 45, 78" yazar. Arama motorunun dizini de aynı işi Web ölçeğinde yapar.

Örnek 1.4.12

Dört sayfa taranmış olsun: Sayfa 1 "kedi mama", Sayfa 2 "kedi bakım", Sayfa 3 "köpek mama", Sayfa 4 "kedi mama fiyat". Ters dizini kuralım ve üç sorguyu yanıtlayalım: (a) `kedi mama`, (b) `mama -kedi`, (c) `köpek bakım`.

1. Her sayfanın sözcükleri dizine eklenir: kedi → {1, 2, 4}; mama → {1, 3, 4}; bakım → {2}; köpek → {3}; fiyat → {4}.
2. (a) İki sözcük de geçmeli: {1, 2, 4} ∩ {1, 3, 4} = {1, 4}. Sonuç: Sayfa 1 ve Sayfa 4; hangisinin önce görüneceğine sıralama karar verir.
3. (b) "mama" geçmeli, "kedi" geçmemeli: {1, 3, 4} − {1, 2, 4} = {3}. Sonuç: Sayfa 3.
4. (c) {3} ∩ {2} = ∅. Hiçbir sayfa iki sözcüğü birlikte içermez; sonuç yoktur.

Sorgu sırasında hiçbir sayfa okunmamıştır; yalnız iki kısa liste karşılaştırılmıştır. Gerçek dizinde "kedi" sözcüğünün listesi milyonlarca sayfa içerir, ama kesişim işlemi yine de çok hızlıdır.



Şekil 1.4.30: Üstte dizinleme: taranan dört sayfanın sözcükleri ters dizine yazılır; "kedi mama" için kesişim {1, 4}'tür. Altta sıralama: kesişimde kalan iki aday sayfa ilgi, güncellik, gelen bağlantılar gibi ölçütlerle puanlanır; Sayfa 1 (0,92) Sayfa 4'ün (0,78) önüne geçer (puanlar yalnızca örnektir).

1.4.5.5 Etkili arama

Arama motorları, sorguyu daraltmak için özel **operatörler** tanır. Doğru operatörü kullanmak, aranan bilgiyi binlerce ilgisiz sonuç arasından ayıklar.

Yöntem	Örnek	Amaç
Tam ifade	`"hesaplamalı düşünme"`	Sözcükleri yan yana ve bu sırayla ara.
Hariç tutma	`java -kahve`	İstenmeyen anlamı çıkar.
Site sınırı	`site:gov.tr burs`	Yalnız belirli alanlardaki sayfalarda ara.
Dosya türü	`filetype:pdf ağ güvenliği`	Belge biçimini sınırla.

Tablo 1.4.17: Arama operatörleri.

Operatörler birleştirilebilir. Örneğin yalnız üniversite sitelerinde, “staj başvurusu” ifadesini içeren PDF belgeleri bulmak için `site:edu.tr filetype:pdf "staj başvurusu"` yazılır. `java -kahve` sorgusu da ters dizin işlemlerinin tam karşılığıdır: “java” listesinden “kahve” listesindeki sayfalar çıkarılır.

Sık yapılan hata

Operatörleri tırnak içine almak ya da yerlerini karıştırmak. `"site:gov.tr burs"` yazılırsa bu dizi tam ifade olarak aranır ve operatör çalışmaz. `filetype:gov.tr` gibi bir yazım da anlamsızdır, çünkü `filetype:` bir dosya uzantısı, `site:` ise bir alan adı bekler.

1.4.5.6 Kaynak değerlendirme ve yapay zekâ çıktıları

Arama motoru bir sayfayı üst sıraya koyduğunda, o sayfanın doğru olduğunu söylemiş olmaz. Web'de herkes yayın yapabilir; yanlış, eski ya da kasıtlı olarak yanıltıcı bilgiler de aynı kolaylıkla bulunur. Bu yüzden bulunan her bilgi birkaç soruyla sınanmalıdır.

Soru	Neye bakılır?
Yazar	Kim yazmış? Konuda uzman bir kişi ya da sorumlu bir kurum mu?
Tarih	Ne zaman yayımlanmış ya da güncellenmiş? Konu için güncel mi?
Kaynak	Bilgi nereden alınmış? Alıntı ve bağlantılar verilmiş mi?
Doğrulama	Veriler ve iddialar kanıtlanabiliyor mu?
Çapraz kontrol	Aynı bilgiyi birbirinden bağımsız başka güvenilir kaynaklar da veriyor mu?

Tablo 1.4.18: Kaynak değerlendirme soruları.

Bir örnekle düşünelim. Bir forum sayfasında “Wi-Fi 6, 6 GHz bandını kullanır” yazdığını görelim. Yazarı belli değil, tarih yok, kaynak gösterilmemiş. Tablo 1.4.4'e ya da standardı yayımlayan kuruluşların sayfalarına bakıldığında 6 GHz'in Wi-Fi 6 ile değil Wi-Fi 6E ile geldiği görülür. Tek bir çapraz kontrol, yanlışı ortaya çıkarmaya yetmiştir.

Yapay zekâ çıktıları. Sohbet tabanlı yapay zekâ araçları akıcı ve kendinden emin metinler üretir. Bu metinler de yanlış olabilir; hatta var olmayan bilgiler ve var olmayan kaynaklar içerebilir. Araç, cümlenin doğru olup olmadığını değil, dil olarak akla yatkın olup olmadığını önceler. Bu yüzden yapay zekâdan alınan önemli bir bilgi, yukarıdaki ölçütlerle tıpkı bir web sayfası gibi doğrulanmalıdır. Özellikle verilen kaynakların gerçekten var olup olmadığı ve gerçekten o bilgiyi içerip içermediği denetlenmelidir.

1.4.6 Dijital hizmetler

İnternet altyapısının ve Web'in üzerinde gündelik hayatın büyük bir bölümü yürür: kamu işlemleri, iletişim, eğlence, alışveriş, dosya saklama ve yapay zekâ araçları. Bu kesim bu hizmetlerin bazılarının nasıl çalıştığını anlatır: bulut hizmet modelleri, elektronik ticaret türleri ve e-postanın yolculuğu.

1.4.6.1 Çevrim içi hizmetler

Alan	Örnekler
Kamu	E-Devlet, sağlık, vergi ve eğitim hizmetleri.
İletişim	Sosyal ağlar, anlık mesajlaşma ve topluluk platformları.
Medya	Video, müzik ve dijital yayın hizmetleri.
Ticaret	Mağazalar, pazar yerleri ve ödeme sistemleri.
Bulut	Dosya saklama, ortak çalışma ve uygulamalar.
Yapay zekâ	Metin, görüntü ve karar desteği sunan araçlar.

Tablo 1.4.19: Çevrim içi hizmet türleri.

1.4.6.2 Bulut hizmet modelleri ve CDN

Bulut bilişim (*cloud computing*), işlem gücünü, depolamayı ve yazılımı kendi makinelerinde barındırmak yerine bir sağlayıcıdan internet üzerinden kiralamaktır. Ne kadarının kiralandığına göre üç model vardır. Modelleri ayırt etmenin en kolay yolu, bir uygulamanın çalışması için gereken katmanları alttan üste sıralayıp her katmandan kimin sorumlu olduğuna bakmaktır.

Katman	Yerinde	IaaS	PaaS	SaaS
Uygulama	Kullanıcı	Kullanıcı	Kullanıcı	Sağlayıcı
Veri	Kullanıcı	Kullanıcı	Kullanıcı	Kullanıcı (içerik)
Çalışma zamanı	Kullanıcı	Kullanıcı	Sağlayıcı	Sağlayıcı
İşletim sistemi	Kullanıcı	Kullanıcı	Sağlayıcı	Sağlayıcı
Sunucu ve ağ donanımı	Kullanıcı	Sağlayıcı	Sağlayıcı	Sağlayıcı
Örnek	Kendi sunucu odası	Sanal sunucu kiralama	Uygulama barındırma platformu	E-posta, çevrim içi ofis

Tablo 1.4.20: Bulut hizmet modellerinde sorumluluk paylaşımı (“Kullanıcı”: hizmeti alan kurum ya da kişi).

- **Yerinde** (*on-premises*): kurum her şeyi kendisi kurar ve işletir; donanımdan uygulamaya kadar bütün sorumluluk kurumdadır.
- **IaaS** (*Infrastructure as a Service*, hizmet olarak altyapı): sağlayıcı fiziksel makineleri ve ağı verir; kullanıcı sanal sunucuya kendi işletim sistemini ve yazılımlarını kurar.

- **PaaS** (*Platform as a Service*, hizmet olarak platform): sağlayıcı işletim sistemini ve çalışma ortamını da yönetir; kullanıcı yalnız kendi uygulamasını yükler.
- **SaaS** (*Software as a Service*, hizmet olarak yazılım): uygulamanın kendisi hazır gelir; kullanıcı tarayıcıdan girip kullanır. Çevrim içi ofis programları ve web tabanlı e-posta bu modele örnektir.

Tablonun "Veri" satırına dikkat etmek gerekir. SaaS'ta bile verinin *içeriğinden* kullanıcı sorumludur: yanlışlıkla silinen bir belge, zayıf bir parolayla ele geçirilen bir hesap ya da gereksiz yere herkesle paylaşılan bir klasör sağlayıcının değil, kullanıcının sorunudur.

CDN. İçerik dağıtım ağı (*content delivery network*, CDN), aynı içeriğin kopyalarını dünyanın çeşitli yerlerindeki sunucularda tutar ve her kullanıcıya kendisine en yakın kopyadan sunar. Böylece hem gecikme azalır hem de asıl sunucunun yükü hafifler. Popüler video siteleri ve büyük haber siteleri içeriklerini bu yolla dağıtır.

Örnek 1.4.13

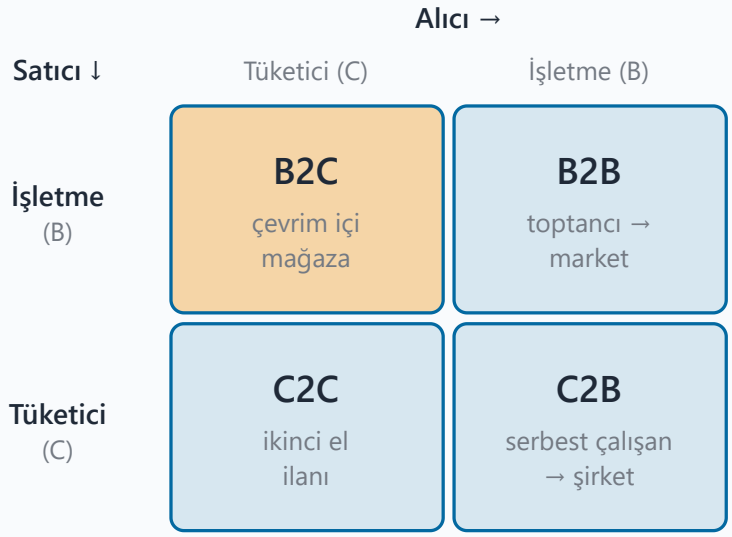
Giresun'daki bir kullanıcı, asıl sunucusu yaklaşık 2500 km uzaktaki bir şehirde bulunan bir siteye bağlanıyor. Site, İstanbul'da (yaklaşık 800 km uzakta) bir CDN sunucusu da kullanıyor. Fiberdeki ışık hızını 200.000 km/s alarak yalnız yol gecikmesini karşılaştıralım.

1. Asıl sunucu: tek yön 2500 ÷ 200.000 = 0,0125 s = 12,5 ms; gidiş-dönüş 25 ms.
2. CDN sunucusu: tek yön 800 ÷ 200.000 = 0,004 s = 4 ms; gidiş-dönüş 8 ms.
3. Bir sayfa açılırken en az dört gidiş-dönüş gerekiyorsa (DNS, TCP, TLS, HTTP): asıl sunucuda en az 4 × 25 = 100 ms, CDN'de en az 4 × 8 = 32 ms.

Kablolar hiçbir zaman düz bir çizgi izlemediği ve yönlendiricilerde bekleme olduğu için gerçek süreler daha uzundur; ama oran aşağı yukarı korunur. Mesafeyi kısaltmak, hattı genişletmenin yapamayacağı bir kazanç sağlar.

1.4.6.3 Elektronik ticaret

Elektronik ticaret (*e-commerce*), ürün ve hizmetlerin internet üzerinden alınıp satılmasıdır. Tarafların kim olduğuna göre dört tür ayırt edilir. Kısaltmalardaki B işletmeyi (*business*), C tüketiciyi (*consumer*) gösterir; ilk harf satıcıyı, son harf alıcıyı belirtir.



Şekil 1.4.31: Elektronik ticaretin dört türü: satıcı (satırlar) ve alıcı (sütunlar) işletme ya da tüketici olabilir.

- **B2C** (işletmeden tüketiciye): bir mağazanın internet sitesinden ayakkabı almak.
- **B2B** (işletmeler arası): bir lokantanın toptancıdan çevrim içi sipariş vermesi.
- **C2C** (tüketiciler arası): ikinci el ilan sitesinde bir öğrencinin eski kitabını başka bir öğrenciye satması.
- **C2B** (tüketiciden işletmeye): serbest çalışan bir tasarımcının bir şirkete logo tasarlaması ya da bir fotoğrafçının fotoğraflarını bir ajansa satması.

Hangi tür olursa olsun, işlemden önce birkaç şey denetlenmelidir: satıcının kimliği ve iletişim bilgisi, fiyatın kargo ve vergiler dâhil son hâli, teslimat süresi, iade koşulları ve ödeme sayfasının güvenliği. Ödeme sayfası adres çubuğunda doğru alan adıyla ve HTTPS ile görünmelidir. Piyasa fiyatının çok altındaki "kaçırılmayacak fırsatlar" dolandırıcılığın en yaygın işaretlerindendir.

1.4.6.4 E-posta adresi ve bileşenleri

Bir e-posta adresi iki parçadan ve bunları ayıran bir işaretten oluşur. `sercan.kulcu@giresun.edu.tr` adresinde:

- **Kullanıcı adı** (`sercan.kulcu`) posta kutusunu tanımlar.
- **@ işareti** kullanıcı adını alan adından ayırır.
- **Alan adı** (`giresun.edu.tr`) posta hizmetini yöneten kurumu gösterir. İletiyi hangi sunucuya teslim edeceğini bulmak için gönderen sunucu bu kısmı DNS'e sorar.

Bir ileti yazılırken doldurulan alanlar, iletinin kime nasıl ulaşacağını belirler.



Şekil 1.4.32: Yeni ileti penceresi: Kime, Bilgi ve Gizli alıcılar, konu, ileti metni, ek dosya ve Gönder düğmesi. Gizli alıcı öteki alıcılara görünmez.

Alan	Anlamı
Kime (To)	İletinin asıl alıcısı.
Bilgi (Cc)	İletiyi bilgi amacıyla alan kişiler; bütün alıcılar onları görür.
Gizli (Bcc)	Öteki alıcıların göremediği alıcılar.
Konu	İletinin kısa ve açıklayıcı başlığı.
Ek	İletiyle gönderilen dosya.
Yanıtla / Yönlendir	Gönderene yanıt verme ya da iletiyi başka birine iletme. “Tümünü yanıtla” görünen bütün alıcılara gider.

Tablo 1.4.21: E-posta bileşenleri.

Örnek 1.4.14

Bölüm sekreteri Elif bir duyuruyu şöyle gönderiyor: Kime: Can · Bilgi: Deniz ve Ece · Gizli: Bölüm başkanı. Can “Tümünü yanıtla” derse yanıt kimlere gider? Ece alıcı listesinde kimleri görür?

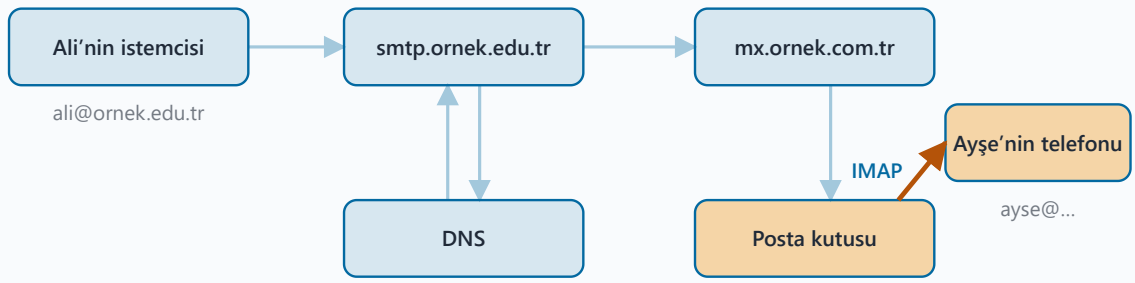
1. Her alıcı, iletinin başlığında Kime ve Bilgi alanlarını görür; Gizli alanı yalnız gönderende kalır.
2. Ece'nin gördüğü liste: Kime: Can; Bilgi: Deniz ve Ece. Bölüm başkanı listede yoktur.
3. Can “Tümünü yanıtla” derse yanıt gönderene ve görünen bütün alıcılara gider: Elif, Deniz ve Ece. Bölüm başkanı listede görünmediği için yanıtı almaz.

Gizli alan, büyük gruplara duyuru gönderirken alıcıların adreslerini birbirinden saklamak için de kullanılır. Yüz kişiye Bilgi alanıyla gönderilen bir duyuru, yüz adresin hepsini herkese açar.

1.4.6.5 Bir e-posta nasıl ulaşır?

E-posta, Web'den farklı protokollerle çalışan ayrı bir hizmettir. Gönderme işi **SMTP** ile, okuma işi **IMAP** ile yapılır. Arada en az iki posta sunucusu ve bir DNS sorgusu vardır. Örnekte Ali, `ayse@ornek.com.tr` adresine bir ileti gönderiyor.

1. Ali iletiyi yazar ve Gönder'e basar.
2. Ali'nin e-posta istemcisi iletiyi SMTP ile kendi kurumunun giden posta sunucusuna (`smtp.ornek.edu.tr`) teslim eder.
3. Sunucu, @ işaretinden sonraki alan adının (`ornek.com.tr`) posta sunucusunu DNS'e sorar. Bu soru bir **MX kaydı** sorgusudur (*mail exchanger*, posta değiştirici).
4. DNS yanıtlar: ornek.com.tr adresine gelen postaları `mx.ornek.com.tr` kabul eder.
5. İleti SMTP ile alıcının posta sunucusuna aktarılır.
6. Alıcı sunucu iletiyi spam ve virüs filtresinden geçirip Ayşe'nin **posta kutusuna** kaydeder.
7. Ayşe iletiyi IMAP ile okur. İleti sunucuda kaldığı için telefonda ve bilgisayarda aynı görünür.



Şekil 1.4.33: E-postanın yolu: Ali'nin istemcisi → smtp.ornek.edu.tr (DNS'ten MX kaydını sorar) → mx.ornek.com.tr → filtre → Ayşe'nin posta kutusu → IMAP ile Ayşe'nin telefonu.

Sık yapılan hata

SMTP'nin e-postayı okumak için de kullanıldığını sanmak. SMTP yalnız göndermek ve sunucular arasında aktarmak içindir. Posta kutusundaki iletiyi okumak için IMAP kullanılır. (Eski POP3 protokolü de okuma içindir, ama iletiyi çoğunlukla sunucudan indirip siler; bu yüzden birden çok cihazla kullanıma uygun değildir.)

1.4.7 Güvenlik

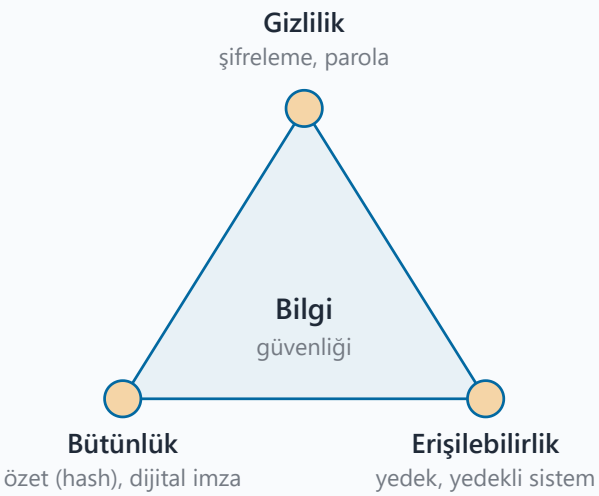
İnternetin açıklığı, onu bu kadar kullanışlı yapan özelliktir; aynı açıklık kötü niyetli kişilere de kapı aralar. Saldırıların büyük bölümü teknik açıklardan çok insanların dikkatsizliğinden ve alışkanlıklarından yararlanır. Bu kesim önce güvenliğin neyi korumaya çalıştığını tanımlar, sonra tehditleri ve bu tehditlere karşı alınabilecek önlemleri sırayla ele alır.

1.4.7.1 Bilgi güvenliğinin üç hedefi

Tanım 1.4.19

Bilgi güvenliği (*information security*) üç hedefi korumaya çalışır; İngilizce baş harflerinden ötürü bu üçlüye **CIA** denir.

- **Gizlilik** (*confidentiality*): Bilgiye yalnız yetkililer erişir.
- **Bütünlük** (*integrity*): Bilgi izinsiz değiştirilmez.
- **Erişilebilirlik** (*availability*): Bilgi ve hizmet gerektiğinde ulaşılabilirdir.



Şekil 1.4.34: Bilgi güvenliğinin üç hedefi ve her birinin tipik koruma yöntemi: gizlilik için şifreleme ve parola, bütünlük için özet ve dijital imza, erişilebilirlik için yedek ve yedekli sistem.

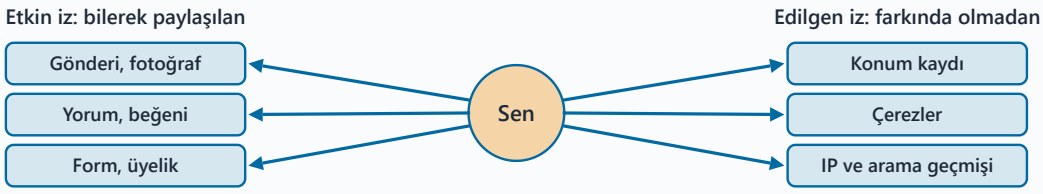
Her hedefin kendine özgü tehditleri vardır. Bir öğrencinin parolasının çalınıp not dökümünün başkasınca okunması **gizlilik** ihlalidir. Bir saldırganın öğrenci bilgi sistemine sızıp bir notu değiştirmesi **bütünlük** ihlalidir; bilgi gizli kalmış olsa bile artık doğru değildir. Fidye yazılımının sunucudaki dosyaları kilitlemesi ya da sistemin aşırı yükle çökertilmesi ise **erişilebilirlik** ihlalidir; kimse bilgiyi çalmamış ya da değiştirmemiştir, ama bilgiye ulaşılamaz.

1.4.7.2 Kişisel veri, KVKK ve e-Devlet

Tanım 1.4.20

Kişisel veri (*personal data*), kimliği belirli ya da belirlenebilir bir kişiyle ilgili her türlü bilgidir: ad, kimlik numarası, telefon, adres, fotoğraf, konum, sağlık bilgisi. **Dijital ayak izi** (*digital footprint*), çevrim içi etkinliklerin bıraktığı kayıtların toplamıdır. **Mahremiyet** ise bilginin kimlerle, hangi amaçla ve ne kadar süre paylaşıldığını denetleyebilmektir.

Dijital ayak izinin iki türü vardır (Şekil 1.4.35). **Etkin iz**, bilerek paylaşılan içeriktir: gönderiler, fotoğraflar, yorumlar, doldurulan formlar. **Edilgen iz** ise farkında olmadan bırakılır: konum kaydı, çerezler, IP adresi, arama geçmişi. İkinci tür çoğu zaman daha kapsamlıdır ve kullanıcının denetimi daha azdır.



Şekil 1.4.35: Dijital ayak izi. Solda etkin iz (bilerek paylaşılanlar: gönderi, fotoğraf, yorum, beğeni, form, üyelik); sağda edilgen iz (farkında olmadan bırakılanlar: konum kaydı, çerezler, IP ve arama geçmişi).

İnternete bir kez çıkan bilgiyi tamamen geri almak çoğu zaman mümkün değildir: kopyalanmış, ekran görüntüsü alınmış ya da arşivlenmiş olabilir. Bu yüzden bir bilgi paylaşılmadan önce üç soru sorulmalıdır: Bu bilgiyi paylaşmak gerekli mi? Kimler görebilecek? Daha sonra silinebilecek mi? Örneğin tatil fotoğrafını "şu an" paylaşmak, evin boş olduğunu herkese duyurmak demektir.

KVKK. Türkiye'de kişisel verilerin korunması 2016'da yürürlüğe giren **6698 sayılı Kişisel Verilerin Korunması Kanunu** (KVKK) ile düzenlenir. Kanun, kişisel verilerin işlenmesinde kişilerin haklarını korur. Veriyi işleyen kurum ya da şirket, yani **veri sorumlusu**, verinin hangi amaçla işlendiğini açıklamak ve verinin güvenliğini sağlamakla yükümlüdür. Kişi de kendisi hakkında hangi verilerin işlendiğini öğrenme ve yanlış verilerin düzeltilmesini isteme hakkına sahiptir.

e-Devlet güvenliği. Kamu hizmetleri e-Devlet üzerinden kimlik doğrulamasıyla sunulur. Bir e-Devlet hesabı tapu, vergi, sağlık ve eğitim kayıtlarına kapı açtığı için en iyi korunması gereken hesaplardandır. Güçlü ve başka hiçbir yerde kullanılmayan bir parola, iki adımlı doğrulama ve parolanın kimseyle paylaşılmaması temel kurallardır. Kimlik bilgisi isteyen e-posta ya da SMS'lerdeki bağlantılara tıklanmamalı; resmî siteye adres tarayıcıya elle yazılarak girilmelidir.

1.4.7.3 Kötü amaçlı yazılımlar

Kötü amaçlı yazılım (*malware*), bilgisayara ya da kullanıcısına zarar vermek için yazılmış programların genel adıdır. Türleri, nasıl yayıldıklarına ve ne yaptıklarına göre ayrılır.

Tür	Temel davranış	Tipik örnek
Virüs	Dosyalara bulaşır; kullanıcı bulaşmış dosyayı açınca yayılır.	USB bellekteki bulaşmış bir programı çalıştırmak.
Solucan	Ağ üzerinden kendi kendine yayılabilir; kullanıcı eylemi gerekmez.	Güncellenmemiş bir hizmetteki açığı kullanıp ağdaki öteki makinelere geçen program.
Truva atı	Yararlı görünür, ama zararlı bir işlev gizler.	"Ücretsiz" sunulan korsan bir oyunun içine gizlenmiş uzaktan erişim programı.
Fidye yazılımı	Dosyaları şifreleyip erişilemez hâle getirir ve açmak için ödeme ister.	Bütün belgelerin açılmaz hâle gelip ekranda ödeme talebi çıkması.
Casus yazılım	Kullanıcının etkinliklerini gizlice izler ve dışarı gönderir.	Tuş vuruşlarını kaydedip parolaları saldırgana yollayan program.

Tablo 1.4.22: Kötü amaçlı yazılım türleri.



Şekil 1.4.36: Virüs ile solucanın yayılma farkı. Solda virüs: bulaşmış dosya kullanıcı açınca ya da paylaşınca öteki dosyalara geçer. Sağda solucan: PC 1'den ağ üzerinden PC 2 ve PC 3'e kendiliğinden yayılır.

Virüs ile solucan arasındaki fark yayılma biçimidir: virüs bir kullanıcının bir dosyayı açmasını bekler; solucan hiç beklemez ve ağdaki açık makineleri kendisi bulur. Bu yüzden solucanlar çok kısa sürede binlerce makineye yayılabilir. Korunmanın en etkili yolu, işletim sistemini ve uygulamaları güncel tutmaktır; solucanların kullandığı açıkların çoğu, güncellemelerle aylar önce kapatılmış olur.

1.4.7.4 Sosyal mühendislik ve oltalama

Tanım 1.4.21

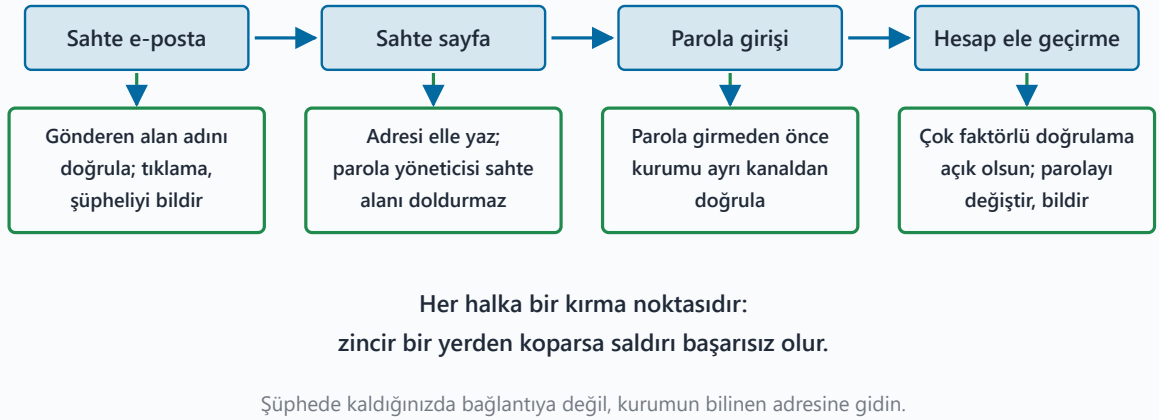
Sosyal mühendislik (*social engineering*), teknik açıklar yerine insanların güvenini, korkusunu ya da merakını kullanarak bilgi elde etme ya da bir eylem yaptırma yöntemidir. **Oltalama** (*phishing*), kendini güvenilir bir kurum gibi gösteren sahte iletilerle kullanıcının parolasını, kart bilgisini ya da doğrulama kodunu ele geçirme saldırısıdır.

Oltalama saldırısı dört halkalı bir zincir gibi ilerler: acil ya da cazip bir ileti gelir, ileti sahte bir bağlantı içerir, kullanıcı bağlantıdaki sayfaya bilgilerini girer ve hesap ele geçirilir. Her halkada saldırganın bir hamlesi, kullanıcının da bir kırma noktası vardır.

Halka	Saldırgan ne yapar?	Zincir nasıl kırılır?
1. Sahte e-posta	Resmî görünen, acil ve tehditkâr bir ileti gönderir; gönderen alan adı kurumunki değildir.	Gönderen adresi ve alan adı denetlenir; tıklamadan şüphelenilir, bildirilir.

Halka	Saldırgan ne yapar?	Zincir nasıl kırılır?
2. Sahte sayfa	Bağlantı, gerçeğine çok benzeyen bir giriş sayfasına götürür; alan adı farklıdır.	Adres elle yazılır; parola yöneticisi yanlış alan adında parolayı doldurmaz.
3. Parola girişi	Yazılan bilgiler doğrudan saldırganın sunucusuna gider; kurban çoğu zaman gerçek siteye yönlendirilir ve fark etmez.	Parola girmeden önce kuruma bilinen bir kanaldan doğrulama yapılır.
4. Hesap ele geçirme	Hesaba girer, parolayı değiştirir, aynı parolayı başka hesaplarda dener, rehberdekilere yeni oltalama iletileri gönderir.	Çok faktörlü doğrulama açıktır; parola çalınsa bile giriş reddedilir. Her hesapta farklı parola vardır.

Tablo 1.4.23: Oltalama saldırı zinciri ve her halkadaki kırma noktası.



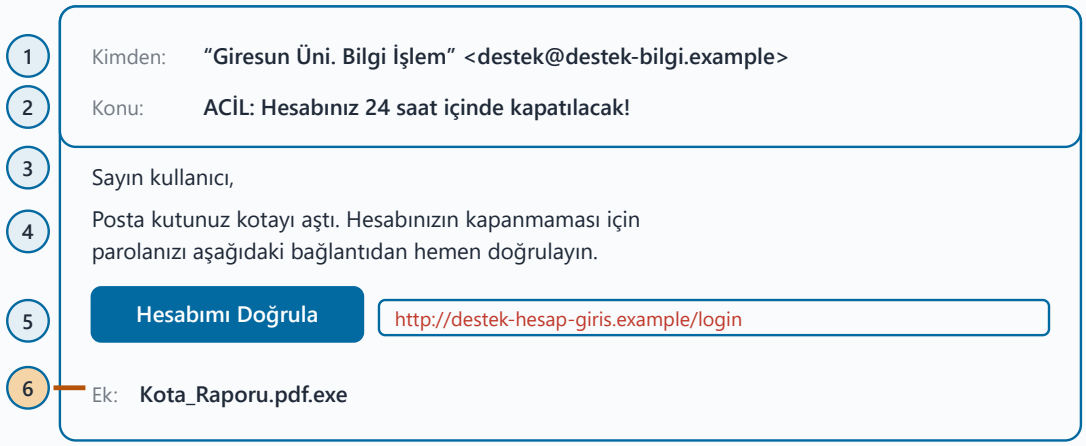
Şekil 1.4.37: Oltalama zincirinin dört halkası ve her halkanın altındaki kırma noktası. Tek bir halka kırılırsa saldırı başarısız olur.

Zincir benzetmesinin önemli bir sonucu vardır: korunmak için dört halkanın hepsinde kusursuz olmak gerekmez; tek bir halkayı kırmak yeter. Dikkatli bir kullanıcı ilk halkada iletiyi tanır. Gözden kaçırırsa, adresi elle yazma alışkanlığı ikinci halkada onu korur. Parolayı yine de girdiyse, çok faktörlü doğrulama dördüncü halkada hesabı kurtarır.

Oltalama yalnız e-postayla yapılmaz. SMS ile yapılan oltalama (“Kargonuz teslim edilemedi, adres bilgilerinizi güncelleyin”), sesli aramayla yapılan oltalama (“Bankanızdan arıyoruz, telefonunuza gelen kodu okur musunuz?”) ve sosyal medya mesajlarıyla yapılan oltalama da yaygındır. Ortak işaretler aynıdır: aciliyet, korku ya da cazip bir teklif, ve bir bağlantıya tıklatma ya da bir kod istetme.

1.4.7.5 Bir oltalama e-postasını incelemek

Örnek ileti, bir üniversitenin bilgi işlem birimi adına gönderilmiş gibi görünüyor. İletide altı ayrı uyarı işareti vardır (Şekil 1.4.38).



Şekil 1.4.38: İşaretli oltalama e-postası. 1: gönderen adresi, 2: acil konu, 3: genel hitap, 4: parola doğrulama isteği, 5: düğmenin gerçek adresi, 6: çift uzantılı ek dosya.

1. **Gönderen adresi.** Görünen ad "Giresun Üni. Bilgi İşlem"dir, ama adres `destek@destek-bilgi.example`'dır. Gerçek bir iletinin alan adı `giresun.edu.tr` olmalıydı. Görünen ad istenildiği gibi yazılabilir; asıl bakılacak yer adresin @'ten sonraki kısmıdır.
2. **Aciliyet ve tehdit.** Konu satırı "ACİL: Hesabınız 24 saat içinde kapatılacak!" der. Amaç, kullanıcının düşünmeden tıklamasıdır.
3. **Genel hitap.** "Sayın kullanıcı" yazar. Kurum kendi öğrencisini ya da çalışanını adıyla tanır.
4. **Parola doğrulama isteği.** Gerçek kurumlar parolayı e-postayla ya da bir bağlantı üzerinden "doğrulatmaz".
5. **Bağlantının gerçek adresi.** Düğmenin üzerine gelindiğinde (tıklamadan) görünen adres `destek-hesap-giris.example` alan adına gider; kurumla hiçbir ilgisi yoktur.
6. **Çift uzantılı ek.** Ek dosyanın adı `Kota_Raporu.pdf.exe`'dir. Son uzantı `.exe` olduğu için bu bir belge değil, çalıştırılabilir bir programdır. Windows uzantıları gizleyecek biçimde ayarlıysa kullanıcı yalnız "Kota_Raporu.pdf" görür.

Bu işaretlerden yalnız biri bile iletiyi şüpheli kılmaya yeter. Böyle bir iletiyle karşılaşıldığında en güvenli davranış, iletideki hiçbir bağlantıyı kullanmadan kurumun resmî sitesine adresi elle yazarak girmek ya da kurumu bilinen bir telefonla aramaktır. Duyuru gerçekse resmî sitede de görünür.

Sık yapılan hata

Bağlantıdaki adreste kurumun adının geçmesini yeterli saymak. `giresun.edu.tr.hesap-dogrula.example` adresi Giresun Üniversitesine değil, `hesap-dogrula.example` alanına aittir. Alan adı sağdan sola okunur (1.4.3.6); belirleyici olan, en sağdaki kayıtlı alandır.

1.4.7.6 Parola gücü

Bir saldırgan parolayı tahmin etmek için olası bütün parolaları sırayla deneyebilir; buna **kaba kuvvet** saldırısı denir. Saldırının ne kadar süreceği, olası parola sayısına bağlıdır. Parolada kullanılabilecek simge sayısı S, parolanın uzunluğu L ise olası parola sayısı

> $N = S^L$

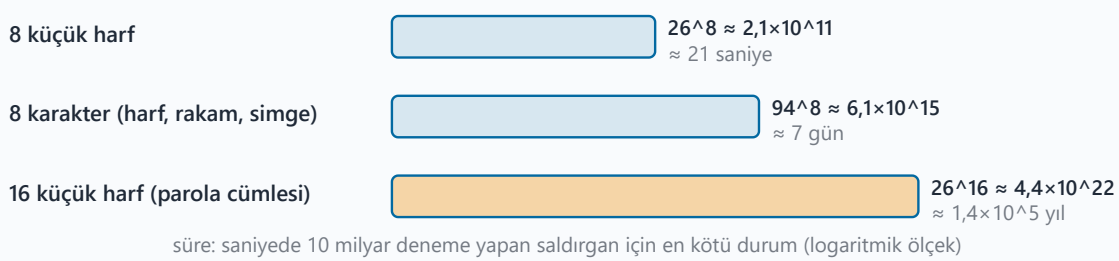
olur. Her karakter için S seçenek vardır ve L karakterin seçimleri birbirinden bağımsızdır; bu yüzden seçenekler çarpılır. Saldırgan saniyede D deneme yapabiliyorsa en kötü durumda süre $N \div D$ saniyedir. Bu notta $D = 10^{10}$, yani saniyede 10 milyar deneme alınır; özetlenmiş parolaları güçlü ekran kartlarıyla deneyen bir saldırgan için bu gerçekçi bir değerdir.

Örnek 1.4.15

Saniyede 10^{10} deneme yapan bir saldırgan için en kötü durum süresini üç parola türü için hesaplayalım: (a) 8 küçük harf, (b) harf, rakam ve simge içeren 8 karakter, (c) 16 küçük harften oluşan bir parola cümlesi. İngilizce alfabede 26 küçük harf vardır; klavyedeki bütün yazdırılabilir karakterler 94 tanedir.

1. (a) $N = 26^8 \approx 2{,}1 \times 10^{11}$. Süre: $2{,}1 \times 10^{11} \div 10^{10} \approx 21$ saniye.
2. (b) $N = 94^8 \approx 6{,}1 \times 10^{15}$. Süre: $6{,}1 \times 10^5$ s. Bir günde 86.400 s olduğundan $6{,}1 \times 10^5 \div 86.400 \approx 7$ gün.
3. (c) $N = 26^{16} \approx 4{,}4 \times 10^{22}$. Süre: $4{,}4 \times 10^{12}$ s. Bir yılda yaklaşık $3{,}16 \times 10^7$ s olduğundan $\approx 1{,}4 \times 10^5$ yıl, yani yaklaşık 140 bin yıl.

Simge çeşidini 26'dan 94'e çıkarmak süreyi 21 saniyeden 7 güne çıkarmıştır (yaklaşık 29.000 kat). Uzunluğu 8'den 16'ya çıkarmak ise yalnız küçük harfle 140 bin yıla ulaştırmıştır. Uzunluk, simge eklemekten çok daha etkilidir.



Şekil 1.4.39: Üç parola türünün olası parola sayısı ve saniyede 10 milyar deneme yapan bir saldırgan için en kötü durum süresi. Çubuklar logaritmik ölçektedir: her çubuğun boyu olası parola sayısının basamak sayısıyla orantılıdır.

Uzunluğun neden bu kadar etkili olduğu formülden görülür. S'yi artırmak tabanı büyütür; L'yi artırmak üssü büyütür. Üsse eklenen her karakter, olası sayıyı S katına çıkarır. 8 küçük harfe bir karakter daha eklemek sayıyı 26 katına; dört karakter daha eklemek $26^4 \approx 457.000$ katına çıkarır. 12 küçük harflik bir parola ($26^{12} \approx 9{,}5 \times 10^{16}$), 8 karakterlik karışık bir paroladan (94^8) yaklaşık 16 kat daha çok seçenek sunar ve akılda tutmak da daha kolaydır.

Hesabın bir varsayımı vardır: saldırgan bütün olasılıkları rastgele bir sırayla dener. Gerçek saldırganlar önce sözlükteki sözcükleri, sık kullanılan parolaları (`123456`, `qwerty`), doğum tarihlerini, takım adlarını ve bunların küçük değişikliklerini (`Galatasaray1905!`) dener. Bu tür parolalar uzun ve karışık görünse bile saniyeler içinde bulunur. Başka bir sitede kullanılıp sızdırılmış bir parola ise artık hiç güvenli değildir: saldırgan onu ilk denediği parolalar arasına koyar.

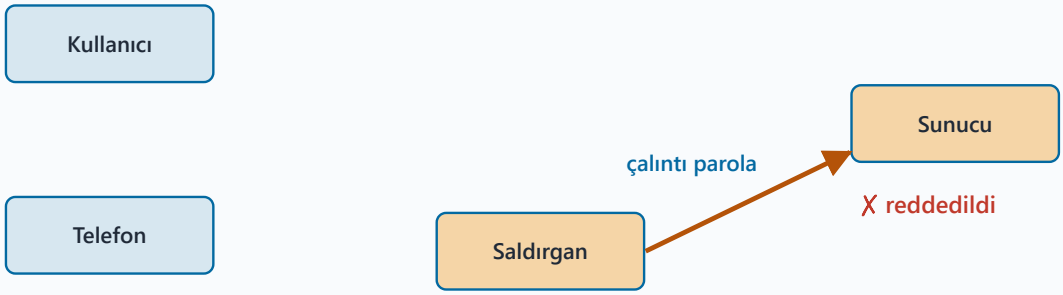
Parola yöneticisi (*password manager*) bu sorunların pratik çözümüdür. Her hesap için uzun ve rastgele bir parola üretir, şifreli bir kasada saklar ve gerektiğinde doldurur. Kullanıcı yalnız kasanın tek bir ana parolasını hatırlar. Parola yöneticisi bir oltalama sayfasında parolayı doldurmaz, çünkü kaydedilen alan adıyla sayfanın alan adı tutmaz; bu da ek bir koruma sağlar.

1.4.7.7 Çok faktörlü doğrulama

Tanım 1.4.22

Çok faktörlü doğrulama (*multi-factor authentication*, MFA), giriş için birbirinden farklı türde en az iki kanıt istenmesidir. Kanıt türleri: kullanıcının **bildiği** bir şey (parola, PIN), **elinde bulunan** bir şey (telefon, donanım anahtarı) ve **kendisi olan** bir şey (parmak izi, yüz).

1. Kullanıcı kullanıcı adını ve parolasını girer. Bu birinci faktördür: bildiği bir şey.
2. Parola doğruysa sunucu ikinci bir kanıt ister. Telefondaki doğrulama uygulaması 30 saniyede bir değişen 6 haneli bir kod üretir. Bu ikinci faktördür: elinde bulunan bir şey.
3. Kullanıcı kodu girer; sunucu kodu doğrular ve oturum açılır.
4. Parola sızdırılmış olsa bile saldırganın elinde kullanıcının telefonu olmadığı için giriş çoğunlukla reddedilir.



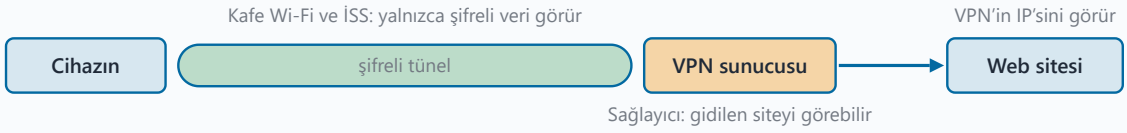
Şekil 1.4.40: Çalınmış parolayla giriş denemesi. Saldırgan parolayı bilir, ama telefondaki kodu üretemediği için sunucu girişi reddeder.

MFA kusursuz değildir. **Gerçek zamanlı oltalamada** saldırgan, kullanıcının sahte sayfaya girdiği kodu aynı anda gerçek siteye girer. **SIM değiştirme** saldırısında ise saldırgan operatörü kandırıp kurbanın telefon numarasını kendi SIM kartına aktarır ve SMS ile gelen kodları alır. Bu yüzden SMS kodu, doğrulama uygulamasından daha zayıf bir ikinci faktördür. En dayanıklı yöntemler **FIDO2/geçiş anahtarı** (*passkey*) ve **donanım anahtarıdır**: bu yöntemler sitenin alan adını kendileri denetler ve sahte bir sayfaya hiçbir şey vermez.

1.4.7.8 VPN

Tanım 1.4.23

VPN (*virtual private network*, sanal özel ağ), cihaz ile bir VPN sunucusu arasında internet üzerinden şifreli bir **tünel** kuran hizmettir. Cihazın bütün trafiği bu tünelden VPN sunucusuna gider; sunucu trafiği oradan asıl hedefine iletir.



Şekil 1.4.41: VPN tüneli. Kafe Wi-Fi'ı ve İSS yalnız şifreli veriyi görür; web sitesi kullanıcının değil, VPN sunucusunun IP adresini görür; VPN sağlayıcısı ise kullanıcının hangi sitelere gittiğini görebilir.

VPN'in ne yaptığını ve ne yapmadığını ayırmak önemlidir:

VPN yapar	VPN yapmaz
Halka açık Wi-Fi'da yerel dinlemeyi zorlaştırır.	Mutlak anonimlik sağlamaz; VPN sağlayıcısı trafiğin hangi sitelere gittiğini görebilir.

VPN yapar	VPN yapmaz
Sitelere kullanıcının IP adresi yerine VPN sunucusunun IP adresini gösterir.	Hesaba giriş yapıldıysa ya da çerezler duruyorsa siteler kullanıcıyı yine tanır.
Kurumsal ağa uzaktan güvenli erişim sağlar (örneğin kampüs dışından kütüphane veri tabanlarına).	Kötü amaçlı yazılımı ve oltalamayı engellemez.

Tablo 1.4.24: VPN'in yaptıkları ve yapmadıkları.

VPN kullanmak, trafiği İSS'nin ya da kafe ağının görmesinden alıp VPN sağlayıcısının görmesine aktarmak demektir. Sağlayıcıların "kayıt tutmuyoruz" iddiaları kendi beyanlarıdır ve dışarıdan doğrulanması zordur. Bu yüzden VPN, güvenilir bir sağlayıcı seçmeyi gerektirir. VPN HTTPS'in yerini de tutmaz: tünel yalnız VPN sunucusuna kadardır; sunucudan siteye giden kısım ancak HTTPS ile şifrelidir. İkisi birbirini tamamlar.

1.4.7.9 Güvenlik duvarı

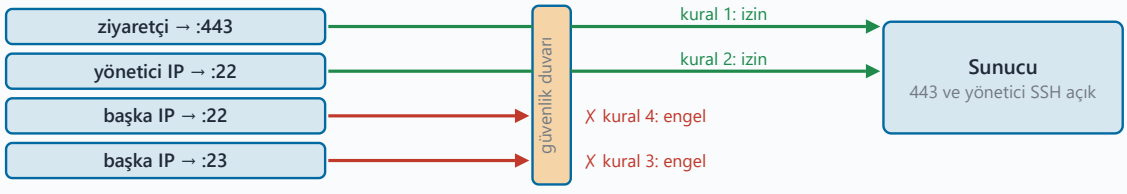
Tanım 1.4.24

Güvenlik duvarı (*firewall*), gelen ve giden bağlantılara kurallara göre izin veren ya da onları engelleyen yazılım ya da donanımdır. Kurallar çoğunlukla IP adresine ve port numarasına dayanır ve **yukarıdan aşağı sırayla** denenir: ilk eşleşen kural uygulanır, sonrakilere bakılmaz.

Bir web sunucusunu koruyan güvenlik duvarının kuralları Tablo 1.4.25'teki gibi olsun. Yöneticinin IP adresi 192.0.2.50'dir.

Sıra	Yön	Hedef port	Kaynak	Eylem
1	Gelen	443 (HTTPS)	Herkes	İzin ver (web sunucusu)
2	Gelen	22 (SSH)	Yalnız yönetici IP'si	İzin ver
3	Gelen	23 (Telnet)	Herkes	Engelle (şifresiz, eski protokol)
4	Gelen	Diğer tümü	Herkes	Engelle (varsayılan ret)

Tablo 1.4.25: Güvenlik duvarı kural örneği.



Şekil 1.4.42: Gelen dört bağlantı ve güvenlik duvarının kararları: ziyaretçinin 443'e ve yöneticinin 22'ye bağlantısı geçer; başka bir IP'nin 22'ye bağlantısı 4. kuralla, 23'e bağlantısı 3. kuralla engellenir.

Örnek 1.4.16

Aşağıdaki gelen bağlantılar için hangi kural eşleşir ve karar ne olur? (a) 198.51.100.7 → 443, (b) 192.0.2.50 → 22, (c) 198.51.100.7 → 22, (d) 192.0.2.50 → 3389 (uzak masaüstü).

1. (a) Kural 1'in portu (443) tutar ve kaynak "herkes"tir: **kural 1, izin**.
2. (b) Kural 1'in portu tutmaz. Kural 2'nin portu (22) ve kaynağı (yönetici IP'si) tutar: **kural 2, izin**.
3. (c) Kural 2'nin portu tutar ama kaynak yönetici değildir; kural uygulanmaz. Kural 3'ün portu 23'tür, tutmaz. Kural 4 her şeyi kapsar: **kural 4, engel**.
4. (d) Kaynak yönetici olsa bile 3389 için izin veren bir kural yoktur. Kural 1, 2 ve 3'ün portları tutmaz: **kural 4, engel**.

(d) şıkkı "varsayılan ret" ilkesini gösterir: açıkça izin verilmeyen her şey engellenir. Yönetici bile, yalnız kendisi için açılmış porta bağlanabilir.

İyi uygulama budur: önce her şeyi engellemek, yalnız gerekli portlara izin vermek. Telnet (23) şifresiz çalıştığı için parolayı düz metin olarak taşır; bu yüzden açıkça kapatılmıştır. SSH (22) ise yönetim için gereklidir ama yalnız yöneticinin adresine açılmıştır. Kural sırası da önemlidir: 4. kural (her şeyi engelle) listenin başına yazılsaydı, hiçbir bağlantı 1. ve 2. kurallara ulaşamazdı.

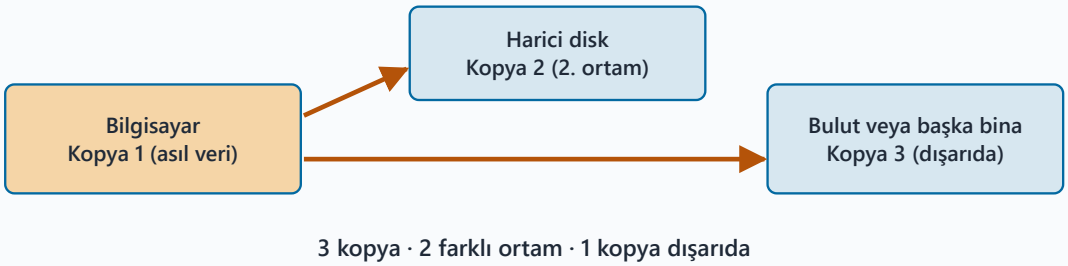
Güvenlik duvarının bir sınırı vardır: ağa girişi sınırlar, ama kullanıcının kendi isteğiyle indirip çalıştırdığı zararlı dosyayı tek başına durdurmaz. Kullanıcı bir web sitesinden (443 üzerinden, yani izin verilen bir bağlantıyla) bir Truva atı indirirse duvar bunu olağan bir web trafiği olarak görür.

1.4.7.10 Yedekleme: 3-2-1 kuralı

Disk arızası, hırsızlık, yangın, yanlışlıkla silme ya da fidye yazılımı… Verinin kaybolmasının pek çok yolu vardır ve bunların bir kısmına karşı hiçbir önlem tam koruma sağlamaz. Son savunma hattı yedektir.

Tanım 1.4.25

3-2-1 yedekleme kuralı: verinin en az **3 kopyası** bulunur (asıl veri ve en az iki yedek); kopyalar en az **2 farklı türde ortamda** tutulur; kopyalardan en az **1'i başka bir konumdadır** (bulut ya da başka bir bina).



Şekil 1.4.43: 3-2-1 kuralı: bilgisayardaki asıl veri (kopya 1), harici diskteki yedek (kopya 2, ikinci ortam) ve bulutta ya da başka bir binada tutulan yedek (kopya 3, dışarıda).

Kuralın her parçası farklı bir tehdide karşıdır. **Üç kopya**, bir kopyanın bozulması ihtimaline karşıdır. **İki ortam**, aynı türden iki ortamın (örneğin aynı seriden iki diskin) aynı nedenle aynı anda bozulmasına karşıdır. **Bir kopyanın dışarıda olması**, yangın, hırsızlık ya da su baskını gibi bütün odayı etkileyen olaylara karşıdır.

Örnek 1.4.17

Mert proje dosyalarını dizüstünde tutuyor. Her akşam dosyaları masasının çekmecesindeki bir USB belleğe kopyalıyor; ayrıca dizüstündeki ikinci bir klasöre de bir kopya alıyor. 3-2-1 kuralına göre durumu değerlendirelim.

1. Kopya sayısı: dizüstündeki asıl dosyalar, dizüstündeki ikinci klasör, USB bellek. Üç kopya vardır.
2. Ortam sayısı: dizüstü diski ve USB bellek. İki farklı ortam vardır. Ama dizüstündeki iki kopya aynı diskte durduğu için disk bozulduğunda ikisi birden gider; bu kopya, ortam açısından ayrı sayılmaz.
3. Dışarıdaki kopya: yoktur. Çanta çalınırsa ya da oda yanarsa dizüstü ve USB bellek birlikte kaybolur.

Eksik, farklı konumdaki kopyadır. Dosyaları bir bulut depolama hizmetine ya da evdeki bir diske yedeklemek kuralı tamamlar.

İki uyarı daha yapılmalıdır. Birincisi: yedek ancak geri yüklenebiliyorsa işe yarar. Yıllarca alınan yedeklerin, gerektiği gün açılamadığı çok görülmüş bir durumdur; bu yüzden ara sıra bir dosyayı yedekten geri yükleyip denemek gerekir. İkincisi: bilgisayara sürekli bağlı bırakılan bir disk, fidye yazılımı bulaştığında asıl dosyalarla birlikte şifrelenebilir. Yedek diskin yalnız yedek alınırken bağlanması bu riski azaltır. Eşitleme yapan bulut klasörlerinde de benzer bir risk vardır: yanlışlıkla silinen ya da şifrelenen dosya buluttaki kopyaya da yansıyabilir. Sürüm geçmişi tutan bir hizmet bu durumda eski sürüme dönmeyi sağlar.

1.4.8 Özet

- **İnternet**, ağları ortak protokollerle (TCP/IP) birleştiren, merkezî olmayan küresel ağlar ağıdır. Türkiye 12 Nisan 1993'te ODTÜ–NSFNET hattıyla internete bağlandı.
- Kullanıcı internete bir **İSS** ve bir erişim hattı (fiber, DSL, mobil, uydu) üzerinden bağlanır. **Bant genişliği** saniyede taşınan bit, **gecikme** ilk bitin varış süresidir; indirme süresi = MB × 8 ÷ Mbit/s.
- **İstemci** ister, **sunucu** karşılar, **protokol** kuralları koyar. **Switch** yerel ağ içinde MAC adresine, **yönlendirici** ağlar arasında IP adresine göre çalışır. Ağlar kapsamlarına göre PAN, LAN, MAN ve WAN'dır; Wi-Fi'da frekans yükseldikçe hız artar, menzil azalır; güncel güvenlik protokolleri WPA2 ve WPA3'tür.
- Protokoller dört katmanda çalışır ve her katman kendi başlığını ekler (**kapsülleme**). Veri **paketlere** bölünür; paketler farklı yollardan ve sırasız gelebilir, sıralama ve yeniden gönderme TCP'nin işidir.
- Yönlendirici, hedefi kapsayan satırlardan **en uzun önekliyi** seçer ve **TTL**'yi 1 azaltır; TTL sıfırlanınca paket atılır.
- **IPv4** adresi 32 bit (dört sekizli, 0–255), **IPv6** adresi 128 bittir. Yerel adresler (10/8, 172.16/12, 192.168/16) internette yönlendirilmez; yönlendirici **NAT** ile onları kendi genel adresine ve bir porta çevirir.
- **DNS** alan adını IP adresine çevirir: çözümleyici kökten üst düzey alana ve yetkili sunucuya iner, öğrendiklerini TTL süresince önbellekte tutar. Alan adı sağdan sola okunur.
- **URL**; protokol, alan adı, port, yol, sorgu ve parçadan oluşur; parça sunucuya gönderilmez. **Port** hizmeti seçer: 80 HTTP, 443 HTTPS, 22 SSH, 53 DNS.
- **TCP** üçlü el sıkışmayla kurulur, dört bölütle kapanır; ack = alınan son bayt + 1. **HTTPS**, TLS ile şifreler ve sertifikayla sunucunun kimliğini doğrular; kilit sitenin dürüstlüğünü garanti etmez.
- **Simetrik şifreleme** hızlıdır, **asimetrik şifreleme** anahtar dağıtımını çözer; HTTPS ikisini birlikte kullanır. **Özet** şifreleme değildir, geri çözülemez.
- **Web**, internet üzerindeki hizmetlerden biridir; sayfa HTML, CSS ve JavaScript'ten oluşur. **Arama motoru** önceden tarar, **ters dizin** kurar ve sorguda listeleri kesiştirir. Bulunan bilgi ve yapay zekâ çıktıları yazar, tarih, kaynak ve çapraz kontrolle sınanır.
- Bulut modelleri IaaS, PaaS ve SaaS'tır; CDN gecikmeyi azaltır. E-posta SMTP ile gönderilir, MX kaydıyla alıcı sunucu bulunur, IMAP ile okunur.
- Bilgi güvenliği **gizlilik, bütünlük ve erişilebilirliği** korur. **Oltalama** dört halkalı bir zincirdir ve tek halkayı kırmak saldırıyı durdurur. Olası parola sayısı S^L'dir; uzunluk simgeden etkilidir. Parola yöneticisi, **çok faktörlü doğrulama**, güncelleme, güvenlik duvarında varsayılan ret ve **3-2-1 yedekleme** temel önlemlerdir; VPN tüneli şifreler ama anonimlik sağlamaz.

1.4.9 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki bölümlerdeki kullanımları terim sözlüğünde bulunur.

bilgisayar ağı *computer network* · internet *Internet* · internet servis sağlayıcısı (İSS) *Internet service provider* · bant genişliği *bandwidth* · gecikme *latency* · istemci *client* · sunucu *server* · protokol *protocol* · switch *switch* · yönlendirici *router* · modem *modem* · güvenlik duvarı *firewall* · yerel ağ (LAN) *local area network* · geniş alan ağı (WAN) *wide area network* · Wi-Fi *Wi-Fi* · katman *layer* · başlık *header* · paket *packet* · bölüt *segment* · sıra numarası *sequence number* · MAC adresi *MAC address* · yönlendirme tablosu *routing table* · en uzun önek eşleşmesi *longest prefix match* · varsayılan ağ geçidi *default gateway* · TTL *time to live* · IP adresi *IP address* · sekizli *octet* · yerel IP adresi *private IP address* · genel IP adresi *public IP address* · NAT *network address translation* · alan adı *domain name* · DNS çözümleyicisi *DNS resolver* · kök sunucu *root server* · yetkili sunucu *authoritative server* · önbellek *cache* · üst düzey alan *top-level domain* · URL *Uniform Resource Locator* · port *port* · üçlü el sıkışma *three-way handshake* · onay (ACK) *acknowledgment* · TLS *Transport Layer Security* · sertifika *certificate* · simetrik şifreleme *symmetric encryption* · asimetrik şifreleme *asymmetric encryption* · özet (hash) *hash* · oturum anahtarı *session key* · Web *World Wide Web* · web tarayıcısı *web browser* · durum kodu *status code* · web robotu *web crawler* · ters dizin *inverted index* · bulut bilişim *cloud computing* · içerik dağıtım ağı (CDN) *content delivery network* · elektronik ticaret *e-commerce* · posta sunucusu *mail server* · bilgi güvenliği *information security* · kişisel veri *personal data* · dijital ayak izi *digital footprint* · kötü amaçlı yazılım *malware* · fidye yazılımı *ransomware* · sosyal mühendislik *social engineering* · oltalama *phishing* · parola yöneticisi *password manager* · çok faktörlü doğrulama *multi-factor authentication* · geçiş anahtarı *passkey* · VPN *virtual private network* · yedekleme *backup*

1.4.10 Kendini Dene

1. 25 Mbit/s'lik bir hatta 750 MB'lık bir video en iyi durumda kaç dakikada iner? Sekizle çarpma unutulursa hangi sonuç bulunur?

Cevap

750 MB = 750 × 8 = 6000 Mbit; 6000 ÷ 25 = 240 s = 4 dakika. Sekizle çarpma unutulursa 750 ÷ 25 = 30 s bulunur; sonuç 8 kat kısa çıkar.

2. 10.20.30.40 adresinin ikili gösterimi nedir? Bu adres yerel (özel) bir adres midir?

> **Cevap**
>
> 10 = 8 + 2 → 00001010; 20 = 16 + 4 → 00010100; 30 = 16 + 8 + 4 + 2 → 00011110; 40 = 32 + 8 → 00101000. Adres: `00001010.00010100.00011110.00101000`. İlk sekizli 10 olduğu için 10.0.0.0/8 bloğundadır; yerel bir adrestir ve internette yönlendirilmez.

3. Bir yönlendiricinin tablosu: 10.0.0.0/8 → A, 10.1.0.0/16 → B, 10.1.2.0/24 → C, 0.0.0.0/0 → D. 10.1.2.7, 10.1.9.9, 10.200.0.1 ve 8.8.8.8 hedefli paketler hangi komşuya gider?

> **Cevap**
>
> 10.1.2.7 dört satıra da uyar; en uzun önek /24'tür → **C**. 10.1.9.9, /8, /16 ve /0'a uyar (üçüncü sekizli 9, /24'ü tutmaz) → **B**. 10.200.0.1 yalnız /8 ve /0'a uyar → **A**. 8.8.8.8 yalnız varsayılan yola uyar → **D**.

4. Bir paket TTL = 5 ile gönderiliyor; hedef, yolda 7 yönlendirici ötededir. Ne olur?

> **Cevap**
>
> Her yönlendirici TTL'yi 1 azaltır: 1. yönlendiricide 4, 2.'de 3, 3.'te 2, 4.'te 1, 5.'te 0 olur. 5. yönlendirici paketi atar ve gönderene "süre aşıldı" iletisi döndürür. Paket hedefe ulaşmaz. traceroute bu iletileri kullanarak yolu keşfeder.

5. `https://ders.ornek.edu.tr:8443/odev/liste?hafta=5&sinif=b#teslim` adresinde bağlanılan port hangisidir? Sunucuya hangi kısım gönderilir?

> **Cevap**
>
> Port açıkça yazıldığı için HTTPS'in varsayılanı 443 değil, **8443** kullanılır. Alan adı ders.ornek.edu.tr'dir ve DNS ile IP'ye çevrilir. Sunucuya giden istek yol ve sorgudan oluşur: `/odev/liste?hafta=5&sinif=b` (iki ad=değer çifti). `#teslim` parçası sunucuya gönderilmez.

6. Bir çözümleyicinin önbelleğinde ".tr sunucusu" ve "ornek.edu.tr'nin yetkili sunucusu" kayıtları var, başka kayıt yok. `mail.ornek.edu.tr` ve `www.baska.edu.tr` soruları için kaç dış sorgu gerekir?

> **Cevap**
>
> `mail.ornek.edu.tr` : ornek.edu.tr'nin yetkili sunucusu bilindiği için doğrudan ona sorulur; **1 sorgu**. `www.baska.edu.tr` : baska.edu.tr'nin yetkili sunucusu bilinmez, ama .tr sunucusu bilinir; kök atlanır, önce .tr'ye sonra yetkiliye sorulur; **2 sorgu**.

7. İstemcinin başlangıç sıra numarası 7000'dir. Üçlü el sıkışmadan sonra 500 baytlık veri gönderip bağlantıyı kapatıyor. Sunucunun SYN + ACK'teki ack değeri, veri bölütünün seq değeri, verinin onayı ve istemcinin FIN'ine verilen onay nedir?

Cevap

SYN + ACK: ack = 7001 (SYN bir numara harcar). Veri: seq = 7001; baytlar 7001–7500. Onay: ack = 7501. İstemcinin FIN'i seq = 7501'dir; onayı ack = 7502'dir, çünkü FIN de bir numara harcar.

8. Saniyede 10^{10} deneme yapan bir saldırgan için yalnız rakamlardan oluşan 10 karakterlik bir parola ile 10 küçük harften oluşan bir parolanın en kötü durum süreleri nedir?

Cevap

Rakamlar: 10^{10} olasılık; $10^{10} \div 10^{10}$ = **1 saniye**. Küçük harfler: $26^{10} \approx 1{,}41 \times 10^{14}$ olasılık; $\approx 1{,}41 \times 10^{4}$ s $\approx$ **3,9 saat**. Aynı uzunlukta simge çeşidi 10'dan 26'ya çıkınca süre yaklaşık 14.000 katına çıkar; ama ikisi de güçlü sayılmaz. Güçlü bir parola için uzunluk artırılmalıdır.

9. Aşağıdaki olaylarda bilgi güvenliğinin hangi hedefi ihlal edilmiştir? (a) Bir hastanenin randevu sistemi, sahte isteklerle yüklenip saatlerce açılmıyor. (b) Hasta kayıtlarını içeren şifresiz bir USB bellek otobüste unutuluyor. (c) Bir e-ticaret sitesinde bir saldırgan ürün fiyatlarını değiştiriyor.

Cevap

(a) **Erişilebilirlik**: bilgi ne çalınmış ne değiştirilmiştir, ama ulaşılamaz. (b) **Gizlilik**: belleği bulan herkes kayıtları okuyabilir; şifreleme bu riski önlerdi. (c) **Bütünlük**: bilgi izinsiz değiştirilmiştir.

10. Tablo 1.4.25'teki kurallarda 4. kural (“diğer tümü, engelle”) listenin en başına taşınırsa 198.51.100.7 → 443 ve 192.0.2.50 → 22 bağlantıları ne olur?

Cevap

Kurallar yukarıdan aşağı denenir ve ilk eşleşen uygulanır. Yeni ilk kural her bağlantıyı kapsadığı için iki bağlantı da **engellenir**; izin veren kurallara hiç sıra gelmez. Web sitesi dışarıdan erişilemez olur, yönetici de sunucuya bağlanamaz. Varsayılan ret kuralı her zaman listenin sonunda durmalıdır.

1.4.11 İleri okuma

- J. F. Kurose ve K. W. Ross, *Computer Networking: A Top-Down Approach* (8. baskı, 2021): bu notta izlenen "yukarıdan aşağı" sırayla uygulama katmanından başlayıp HTTP, DNS, TCP, IP yönlendirmesi ve bağlantı katmanını ayrıntılı ve bol şekilli anlatır.
- A. S. Tanenbaum, N. Feamster ve D. J. Wetherall, *Computer Networks* (6. baskı, 2021): ağları fiziksel katmandan yukarı doğru inceler; elektrik-elektronik öğrencileri için modülasyon ve kablosuz iletim bölümleri özellikle yararlıdır.
- R. Anderson, *Security Engineering* (3. baskı, 2020): oltalama, parolalar, kimlik doğrulama ve şifrelemenin gerçek sistemlerde neden başarısız olduğunu çok sayıda vaka üzerinden anlatır.
- Internet Society, *A Brief History of the Internet*: ARPANET'ten bugüne internetin gelişimini, onu tasarlayan kişilerin kaleminden kısa ve okunaklı biçimde aktarır.

1.4.12 Kaynaklar

1. Internet Society, *A Brief History of the Internet*, B. M. Leiner ve diğerleri, 1997.
2. IETF, *Internet Standards and RFC documents*; özellikle RFC 1918 (yerel adres blokları, 1996), RFC 5737 (belgelerde kullanılan IPv4 adresleri, 2010), RFC 8446 (TLS 1.3, 2018) ve RFC 9293 (TCP, 2022).
3. W3C, *Web Standards*.
4. NIST, *Cybersecurity and Privacy guidance*; özellikle SP 800-63B, *Digital Identity Guidelines: Authentication and Lifecycle Management*.
5. 6698 sayılı Kişisel Verilerin Korunması Kanunu, Resmî Gazete, 7 Nisan 2016, sayı 29677.
6. TÜBİTAK ULAKBİM, *Faaliyet Raporu*, 2001 ve 2012 (ULAKBİM'in 1996'da kuruluşu ve ULAKNET'in 1997'den itibaren işletilmesi).
7. J. F. Kurose, K. W. Ross, *Computer Networking: A Top-Down Approach*, 8. baskı, Pearson, 2021.

KISIM I · BÖLÜM 1.5

Hesaplama ve Hesaplamalı Düşünme

Bilgisayar çoğu zaman hızlı bir hesap makinesi gibi düşünülür. Oysa bir arama motorunun sonuç listesi, bir haritanın önerdiği yol ve bir telefonun yüz tanıması da hesaplamadır. Hepsinde veri belirli kurallara göre işlenir. Not, önce bu geniş hesaplama kavramını ve bilgisayar biliminin neyle uğraştığını anlatır. Ardından hesaplamanın kısa tarihini, sabit programlı makinelerden depolanan programlı bilgisayarlara geçişi ve bu geçişin kuramsal temeli olan Turing makinesini ele alır. Bir Turing makinesinin ikili bir sayıya 1 eklemesi adım adım izlenir. Bilginin iki türü, "ne" bilgisi ile "nasıl" bilgisi, karekök bulma ve Öklid algoritması üzerinden karşılaştırılır. Son kesim, büyük bir problemi bilgisayarın çözebileceği biçime getiren dört düşünme aracını, yani ayrıştırmayı, örüntü tanımayı, soyutlamayı ve algoritma tasarımını örneklerle gösterir.

ÖĞRENME ÇIKTILARI

Hesaplama

Hesaplamayı verinin kurallara göre işlenmesi olarak açıklamak; aritmetik dışındaki hesaplama türlerine örnek vermek; bir işin girdisini, işlemini, çıktısını ve kullandığı kaynakları belirlemek.

Hesaplama modelleri

Sabit programlı ve depolanan programlı makineleri ayırt etmek; bir Turing makinesinin çalışmasını bant, başlık, durum ve geçiş kurallarıyla adım adım izlemek; bilgisayarların sınırlarını sıralamak.

Bilgi ve algoritma

Bildirimsel ve buyurgan bilgiyi ayırt etmek; karekök bulma yöntemini ve Öklid algoritmasını izleme tablosuyla izlemek; bir algoritmanın neden durduğunu gerekçelendirmek.

Hesaplamalı düşünme

Ayrıştırma, örüntü tanıma, soyutlama ve algoritma tasarımını gerçek bir probleme uygulamak; bir algoritma için normal, sınır ve geçersiz girdilerden oluşan test durumları seçmek.

1.5.1 Hesaplama

"Hesaplama" sözcüğü çoğu kişiye okul yıllarındaki dört işlemi hatırlatır: toplama, çıkarma, çarpma ve bölme. Bilgisayar biliminde kavramın kapsamı çok daha geniştir. Bir sözlükte kelime aramak, sınıf listesini soyadına göre sıralamak, bir trafik ışığının ne zaman yeşile döneceğine karar vermek de hesaplamadır. Bu kesim hesaplamanın ortak yapısını ve hesaplamayı mümkün kılan parçaları tanıtır.

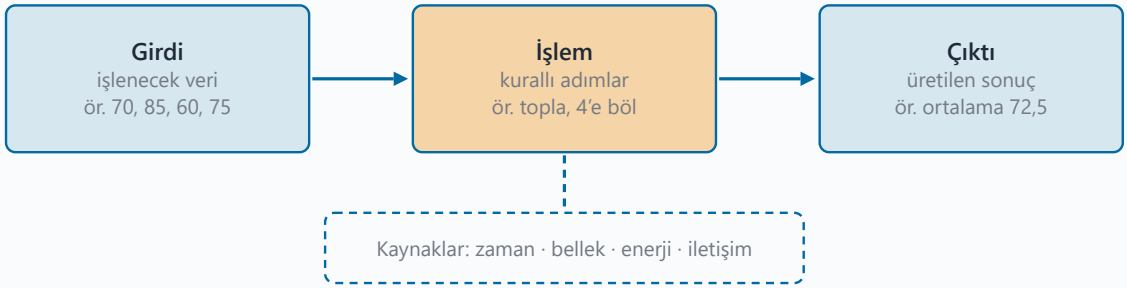
1.5.1.1 Hesaplama nedir?

Bir öğrencinin dört sınav notunun ortalamasını bulmak, hesaplamanın en yalın örneklerinden biridir. Elde notlar vardır: 70, 85, 60 ve 75. Bu sayılar işlenecek malzemedir. Malzemeye uygulanacak kurallar da bellidir: önce notlar toplanır, sonra toplam 4'e bölünür. Kurallar uygulanınca bir sonuç çıkar: 72,5. Hesaplamanın her örneğinde bu üç parça bulunur: işlenecek veri, uygulanacak kurallar ve üretilen sonuç.

Tanım 1.5.1

Hesaplama (*computation*), verinin belirli kurallara göre işlenerek bir sonuca dönüştürülmesidir. İşlenecek veriye **girdi** (*input*), kuralların adım adım uygulanmasına **işlem**, üretilen sonuca **çıktı** (*output*) denir.

Tanımda iki sözcük özellikle önemlidir. Birincisi "kurallar"dır. Hesaplama keyfe göre yapılmaz; aynı girdiye aynı kurallar uygulanınca her seferinde aynı çıktı elde edilir. Ortalamayı bugün hesaplayan da, yarın hesaplayan da 72,5 bulur. İkincisi "işlenmesi"dir. Veriyi yalnızca saklamak ya da göstermek hesaplama sayılmaz; verinin bir kurala göre dönüştürülmesi gerekir.



Şekil 1.5.1: Hesaplamanın üç parçası. Girdi (70, 85, 60, 75) kurallı adımlarla işlenir (topla, 4'e böl) ve çıktı (ortalama 72,5) üretilir. İşlem sırasında zaman, bellek, enerji ve iletişim kaynakları kullanılır.

Şekil 1.5.1 bu yapıyı gösterir. Kesik çizgili kutudaki kaynaklar ilk bakışta gözden kaçar ama her hesaplama onlara muhtaçtır. İşlem **zaman** alır: dört sayıyı toplamak bir an sürer, milyonlarca kaydı sıralamak dakikalar sürebilir. İşlem **bellek** ister: notlar ve ara toplam bir yerde tutulmalıdır. Her adım

enerji harcar: telefonun pili de, veri merkezinin elektrik faturası da bunun sonucudur. Veri başka bir yerden geliyor ya da başka bir yere gidiyorsa **iletişim** gerekir: notlar öğrenci bilgi sisteminden ağ üzerinden okunur. Kaynaklar sonsuz değildir; bu yüzden aynı işi daha az zaman, bellek ve enerjiyle yapan yöntem daha değerlidir.

Örnek 1.5.1

70, 85, 60 ve 75 notlarının ortalaması nasıl hesaplanır? Her adımda hangi ara sonuç tutulur?

Adım	Yapılan işlem	Ara toplam
1	Ara toplam 0 ile başlar	0
2	0 + 70	70
3	70 + 85	155
4	155 + 60	215
5	215 + 75	290
6	290 ÷ 4	72,5

Sonuç: ortalama 72,5'tir. Toplama dört kez, bölme bir kez yapılmıştır. Bellekte yalnızca o ana kadarki toplam ve kaç not okunduğu tutulur. Not sayısı 40'a çıksa yine yalnız bu iki değer tutulur; değişen, toplama sayısıdır.

Örnekteki son gözlem önemlidir. Notların sayısı arttıkça yapılan toplama sayısı da artar; 40 not için 40 toplama gerekir. İşin süresinin girdinin büyüklüğüyle nasıl büyüdüğü, bir yöntemin iyi olup olmadığını belirleyen ölçütlerden biridir.

1.5.1.2 Hesaplama yalnız aritmetik değildir

Ortalama bulmak sayılarla yapılan bir iştir. Bilgisayarların günlük işlerinin çoğu ise sayılarla doğrudan ilgili görünmez. Bir e-posta programı gelen iletileri tarihe göre dizer, istenmeyen iletileri ayrı bir klasöre taşır. Bir harita uygulaması iki nokta arasındaki en kısa yolu bulur. Bir çeviri programı bir dildeki cümleyi başka bir dile aktarır. Bu işlerin hepsi verinin kurallara göre işlenmesidir; dolayısıyla hepsi hesaplamadır.

Tür	Girdi	Kural	Çıktı
Arama	Sözlük ve “bant” kelimesi	Kelimeyi alfabetik sıraya göre bul	Kelimenin anlamı
Sıralama	Sınıf listesi	Soyadlarını alfabetik sıraya diz	Sıralı liste
Karar verme	Sınav notu 47	Not 50'den küçükse “kaldı”, değilse “geçti”	“kaldı”
Dönüştürme	Fotoğraf	Her pikselin rengini griye çevir	Siyah-beyaz fotoğraf
İletişim	Bir mesaj	Mesajı paketlere böl, adresle, gönder	Alıcıya ulaşan mesaj

Tablo 1.5.1: Aritmetik dışındaki hesaplama türleri.

Tablo 1.5.1 beş farklı hesaplama türünü yan yana koyar. Hiçbirinde dört işlem ön planda değildir ama hepsinde aynı üç parça vardır. Arama işleminde girdi bir sözlük ve aranan kelimedir; kural, alfabetik sırayı kullanarak doğru sayfaya gitmektir. Karar verme işleminde kural bir karşılaştırmadır: $47 < 50$ doğru olduğu için çıktı "kaldı" olur. Dönüştürme işleminde bir fotoğrafın her pikseli aynı kurala göre değiştirilir. İletişimde bile kurallar vardır: mesaj belirli büyüklükte parçalara bölünür ve her parçanın başına adres yazılır.

Bu genişlik bilgisayarın neden bu kadar çok alanda kullanıldığını da açıklar. Bilgisayar özünde kurallara göre veri işleyen bir makinedir. Bir iş açık kurallarla anlatılabiliyorsa ve veri sayıya çevrilebiliyorsa, o iş bilgisayara yaptırılabilir. Metin, görüntü ve ses bilgisayarda zaten sayı dizileri olarak tutulur; bu yüzden onlar üzerindeki işlemler de hesaplamadır.

Örnek 1.5.2

Kampüsteki bir kantin, öğrenci kartı okutulunca öğle yemeği ücretini karttaki bakiyeden düşüyor. Ücret 45 TL, karttaki bakiye 120 TL'dir. Bu işlemde hangi hesaplama türleri vardır?

1. **İletişim.** Kart okuyucu kart numarasını kantinin sunucusuna gönderir.
2. **Arama.** Sunucu, kart numarasını kayıtlar arasında arar ve bakiyeyi bulur: 120 TL.
3. **Karar verme.** $120 \geq 45$ doğru olduğu için ödeme kabul edilir. Bakiye 45 TL'den az olsaydı ödeme reddedilirdi.
4. **Aritmetik.** Yeni bakiye hesaplanır: $120 - 45 = 75$ TL.
5. **İletişim.** Sonuç okuyucuya geri gönderilir ve ekranda "Kalan: 75 TL" yazar.

Tek bir kart okutma işleminde dört farklı hesaplama türü art arda çalışır. Aritmetik bunlardan yalnızca biridir.

Sık yapılan hata

Hesaplamayı yalnızca elektronik bilgisayarların yaptığı iş sanmak. Hesaplama bir makine değil, bir süreçtir. Kâğıt kalemle sınıf listesini sıralayan bir öğretmen de, yolları karşılaştırarak en kısasını seçen bir sürücü de hesaplama yapar. Bilgisayar, aynı süreci çok daha hızlı ve hatasız yürüten bir araçtır.

1.5.1.3 Hesaplamanın yapı taşları

Bir hesaplamanın kâğıt üzerinde tasarlanıp bir makinede çalıştırılabilmesi için birkaç farklı parçanın bir araya gelmesi gerekir. Bu parçalar altı başlık altında toplanabilir.

Matematik
: Nicelikler ve ilişkiler üzerinde işlem yapmayı sağlar. Toplama, bölme ve karşılaştırma gibi işlemler matematiğin araçlarıdır.

Mantık
: Bir koşulun doğru mu yanlış mı olduğuna karar verir. “Not 50'den küçük mü?” sorusunun cevabı mantıksal bir değerdir: doğru ya da yanlış.

Veri
: Problemi temsil eden girdiler ve hesaplama sırasında oluşan ara sonuçlardır. Ortalama örneğindeki notlar ve ara toplam veridir.

Algoritma
: İşlemlerin sonlu ve düzenli sırasıdır. “Toplamı 0 yap, her notu ekle, sonunda not sayısına böl” bir algoritmadır.

Donanım
: Adımları fiziksel olarak yürüten sistemdir: işlemci, bellek, kablolar. Algoritma ne kadar iyi olursa olsun, onu çalıştıracak bir makine olmadan sonuç elde edilmez.

Programlama dili
: Algoritmayı bilgisayarın anlayacağı biçimde yazmanın aracıdır. Aynı algoritma Java, Python ya da C ile yazılabilir.

Bu parçalar birbirinin yerini tutmaz, birbirini tamamlar. Matematik ve mantık hangi işlemlerin yapılabileceğini, veri neyin işleneceğini, algoritma işlemlerin sırasını belirler. Programlama dili algoritmayı makinenin anlayacağı biçime getirir, donanım da onu çalıştırır. Bir yemek tarifi benzetmesi bu ilişkiyi somutlaştırır: malzemeler veridir, tarif algoritmadır, tarifin yazıldığı dil programlama dilidir, ocak ve tencere donanımdır. Ölçüler ve “kıvam geldi mi?” gibi kontroller de matematik ve mantığın payıdır.

Yapı taşı	Ortalama hesabındaki karşılığı
Matematik	Toplama ve bölme: *(70 + 85 + 60 + 75) ÷ 4*
Mantık	“Okunacak not kaldı mı?” sorusu; cevap yanlış olunca toplama biter
Veri	Dört not, ara toplam ve not sayısı
Algoritma	Toplamı 0 yap; her notu toplama ekle; toplamı not sayısına böl; sonucu yaz
Donanım	Toplamı bellekte tutan ve toplamayı yapan işlemci
Programlama dili	Algoritmanın Java gibi bir dilde yazılmış hâli

Tablo 1.5.2: Ortalama hesabında yapı taşlarının karşılıkları.

Sık yapılan hata

Algoritma ile programı aynı şey sanmak. Algoritma, işin hangi adımlarla yapılacağını anlatan fikirdir; Türkçe cümlelerle ya da bir akış diyagramıyla da yazılabilir. Program ise bu fikrin belirli bir programlama dilindeki karşılığıdır. Bir algoritmanın birçok farklı program karşılığı olabilir. Algoritma yanlışsa, hangi dilde yazılırsa yazılsın program da yanlış sonuç verir.

1.5.2 Bilgisayar bilimi

Bilgisayar kullanmayı bilmek ile bilgisayar bilimini bilmek farklı şeylerdir. Bir kelime işlemcide belge hazırlayan kişi bilgisayar kullanır. Bilgisayar bilimi ise o kelime işlemcinin metni nasıl sakladığı, yazım denetiminin hangi yöntemle çalıştığı, belge büyüdükçe programın neden yavaşladığı gibi soruları sorar. Bu kesim bilgisayar biliminin temel ilgi alanlarını ve bu alanın topluma karşı sorumluluklarını tanıtır.

1.5.2.1 Bilgisayar bilimi neyi inceler?

Tanım 1.5.2

Bilgisayar bilimi (*computer science*), bilginin nasıl temsil edildiğini, nasıl işlendiğini ve problemlerin hangi yöntemlerle çözülebileceğini inceleyen bilim dalıdır. Yalnızca araç kullanmayı değil, araçların arkasındaki yöntemleri ve bu yöntemlerin sınırlarını da konu edinir.

Bilgisayar biliminin geniş alanı dört temel ilgi başlığı altında toplanabilir.

Soyutlama. Gereksiz ayrıntıları gizleyip önemli yapıya odaklanmaktır. Bir dosyayı kaydeden kullanıcı, verinin diskte hangi manyetik bölgeye ya da hangi bellek hücresine yazıldığını bilmez; yalnızca "kaydet" düğmesini görür. Diskin ayrıntıları bir soyutlama katmanının arkasına gizlenmiştir. Bilgisayar sistemleri üst üste binmiş çok sayıda böyle katmandan oluşur. Soyutlama, hesaplamalı düşünmenin de bileşenlerinden biridir.

Algoritmalar. Çözüm adımlarını tasarlamak ve bu adımları analiz etmektir. Analiz, bir algoritmanın doğru sonuç verip vermediğini ve ne kadar zaman ile bellek harcadığını sorar. Aynı problemi çözen iki algoritmadan biri bir saniyede, öteki bir saatte bitebilir.

Sistemler. Donanım ile yazılımın birlikte nasıl çalıştığını anlamaktır. İşletim sistemi, ağlar, veri tabanları ve derleyiciler bu başlığın konularıdır. Bir programın neden yavaş çalıştığı çoğu zaman algoritmaya değil, sistemin bir parçasına bağlıdır; örneğin verinin ağdan gelmesini beklemek.

İnsan ve toplum. Teknolojinin insanlar üzerindeki etkilerini, güvenliğini ve etiğini değerlendirmektir. Bir yazılım teknik olarak kusursuz çalışıp yine de insanlara zarar verebilir.

Örnek 1.5.3

Bir telefonun navigasyon uygulaması bilgisayar biliminin dört ilgi alanına nasıl dokunur?

1. **Soyutlama.** Şehir; kavşaklar ve aralarındaki yollardan oluşan bir ağ olarak tutulur. Binaların rengi, ağaçlar ve tabelalar modele girmez. Her yolun yalnız uzunluğu ve ortalama geçiş süresi saklanır.

2. **Algoritmalar.** İki nokta arasındaki en kısa süreli yolu bulan bir algoritma çalışır. Binlerce kavşaklı bir ağda bütün yolları tek tek denemek çok uzun sürer; bu yüzden aramayı akıllıca daraltan bir yöntem gerekir.
3. **Sistemler.** Konum GPS alıcısından gelir, trafik bilgisi ağ üzerinden bir sunucudan okunur, harita ekrana çizilir. Telefonun işlemcisi, belleği, ağ bağlantısı ve işletim sistemi birlikte çalışır.
4. **İnsan ve toplum.** Uygulama kullanıcının nerede olduğunu bilir. Bu konum verisinin kimlerle paylaşıldığı bir mahremiyet sorusudur. Ekranı göremeyen bir kullanıcının sesli yönlendirme alabilmesi de bir erişilebilirlik sorusudur.

1.5.2.2 Teknoloji ve sorumluluk

Bilgisayarlar artık bankacılıktan sağlığa, eğitimden işe alıma kadar insan hayatını doğrudan etkileyen kararlarda kullanılıyor. Bu yüzden bir yazılımın yalnızca doğru çalışması yetmez; adil, güvenli ve herkes için kullanılabilir olması da gerekir. Dört sorumluluk alanı öne çıkar.

Erişilebilirlik (*accessibility*)
: Farklı kullanıcıların teknolojiye eşit biçimde erişebilmesidir. Görme engelli bir kullanıcı ekran okuyucuyla bir web sayfasını dinleyebilmelidir. Bunun için sayfadaki her görselin metin açıklaması bulunmalıdır. Renk körü bir kullanıcı yalnızca kırmızı ve yeşille ayrılmış bir grafiği okuyamaz; bilgi renge ek olarak şekil ya da yazıyla da verilmelidir.

Mahremiyet (*privacy*)
: Kişisel verilerin amaçla sınırlı ve güvenli kullanılmasıdır. Bir alışveriş uygulaması teslimat için adres ister; aynı adresi kullanıcının haberi olmadan reklam şirketlerine satması amacın dışına çıkmaktır. Türkiye'de 6698 sayılı Kişisel Verilerin Korunması Kanunu bu konuda kurallar koyar.

Adalet (*fairness*)
: Veri ve algoritmaların ayrımcı sonuç üretmemesidir. Bir algoritma geçmiş verilerden öğreniyorsa, geçmişteki önyargıları da öğrenebilir. Algoritmanın kendisi tarafsız görünse bile sonuçları belirli gruplar aleyhine olabilir.

Sürdürülebilirlik (*sustainability*)
: Enerji tüketiminin, donanım ömrünün ve elektronik atığın dikkate alınmasıdır. Veri merkezleri büyük miktarda elektrik harcar. Kısa sürede eskiyen cihazlar elektronik atığa dönüşür ve bu atığın içinde doğaya zararlı metaller bulunur.

Örnek 1.5.4

Bir banka, kredi başvurularını değerlendiren bir yazılım kullanıyor. Yazılım başvuranın gelirine, borç geçmişine ve oturduğu semte bakarak bir puan veriyor. Denetim sırasında belirli semtlerden gelen başvuruların, gelir ve borç geçmişi benzer olsa bile, daha düşük puan aldığı fark ediliyor. Bu durum hangi sorumluluk alanıyla ilgilidir ve ne yapılabilir?

1. **Alanı belirleme.** Benzer durumdaki kişiler yalnızca oturdukları yer yüzünden farklı sonuç alıyor. Bu, ayrımcı bir sonuçtur; konu *adalettir*.
2. **Nedeni arama.** Semt bilgisi, gelir düzeyi ya da başka toplumsal özelliklerle ilişkili olabilir. Algoritma semti kullanarak dolaylı bir ayrım yapıyor olabilir.
3. **Önlem.** Semt bilgisi puanlamadan çıkarılabilir. Sonuçlar düzenli aralıklarla gruplara göre karşılaştırılır. Başvuru reddedilen kişiye nedenini öğrenme ve itiraz etme yolu açılır.

Aynı sistemde başvuranların kimlik ve gelir bilgileri de işlenir. Bu bilgilerin yalnızca kredi değerlendirmesi için kullanılması ve güvenli saklanması ise mahremiyet sorumluluğudur. Tek bir yazılımda birden çok sorumluluk alanı aynı anda bulunabilir.

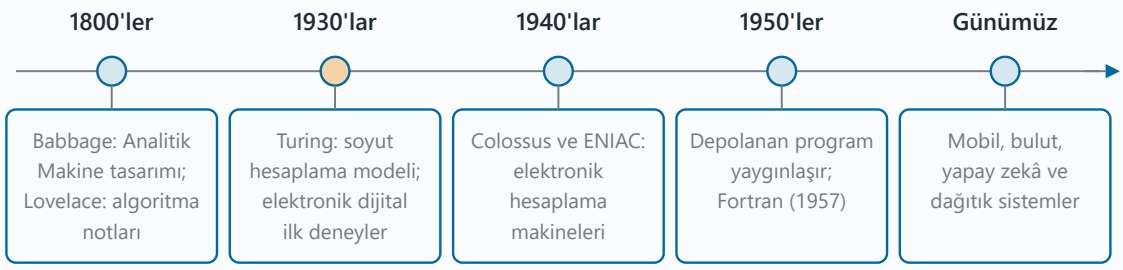
Sık yapılan hata

"Bilgisayar karar verdiyse tarafsızdır" diye düşünmek. Algoritma, kendisine verilen veriden ve kendisini tasarlayanların seçimlerinden etkilenir. Eğitildiği veri önyargılıysa sonuçları da önyargılı olur. Tarafsızlık kendiliğinden gelmez; ölçülmesi ve denetlenmesi gerekir.

1.5.3 Hesaplamanın tarihi

Bugünün bilgisayarları birkaç kuşak boyunca biriken fikirlerin ürünüdür. Bu fikirlerin çoğu, onları uygulayacak teknoloji ortaya çıkmadan çok önce düşünülmüştü. Bu kesim hesaplama tarihindeki başlıca dönüm noktalarını ve bunlar arasında belki de en önemlisini, depolanan program fikrini anlatır.

1.5.3.1 Dönüm noktaları



Şekil 1.5.2: Hesaplama tarihinin beş dönemi. 1930'lar, hesaplamanın kuramsal temelinin atıldığı dönem olarak vurgulanmıştır.

Şekil 1.5.2 beş dönemi bir zaman çizgisi üzerinde gösterir. Her dönem bir öncekinin fikirlerini alıp daha hızlı, daha küçük ve daha ucuz bir teknolojiyle uygular.

1800'ler: mekanik tasarımlar. İngiliz matematikçi Charles Babbage, 1830'larda Analitik Makine (*Analytical Engine*) adını verdiği mekanik bir hesaplama makinesi tasarladı. Makine dişlilerle çalışacaktı. Ayrı bir hesaplama birimi ve sayıları tutan bir bellek birimi olacaktı. Komutları da delikli kartlardan okuyacaktı. Delikli kart fikri, desenleri kartlardaki deliklerle belirlenen Jacquard dokuma tezgâhından alınmıştı. Babbage'ın makinesi o dönemin işçilik olanaklarıyla tamamlanamadı. Yine de tasarımı, bugünkü bilgisayarların temel parçalarını ilk kez bir arada içeriyordu. Matematikçi Ada Lovelace, 1843'te makine üzerine yazdığı notlarda Bernoulli sayılarını hesaplayan adım adım bir yöntem verdi. Bu yöntem, bir makinede çalıştırılmak üzere yazılmış ilk algoritmalardan biri sayılır. Lovelace, makinenin yalnızca sayılarla değil, sembollerle ifade edilebilen her şeyle, örneğin müzik notalarıyla da işlem yapabileceğini öngördü.

1930'lar: kuramsal temel. Alan Turing 1936'da yayımladığı makalede hesaplamanın soyut bir modelini tanımladı. Bugün Turing makinesi adıyla bilinen bu model, "bir hesaplama tam olarak nedir?" sorusuna matematiksel bir cevap verdi. Aynı makale, bazı problemlerin hiçbir makineyle çözülemeyeceğini de gösterdi. Bu yıllarda elektronik devrelerle sayısal hesap yapan ilk deneysel makineler de kurulmaya başladı.

1940'lar: elektronik makineler. İkinci Dünya Savaşı yıllarında İngiltere'de şifreli mesajları çözmek için Colossus makineleri kuruldu. ABD'de top mermilerinin yörünge tablolarını hesaplamak için ENIAC geliştirildi. Bu makineler binlerce vakum tüpüyle çalışıyor ve bir oda dolusu yer kaplıyordu. Mekanik makinelerden binlerce kat hızlıydılar. Ancak yeni bir işe geçmek için kabloların ve anahtarların elle yeniden düzenlenmesi gerekiyordu.

1950'ler: depolanan program ve ilk yüksek düzeyli diller. Komutları da bellekte saklayan bilgisayarlar yaygınlaştı. Programcılar artık makinenin kablolarıyla değil, bellekteki komutlarla uğraşıyordu. 1957'de IBM'in geliştirdiği Fortran, bilimsel hesapları matematik formüllerine benzeyen bir dille yazmayı mümkün kıldı. Fortran ilk yaygın yüksek düzeyli programlama dili oldu.

Günümüz: her yerde hesaplama. Transistör, tümleşik devre ve mikroişlemci sayesinde bilgisayarlar cebe sığacak kadar küçüldü. Bugün telefonlar, bulut hizmetleri, yapay zekâ uygulamaları ve birbirine bağlı binlerce makineden oluşan dağıtık sistemler aynı temel fikirlerle çalışır.

Bu kısa tarihten çıkan ders şudur: donanım hızlanıp küçüldükçe, algoritma ve depolanan program gibi temel fikirler daha geniş alanlarda uygulanabilir hâle geldi. Fikirler değişmedi; onları taşıyan teknoloji değişti. Babbage'ın dişlileri, ENIAC'ın vakum tüpleri ve bugünün transistörleri aynı işi yapar: kurallara göre veri işler.

Dönem	Teknoloji	Program nasıl verilir?
1800'ler	Dişliler, kollar	Delikli kartlar (tasarım)
1930'lar	Kâğıt üzerinde soyut model	Geçiş kuralları tablosu
1940'lar	Vakum tüpleri	Kablo ve anahtar düzeni
1950'ler	Vakum tüpleri, sonra transistörler	Bellekte saklanan komutlar; Fortran
Günümüz	Milyarlarca transistörlü yongalar	Yüksek düzeyli dillerde yazılmış programlar

Tablo 1.5.3: Dönemlere göre hesaplamayı taşıyan teknoloji ve programın veriliş biçimi.

Sık yapılan hata

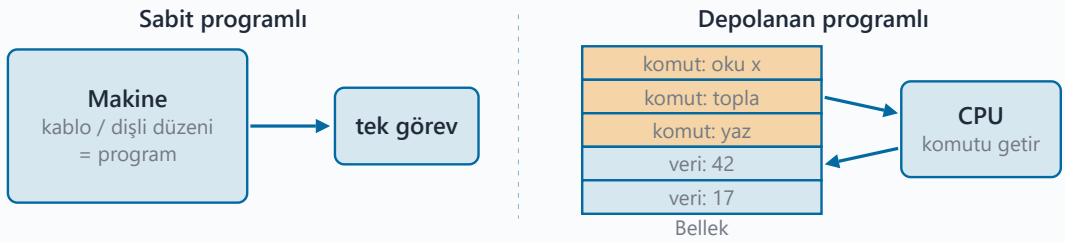
Bilgisayarı tek bir kişinin tek bir anda icat ettiğini düşünmek. Bilgisayar, yüz yılı aşkın bir sürede farklı ülkelerdeki pek çok kişinin katkısıyla gelişti. Babbage tasarımı, Lovelace algoritma fikrini, Turing kuramsal modeli, savaş yıllarının mühendisleri elektronik makineleri ortaya koydu. Her biri bir öncekinin üzerine inşa edildi.

1.5.3.2 Sabit ve depolanan program

Erken dönem makinelerin büyük çoğunluğu tek bir işi yapmak için kurulurdu. Bir masa hesap makinesi yalnızca dört işlem yapar. Eski bir çamaşır makinesinin döner düğmesi, dişlilerle belirlenmiş birkaç sabit programdan birini seçer. Bu makinelerde "program", makinenin fiziksel yapısının kendisidir. Makineye yeni bir iş yaptırmak için onu yeniden kurmak gerekir.

Tanım 1.5.3

Sabit programlı (*fixed program*) makinede yapılacak iş, makinenin fiziksel düzeniyle ya da özel bir mekanizmayla belirlenir. **Depolanan programlı** (*stored program*) makinede ise komutlar, işlenecek veriler gibi bellekte saklanır. Makine komutları bellekten tek tek okuyup uygular.



Şekil 1.5.3: Solda sabit programlı makine: kablo ya da dişli düzeni programın kendisidir ve makine tek bir görev yapar. Sağda depolanan programlı makine: komutlar (oku x, topla, yaz) ve veriler (42, 17) aynı bellekte durur; merkezî işlem birimi (CPU) komutları sırayla getirir.

Şekil 1.5.3 iki yaklaşımı yan yana koyar. Sağdaki bellekte beş hücre vardır. Üstteki üç hücrede komutlar, alttaki iki hücrede veriler durur. Bellek açısından bu hücreler arasında hiçbir fark yoktur; hepsi bit dizileridir. Bir hücrenin içeriğini komut yapan, işlemcinin onu komut olarak okumasıdır. Merkezî işlem birimi (*central processing unit*, CPU) bellekten bir komut getirir, onu çözer ve uygular, sonra sıradaki komuta geçer. Komut gereken veriyi yine aynı bellekten okur.

Bu basit fikrin sonucu büyüktür. Makineye yeni bir iş yaptırmak için kablolara dokunmak gerekmez; belleğe yeni komutlar yüklemek yeter. Aynı bilgisayar sabah bir kelime işlemci, öğleden sonra bir oyun, akşam bir video oynatıcı olur. Fiziksel olarak hiçbir şey değişmez; yalnızca bellekteki komutlar

değişir. ENIAC'ta yeni bir işe geçmek, kablolar yeniden bağlanması gerektiği için günler sürebiliyordu. Depolanan programlı bir makinede aynı geçiş, yeni bir programı belleğe yükleme süresi kadardır.

Depolanan program fikri, 1945'te John von Neumann'ın EDVAC adlı makine için yazdığı raporla geniş bir çevreye yayıldı. Komutların ve verilerin aynı bellekte durduğu bu yapıya bugün **von Neumann mimarisi** (*von Neumann architecture*) denir. Bellekte saklanan bir programı çalıştıran ilk elektronik bilgisayar, 1948'de İngiltere'de Manchester Üniversitesinde çalıştırılan küçük deneysel bir makinedir. Bugünkü bilgisayarların hemen hepsi bu mimarinin bir türevidir.

Yaklaşım	Program nasıl belirlenir?	Esneklik
Sabit programlı	Makinenin fiziksel düzeni veya özel mekanizması	Belirli görev
Depolanan programlı	Komutlar bellekte veri gibi saklanır	Program değiştirilerek farklı görevler
Genel amaçlı	Çok farklı programları çalıştırabilen mimari	Geniş uygulama alanı

Tablo 1.5.4: Sabit programlı, depolanan programlı ve genel amaçlı makinelerin karşılaştırılması.

Tablo 1.5.4 üçüncü bir satır da içerir: **genel amaçlı** (*general purpose*) bilgisayar. Depolanan program fikri genel amaçlı bilgisayarın ön koşuludur. Komutları değiştirilebilen bir makine, uygun komutlarla hemen her hesaplamayı yapabilir. Kişisel bilgisayarlar ve akıllı telefonlar genel amaçlıdır. Bunun karşıtı, tek bir iş için tasarlanmış özel amaçlı cihazlardır.

Örnek 1.5.5

Aşağıdaki üç cihaz sabit programlı mı, yoksa depolanan programlı mı çalışır? Mekanik bir kol saati, bir oyun konsolu ve bir dijital fotoğraf makinesi.

1. **Mekanik kol saati.** Saatin yaptığı iş dişlilerin oranlarıyla belirlenmiştir. Saate yeni bir iş yaptırmak için dişlileri değiştirmek gerekir. Sabit programlıdır.
2. **Oyun konsolu.** Her yeni oyun, belleğe yüklenen yeni bir programdır. Aynı donanım binlerce farklı oyunu çalıştırır; güncellemelerle yeni özellikler de kazanır. Depolanan programlıdır.
3. **Dijital fotoğraf makinesi.** Kullanıcıya tek bir iş yapıyormuş gibi görünür: fotoğraf çekmek. İçinde ise bellekte saklanan bir yazılım çalışan bir işlemci vardır. Üretici yazılım güncellemesiyle makineye yeni çekim modları ekleyebilir. İçeriden bakıldığında depolanan programlıdır; ancak kullanıcı kendi programını yükleyemediği için genel amaçlı değildir.

Üçüncü cihaz, iki ayrımın birbirinden bağımsız olduğunu gösterir. Bir cihaz depolanan programla çalışıp yine de tek bir iş için tasarlanmış olabilir. Bugünkü çamaşır makinelerinin, fırınların ve arabaların içindeki küçük bilgisayarlar da böyledir.

Sık yapılan hata

Depolanan programlı makineyi "daha hızlı" ya da "daha büyük belleği olan" makine sanmak. Ayırt edici özellik hız değil, komutların nerede durduğudur. Komutlar bellekte veri gibi saklanıyor ve değiştirilebiliyorsa makine depolanan programlıdır. Çok hızlı bir makine de sabit programlı olabilir; yavaş bir makine de depolanan programlı olabilir.

1.5.4 Hesaplama modelleri

Bir bilgisayarın neleri hesaplayabileceğini, neleri hesaplayamayacağını sormak için önce "hesaplama" kavramını kesin biçimde tanımlamak gerekir. Gerçek bilgisayarlar bu iş için fazla karmaşıktır: işlemci türleri, bellek boyutları ve programlama dilleri birbirinden çok farklıdır. Bu yüzden bilgisayar bilimi, ayrıntılardan arındırılmış basit makine modelleri kullanır. Bu modellerin en ünlüsü Turing makinesidir. Bu kesim Turing makinesinin parçalarını tanıtır, bir makineyi adım adım çalıştırır ve bu modelin bilgisayarların sınırları hakkında ne söylediğini anlatır.

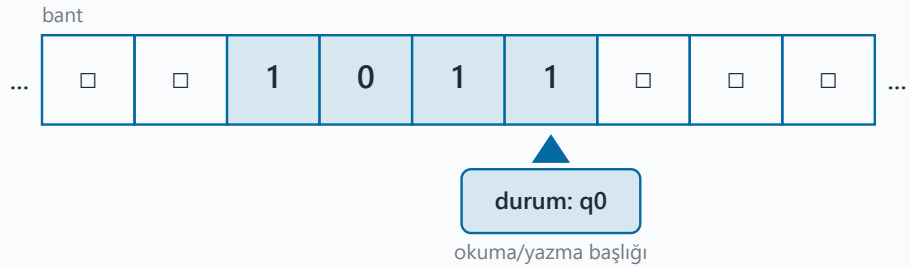
1.5.4.1 Turing makinesi

Turing, modelini bir insanın kâğıt kalemle hesap yapışını gözleyerek kurdu. Hesap yapan kişi kâğıda simgeler yazar, yazdıklarını okur, bir sonraki adımda ne yapacağına o an aklında tuttuğu bilgiye ve gördüğü simgeye bakarak karar verir. Turing bu davranışı en yalın parçalarına ayırdı.

Tanım 1.5.4

Turing makinesi (*Turing machine*), algoritmik hesaplamayı açıklamak için kullanılan soyut bir makine modelidir. Dört parçadan oluşur:

- **Bant** (*tape*): hücrelere bölünmüş, iki yönde istenildiği kadar uzayabilen bir şerittir. Her hücrede bir **simge** (*symbol*) bulunur. Boş hücre □ ile gösterilir.
- **Okuma/yazma başlığı** (*read/write head*): her an tek bir hücrenin üzerindedir. O hücredeki simgeyi okur, yerine yeni bir simge yazabilir, sonra bir hücre sağa ya da sola kayar veya olduğu yerde kalır.
- **Durum** (*state*): makinenin o an "aklında tuttuğu" bilgidir. Durumlar q0, q1 gibi adlarla gösterilir. Makine özel bir durum olan DUR'a gelince çalışmayı bitirir.
- **Geçiş kuralları** (*transition rules*): her (durum, okunan simge) çifti için ne yapılacağını söyleyen tablodur. Bir kural üç şeyi belirler: yazılacak simge, başlığın hareketi ve yeni durum.



Şekil 1.5.4: Bir Turing makinesi. Bantta □□1011□□□ yazılıdır; iki yandaki üç nokta bandın iki yönde sürdüğünü gösterir. Okuma/yazma başlığı en sağdaki 1'in (6. hücre) altındadır ve makine q0 durumundadır.

Şekil 1.5.4 makinenin bir anlık görüntüsüdür. Bantta ikili 1011 sayısı yazılıdır; geri kalan hücreler boştur. Başlığın altındaki üçgen, okunacak hücreyi gösterir. Başlığın altındaki kutuda makinenin durumu yazar: q0.

Modelin gücü basitliğinden gelir. Makinenin bir adımı her zaman aynı biçimdedir:

1. Başlığın altındaki simge okunur.
2. Geçiş kuralları tablosunda (şu anki durum, okunan simge) çiftine uyan satır bulunur.
3. O satırın söylediği simge hücreye yazılır.
4. Başlık satırın söylediği yöne bir hücre kayar ya da yerinde kalır.
5. Makine satırın söylediği yeni duruma geçer. Yeni durum DUR ise makine durur; değilse ilk adıma dönülür.

Makinenin "zekâsı" yalnızca kural tablosundadır. Makine tabloya bakar ve söyleneni yapar; geçmiş adımları hatırlamaz, ileriyi planlamaz. Hatırlaması gereken bir şey varsa bunu ya bantta bir simge olarak ya da durumunda taşır. Sonsuz uzunluktaki bant, makinenin belleğidir. Gerçek bilgisayarların belleği sonludur; model ise bellek sıkıntısını bir kenara bırakıp yalnızca "bu iş kurallarla yapılabilir mi?" sorusuna odaklanır.

Sık yapılan hata

Turing makinesini gerçekten üretilmiş bir cihaz sanmak. Turing makinesi kâğıt üzerinde tanımlanmış bir modeldir; amacı bir bilgisayar yapmak değil, hesaplamanın ne olduğunu kesin biçimde tanımlamaktır. Meraklılar sonradan küçük gösteri makineleri kurmuştur ama modelin değeri kuramsaldır.

1.5.4.2 Geçiş kuralları

Bir Turing makinesinin ne yapacağını yalnızca kural tablosu belirler. İlk örnek, bantta yazılı ikili bir sayıya 1 ekleyen makinedir. İkili sistemde 1 eklemek onluk sistemdekine benzer. En sağdaki basamaktan başlanır. Basamak 0 ise 1 yapılır ve iş biter. Basamak 1 ise 0 yapılır ve elde bir soldaki basamağa aktarılır; aynı işlem orada tekrarlanır. Sayının solunda basamak kalmamışsa, oraya 1 yazılır.

Kural	Durum	Okunan	Yaz	Git	Yeni durum	Anlamı
1	q0	1	0	sola	q0	1 + 1 = 10: basamak 0 olur, elde sola taşınır
2	q0	0	1	dur	DUR	0 + 1 = 1: elde bu basamakta biter
3	q0	□	1	dur	DUR	Sayının solu boş: elde yeni basamak olur

Tablo 1.5.5: İkili sayıya 1 ekleyen Turing makinesinin geçiş kuralları. Başlangıç durumu q0'dır; başlık sayının en sağdaki basamağında başlar.

Tablo 1.5.5 üç kuraldan oluşur. Makinenin tek bir çalışma durumu vardır, q0. Bu durumda okunabilecek üç simge vardır: 1, 0 ve □. Her biri için tabloda tam olarak bir kural bulunur. Bu yüzden makine hiçbir adımda ne yapacağı konusunda kararsız kalmaz. Kural 1, elde taşıma işini yapar ve makineyi yine q0'da bırakır; böylece aynı kural, art arda gelen bütün 1'lere uygulanabilir. Kural 2 ve 3 makineyi durdurur.

Bu tablo, ikili toplama bilgisini yalnızca üç satıra sığdırır. Makine "ikili sayı" ya da "elde" kavramlarını bilmez; yalnızca simgeleri okur ve yazar. Kuralların anlamını veren, tabloyu tasarlayan insandır.

1.5.4.3 Adım adım: 111 + 1

Makineyi bantta 111 yazılıyken çalıştıralım. 111_2 onluk sistemde *4 + 2 + 1 = 7*'dir; beklenen sonuç 8, yani 1000_2'dir. Başlık en sağdaki basamağın üzerindedir ve durum q0'dır.

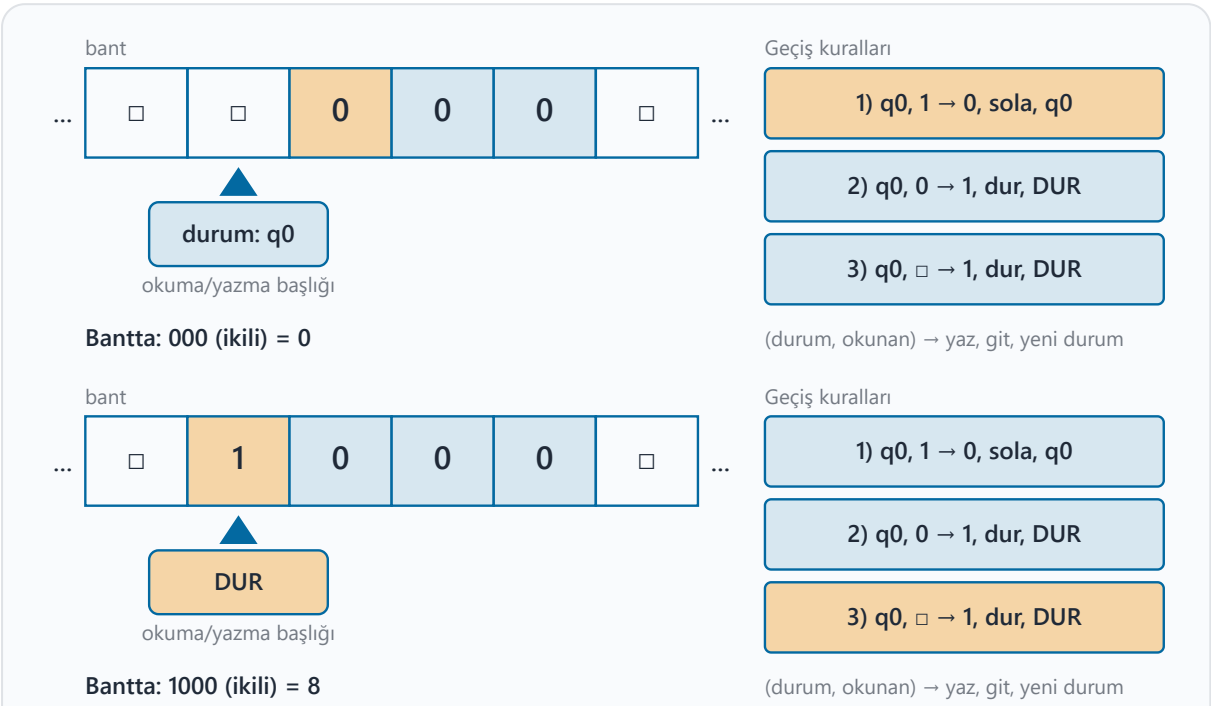


Şekil 1.5.5: Başlangıç. Bantta 111 (ikili 7) yazılıdır; başlık 5. hücrede, en sağdaki 1'in altındadır; durum q0'dır. Sağda geçiş kuralları görülür.

Şekil 1.5.5'teki başlangıç durumundan sonra makine şu adımları izler:

1. **Adım 1.** Okunan 1, durum q0. Uyan kural 1'dir: 0 yazılır, başlık sola kayar, durum q0 kalır. Bant □□110□ olur. En sağdaki basamakta 1 + 1 = 10 hesaplandı: basamağa 0 yazıldı, elde 1 bir sola taşındı.
2. **Adım 2.** Başlık 4. hücrede, okunan yine 1. Kural 1 yeniden uygulanır. Bant □□100□ olur; elde hâlâ sürüyor.
3. **Adım 3.** Başlık 3. hücrede, okunan 1. Kural 1 üçüncü kez uygulanır. Bant □□000□ olur. Sayının bütün basamakları 0'a döndü ama elde henüz yazılmadı.

4. **Adım 4.** Başlık 2. hücrede, okunan □. Uyan kural 3'tür: 1 yazılır ve makine DUR durumuna geçer. Bant □1000□ olur.



Şekil 1.5.6: Son iki adım. (a) Üç kez kural 1 uygulandıktan sonra bant □□000□'dır; başlık 2. hücredeki boş hücreyi okumak üzeredir. (b) Kural 3 ile 1 yazılmış ve makine durmuştur: bant □1000□, yani ikili 8.

Şekil 1.5.6 son iki adımın görüntüsünü verir. Bütün çalışma aşağıdaki izleme tablosunda özetlenmiştir. Tablonun her satırı, bir kural uygulanmadan önceki durumu ve uygulandıktan sonraki bandı gösterir.

Adım	Başlık	Durum	Okunan	Kural	Sonraki bant	Sonraki başlık	Sonraki durum
1	5	q0	1	1	□□110□	4	q0
2	4	q0	1	1	□□100□	3	q0
3	3	q0	1	1	□□000□	2	q0
4	2	q0	□	3	□1000□	2	DUR

Tablo 1.5.6: 111 + 1 için izleme tablosu. Hücreler soldan 1'den başlayarak numaralanmıştır.

Makine dört adımda durdu ve bantta 1000 kaldı: $1000_2 = 8 = 7 + 1$. Sonuç doğrudur. Tablo 1.5.6 önemli bir gözlemi de içerir: her satırda yapılacak işi yalnızca “Durum” ve “Okunan” sütunları belirledi. Makinenin önceki adımları, bandın geri kalanı ya da sayının kaç basamaklı olduğu hiçbir kararı etkilemedi.

Adım sayısı da girdiye bağlıdır. Makine sağdan itibaren art arda gelen her 1 için bir adım, elde nerede biterse orada bir adım daha harcar. Sağ ucunda art arda k tane 1 bulunan bir sayı için makine $k + 1$ adımda durur. 111'in sağ ucunda art arda üç tane 1 vardır; adım sayısı $3 + 1 = 4$'tür.

Örnek 1.5.6

Aynı makine bantta 10011 (ikili 19) ile çalıştırılıyor; başlık en sağdaki basamakta. Makine kaç adımda durur ve bantta ne kalır?

Adım	Okunan	Kural	Yapılan	Sonraki bant
1	1	1	0 yaz, sola	10010
2	1	1	0 yaz, sola	10000
3	0	2	1 yaz, DUR	10100

Makine üç adımda durur. Bantta 10100 kalır: *16 + 4 = 20 = 19 + 1*. Sağda iki tane 1 olduğu için adım sayısı *2 + 1 = 3*'tür. Eldenin bittiği 0'ın solundaki basamaklara (en soldaki 1 ve 0) hiç dokunulmadı. Kural 3 bu girdide hiç kullanılmadı; çünkü başlık boş bir hücreye ulaşmadan elde bitti.

Sık yapılan hata

Kural 1'i uyguladıktan sonra makinenin durduğunu sanmak. Kural 1'in yeni durumu DUR değil, q0'dır; makine çalışmaya devam eder. Makine yalnızca yeni durumu DUR olan bir kural uygulandığında durur. Bu makinede o kurallar 2 ve 3'tür.

1.5.4.4 Durumun belleği: eşlik denetimi

1 ekleyen makinenin tek bir çalışma durumu vardı. Bazı işlerde makinenin, okuduğu simgelerle ilgili bir bilgiyi aklında tutması gerekir. Bu bilgi durumda saklanır. Bir bit dizisindeki 1'lerin sayısının çift mi tek mi olduğunu bulan makine bunu açıkça gösterir. Simülatördeki hazır makinelerden biri olan bu makinenin iki çalışma durumu vardır: “çift” ve “tek”. Başlık dizinin en solundaki simgede başlar ve sağa doğru ilerler.

Kural	Durum	Okunan	Yaz	Git	Yeni durum	Anlamı
1	çift	0	0	sağa	çift	0, 1'lerin sayısını değiştirmez
2	çift	1	1	sağa	tek	Bir 1 daha: sayı tek oldu
3	tek	0	0	sağa	tek	0, 1'lerin sayısını değiştirmez
4	tek	1	1	sağa	çift	Bir 1 daha: sayı çift oldu
5	çift	□	Ç	dur	DUR	Dizinin sonu; 1'lerin sayısı çift
6	tek	□	T	dur	DUR	Dizinin sonu; 1'lerin sayısı tek

Tablo 1.5.7: Eşlik denetimi yapan Turing makinesinin geçiş kuralları. Başlangıç durumu “çift”tir.

Tablo 1.5.7'deki makine bantta hiçbir şeyi değiştirmeden ilerler; her hücreye okuduğu simgeyi geri yazar. Değişen tek şey durumdur. Makine bir 1 okuduğunda durum çiftten teke ya da tekten çifte döner, 0 okuduğunda durum aynı kalır. Dizinin sonundaki boş hücreye gelindiğinde durum, o ana kadar okunan 1'lerin sayısının çift mi tek mi olduğunu söyler. Makine bu cevabı boş hücreye Ç ya da T olarak yazar ve durur.

Örnek 1.5.7

Eşlik denetimi makinesi bantta 1101 ile çalıştırılıyor. Her adımda durum nasıl değişir, bantta ne yazılır?

Adım	Başlık	Durum	Okunan	Kural	Yeni durum
1	1	çift	1	2	tek
2	2	tek	1	4	çift
3	3	çift	0	1	çift
4	4	çift	1	2	tek
5	5	tek	□	6	DUR (T yazıldı)

Makine beş adımda durur ve bant 1101T olur. Dizide üç tane 1 vardır; üç tek sayıdır. Makine sayma işlemini hiç yapmadan, yalnızca iki durum arasında gidip gelerek doğru cevaba ulaştı. Adım sayısı dizinin uzunluğundan bir fazladır: dört simge ve bir boş hücre.

Bu makine, Turing makinesindeki durumun bir tür küçük bellek olduğunu gösterir. Makine 1'lerin tam sayısını bilmez; yalnızca işin gerektirdiği kadarını, yani sayının çift mi tek mi olduğunu bilir. Aynı fikir gerçek bilgisayarlarda da kullanılır: verinin iletim sırasında bozulup bozulmadığını anlamak için gönderilen eşlik biti, tam olarak bu hesapla bulunur.

1.5.4.5 Evrensel Turing makinesi

Şimdiye kadar görülen her Turing makinesi tek bir iş yapıyordu: biri sayıya 1 ekliyordu, öteki eşlik denetliyordu. Yeni bir iş için yeni bir kural tablosu, yani yeni bir makine gerekiyordu. Bu açıdan her Turing makinesi sabit programlı bir makineye benzer. Turing, aynı makalede bu sınırı aşan bir makine de tanımladı.

Tanım 1.5.5

Evrensel Turing makinesi (*universal Turing machine*), çalışmaya başladığında bandında iki şey yazılı olan bir Turing makinesidir: başka bir Turing makinesinin kural tablosu ve o makinenin girdisi. Evrensel makine bu tabloyu okuyup adım adım uygulayarak o makinenin yapacağı işin aynısını yapar; yani onu taklit eder.

Evrensel makinenin kendi kural tablosu sabittir ve hiç değişmez. Değişen, bandına yazılan tablodur. Bandına 1 ekleyen makinenin tablosu yazılırsa 1 ekler; eşlik makinesinin tablosu yazılırsa eşlik denetler. Kural tablosu artık makinenin parçası değil, bantta duran bir veridir.

Bu fikir, Tanım 1.5.3'te tanımlanan depolanan program fikrinin kuramsal temelidir. Evrensel Turing makinesinin bandı bilgisayarın belleğine, banttaki kural tablosu bellekteki programa, evrensel makinenin sabit kuralları da işlemcinin komutları çözüp uygulayan devrelerine karşılık gelir. Turing bu fikri 1936'da kâğıt üzerinde ortaya koydu. Gerçek depolanan programlı bilgisayarlar ise on yıldan fazla bir süre sonra kurulabildi.

Evrensel Turing makinesi	Depolanan programlı bilgisayar
Bant	Bellek
Bantta yazılı kural tablosu	Bellekteki program
Bantta yazılı girdi	Bellekteki veri
Evrensel makinenin sabit kuralları	İşlemcinin komutları getirip uygulayan devreleri

Tablo 1.5.8: Evrensel Turing makinesi ile depolanan programlı bilgisayar arasındaki karşılıklar.

Turing makinesi basit görünse de çok güçlüdür. Bugün bilinen her bilgisayarın, yeterli zaman ve bellek verildiğinde yapabildiği her hesabı bir Turing makinesi de yapabilir. Başka bir deyişle, bir Turing makinesinin yapamadığı bir hesabı bugünkü hiçbir bilgisayar yapamaz. Bilgisayarlar Turing makinesinden çok daha hızlıdır ama hesaplayabildikleri şeylerin kümesi aynıdır. Bu yüzden Turing makinesi, "bilgisayar neyi hesaplayabilir?" sorusunu incelemek için yeterli bir modeldir.

Sık yapılan hata

Evrensel Turing makinesini, sonsuz bandı ya da çift yönlü başlığı olan makine sanmak. Sonsuz bant ve iki yöne gidebilen başlık her Turing makinesinde vardır. Evrensel makineyi ayıran özellik, başka bir makinenin tanımını girdi olarak alıp onu taklit edebilmesidir.

1.5.4.6 Bilgisayarların sınırları

Bilgisayarlar çok hızlıdır ama her şeyi yapamaz. Sınırların bir kısmı kaynaklardan, bir kısmı verinin ve modelin doğasından, bir kısmı da hesaplamanın kendisinden gelir. Altı sınır öne çıkar.

Zaman
: Büyük girdiler bazı algoritmaları pratik olmaktan çıkarabilir. Bir algoritmanın süresi girdi büyüdükçe çok hızlı artıyorsa, sonuç kullanıcının bekleyebileceği süre içinde gelmez.

Bellek
: Veri ve ara sonuçlar sonlu bir depolama alanında tutulur. Turing makinesinin bandı sonsuzdur; gerçek bilgisayarın belleği değildir.

Enerji

Her işlem ve her veri aktarımı enerji tüketir. Telefonun pili, bir veri merkezinin soğutma sistemi ve elektrik faturası bu sınırın günlük yüzleridir.

Hesaplanabilirlik (*computability*)

Bazı problemler için her girdide doğru sonuç veren bir algoritma yoktur. En bilinen örnek durma problemidir.

Veri kalitesi

Eksik ya da hatalı girdi, doğru bir algoritmayı bile yanlış sonuca götürür.

Model sınırı

Program gerçek dünyanın yalnızca seçilmiş özelliklerini temsil eder. Modelde olmayan bir özellik, sonuçta da hesaba katılmaz.

İlk üç sınır kaynak sınırıdır. Daha hızlı bir işlemci, daha büyük bir bellek ya da daha verimli bir algoritma bu sınırları geriye iter. Hesaplanabilirlik ise başka türden bir sınırdır: donanım ne kadar güçlenirse güçlensin aşılamaz. Son iki sınır hesaplamanın gerçek dünyayla bağından kaynaklanır.

Örnek 1.5.8

Bir program, bir şifreyi bütün olası anahtarları tek tek deneyerek çözmeye çalışıyor ve saniyede 10^9 (bir milyar) anahtar deneyebiliyor. Anahtar 40 bit uzunluğundaysa en kötü durumda ne kadar sürer? Anahtar 64 bit olursa ne olur?

1. **Olası anahtar sayısı.** n bitle 2^n farklı anahtar yazılabilir. 40 bit için $2^{40} = 1.099.511.627.776$, yaklaşık $1{,}1 \times 10^{12}$ anahtar vardır.
2. **40 bit için süre.** $2^{40} \div 10^9 \approx 1.099{,}5$ saniye, yani yaklaşık 18 dakika.
3. **64 bit için süre.** $2^{64} \approx 1{,}84 \times 10^{19}$ anahtar vardır. $1{,}84 \times 10^{19} \div 10^9 \approx 1{,}84 \times 10^{10}$ saniye eder. Bir yıl yaklaşık $3{,}16 \times 10^7$ saniye olduğundan süre yaklaşık 584 yıldır.
4. **Donanımın etkisi.** İki kat hızlı bir bilgisayar süreyi yarıya, yaklaşık 292 yıla indirir. Anahtara yalnızca bir bit eklemek ise süreyi yeniden iki katına çıkarır.

Anahtar uzunluğu 40 bitten 64 bite çıkınca, yani yalnızca 24 bit artınca, süre $2^{24} \approx 16{,}8$ milyon kat uzadı. Süresi bu biçimde, girdinin her bitiyle iki katına çıkan algoritmalar büyük girdilerde pratik olmaktan çıkar. Şifreleme sistemleri tam da bu zaman sınırına güvenir.

Durma problemi. Hesaplanabilirlik sınırının en bilinen örneği **durma problemidir** (*halting problem*). Soru şudur: herhangi bir program ve onun girdisi verildiğinde, programın bir gün durup durmayacağına her durumda doğru karar veren genel bir algoritma yazılabilir mi? Bazı programlar için cevap kolaydır. 1 ekleyen Turing makinesinin her girdide durduğu açıktır. Bazı programlar için ise cevabı bulmak çok zordur. Aşağıdaki kısa program bunun ünlü bir örneğidir:

```
n = 6
while n ≠ 1
    if n mod 2 = 0
        n = n / 2
    else
        n = 3 * n + 1
yaz("durdu")
```

Program n = 6 ile çalıştırılırsa n sırasıyla 6, 3, 10, 5, 16, 8, 4, 2, 1 değerlerini alır ve program sekiz turda durur. n = 27 ile çalıştırılırsa 111 tur sürer ve ara değerler 9.232'ye kadar çıkar. Bu programın her pozitif n için durup durmadığı bugün hâlâ bilinmiyor. Bilgisayarlarla çok büyük sayılara kadar denenmiş ve hep durduğu görülmüştür; ama her n için durduğu kanıtlanamamıştır.

Turing 1936'da daha güçlü bir şey gösterdi: her program ve her girdi için durup durmayacağına doğru karar veren genel bir algoritma *hiçbir zaman* yazılamaz. Gerekçenin ana fikri şöyledir. Böyle bir "kahin" algoritmanın var olduğunu varsayalım. Bu kahini kullanarak ters davranan bir program yazılabilir: program kendisi hakkında kahine sorar; kahin "durur" derse sonsuza kadar döner, "durmaz" derse hemen durur. Bu ters program kendisi üzerinde çalıştırıldığında kahin ne cevap verirse versin yanılmış olur. Demek ki böyle bir kahin var olamaz. Bu sonuç hız ya da bellekle ilgili değildir; mantıksal bir imkânsızlıktır.

Örnek 1.5.9

Bir hava istasyonu bir saat içinde dört sıcaklık ölçümü gönderiyor: 21,4 °C, 22,1 °C, 23 °C ve −99 °C. Sensör, ölçüm alamadığı anlarda −99 değerini gönderiyor. Ortalama hesaplayan program bu dört değerin hepsini kullanırsa sonuç ne olur?

1. **Toplam.** *21,4 + 22,1 + 23 + (−99) = −32,5*.
2. **Ortalama.** *−32,5 ÷ 4 = −8,125* °C.
3. **Doğru hesap.** −99 bir ölçüm değil, "ölçüm yok" anlamına gelen bir işarettir; hesaptan çıkarılmalıdır. Kalan üç değerin ortalaması *(21,4 + 22,1 + 23) ÷ 3 = 66,5 ÷ 3 ≈ 22,17* °C'dir.

Ortalama algoritması kusursuz çalıştı; hata verideydi. Bu duruma "çöp girerse çöp çıkar" denir. Veri kalitesi sınırı ancak verinin önceden denetlenmesiyle aşılır.

Model sınırı. Bir navigasyon uygulaması yolları uzunlukları ve ortalama hızlarıyla modeller. Bir yolun o sabah su baskını yüzünden kapandığı modelde yoksa, uygulama kapalı yolu önermeye devam eder. Program yanlış hesap yapmamıştır; gerçek dünyanın modelde bulunmayan bir özelliğini bilememiştir. Her program bir modelle çalışır ve modelin dışında kalan her şey programın da dışında kalır.

Sık yapılan hata

Her sınırın daha güçlü donanımla aşılabileceğini düşünmek. Zaman, bellek ve enerji sınırları daha iyi donanım ya da daha iyi algoritmayla hafifletilebilir. Durma problemi gibi hesaplanabilirlik sınırları ise kuramsaldır: böyle bir algoritma hiç yoktur, bu yüzden hiçbir donanım onu çalıştıramaz.

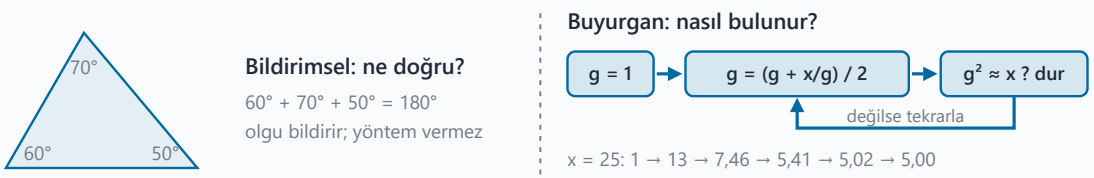
1.5.5 Bilgi ve algoritma

Bir şeyi bilmenin iki farklı yolu vardır. İnsan bir olguyu bilebilir: "Ankara Türkiye'nin başkentidir." Bir işin nasıl yapıldığını da bilebilir: "Ankara'ya gitmek için şu otobüse binilir, şu durakta inilir." Bilgisayar biliminin asıl ilgilendiği ikinci türdür, çünkü bilgisayara bir işi yaptırmak için o işin nasıl yapıldığını adım adım anlatmak gerekir. Bu kesim iki bilgi türünü karşılaştırır, algoritmayı tanımlar ve iki klasik algoritmayı, karekök bulmayı ve Öklid algoritmasını, ayrıntılı izler.

1.5.5.1 Bildirimsel ve buyurgan bilgi

Tanım 1.5.6

Bildirimsel bilgi (*declarative knowledge*), neyin doğru olduğunu ya da bir kavramın ne olduğunu açıklar. **Buyurgan bilgi** (*imperative knowledge*), emirsel bilgi diye de anılır; bir sonuca nasıl ulaşılacağını adımlarla açıklar.



Şekil 1.5.7: İki bilgi türü. Solda bildirimsel bilgi: bir üçgenin iç açıları 60°, 70° ve 50°'dir ve toplamları 180°'dir; bu bir olgudur, yöntem vermez. Sağda buyurgan bilgi: karekökü bulmak için bir tahminle başlanır, tahmin bir formülle güncellenir ve tahminin karesi x'e yeterince yaklaşınca durulur. x = 25 için tahminler 1, 13, 7,46, 5,41, 5,02 ve 5,00 olur.

Şekil 1.5.7 iki türü yan yana koyar. "Bir üçgenin iç açıları toplamı 180°'dir" cümlesi doğru bir bilgi verir ama bir işi nasıl yapacağımızı söylemez. Bu bilgiyle bir üçgenin eksik açısı bulunabilir; ancak bunun için yine bir yöntem gerekir. Sağdaki adımlar ise doğrudan uygulanabilir. Adımları izleyen kişi ya da makine, karekökün ne olduğunu tam olarak anlamasa bile sonuca ulaşır.

Karekökün tanımı da bildirimsel bir bilgidir: “x'in karekökü, karesi x olan pozitif sayıdır.” Bu tanım bir sayının karekök olup olmadığını denetlemeye yarar. 5'in karesi 25 olduğu için 5, 25'in kareköküdür. Ama tanım, 10'un karekökünün kaç olduğunu bulmanın yolunu göstermez. Bildirimsel bilgi “ne” sorusuna, buyurgan bilgi “nasıl” sorusuna cevap verir.

Konu	Bildirimsel bilgi (ne?)	Buyurgan bilgi (nasıl?)
Karekök	√x, karesi x olan pozitif sayıdır.	Bir tahminle başla, tahmini ortalama formülüyle güncelle, yeterince yaklaşınca dur.
Asal sayı	Asal sayı yalnızca 1'e ve kendisine bölünen, 1'den büyük tam sayıdır.	Sayıyı 2'den başlayarak karekökünü aşmayan her sayıya böl; hiçbiri tam bölmezse sayı asaldır.
Sıralı liste	Her öğe kendinden sonrakinden büyük değildir.	En küçük öğeyi bul, başa koy; kalan öğelerle aynı işi tekrarla.
Yol tarifi	Kütüphane, merkez binanın kuzeyindedir.	Ana kapıdan gir, sağa dön, ikinci binaya kadar yürü.

Tablo 1.5.9: Bildirimsel ve buyurgan bilgi örnekleri.

Tablo 1.5.9 her satırda aynı konunun iki yüzünü gösterir. Sıralı liste satırı özellikle öğreticidir: bir listenin sıralı olup olmadığını denetlemek kolaydır, ama karışık bir listeyi sıralamak bir yöntem gerektirir. Bilgisayar programları buyurgan bilginin yazılı hâlidir. Bildirimsel bilgi ise programın doğru çalışıp çalışmadığını denetlemekte kullanılır: sıralama programının çıktısının gerçekten sıralı olduğu, bildirimsel tanımla sınanır.

Sık yapılan hata

Bir formül ya da tanım içeren her cümleyi buyurgan bilgi sanmak. “√49 = 7'dir, çünkü 7 · 7 = 49” cümlesi bir sonucu ve gerekçesini bildirir; bildirimseldir. Cümlede adımlar, tekrar ya da durma koşulu yoktur. Bir bilgi ancak bir sonuca ulaşmak için izlenecek adımları söylüyorsa buyurgandır.

1.5.5.2 Algoritma

Buyurgan bilginin her türü bir makineye verilemez. “Güzel bir yemek yap” bir talimattır ama hangi adımların izleneceğini söylemez. “Tuzu kararınca ekle” bir adımdır ama ne kadar tuz ekleneceği belli değildir. Bir makinenin izleyebileceği talimatların açık, uygulanabilir ve sonlu olması gerekir.

Tanım 1.5.7

Algoritma (*algorithm*), bir problemi sonlu sayıda açık ve uygulanabilir adımla çözen yöntemdir. Bir algoritmanın şu özellikleri vardır:

- **Açıklık:** her adım tek bir anlama gelir; adımı uygulayan iki kişi aynı şeyi yapar.
- **Uygulanabilirlik:** her adım, elde bulunan araçlarla gerçekten yapılabilir.
- **Sonluluk:** algoritma her girdi için sonlu sayıda adımdan sonra durur.
- **Girdi ve çıktı:** algoritma sıfır ya da daha fazla girdi alır ve en az bir çıktı üretir.

Algoritmalar buyurgan bilgiyi açık, uygulanabilir ve sonlu adımlara dönüştürür. Algoritmayı yazmanın tek bir yolu yoktur. Türkçe cümleler, akış diyagramları ya da bir programlama diline benzeyen ama onun kadar katı olmayan **sözde kod** (*pseudocode*) kullanılabilir. Bu notta algoritmalar sunumdaki gibi kısa sözde kodlarla verilir. Sözde kodda `while` "koşul doğru olduğu sürece altındaki satırları tekrarla", `return` "sonucu döndür ve bitir", `yaz` "sonucu ekrana yaz" anlamına gelir. İçeri girintili satırlar, üstlerindeki `while` satırına bağlıdır.

Sık yapılan hata

Durma koşulunu unutmak. "Sayıya 1 ekle ve tekrarla" talimatı açıktır ve uygulanabilir ama hiç bitmez. Böyle bir talimat dizisi bir algoritma değildir; programda **sonsuz döngü** (*infinite loop*) olarak ortaya çıkar. Her tekrarlı adımın, bir gün mutlaka sağlanacak bir durma koşulu olmalıdır.

1.5.5.3 Karekökü yaklaşık bulma

Karekök bulmanın eski ve çok etkili bir yöntemi, İskenderiyeli Heron'a atfedilir ve Babil yöntemi adıyla da anılır. Yöntemin fikri şudur: bir tahmin karekökten büyükse, x'in tahmine bölümü karekökten küçük olur; tahmin karekökten küçükse bölüm büyük olur. Gerçek karekök bu iki sayının arasındadır. Bu yüzden ikisinin ortalaması, çoğu zaman tahminin kendisinden daha iyi bir tahmindir.

Bir örnekle görelim. x = 25 ve tahmin 13 olsun. *25 ÷ 13 ≈ 1,92*'dir. Gerçek karekök 5, 13 ile 1,92 arasındadır. İki sayının ortalaması *(13 + 1,92) ÷ 2 ≈ 7,46*'dır; 7,46, 13'ten 5'e çok daha yakındır. Aynı işlem yeni tahminle tekrarlanır.

Sunumdaki sözde kod yöntemi beş satırda yazar:

```
x = 25
tahmin = 1
while |tahmin * tahmin - x| > 0.01
    tahmin = (tahmin + x / tahmin) / 2
yaz(tahmin)
```

Satırlar sırasıyla şunu yapar:

1. `x = 25` : karekökü aranan sayı x değişkenine konur.
2. `tahmin = 1` : kaba bir başlangıç tahmini seçilir. Pozitif her başlangıç değeri işe yarar; 1 en basit seçimdir.
3. `while |tahmin * tahmin - x| > 0.01` : tahminin karesi ile x arasındaki fark, yani **hata**, hesaplanır. Dikey çizgiler **mutlak değeri** gösterir: farkın işareti atılır, yalnızca büyüklüğü kalır. Hata 0,01'den büyükse döngünün içindeki satır çalışır; değilse döngü biter. 0,01 değeri "yeterince yakın" sınırıdır; bu tür sınırlara **eşik** (*tolerance*) denir.
4. `tahmin = (tahmin + x / tahmin) / 2` : tahmin ile x/tahmin'in ortalaması yeni tahmin olur. Sonra 3. satıra dönülür ve hata yeniden denetlenir.
5. `yaz(tahmin)` : döngü bittiğinde son tahmin ekrana yazılır.

Döngünün her çalışmasına bir **tur** (*iteration*) denir. Sunum, algoritmayı x = 25 için tur tur izler. Şekil 1.5.8 tahminin sayı doğrusundaki yolculuğunu, Tablo 1.5.10 ise her turdaki değerleri gösterir.



Şekil 1.5.8: x = 25 için tahminin sayı doğrusundaki konumu; kesikli yeşil çizgi gerçek karekökü (5) gösterir. (a) 1. turdan sonra tahmin 13'tür; 1'den 5'in öbür yanına sıçramıştır. (b) 3. turdan sonra tahmin ≈ 5,406'dır. (c) 5. turdan sonra tahmin 5'e çok yakındır; soluk noktalar önceki tahminlerdir.

Tur	Eski tahmin	x / tahmin	Yeni tahmin	Hata |tahmin² − 25|	Karar
başlangıç	—	—	1	24	devam
1	1	25	13	144	devam
2	13	1,9231	7,4615	30,6746	devam
3	7,4615	3,3505	5,4060	4,2251	devam
4	5,4060	4,6245	5,0152	0,1527	devam

Tur	Eski tahmin	x / tahmin	Yeni tahmin	Hata |tahmin² − 25|	Karar
5	5,0152	4,9848	5,0000	0,0002	dur

Tablo 1.5.10: x = 25, başlangıç tahmini 1 için izleme tablosu. Değerler dört ondalık basamağa yuvarlanmıştır.

Tabloda dikkat çeken üç nokta vardır. Birincisi, ilk turda hata 24'ten 144'e çıktı. Tahmin 1'den 13'e sıçradı ve 5'i geçti. Bu bir hata değildir. 1 çok küçük bir tahmindi ve x/tahmin = 25 çok büyüktü; ortalamaları karekökün öbür yanına düştü. İkinci turdan sonra tahmin her zaman 5'in üstünde kalır ve her turda ona yaklaşır.

İkincisi, yaklaşma hızı giderek artar. Hata 144'ten 30,67'ye, sonra 4,23'e, 0,15'e ve 0,0002'ye indi. Son turlarda her tur, doğru basamak sayısını yaklaşık iki katına çıkarır. Bu yüzden yöntem yalnızca beş turda dört ondalık basamaklık doğruluğa ulaştı. Son tahmin tam olarak 5 değil, yaklaşık 5,00002'dir; ama hatası eşiğin çok altındadır.

Üçüncüsü, durma koşulu tahminin kendisine değil, karesine bakar. Algoritma karekökün gerçek değerini bilmez; bilseydi hesaplamaya gerek kalmazdı. Bildiği tek şey bildirimsel tanımdır: karekökün karesi x'e eşittir. Durma koşulu bu tanımı kullanır. Buyurgan yöntem ilerler, bildirimsel tanım ne zaman durulacağına karar verir.

Örnek 1.5.10

Aynı algoritma x = 2 ve tahmin = 1 ile çalıştırılıyor. Kaç tur sürer, hangi değer yazılır? Yazılan değer gerçek $\sqrt{2} \approx 1{,}41421$'e ne kadar yakındır?

Tur	Eski tahmin	x / tahmin	Yeni tahmin	tahmin²	Hata	Karar
başlangıç	—	—	1	1	1	devam
1	1	2	1,5	2,25	0,25	devam
2	1,5	1,3333	1,4167	2,0069	0,0069	dur

Algoritma iki turda durur ve yaklaşık 1,4167 yazar. İkinci turdan sonra hata 0,0069'dur; bu değer 0,01'den küçük olduğu için döngü biter. Yazılan değer gerçek $\sqrt{2}$'den yaklaşık $1{,}41667 - 1{,}41421 \approx 0{,}0025$ uzaktadır. Daha doğru bir sonuç isteniyorsa eşik küçültülür; örneğin 0,000001 seçilirse algoritma iki tur daha yapar ve 1,414214 bulur.

Sık yapılan hata

Başlangıç tahminini 0 seçmek. Yeni tahmin hesaplanırken x tahmine bölünür; tahmin 0 ise sıfıra bölme hatası oluşur ve algoritma çalışamaz. Negatif bir x de sorun çıkarır: hiçbir gerçek sayının karesi negatif olmadığı için hata hiçbir zaman 0,01'in altına inmez ve döngü sonsuza kadar sürer. Algoritmayı kullanan program, girdinin pozitif olduğunu önceden denetlemelidir.

1.5.5.4 Öklid algoritması

Bilinen en eski algoritmalardan biri, Yunan matematikçi Öklid'in MÖ 300 dolaylarında yazdığı *Elemanlar* adlı eserde yer alır. Algoritma iki pozitif tam sayının en büyük ortak bölenini bulur. Yaklaşık 2.300 yıllık olmasına rağmen bugün hâlâ kullanılır; örneğin kesirleri sadeleştirmede ve internet güvenliğini sağlayan şifreleme yöntemlerinde.

> **Tanım 1.5.8**
>
> İki tam sayının **en büyük ortak böleni** (*greatest common divisor*), kısaca **EBOB**, ikisini de kalansız bölen en büyük pozitif tam sayıdır. Sunumdaki sözde kodda bu işlem İngilizce kısaltmasıyla `GCD(a, b)` diye yazılır. a sayısının b'ye bölümünden kalan `a mod b` ile gösterilir; **mod** işlemi bölmeden kalanı verir.

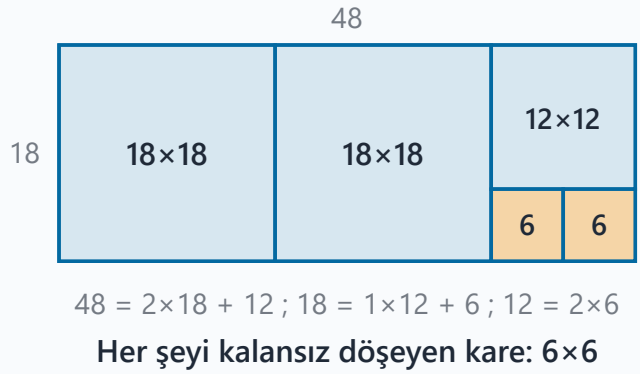
Tanımı doğrudan uygulayan bir yöntem düşünülebilir: küçük sayıdan başlayıp aşağı doğru her sayıyı denemek ve iki sayıyı da bölen ilk sayıda durmak. 48 ile 18 için denemeler 18, 17, 16, … diye gider ve 6'da durur; 13 deneme gerekir. Bu yöntem doğrudur ama yavaştır. 1.000.000 ile 999.999 için 999.999 deneme yapar, çünkü bu iki sayının EBOB'u 1'dir. Öklid'in yöntemi aynı soruyu yalnızca iki turda cevaplar.

Öklid algoritmasının sözde kodu şudur:

```
GCD(a, b)
while b ≠ 0
    r = a mod b
    a = b
    b = r
return a
```

Algoritma iki değişkenle çalışır: a ve b. Döngünün her turunda önce a'nın b'ye bölümünden kalan hesaplanıp r'ye konur. Sonra eski b, a'nın; kalan r de b'nin yerini alır. Böylece her turda problem daha küçük sayılarla aynı probleme dönüşür. b sıfır olunca döngü biter ve a'daki değer döndürülür.

Mod işlemine kısaca bakalım. *48 mod 18*, 48'in 18'e bölümünden kalandır. 18, 48'in içine iki kez sığar: *2 × 18 = 36*. Geriye *48 − 36 = 12* kalır. Demek ki *48 mod 18 = 12*'dir. Genel olarak $a = q \times b + r$ yazılır; burada q bölüm, r kalandır ve $0 \le r < b$'dir.



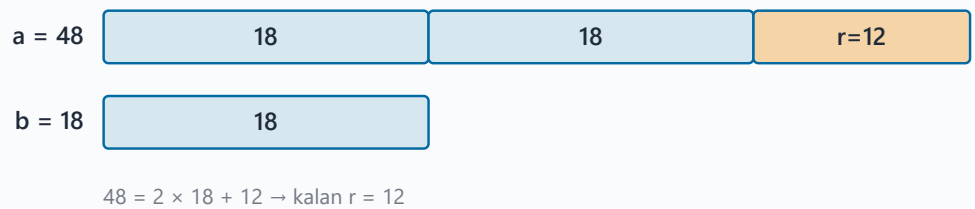
Şekil 1.5.9: Öklid algoritmasının geometrik yorumu. 48 × 18 boyutlarındaki dikdörtgenden önce iki tane 18 × 18'lik kare kesilir; 12 × 18'lik şerit kalır. Ondan bir 12 × 12'lik kare kesilir; 12 × 6'lık şerit kalır. O da iki tane 6 × 6'lık kareyle tam doldurulur. Dikdörtgeni kalansız döşeyen en büyük kare 6 × 6'dır.

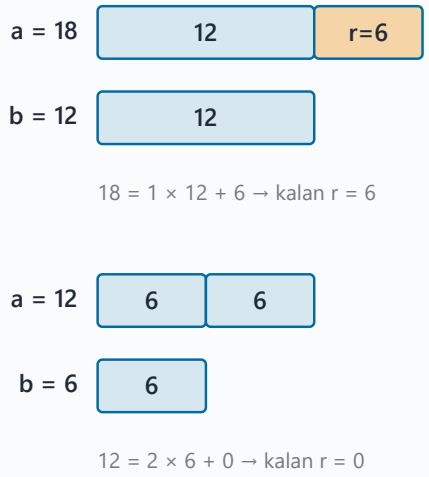
Şekil 1.5.9 algoritmanın gözle görülür bir yorumunu verir. 48 birim uzunluğunda ve 18 birim genişliğinde bir dikdörtgen, eşit karelerle kalansız döşenmek isteniyor. Kullanılabilecek en büyük karenin kenarı, 48 ile 18'in EBOB'udur. Dikdörtgenden kenarı 18 olan kareler kesildikçe geriye daha küçük bir dikdörtgen kalır. Her kesme, algoritmanın bir turuna karşılık gelir. Kalan dikdörtgen bir gün eş karelerle tam dolar; o karelerin kenarı aranan sayıdır.

1.5.5.5 Öklid algoritması adım adım

GCD(48, 18) çağrısını satır satır izleyelim. Başlangıçta a = 48 ve b = 18'dir.

1. **1. tur.** b ≠ 0 koşulu doğrudur (18 ≠ 0). r = 48 mod 18 = 12 olur: 48'den 18 uzunluğunda iki parça çıkar, 12 kalır. Sonra a = 18, b = 12 olur. Artık küçük sayı bölünen olmuştur.
2. **2. tur.** 12 ≠ 0 doğrudur. r = 18 mod 12 = 6 olur: 18'den 12 uzunluğunda bir parça çıkar, 6 kalır. Sonra a = 12, b = 6 olur.
3. **3. tur.** 6 ≠ 0 doğrudur. r = 12 mod 6 = 0 olur: 12'den 6 uzunluğunda iki parça çıkar, hiçbir şey kalmaz. Sonra a = 6, b = 0 olur.
4. **Döngü sonu.** 0 ≠ 0 koşulu yanlıştır; döngü biter. `return a` satırı 6'yı döndürür.





Şekil 1.5.10: GCD(48, 18) için her turdaki kalan hesabı; çubukların uzunluğu sayıların büyüklüğüdür. (a) 1. tur: 48 = 2 × 18 + 12. (b) 2. tur: 18 = 1 × 12 + 6. (c) 3. tur: 12 = 2 × 6 + 0; kalan sıfırdır.

Şekil 1.5.10 her turda a çubuğundan b uzunluğunda kaç parça çıktığını ve geriye ne kaldığını gösterir. Kalan, bir sonraki turda b'nin yerine geçer. Bütün çalışma Tablo 1.5.11'de özetlenmiştir.

Tur	a	b	b ≠ 0?	Bölme	r = a mod b	Yeni a	Yeni b
1	48	18	doğru	48 = 2 × 18 + 12	12	18	12
2	18	12	doğru	18 = 1 × 12 + 6	6	12	6
3	12	6	doğru	12 = 2 × 6 + 0	0	6	0
—	6	0	yanlış	Döngü biter; return a → 6			

Tablo 1.5.11: GCD(48, 18) için izleme tablosu. Her satır, bir turun başındaki a ve b değerlerini, hesaplanan kalanı ve turun sonundaki yeni değerleri gösterir.

Sonuç: GCD(48, 18) = 6'dır. Denetim kolaydır: *48 = 8 × 6* ve *18 = 3 × 6*. 8 ile 3'ün 1'den büyük ortak böleni yoktur; bu yüzden 6'dan büyük bir ortak bölen de yoktur. Algoritma üç turda bitti. Tanımı doğrudan uygulayan yöntem 13 deneme yapmıştı.

Örnek 1.5.11

GCD(270, 192) Öklid algoritmasıyla hesaplanırsa her turda a, b ve r nasıl değişir? Sonuç kaçtır?

Tur	a	b	Bölme	r
1	270	192	270 = 1 × 192 + 78	78
2	192	78	192 = 2 × 78 + 36	36
3	78	36	78 = 2 × 36 + 6	6
4	36	6	36 = 6 × 6 + 0	0

Dördüncü turun sonunda a = 6 ve b = 0 olur; döngü biter ve 6 döndürülür. Denetim: *270 = 45 × 6*, *192 = 32 × 6*; 45 ile 32'nin 1'den büyük ortak böleni yoktur. Bu sonuçla *192/270* kesri, pay ve payda 6'ya bölünerek *32/45* biçiminde sadeleştirilir.

Sık yapılan hata

İki atama satırının sırasını değiştirmek. Döngüde önce `a = b`, sonra `b = r` yazılır. Önce `b = r` yazılırsa b'nin eski değeri kaybolur ve ardından gelen `a = b` satırı a'ya da r'yi koyar. 48 ile 18 için birinci turdan sonra a ve b'nin ikisi de 12 olur. İkinci turda 12 mod 12 = 0 çıkar; bu kez a ile b'nin ikisi de 0 olur ve algoritma 6 yerine 0 döndürür. Değişkenlerin değerlerini sırayla güncellerken hangi değerin hâlâ gerektiği her zaman denetlenmelidir.

1.5.5.6 Öklid algoritması neden doğru sonuç verir ve durur?

Bir algoritmanın birkaç örnekte doğru sonuç vermesi, her girdide doğru çalıştığını göstermez. Öklid algoritması için iki soru sorulur: neden her zaman doğru sonucu bulur ve neden her zaman durur?

Teorem 1.5.1 (Öklid)

a ve b pozitif tam sayılar ve *r = a mod b* olsun. a ile b'nin ortak bölenleri, b ile r'nin ortak bölenleriyle aynıdır. Dolayısıyla *EBOB(a, b) = EBOB(b, r)*'dir.

Teoremin gerekçesi şöyledir. Bölme işlemi $a = q \times b + r$ biçiminde yazılır; buradan $r = a - q \times b$ elde edilir. Bir d sayısı hem a'yı hem b'yi bölüyorsa, $a - q \times b$ farkını da böler; yani r'yi de böler. Tersine, d hem b'yi hem r'yi bölüyorsa $q \times b + r$ toplamını, yani a'yı da böler. Demek ki iki sayı çiftinin ortak bölenleri aynıdır. Ortak bölenleri aynı olan iki çiftin en büyük ortak böleni de aynıdır.

Örnekte bunu sayılarla görmek mümkündür. 48 ile 18'in ortak bölenleri 1, 2, 3 ve 6'dır. 18 ile 12'nin ortak bölenleri de 1, 2, 3 ve 6'dır. Sayılar küçüldü ama ortak bölenler değişmedi. Algoritmanın her turu, EBOB'u koruyarak problemi küçültür. Son turda ikinci sayı 0 olur. Her pozitif sayı 0'ı böldüğü için a ile 0'ın ortak bölenleri a'nın bölenleridir ve bunların en büyüğü a'nın kendisidir. Bu yüzden algoritma a'yı döndürür.

Durma sorusunun cevabı kalanın özelliğinden gelir. Kalan her zaman bölenden küçüktür: $0 \le r < b$. Her turda b'nin yerine r geçtiği için b her turda kesinlikle küçülür. b, negatif olmayan bir tam sayıdır. Negatif olmayan bir tam sayı sonsuza kadar küçülemez; en geç b kez küçüldükten sonra 0'a ulaşır. Bu yüzden döngü her girdide sonlu sayıda turdan sonra biter. Pratikte tur sayısı b'den çok daha küçüktür. Kalanlar en az her iki turda bir yarıya iner. Bu yüzden bir milyondan küçük sayılar için bile 30 civarından fazla tur gerekmez; en kötü durumdaki 832.040 ile 514.229 çifti 28 turda biter.

Örnek 1.5.12

Algoritma GCD(36, 84) ile, yani küçük sayı önce verilerek çağrılırsa ne olur?

1. **1. tur.** 36 mod 84 = 36'dır; 84, 36'nın içine hiç sığmaz ve sayının tamamı kalır. Sonra a = 84, b = 36 olur. Sayılar yer değiştirmiştir.
2. **2. tur.** 84 mod 36 = 12 ($84 = 2 \times 36 + 12$). a = 36, b = 12 olur.
3. **3. tur.** 36 mod 12 = 0. a = 12, b = 0 olur ve döngü biter.

Algoritma 12 döndürür. Sayıların sırası sonucu değiştirmez; yalnızca sayıları yer değiştirmek için bir tur fazladan harcanır. Bu yüzden algoritmanın girdilerin sıralı verilmesini şart koşmasına gerek yoktur.

Sık yapılan hata

Algoritmayı b = 0 ile çağırmanın hata vereceğini sanmak. GCD(15, 0) çağrısında döngü koşulu ilk denetimde yanlıştır; döngü hiç çalışmaz ve 15 döndürülür. Bu doğrudur, çünkü 15 hem 15'i hem 0'ı böler. Gerçek sorun GCD(0, 0) çağrısıdır: her sayı 0'ı böldüğü için en büyük ortak bölen tanımsızdır, ama algoritma 0 döndürür. Algoritmanın tanımı bu yüzden "iki pozitif tam sayı" der.

1.5.6 Hesaplamalı düşünme

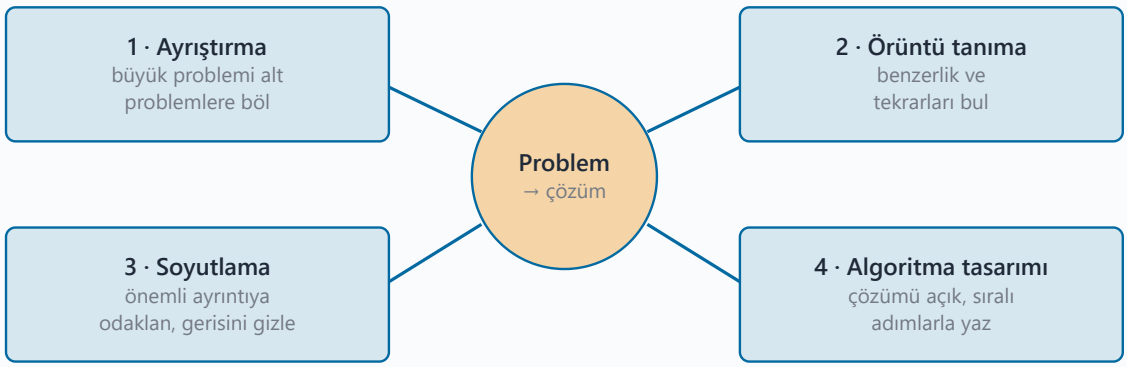
Karekök ve Öklid algoritmaları hazır problemlere hazır çözümlerdi. Gerçek hayattaki problemler ise çoğu zaman dağınık gelir: "Öğrencilerin derse kaydını kolaylaştıran bir sistem lazım." Böyle bir istekten çalışan programa giden yol birkaç işten geçer. Problem parçalara ayrılır, parçalar arasındaki benzerlikler görülür, önemli olan önemsizden ayrılır ve en sonunda adımlar yazılır. Bu düşünme biçimine hesaplamalı düşünme denir. Bu kesim hesaplamalı düşünmenin dört bileşenini tek tek ele alır ve sonunda hepsini bir örnekte birleştirir.

1.5.6.1 Hesaplamalı düşünme nedir?

Tanım 1.5.9

Hesaplamalı düşünme (*computational thinking*), problemleri bir bilgisayarın ya da bir insanın sistemli biçimde çözebileceği yapılara dönüştürme yaklaşımıdır. Dört bileşeni vardır: **ayrıştırma**, **örüntü tanıma**, **soyutlama** ve **algoritma tasarımı**.

Kavram, bilgisayar bilimci Jeannette Wing'in 2006'da yayımladığı kısa bir makaleyle yaygınlaştı. Wing'e göre hesaplamalı düşünme yalnızca programcıların değil, herkesin öğrenmesi gereken temel bir beceridir; tıpkı okuma, yazma ve aritmetik gibi. Hesaplamalı düşünmek bilgisayar gibi düşünmek demek değildir. Bilgisayar yalnızca kendisine söyleneni yapar; hesaplamalı düşünme ise ona ne söyleneceğini bulma becerisidir.



Şekil 1.5.11: Hesaplamalı düşünmenin dört bileşeni problemin çevresinde: (1) ayrıştırma büyük problemi alt problemlere böler; (2) örüntü tanıma benzerlikleri ve tekrarları bulur; (3) soyutlama önemli ayrıntıya odaklanıp gerisini gizler; (4) algoritma tasarımı çözümü açık ve sıralı adımlarla yazar.

Şekil 1.5.11 bileşenleri bir sıra içinde numaralandırır ama uygulamada bu sıra katı değildir. Algoritma yazılırken yeni bir alt problem fark edilebilir ve ayrıştırmaya geri dönülür. Bir örüntü, soyutlamanın neye odaklanacağını belirleyebilir. Dört bileşen birlikte ve birbirini besleyerek çalışır.

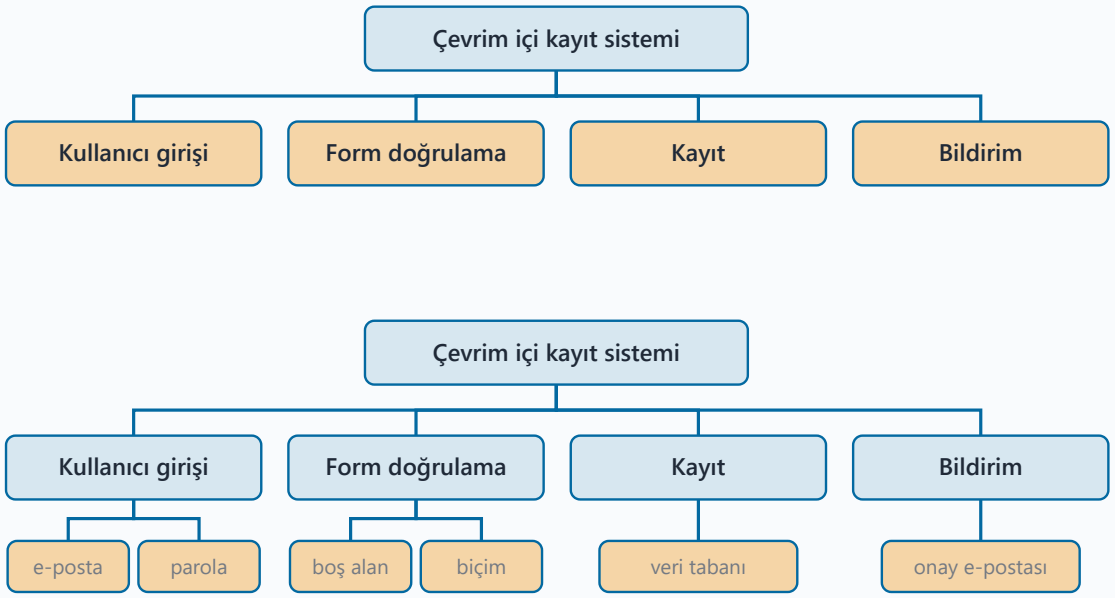
Hesaplamalı düşünmenin izleri günlük hayatta da görülür. Bir aşçı büyük bir yemeği hazırlık, pişirme ve sunum aşamalarına ayırır; bu ayrıştırmadır. Bir öğretmen her yıl benzer hataları yapan öğrenciler için aynı ek çalışma sayfasını kullanır; bu örüntü tanımadır. Bir şehir haritası binaların renklerini değil, sokakları gösterir; bu soyutlamadır. Bir yemek tarifi de bir algoritmadır.

1.5.6.2 Ayrıştırma

Tanım 1.5.10

Ayrıştırma (*decomposition*), büyük ve karmaşık bir problemi daha küçük, daha kolay çözülebilen alt problemlere bölmektir. Her alt problemin girdisi ve çıktısı ayrı ayrı tanımlanır.

Sunumdaki örnek bir çevrim içi kayıt sistemidir. Kullanıcılar bir web sayfasından kayıt olacak, bilgileri saklanacak ve kendilerine bir onay e-postası gidecektir. Bu sistemi tek parça olarak düşünmek zordur: nereden başlanacağı belli değildir ve bir hata çıktığında nerede olduğu bulunamaz.



Şekil 1.5.12: Çevrim içi kayıt sisteminin ayrıştırılması. (a) Birinci düzey: sistem kullanıcı girişi, form doğrulama, kayıt ve bildirim alt görevlerine bölünür. (b) İkinci düzey: her alt görev gerekirse daha küçük işlere bölünür; örneğin form doğrulama, boş alan denetimi ve biçim denetiminden oluşur.

Şekil 1.5.12 ayrıştırmanın iki düzeyini gösterir. Birinci düzeyde sistem dört ana alt göreve bölünür. İkinci düzeyde bazı alt görevler yeniden bölünür. Ağacın en altındaki her yaprak küçük, tek başına yazılıp test edilebilen bir iştir. Örneğin "biçim denetimi" yalnızca e-posta adresinde @ işareti olup olmadığına ve parolanın en az sekiz karakter olup olmadığına bakan küçük bir parça olabilir.

Alt görev	Girdi	Çıktı
Kullanıcı girişi	E-posta ve parola	Oturum açıldı ya da hata mesajı
Form doğrulama	Formdaki alanlar	"Geçerli" ya da hatalı alanların listesi
Kayıt	Doğrulanmış form bilgileri	Veri tabanında yeni kayıt ve kayıt numarası
Bildirim	E-posta adresi ve kayıt numarası	Kullanıcıya gönderilmiş onay e-postası

Tablo 1.5.12: Kayıt sisteminin alt görevleri ve her birinin girdisi ile çıktısı.

Tablo 1.5.12 ayrıştırmanın asıl değerini ortaya koyar: alt görevler girdi ve çıktılarıyla birbirine bağlanır. Form doğrulamanın çıktısı, kaydın girdisidir. Kaydın ürettiği kayıt numarası, bildirimin girdisidir. Bu bağlar açıkça yazıldığında üç önemli yarar ortaya çıkar.

- **Ayrı geliştirme ve test.** Her alt problem ayrı geliştirilebilir ve test edilebilir. Bildirim parçasını yazan kişi, form doğrulamanın nasıl çalıştığını bilmek zorunda değildir; yalnızca kendisine hangi bilgilerin geleceğini bilmelidir.
- **Açık arayüzler.** Alt problemlerin girdi ve çıktıları açıkça tanımlanır. Bir parça değiştirildiğinde, girdisi ve çıktısı aynı kaldığı sürece öteki parçalar etkilenmez.

- **Bağımlılıkların erken görülmesi.** Hangi parçanın hangisini beklediği erken fark edilir. Bildirim, kayıt bitmeden çalışamaz; çünkü kayıt numarasına ihtiyaç duyar. Bu bilgi, işlerin hangi sırayla yapılacağını belirler.

Örnek 1.5.13

Bir bölüm, final sınavı için oturma planı hazırlayan bir program istiyor. 250 öğrenci var; salonların kapasiteleri 80, 80, 60 ve 40 kişidir. Problem nasıl ayrıştırılır ve öğrenciler salonlara nasıl dağılır?

Alt görev	Girdi	Çıktı
Öğrenci listesini oku	Öğrenci bilgi sisteminden alınan dosya	250 öğrencilik liste
Salonları oku	Salon adları ve kapasiteleri	4 salonluk liste, toplam kapasite
Kapasiteyi denetle	Öğrenci sayısı, toplam kapasite	"Yeterli" ya da "yetersiz" uyarısı
Öğrencileri dağıt	Öğrenci listesi, salon listesi	Her öğrenci için salon ve sıra numarası
Listeleri yazdır	Dağıtım sonucu	Her salonun kapısına asılacak liste

1. **Kapasite denetimi.** $80 + 80 + 60 + 40 = 260 \geq 250$; kapasite yeterlidir ve 10 sıra boş kalır.
2. **Dağıtım.** Salonlar sırayla doldurulursa ilk salona 80, ikincisine 80, üçüncüsüne 60 öğrenci yerleşir. Bu üç salonda $80 + 80 + 60 = 220$ öğrenci vardır. Kalan $250 - 220 = 30$ öğrenci dördüncü salona gider; orada $40 - 30 = 10$ sıra boş kalır.

Ayrıştırma bir bağımlılığı da ortaya çıkardı: dağıtım, kapasite denetiminden önce yapılamaz. Öğrenci sayısı 270 olsaydı kapasite denetimi "yetersiz" uyarısı verir ve dağıtım hiç başlamazdı.

Sık yapılan hata

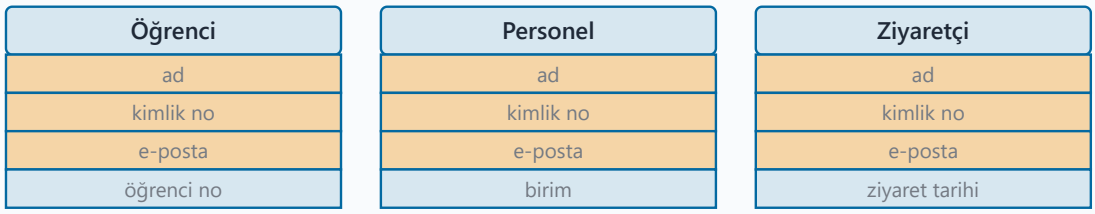
Alt görevleri girdi ve çıktılarını yazmadan yalnızca adlarıyla listelemek. "Doğrulama", "kayıt", "bildirim" diye yazılmış bir liste, parçaların birbirine ne verdiğini söylemez. Parçalar ayrı ayrı yazılıp birleştirildiğinde biri telefon numarası beklerken öteki e-posta adresi gönderebilir. Her alt görev için "ne alır, ne verir?" sorusu cevaplanmalıdır.

1.5.6.3 Örüntü tanıma

Tanım 1.5.11

Örüntü tanıma (*pattern recognition*), farklı problemler ya da bir problemin farklı parçaları arasındaki benzerlikleri ve tekrarları bulmaktır.

Kayıt sistemine geri dönelim. Sistem üç tür kişiyi kaydedecektir: öğrenciler, personel ve ziyaretçiler. Her biri için ayrı bir kayıt formu tasarlanabilir. Ancak formlar yan yana konunca bir şey dikkati çeker.



Şekil 1.5.13: Üç kayıt türü: öğrenci, personel ve ziyaretçi. Ad, kimlik no ve e-posta alanları üçünde de tekrar eder; vurgulu satırlar bu ortak alanlardır. Her türün yalnızca bir alanı kendine özgüdür: öğrenci no, birim ve ziyaret tarihi.

Şekil 1.5.13 tekrar eden yapıyı gösterir. Üç formun dört alanından üçü aynıdır. Bu bir örüntüdür. Örüntüyü fark etmenin pratik bir değeri vardır. Ad alanının boş bırakılmadığını, kimlik numarasının 11 haneli olduğunu ve e-posta adresinin doğru biçimde yazıldığını denetleyen kodu üç kez yazmak gerekmez. Bir kez yazılır, üç formda kullanılır. Kimlik numarası denetiminde bir hata bulunduğunda da tek bir yerde düzeltilir.

Örüntüler yalnızca veride değil, işlemlerde de görülür. Ortalama hesabı ve Öklid algoritması farklı işler yapar, ama ikisi de aynı kalıbı izler: bir koşul doğru olduğu sürece bir işlemi tekrarla, koşul yanlış olunca sonucu ver. Karekök algoritması da bu kalıba uyar. Bu kalıp o kadar sık görülür ki her programlama dilinde onun için özel bir yapı, döngü vardır.

Örnek 1.5.14

Bir yurt yönetimi, aylık su, elektrik ve doğal gaz faturalarını hesaplayan bir program yazdırmak istiyor. Bu ayın değerleri şöyle olsun (fiyatlar örnek amaçlıdır): su 12 m^3, birim fiyat 30 TL, sabit ücret 40 TL; elektrik 180 kWh, birim fiyat 3 TL, sabit ücret 60 TL; doğal gaz 90 m^3, birim fiyat 8 TL, sabit ücret 20 TL. Üç hesapta hangi örüntü vardır?

1. **Su.** *12 × 30 + 40 = 360 + 40 = 400* TL.
2. **Elektrik.** *180 × 3 + 60 = 540 + 60 = 600* TL.
3. **Doğal gaz.** *90 × 8 + 20 = 720 + 20 = 740* TL.
4. **Örüntü.** Üç hesap da aynı kalıba uyar: *tutar = tüketim × birim fiyat + sabit ücret*. Değişen yalnızca üç sayıdır.

Toplam fatura *400 + 600 + 740 = 1.740* TL'dir. Örüntü fark edilince program üç ayrı hesap yerine tek bir hesap kuralı içerir ve bu kural üç kez farklı sayılarla çalıştırılır. Yurt ileride internet faturasını da eklemek isterse, aynı kurala yeni üç sayı vermek yeter.

Sık yapılan hata

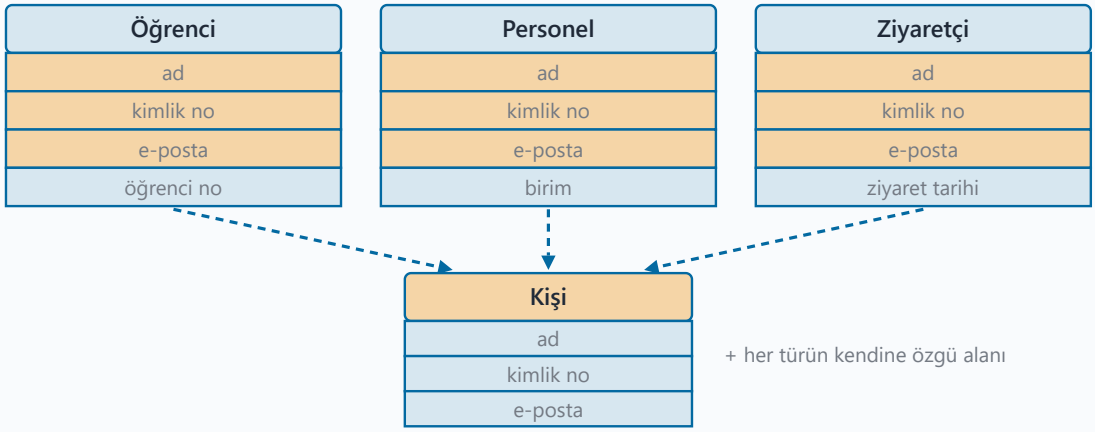
Yüzeysel benzerliği örüntü sanmak. Öğrenci numarası ile kimlik numarasının ikisi de rakamlardan oluşur, ama biri 11 haneli ve kurallı bir kimlik numarası, öteki üniversiteye özgü bir numaradır. Bunları aynı denetimden geçirmek hatalı sonuç verir. Örüntü, yalnızca görünüşte değil, anlamda ve kurallarda da benzerlik demektir.

1.5.6.4 Soyutlama

Tanım 1.5.12

Soyutlama (*abstraction*), bir amaç için önemli olan ayrıntılara odaklanıp gereksiz ayrıntıları gizlemektir. Soyutlamanın ürünü, gerçeğin basitleştirilmiş bir **modelidir**.

Örüntü tanıma tekrar eden yapıyı bulur; soyutlama bu yapıya bir ad verir ve onu ayrıntılardan ayırır. Kayıt örneğinde üç türün ortak alanları tek bir “Kişi” modelinde toplanır.



Şekil 1.5.14: Soyutlama. Üç kayıt türündeki ortak alanlar (ad, kimlik no, e-posta) tek bir Kişi modelinde toplanır. Her tür, Kişi modeline yalnızca kendine özgü alanı ekler.

Şekil 1.5.14 sonucu gösterir. Kişi modelinde üç alan vardır. Öğrenci kaydında Kişi'nin bütün alanları bulunur; bunlara ek olarak bir öğrenci numarası vardır. Personel bir birim, ziyaretçi bir ziyaret tarihi ekler. Sistemin büyük bölümü, örneğin e-posta gönderen bildirim parçası, yalnızca Kişi modeliyle çalı-

şır. Bildirim parçası, e-posta gönderdiği kişinin öğrenci mi, personel mi olduğunu bilmek zorunda değildir; yalnızca e-posta alanına ihtiyaç duyar. Farklılıklar yalnızca gerektiği yerde, örneğin öğrenci listesini hazırlayan parçada, görülür.

İyi bir soyutlama, önemli farkları kaybetmeden tekrar eden yapıyı ortaklaştırır. Kişi modeline öğrenci numarasını da koymak yanlış olurdu; çünkü ziyaretçinin öğrenci numarası yoktur ve bu alan çoğu kayıtta boş kalırdı. E-posta alanını modelden çıkarmak da yanlış olurdu; çünkü bildirim parçası çalışamazdı. Soyutlamanın ölçüsü amaçtır: amaca hizmet eden ayrıntı kalır, hizmet etmeyen gizlenir.

Örnek 1.5.15

Bir fakülte binasının koridorlarına asılan yangın kaçış planı, binanın hangi özelliklerini gösterir, hangilerini gizler? Bu soyutlama neden işe yarar?

1. **Amaç.** Plan, yangın anında binadaki birinin en kısa yoldan dışarı çıkmasını sağlamalıdır.
2. **Korunan ayrıntılar.** Koridorlar, merdivenler, çıkış kapıları, yangın söndürücülerin yeri ve "Buradasınız" işareti.
3. **Gizlenen ayrıntılar.** Odalardaki mobilyalar, duvarların rengi, pencerelerin sayısı, odaların kime ait olduğu, binanın yapım yılı.
4. **Neden işe yarar?** Panik anında okuyan kişinin dikkati yalnızca çıkış yoluna yönelir. Mobilyalar planda olsaydı çizim kalabalıklaşır ve yol daha geç bulunurdu.

Aynı bina için mimarın kullandığı plan çok daha ayrıntılıdır; duvar kalınlıklarını ve elektrik hatlarını da gösterir. Bir bina için tek bir "doğru" model yoktur. Her amaç kendi soyutlamasını gerektirir.

Sık yapılan hata

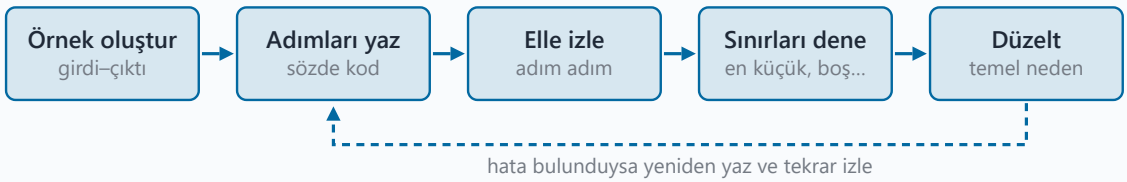
Soyutlamayı "ayrıntıları silmek" sanmak. Soyutlamada ayrıntılar yok olmaz, gizlenir. Öğrenci numarası Kişi modelinde görünmez ama öğrenci kaydında durmaya devam eder. Gerektiğinde, örneğin bir öğrencinin numarası sorgulanırken, o ayrıntıya yeniden ulaşılır. Gereken ayrıntıyı gerçekten silen bir model, işe yaramaz hâle gelir.

1.5.6.5 Algoritma tasarımı ve test

Tanım 1.5.13

Algoritma tasarımı (*algorithm design*), bir problemin çözümünü açık ve sıralı adımlarla yazmaktır. **Test** (*testing*), yazılan adımların farklı girdilerde beklenen çıktıyı üretip üretmediğini denetlemektir. Denetim için seçilen her girdi ve beklenen çıktı çiftine **test durumu** (*test case*) denir.

Algoritma tasarımı tek seferde biten bir iş değildir; bir döngü içinde ilerler.



Şekil 1.5.15: Algoritma tasarım ve test döngüsü: girdi ve beklenen çıktıdan oluşan örnekler hazırlanır, adımlar sözde kodla yazılır, elle adım adım izlenir, sınır durumları denenir ve bulunan hatanın temel nedeni düzeltilir. Hata bulunduysa adımlar yeniden yazılır ve tekrar izlenir.

Şekil 1.5.15'teki döngünün beş aşaması vardır.

1. **Örnek oluşturma.** Kod yazmadan önce birkaç girdi ve bu girdiler için beklenen çıktılar hazırlanır. Ortalama için: 70, 85, 60, 75 girdisinin beklenen çıktısı 72,5'tir. Beklenen çıktı elle hesaplanır; algoritmanın sonucu bununla karşılaştırılacaktır.
2. **Adımları yazma.** Çözüm sözde kodla yazılır.
3. **Elle izleme.** Sözde kod, hazırlanan örnek girdiyle kâğıt üzerinde satır satır çalıştırılır. Her adımda değişkenlerin değeri bir **izleme tablosuna** (*trace table*) yazılır. Bu notta Turing makinesi, karekök ve Öklid için kullanılan tablolar böyle tablolardır.
4. **Sınırları deneme.** Normal girdilerin yanında en küçük, en büyük ve geçersiz girdiler de denenir. Bu tür girdilere **sınır durumu** (*edge case*) denir. Hataların çoğu bu girdilerde ortaya çıkar.
5. **Düzeltme.** Bir hata bulunduğunda yalnızca belirti değil, hatanın **temel nedeni** düzeltilir. Sonra döngü yeniden başlar: düzeltilen algoritma bütün örneklerle tekrar izlenir.

Bu döngüyü bir örnekle izleyelim. Bir not listesinin ortalamasını bulan şu algoritma yazılmış olsun:

```
toplam = 0
sayac = 0
while okunacak not var
    toplam = toplam + not
    sayac = sayac + 1
yaz(toplam / sayac)
```

Algoritma, Örnek 1.5.1'deki hesabın aynısını yapar: toplam ve sayaç 0'dan başlar, her not toplama eklenir ve sayaç bir artar; sonunda toplam sayaca bölünür. Şimdi farklı türden girdilerle test edelim.

Tür	Girdi	Beklenen çıktı	Algoritmanın çıktısı	Sonuç
Normal	70, 85, 60, 75	72,5	290 / 4 = 72,5	geçti
En küçük	100 (tek not)	100	100 / 1 = 100	geçti
Uç değerler	0, 100	50	100 / 2 = 50	geçti

Tür	Girdi	Beklenen çıktı	Algoritmanın çıktısı	Sonuç
Geçersiz	(hiç not yok)	"Hiç not girilmedi" uyarısı	0 / 0: sıfıra bölme hatası	kaldı

Tablo 1.5.13: Ortalama algoritması için test durumları.

Tablo 1.5.13 son satırda bir hata yakaladı. Hiç not girilmezse döngü hiç çalışmaz; toplam ve sayaç 0'da kalır ve son satırda 0, 0'a bölünür. Program çöker. Normal girdilerle yapılan testler bu hatayı hiçbir zaman göstermezdi.

Hatayı düzeltmenin iki yolu vardır. Birincisi belirtiyi gidermektir: "sayaç 0 ise ortalamayı 0 kabul et." Program artık çökmez ama yanlış bir şey söyler: notu hiç olmayan bir öğrencinin ortalaması 0 görünür ve öğrenci dersten kalmış sanılır. İkincisi temel nedeni düzeltmektir. Temel neden şudur: boş bir listenin ortalaması tanımsızdır ve algoritma bu durumu hiç düşünmemiştir. Doğru düzeltme, bu durumu ayrıca ele almaktır:

```
toplam = 0
sayac = 0
while okunacak not var
    toplam = toplam + not
    sayac = sayac + 1
if sayac = 0
    yaz("Hiç not girilmedi")
else
    yaz(toplam / sayac)
```

Düzeltilmiş algoritmada `if` satırı bir karar verir: sayaç 0 ise uyarı yazılır, değilse (`else`) ortalama hesaplanır. Algoritma dört test durumuyla yeniden izlendiğinde ilk üçü yine 72,5, 100 ve 50 verir; dördüncüsü artık beklenen uyarıyı yazar.

> **Sık yapılan hata**
>
> Yalnızca normal girdilerle test etmek. Rastgele seçilmiş yüzlerce olağan liste, boş liste hatasını bir kez bile göstermez. Test durumlarının sayısı değil, çeşitliliği önemlidir. İyi bir test kümesi en az bir normal, bir en küçük, bir uç değer ve bir geçersiz girdi içerir.

1.5.6.6 Dört bileşen bir arada: not hesaplama aracı

Hesaplamalı düşünmenin dört bileşeni bir problemi baştan sona çözerken nasıl birlikte çalışır? Bunu küçük bir problemle görelim. Bir öğretim elemanı, öğrencilerin dönem sonu puanını hesaplayan basit bir araç istiyor. Puan, vize notunun %40'ı ile final notunun %60'ının toplamıdır. Puanı 50 ve üstü olan öğrenci geçer.

Ayrıştırma. Problem dört alt göreve bölünür. Notları okuma: girdi öğrencinin vize ve final notu, çıktı iki sayı. Notları doğrulama: girdi iki sayı, çıktı "geçerli" ya da "geçersiz". Puan hesaplama: girdi iki geçerli not, çıktı puan. Karar: girdi puan, çıktı "geçti" ya da "kaldı".

Örüntü tanıma. Vize ve final notları aynı kurallara uyar: ikisi de 0 ile 100 arasında olmalıdır. Doğrulama kuralı bir kez yazılır ve iki nota da uygulanır. Puan hesabı da bir örüntüdür: her not kendi ağırlığıyla çarpılır ve sonuçlar toplanır. İleride ödev notu eklenirse aynı kalıba bir terim daha eklenir.

Soyutlama. Araç için bir öğrenci yalnızca iki sayıdan ibarettir: vize ve final. Öğrencinin adı, bölümü, devam durumu ya da sınav kâğıdındaki cevaplar bu hesabı etkilemez ve modelde yer almaz.

Algoritma tasarımı. Adımlar sözde kodla yazılır:

```
oku(vize, final)
if vize < 0 or vize > 100 or final < 0 or final > 100
    yaz("Geçersiz not")
else
    puan = (40 * vize + 60 * final) / 100
    if puan >= 50
        yaz(puan, "geçti")
    else
        yaz(puan, "kaldı")
```

Puan formülünde yüzdeler 0,4 ve 0,6 yerine 40 ve 60 olarak yazılıp sonuç 100'e bölünür. İki yazım matematikte aynıdır. Ancak bilgisayarlar 0,4 gibi ondalık sayıları ikili sistemde tam olarak saklayamaz ve küçük yuvarlama farkları oluşabilir. Örneğin *55 × 0,4 + 72 × 0,6* işlemi bir bilgisayarda 65,2 yerine 65,19999… sonucunu verebilir. Tam sayılarla çarpıp en sonda bölmek bu riski azaltır.

Örnek 1.5.16

Not hesaplama aracı aşağıdaki girdilerle test ediliyor. Her girdi için araç ne yazar?

Tür	Vize	Final	Hesap	Çıktı
Normal	55	72	(40 × 55 + 60 × 72) / 100 = (2.200 + 4.320) / 100	65,2 geçti
Normal	40	45	(1.600 + 2.700) / 100	43 kaldı
Sınır	80	30	(3.200 + 1.800) / 100	50 geçti
Uç değer	100	100	(4.000 + 6.000) / 100	100 geçti
Uç değer	0	0	(0 + 0) / 100	0 kaldı
Geçersiz	−5	70	Vize 0'dan küçük	Geçersiz not
Geçersiz	60	105	Final 100'den büyük	Geçersiz not

> Üçüncü satır en öğretici olanıdır. Puan tam 50'dir ve kural "50 ve üstü geçer" der. Algoritmada `puan >= 50` yerine yanlışlıkla `puan > 50` yazılsaydı bu öğrenci kalmış görünürdü. Normal girdiler bu farkı hiçbir zaman göstermez; yalnızca tam sınırdaki girdi gösterir.

Bu küçük örnekte dört bileşenin her biri bir iş gördü. Ayrıştırma işi parçalara böldü ve hangi parçanın hangisini beklediğini gösterdi. Örüntü tanıma doğrulama kuralının iki kez yazılmasını önledi. Soyutlama gereksiz öğrenci bilgilerini dışarıda bıraktı. Algoritma tasarımı ve test, sınırdaki bir öğrencinin haksız yere kalmasına yol açabilecek bir hatayı daha kod yazılmadan yakaladı.

1.5.7 Özet

- **Hesaplama**, verinin kurallara göre işlenmesidir; aritmetikle sınırlı değildir. Arama, sıralama, karar verme, dönüştürme ve iletişim de hesaplamadır. Her hesaplamada bir **girdi**, bir **işlem** ve bir **çıktı** vardır; işlem zaman, bellek, enerji ve iletişim kaynaklarını kullanır.
- Hesaplamanın yapı taşları matematik, mantık, veri, algoritma, donanım ve programlama dilidir. Algoritma bir fikirdir; program o fikrin belirli bir dildeki yazılışıdır.
- **Bilgisayar bilimi** bilginin temsilini, işlenmesini ve problemlerin çözüm yöntemlerini inceler. Başlıca ilgi alanları soyutlama, algoritmalar, sistemler ve insan ile toplumdur. Teknolojinin sorumlulukları **erişilebilirlik**, **mahremiyet**, **adalet** ve **sürdürülebilirliktir**.
- Babbage ve Lovelace'tan Turing'e, Colossus ve ENIAC'tan bugünkü telefonlara kadar temel fikirler aynı kaldı; donanım hızlanıp küçüldükçe bu fikirler daha geniş alanlara yayıldı.
- **Sabit programlı** makinede iş, makinenin fiziksel düzeniyle belirlenir. **Depolanan programlı** makinede komutlar bellekte veri gibi saklanır; yeni bir program yüklemek makineye yeni bir iş verir. Bu fikir genel amaçlı bilgisayarı mümkün kıldı.
- **Turing makinesi** bant, okuma/yazma başlığı, durum ve geçiş kurallarından oluşan soyut bir hesaplama modelidir. Her adımı yalnızca (durum, okunan simge) çifti belirler. 1 ekleyen makine 111'i dört adımda 1000'e çevirir.
- **Evrensel Turing makinesi**, bantta verilen başka bir makinenin kurallarını okuyup onu taklit eder. Depolanan program fikrinin kuramsal temeli budur.
- Bilgisayarların sınırları zaman, bellek, enerji, hesaplanabilirlik, veri kalitesi ve modelden gelir. **Durma problemi** gibi hesaplanabilirlik sınırları hiçbir donanımla aşılamaz.
- **Bildirimsel bilgi** neyin doğru olduğunu, **buyurgan bilgi** bir sonuca nasıl ulaşılacağını anlatır. **Algoritma**, buyurgan bilgiyi açık, uygulanabilir ve sonlu adımlara dönüştürür.
- Karekök yöntemi tahmin ile x/tahmin'in ortalamasını alarak ilerler ve tahminin karesi x'e yeterince yaklaşınca durur; x = 25 için beş turda 5,0000'a ulaşır.
- **Öklid algoritması** *EBOB(a, b) = EBOB(b, a mod b)* eşitliğini kullanır; kalan her turda küçüldüğü için her zaman durur. GCD(48, 18) üç turda 6 verir.

- **Hesaplamalı düşünme** dört bileşeni birleştirir: **ayrıştırma** (alt problemlere bölme, girdi ve çıktıları tanımlama), **örüntü tanıma** (tekrarları bulma), **soyutlama** (önemliye odaklanıp gerisini gizleme) ve **algoritma tasarımı** (açık ve sıralı adımlar). Tasarım normal, en küçük, uç ve geçersiz girdilerle test edilir; hatanın belirtisi değil, temel nedeni düzeltilir.

1.5.8 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

hesaplama *computation* · girdi *input* · bilgisayar bilimi *computer science* · erişilebilirlik *accessibility* · mahremiyet *privacy* · adalet *fairness* · sürdürülebilirlik *sustainability* · sabit program *fixed program* · depolanan program *stored program* · genel amaçlı *general purpose* · von Neumann mimarisi *von Neumann architecture* · Turing makinesi *Turing machine* · bant *tape* · simge *symbol* · okuma/yazma başlığı *read/write head* · durum *state* · geçiş kuralı *transition rule* · evrensel Turing makinesi *universal Turing machine* · hesaplanabilirlik *computability* · durma problemi *halting problem* · bildirimsel bilgi *declarative knowledge* · buyurgan bilgi *imperative knowledge* · algoritma *algorithm* · sözde kod *pseudocode* · tur *iteration* · eşik *tolerance* · sonsuz döngü *infinite loop* · Öklid algoritması *Euclid's algorithm* · en büyük ortak bölen (EBOB) *greatest common divisor* · kalan *remainder* · mod *modulo* · hesaplamalı düşünme *computational thinking* · ayrıştırma *decomposition* · örüntü tanıma *pattern recognition* · soyutlama *abstraction* · algoritma tasarımı *algorithm design* · test *testing* · test durumu *test case* · sınır durumu *edge case* · izleme tablosu *trace table*

1.5.9 Kendini Dene

1. İkili sayıya 1 ekleyen Turing makinesi bantta 1111 (ikili 15) ile çalıştırılıyor; başlık en sağdaki basamakta. Makine kaç adımda durur, hangi kurallar kullanılır, bantta ne kalır?

   Cevap

   Dört basamağın hepsi 1 olduğu için kural 1 dört kez uygulanır; her seferinde 0 yazılır ve başlık sola kayar. Bant 0000 olur ve başlık sayının solundaki boş hücreye gelir. Beşinci adımda □ okunur; kural 3 ile 1 yazılır ve makine durur. Bantta 10000 kalır: *16 = 15 + 1*. Adım sayısı *4 + 1 = 5*'tir; kural 2 hiç kullanılmaz.

2. Eşlik denetimi yapan makine bantta 10010 ile çalıştırılıyor. Her simgeden sonra durum ne olur,

makine sonunda ne yazar?

Cevap

Durum “çift” ile başlar. 1 okunur: tek. 0: tek. 0: tek. 1: çift. 0: çift. Altıncı adımda boş hücre okunur; durum “çift” olduğu için Ç yazılır ve makine durur. Bant 10010Ç olur. Dizide iki tane 1 vardır ve iki çifttir.

3. GCD(105, 75) Öklid algoritmasıyla hesaplanıyor. Her turda a, b ve r ne olur? Kaç tur sürer, sonuç nedir?

Cevap

1. tur: *105 = 1 × 75 + 30*, r = 30; a = 75, b = 30. 2. tur: *75 = 2 × 30 + 15*, r = 15; a = 30, b = 15. 3. tur: *30 = 2 × 15 + 0*, r = 0; a = 15, b = 0. Döngü biter; üç turda 15 döndürülür. Denetim: *105 = 7 × 15*, *75 = 5 × 15*; 7 ile 5'in 1'den büyük ortak böleni yoktur.

4. Karekök algoritması x = 9 ve tahmin = 1 ile çalıştırılıyor. Her turdan sonra tahmin ve hata ne olur? Algoritma kaç turda durur?

Cevap

Başlangıçta hata *|1 − 9| = 8*'dir. 1. tur: *(1 + 9) ÷ 2 = 5*, hata *|25 − 9| = 16*. 2. tur: *(5 + 1,8) ÷ 2 = 3,4*, hata *|11,56 − 9| = 2,56*. 3. tur: *(3,4 + 2,6471) ÷ 2 ≈ 3,0235*, hata ≈ 0,1417. 4. tur: *(3,0235 + 2,9767) ÷ 2 ≈ 3,0001*, hata ≈ 0,0005. Hata 0,01'in altına indiği için algoritma dört turda durur ve yaklaşık 3,0001 yazar.

5. Aşağıdaki bilgilerin her biri bildirimsel mi, buyurgan mıdır? (a) Su deniz seviyesinde 100 °C'de kaynar. (b) Bir sayının 3'e bölünüp bölünmediğini anlamak için rakamları toplanır; toplam 3'e bölünüyorsa sayı da bölünür. (c) Bir dikdörtgenin alanı, kenar uzunluklarının çarpımıdır.

Cevap

(a) bildirimseldir; bir olgu bildirir. (b) buyurgandır; bir sonuca ulaşmak için izlenecek adımları söyler: rakamları topla, toplamı denetle. (c) bildirimseldir; alanın ne olduğunu tanımlar. Alan hesaplamak için bu tanım kullanılabilir ama cümle bir yöntem tarif etmez, bir eşitlik bildirir.

6. Silindirindeki çıkıntılarla tek bir şarkı çalan mekanik bir müzik kutusu ile internetten indirilen

şarkıları çalan bir akıllı hoparlör, program açısından nasıl ayrılır?

Cevap

Müzik kutusu sabit programlıdır: çalacağı şarkı silindirdeki çıkıntıların fiziksel düzenidir; başka bir şarkı için silindiri değiştirmek gerekir. Akıllı hoparlör depolanan programlıdır: içindeki işlemci bellekte saklanan bir yazılımı çalıştırır; yazılım güncellemesiyle yeni özellikler kazanabilir. Akıllı hoparlörün kendisi yine de belirli işler için tasarlanmıştır ve kullanıcı ona kendi programını yükleyemez; bu yüzden genel amaçlı bir bilgisayar sayılmaz.

7. Aşağıdaki durumların her biri bilgisayarların hangi sınırına örnektir? (a) Bir öğrenci, her programın sonsuz döngüye girip girmeyeceğini önceden söyleyen bir araç yazmaya çalışıyor. (b) Saniyede 10^9 deneme yapan bir program, 50 bitlik bir anahtarın bütün olasılıklarını en kötü durumda yaklaşık 13 günde deneyebiliyor. (c) Bir anket formunda yaş alanına 250 yazılmış ve yaş ortalaması anlamsız çıkmış.

Cevap

(a) hesaplanabilirlik sınırıdır; bu durma problemidir ve böyle genel bir araç yazılamaz. (b) zaman sınırıdır: $2^{50} \div 10^9 \approx 1{,}13 \times 10^6$ saniye, yaklaşık 13 gün eder; daha hızlı donanım süreyi kısaltır ama anahtara eklenen her bit süreyi yeniden iki katına çıkarır. (c) veri kalitesi sınırıdır; algoritma doğru çalışsa bile hatalı girdi yanlış sonuç verir.

8. Bir metindeki kelime sayısını sayan bir algoritma test edilecek. Normal bir cümlenin yanında hangi sınır ve geçersiz durumlar denenmelidir? En az dört durum ve her biri için beklenen sonuç nedir?

Cevap

Örnek bir test kümesi: boş metin → 0; tek kelime (“Merhaba”) → 1; kelimeler arasında birden fazla boşluk (“bir␣␣iki”) → 2; metnin başında ve sonunda boşluk (“␣bir iki␣”) → 2; yalnızca boşluklardan oluşan metin → 0. Algoritma her boşlukta sayacı artırıyorsa son üç durumda yanlış sonuç verir. Temel neden, kelimeler yerine boşlukları saymaktır; doğru düzeltme, bir kelimenin başladığı yerleri saymaktır.

1.5.10 İleri okuma

- J. M. Wing, “Computational Thinking”, *Communications of the ACM*, 49(3), 2006: hesaplamalı düşünmenin neden herkes için temel bir beceri olduğunu üç sayfada anlatan, kavramı yaygınlaştıran kısa makale.
- J. V. Guttag, *Introduction to Computation and Programming Using Python* (3. baskı, MIT Press, 2021), Bölüm 1: bildirimsel ve buyurgan bilgi ayrımı, karekök bulma yöntemi ve sabit/depolanan programlı makineler bu nottakine çok yakın bir dille anlatılır.

- C. Petzold, *The Annotated Turing* (Wiley, 2008): Turing'in 1936 makalesini satır satır açıklayarak Turing makinesini, evrensel makineyi ve hesaplanabilirliği matematik ön bilgisi gerektirmeden kurar.
- M. Sipser, *Introduction to the Theory of Computation* (3. baskı, Cengage, 2012), Bölüm 3–4: Turing makinelerinin ve durma probleminin kesin matematiksel anlatımı; konuya ileri düzeyde devam etmek isteyenler için.

1.5.11 Kaynaklar

1. J. M. Wing, “Computational Thinking”, *Communications of the ACM*, 2006.
2. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, MIT Press.
3. A. B. Downey, C. Mayfield, *Think Java*, Green Tea Press.
4. M. Sipser, *Introduction to the Theory of Computation*, Cengage.

KISIM I · BÖLÜM 2

İşletim Sistemleri

İşletim sistemi donanım kaynaklarını yönetir ve uygulamalara hizmet sunar. İlk sunum yazılım türleri ve lisans modelleriyle başlar; önyükleme, çekirdek ve sistem çağrıları, süreç ve iş parçacığı, zaman paylaşımı, bellek yönetimi ve sanal bellek, dosya sistemleri, aygıt sürücüleri ve kullanıcı yetkilerini ele alır. İkinci sunum Windows üzerinde masaüstü, pencere ve dosya yönetimini, mutlak ve göreli dosya yollarını, ayarları, güvenliği ve sistem bakımını uygulamalı olarak gösterir.

BU BÖLÜMÜN NOTLARI

2.1 Yazılım ve İşletim Sistemleri

2.2 Windows İşletim Sistemi

KISIM I · BÖLÜM 2.1

Yazılım ve İşletim Sistemleri

Donanım ancak bir program ona ne yapacağını söylediğinde işe yarar. Bu programların en önemlisi, bilgisayar açılınca çalışmaya başlayan ve kapanana kadar bütün kaynakları yöneten işletim sistemidir. Not önce yazılımın ne olduğunu, hangi türlere ayrıldığını ve donanımla nasıl katman katman konuştuğunu anlatır. Ardından işletim sisteminin görevlerine, bilgisayarın açılış sürecine ve çekirdeğe geçer. Süreç, bellek, aygıt ve dosya yönetimi adım adım izlenen örneklerle ele alınır. Döner sıralı çizelgeleme, sanal adres çevirisi, sayfa yer değiştirme ve dosya okuma hesaplarının her ara adımı gösterilir. Son kesimler kullanıcı arayüzlerini, yetki yönetimini, işletim sistemi türlerini, yazılım lisanslarını, güncellemeleri ve sanallaştırmayı karşılaştırır.

ÖĞRENME ÇIKTILARI

Yazılım türleri
Sistem yazılımını, uygulama yazılımını, programlama araçlarını ve ürün yazılımını örnekleriyle ayırt etmek; yazılımla donanım arasındaki katmanları açıklamak.

İşletim sistemi
İşletim sisteminin kullanıcı, uygulama ve donanım arasındaki rolünü açıklamak; açılış sürecini, çekirdeği ve sistem çağrısını adım adım anlatmak.

Kaynak yönetimi
Süreç durumlarını ve çizelgelemeyi, bellek ayırma ve korumayı, sanal belleği, aygıt sürücülerini ve dosya sistemini yorumlamak; Gantt çizelgesi, adres çevirisi ve sayfa yer değiştirme hesaplarını yapmak.

Sistem seçimi
Kişisel, mobil, sunucu, gömülü ve gerçek zamanlı işletim sistemlerini gereksinimlere göre karşılaştırmak; lisans, güncelleme ve sanallaştırma seçeneklerini değerlendirmek.

2.1.1 Yazılım

Donanım tek başına hiçbir iş yapmaz; işlemciye, belleğe ve diske adım adım ne yapacaklarını söyleyen komutlar gerekir. Bu komutların bütününe yazılım denir.

2.1.1.1 Yazılım nedir?

Günlük dilde yazılım denince çoğunlukla yalnızca program akla gelir. Oysa bir yazılım ürünü programdan fazlasını içerir. Bir oyun, kodun yanında resim, ses ve harita dosyalarıyla gelir. Bir muhasebe programı vergi oranlarını bir tablo dosyasından okur. Her ikisinin de kurulum yönergesi ve kullanım kılavuzu vardır. Bu parçaların biri eksik olunca yazılım ya çalışmaz ya da doğru kullanılamaz.

> **Tanım 2.1.1**
>
> **Yazılım** (*software*), bir bilgisayarın yapacağı işlemleri tanımlayan programların, bu programların kullandığı verilerin ve ilgili belgelerin bütünüdür. **Donanım** (*hardware*) ise bilgisayarın elle tutulabilen fiziksel parçalarıdır: işlemci, bellek, disk, klavye, ekran.

Donanım ile yazılım arasındaki iş bölümü nettir. Donanım işlemi fiziksel olarak yürütür: işlemci toplar, bellek saklar, ekran gösterir. Yazılım ise hangi işlemin, ne zaman ve nasıl yapılacağını belirler. Aynı donanım, üzerine yüklenen yazılıma göre bir gün kelime işlemci, ertesi gün oyun makinesi olur. Bu esneklik bilgisayarı tek amaçlı makinelerden ayıran temel özelliktir.

Bir yazılımın ortaya çıkışı da bir zincir izler: **kullanıcı gereksinimi → algoritma → program kodu → çalışan yazılım → sonuç**. “Dönem ortalaması hesaplansın” gereksinimi, “notları al, ağırlığıyla çarp, topla” algoritmasına dönüşür; algoritma bir programlama dilinde yazılır ve derlenip çalıştırılınca ortalamayı verir. Bir halkadaki hata, örneğin yanlış anlaşılmış bir gereksinim, sonraki bütün halkalara taşınır.

2.1.1.2 Yazılımın kalite özellikleri

Aynı işi yapan iki program arasında seçim yaparken yalnızca “çalışıyor mu?” sorusu yetmez. İki hesap makinesi uygulaması da doğru sonucu verebilir; ama biri her açılışta on saniye bekletir, öteki tuş yerlerini her sürümde değiştirir. Yazılım mühendisliğinde bir yazılımın ne kadar iyi olduğu altı temel özellikle değerlendirilir.

Özellik	Anlamı	Bankacılık uygulamasında örnek
İşlevsellik	Beklenen görevleri doğru yerine getirme	Havale tutarı hesaptan eksiksiz düşer
Kullanılabilirlik	Kolay öğrenilme ve etkili kullanım	Para gönderme üç dokunuşta biter
Performans	Zamanı ve kaynakları verimli kullanma	Hesap özeti bir saniyede açılır
Güvenilirlik	Hata durumunda öngörülebilir davranma	Bağlantı koparsa işlem yarım kalmaz, ya tamamlanır ya iptal olur
Güvenlik	Yetkisiz erişime ve değişikliğe karşı koruma	Başkası şifreyi bilse bile telefona gelen kod olmadan giremez
Bakım kolaylığı	Düzeltme ve geliştirmenin yönetilebilir olması	Yeni bir vergi kuralı birkaç günde eklenir

Tablo 2.1.1: Yazılımın kalite özellikleri ve bir bankacılık uygulamasından örnekler.

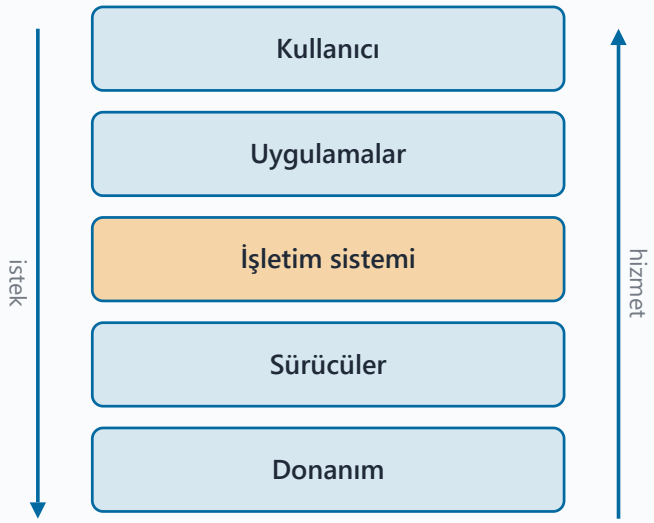
Tablo 2.1.1 bu özellikleri tanıdık bir uygulama üzerinde gösterir. Özellikler bazen çekişir: güvenlik için eklenen ek doğrulama adımı kullanılabilirliği azaltır. İyi tasarım, kullanım amacına göre doğru dengeyi kurar; bir uçak kontrol yazılımında güvenilirlik, bir fotoğraf uygulamasında kullanılabilirlik öndedir.

Sık yapılan hata

Güvenilirlik ile güvenliği aynı şey sanmak. Güvenilirlik, yazılımın hata karşısında tutarlı davranmasıdır: elektrik kesilince veri bozulmaz. Güvenlik ise kötü niyetli birine karşı korumadır: yetkisi olmayan biri veriye erişemez. Çok güvenilir bir program güvensiz olabilir; tersi de mümkündür.

2.1.1.3 Yazılım ile donanım arasındaki katmanlar

Bir kullanıcı kelime işlemcide "Yazdır" düğmesine bastığında yazıcının mürekkep püskürtme kafasını hangi gerilimle süreceğini düşünmez. Kelime işlemci de bunu bilmez. Bu ayrıntılar alt katmanlarda saklıdır. Bilgisayar sistemi üst üste dizilmiş katmanlar biçiminde düzenlenir ve her katman yalnızca hemen altındakiyle konuşur.



Şekil 2.1.1: Kullanıcıdan donanıma beş katman. İstekler yukarıdan aşağı iner; sonuçlar, yani alt katmanın sunduğu hizmetler, aşağıdan yukarı döner.

Şekil 2.1.1 beş katmanı gösterir. En üstte **kullanıcı** vardır ve yalnızca uygulamalarla etkileşir. **Uygulamalar** işlerini işletim sisteminden ister. **İşletim sistemi** bu istekleri **sürücülere** iletir. Sürücüler de her donanım parçasına özgü komutları üretir. En altta, işi fiziksel olarak yapan **donanım** durur.

Tanım 2.1.2

Soyutlama (*abstraction*), bir katmanın altındaki ayrıntıları gizleyerek üstündeki katmana daha basit ve ortak bir hizmet sunmasıdır. Üst katman hizmeti nasıl kullanacağını bilir, hizmetin nasıl yapıldığını bilmek zorunda değildir.

Soyutlamanın en büyük kazancı taşınabilirliktir. Kelime işlemci yalnızca "bu sayfayı yazdır" der. Yazıcı ister mürekkep püskürtmeli ister lazer olsun, istek aynıdır. Farkı sürücü katmanı karşılar: her yazıcı üreticisi kendi aygıtı için bir sürücü yazar. Bu sayede aynı uygulama, sürücüler sayesinde binlerce farklı donanım bileşiminde çalışır. Uygulama yazan bir programcının her yazıcı modelini tanıması gerekmez.

2.1.1.4 Yazılım türleri

Bilgisayarda çalışan yazılımlar görevlerine göre dört ana gruba ayrılır. Ayrımın ölçütü, yazılımın kime hizmet ettiğidir: donanıma ve öteki programlara mı, kullanıcının işine mi, yazılım geliştirene mi, yoksa tek bir cihazın kendisine mi?

Tanım 2.1.3

Sistem yazılımı (*system software*) donanımı yönetir ve öteki yazılımlara çalışma ortamı sağlar. **Uygulama yazılımı** (*application software*) kullanıcının belirli bir işini yapar. **Programlama araçları** kod yazmayı, kodu çevirmeyi, sınamayı ve hata ayıklamayı destekler. **Ürün yazılımı** (*firmware*) bir cihazın kalıcı belleğine gömülü olan ve cihazın temel işlevlerini yöneten yazılımdır.



Şekil 2.1.2: Yazılım türleri ve örnekleri: sistem yazılımı (işletim sistemi, sürücü), uygulama yazılımı (tarayıcı, kelime işlemci), programlama araçları (derleyici, IDE) ve ürün yazılımı (UEFI, yönlendirici yazılımı).

Şekil 2.1.2 dört türü örnekleriyle bir ağaçta toplar. **Sistem yazılımının** en bilinen örneği işletim sistemidir; aygıt sürücüleri ve disk temizleme, yedekleme gibi **yardımcı araçlar** (*utility*) da bu gruba girer. **Uygulama yazılımı** kullanıcının asıl amacına hizmet eder: tarayıcıyla web sayfası açılır, kelime işlemciyle rapor yazılır.

Programlama araçları yeni yazılım üretmek için kullanılır. **Derleyici** (*compiler*) insanın yazdığı kaynak kodu işlemcinin çalıştırabileceği makine koduna çevirir. **Tümleşik geliştirme ortamı** (*integrated development environment, IDE*) kod düzenleyiciyi, derleyiciyi ve hata ayıklayıcıyı tek bir pencerede birleştirir.

Ürün yazılımı donanımla yazılımın kesiştiği yerde durur. Anakarttaki UEFI, bir yönlendiricinin menüsü, bir çamaşır makinesinin program düğmelerini yöneten kod bu türdendir. Ürün yazılımı diskte değil, cihazın kendi kalıcı belleğinde (ROM ya da flaş bellek) saklanır. Cihaz açılınca hiçbir şey yüklenmeden çalışmaya başlar. Kullanıcı onu çoğu zaman fark etmez; yalnızca üretici bir güncelleme yayımlayınca ondan haberdar olur.

Örnek 2.1.1

Bir bilgisayardaki şu yazılımlar hangi türe girer? Yazıcı sürücüsü, Java derleyicisi, hesap tablosu programı, akıllı saatin içindeki yazılım, disk birleştirme aracı.

1. **Yazıcı sürücüsü:** işletim sistemiyle birlikte donanımı yönetir → sistem yazılımı.
2. **Java derleyicisi:** kaynak kodu çevirerek yeni yazılım üretilmesini sağlar → programlama aracı.
3. **Hesap tablosu programı:** kullanıcının hesap işini yapar → uygulama yazılımı.
4. **Akıllı saatin yazılımı:** cihazın kalıcı belleğine gömülüdür ve onun temel işlevlerini yönetir → ürün yazılımı.
5. **Disk birleştirme aracı:** kullanıcının bir belgesiyle değil, diskin düzeniyle uğraşır → sistem yazılımı (yardımcı araç).

2.1.1.5 Sistem ve uygulama yazılımları

Bu dört türün yanında sık geçen bir tür daha vardır: **ara yazılım** (*middleware*). Ara yazılım, uygulamaların işletim sistemiyle ve birbiriyle haberleşmesini kolaylaştırır. Örneğin bir okulun not sistemi, kayıtları tuttuğu veri tabanına bir bağlantı katmanı üzerinden ulaşır; programcı ağ bağlantısının ayrıntılarını yazmak zorunda kalmaz.

Tür	Örnekler	Temel amaç
Sistem	İşletim sistemi, sürücü, yardımcı araç	Bilgisayarı ve kaynakları yönetmek
Uygulama	Kelime işlemci, tarayıcı, oyun	Kullanıcının görevini yapmak
Ara yazılım	Veri tabanı bağlantı katmanı, mesaj kuyruğu	Uygulamaların sistemle ve birbiriyle haberleşmesini kolaylaştırmak
Programlama araçları	Derleyici, tümleşik geliştirme ortamı (IDE), hata ayıklayıcı	Yeni yazılım üretmek

Tablo 2.1.2: Yazılım türlerinin örnekleri ve temel amaçları.

Tablo 2.1.2 ayrımı özetler. Sınırlar her zaman keskin değildir. Bir tarayıcı, web uygulamalarına çalışma ortamı sağladığı için kısmen bir platform gibi davranır. Yine de temel soru işe yarar: yazılım kullanıcının işini mi yapıyor, yoksa başka yazılımların çalışmasını mı sağlıyor? İlk durumda uygulama, ikincisinde sistem yazılımıdır.

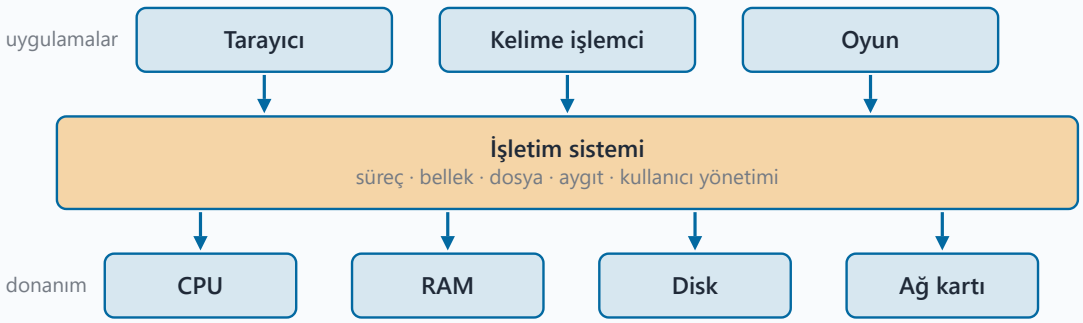
2.1.2 İşletim sistemi

Bir bilgisayarda aynı anda onlarca program çalışır ve hepsi aynı işlemciyi, belleği ve diski kullanmak ister. Bu kalabalığı düzene sokan yazılım işletim sistemidir.

2.1.2.1 İşletim sistemi nedir?

Tanım 2.1.4

İşletim sistemi (*operating system, OS*), donanım kaynaklarını yöneten ve uygulamalara ortak hizmetler sunan temel sistem yazılımıdır. Bilgisayar açılınca ilk yüklenen büyük yazılımdır ve kapanana kadar çalışır.



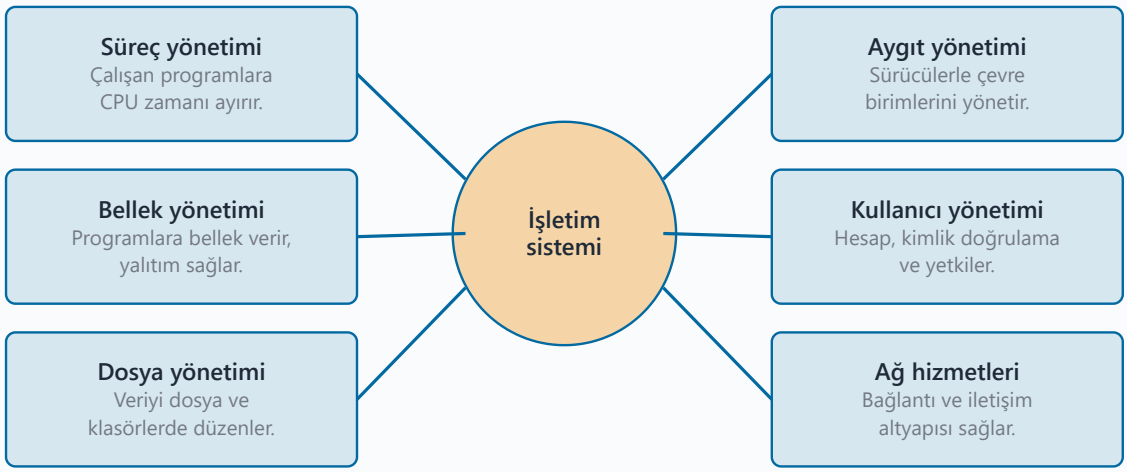
Şekil 2.1.3: İşletim sistemi uygulamalar (tarayıcı, kelime işlemci, oyun) ile donanım (CPU, RAM, disk, ağ kartı) arasında durur. Süreç, bellek, dosya, aygıt ve kullanıcı yönetimi bu katmanın işidir.

Şekil 2.1.3 işletim sisteminin konumunu gösterir. Üstteki uygulamalar işletim sistemi olmadan donanıma ulaşamaz. Alttaki donanım da hangi uygulamaya hizmet edeceğini kendisi seçemez. İşletim sistemi bu iki dünyanın arasında üç temel iş yapar.

- **Donanım ayrıntılarını soyutlar.** Bir uygulama “dosyayı aç” der; dosyanın SSD'de mi, USB bellekte mi, ağdaki bir sunucuda mı olduğunu bilmek zorunda değildir.
- **Kaynakları güvenli biçimde paylaştırır.** Birden fazla program aynı işlemciyi ve belleği kullanır. İşletim sistemi her birine sıra ve alan verir; bir programın ötekinin belleğine yazmasını engeller.
- **Kullanıcıya bir arayüz sunar.** Pencereli grafik arayüz ya da metin komutlarıyla çalışan komut satırı, kullanıcının bilgisayarla konuşma yoludur.

2.1.2.2 İşletim sisteminin görevleri

İşletim sisteminin işi altı ana göreve ayrılır. Her görev bir kaynak türünü yönetir.



Şekil 2.1.4: İşletim sisteminin altı görevi: süreç, bellek, dosya, aygıt ve kullanıcı yönetimi ile ağ hizmetleri.

Şekil 2.1.4 bu görevleri bir arada gösterir.

- **Süreç yönetimi:** çalışan programlara işlemci zamanı ayırır. Hangi programın ne zaman ve ne kadar süre işlemciyi kullanacağına karar verir.
- **Bellek yönetimi:** programlara bellek verir ve onları birbirinden yalıtır. Bir programın çökmesi ötekinin verisini bozmaz.
- **Dosya yönetimi:** veriyi dosya ve klasörlerde düzenler; dosyaların diskte nerede durduğunu ve kimin okuyabileceğini izler.
- **Aygıt yönetimi:** klavye, yazıcı, ekran kartı gibi çevre birimlerini sürücüler aracılığıyla yönetir.
- **Kullanıcı yönetimi:** hesapları, kimlik doğrulamayı ve yetkileri yönetir.
- **Ağ hizmetleri:** bağlantı kurmak ve veri alışverişi yapmak için gereken altyapıyı sağlar.

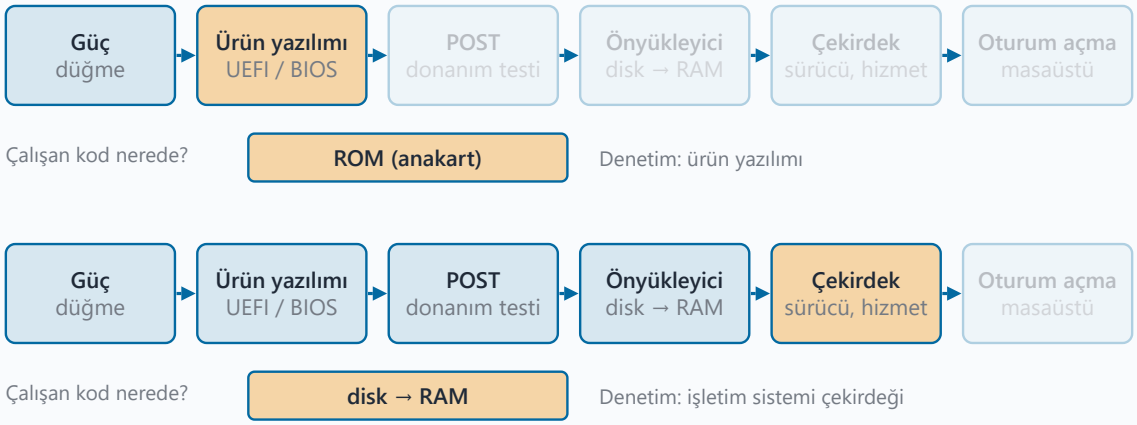
2.1.2.3 Önyükleme: bilgisayar nasıl açılır?

İşletim sistemi diskte durur, ama onu diskten belleğe kim yükler? Bilgisayar kapalıyken RAM boştur. Güç düğmesine basıldığı anda çalışacak hiçbir program bellekte yoktur. Bu "tavuk mu yumurtadan, yumurta mı tavuktan" sorununu çözen sürece önyükleme denir.

Tanım 2.1.5

Önyükleme (*boot, booting*), güç verildiği andan işletim sisteminin kullanıma hazır olduğu ana kadar geçen ve her aşamanın bir sonrakini yüklediği açılış sürecidir. İşletim sistemini diskten RAM'e taşıyan küçük programa **önyükleyici** (*boot loader*) denir.

Sorunun çözümü, kapalıyken de içeriğini koruyan küçük bir bellektir. Anakart üzerindeki kalıcı bellekte (ROM ya da flaş bellek), üreticinin yazdığı bir ürün yazılımı bulunur. İşlemci güç alınca her zaman bu belleğin belirli bir adresinden çalışmaya başlar. Açılış altı aşamada ilerler.



Şekil 2.1.5: Önyükleme aşamaları. (a) Açılışın başında denetim ürün yazılımındadır ve çalışan kod anakarttaki ROM'dadır. (b) Önyükleyici çekirdeği diskten RAM'e yükleyip denetimi ona vermiştir; artık çalışan kod RAM'deki işletim sistemi çekirdeğidir.

Aşama	Ne olur?	Çalışan kod nerede?	Denetim kimde?
1. Güç	Güç kaynağı gerilimler kararlı olunca anakarta "güç hazır" sinyali verir. RAM boştur.	—	Kullanıcı
2. Ürün yazılımı	İşlemci anakarttaki UEFI'yi (eski adıyla BIOS) çalıştırmaya başlar.	ROM (anakart)	Ürün yazılımı
3. POST	Açılışta öz sınama: bellek, klavye, disk gibi temel donanım denetlenir.	ROM (anakart)	Ürün yazılımı
4. Önyükleyici	Önyükleme sırasındaki ilk uygun aygıttan önyükleyici RAM'e yüklenir.	Disk → RAM	Önyükleyici
5. Çekirdek	Çekirdek RAM'e yüklenir; sürücüleri, ilk süreçleri ve sistem hizmetlerini başlatır.	Disk → RAM	İşletim sistemi çekirdeği
6. Oturum açma	Kimlik doğrulanınca masaüstü ve başlangıç uygulamaları açılır.	RAM	İşletim sistemi

Tablo 2.1.3: Önyükleme aşamalarında çalışan kodun yeri ve denetimin sahibi.

Tablo 2.1.3 altı aşamayı iki soruya göre izler: o anda çalışan kod neredeдir ve denetim kimdedir? Şekil 2.1.5 bu yolculuğun iki ara anını gösterir.

Güç ve ürün yazılımı. Güç düğmesi, güç kaynağına çalışma sinyali verir. Güç kaynağı gerilimleri kararlı hâle gelince anakarta "güç hazır" sinyali gönderir. Sinyal gelince işlemci, anakarttaki kalıcı bellekte duran ürün yazılımını çalıştırır. Günümüz bilgisayarlarında bu yazılım **UEFI**'dir (*Unified*

Extensible Firmware Interface, birleşik genişletilebilir ürün yazılımı arayüzü). Daha eski bilgisayarlarda aynı görevi **BIOS** (*Basic Input/Output System*, temel giriş/çıkış sistemi) görürdü.

POST. Ürün yazılımının ilk işi **açılışta öz sınamadır** (*power-on self-test, POST*). Bellek modüllerinin takılı ve çalışır olduğu, klavyenin ve disklerin bağlı olduğu denetlenir. Bir sorun varsa bilgisayar bir dizi bip sesi çıkarır ya da ekrana bir ileti yazar. RAM modülü hiç takılı değilse açılış burada durur, çünkü sonraki aşamaların hepsi RAM'e bir şey yüklemeyi gerektirir.

Önyükleyici. Ürün yazılımı ayarlarında bir **önyükleme sırası** (*boot order*) bulunur, örneğin "önce USB, sonra SSD". Ürün yazılımı bu sıradaki aygıtları dener; takılı olmayanı atlar. Uygun aygıtı bulunca diskteki küçük **EFI sistem bölümüne** bakar. FAT32 ile biçimlendirilmiş bu bölümde önyükleyici programı durur. Windows'ta önyükleyicinin adı Windows Boot Manager'dır (dosya adı *bootmgfw.efi*); Linux'ta en yaygın önyükleyici GRUB'dur. Önyükleyici RAM'e yüklenir ve denetim ona geçer.

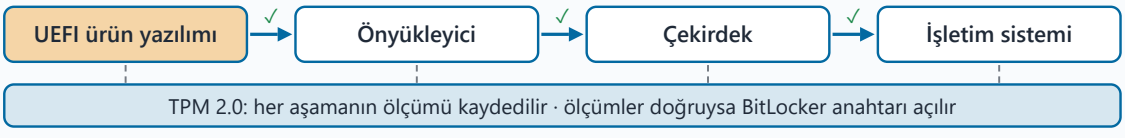
Çekirdek. Önyükleyici işletim sisteminin çekirdeğini bulur, RAM'e yükler ve denetimi ona verir. Windows'ta bu iş iki adımda olur: önyükleyici *winload.efi* programını çalıştırır, o da çekirdek dosyası *ntoskrnl.exe*'yi yükler. Çekirdek çalışmaya başlayınca aygıt sürücülerini yükler, ilk süreçleri ve arka planda çalışan sistem hizmetlerini başlatır.

2.1.2.4 Güvenli Önyükleme, TPM ve disk bölümleme

Önyükleme zincirinin her halkası bir sonrakine güvenir. Ürün yazılımı diskte bulduğu önyükleyiciyi çalıştırır, önyükleyici çekirdeği çalıştırır. Kötü amaçlı bir program önyükleyicinin yerine kendini koyabilirse işletim sisteminden önce çalışır. O noktadan sonra virüs tarayıcıları bile onu göremez, çünkü onlar da ondan sonra yüklenir. Modern bilgisayarlar bu tehlikeye karşı iki araç kullanır.

Tanım 2.1.6

Güvenli Önyükleme (*Secure Boot*), UEFI'nin yalnızca güvenilir bir dijital imzası olan önyükleyiciyi ve çekirdeği çalıştırmasıdır. **TPM** (*Trusted Platform Module*, güvenilir platform modülü), şifreleme anahtarlarını ve açılış ölçümlerini saklayan bir güvenlik yongasıdır. TPM anakartta ayrı bir yonga, işlemcinin içinde bir birim ya da ürün yazılımı içinde bir bileşen olabilir.



Şekil 2.1.6: Güven zinciri. UEFI ürün yazılımı önyükleyicinin, önyükleyici de çekirdeğin dijital imzasını denetler (✓). TPM her aşamanın ölçümünü kaydeder; ölçümler doğruysa disk şifreleme (BitLocker) anahtarı açılır.

Şekil 2.1.6 güven zincirini soldan sağa gösterir. Zincirin ilk halkası ürün yazılımıdır ve üretici onu fabrikada güvenilir anahtarlarla birlikte yazar. Her halka bir sonrakini çalıştırmadan önce onun imzasını denetler. Dijital imza, bir dosyayı belirli bir kuruluşun ürettiğini ve dosyanın sonradan değiştirilmedi-

ğini gösteren matematiksel bir damgadır. Dosyanın tek bir baytı değişse imza tutmaz. Değiştirilmiş bir önyükleyicinin imzası doğrulanamadığı için UEFI onu çalıştırmaz ve açılış durur.

TPM her aşamada yüklenen kodun içeriğinden kısa bir parmak izi hesaplar ve kaydeder. Disk şifreleme aracı BitLocker, şifre çözme anahtarını TPM'de saklar; TPM anahtarı yalnızca ölçümler beklenen değerlerle aynıysa verir. Disk başka bir bilgisayara takılsa ya da açılış zinciri değiştirilse veri okunamaz.

Örnek 2.1.2

Güvenli Önyükleme açık bir bilgisayarda bir zararlı yazılım, diskteki önyükleyici dosyasının birkaç baytını değiştiriyor. Bir sonraki açılışta ne olur?

1. UEFI çalışır ve POST'u tamamlar.
2. UEFI, EFI sistem bölümündeki önyükleyiciyi bulur ve çalıştırmadan önce imzasını denetler.
3. Dosya değiştiği için hesaplanan değer imzadaki değerle tutmaz. İmza doğrulanamaz.
4. UEFI önyükleyiciyi çalıştırmaz ve bir güvenlik ihlali iletisi gösterir. Zararlı kod hiç çalışma fırsatı bulamaz.

Windows 11 kurulumu için UEFI ile Güvenli Önyükleme desteği ve TPM 2.0 gerekir.

Önyükleme ile yakından ilgili bir konu da diskin nasıl bölündüğüdür. Bir disk, her biri ayrı bir sürücü gibi kullanılan **bölümlere** (*partition*) ayrılabilir. Bölümlerin yerini diskin başındaki bir tablo tutar. Bu tablonun biçimine **bölümleme düzeni** (*partitioning scheme*) denir ve iki düzen yaygındır.

Özellik	MBR	GPT
Birlikte kullanıldığı ürün yazılımı	BIOS	UEFI
En büyük disk	2 TB'a kadar	2 TB'ın çok üzeri
Bölüm sayısı	En çok 4 birincil bölüm	Çok sayıda bölüm

Tablo 2.1.4: MBR ve GPT bölümleme düzenlerinin karşılaştırması.

MBR (*Master Boot Record*, ana önyükleme kaydı) eski düzendir. Disk üzerindeki konumları 32 bitlik sektör numaralarıyla tutar. Bir sektör 512 bayt olduğundan adreslenebilecek en büyük alan şöyle hesaplanır:

> $2^{32} \times 512$ bayt = 4.294.967.296 × 512 = 2.199.023.255.552 bayt ≈ 2,2 TB

Bu sayı 2 TiB'a, yani 2×1.024^4 bayta eşittir; “2 TB sınırı” buradan gelir. **GPT** (*GUID Partition Table*, genel benzersiz kimlikli bölüm tablosu) 64 bitlik numaralar kullandığı için bu sınır pratikte ortadan kalkar. Tablo 2.1.4 iki düzeni karşılaştırır.

Örnek 2.1.3

4 TB'lık bir diske Windows 11 kurulacak ve disk sistem, programlar, veriler, yedek ve kurtarma için beş bölüme ayrılacak. Hangi bölümleme düzeni seçilmelidir?

1. **Disk boyutu:** 4 TB, MBR'nin yaklaşık 2,2 TB'lık sınırını aşar. MBR ile diskin yarısına yakını (yaklaşık 1,8 TB) kullanılamaz.
2. **Bölüm sayısı:** beş bölüm, MBR'nin dört birincil bölüm sınırını aşar.
3. **İşletim sistemi:** Windows 11 UEFI ister; UEFI ile birlikte kullanılan düzen GPT'dir.

Üç ölçüt de aynı sonucu verir: GPT + UEFI.

2.1.3 Çekirdek

İşletim sisteminin merkezinde, donanıma doğrudan dokunma yetkisi olan küçük ama kritik bir bölüm durur: çekirdek.

2.1.3.1 Çekirdek ve ayrıcalıklı çalışma

Tanım 2.1.7

Çekirdek (*kernel*), işletim sisteminin ayrıcalıklı çalışan merkezî bölümüdür. İşlemci iki modda çalışabilir: **çekirdek modunda** (*kernel mode*) her komutu yürütebilir ve donanıma doğrudan erişebilir; **kullanıcı modunda** (*user mode*) donanıma erişen komutlar yasaktır. Uygulamalar kullanıcı modunda çalışır ve çekirdekten **sistem çağrısıyla** (*system call*) hizmet ister.



Şekil 2.1.7: Kullanıcı modundaki uygulamalar (metin editörü, tarayıcı) sistem çağrısı arayüzünden geçerek çekirdek moduna ulaşır. Çekirdeğin süreç, bellek, dosya sistemi ve aygıt sürücüsü alt sistemleri donanımı (CPU, RAM, disk) yönetir.

Şekil 2.1.7 iki modu iki kat gibi gösterir. Üst katta uygulamalar vardır ve aralarında bir sınır çizgisi bulunur. Uygulamalar bu çizgiyi yalnızca tek bir kapıdan, sistem çağrısı arayüzünden geçebilir. Alt katta çekirdeğin dört alt sistemi donanımla konuşur.

Bu ayrımın nedeni güvenlik ve kararlılıktır. Her program diske doğrudan yazabilseydi hatalı bir program başka bir programın dosyalarının üstüne yazabilir, kötü niyetli bir program başkalarının parolalarını okuyabilirdi. İşlemci kullanıcı modunda donanıma erişen bir komutla karşılaşınca komutu yürütmez ve denetimi çekirdeğe verir. Çekirdek de o programı çoğunlukla sonlandırır. Böylece bir uygulamanın hatası yalnızca kendisini etkiler.

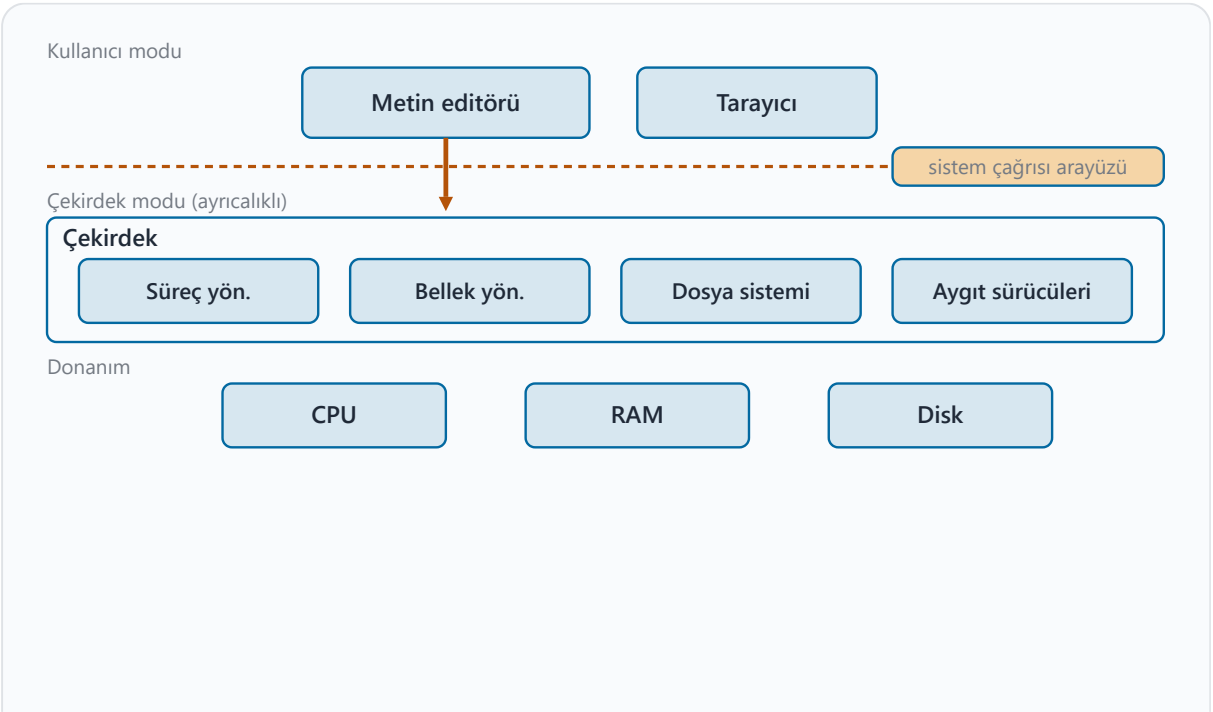
Bedeli de vardır. Çekirdekteki bir hata bütün sistemi etkileyebilir, çünkü çekirdeğin üstünde onu durduracak bir yetki yoktur. Windows'taki mavi hata ekranı ve Linux'taki “kernel panic” iletisi, çekirdek düzeyinde düzeltilemeyen bir hata olduğunu gösterir. Bu yüzden çekirdeğe giren her kod, özellikle aygıt sürücüleri, dikkatle sınanır ve imzalanır.

Sık yapılan hata

“Yönetici hesabıyla açılan program çekirdek modunda çalışır” sanmak. Yönetici hesabı işletim sistemi içindeki *yetkileri* artırır: sistem dosyalarını değiştirmek, program kurmak. Ama yönetici olarak açılan bir program da işlemcinin kullanıcı modunda çalışır ve donanıma yine sistem çağrısı üzerinden ulaşır. Çekirdek modu yalnızca çekirdeğin ve sürücülerin kodu içindir.

2.1.3.2 Sistem çağrısı adım adım

Sistem çağrısının işleyişini somut bir örnekte izlemek, çekirdeğin bütün alt sistemlerini bir arada görmeyi sağlar. Bir metin editörü, diskteki *rapor.txt* dosyasını okumak istiyor.





Şekil 2.1.8: Dosya okuma sistem çağrısının iki anı. (a) İşlemci kullanıcı modundan çekirdek moduna geçmiştir; artık yalnızca çekirdek kodu çalışır. (b) Disk sürücüsü diske okuma komutu vermiştir; disk işi bitince kesmeyle işlemciye haber verir.

Örnek 2.1.4

Metin editörü *rapor.txt* dosyasını okuyor. Her adımda işlemci hangi moddadır ve hangi alt sistem çalışır?

Adım	Ne olur?	İşlemci modu	Çalışan bölüm
1	Editör diske kendisi erişemez; `read` sistem çağrısını yapar.	Kullanıcı	Editör
2	İşlemci çekirdek moduna geçer; denetim çekirdeğe verilir.	Çekirdek	Sistem çağrısı arayüzü
3	Dosyanın diskteki yeri bulunur; kullanıcının okuma izni denetlenir.	Çekirdek	Dosya sistemi
4	Diske okuma komutu verilir; disk işi bitince kesmeyle haber verir.	Çekirdek	Aygıt sürücüsü
5	Okunan veri, RAM'de editörün kendi bellek alanına kopyalanır.	Çekirdek	Bellek yönetimi
6	Çağrı döner; işlemci kullanıcı moduna geçer. Editör veriyi ekranda gösterir.	Kullanıcı	Editör

Editör yalnızca birinci ve altıncı adımda çalışır. Aradaki dört adımın hepsi çekirdek modundadır. Editör donanımı hiç doğrudan görmez.

Şekil 2.1.8 bu adımların ikisini gösterir. Üçüncü adımdaki izin denetimi düzenin güvenlik değerini gösterir: editörün okumaya hakkı olmayan bir dosya istenirse çağrı bir hata koduyla döner ve diske hiç gidilmez.

Dördüncü adımda iki önemli kavram geçer. Disk, işlemciye göre çok yavaştır; bir SSD okuması on binlerce işlemci komutu kadar sürer. İşlemci bu sürede boş beklemez, başka işlere geçer. Disk işini bitirince işlemciye bir sinyal gönderir.

Tanım 2.1.8

Kesme (*interrupt*), bir aygıtın ya da zamanlayıcının işlemciye bir olayı bildiren sinyalidir. Kesme gelince işlemci o anki işini kısa bir süre bırakır, çekirdekteki ilgili kodu çalıştırır ve sonra kaldığı yerden devam eder.

Kesme sayesinde işlemci "disk bitti mi?" diye sürekli sormaz. Tuşa basmak, ağdan paket gelmesi ve zamanlayıcının "zaman dilimi doldu" bildirimi de birer kesmedir.

2.1.3.3 Çekirdek türleri

Çekirdeğin içine neyin konacağı bir tasarım kararıdır. Her şeyi çekirdeğe koymak hızlıdır, çünkü alt sistemler arasında mod geçişi gerekmez. Ama çekirdek büyüdükçe içinde hata olma olasılığı da büyür. Çekirdeği küçük tutmak güvenliği artırır, ama sık mod geçişi hızı düşürür. Bu iki uç arasında üç tasarım yaygındır.

Tür	Fikir	Örnek
Monolitik çekirdek	Süreç, bellek, dosya sistemi ve sürücüler tek ayrıcalıklı çekirdekte; hızlı ama büyük	Linux (modüller yüklenebilir)
Mikro çekirdek	Çekirdekte yalnızca temel işler; sürücü ve dosya sistemi kullanıcı modunda; yalıtım güçlü	MINIX 3, QNX
Hibrit çekirdek	Mikro çekirdek tasarımını alır, hız için bazı hizmetleri çekirdekte tutar	Windows NT çekirdeği, macOS/iOS (XNU)

Tablo 2.1.5: Çekirdek türleri.

Tablo 2.1.5 üç tasarımı karşılaştırır. **Monolitik çekirdekte** (*monolithic kernel*) bütün alt sistemler tek büyük programdır. Linux bu yapıdadır, ama sürücülerin gerektiğinde çekirdeğe eklenip çıkarılabilen *modüller* olarak yüklenmesine izin verir. **Mikro çekirdekte** (*microkernel*) çekirdek yalnızca süreçler arası haberleşme, temel çizelgeleme ve bellek koruması gibi en temel işleri yapar. Sürücüler ve dosya sistemi sıradan programlar gibi kullanıcı modunda çalışır. Bir sürücü çökerse yalnızca o program yeniden başlatılır; sistem ayakta kalır. QNX bu yüzden otomobil ve tıbbi cihazlarda yaygındır. **Hibrit çekirdek** (*hybrid kernel*) ikisinin arasındadır. Windows ve Apple'ın sistemleri bu yolu izler.

Çekirdek ile işletim sistemi arasındaki farkı en iyi Linux ve Android örneği gösterir. **Linux**, tam anlamıyla yalnızca bir çekirdektir. Bir bilgisayara kurulan "Linux" aslında bir **Linux dağıtımıdır** (*Linux distribution*): Linux çekirdeğinin üstüne komut satırı araçları, bir masaüstü ortamı ve uygulamalar eklenmiş bir pakettir. Ubuntu, Fedora ve Debian böyle dağıtımlardır. **Android** de Linux çekirdeğini kullanır, ama üstündeki katmanlar tamamen Android'e özgüdür.

Katman	Görev
Uygulamalar	Kullanıcının çalıştırdığı programlar: mesajlaşma, harita, oyun
Android çatısı ve çalışma zamanı	Uygulamaların kullandığı ortak hizmetler ve uygulama kodunu çalıştıran ortam

Katman	Görev
Linux çekirdeği	Süreç, bellek, sürücü ve güç yönetimi
Donanım	İşlemci, dokunmatik ekran, kamera, sensörler

Tablo 2.1.6: Android yazılım yığını (yukarıdan aşağıya).

Tablo 2.1.6 Android'in katmanlarını sıralar. Bir Linux masaüstü programı Android telefonda çalışmaz, bir Android uygulaması da Ubuntu'da doğrudan açılmaz. İkisi aynı çekirdeği paylaşsa da üst katmanları farklıdır.

2.1.4 Süreç yönetimi

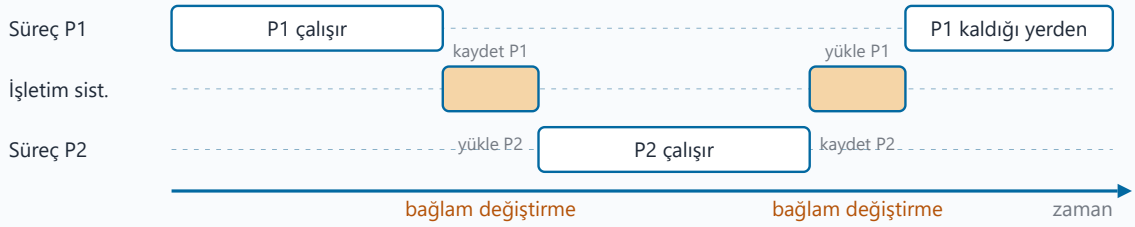
Tek çekirdekli bir işlemci belirli bir anda yalnızca tek bir programın komutunu yürütür. Yine de böyle bir bilgisayarda müzik çalarken yazı yazmak mümkündür, çünkü işletim sistemi işlemciyi programlar arasında çok hızlı paylaştırır.

2.1.4.1 Süreç yönetiminin kavramları

Süreç yönetimi altı kavram üzerine kurulur. İlk üçü yönetilen şeyleri, son üçü yönetimin yollarını anlatır.

Tanım 2.1.9

Program, diskte saklanan komut ve veri bütünüdür. **Süreç** (*process*), çalışmakta olan bir program örneğidir. **İş parçacığı** (*thread*), bir süreç içindeki tek bir yürütme akışıdır. **Çizelgeleme** (*scheduling*), işlemcinin hangi sürece ne zaman verileceğine karar verilmesidir; bu kararı veren çekirdek parçasına **çizelgeleyici** (*scheduler*) denir. **Bağlam değiştirme** (*context switch*), işlemcinin durumunu bir süreçten kaydedip başka bir sürecin durumunu yüklemektir. **Eş zamanlılık** (*concurrency*), birden fazla işin aynı zaman aralığında ilerlemesinin düzenlenmesidir.



Şekil 2.1.9: Tek CPU'da bağlam değiştirme. P1 çalışır; işletim sistemi P1'in durumunu kaydeder ve P2'nin durumunu yükler; P2 çalışır; yeniden geçişle P1 kaldığı yerden devam eder.

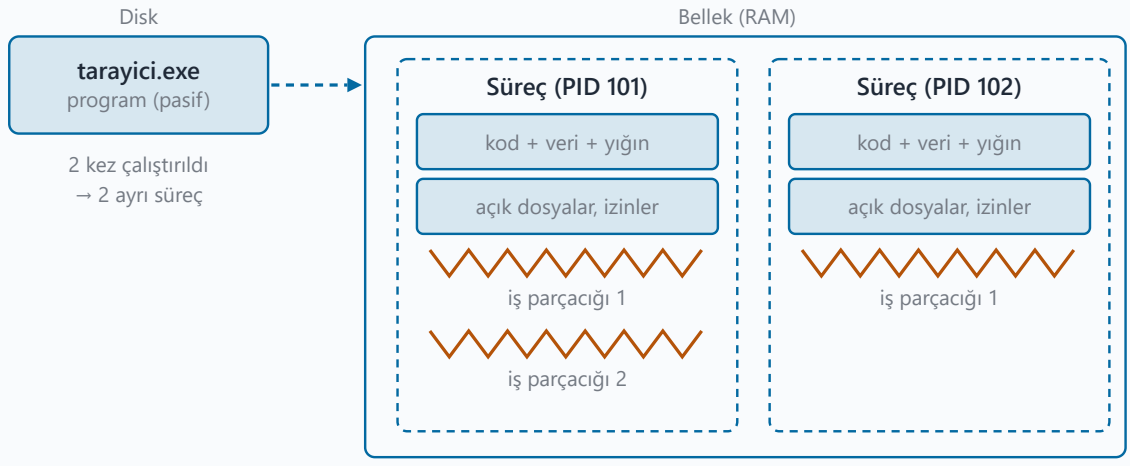
Şekil 2.1.9 bağlam değiştirmeyi zaman ekseninde gösterir. Bir sürecin "bağlamı", işlemcinin o süreç için tuttuğu bütün bilgidir: yazmaçların değerleri, sıradaki komutun adresi, sürecin bellek tablosunun yeri. P1'den P2'ye geçerken işletim sistemi P1'in bağlamını bellekte bu sürece ayrılmış bir kayda yazar. Ardından P2'nin daha önce kaydedilmiş bağlamını işlemciye yükler. P2 sanki hiç durmamış gibi kaldığı komuttan devam eder.

Bağlam değiştirme süresince işlemci hiçbir kullanıcı programını ilerletmez; yalnızca kayıt tutar. Tek bir geçiş birkaç mikrosaniye sürer, ama saniyede binlerce kez yapılınca kayıp hissedilir hâle gelir.

> **Sık yapılan hata**
>
> Eş zamanlılığı "aynı anda çalışma" sanmak. Tek çekirdekli bir işlemcide iki süreç hiçbir zaman aynı anda çalışmaz; yalnızca sırayla, çok kısa aralıklarla çalışır ve dışarıdan aynı anda çalışıyor gibi görünür. Gerçekten aynı anda çalışma, ancak birden fazla çekirdek varsa mümkündür ve buna *paralellik* denir.

2.1.4.2 Program, süreç ve iş parçacığı

Program ile süreç arasındaki fark, yemek tarifi ile tencerede pişen yemek arasındaki farka benzer. Aynı tariften iki ayrı tencerede yemek pişebilir; her tencerenin kendi malzemesi ve aşaması vardır. Program diskte duran pasif bir dosyadır; süreç onun bellekte çalışan bir örneğidir.



Şekil 2.1.10: Diskteki *tarayici.exe* programı iki kez çalıştırılınca bellekte iki ayrı süreç oluşur (PID 101 ve PID 102). Her sürecin kendi kodu, verisi, yığını, açık dosyaları ve izinleri vardır. Birinci sürecin iki, ikincisinin bir iş parçacığı vardır.

Şekil 2.1.10 bu ilişkiyi gösterir. İşletim sistemi her süreci bir **süreç kimliğiyle** (*process ID, PID*) tanır. PID, süreç yaşadığı sürece değişmeyen bir numaradır. Görev Yöneticisi'nde ya da Linux'taki `ps` komutunun çıktısında her satır bir süreçtir ve yanında PID'si yazar.

Her sürecin **ayrı bir bellek alanı** vardır. Bu alanda programın kodu, değişkenlerin tutulduğu veri bölgesi ve fonksiyon çağrılarının izlendiği yığın bulunur. Süreç ayrıca açtığı dosyaların ve izinlerinin bir listesini taşır. İki süreç birbirinin belleğini göremez. Bir tarayıcı penceresi çökünce öteki pencere çoğu zaman çalışmaya devam eder; modern tarayıcılar bu nedenle her sekmeyi ayrı bir süreçte açar.

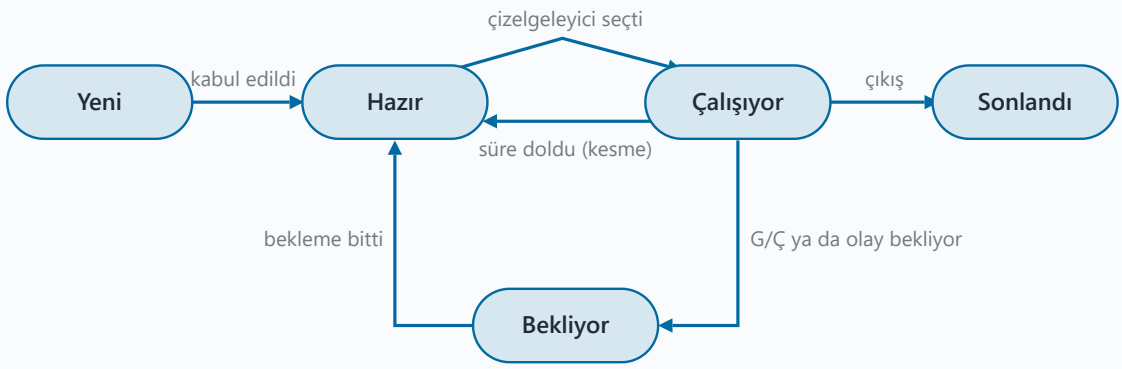
İş parçacıkları ise aynı süreç içinde, o sürecin belleğini *paylaşarak* çalışan yürütme akışlarıdır. Bir kelime işlemci bir iş parçacığıyla klavyeden gelen harfleri işlerken öteki iş parçacığıyla arka planda yazım denetimi yapabilir. İkisi aynı belgeyi, yani aynı bellek alanını görür. Paylaşım haberleşmeyi kolaylaştırır, ama dikkat ister: iki iş parçacığı aynı veriyi aynı anda değiştirmeye kalkarsa veri bozulabilir.

2.1.4.3 Süreç durumları

Bir süreç oluşturulduğu andan bitene kadar hep çalışmaz. Kimi zaman sırasını bekler, kimi zaman diskten veri gelmesini bekler. İşletim sistemi her süreç için "şu anda hangi durumdasın?" bilgisini tutar.

Tanım 2.1.10

Bir sürecin **durumu** (*process state*) şu beşinden biridir. **Yeni**: süreç oluşturuluyor. **Hazır**: çalışmaya hazır, işlemci sırası bekliyor. **Çalışıyor**: komutları şu anda işlemcide yürütülüyor. **Bekliyor**: bir olayı, örneğin disk okumasının bitmesini ya da bir tuşa basılmasını bekliyor. **Sonlandı**: işini bitirdi. Hazır süreçlerin sıralandığı listeye **hazır kuyruğu** (*ready queue*) denir.



Şekil 2.1.11: Süreç durum diyagramı. Oklar izin verilen geçişleri ve her geçişin nedenini gösterir: kabul edildi, çizelgeleyici seçti, süre doldu (kesme), G/Ç ya da olay bekliyor, bekleme bitti, çıkış.

Şekil 2.1.11 beş durumu ve aralarındaki altı geçişi gösterir. Diyagramda olmayan geçişler de en az olanlar kadar öğreticidir.

- **Yeni → Hazır** (kabul edildi): işletim sistemi sürece bellek ayırır ve onu hazır kuyruğuna koyar.
- **Hazır → Çalışıyor** (çizelgeleyici seçti): süreç işlemciyi alır.

- **Çalışıyor → Hazır** (süre doldu): sürecin işlemci payı biter; bir zamanlayıcı kesmesi denetimi çekirdeğe verir ve süreç kuyruğun sonuna döner.
- **Çalışıyor → Bekliyor** (G/Ç ya da olay bekliyor): süreç diskten veri, klavyeden tuş ya da ağdan paket ister.
- **Bekliyor → Hazır** (bekleme bitti): beklenen olay gerçekleşir ve süreç yeniden kuyruğa girer.
- **Çalışıyor → Sonlandı** (çıkış): süreç işini bitirir.

Burada G/Ç, **giriş/çıkış** (*input/output, I/O*) işlemlerinin kısaltmasıdır. Diyagramda **Bekliyor → Çalışıyor** oku yoktur. Beklemesi biten süreç işlemciyi doğrudan almaz; önce hazır kuyruğuna girer ve sırasını bekler. **Hazır → Bekliyor** oku da yoktur: bir süreç ancak çalışırken bir şey isteyebilir, dolayısıyla ancak çalışırken beklemeye geçebilir.

Tek çekirdekli bir sistemde belirli bir anda en çok bir süreç Çalışıyor durumundadır. Öteki hazır süreçler kuyrukta bekler. Bekliyor durumundaki bir süreç ise kuyrukta bile değildir; çizelgeleyici onu hiç seçmez, çünkü işlemciyi alsa bile yapabileceği bir şey yoktur.

Sık yapılan hata

Hazır ile Bekliyor durumlarını karıştırmak. İkisi de "çalışmıyor"dur, ama nedenleri farklıdır. Hazır süreç işlemci verilse hemen çalışabilir; eksiği yalnızca işlemcidir. Bekliyor durumundaki süreç ise işlemci verilse bile ilerleyemez, çünkü bir dış olayı bekler.

2.1.4.4 Durum geçişleri adım adım

Durum diyagramını tek bir sürecin yaşamı boyunca izlemek, geçişlerin hangi olaylarla tetiklendiğini somutlaştırır.

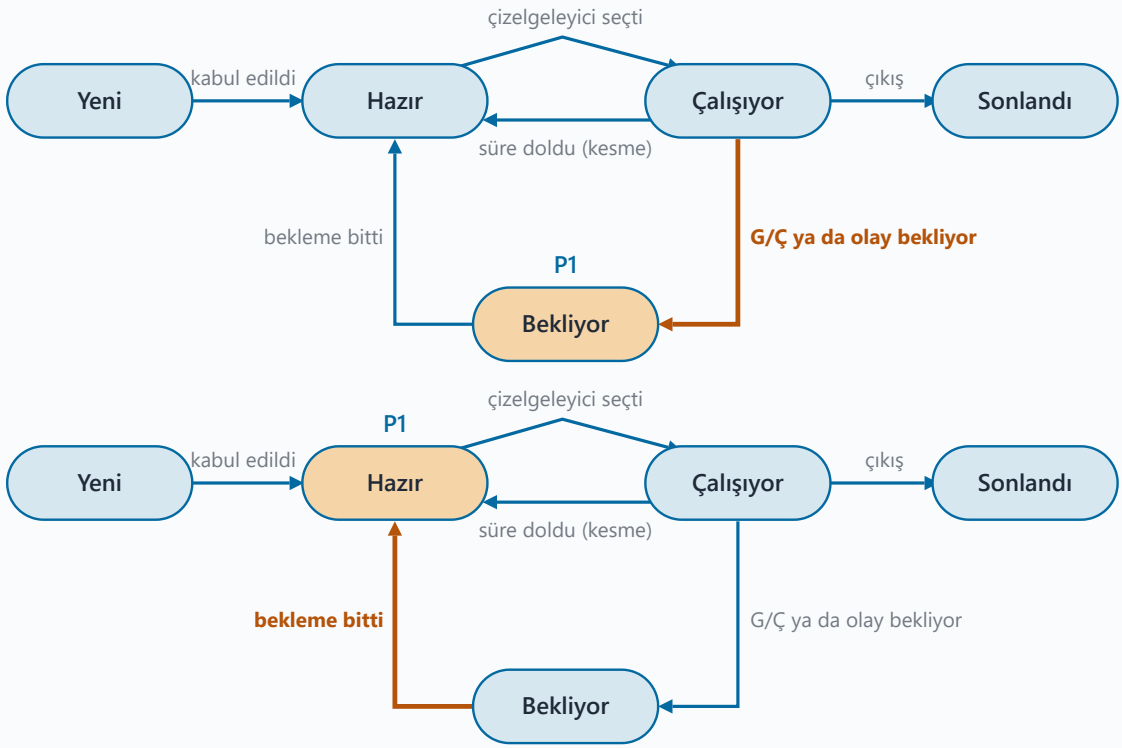
Örnek 2.1.5

Kullanıcı bir programa çift tıklıyor ve P1 süreci oluşuyor. P1 bir süre hesap yapıyor, sonra diskten veri okuyor, sonra yine hesap yapıyor, bu arada zaman dilimi doluyor ve sonunda işini bitiriyor.

Adım	Olay	Geçiş	P1'in durumu
1	Kullanıcı programa çift tıklar	—	Yeni
2	İşletim sistemi süreci kabul eder, bellek ayırır	Yeni → Hazır	Hazır
3	Çizelgeleyici P1'i seçer, bağlam yüklenir	Hazır → Çalışıyor	Çalışıyor
4	P1 diskten veri ister	Çalışıyor → Bekliyor	Bekliyor
5	Disk işini bitirir, kesme gelir	Bekliyor → Hazır	Hazır
6	Sırası gelir, yeniden seçilir	Hazır → Çalışıyor	Çalışıyor
7	Zaman dilimi dolar, zamanlayıcı kesmesi	Çalışıyor → Hazır	Hazır

Adım	Olay	Geçiş	P1'in durumu
8	Tekrar seçilir	Hazır → Çalışıyor	Çalışıyor
9	İşini bitirip çıkar; kaynakları geri alınır	Çalışıyor → Sonlandı	Sonlandı

P1 dokuz adımda üç kez Çalışıyor durumuna girer. Her girişten önce mutlaka Hazır durumundan geçer.



Şekil 2.1.12: Örnek 2.1.5'in iki adımı. (a) Adım 4: P1 diskten veri istediği için Çalışıyor'dan Bekliyor'a geçer. (b) Adım 5: disk işi bitince P1 hemen çalışmaz, önce Hazır durumuna döner.

Şekil 2.1.12 tablonun en çok karıştırılan iki adımını gösterir. Dördüncü adımda P1 işlemciyi kendi isteğiyle bırakır, çünkü veri gelene kadar yapacak işi yoktur. İşlemciyi boşuna tutsaydı başka süreçler de bekletilmiş olurdu. Beşinci adımda disk kesmesi gelir, ama o anda işlemcide başka bir süreç çalışıyor olabilir. P1 bu yüzden kuyruğa döner. Yedinci adımdaki geçiş ise P1'in isteğiyle olmaz: zamanlayıcı kesmesi P1'i işlemciden zorla alır. Bu ayrım, işlemciyi süreçten zorla alabilen sistemlerin temelidir.

2.1.4.5 İki süreç, tek CPU

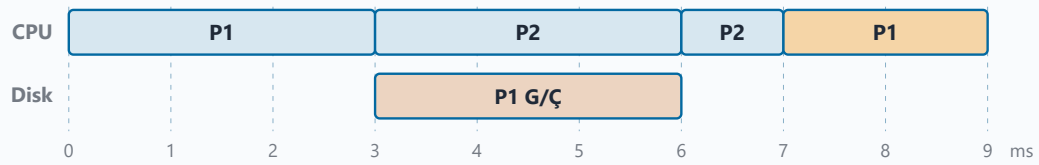
Bir sürecin Bekliyor durumuna geçmesi, işlemcinin boşa çıkması demektir. İşletim sistemi bu boşluğu başka bir süreçle doldurursa toplam iş daha kısa sürede biter. Bu fikre **çoklu programlama** (*multiprogramming*) denir. Etkisini bir zaman çizelgesinde ölçmek mümkündür.

Örnek 2.1.6

Tek çekirdekli bir işlemcide iki süreç var. P1 önce 3 ms hesap yapacak, sonra diskten okuyacak (disk 3 ms sürüyor), sonra 2 ms daha hesap yapacak. P2 yalnızca 4 ms hesap yapacak. İkisi de t = 0'da hazır; çizelgeleyici önce P1'i seçiyor. Her süreç işlemciyi ancak beklemeye geçince ya da işi bitince bırakıyor.

Zaman (ms)	CPU	Disk	P1	P2
0–3	P1	boş	Çalışıyor	Hazır
3	P1 okuma ister, bağlam değiştirilir		Bekliyor	Çalışıyor
3–6	P2	P1'in verisi	Bekliyor	Çalışıyor
6	Disk biter, kesme gelir		Hazır	Çalışıyor
6–7	P2	boş	Hazır	Çalışıyor
7	P2 4 ms'sini tamamlar		Çalışıyor	Sonlandı
7–9	P1	boş	Çalışıyor	Sonlandı
9	P1 biter		Sonlandı	Sonlandı

İki süreç toplam 9 ms'de biter. 0–9 ms arasında işlemci hiç boş kalmaz: 3 ms P1, 4 ms P2, 2 ms yine P1 çalışır ve *3 + 4 + 2 = 9* eder.



Şekil 2.1.13: Örnek 2.1.6'nın zaman çizelgesi. Üst satır CPU'yu, alt satır diski gösterir. 3–6 ms arasında CPU P2'yi çalıştırırken disk P1'in verisini okur; ikisi paralel çalışır.

Şekil 2.1.13 örneğin can alıcı noktasını görünür kılar: 3–6 ms arasında iki ayrı donanım aynı anda iş yapar. CPU P2'yi çalıştırırken disk P1 için okuma yapar. İşletim sistemi bu örtüşmeyi kullanmasaydı ne olurdu? P1 diski beklerken CPU boş dursaydı, yani süreçler art arda çalışsaydı toplam süre şöyle olurdu:

> 3 (P1 hesap) + 3 (disk bekleme) + 2 (P1 hesap) + 4 (P2 hesap) = 12 ms

Çoklu programlama 3 ms kazandırır ve toplam süre 12 ms'den 9 ms'ye iner. Ardışık çalışmada CPU 12 ms'nin yalnızca 9 ms'sinde iş yapar, yani zamanın %75'inde doludur; örtüşmeli çalışmada bu oran %100'dür. Gerçek sistemlerde G/Ç beklemeleri çok daha uzun olduğu için kazanç da çok daha büyüktür.

Örnekte bir ayrıntı daha vardır. P1'in disk işi t = 6'da biter, ama P1 ancak t = 7'de çalışır. Aradaki 1 ms'yi hazır kuyruğunda geçirir, çünkü işlemcide P2 vardır ve bu çizelgelemede çalışan süreç işlemciden zorla alınmaz.

2.1.4.6 Zaman paylaşımı ve döner sıralı çizelgeleme

Önceki örnekte süreçler işlemciyi yalnızca beklemeye geçince bırakıyordu. Hiç G/Ç yapmayan, uzun bir hesap yapan bir süreç bu düzende işlemciyi dakikalarca tutabilir; o sırada kullanıcı fareyi kıpırdatsa bile ekran tepki vermez. Etkileşimli sistemler bu yüzden işlemciyi süreçlere kısa dilimler hâlinde dağıtır.

Tanım 2.1.11

Zaman paylaşımı (*time sharing*), işlemcinin süreçler arasında kısa aralıklarla dönüşümlü dağıtılmasıdır. **Döner sıralı çizelgeleme** (*round robin*), hazır kuyruğundaki her sürece sırayla en çok bir **zaman dilimi** (*time slice, quantum*) kadar işlemci verir. Süreç dilim içinde bitmezse kuyruğun sonuna gider ve bağlam değiştirilir. Zaman dilimi çoğu zaman q harfiyle gösterilir.

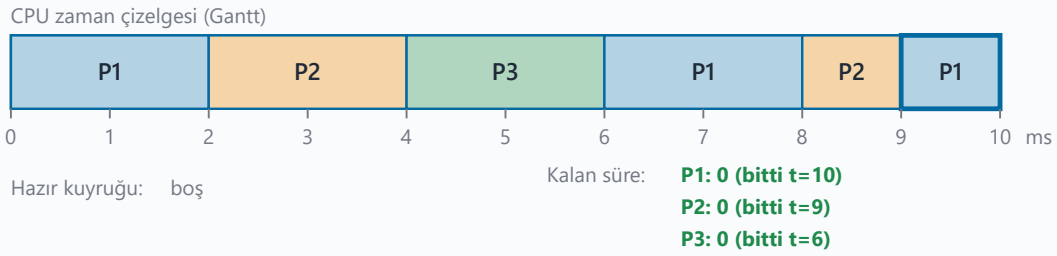
Döner sıralı çizelgelemenin kuralı basittir: kuyruğun başındaki süreci al, en çok q kadar çalıştır, bitmediyse kuyruğun sonuna koy. Kuralın sonuçlarını görmek için hesabı adım adım yapmak gerekir. Bu tür hesaplarda bir **Gantt çizelgesi** (*Gantt chart*) kullanılır: zaman ekseni üzerinde işlemcinin her aralıkta hangi süreçte olduğunu gösteren yatay çubuklar.

Örnek 2.1.7

Üç süreç t = 0'da hazır kuyruğunda, sıra P1, P2, P3. İşlemci süreleri P1 = 5 ms, P2 = 3 ms, P3 = 2 ms. Zaman dilimi $q = 2$ ms. Her sürecin bitiş zamanını bulalım.

Aralık (ms)	CPU	Ne olur?	Kalan (P1, P2, P3)	Kuyruk (sonra)
başlangıç	—	Üçü de hazır	5, 3, 2	P1, P2, P3
0–2	P1	Dilim doldu, P1 sona gider	3, 3, 2	P2, P3, P1
2–4	P2	Dilim doldu, P2 sona gider	3, 1, 2	P3, P1, P2
4–6	P3	P3 biter (t = 6)	3, 1, 0	P1, P2
6–8	P1	Dilim doldu, P1 sona gider	1, 1, 0	P2, P1
8–9	P2	P2 1 ms kullanır, biter (t = 9)	1, 0, 0	P1
9–10	P1	P1 1 ms kullanır, biter (t = 10)	0, 0, 0	boş

Bitiş zamanları: P3 = 6 ms, P2 = 9 ms, P1 = 10 ms. Toplam iş $5 + 3 + 2 = 10$ ms olduğu için son süreç t = 10'da biter.



Şekil 2.1.14: Örnek 2.1.7'nin tamamlanmış Gantt çizelgesi: P1, P2, P3, P1, P2, P1. Hazır kuyruğu boştur; P3 t = 6'da, P2 t = 9'da, P1 t = 10'da bitmiştir.

Tablonun iki ayrıntısı dikkat ister. Birincisi, 8–9 ve 9–10 aralıkları birer milisaniyedir. İşi dilimden önce biten süreç dilimin geri kalanını beklemez; işlemci hemen sıradakine geçer. İkincisi, kuyruk sütunu her adımda kimin sona gittiğini gösterir ve hesabın doğruluğunu bu sütun belirler. Kuyruk sırası atlanırsa bütün sonraki satırlar yanlış çıkar.

Şekil 2.1.14 sonucu çizelge olarak verir. Çizelgede altı çubuk vardır, yani işlemci altı kez bir sürece verilmiştir. Ardışık iki çubuk farklı süreçlere ait olduğunda araya bir bağlam değiştirme girer; bu örnekte 5 bağlam değiştirme yapılır. Hızlı geçişler sayesinde üç süreç de "aynı anda" ilerliyor gibi görünür. Gerçek sistemlerde zaman dilimi birkaç milisaniye ile birkaç on milisaniye arasındadır; insan bu kadar kısa geçişleri fark etmez.

Zaman diliminin seçimi bir dengedir. Dilim küçüldükçe süreçler daha sık sıra bulur, ama bağlam değiştirme sayısı artar. Dilim çok büyük seçilirse ($q = 5$) her süreç ilk seferde biter ve kısa işler uzun işin arkasında bekler.

Zaman dilimi	Bitişler (P1, P2, P3)	Bağlam değiştirme
$q = 1$	10, 8, 6	8
$q = 2$	10, 9, 6	5
$q = 5$	5, 8, 10	2

Tablo 2.1.7: Aynı üç sürecin (P1 = 5, P2 = 3, P3 = 2 ms) farklı zaman dilimleriyle çizelgelenmesi.

Tablo 2.1.7 etkiyi sayılarla gösterir; bağlam değiştirmenin süresi bu hesaplarda sıfır sayılmıştır.

Sık yapılan hata

Dilimi dolan sürecin kuyruğun başına döndüğünü varsaymak. Süreç her zaman kuyruğun *sonuna* gider; yoksa aynı süreç işlemciyi bırakmamış olurdu. İkinci yaygın hata da kalan süresi dilimden kısa olan sürece yine tam dilim vermektir: 8–9 aralığında P2'nin 1 ms'si kalmıştır ve işlemci t = 9'da P1'e geçer.

2.1.4.7 Çizelgeleme algoritmalarını karşılaştırma

Döner sıralı çizelgeleme tek seçenek değildir. Süreç çizelgeleme simülatöründe dört algoritma karşılaştırılabilir. Bu kesim bunların üçünü aynı örnek üzerinde hesaplar. Karşılaştırma için iki ölçü kullanılır.

Tanım 2.1.12

Bir sürecin **tamamlanma süresi** (*turnaround time*), hazır kuyruğuna geldiği andan işini bitirdiği ana kadar geçen süredir: tamamlanma süresi = bitiş − varış. **Bekleme süresi** (*waiting time*), bu sürenin işlemcide çalışmadan, hazır kuyruğunda geçen kısmıdır: bekleme süresi = tamamlanma süresi − işlemci süresi.

Üç algoritma şöyle çalışır. **FCFS** (*first come, first served*, ilk gelen ilk hizmet alır) süreçleri geliş sırasıyla, her birini sonuna kadar çalıştırır. **SJF** (*shortest job first*, en kısa iş önce) işlemci boşalınca hazır süreçlerden süresi en kısa olanı seçer ve onu sonuna kadar çalıştırır. **Döner sıralı** çizelgeleme ise Örnek 2.1.7'deki gibi dilimlerle dağıtır.

Örnek 2.1.8

Yine P1 = 5, P2 = 3, P3 = 2 ms; üçü de t = 0'da gelmiş, kuyruk sırası P1, P2, P3. Her algoritma için ortalama bekleme süresini bulalım. Varış 0 olduğundan tamamlanma süresi bitiş zamanına eşittir.

1. **FCFS.** Çizelge: P1 0–5, P2 5–8, P3 8–10. Bitişler 5, 8, 10. Beklemeler: P1 = 5 − 5 = 0, P2 = 8 − 3 = 5, P3 = 10 − 2 = 8. Ortalama *(0 + 5 + 8) / 3 = 13 / 3 ≈ 4,33* ms.
2. **SJF.** En kısa iş P3, sonra P2, sonra P1. Çizelge: P3 0–2, P2 2–5, P1 5–10. Beklemeler: P3 = 0, P2 = 5 − 3 = 2, P1 = 10 − 5 = 5. Ortalama *(0 + 2 + 5) / 3 = 7 / 3 ≈ 2,33* ms.
3. **Döner sıralı (q = 2).** Bitişler P1 = 10, P2 = 9, P3 = 6. Beklemeler: P1 = 10 − 5 = 5, P2 = 9 − 3 = 6, P3 = 6 − 2 = 4. Ortalama *(5 + 6 + 4) / 3 = 15 / 3 = 5* ms.

Ortalama bekleme en düşük SJF'de (≈ 2,33 ms), en yüksek döner sıralı çizelgelemededir (5 ms).

Algoritma	Bitişler (P1, P2, P3)	Ortalama tamamlanma	Ortalama bekleme
FCFS	5, 8, 10	23 / 3 ≈ 7,67 ms	13 / 3 ≈ 4,33 ms
SJF	10, 5, 2	17 / 3 ≈ 5,67 ms	7 / 3 ≈ 2,33 ms
Döner sıralı (q = 2)	10, 9, 6	25 / 3 ≈ 8,33 ms	15 / 3 = 5 ms

Tablo 2.1.8: Üç algoritmanın aynı örnekteki sonuçları (P1 = 5, P2 = 3, P3 = 2 ms; hepsi t = 0'da).

Tablo 2.1.8 sonuçları yan yana koyar. Döner sıralı çizelgelemenin amacı ortalama süreyi küçültmek değil, hiçbir süreci uzun süre işlemcisiz bırakmamaktır. FCFS'de P3 ilk kez t = 8'de çalışır; döner sıralıda ise t = 4'te çalışmaya başlar. Etkileşimli bir sistemde kullanıcının tıklamasına hızlı tepki vermek, ortalama sürenin biraz uzamasından daha önemlidir.

SJF ortalama beklemeyi en aza indirir, ama iki sorunu vardır. Birincisi, bir işin ne kadar süreceği çoğu zaman önceden bilinmez; işletim sistemi bunu ancak geçmiş davranıştan tahmin edebilir. İkincisi, sürekli kısa işler gelirse uzun bir iş hiç sıra bulamayabilir. Bir sürecin sonsuza dek sıra bekleyebilmesine *açlık* denir.

Sık yapılan hata

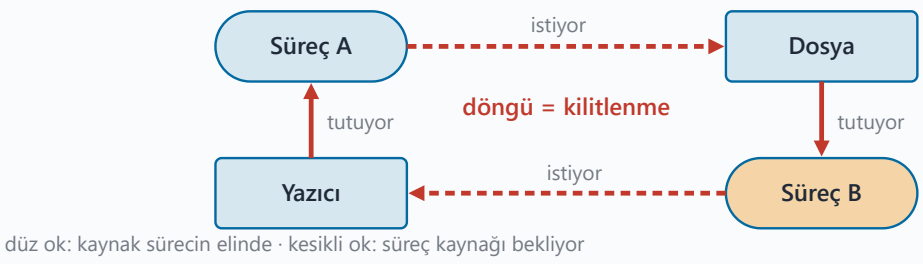
Bekleme süresini "sürecin ilk kez çalışmaya başladığı an" sanmak. Bekleme süresi sürecin hazır kuyruğunda geçirdiği *bütün* aralıkların toplamıdır. Döner sıralı örnekte P1 0–2'de çalışır, sonra 2–6 arasında 4 ms ve 8–9 arasında 1 ms bekler; toplam bekleme 5 ms'dir ve bu değer 10 − 5 formülüyle aynıdır.

2.1.4.8 Kilitlenme

Süreçler yalnızca işlemciyi değil, yazıcı, dosya ya da veri tabanı kaydı gibi kaynakları da paylaşır. Bu kaynakların çoğunu aynı anda yalnızca bir süreç kullanabilir. Bir yazıcıya iki süreç aynı anda yazsa sayfalara iki belgenin satırları karışık basılır. Süreçler bu yüzden kaynağı kullanmadan önce onu "tutar", işi bitince bırakır. Bu düzen, belirli koşullarda süreçlerin birbirini sonsuza dek beklemesine yol açabilir.

Tanım 2.1.13

Kilitlenme (*deadlock*), bir grup sürecin her birinin, gruptaki başka bir sürecin tuttuğu kaynağı beklediği ve hiçbirinin ilerleyemediği durumdur.



Şekil 2.1.15: Kaynak tahsis grafiği. Düz ok kaynağın bir sürecin elinde olduğunu, kesikli ok sürecin o kaynağı beklediğini gösterir. A yazıcıyı tutup dosyayı, B dosyayı tutup yazıcıyı bekler; oklar bir döngü oluşturur.

Örnek 2.1.9

Süreç A bir raporu yazdıracak ve yazdırırken bir dosyaya kayıt düşecek. Süreç B dosyayı güncelleyecek ve sonra yazıcıdan bir özet alacak. Olaylar şu sırayla gerçekleşiyor:

1. A yazıcıyı alır ve kullanır.
2. B dosyayı kilitler ve kullanır.
3. A, yazıcıyı bırakmadan dosyayı ister. Dosya B'de olduğu için A bekler.
4. B de dosyayı bırakmadan yazıcıyı ister. Yazıcı A'da olduğu için B bekler.

Bekleme zinciri A → Dosya → B → Yazıcı → A biçiminde kapanır. İkisi de elindekini bırakmadığı için ikisi de sonsuza dek bekler.

Şekil 2.1.15 dördüncü adımdan sonraki durumu gösterir. Kilitlenmenin bir grafikte görünen işareti oklardaki döngüdür. Döngüdeki her süreç bir kaynağı tutar ve döngüdeki bir sonraki kaynağı bekler.

Kilitlenme ancak dört koşul aynı anda sağlanırsa oluşur. Bu koşulları 1971'de Coffman ve arkadaşları tanımlamıştır; kilitlenme bugün de bu dört koşulla açıklanır.

Teorem 2.1.1 (Kilitlenmenin dört koşulu)

Kilitlenme için şu dört koşulun *hepsi* gereklidir:

1. **Karşılıklı dışlama** (*mutual exclusion*): kaynağın aynı anda tek sahibi olabilir.
2. **Tut ve bekle** (*hold and wait*): süreç elindeki kaynağı bırakmadan başka bir kaynak bekler.
3. **Ön alınamazlık** (*no preemption*): kaynak sahibinin elinden zorla alınamaz; ancak sahibi bırakınca boşalır.
4. **Döngüsel bekleme** (*circular wait*): bekleyen süreçler halka biçiminde birbirine bağlıdır.

Koşullardan biri bile kırılırsa kilitlenme oluşamaz.

Teorem 2.1.1 çözümün de yolunu gösterir: dört koşuldan birini kırmak yeter. Pratikte en kolay kırılan koşul döngüsel beklemedir. Bütün süreçlerin kaynakları *aynı sırayla* istemesi kural hâline getirilir, örneğin önce Yazıcı, sonra Dosya. Örnek 2.1.9'daki B de önce yazıcıyı istemek zorunda kalırsa, yazıcı A'dayken B hiçbir şey tutmadan bekler. A dosyayı da alır, işini bitirir, iki kaynağı bırakır ve B devam eder. Halka kurulamaz.

Karşılıklı dışlamayı kırmak çoğu zaman mümkün değildir: bir yazıcı doğası gereği tek sahiplidir.

2.1.5 Bellek yönetimi

İşlemci komutları ve verileri yalnızca RAM'den okur; bu yüzden her program çalışmak için belleğe yüklenmek zorundadır. İşletim sistemi her sürece yer bulmalı, süreçlerin birbirinin alanına girmesini engellemeli ve RAM yetmediğinde bir çıkış yolu sunmalıdır.

2.1.5.1 Bellek yönetiminin görevleri

Görev	Anlamı
Ayırma	Her programa gereksinimine göre bellek alanı sağlamak
Koruma	Bir sürecin başka bir sürecin belleğini izinsiz değiştirmesini önlemek
Sanal bellek	Programlara fiziksel RAM'den bağımsız bir adres alanı sunmak
Sayfalama	Belleği sabit boyutlu parçalarla yönetmek
Takas	Gerektiğinde bazı verileri geçici olarak diske taşımak
Paylaşım	Ortak kodu ya da veriyi denetimli biçimde birden fazla sürece açmak

Tablo 2.1.9: Bellek yönetiminin altı görevi.

Tablo 2.1.9 görevleri listeler. İlk ikisi, ayırma ve koruma, her bellek yönetiminin olmazsa olmazıdır. Sanal bellek ve sayfalama, bu iki görevi esnek biçimde yapmanın bugünkü yoludur. Takas, RAM yetmeyince devreye girer. Paylaşım ise belleği verimli kullanmaya yarar. Örneğin aynı tarayıcıdan açılan beş süreç, programın kodunu RAM'de beş kez değil bir kez tutar; kod yalnızca okunduğu için paylaşmak güvenlidir. Her sürecin kendi verisi ise ayrı kalır.

2.1.5.2 Ayırma, koruma ve takas adım adım

Bellek yönetiminin temel fikirleri, gerçek sistemlerden daha basit bir modelde açıkça görülür. Bu modelde her sürece RAM'de *bitişik* tek bir bölge verilir ve bölge iki sayıyla tanımlanır.

Tanım 2.1.14

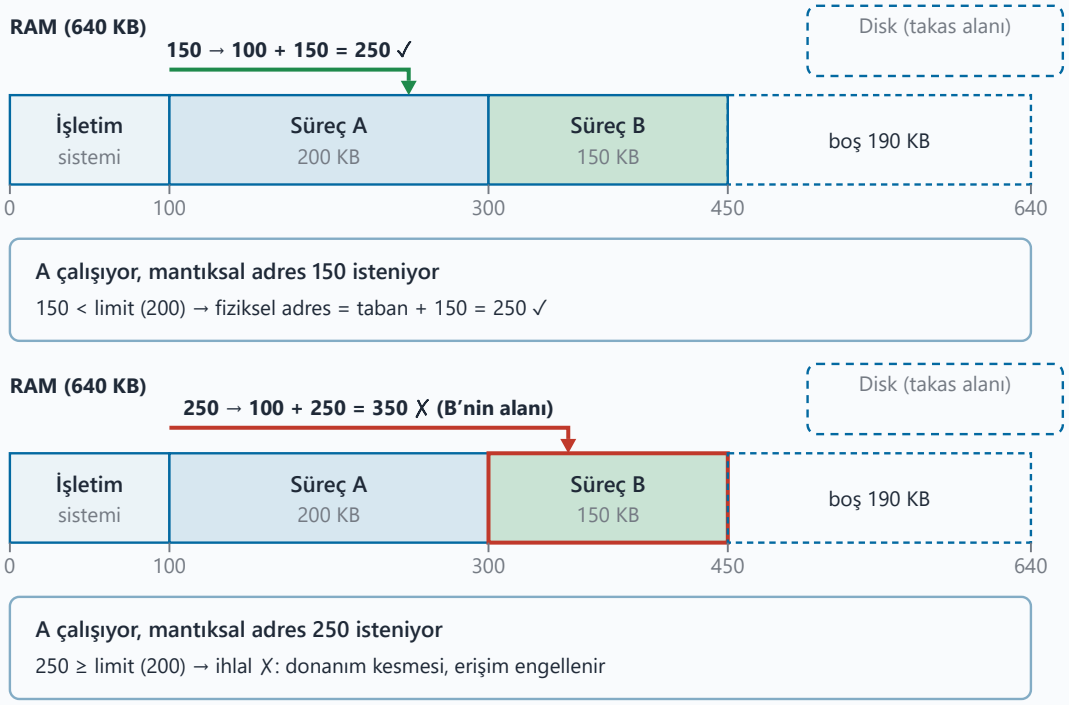
Taban yazmacı (*base register*) sürecin bölgesinin RAM'deki başlangıç adresini, **limit yazmacı** (*limit register*) bölgenin uzunluğunu tutar. Süreç kendi bölgesinin başını 0 kabul eden **mantıksal adreslerle** çalışır. Donanım her erişimde mantıksal adresi limitle karşılaştırır. Adres limitten küçükse **fiziksel adres = taban + mantıksal adres** olur; değilse erişim engellenir.

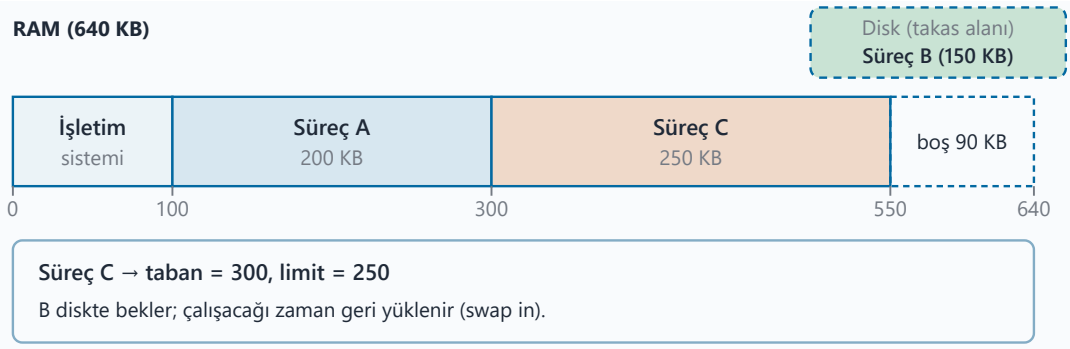
Bu iki yazmaç işlemcinin içindedir ve donanım onları her bellek erişiminde kendiliğinden kullanır. Süreç değiştiğinde işletim sistemi, yeni sürecin taban ve limit değerlerini yazmaçlara yükler; bu iş bağlam değiştirmenin bir parçasıdır. Süreç bu yazmaçları kendisi değiştiremez, çünkü onlara yazan komutlar yalnızca çekirdek modunda çalışır.

Örnek 2.1.10

RAM 640 KB'dır ve ilk 100 KB'ı işletim sistemi kullanır. Sırayla A (200 KB) ve B (150 KB) süreçleri yükleniyor, A iki bellek erişimi yapıyor, sonra 250 KB isteyen C süreci geliyor. Adresler KB cinsindendir.

Adım	Olay	Hesap	Sonuç
1	Başlangıç	640 − 100 = 540	0–100 işletim sistemi, 540 KB boş
2	A yüklenir	taban = 100, limit = 200; 100 + 200 = 300	A: 100–300; 340 KB boş
3	B yüklenir	taban = 300, limit = 150; 300 + 150 = 450	B: 300–450; 640 − 450 = 190 KB boş
4	A, 150 adresini ister	150 < 200 ✓; 100 + 150 = 250	Fiziksel adres 250'ye gidilir
5	A, 250 adresini ister	250 ≥ 200 ✗	Erişim engellenir; denetim çekirdeğe geçer
6	C 250 KB ister	190 < 250	Yer yok; beklemedeki B diske yazılır, 640 − 300 = 340 KB bitişik boş alan
7	C yüklenir	taban = 300, limit = 250; 300 + 250 = 550	C: 300–550; 640 − 550 = 90 KB boş; B diskte





Şekil 2.1.16: Örnek 2.1.10'un üç adımı. (a) Adım 4: A'nın 150 adresi limitin altındadır ve 250 fiziksel adresine çevrilir. (b) Adım 5: 250 adresi limiti aşar; izin verilseydi 350 ile B'nin alanına yazılırdı. (c) Adım 7: B diske yazılmış, C 300–550 arasına yüklenmiştir.

Şekil 2.1.16 tablonun önemli anlarını RAM şeridi üzerinde gösterir. Ayırmada (adım 1–3) her yeni süreç boş alanın başından yer alır ve bölgeler çakışmaz. Geçerli erişimde (adım 4) A kendi gözünden 150 adresini ister; A, bölgesinin RAM'de 100'den başladığını bilmez. Donanım önce 150'yi limitle karşılaştırır, sonra tabanı ekler: 100 + 150 = 250.

Koruma ihlali (adım 5). A bu kez 250 adresini ister. 250, 200'den küçük değildir; adres A'nın bölgesinin dışındadır. Donanım erişimi yapmaz, bir kesme üretir ve denetimi çekirdeğe verir. Gerçek bir sistemde işletim sistemi A'yı "erişim ihlali" (*segmentation fault*) hatasıyla sonlandırır; örneği sürdürmek için burada A'nın bölgesi yerinde bırakılır. Donanım engellemeseydi fiziksel adres 100 + 250 = 350 olurdu, yani B'nin bölgesine yazılırdı. Koruma sayesinde B'nin belleği bozulmaz.

Takas (adım 6–7). C 250 KB ister. RAM'de 450–640 arasında 190 KB boş alan vardır; 190 < 250 olduğu için C sığmaz. O sırada B bir disk işini bekliyor ve bir süre çalışmayacak. İşletim sistemi B'nin belleğinin tamamını diskteki **takas alanına** (*swap space*) yazar. Bu işleme *dışarı takas* denir. 300–640 arasında 340 KB'lık bitişik boş alan oluşur ve C 300–550 arasına yüklenir; geriye 90 KB kalır. B yeniden çalışacağı zaman RAM'e geri alınır. Yer yoksa bu kez başka bir süreç diske yazılır.

Tanım 2.1.15

Takas (*swapping*), RAM'de yer açmak için bir sürecin ya da onun bir parçasının belleğini geçici olarak diske yazmak, gerektiğinde de geri okumaktır. Diskte bu iş için ayrılan alana **takas alanı** denir.

Takas kapasiteyi artırır, ama ağır bir bedeli vardır: disk RAM'den çok yavaştır. Bu fark sanal bellek kesiminde sayılarla verilir.

Sık yapılan hata

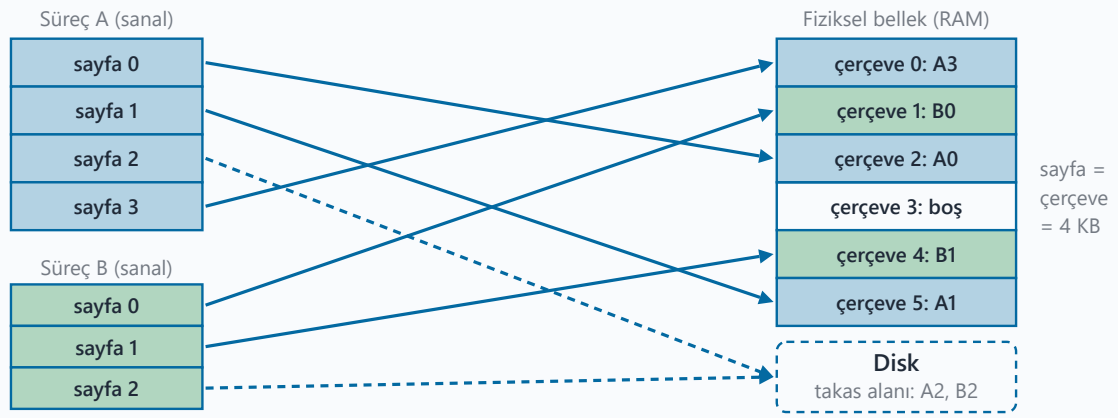
Limit yazmacının bölgenin *bitiş adresini* tuttuğunu düşünmek. Limit bölgenin uzunluğudur. A için limit 200'dür, bitiş adresi 300 değil. Bu yüzden karşılaştırma mantıksal adresle yapılır (150 < 200), fiziksel adresle değil. Geçerli mantıksal adresler 0'dan 199'a kadardır; 200 adresi bile bölgenin dışındadır.

Bu basit modelin bir zayıflığı vardır: süreçler gelip gittikçe RAM'de küçük boşluklar oluşur ve toplam boş alan yeterli olsa bile bitişik bir parça bulunamayabilir. Sanal bellek ve sayfalama bu sorunu çözer.

2.1.5.3 Sanal bellek ve sayfalama

Tanım 2.1.16

Sanal bellek (*virtual memory*), her sürece 0'dan başlayan, fiziksel RAM'den bağımsız kendi **sanal adres** alanını sunan düzendir. **Sayfalamada** (*paging*) sanal adres alanı eşit boyutlu **sayfalara** (*page*), fiziksel RAM de aynı boyutta **çerçevelere** (*frame*) bölünür. Her sürecin bir **sayfa tablosu** (*page table*) vardır ve tablo, her sayfanın hangi çerçevede ya da diskte olduğunu tutar. Yaygın sayfa boyutu 4 KB'dır (4.096 bayt).



Şekil 2.1.17: İki sürecin sanal sayfaları fiziksel çerçevelere ya da diske eşlenir. A'nın 0, 1 ve 3 numaralı sayfaları sırasıyla 2, 5 ve 0 numaralı çerçevelerde, 2 numaralı sayfası diskteki takas alanındadır. B'nin 0 ve 1 numaralı sayfaları 1 ve 4 numaralı çerçevelerde, 2 numaralı sayfası diskte durur. Çerçeve 3 boştur.

Şekil 2.1.17 sayfalamanın üç önemli sonucunu gösterir.

- **Her süreç kendi dünyasını görür.** A da B de 0 numaralı sayfadan başlar. İki sürecin "0 numaralı adresi" RAM'de farklı yerlerdir: A'nınki çerçeve 2'de, B'ninki çerçeve 1'de. Bir süreç ötekinin sayfasına hiç adres üretemez, çünkü kendi tablosunda o çerçeve yoktur. Koruma kendiliğinden sağlanır.

- **Sayfalar RAM'de ardışık olmak zorunda değildir.** A'nın sayfaları 2, 5 ve 0 numaralı çerçevelere dağılmıştır. Her boş çerçeve her sayfaya uyar; bitişik yer arama sorunu ortadan kalkar.
- **Az kullanılan sayfa diske çıkarılabilir.** A2 ve B2 diskte bekler. Süreç onlara ihtiyaç duyana kadar RAM'de yer kaplamazlar. Takas artık sürecin tamamıyla değil, tek tek sayfalarla yapılır.

Takas alanı Windows'ta **sayfa dosyası** (*page file*), Linux ve macOS'ta *swap* adını alır. Sanal belleğin bedeli hızdır. Bir RAM erişimi yaklaşık 50–100 nanosaniye (ns) sürer; bir SSD'den okuma ise yaklaşık 0,1 milisaniyedir (ms). İki süreyi aynı birime çevirerek karşılaştıralım:

> 0,1 ms = 100.000 ns; 100.000 ÷ 100 = 1.000 ve 100.000 ÷ 50 = 2.000

Diskten bir sayfa okumak, bir RAM erişiminden 1.000 ile 2.000 kat daha uzun sürer. Takas az olduğu sürece bu fark hissedilmez. Ama RAM çok yetersizse işletim sistemi sürekli sayfa yazıp okumaya başlar, disk ışığı yanıp söner ve bilgisayar belirgin biçimde yavaşlar. Disk boş olsa bile RAM yetersizliği bu yüzden yavaşlığa yol açar.

2.1.5.4 Adres çevirisi adım adım

Süreç bir sanal adres ürettiğinde, işlemcinin içindeki **bellek yönetim birimi** (*memory management unit, MMU*) bu adresi fiziksel adrese çevirir. Çeviri iki aşamadır: önce adres *sayfa numarası* ve sayfa içindeki *bağıl konum* (*offset*) olarak ikiye ayrılır, sonra sayfa numarası tabloda aranır.

> sayfa = sanal adres ÷ sayfa boyutu (tam bölüm); bağıl konum = sanal adres mod sayfa boyutu

> fiziksel adres = çerçeve × sayfa boyutu + bağıl konum

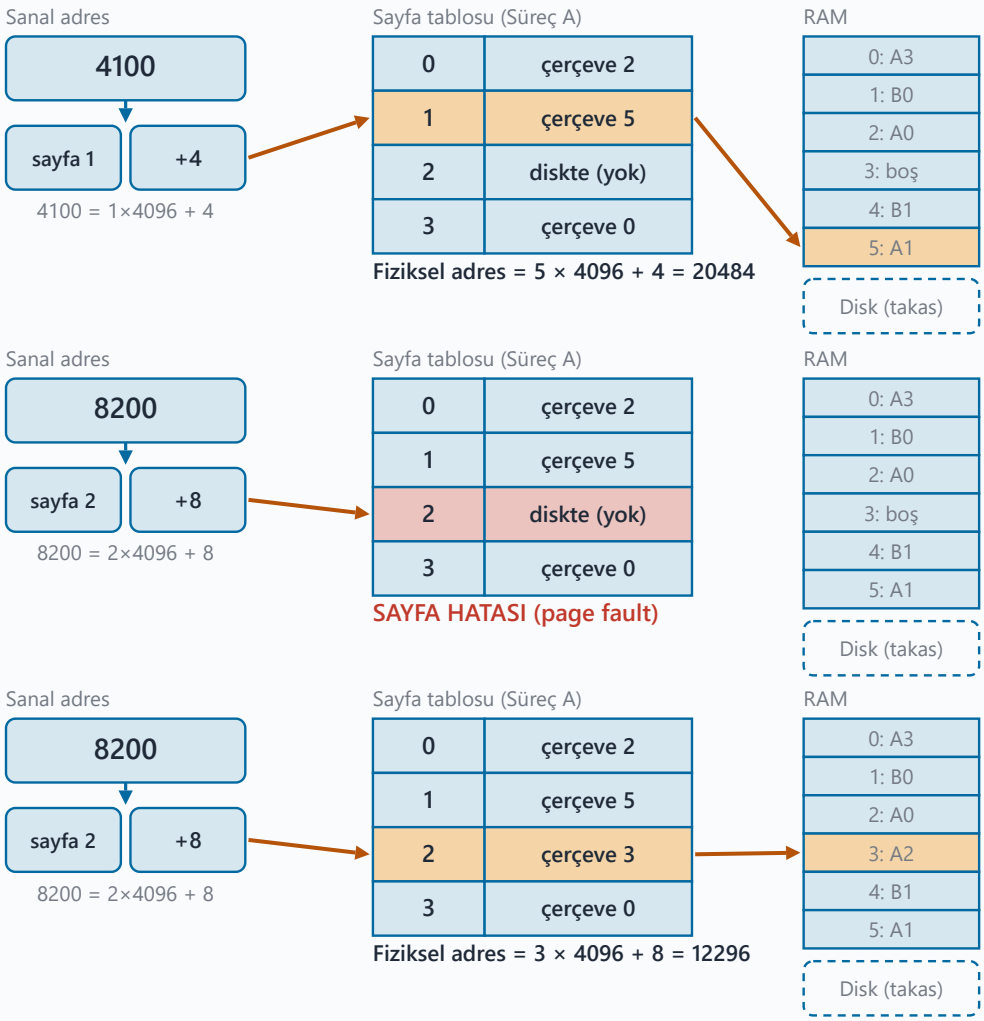
Burada "mod", bölmeden kalanı gösterir. Bağıl konum çeviride hiç değişmez: sayfa bütün hâlinde bir çerçeveye taşındığı için, sayfanın 4. baytı çerçevenin de 4. baytıdır. Değişen yalnızca sayfa numarasının yerine çerçeve numarasının gelmesidir.

Örnek 2.1.11

Sayfa boyutu 4.096 bayt. Süreç A'nın sayfa tablosu: sayfa 0 → çerçeve 2, sayfa 1 → çerçeve 5, sayfa 2 → diskte, sayfa 3 → çerçeve 0. Çerçeve 3 boş. A önce 4100, sonra 8200 sanal adresini okuyor.

1. **Bölme.** 4100 ÷ 4096 = 1, kalan 4. Yani 4100 = 1 × 4096 + 4: sayfa 1, bağıl konum 4.
2. **Tablo.** Sayfa 1 → çerçeve 5. Sayfa RAM'dedir.
3. **Fiziksel adres.** 5 × 4096 + 4 = 20.480 + 4 = 20.484. Okuma bu adresten yapılır.
4. **Bölme.** 8200 ÷ 4096 = 2, kalan 8. Yani 8200 = 2 × 4096 + 8: sayfa 2, bağıl konum 8.
5. **Tablo.** Sayfa 2 RAM'de değil, diskte. MMU bir **sayfa hatası** üretir; komut durdurulur ve işletim sistemi devreye girer.

6. **Yükleme.** İşletim sistemi sayfa 2'yi diskten boş çerçeve 3'e okur ve tabloyu günceller: sayfa 2 → çerçeve 3.
7. **Yeniden deneme.** 3 × 4096 + 8 = 12.288 + 8 = 12.296. Bu kez çeviri başarılıdır.



Şekil 2.1.18: Örnek 2.1.11'in üç anı. (a) 4100 = 1 × 4096 + 4; sayfa 1 çerçeve 5'te, fiziksel adres 20484. (b) 8200 adresinin sayfası (2) diskte: sayfa hatası. (c) Sayfa 2 çerçeve 3'e yüklenmiş; fiziksel adres 3 × 4096 + 8 = 12296.

Tanım 2.1.17

Sayfa hatası (*page fault*), sürecin eriştiği sayfanın o anda RAM'de olmadığını MMU'nun bildirmesidir. İşletim sistemi sayfayı diskten bir çerçeveye yükler, sayfa tablosunu günceller ve kesilen komutu yeniden çalıştırır.

Şekil 2.1.18 çevirinin üç anını gösterir. Adının aksine sayfa hatası çoğunlukla bir programlama hatası değildir; sanal belleğin olağan işleyişidir. Bir program açılırken sayfalarının çoğu henüz diskte durur ve ilk erişimlerde sayfa hatalarıyla birer birer RAM'e gelir. Program bunu hiç fark etmez; yalnızca o erişim biraz uzun sürer. Gerçek bir hata ancak sürecin kendisine ait olmayan, tabloda hiç bulunmayan bir adrese erişmesiyle ortaya çıkar; o zaman işletim sistemi süreci erişim ihlaliyle sonlandırır.

Bölme ve kalan hesabı ikili sayılarla daha da kolaylaşır. $4096 = 2^{12}$ olduğu için bir sanal adresin *son 12 biti* bağıl konumu, *kalan bitleri* sayfa numarasını verir. Donanım bölme yapmaz, yalnızca bitleri ikiye ayırır. Onaltılık yazımda her basamak 4 bit olduğundan son üç onaltılık basamak bağıl konumdur:

Sanal adres	Onaltılık	Sayfa · bağıl konum	Çerçeve	Fiziksel adres (onaltılık)	Fiziksel adres
4100	1004_{16}	1 · 004	5	5004_{16}	20484
8200	2008_{16}	2 · 008	3	3008_{16}	12296

Tablo 2.1.10: Adres çevirisi onaltılık gösterimde: son üç basamak (12 bit) aynen korunur.

Tablo 2.1.10 bu kısayolu gösterir. 4100 onaltılıkta 1004_{16}'dır. Baştaki 1 sayfa numarası, sondaki 004 bağıl konumdur. Sayfa 1 çerçeve 5'te olduğu için baştaki 1 yerine 5 yazılır: $5004_{16} = 20484$. Aynı sonucu onluk hesap da vermişti.

Sık yapılan hata

Fiziksel adresi bulurken çerçeve numarasına bağıl konumu doğrudan eklemek: 5 + 4 = 9. Çerçeve numarası bir sıra numarasıdır; çerçevenin başlangıç adresi için sayfa boyutuyla çarpılması gerekir: 5 × 4096 = 20.480. Bağıl konum bu başlangıca eklenir.

2.1.5.5 Sayfa yer değiştirme: LRU ve FIFO

Sayfa hatası olduğunda boş çerçeve varsa iş kolaydır. RAM doluysa işletim sistemi bir sayfayı diske çıkarıp yerine yenisini yüklemek zorundadır. Hangi sayfanın çıkarılacağına karar vermeye **sayfa yer değiştirme** (*page replacement*) denir. Yanlış seçim, hemen ardından aynı sayfa için yeni bir sayfa hatası doğurur. Her hata da yavaş bir disk okuması demektir.

Tanım 2.1.18

LRU (*least recently used*, en uzun süredir kullanılmayan) kuralı, son kullanımı en eskiye dayanan sayfayı çıkarır. **FIFO** (*first in, first out*, ilk giren ilk çıkar) kuralı ise RAM'e en önce yüklenmiş sayfayı çıkarır. İstenen sayfanın zaten RAM'de olmasına **isabet** denir.

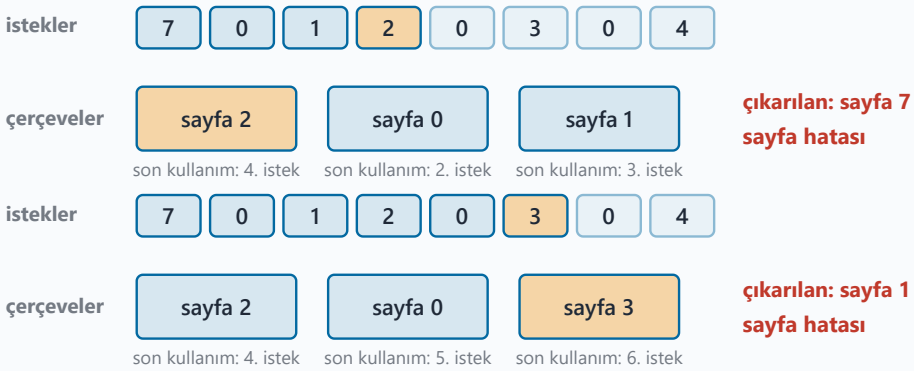
LRU'nun arkasındaki fikir şudur: programlar yakın zamanda kullandıkları sayfaları yakında yeniden kullanma eğilimindedir. Bir döngü aynı komutları tekrar tekrar çalıştırır; bir fonksiyon aynı değişkenlere tekrar tekrar erişir. Uzun süredir dokunulmamış bir sayfanın yakında gerekme olasılığı daha düşüktür.

Örnek 2.1.12

Sürecin RAM'de yalnızca 3 çerçevesi var. Süreç sayfaları sırayla 7, 0, 1, 2, 0, 3, 0, 4 diye istiyor. LRU kuralıyla kaç sayfa hatası olur?

İstek no	Sayfa	Sonuç	Çıkarılan	RAM'deki sayfalar (son kullanım)
1	7	hata	—	7 (1)
2	0	hata	—	7 (1), 0 (2)
3	1	hata	—	7 (1), 0 (2), 1 (3)
4	2	hata	7	2 (4), 0 (2), 1 (3)
5	0	isabet	—	2 (4), 0 (5), 1 (3)
6	3	hata	1	2 (4), 0 (5), 3 (6)
7	0	isabet	—	2 (4), 0 (7), 3 (6)
8	4	hata	2	4 (8), 0 (7), 3 (6)

8 istekte 6 sayfa hatası, 2 isabet olur. İlk üç hata boş çerçevelere yüklemedir; sonraki üç hatada bir sayfa çıkarılır.



Şekil 2.1.19: Örnek 2.1.12'nin iki çıkarma anı. (a) 4. istek: sayfa 2 için yer yoktur; son kullanımı en eski olan sayfa 7 (1. istek) çıkarılır. (b) 6. istek: sayfa 3 için en eski kullanım sayfa 1'dedir (3. istek); sayfa 0 az önce, 5. istekte kullanıldığı için korunur.

Şekil 2.1.19 çıkarma kararlarının ikisini gösterir. Tablodaki parantez içindeki sayılar, LRU'nun karar için tuttuğu bilgidir: her sayfanın en son hangi istekte kullanıldığı. Çıkarılacak sayfa, bu sayısı en küçük olandır. Beşinci istekte sayfa 0 isabet ettiğinde yalnızca son kullanım zamanı 2'den 5'e güncellenir. Bu küçük güncelleme, altıncı istekte sayfa 0'ı çıkarılmaktan kurtarır.

Aynı istek dizisi FIFO ile işlenirse ne olur? FIFO yalnızca sayfaların RAM'e giriş sırasına bakar; isabetler sırayı değiştirmez.

İstek no	Sayfa	Sonuç	Çıkarılan	RAM (eskiden yeniye)
1–3	7, 0, 1	3 hata	—	7, 0, 1
4	2	hata	7	0, 1, 2
5	0	isabet	—	0, 1, 2
6	3	hata	0	1, 2, 3
7	0	hata	1	2, 3, 0
8	4	hata	2	3, 0, 4

Tablo 2.1.11: Aynı dizinin (7, 0, 1, 2, 0, 3, 0, 4; 3 çerçeve) FIFO ile işlenmesi. RAM sütunu giriş sırasını gösterir: soldaki en eskidir.

Tablo 2.1.11 FIFO'nun 7 sayfa hatası verdiğini gösterir; LRU'dan bir fazla. Fark altıncı istekte doğar. FIFO, sayfa 0'ın az önce kullanıldığını dikkate almaz; 0 en eski giren sayfa olduğu için onu çıkarır. Hemen ardından yedinci istek yine sayfa 0'ı ister ve gereksiz bir hata oluşur. LRU ise sık kullanılan sayfa 0'ı korumuştu.

Her sayfa hatası yaklaşık 0,1 ms'lik bir SSD okuması gerektirirse LRU'nun 6 hatası yaklaşık 0,6 ms, FIFO'nun 7 hatası 0,7 ms tutar; isabetler ise nanosaniyeler içinde biter.

LRU her dizide FIFO'dan iyi değildir, ama gerçek programların çoğunda daha az hata verir. Kesin LRU her erişimde zaman kaydı gerektirdiği için pahalıdır; işletim sistemleri ona yaklaşan daha ucuz kurallar kullanır.

> **Sık yapılan hata**
>
> LRU ile FIFO'yu aynı sanmak, çünkü ikisi de "eski" sayfayı çıkarır. FIFO *RAM'e girişi* en eski olanı, LRU *son kullanımı* en eski olanı çıkarır. İsabet FIFO'da hiçbir şeyi değiştirmez, LRU'da ise sayfanın son kullanım zamanını yeniler.

2.1.6 Aygıt ve dosya yönetimi

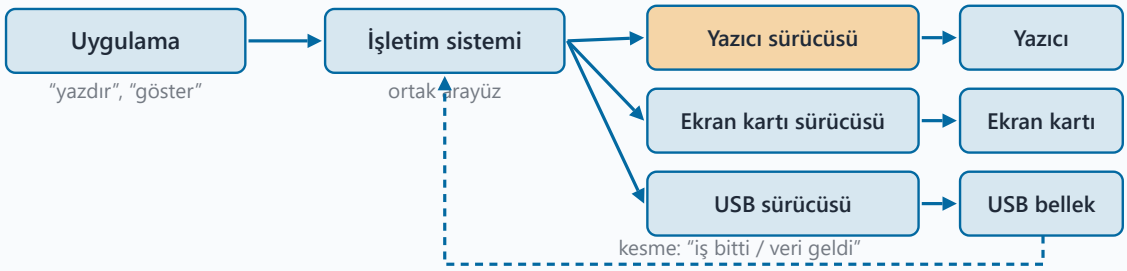
Kullanıcı bilgisayarla klavye, ekran, yazıcı ve diskler aracılığıyla konuşur. İşletim sistemi bu aygıtları sürücülerle yönetir, disk üzerindeki ham veriyi de adı, yeri ve sahibi olan dosyalar biçiminde düzenler.

2.1.6.1 Aygıtlar ve sürücüler

Bir lazer yazıcı ile bir mürekkep püskürtmeli yazıcı, aynı "sayfa yazdır" isteği için tamamen farklı komutlar bekler. İki farklı üreticinin ekran kartları birbirinden farklı yazmaçlara sahiptir. İşletim sistemi bu farklılıkların hepsini kendi içinde tutsaydı her yeni aygıt için işletim sisteminin değişmesi gerekirdi.

Tanım 2.1.19

Aygıt sürücüsü (*device driver*), işletim sisteminin ortak isteklerini belirli bir donanıma özgü işlemlere çeviren yazılımdır. Sürücüyü çoğunlukla aygıtın üreticisi yazar; sürücü sonradan işletim sistemine eklenir.



Şekil 2.1.20: Uygulama "yazdır" ya da "göster" gibi ortak istekler verir. İşletim sistemi bunları ortak bir arayüzle ilgili sürücüye (yazıcı, ekran kartı, USB) iletir. Aygıt işini bitirince kesmeyle "iş bitti / veri geldi" bildirir.

Şekil 2.1.20 sürücülerin yerini gösterir: işletim sistemiyle donanım arasında, her aygıt için ayrı bir çevirmen. Aygıt yönetimiyle ilgili üç kavram daha bu düzenin parçasıdır.

- **Tak ve çalıştır** (*plug and play*): işletim sistemi yeni takılan uyumlu bir aygıtı algılar, kimliğini okur ve uygun sürücüyü kendiliğinden yükleyip yapılandırır. Bir USB bellek takıldığında birkaç saniye içinde Dosya Gezgini'nde görünmesi bu sayededir.
- **Kesme:** aygıt bir olayı işlemciye bildirir. Klavyede tuşa basılması, yazıcının işi bitirmesi, ağdan paket gelmesi birer kesmedir.
- **Tampon** (*buffer*): hızları farklı bileşenler arasında veriyi geçici olarak tutan bellek alanıdır.

Örnek 2.1.13

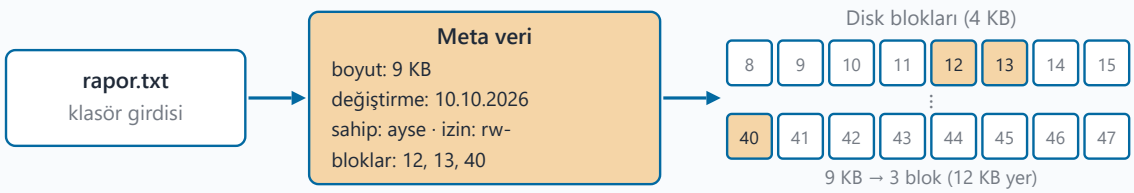
20 sayfalık bir belge, dakikada 10 sayfa basan bir yazıcıya gönderiliyor. Tampon olmasaydı kelime işlemci her sayfada yazıcıyı bekler ve 2 dakika boyunca donmuş gibi görünürdü. Tamponla işletim sistemi 20 sayfanın tamamını kısa sürede yazdırma kuyruğuna alır ve kelime işlemci hemen serbest kalır. Sürücü sayfaları tampondan yazıcının hızında gönderir; her sayfa bitince yazıcı bir kesmeyle hazır olduğunu bildirir. Yazdırma yine 2 dakika sürer, ama kullanıcı o sürede çalışmaya devam eder.

2.1.6.2 Dosya sistemi

Bir disk, kendi başına numaralanmış milyonlarca bloktan oluşan büyük bir depodur. "Ödevim 18.452 numaralı blokta başlıyor" diye bir şey hatırlamak mümkün değildir. Kullanıcı ve programlar dosyaları adlarıyla anar. Ad ile blok numarası arasındaki köprüyü dosya sistemi kurar.

Tanım 2.1.20

Dosya sistemi (*file system*), verinin depolama aygıtında nasıl adlandırıldığını, düzenlendiğini ve bulunduğunu tanımlayan düzendir. **Dosya** adı ve içeriği olan veri birimidir. **Klasör** (*folder, directory*) dosyaları ve alt klasörleri gruplar. **Yol** (*path*) dosyanın klasör yapısındaki konumunu belirtir. **Meta veri** (*metadata*) dosyanın içeriği dışındaki özellikleridir: boyut, tarih, sahip ve izinler. **İzinler** (*permission*) okuma, yazma ve çalıştırma yetkilerini belirler. **Günlükleme** (*journaling*) beklenmeyen kesintilerden sonra dosya sisteminin tutarlı kalmasını destekler.



Şekil 2.1.21: Klasördeki *rapor.txt* girdisi dosyanın meta verisini gösterir: boyut 9 KB, değiştirme 10.10.2026, sahip ayse, izin rw-, bloklar 12, 13 ve 40. Disk 4 KB'lık bloklara bölünmüştür; 9 KB'lık dosya 3 blok, yani 12 KB yer kaplar.

Şekil 2.1.21 bir dosyanın diskte nasıl bulunduğunu üç halkalı bir zincir olarak gösterir. Klasör, içindeki her dosyanın adını ve o dosyanın meta veri kaydının yerini tutar. Meta veri kaydında dosyanın bütün özellikleri ve içeriğin hangi disk bloklarında olduğu yazar. İçeriğin kendisi de o bloklardadır. Bloklar ardışık olmak zorunda değildir: *rapor.txt*'nin ilk iki parçası 12 ve 13 numaralı bloklarda, üçüncü parçası 40 numaralı bloktadır.

Örnek 2.1.14

Blok boyutu 4 KB'dır. 9 KB'lık *rapor.txt* diskte kaç blok ve ne kadar yer kaplar? Aynı diskte 1 baytlık bir dosya ne kadar yer kaplar?

1. 9 ÷ 4 = 2,25. Kesirli blok olmadığı için yukarı yuvarlanır: 3 blok.
2. Kaplanan yer: 3 × 4 = 12 KB. Son blokta 12 − 9 = 3 KB boş kalır; bu alan başka bir dosyaya verilemez.
3. 1 baytlık dosya da en az bir blok ister: 4 KB yer kaplar.

Windows'ta bir dosyanın Özellikler penceresinde “Boyut” ve “Diskteki boyut” diye iki ayrı değer görünmesi bu yüzdendir.

Günlükleme, dosya sisteminin bir değişikliği yapmadan önce ne yapacağını ayrı bir günlük alanına yazmasıdır. Bir dosyayı kaydetmek birkaç adım ister: içerik bloklara yazılır, meta veri güncellenir, boş blok listesi değişir. Elektrik bu adımların ortasında kesilirse dosya sistemi tutarsız kalabilir: örneğin

bir blok hem boş hem dolu görünür. Günlüklü bir dosya sistemi açılışta günlüğe bakar ve yarım kalan işi ya tamamlar ya geri alır. Böylece uzun süren bir disk taraması gerekmez.

2.1.6.3 Dosya sistemi türleri

Her işletim sistemi ailesi kendi dosya sistemini geliştirmiştir. Bir USB belleğin hangi bilgisayarlarda okunup yazılabileceği, hangi dosya sistemiyle **biçimlendirildiğine** (*format*) bağlıdır. Biçimlendirme, bir bölüme boş bir dosya sistemi yapısı yazmaktır ve bölümdeki eski içeriği siler.

Dosya sistemi	Platform	Günlükleme	Ne zaman kullanılır?
NTFS	Windows (macOS yalnızca okur)	Var	Windows sistem diski; büyük dosya, izin ve şifreleme gerektiğinde
FAT32	Hemen her aygıt ve işletim sistemi	Yok	Eski aygıtlar, kamera, basit bellek kartı; tek dosya en çok 4 GB
exFAT	Windows, macOS ve güncel Linux	Yok	Büyük dosyalı USB bellek ve SD kart; platformlar arası taşıma
ext4	Linux	Var	Linux sistem ve veri diski
APFS	macOS, iOS, iPadOS	Klasik günlük yok; yazınca kopyalama ile tutarlılık	Apple cihazlarının varsayılan dosya sistemi

Tablo 2.1.12: Yaygın dosya sistemleri.

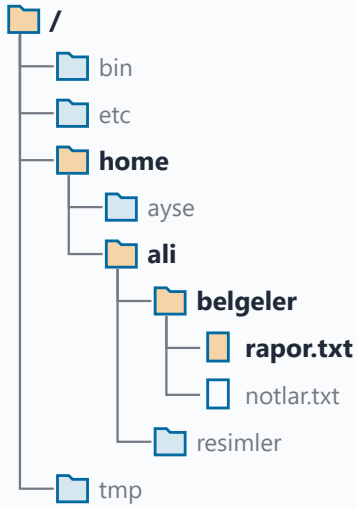
Tablo 2.1.12 beş yaygın dosya sistemini karşılaştırır. Tablodaki en pratik sınır FAT32'nin dosya boyutu sınırıdır. FAT32 bir dosyanın boyutunu 32 bitlik bir sayıyla tutar. 32 bitle yazılabilecek en büyük sayı $2^{32} - 1 = 4.294.967.295$'tir; bu yüzden tek bir dosya en çok 4 GiB'tan bir bayt eksik olabilir. APFS'nin "yazınca kopyalama" yöntemi ise değişen veriyi eski verinin üstüne değil, yeni bir yere yazar; işlem tamamlanınca işaretçiyi yeni yere çevirir. Kesinti olursa eski sürüm sağlam kalır.

Örnek 2.1.15

FAT32 biçimli bir USB belleğe 3 GB'lık bir film ve 5 GB'lık bir video (1 GB = 10^9 bayt) kopyalanacak. Sınır 4.294.967.295 bayttır. 3.000.000.000 bayt sınırın altında kaldığı için film kopyalanır; 5.000.000.000 bayt sınırı aştığı için video, bellekte yer olsa bile kopyalanamaz. Bellek hem Windows hem macOS'ta yazılacaksa exFAT seçilir, çünkü macOS NTFS'yi yalnızca okuyabilir. Biçimlendirme içeriği sileceği için önce yedek alınır.

2.1.6.4 Dosya sistemi ağacı ve yol

Klasörlerin içinde klasörler bulunabildiği için dosya sistemi bir ağaç biçimindedir. Ağacın en üstünde **kök** (*root*) klasör durur. Klasörler ağacın dalları, dosyalar yapraklarıdır.



Şekil 2.1.22: Bir Linux dosya ağacı. Kök klasörden (/) *rapor.txt*'ye giden yol vurgulanmıştır: / → home → ali → belgeler → rapor.txt.

Şekil 2.1.22 bir Linux sistemindeki ağacı gösterir. Kökte *bin*, *etc*, *home* ve *tmp* klasörleri vardır. Kullanıcıların kişisel klasörleri *home* altındadır. Vurgulu yol, bir dosyanın adresinin klasör adlarının art arda yazılmasıyla nasıl oluştuğunu gösterir.

Tanım 2.1.21

Mutlak yol (*absolute path*) kökten başlar ve dosyanın tam adresini verir: `/home/ali/belgeler/rapor.txt`. **Göreli yol** (*relative path*) **geçerli klasöre** (*current directory*), yani o anda içinde bulunulan klasöre göre yazılır. `..` bir üst klasörü, `.` geçerli klasörün kendisini gösterir.

Örnek 2.1.16

Şekil 2.1.22'deki ağaçta *rapor.txt*'ye üç farklı geçerli klasörden göreli yolla nasıl ulaşılır?

1. **Geçerli klasör /home/ali:** `belgeler/rapor.txt` . Bir klasör aşağı inilir.
2. **Geçerli klasör /home/ali/belgeler:** `rapor.txt` . Dosya zaten buradadır.
3. **Geçerli klasör /home/ayse:** `../ali/belgeler/rapor.txt` . Önce bir üst klasöre (home) çıkılır, sonra ali ve belgeler klasörlerine inilir.

Üç göreli yol da aynı dosyayı gösterir ve hepsinin mutlak karşılığı `/home/ali/belgeler/rapor.txt` 'dir.

Windows'ta aynı fikir iki farkla yazılır. Kök tek değildir; her sürücünün kendi kökü vardır ve sürücü bir harfle gösterilir (C:, D:). Klasör ayırıcı olarak eğik çizgi yerine ters eğik çizgi kullanılır: `C:\Users\Ali\Documents\rapor.txt`. Linux ve macOS'ta büyük-küçük harf ayrımı vardır (*Rapor.txt* ile *rapor.txt* iki ayrı dosyadır); Windows'ta varsayılan olarak yoktur.

2.1.6.5 Dosya açma ve okuma adım adım

Bir program bir dosyayı okurken arka planda çekirdekle bir dizi sistem çağrısı yapar: önce dosyayı açar, sonra parça parça okur, en sonda kapatır. Bu üç çağrı, dosya sistemi ile süreç yönetiminin nasıl birlikte çalıştığını gösterir.

Tanım 2.1.22

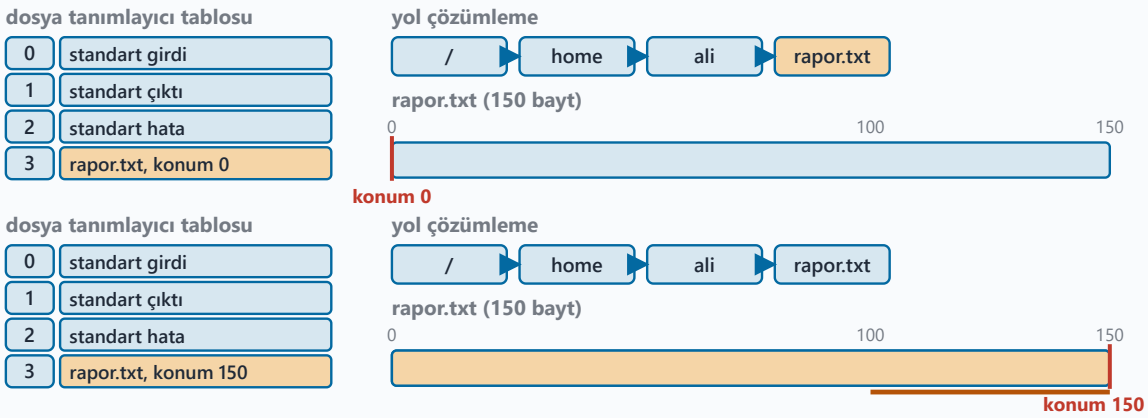
Dosya tanımlayıcısı (*file descriptor, fd*), çekirdeğin açık bir dosya için sürece verdiği küçük tam sayıdır. Süreç daha sonraki okuma ve yazma çağrılarında dosyayı adıyla değil bu numarayla anar. Her süreçte 0 standart girdi, 1 standart çıktı, 2 standart hata için baştan açıktır. Windows'ta aynı fikrin adı **tanıtıcıdır** (*handle*).

Örnek 2.1.17

Bir editör 150 baytlık `/home/ali/rapor.txt` dosyasını açıyor ve 100'er baytlık parçalarla sonuna kadar okuyor. Her çağrıda ne olur?

Çağrı	Çekirdeğin yaptığı	Dönen değer	Konum
`open`	Yolu parça parça çözer: / → home → ali → rapor.txt. Meta veriye bakar: boyut 150 bayt, okuma izni var. Açık dosya kaydı oluşturur.	fd = 3	0
`read` (100)	0–99 arası baytları okur	n = 100	100
`read` (100)	Yalnızca 50 bayt kaldı; 100–149 arasını okur	n = 50	150
`read` (100)	Okunacak bayt yok	n = 0 (dosya sonu)	150
`close`	Kaydı siler; 3 numarası yeniden boşalır	0 (başarılı)	—

Editör üç `read` çağrısıyla 100 + 50 = 150 bayt alır. Üçüncü çağrının 0 döndürmesi, programın okumayı bitirmesi gerektiğini bildirir.



Şekil 2.1.23: Örnek 2.1.17'nin iki anı. (a) `open` sonrası: dosya tanımlayıcı tablosunda 3 numaralı kayıt *rapor.txt*'yi gösterir, konum 0. (b) İkinci `read` sonrası: konum 150'ye, yani dosyanın sonuna gelmiştir.

Şekil 2.1.23 dosya tanımlayıcı tablosunu ve dosya içindeki konumu gösterir. Tabloyu adım adım izleyelim.

Yol çözümleme. Çekirdek `open` çağrısında yolu baştan sona tek tek çözer. Önce kök klasörü okur ve içinde *home* adını arar. Bulunca *home* klasörünü okur ve içinde *ali*'yi arar. Her klasör, içindeki adların ve bu adların diskteki yerlerinin bir listesidir. Sonunda *rapor.txt* bulunur ve meta verisine bakılır. Kullanıcının okuma izni yoksa `open` bir hata döndürür ve dosya açılmaz. Sonraki okumalarda yolun yeniden çözülmesi gerekmez; bu iş yalnızca açılışta yapılır.

Dosya tanımlayıcısı ve konum. Çekirdek sürece kullanılmayan *en küçük* numarayı verir; 0, 1 ve 2 dolu olduğu için ilk açılan dosya 3'ü, ikincisi 4'ü alır. Her `read` konumu okuduğu bayt sayısı kadar kendiliğinden ilerletir; uygulama blok numaralarını hiç bilmez.

Kapatma. `close` kaydı siler ve numara boşalır. Süreç bittiğinde açık kalan dosyalarını işletim sistemi kendisi kapatır. Ama uzun süre çalışan bir sunucu programı açtığı dosyaları kapatmazsa her sürece izin verilen tanımlayıcı sayısı bir gün tükenir ve program yeni dosya açamaz hâle gelir.

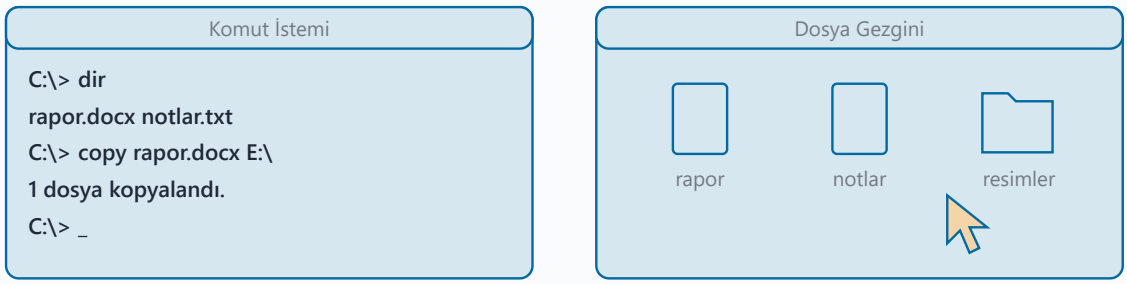
2.1.7 Kullanıcı arayüzleri

Kullanıcının işletim sistemiyle tanıştığı yer arayüzdür: pencereler, dokunmatik ekrandaki hareketler ya da bir komut satırı.

2.1.7.1 Arayüz türleri

Tanım 2.1.23

Grafik kullanıcı arayüzü (*graphical user interface, GUI*) pencere, simge, menü ve fareyle etkileşime dayanır. **Komut satırı arayüzü** (*command-line interface, CLI*) kullanıcının metin komutları yazdığı ve sonuçları metin olarak gördüğü arayüzdür. **Dokunmatik arayüz** dokunma, kaydırma ve parmak hareketleriyle çalışır.



Şekil 2.1.24: Aynı iş iki arayüzde. Solda Komut İstemi'nde `dir` ile klasör listelenir ve `copy rapor.docx E:\` ile dosya kopyalanır. Sağda Dosya Gezgini aynı klasörü simgelerle gösterir.

Şekil 2.1.24 aynı klasörü iki arayüzde gösterir. Grafik arayüzde dosyalar simge olarak görünür ve kopyalamak için dosya sürüklenir. Komut satırında ise klasörün içeriği `dir` komutuyla listelenir ve dosya `copy` komutuyla kopyalanır. İki yol da aynı sistem çağrılarına varır; fark yalnızca kullanıcının isteği nasıl ilettiğindedir.

Grafik arayüzün öğrenmesi kolaydır: menüler ne yapılabileceğini gösterir. Komut satırı ise hızlı, tekrarlanabilir ve otomasyona uygundur. 500 fotoğrafın adını fareyle tek tek değiştirmek saatler sürer; komut satırında kısa bir betik bu işi saniyeler içinde bitirir ve her gece kendiliğinden çalıştırılabilir. Sunucuların çoğunda bu yüzden grafik arayüz hiç kurulmaz.

Her arayüz **erişilebilirlik** araçlarıyla da desteklenir: ekran okuyucu metni sesli okur, büyüteç ekranı büyütür, yüksek kontrast teması metni belirginleştirir.

2.1.7.2 Komut satırı temelleri

Komut satırında her komut bir satırdır: önce komutun adı, ardından boşlukla ayrılmış *argümanlar* gelir. `copy a.txt b.txt` komutunda `copy` komutun adı, `a.txt` ve `b.txt` ise argümanlardır. Komutlar her zaman geçerli klasörde çalışır; göreli yollar o klasöre göre yorumlanır. Linux ve macOS'taki kabuk ile Windows'taki Komut İstemi (cmd) aynı işler için farklı komut adları kullanır.

İş	Linux / macOS	Windows (cmd)
Geçerli klasörü göster	`pwd`	`cd` (parametresiz)

İş	Linux / macOS	Windows (cmd)
Klasör içeriğini listele	`ls`	`dir`
Klasör değiştir	`cd klasör`	`cd klasör`
Klasör oluştur	`mkdir yeni`	`mkdir yeni`
Dosya kopyala	`cp a.txt b.txt`	`copy a.txt b.txt`
Dosya sil	`rm a.txt`	`del a.txt`

Tablo 2.1.13: Temel komutların Linux/macOS ve Windows (cmd) karşılıkları.

Tablo 2.1.13 en sık kullanılan altı işi iki dünyada karşılaştırır. `pwd`, adını "print working directory" (geçerli klasörü yazdır) sözünden alır. `cd`, "change directory" (klasör değiştir) demektir. Komutları ve yolları birlikte kullanınca dosya ağacında gezinmek mümkün olur.

Örnek 2.1.18

Linux'ta geçerli klasör `/home/ali`. Ağaçta `/home/ali` altında *belgeler* ve *resimler*, `/home/ayse` altında *notlar.txt* vardır. Aşağıdaki komutlar sırayla çalıştırılınca geçerli klasör nasıl değişir?

Komut	Açıklama	Sonraki geçerli klasör	Çıktı
`cd belgeler`	Göreli yol: bir klasör aşağı	/home/ali/belgeler	—
`cd ../resimler`	Bir üst (ali), sonra resimler	/home/ali/resimler	—
`cd ../../ayse`	İki üst (ali, home), sonra ayse	/home/ayse	—
`ls`	İçeriği listeler	/home/ayse	notlar.txt
`cd /tmp`	"/" ile başlar: mutlak yol	/tmp	—
`pwd`	Geçerli klasörü yazar	/tmp	/tmp

Her `..` bir klasör yukarı çıkar. `/` ile başlayan yol ise geçerli klasör ne olursa olsun kökten başlar.

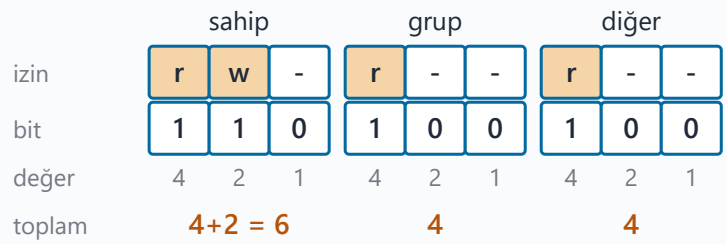
Sık yapılan hata

`rm` ve `del` komutlarıyla silinen dosyanın Geri Dönüşüm Kutusu'na gittiğini sanmak. Bu komutlar dosyayı doğrudan siler; geri almanın kolay bir yolu yoktur. Komutu çalıştırmadan önce geçerli klasörün ve dosya adının doğru olduğundan emin olmak gerekir.

2.1.7.3 Dosya izinleri ve chmod

Linux ve macOS'ta her dosyanın izinleri üç kullanıcı grubu için ayrı ayrı tutulur: dosyanın **sahibi**, sahibin dahil olduğu **grup** ve **diğerleri**. Her grup için üç izin vardır: **r** (*read*, oku), **w** (*write*, yaz) ve **x** (*execute*, çalıştır). Böylece bir dosyanın izinleri dokuz karakterle yazılır: `rw-r--r--`. Bir harf yerine kısa çizgi varsa o izin yoktur.

İzinler `chmod` komutuyla değiştirilir ve çoğunlukla üç basamaklı bir sayıyla verilir. Her basamak bir grubu gösterir. Basamağın değeri, o gruptaki izinlerin ağırlıkları toplanarak bulunur: r = 4, w = 2, x = 1.



Şekil 2.1.25: `chmod 644` : sahip rw- (110 → 4 + 2 = 6), grup r-- (100 → 4), diğer r-- (100 → 4).

Şekil 2.1.25 644'ün nereden geldiğini gösterir. Ağırlıklar 4, 2 ve 1 olduğu için her grubun üç izni aslında üç bitlik bir ikili sayıdır. Sahibin izni rw- ikili 110'dur ve 110_2 = 6'dır. r-- ise 100_2 = 4'tür. Üç bitle 0'dan 7'ye kadar sekiz değer yazılabildiği için her basamak 0–7 arasındadır.

Basamak	İkili	İzin	Anlamı
7	111	rwx	Oku, yaz, çalıştır
6	110	rw-	Oku, yaz
5	101	r-x	Oku, çalıştır
4	100	r--	Yalnızca oku
0	000	---	Hiçbir izin yok

Tablo 2.1.14: Sık kullanılan izin basamakları.

Örnek 2.1.19

(a) `chmod 755 yedekle.sh` komutundan sonra izinler nedir? (b) Bir günlük dosyasını yalnızca sahibi okuyup yazabilsin, başka kimse erişemesin. Hangi sayı verilmelidir? (c) `rw-rw-r--` izni hangi sayıyla yazılır?

1. **(a)** 7 = 4 + 2 + 1 → rwx; 5 = 4 + 1 → r-x; 5 → r-x. Sonuç `rwxr-xr-x` : sahip her şeyi yapar, grup ve diğerleri okuyup çalıştırabilir ama değiştiremez.
2. **(b)** Sahip: rw- = 4 + 2 = 6. Grup: --- = 0. Diğer: --- = 0. Sayı `600` 'dür.

3. **(c)** rw- = 6, rw- = 6, r-- = 4. Sayı `664` 'tür.

Üç grup ve her grupta üç izin, toplam dokuz bit eder; bu yüzden $2^9 = 512$ farklı izin bileşimi vardır. Her basamağın 8 değeri olduğu için aynı sayı $8 \times 8 \times 8 = 512$ olarak da bulunur.

Sık yapılan hata

İzin basamaklarını onluk sayı gibi okumak: "644, 600'den büyüktür, o hâlde daha çok izin verir." Her basamak ayrı bir grubu gösterir. 644'te grup ve diğerleri okuyabilir; 600'de okuyamaz. Ayrıca bir sorun çözülmüyor diye `chmod 777` vermek, dosyayı herkesin değiştirebileceği hâle getirir ve ciddi bir güvenlik açığıdır.

2.1.8 Kullanıcı ve yetki yönetimi

Bir bilgisayarı birden fazla kişi kullanabilir; bir sunucuya binlerce kişi bağlanabilir. İşletim sistemi her işlemde iki soru sorar: işlemi yapan kim ve bu kişinin bu işlemi yapmaya hakkı var mı? Bu iki soru iki ayrı aşamada cevaplanır.

Tanım 2.1.24

Kimlik doğrulama (*authentication*) kullanıcının iddia ettiği kişi olduğunu denetler: "Kim?" Parola, PIN ya da parmak izi gibi biyometrik veriyle yapılır. **Yetkilendirme** (*authorization*) kimliği doğrulanmış kullanıcının neler yapabileceğini belirler: "Ne yapabilir?" İzinler ve hesap türü bu aşamada denetlenir.



Şekil 2.1.26: Kimlik doğrulama ve yetkilendirme akışı. Standart kullanıcı kendi dosyalarını açabilir (✓), sistem ayarını değiştiremez (✗). İzin verilen ve reddedilen erişimler olay kaydına yazılır.

Şekil 2.1.26 iki aşamayı sırayla gösterir. Kullanıcı önce kimliğini kanıtlar. Ardından her işlem için ayrıca yetki denetimi yapılır. Örnekte aynı kullanıcı kendi dosyasını açabilir ama sistem ayarını değiştiremez. Her iki sonuç da olay kaydına yazılır.

Hesaplar genellikle iki türdür. **Standart hesap** günlük kullanım içindir ve sistemi etkileyen değişiklikleri sınırlar. **Yönetici hesabı** program kurmak, sistem ayarını değiştirmek ya da başka kullanıcıların dosyalarına erişmek gibi sistem genelindeki değişiklikleri yapabilir. Windows'ta yönetici hesabı bile bu tür bir işlemden önce onay penceresi gösterir; Linux'ta aynı işlem için komutun başına `sudo` yazılır ve parola istenir.

Tanım 2.1.25

En az ayrıcalık ilkesi (*principle of least privilege*): her kullanıcıya ve her programa yalnızca işini yapmak için gereken kadar yetki verilir; fazlası verilmez.

Bu ilkenin değeri bir saldırı anında ortaya çıkar. Standart hesapla açılmış bir e-posta ekindeki zararlı yazılım, yalnızca o kullanıcının yetkileriyle çalışır. Kullanıcının kendi dosyalarına zarar verebilir, ama sistem dosyalarını değiştiremez ve öteki kullanıcılara ulaşamaz. Aynı ek her zaman yönetici hesabıyla çalışan birinin bilgisayarında açılsaydı bütün sistemi ele geçirebilirdi.

Kayıt tutma, önemli olayların sonradan incelenebilmesini sağlar: gece yarısı yüzlerce başarısız parola denemesi kayıtta görülürse yönetici saldırıdan haberdar olur.

Örnek 2.1.20

Standart hesaplı bir öğrenci parolasıyla oturum açıyor, kendi ödev dosyasını açıyor, sonra sistem saatini değiştirmeye çalışıyor ve engelleniyor. Her adım hangi aşamadadır?

1. **Oturum açma:** parola doğrudur → kimlik doğrulama başarılı.
2. **Ödev dosyasını açma:** dosyanın sahibi öğrencidir ve okuma izni vardır → yetkilendirme başarılı (✓).
3. **Saati değiştirme:** sistem genelindeki bir ayar standart hesaba kapalıdır → yetkilendirme reddedildi (✗); olay kaydına yazılır.

Engel kimlik doğrulamada değil, yetkilendirmededir. Öğrenci kim olduğunu kanıtlamıştır; yalnızca o işlem için yetkisi yoktur.

2.1.9 İşletim sistemi türleri

Bir akıllı telefon, bir web sunucusu ve bir otomobilin fren birimi aynı temel görevleri yapan işletim sistemleri kullanır, ama öncelikleri farklıdır: telefonda pil ömrü, sunucuda kesintisiz çalışma, fren biriminde zamanında tepki öne çıkar.

2.1.9.1 Kişisel bilgisayar sistemleri

Masaüstü ve dizüstü bilgisayarlarda dört aile öne çıkar.

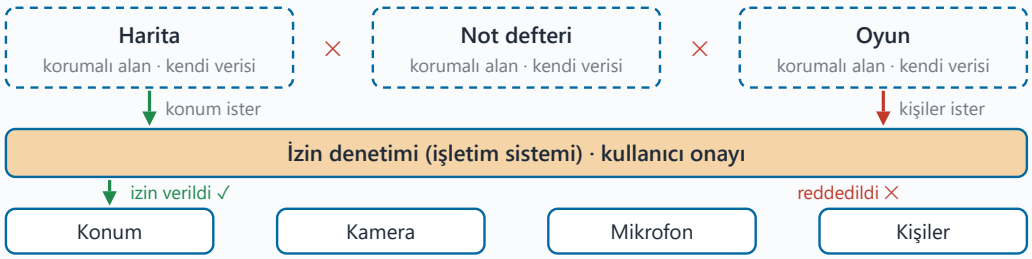
- **Windows:** donanım ve uygulama yelpazesi en geniş olan sistemdir. Çok sayıda üreticinin bilgisayarında çalışır; oyunların ve iş yazılımlarının çoğu önce Windows için yazılır.
- **macOS:** yalnızca Apple bilgisayarlarında çalışır. Donanımı ve yazılımı aynı şirket ürettiği için ikisi sıkı biçimde bütünleşiktir.
- **Linux dağıtımları:** farklı masaüstü ortamları ve kullanım amaçlarına göre özelleştirilebilir. Çoğu ücretsizdir ve eski donanımda da iyi çalışabilir.
- **ChromeOS:** web ve bulut tabanlı kullanıma odaklanır. Uygulamaların ve verinin çoğu internette durduğu için düşük donanımlı cihazlarda da hızlı açılır; çevrim dışı kullanımı ise sınırlıdır.

2.1.9.2 Mobil işletim sistemleri

Mobil dünyada iki sistem vardır. **Android**, farklı üreticilerin cihazlarında çalışan geniş bir ekosistemdir. **iOS ve iPadOS**, Apple'ın telefon ve tabletleri için geliştirdiği bütünleşik sistemlerdir. Mobil sistemlerin öncelikleri masaüstünden ayrılır: dokunmatik arayüz, pil ömrünü uzatan enerji yönetimi, konum ve ivme gibi sensörler ve sürekli değişen kablosuz bağlantılar. Arka planda çalışan uygulamalar sıkı biçimde sınırlanır, çünkü her çalışan süreç pil harcar.

Tanım 2.1.26

Korumalı alan (*sandbox*), bir uygulamanın yalnızca kendi verisine erişebildiği yalıtılmış çalışma ortamıdır. Konum, kamera, mikrofon ve kişiler gibi kaynaklara erişim için işletim sisteminin izin denetiminden ve çoğu zaman kullanıcının onayından geçmek gerekir.

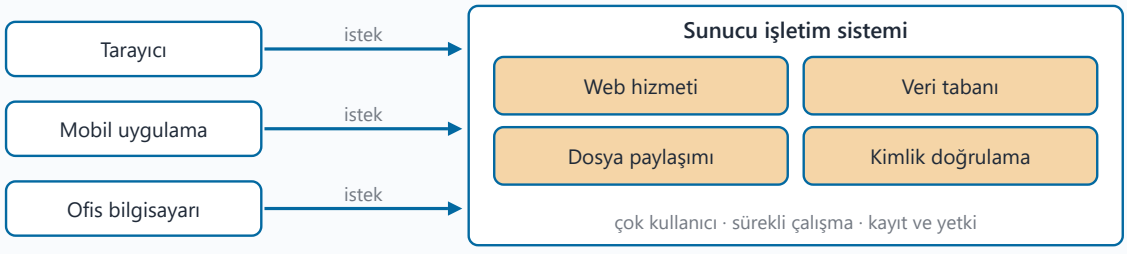


Şekil 2.1.27: Harita, not defteri ve oyun uygulamaları kendi korumalı alanlarında çalışır ve birbirinin verisine erişemez (✕). Harita konum, oyun kişiler listesi ister; istekler işletim sisteminin izin denetiminden ve kullanıcı onayından geçer.

Şekil 2.1.27 uygulama yalıtımını gösterir. Her uygulama kendi kutusunda yaşar. Not defteri uygulaması harita uygulamasının kayıtlı adreslerini okuyamaz. Bir uygulama telefonun ortak kaynaklarına erişmek istediğinde işletim sistemi araya girer ve kullanıcıya sorar. Haritanın konum istemesi beklenen bir istektir ve kullanıcı izin verir. Bir oyunun kişiler listesini istemesi ise şüphelidir ve kullanıcı reddedebilir. Böylece kötü niyetli bir uygulama, kullanıcı fark etmeden kişisel verileri toplayamaz.

2.1.9.3 Sunucu işletim sistemleri

Sunucu, ağ üzerinden başka bilgisayarlara hizmet veren bilgisayardır. Sunucu işletim sistemleri bu işe göre biçimlenir.



Şekil 2.1.28: Tarayıcı, mobil uygulama ve ofis bilgisayarı gibi istemcilerin istekleri sunucu işletim sistemindeki web, veri tabanı, dosya paylaşımı ve kimlik doğrulama hizmetlerine gider. Sunucu çok kullanıcılı ve sürekli çalışır; kayıt ve yetki denetimi tutar.

Şekil 2.1.28 bir sunucunun aynı anda birçok istemciye hizmet verdiğini gösterir. Sunucu işletim sistemlerinin öncelikleri şunlardır:

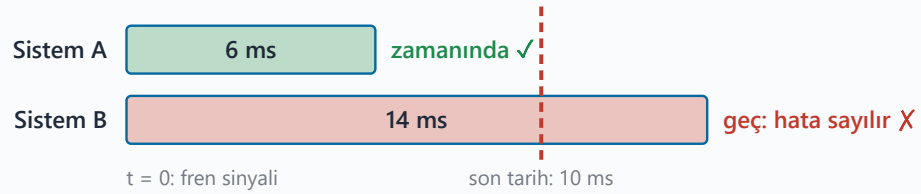
- **Çok kullanıcılı çalışma:** çok sayıda kullanıcının ve hizmetin aynı anda yönetilmesi.
- **Ağ hizmetleri:** web, veri tabanı, dosya paylaşımı ve kimlik doğrulama hizmetleri.
- **Süreklilik:** aylarca kesintisiz çalışma ve uzaktan yönetim. Bir sunucu yeniden başlatıldığında ona bağlı bütün hizmetler kesilir.
- **Güvenlik:** yetki, kayıt, güncelleme ve ağ denetimi.
- **Otomasyon:** komut satırı ve yapılandırma araçlarıyla tekrarlanabilir yönetim. Yüz sunucuyu tek tek elle ayarlamak yerine aynı betik hepsinde çalıştırılır.

Yaygın örnekler Linux dağıtımları, Windows Server ve Unix türevleridir. İnternetteki web sunucularının büyük çoğunluğu Linux kullanır.

2.1.9.4 Gömülü ve gerçek zamanlı sistemler

Tanım 2.1.27

Gömülü işletim sistemi, belirli bir cihaz görevi için sınırlı işlemci, bellek ve enerjiyle çalışan işletim sistemidir; yönlendiricilerde, otomobil denetim birimlerinde ve tıbbi cihazlarda bulunur. **Gerçek zamanlı işletim sistemi** (*real-time operating system*) bir işlemin yalnızca doğru sonucu değil, sonucu belirlenen süre içinde, yani **son tarihten** (*deadline*) önce üretmesini hedefler.



Şekil 2.1.29: Gerçek zamanlı sistemde son tarih. t = 0'da fren sinyali gelir ve son tarih 10 ms'dir. Sistem A 6 ms'de tepki verir (zamanında ✓); sistem B 14 ms'de tepki verir (geç: hata sayılır ✗).

Şekil 2.1.29 gerçek zamanlı sistemin temel kuralını gösterir: geç gelen doğru sonuç da yanlıştır. İki sistem de doğru fren komutunu üretir, ama B son tarihi kaçırdığı için başarısız sayılır. Masaüstü bir işletim sisteminde bir pencerenin 4 ms geç açılmasını kimse fark etmez. Bir fren sisteminde ise aynı gecikme mesafe demektir.

Örnek 2.1.21

Saatte 100 km hızla giden bir otomobilde fren sinyali t = 0'da geliyor. Sistem A 6 ms, sistem B 14 ms sonra tepki veriyor. Otomobil bu sürelerde ne kadar yol alır?

1. Hız metre/saniyeye çevrilir: 100 km/sa = 100.000 m ÷ 3.600 s ≈ 27,78 m/s.

2. Sistem A: 27,78 × 0,006 ≈ 0,17 m.
3. Son tarih 10 ms: 27,78 × 0,010 ≈ 0,28 m.
4. Sistem B: 27,78 × 0,014 ≈ 0,39 m.

B son tarihi 4 ms aşar; fren başlamadan önce otomobil, A'ya göre yaklaşık 0,39 − 0,17 = 0,22 m daha fazla yol alır. Gerçek zamanlı bir sistem, en kötü durumda bile 10 ms'yi aşmayacağını garanti etmek zorundadır.

Gerçek zamanlı sistemlerde ortalama olarak hızlı olmak yetmez; en kötü durumdaki tepki süresi de bilinmeli ve sınırlı olmalıdır. Bu yüzden gerçek zamanlı çizelgeleyiciler acil bir iş gelince daha az önemli işi hemen keser.

Sık yapılan hata

"Gerçek zamanlı" sözünü "çok hızlı" anlamında kullanmak. Gerçek zamanlı bir sistem çok hızlı olmak zorunda değildir; zamanında olmak zorundadır. Son tarihi 1 saniye olan ve onu her zaman tutan bir sistem gerçek zamanlıdır. Ortalama 1 ms'de tepki veren ama arada bir 50 ms gecikebilen bir sistem ise değildir.

2.1.9.5 İşletim sistemi seçimi

Bir okul laboratuvarı, bir yazılım şirketi ya da bir fabrika için işletim sistemi seçerken tek bir ölçüt yetmez. "En iyi işletim sistemi" diye bir şey yoktur; gereksinimlere en uygun olan vardır.

Gereksinim	Değerlendirilecek ölçüt
Uygulama	Gerekli yazılımların ve sürümlerinin bulunması
Donanım	Sürücü ve işlemci mimarisi desteği
Güvenlik	Güncelleme süresi, yetki modeli ve yönetim araçları
Kullanılabilirlik	Arayüz, erişilebilirlik ve öğrenme gereksinimi
Maliyet	Lisans, destek, eğitim ve bakımın toplamı

Tablo 2.1.15: İşletim sistemi seçiminde değerlendirilecek ölçütler.

2.1.10 Lisanslama

Bir yazılım satın alındığında çoğu zaman yazılımın kendisi değil, onu belirli koşullarla kullanma hakkı alınır. Bu koşulları lisans belirler.

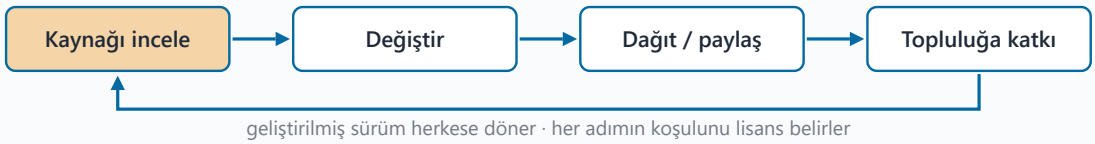
Tanım 2.1.28

Lisans (*license*), yazılımın sahibinin kullanıcıya hangi hakları hangi koşullarla verdiğini belirleyen hukuki metindir. **Kaynak kod** (*source code*), programın insanın okuyup değiştirebileceği, bir programlama dilinde yazılmış hâlidir.

2.1.10.1 Açık kaynak yazılım

Açık kaynak (*open source*) yazılımda kaynak kod herkese açıktır ve lisans dört konuda hak tanır.

- **Kaynak kodu:** lisans koşulları çerçevesinde incelenebilir.
- **Değiştirme:** lisans izin veriyorsa ihtiyaca göre uyarlanabilir.
- **Dağıtım:** özgün ya da değiştirilmiş sürüm paylaşılabilir; koşullarını lisans belirler.
- **Topluluk:** geliştirme ve destek; bireyler, kurumlar ve şirketler birlikte yürütebilir.



Şekil 2.1.30: Açık kaynak döngüsü: kaynağı incele, değiştir, dağıt ve topluluğa katkıda bulun. Geliştirilmiş sürüm herkese döner; her adımın koşulunu lisans belirler.

Şekil 2.1.30 açık kaynağın nasıl büyüdüğünü bir döngü olarak gösterir. Bir geliştirici kodu inceler, bir hatayı düzeltir ya da bir özellik ekler ve değişikliği geri gönderir. Değişiklik kabul edilirse bir sonraki sürümle herkese ulaşır. Linux çekirdeği bu yolla binlerce geliştiricinin katkısıyla büyümüştür.

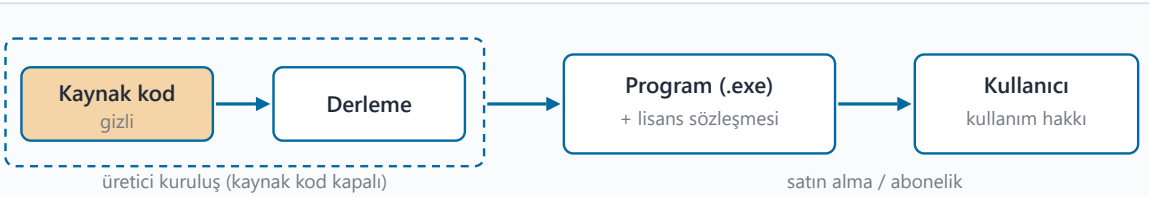
Açık kaynak lisansları iki ana gruba ayrılır. **GPL** (*GNU General Public License*) gibi lisanslar, değiştirilmiş sürüm dağıtılırsa onun da aynı lisansla ve kaynak koduyla birlikte dağıtılmasını şart koşar; bu yaklaşıma *copyleft* denir. **MIT** ve **Apache** gibi serbest lisanslar ise yazılımın kapalı bir ürünün içinde bile kullanılmasına izin verir; genellikle yalnızca özgün lisans metninin ve telif bildiriminin korunmasını ister.

Sık yapılan hata

"Açık kaynak" ile "ücretsiz" ve "desteksiz"i aynı saymak. Açık kaynak bir yazılımın lisansı genellikle ücretsizdir, ama birçok şirket açık kaynak ürünler için ücretli destek, eğitim ve bakım hizmeti satar. Tersine, ücretsiz dağıtılan her program da açık kaynak değildir: ücretsiz bir oyunun kaynak kodu kapalı olabilir. Belirleyici olan fiyat değil, lisansın verdiği haklardır.

2.1.10.2 Ticari ve kapalı kaynak yazılım

Kapalı kaynak (*closed source, proprietary*) yazılımda kaynak kodu geliştirici kuruluşta kalır. Kullanıcıya yalnızca derlenmiş, çalıştırılabilir program verilir.



Şekil 2.1.31: Kapalı kaynak modeli. Üretici kaynak kodu gizli tutar ve derler; derlenmiş program (.exe) lisans sözleşmesiyle kullanıcıya verilir. Kullanıcı satın alma ya da abonelikle yalnızca kullanım hakkı edinir.

Şekil 2.1.31 bu modeli gösterir. Ticari yazılımın dört özelliği öne çıkar.

- **Kullanım hakkı:** yazılım satın alınmaz; çoğunlukla belirli koşullarla bir kullanım lisansı alınır. Koşullar **son kullanıcı lisans sözleşmesinde** (*end-user license agreement, EULA*) yazar: kaç cihaza kurulabileceği, ticari işte kullanılıp kullanılamayacağı, tersine mühendisliğin yasak olup olmadığı.
- **Kaynak kodu:** genellikle geliştirici kuruluşta kapalı tutulur. Kullanıcı bir hatayı kendisi düzeltemez; üreticinin düzeltmesini bekler.
- **Destek:** resmî destek ve hizmet düzeyi sözleşmeleri sunulabilir. Örneğin bir sorun bildirildiğinde kaç saat içinde cevap verileceği sözleşmeyle güvenceye alınır.
- **Maliyet:** lisans, abonelik, bakım ve yükseltme ücretleri olabilir.

2.1.10.3 Lisans modellerinin karşılaştırması

Ölçüt	Açık kaynak	Ticari / kapalı kaynak
Kaynak erişimi	Lisansla açık	Genellikle kapalı
Özelleştirme	Lisans kapsamında mümkün	Üreticinin sunduğu seçeneklerle sınırlı
Destek	Topluluk ya da ücretli sağlayıcı	Üretici ya da yetkili iş ortağı
Maliyet	Lisans ücretsiz olabilir; işletme maliyeti sürer	Lisans ya da abonelik gerekebilir
Lisans örneği	GPL, MIT, Apache	Son kullanıcı lisans sözleşmesi
Ürün örneği	Linux, Firefox, LibreOffice	Windows, Microsoft Office, Adobe Photoshop

Tablo 2.1.16: Açık kaynak ve ticari/kapalı kaynak lisans modellerinin karşılaştırması.

Tablo 2.1.16 iki modeli altı ölçütle karşılaştırır. İkisinin de güçlü yanları vardır. Açık kaynak, kodun herkesçe incelenebilmesi sayesinde şeffaftır ve bir üreticiye bağımlılığı azaltır. Kapalı kaynak ise tek bir sorumlu üretici ve sözleşmeli destek sunar. Birçok kurum ikisini birlikte kullanır: sunucularda Linux, masaüstünde Windows ve Office.

Örnek 2.1.22

Bir öğrenci topluluğu, MIT lisanslı bir kütüphaneyi ve GPL lisanslı başka bir kütüphaneyi kullanarak bir program yazıyor ve programı dağıtmak istiyor. Hangi yükümlülükler doğar?

1. **MIT kütüphanesi:** program istenen lisansla dağıtılabilir. Yalnızca kütüphanenin telif bildirimi ve lisans metni dağıtıma eklenmelidir.
2. **GPL kütüphanesi:** GPL kodu içeren bir program dağıtılıyorsa programın tamamı GPL ile ve kaynak koduyla birlikte sunulmalıdır.
3. **Sonuç:** topluluk programın kaynak kodunu kapalı tutmak isterse GPL kütüphanesini kullanamaz; yerine serbest lisanslı bir seçenek aramalıdır.

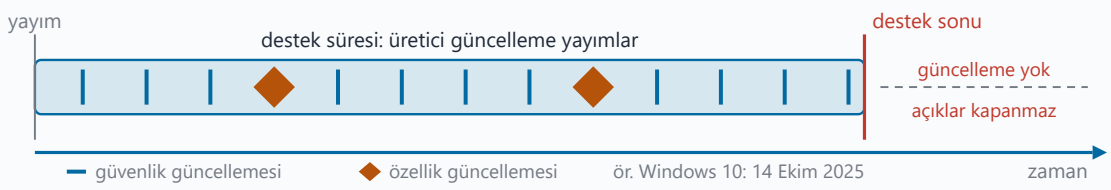
"Açık kaynak" olmak yükümlülük doğurmadığı anlamına gelmez. Hakları ve yükümlülükleri her durumda lisans metni belirler.

2.1.11 Güncelleme ve destek

Bir işletim sistemi kurulduğu günkü hâliyle kalmaz. Yeni güvenlik açıkları bulunur, donanımlar değişir, kullanıcılar yeni özellikler ister. Üretici bunlara güncellemelerle karşılık verir.

Kavram	Anlamı
Güvenlik güncellemesi	Bilinen açıkları kapatır
Özellik güncellemesi	Yeni işlevler ve arayüz değişiklikleri getirir
Sürücü güncellemesi	Donanım uyumluluğunu ya da kararlılığını iyileştirebilir
Destek süresi	Üreticinin güncelleme sağladığı zaman aralığı
Yedekleme	Büyük bir değişiklikten önce geri dönüş imkânı sağlar
Test	Kurumsal ortamlarda güncellemeler yaygın dağıtımdan önce denenir

Tablo 2.1.17: Güncelleme türleri ve bakımla ilgili kavramlar.



Şekil 2.1.32: Destek süresi boyunca düzenli güvenlik güncellemeleri ve ara sıra özellik güncellemeleri gelir. Destek sonundan sonra güncelleme gelmez ve yeni bulunan açıklar kapanmaz. Örnek: Windows 10'un desteği 14 Ekim 2025'te sona erdi.

Şekil 2.1.32 bir sürümün yaşam süresini gösterir. Güvenlik güncellemeleri sık ve düzenlidir; Microsoft bunları her ayın ikinci salı günü yayımlar. Özellik güncellemeleri daha seyrektir.

Destek sonu, bilgisayarın çalışmayı bırakacağı anlamına gelmez. Windows 10 yüklü bir bilgisayar 14 Ekim 2025'ten sonra da açılır, ama o tarihten sonra bulunan güvenlik açıkları kapatılmaz. Desteksiz bir sistem her geçen ay daha savunmasız hâle gelir. Büyük bir güncellemeden önce yedek alınır; kurumlar güncellemeyi önce birkaç test bilgisayarında dener, sonra aşamalı olarak dağıtır.

2.1.12 Sanallaştırma ve konteynerler

Aynı bilgisayarda hem Windows hem Linux denemek ya da tek bir güçlü sunucuyu onlarca müşteriye paylaştırmak için iki teknoloji kullanılır: sanal makineler ve konteynerler. İkisi de yalıtım sağlar, ama yalıtımı farklı katmanlarda kurar.

2.1.12.1 Sanal makine ve hipervizör

Tanım 2.1.29

Sanallaştırma (*virtualization*), tek bir fiziksel bilgisayarın kaynaklarını birden fazla yalıtılmış ortama paylaştırmaktır. **Sanal makine** (*virtual machine, VM*), fiziksel bir bilgisayarda yazılımla oluşturulmuş ve içinde tam bir işletim sistemi çalışan sanal bilgisayardır. Sanal makinede çalışan sisteme **misafir işletim sistemi** (*guest operating system*) denir. **Hipervizör** (*hypervisor*), sanal makinelere işlemci, bellek ve aygıt kaynaklarını sağlayan ve onları birbirinden yalıtan yazılımdır.

Hipervizör, işletim sisteminin süreçlere yaptığını bir kat yukarıda işletim sistemlerine yapar. Her misafir sistem kendine ait bir işlemcisi, belleği ve diski olduğunu sanır. Gerçekte hipervizör fiziksel kaynakları paylaştırır: 32 GB RAM'li bir sunucuda dört sanal makineye 8'er GB verilebilir. Bir sanal makine çökerse ya da bir zararlı yazılıma yakalanırsa ötekiler etkilenmez.

2.1.12.2 Konteyner

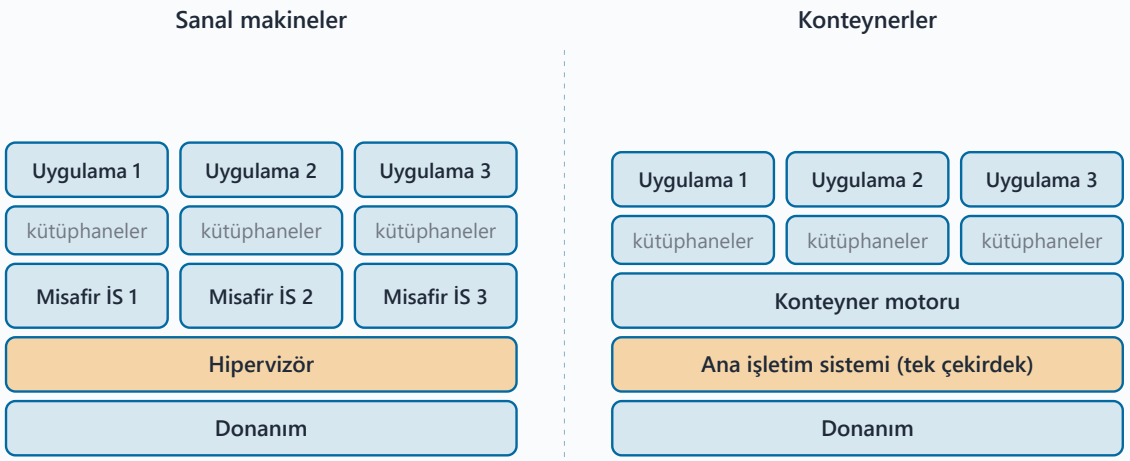
Tanım 2.1.30

Konteyner (*container*), bir uygulamayı ve onun **bağımlılıklarını** (*dependency*), yani çalışması için gereken kütüphane ve ayarları, süreç düzeyinde yalıtan pakettir. Konteynerler ana işletim sisteminin çekirdeğini paylaşır. Konteynerleri başlatan ve yöneten yazılıma **konteyner motoru** (*container engine*) denir; en bilinen örneği Docker'dır.

Konteyner, “benim bilgisayarımda çalışıyordu” sorununa bir çözümdür. Uygulama, gereken kütüphane sürümleriyle birlikte paketlenir; paket nerede çalıştırılırsa çalıştırılsın içindeki ortam aynıdır. Konteynerlerin kendi çekirdekleri olmadığı için saniyeler içinde başlarlar ve az bellek harcarlar.

Sanallaştırma ve konteynerler dört alanda yaygındır: farklı sistemlerde yazılım **testi**, çok sayıda az kullanılan sunucunun tek bir fiziksel makinede **birleştirilmesi**, **bulut** hizmetleri ve uygulamaların her ortama aynı biçimde **taşınabilir** olarak dağıtılması.

2.1.12.3 Sanal makine ile konteynerin karşılaştırması



Şekil 2.1.33: Solda sanal makineler: hipervizörün üstünde her uygulamanın kendi misafir işletim sistemi ve kütüphaneleri vardır. Sağda konteynerler: tek çekirdekli ana işletim sisteminin üstünde konteyner motoru çalışır; her konteyner yalnızca kütüphanelerini ve uygulamasını taşır.

Şekil 2.1.33 iki yaklaşımın katmanlarını yan yana gösterir. Fark, yalıtımın hangi katmanda kurulduğundadır.

Özellik	Sanal makine	Konteyner
Ne taşır?	Tam bir misafir işletim sistemi ve uygulama	Yalnızca uygulama ve bağımlılıkları
Çekirdek	Her sanal makinenin kendi çekirdeği	Ana işletim sisteminin çekirdeği ortak
Yalıtım	Güçlü	Daha zayıf
Kaynak kullanımı	Daha çok bellek, daha yavaş açılış	Hafif, hızlı açılış
Farklı işletim sistemi	Çalıştırabilir (Linux üstünde Windows)	Ana sistemle aynı çekirdeği gerektirir

Tablo 2.1.18: Sanal makine ile konteynerin karşılaştırması.

Örnek 2.1.23

Linux çalışan bir sunucuda iki ihtiyaç var: (a) bir web uygulamasının aynı Linux sürümünde çalışan 20 kopyası, (b) yalnızca Windows'ta çalışan bir muhasebe programı. Her ihtiyaç için hangi teknoloji uygundur?

1. **(a) Web uygulaması:** uygulama Linux çekirdeğinde çalışır; ana sistemin çekirdeğini paylaşabilir. 20 kopya için 20 tam işletim sistemi kurmak bellek israfı olur. Konteyner uygundur: her kopya hafiftir ve saniyeler içinde başlar.
2. **(b) Muhasebe programı:** program Windows çekirdeği ister. Konteynerler ana sistemin Linux çekirdeğini paylaştığı için bu programı çalıştıramaz. Sanal makine uygundur: hipervizör üstünde tam bir Windows misafir sistemi kurulur.

2.1.13 Özet

- **Yazılım** programların, verilerin ve belgelerin bütünüdür. Sistem yazılımı donanımı yönetir, uygulama yazılımı kullanıcının işini yapar, programlama araçları yazılım üretir, ürün yazılımı cihaza gömülüdür. Her katman altındakini **soyutlar**.
- **İşletim sistemi** süreç, bellek, dosya, aygıt ve kullanıcıları yönetir, ağ hizmeti sunar. **Önyükleme**: güç → UEFI → POST → önyükleyici → çekirdek → oturum açma. **Güvenli Önyükleme** imzaları denetler, **TPM** ölçümleri saklar; MBR 2 TB ve 4 birincil bölümle sınırlıdır, GPT değildir.
- **Çekirdek** ayrıcalıklı modda çalışır; uygulamalar ondan **sistem çağrısıyla** hizmet ister. Çekirdekler monolitik, mikro ya da hibrittir.
- **Süreç** Yeni, Hazır, Çalışıyor, Bekliyor ve Sonlandı durumlarından geçer; bekleyen süreç önce Hazır olur. **Çoklu programlama** G/Ç beklerken başka süreci çalıştırır. **Döner sıralı çizelgeleme** her sürece en çok bir zaman dilimi verir; SJF ortalama beklemeyi en aza indirir.
- **Kilitlenme** için karşılıklı dışlama, tut ve bekle, ön alınamazlık ve döngüsel beklemenin hepsi gerekir.
- Taban/limit modeli belleği korur; **takas** yer açar ama yavaştır. **Sanal bellekte** adres sayfa ve bağıl konuma ayrılır; sayfa RAM'de değilse **sayfa hatası** olur. **LRU** son kullanımı, **FIFO** girişi en eski sayfayı çıkarır.
- **Sürücüler** ortak istekleri donanıma çevirir; aygıtlar kesmeyle haber verir. **Dosya sistemi** adları, meta veriyi ve blokları yönetir; açılan dosyaya bir **dosya tanımlayıcısı** verilir.
- Komut satırı otomasyona uygundur; `chmod` izinleri r = 4, w = 2, x = 1 ağırlıklarıyla yazar. **Kimlik doğrulama** "kim?", **yetkilendirme** "ne yapabilir?" sorusunu cevaplar.
- Kişisel, mobil, sunucu, gömülü ve **gerçek zamanlı** sistemlerin öncelikleri farklıdır. Hakları **lisans** belirler; destek bitince güvenlik güncellemesi gelmez. **Sanal makine** kendi çekirdeğiyle güçlü yalıtım, **konteyner** ortak çekirdekle hafiflik sağlar.

Soru	Kural	Örnek
Tamamlanma süresi	bitiş − varış	RR'de P2: 9 − 0 = 9 ms
Bekleme süresi	tamamlanma süresi − işlemci süresi	RR'de P2: 9 − 3 = 6 ms
Taban/limit	adres < limit ise taban + adres	150 < 200 → 100 + 150 = 250
Sayfa ve bağıl konum	adres ÷ 4096 ve adres mod 4096	4100 → sayfa 1, konum 4
Fiziksel adres	çerçeve × 4096 + bağıl konum	5 × 4096 + 4 = 20484
Disk blokları	boyut ÷ blok boyutu, yukarı yuvarlanır	9 KB, 4 KB blok → 3 blok
chmod basamağı	r = 4, w = 2, x = 1 toplanır	rw- → 6; rw-r--r-- → 644
MBR sınırı	2^{32} sektör × 512 bayt	≈ 2,2 TB

Tablo 2.1.19: Notta kullanılan hesap kuralları.

2.1.14 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

yazılım *software* · sistem yazılımı *system software* · uygulama yazılımı *application software* · ürün yazılımı *firmware* · yardımcı araç *utility* · ara yazılım *middleware* · işletim sistemi *operating system* · önyükleme *boot* · önyükleyici *boot loader* · önyükleme sırası *boot order* · UEFI / BIOS *UEFI / BIOS* · açılışta öz sınama (POST) *power-on self-test* · EFI sistem bölümü *EFI system partition* · Güvenli Önyükleme *Secure Boot* · TPM *Trusted Platform Module* · bölüm / bölümleme düzeni *partition / partitioning scheme* · çekirdek *kernel* · çekirdek modu / kullanıcı modu *kernel mode / user mode* · sistem çağrısı *system call* · kesme *interrupt* · monolitik / mikro / hibrit çekirdek *monolithic / micro / hybrid kernel* · Linux dağıtımı *Linux distribution* · süreç *process* · süreç kimliği (PID) *process ID* · iş parçacığı *thread* · süreç durumu *process state* · hazır kuyruğu *ready queue* · çizelgeleme / çizelgeleyici *scheduling / scheduler* · bağlam değiştirme *context switch* · eş zamanlılık *concurrency* · çoklu programlama *multiprogramming* · zaman paylaşımı *time sharing* · döner sıralı çizelgeleme *round robin* · zaman dilimi *time slice, quantum* · Gantt çizelgesi *Gantt chart* · kilitlenme *deadlock* · karşılıklı dışlama *mutual exclusion* · tut ve bekle *hold and wait* · ön alınamazlık *no preemption* · döngüsel bekleme *circular wait* · takas / takas alanı *swapping / swap space* · sanal bellek *virtual memory* · sanal adres *virtual address* · sayfa / çerçeve *page / frame* · sayfa tablosu *page table* · bağıl konum *offset* · bellek yönetim birimi (MMU) *memory management unit* · sayfa hatası *page fault* · sayfa dosyası *page file* · sayfa yer değiştirme *page replacement* · LRU *least recently used* · erişim ihlali *segmentation fault*

aygıt sürücüsü *device driver* · tak ve çalıştır *plug and play* · tampon *buffer* · dosya sistemi *file system* · meta veri *metadata* · günlükleme *journaling* · biçimlendirme *format* · kök / klasör / alt klasör *root / folder / subfolder* · yol / mutlak yol / göreli yol *path / absolute / relative path* · geçerli klasör *current directory* · dosya tanımlayıcısı *file descriptor* · tanıtıcı *handle* · dosya sonu *end of file* · grafik kullanıcı arayüzü (GUI) *graphical user interface* · komut satırı arayüzü (CLI) *command-line interface* · izin *permission* · kimlik doğrulama *authentication* · yetkilendirme *authorization* · en az ayrıcalık *principle of least privilege* · korumalı alan *sandbox* · gerçek zamanlı sistem *real-time system* · lisans *license* · açık kaynak / kapalı kaynak *open source / closed source* · güncelleme *update* · sanallaştırma *virtualization* · sanal makine *virtual machine* · hipervizör *hypervisor* · misafir / ana işletim sistemi *guest / host operating system* · konteyner / konteyner motoru *container / container engine* · bağımlılık *dependency*

2.1.15 Kendini Dene

1. Üç süreç t = 0'da hazır; sıra P1, P2, P3; süreler P1 = 3, P2 = 4, P3 = 1 ms. Döner sıralı çizelgelemede zaman dilimi 2 ms'dir. Gantt çizelgesi, bitiş zamanları ve bağlam değiştirme sayısı nedir?

   Cevap

   0–2 P1 (kalan 1), 2–4 P2 (kalan 2), 4–5 P3 (biter), 5–6 P1 (biter), 6–8 P2 (biter). Bitişler: P3 = 5, P1 = 6, P2 = 8 ms. 4 bağlam değiştirme yapılır. Bekleme süreleri P1 = 6 − 3 = 3, P2 = 8 − 4 = 4, P3 = 5 − 1 = 4 ms'dir.

2. Sayfa boyutu 4.096 bayt; sayfa tablosu: 0 → çerçeve 2, 1 → çerçeve 5, 2 → diskte, 3 → çerçeve 0. 13000 sanal adresi hangi sayfada, hangi bağıl konumdadır? Fiziksel adres nedir?

   Cevap

   13000 ÷ 4096 = 3, kalan *13000 − 12288 = 712*: sayfa 3, bağıl konum 712. Sayfa 3 çerçeve 0'dadır: *0 × 4096 + 712 = 712*. Çerçeve 0 RAM'in en başı olduğu için fiziksel adres bağıl konuma eşittir.

3. Bir sürecin 3 çerçevesi var ve sayfaları 1, 2, 3, 1, 4, 2, 5 sırasıyla istiyor. LRU ve FIFO ile kaçar sayfa

hatası olur?

Cevap

LRU: 1, 2, 3 hata; 1 isabet; 4 hata (son kullanımı en eski 2 çıkar); 2 hata (3 çıkar); 5 hata (1 çıkar) → 6 hata. FIFO: 1, 2, 3 hata; 1 isabet; 4 hata (ilk giren 1 çıkar); 2 isabet; 5 hata (2 çıkar) → 5 hata. Bu dizide FIFO daha iyi sonuç verir; LRU her durumda üstün değildir.

4. C sürecinin taban yazmacı 300, limit yazmacı 250'dir (KB). C, 249 ve 250 mantıksal adreslerine erişmek istiyor. Donanım her biri için ne yapar?

Cevap

249 < 250 olduğu için erişim geçerlidir: fiziksel adres *300 + 249 = 549*, yani C'nin bölgesinin son adresi. 250 ≥ 250 olduğu için erişim engellenir, kesme üretilir ve işletim sistemi C'yi erişim ihlaliyle sonlandırır.

5. Bir işletim sistemi, her sürecin ihtiyaç duyduğu bütün kaynakları çalışmaya başlamadan önce tek seferde istemesini zorunlu kılıyor. Kilitlenmenin dört koşulundan hangisi ortadan kalkar? Bu yöntemin bedeli nedir?

Cevap

Tut ve bekle ortadan kalkar: süreç ya hepsini alır ya hiçbirini almadan bekler; elinde kaynak tutarken başka kaynak beklemez. Bedeli, kaynakların gerçekte kullanılmadığı sürelerde de tutulmasıdır; örneğin işinin sonunda yazıcıya ihtiyaç duyan bir süreç yazıcıyı en baştan meşgul eder.

6. Blok boyutu 4.096 bayt olan bir diske 10.000 baytlık bir dosya yazılıyor. Dosya kaç blok kaplar ve kaç bayt boşa gider?

Cevap

10000 ÷ 4096 ≈ 2,44; yukarı yuvarlanır: 3 blok. *3 × 4096 = 12288* bayt yer kaplar; *12288 – 10000 = 2288* bayt boşa gider.

7. (a) `chmod 700 anahtar.txt` sonrası izinler nedir? (b) `rwxr-xr-x` izni hangi sayıyla verilir?

Cevap

(a) 7 = rwx, 0 = ---, 0 = --- → `rwx------` : yalnızca sahip her şeyi yapabilir. (b) rwx = 4 + 2 + 1 = 7, r-x = 4 + 1 = 5, r-x = 5 → `755` .

8. Bir web uygulamasının aynı Linux sürümünde çalışan 50 kopyası gerekiyor. Sanal makine mi kon-

teyner mi daha uygundur? Neden?

Cevap

Konteyner. Uygulama ana sistemin Linux çekirdeğini paylaşabilir; her kopya yalnızca uygulamayı ve kütüphanelerini taşır, saniyeler içinde başlar ve az bellek harcar. 50 sanal makine ise 50 tam işletim sistemi demektir ve çok daha fazla bellek ve açılış süresi ister.

2.1.16 İleri okuma

- R. H. Arpaci-Dusseau ve A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces* (Arpaci-Dusseau Books, 2018; ostep.org adresinde ücretsiz): sanallaştırma, eş zamanlılık ve kalıcılık başlıkları altında süreçleri, çizelgelemeyi, sanal belleği ve dosya sistemlerini sade bir dille ve bol örnekle anlatır.
- A. Silberschatz, P. B. Galvin ve G. Gagne, *Operating System Concepts* (10. baskı, Wiley, 2018): işletim sistemlerinin standart ders kitabıdır; bu nottaki her konu, özellikle çizelgeleme, kilitlenme ve sayfa yer değiştirme daha ayrıntılı işlenir.
- A. S. Tanenbaum ve H. Bos, *Modern Operating Systems* (5. baskı, Pearson, 2022): Linux, Windows ve Android'in iç yapısını örnek olaylarla inceler; çekirdek türleri ve sanallaştırma bölümleri güçlüdür.
- W. Shotts, *The Linux Command Line* (2. baskı, No Starch Press, 2019): komut satırını, dosya izinlerini ve kabuk betiklerini sıfırdan başlayarak öğretir.

2.1.17 Kaynaklar

1. A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, Wiley.
2. A. S. Tanenbaum, H. Bos, *Modern Operating Systems*, Pearson.
3. The Linux Foundation ve Free Software Foundation lisans kaynakları.
4. R. H. Arpaci-Dusseau, A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, Arpaci-Dusseau Books.
5. E. G. Coffman, M. Elphick, A. Shoshani, “System Deadlocks”, *ACM Computing Surveys* 3(2), 1971.

KISIM I · BÖLÜM 2.2

Windows İşletim Sistemi

Bir bilgisayarı açtığımızda karşımıza çıkan ilk şey işletim sisteminin arayüzüdür. Masaüstündeki simgeler, alttaki görev çubuğu, açılan pencereler ve dosyaların durduğu klasörler bu arayüzün parçalarıdır. Bu not, Windows 11'in günlük kullanımda en çok karşılaşılan bölümlerini anlatır: masaüstü ve Başlat menüsü, pencerelerin yönetimi, klavye kısayolları, dosya ve klasör düzeni, dosya yolları, Dosya Gezgini, Ayarlar uygulaması, hesaplar ve güvenlik, depolama bakımı, Görev Yöneticisi, yedekleme ve kurtarma. Her konu bir ekran taslağı ve adım adım işlem tablosuyla desteklenir. Disk kapasitesi, dosyanın diskte kapladığı alan, sıkıştırma oranı ve ekran ölçeği gibi sayılar da küçük hesaplarla açıklanır.

ÖĞRENME ÇIKTILARI

Arayüz
Başlat menüsünü, görev çubuğunu, pencereleri ve sanal masaüstlerini kullanmak; sık kullanılan klavye kısayollarını uygulamak.

Dosya yönetimi
Dosya, klasör, sürücü, uzantı ve yol kavramlarını uygulamak; mutlak ve göreli yolu adım adım çözmek; kopyalama ile taşımanın sonuçlarını ayırt etmek.

Ayarlar
Hesap, uygulama, varsayılan uygulama, dil ve erişilebilirlik seçeneklerini yönetmek.

Bakım
Depolama, sürücü iyileştirme, güncelleme, yedekleme, kurtarma ve güvenlik araçlarını amacına uygun seçmek ve kullanmak.

2.2.1 Windows ve masaüstü

İşletim sistemi, donanımı yöneten ve uygulamalara ortak hizmetler sunan temel yazılımdır. Belleği uygulamalar arasında paylaştırır, dosyaları diskte düzenler, klavye ve fareden gelen olayları doğru pencereye iletir. Kullanıcı ise bu işlerin çoğunu görmez; yalnızca işletim sisteminin arayüzüyle çalışır. Windows'ta bu arayüzün merkezi masaüstüdür.

2.2.1.1 Windows ve sürümleri

Windows, Microsoft'un kişisel bilgisayarlar için geliştirdiği işletim sistemidir. Masaüstü ve dizüstü bilgisayarlarda en yaygın kullanılan işletim sistemi odur. Windows, pencere tabanlı bir grafik arayüz kullanır. Her uygulama kendi dikdörtgen alanında, yani bir pencerede çalışır; kullanıcı da pencereleri fareyle ya da klavyeyle yönetir.

Tanım 2.2.1

Grafik kullanıcı arayüzü (*graphical user interface*, GUI), kullanıcının bilgisayarla simgeler, pencereler, menüler ve düğmeler üzerinden etkileşim kurduğu arayüzdür. Komutlar yazı yazarak değil, öğelere tıklayarak, sürükleyerek ya da kısayol tuşlarına basarak verilir.

Windows yıllar içinde birçok sürümle geldi. Bu not Windows 11'e göre yazıldı. Windows 10'un desteği 14 Ekim 2025'te sona erdi. Desteği biten bir sürüm çalışmaya devam eder, ama artık güvenlik güncellemesi almaz. Bu yüzden internete bağlı bir bilgisayarda desteklenen bir sürüm kullanmak gerekir.

Windows 11 de kendi içinde yılda bir büyük güncelleme alır. Menü adları ve bazı düğmelerin yeri bu güncellemelerle küçük farklar gösterebilir. Notta kullanılan adlar Windows 11'in Türkçe arayüzündeki adlardır. Bir ad kullanılan bilgisayarda biraz farklıysa, işlevi aynı olan seçenek genellikle aynı menünün içinde bulunur.

Sık yapılan hata

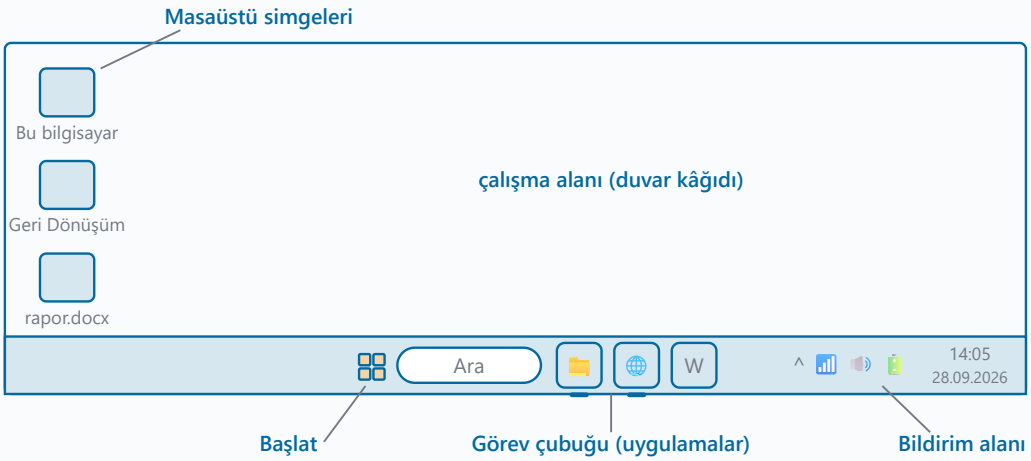
"Bilgisayar çalışıyorsa sürümün eski olması sorun değildir" diye düşünmek. Desteği biten sürümde yeni bulunan güvenlik açıkları kapatılmaz. Bilgisayar her gün aynı görünür, ama zamanla saldırılara karşı daha savunmasız hâle gelir.

2.2.1.2 Masaüstü

Tanım 2.2.2

Masaüstü (*desktop*), oturum açıldıktan sonra görünen ve uygulamalara, dosyalara ve sistem işlevlerine erişim sağlayan ana çalışma alanıdır. Duvar kâğıdı, masaüstü simgeleri ve ekranın altındaki görev çubuğundan oluşur.

Şekil 2.2.1 masaüstünün dört ana bölümünü gösterir. Ekranın büyük kısmını **çalışma alanı** kaplar. Duvar kâğıdı burada görünür ve açılan pencereler bu alanın üstüne yerleşir. Sol üstteki **masaüstü simgeleri** sık kullanılan öğelere hızlı erişim sağlar: "Bu bilgisayar", "Geri Dönüşüm Kutusu" ve kullanıcının masaüstüne koyduğu dosyalar. Altta **görev çubuğu** uzanır. Sağ alt köşede ise saat, ağ, ses ve pil simgelerinin bulunduğu **bildirim alanı** yer alır.



Şekil 2.2.1: Windows masaüstünün taslağı. Sol üstte masaüstü simgeleri (Bu bilgisayar, Geri Dönüşüm Kutusu, rapor.docx), altta Başlat düğmesi, arama kutusu ve uygulama simgelerinin bulunduğu görev çubuğu, sağ altta bildirim alanı (ağ, ses, pil, saat ve tarih) vardır. Geri kalan alan duvar kâğıdının göründüğü çalışma alanıdır.

Masaüstünün görünümü bilgisayardan bilgisayara değişir. Görev çubuğundaki simgeler ortalanmış ya da sola dayalı olabilir. Duvar kâğıdı, renkler ve simgelerin boyutu Ayarlar uygulamasındaki *Kişiselleştirme* bölümünden değiştirilir. Bu yüzden öğelerin yerini ezberlemek yerine her öğenin görevini bilmek daha yararlıdır.

Masaüstü aynı zamanda bir klasördür. Masaüstüne kaydedilen bir dosya, diskte kullanıcının `Desktop` klasöründe durur. Masaüstünü dosya deposu gibi kullanmak kısa sürede karmaşaya yol açar. Yüzlerce simge arasında aranan dosya zor bulunur. Masaüstünde yalnızca o gün üzerinde çalışılan birkaç dosya ve kısayol tutmak, geri kalanını 2.2.4.2'de anlatılan klasör düzenine yerleştirmek daha iyi bir alışkanlıktır.

2.2.1.3 Başlat menüsü ve arama

Görev çubuğundaki Windows simgesi **Başlat düğmesidir**. Bu düğmeye tıklamak ya da klavyedeki Windows tuşuna (Win) basmak **Başlat menüsünü** açar.

> **Tanım 2.2.3**
>
> **Başlat menüsü** (*Start menu*), yüklü uygulamalara, son kullanılan dosyalara, kullanıcı hesabı seçeneklerine ve güç komutlarına tek yerden ulaşılan menüdür.

Şekil 2.2.2 Başlat menüsünün beş bölümünü numaralarla gösterir:

1. **Arama.** Menünün en üstündeki kutudur. Uygulama, dosya, ayar ve yardım içeriği bulur. Başlat menüsü açıldıktan sonra kutuya tıklamaya gerek yoktur; yazmaya başlamak aramayı başlatır.

2. **Sabitlenenler.** Kullanıcının sık kullandığı uygulamaların simgeleridir. Bir uygulama sağ tıklanıp "Başlangıç'a sabitle" seçilerek buraya eklenir. Sağ üstteki *Tüm uygulamalar* bağlantısı yüklü bütün uygulamaların alfabetik listesini açar.
3. **Önerilenler.** Son açılan dosyalar ve yeni yüklenen uygulamalardır. Yarım kalan bir belgeye dönmenin en kısa yoludur.
4. **Kullanıcı menüsü.** Sol alttaki kullanıcı adına tıklanınca açılır: hesap ayarları, kilitle, oturumu kapat.
5. **Güç.** Sağ alttaki düğmedir: uyku, kapat, yeniden başlat.



Şekil 2.2.2: Başlat menüsünün taslağı: (1) arama kutusu, (2) sabitlenen uygulamalar ve Tüm uygulamalar bağlantısı, (3) önerilen dosyalar, (4) kullanıcı adı, (5) güç düğmesi.

Başlat düğmesine *sağ* tıklamak başka bir menü açar. Bu menüye **hızlı bağlantı menüsü** denir ve Win + X kısayoluyla da açılır. İçinde Görev Yöneticisi, Ayarlar, Dosya Gezgini, Aygıt Yöneticisi, Disk Yönetimi ve Terminal gibi sistem araçları bulunur. Bu menü, bir sistem aracına Başlat menüsünde aramadan ulaşmanın en kısa yoludur.

Örnek 2.2.1

Ekran çözünürlüğü değiştirilmek isteniyor, ama ayarın hangi menüde olduğu bilinmiyor. Arama ile şöyle bulunur:

1. Win tuşuna basılır; Başlat menüsü açılır.
2. Doğrudan `çözünürlük` yazılır. Arama kutusuna tıklamaya gerek yoktur.
3. Sonuçların en üstünde "Ekran çözünürlüğünü değiştir" ayarı görünür. Enter'a basılınca Ayarlar uygulaması doğrudan Sistem → Ekran sayfasında açılır.

Aynı yöntem uygulamalar için de geçerlidir. Win tuşuna basıp `word` yazmak ve Enter'a basmak, Word'ü menüde aramaktan daha hızlıdır.

Sık yapılan hata

Bilgisayarı kapatmak için güç düğmesine basılı tutmak. Kasadaki düğmeye uzun basmak elektriği keser; kaydedilmemiş belgeler kaybolur, işletim sistemi de yarım kalan yazma işlemlerini tamamlayamaz. Bilgisayar Başlat menüsündeki Güç → Kapat komutuyla kapatılır. Düğmeye uzun basmak yalnızca sistem hiçbir tuşa yanıt vermediğinde son çaredir.

2.2.1.4 Görev çubuğu

Tanım 2.2.4

Görev çubuğu (*taskbar*), ekranın altında sürekli görünen ve Başlat düğmesini, arama kutusunu, sabitlenmiş ve açık uygulamaları, sistem tepsisini, hızlı ayarları ve saati taşıyan çubuktur.

Şekil 2.2.3 görev çubuğunu soldan sağa yedi parçaya ayırır. Her parçanın görevi Tablo 2.2.1'de özetlenmiştir.



Şekil 2.2.3: Görev çubuğunun parçaları. (1) Başlat, (2) arama, (3) görev görünümü, (4) uygulama simgeleri, (5) sistem tepsisi, (6) hızlı ayarlar, (7) saat ve bildirimler. Uygulama simgesinin altındaki çizgi uygulamanın açık olduğunu, vurgulu simge etkin pencereyi gösterir; çizgisiz simge yalnızca sabitlenmiştir.

No	Parça	Görevi
1	**Başlat**	Uygulamalar, arama, hesap ve güç komutları
2	**Arama**	Dosya, uygulama ve ayar bulma
3	**Görev görünümü**	Açık pencereleri ve sanal masaüstlerini gösterme
4	**Uygulamalar**	Sabitlenmiş uygulamaları başlatma, açık pencereler arasında geçiş
5	**Sistem tepsisi**	Arka planda çalışan uygulamaların simgeleri (^ okunun altında)
6	**Hızlı ayarlar**	Ağ, ses ve pil simgeleri; tıklanınca hızlı ayarlar paneli
7	**Saat ve bildirimler**	Tarih ve saat; tıklanınca takvim ve bildirim merkezi

Tablo 2.2.1: Görev çubuğunun parçaları ve görevleri.

Dördüncü parça, yani uygulama simgeleri, en çok kullanılan bölümdür. Bir simge üç durumdan birinde olabilir. **Sabitlenmiş** simge, uygulama kapalıyken de görev çubuğunda durur ve tıklanınca uygulamayı başlatır. **Açık** uygulamanın simgesinin altında kısa bir çizgi bulunur. O anda klavye girdisini alan pencerenin simgesi ise daha belirgin vurgulanır. Bir uygulamanın birden çok penceresi açıksa, simgenin üzerine gelindiğinde pencerelerin küçük resimleri görünür ve istenen pencere seçilir.

Bir uygulamayı görev çubuğuna sabitlemek için uygulama açıkken simgesine sağ tıklanır ve "Görev çubuğuna sabitle" seçilir. Aynı menüdeki "Görev çubuğundan kaldır" komutu simgeyi kaldırır; uygulamanın kendisi bilgisayardan silinmez.

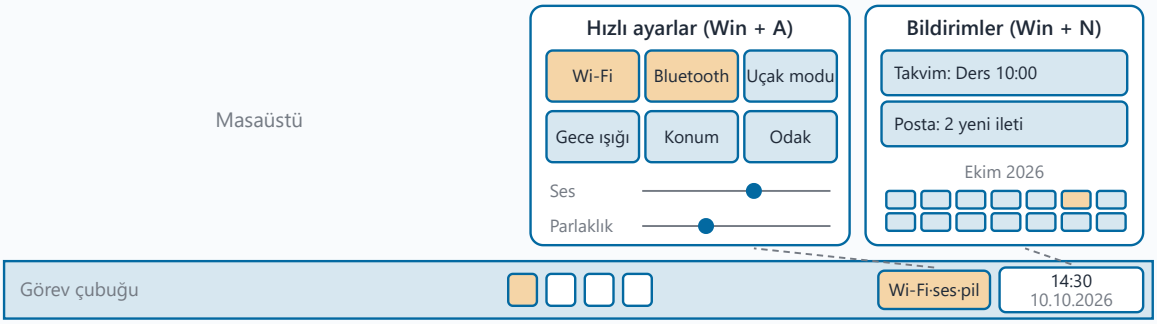
Beşinci parça olan **sistem tepsisi** (*system tray*) ilk bakışta gözden kaçar. Bulut eşitleme, antivirüs ya da müzik uygulaması gibi programlar penceresiz, arka planda çalışır. Bunların simgeleri ^ okuna tıklanınca açılan küçük panelde görünür. Bir arka plan uygulamasını tamamen kapatmak için simgesine sağ tıklanıp "Çıkış" ya da "Kapat" seçilir.

2.2.1.5 Hızlı ayarlar ve bildirimler

Görev çubuğunun en sağındaki iki alan küçük ama sık kullanılan iki paneli açar (Şekil 2.2.4).

Tanım 2.2.5

Hızlı ayarlar (*quick settings*), Wi-Fi, Bluetooth, uçak modu, ses, parlaklık ve erişilebilirlik gibi sık değiştirilen ayarları tek panelde toplayan alandır; ağ, ses ve pil simgelerine tıklanarak ya da Win + A ile açılır. **Bildirim merkezi** (*notification center*) ise uygulama bildirimlerini ve takvimi gösteren paneldir; saate tıklanarak ya da Win + N ile açılır.



Şekil 2.2.4: Görev çubuğunun sağ ucundan açılan iki panel. Ağ, ses ve pil simgeleri Hızlı ayarlar panelini (Wi-Fi, Bluetooth, Uçak modu, Gece ışığı, Konum, Odak düğmeleri ile ses ve parlaklık kaydırıcıları) açar. Saat ise Bildirimler ve takvim panelini açar.

Hızlı ayarlar panelindeki düğmeler bir tıklamayla açılıp kapanır. Örneğin uçak modu bütün kablosuz bağlantıları birden keser. Gece ışığı ekranın mavi ışığını azaltarak akşam saatlerinde göz yorgunluğunu hafifletir. Wi-Fi düğmesinin yanındaki ok ise yakındaki kablosuz ağların listesini açar. Bir ağa bağlanmanın adımları 2.2.8.2'de verilir.

Bildirimler bir uygulamanın kullanıcıya kısa bir haber vermesidir: yeni bir e-posta, yaklaşan bir ders, biten bir indirme. Bildirimler ekranın köşesinde birkaç saniye görünür, sonra bildirim merkezinde birikir. Dikkat dağıtan bildirimler uygulama bazında kapatılabilir. Bunun için Ayarlar → Sistem → Bildirimler sayfasında ilgili uygulamanın anahtarı kapatılır. Aynı sayfadaki **Rahatsız etmeyin** modu bildirimleri belirli saatlerde, örneğin ders ya da sınav sırasında, susturur.

Sık yapılan hata

Bütün bildirimleri okumadan kapatmak. “Disk dolmak üzere”, “Güncelleme için yeniden başlatma gerekiyor” ya da “Virüs ve tehdit koruması kapalı” gibi kritik sistem uyarıları da bildirim olarak gelir. Bu uyarılar göz ardı edilmeden önce okunmalıdır. Kapatılacak olan, yalnızca dikkat dağıtan uygulama bildirimleridir.

2.2.2 Pencere yönetimi

Windows adını pencerelerden alır. Aynı anda açık olan her uygulama kendi penceresinde çalışır. Bir ödev yazılırken bir yanda Word, öbür yanda kaynakların açık olduğu tarayıcı, arka planda da müzik çalan bir uygulama bulunabilir. Pencereleri hızlı düzenlemek, birden çok işi bir arada yürütmeyi kolaylaştırır.

2.2.2.1 Pencerenin bölümleri

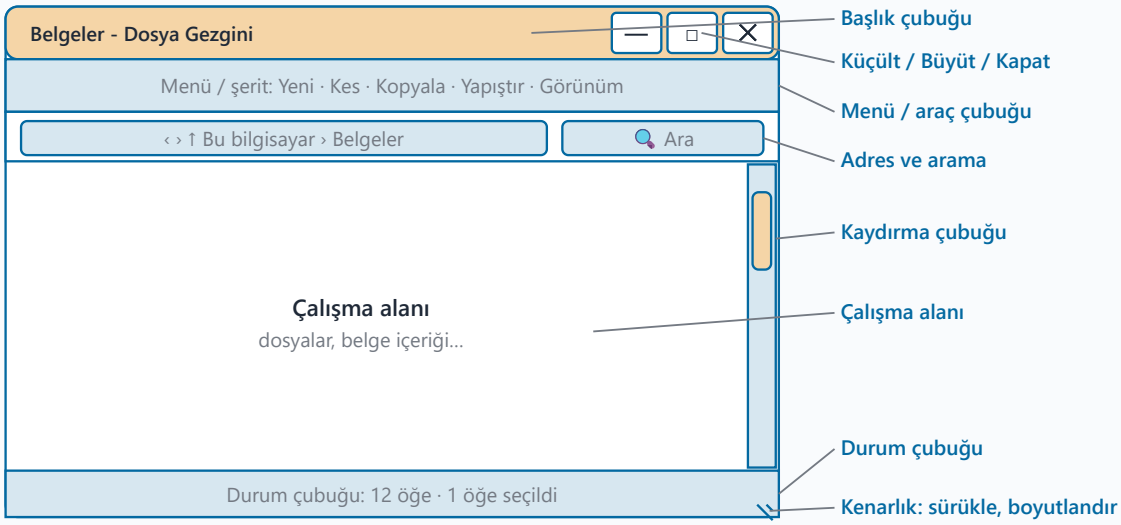
Tanım 2.2.6

Pencere (*window*), bir uygulamanın ekranda kapladığı, taşınabilen ve boyutlandırılabilen dikdörtgen alandır. Hemen her pencerede başlık çubuğu, denetim düğmeleri, çalışma alanı ve kenarlık bulunur.

Şekil 2.2.5 Dosya Gezgini penceresi üzerinde pencerenin bölümlerini gösterir. Uygulamalar farklı olsa da bu bölümlerin çoğu her pencerede aynı yerde bulunur:

- **Başlık çubuğu** (*title bar*) en üstteki şerittir. Uygulamanın ve açık belgenin adını gösterir: “Belgeler - Dosya Gezgini”. Pencere bu çubuktan tutulup sürüklenerek taşınır; çubuğa çift tıklamak pencereyi büyütür ya da eski boyutuna döndürür.
- **Denetim düğmeleri** başlık çubuğunun sağındaki üç düğmedir: Küçült (—), Büyüt (□) ve Kapat (×).
- **Menü ya da araç çubuğu** (*toolbar*) uygulamanın komutlarını taşır. Dosya Gezgini'nde Yeni, Kes, Kopyala, Yapıştır ve Görünüm komutları buradadır; Word'de ise bu alan şerit biçimindedir.
- **Adres ve arama çubuğu** bazı uygulamalarda bulunur. Dosya Gezgini'nde o anda açık klasörün yolunu gösterir.

- **Çalışma alanı** pencerenin asıl içeriğidir: dosya listesi, belge metni, web sayfası.
- **Kaydırma çubuğu** (*scroll bar*) içerik pencereye sığmadığında görünür ve içeriği aşağı-yukarı kaydırır.
- **Durum çubuğu** (*status bar*) en alttaki ince şerittir. Dosya Gezgini'nde klasördeki öğe sayısını ve seçili öğe sayısını, Word'de sayfa ve sözcük sayısını gösterir.
- **Kenarlık** pencerenin dış çizgisidir. Fare kenarlığın ya da köşenin üzerine gelince imleç çift oka dönüşür; sürüklemek pencereyi boyutlandırır.



Şekil 2.2.5: Bir pencerenin bölümleri: başlık çubuğu, Küçült / Büyüt / Kapat düğmeleri, menü ya da araç çubuğu, adres ve arama çubuğu, çalışma alanı, kaydırma çubuğu, durum çubuğu ve kenarlık.

2.2.2.2 Küçült, büyüt, kapat

Bir pencere üç durumdan birinde bulunur: **normal** (ekranın bir kısmını kaplar, taşınabilir ve boyutlandırılabilir), **büyütülmüş** (tüm ekranı kaplar) ya da **küçültülmüş** (ekranda görünmez, yalnızca görev çubuğunda simgesi durur). Denetim düğmeleri pencereyi bu durumlar arasında geçirir:

- **Küçült** (—) pencereyi görev çubuğuna gönderir. Uygulama kapanmaz, çalışmaya devam eder. Görev çubuğundaki simgeye tıklamak pencereyi eski yerine getirir.
- **Büyüt** (□) pencereyi tüm ekrana yayar. Pencere büyütülünce aynı düğmenin simgesi değişir ve adı **Geri yükle** olur; ona basmak pencereyi önceki boyutuna döndürür.
- **Kapat** (×) pencereyi kapatır. Uygulamanın tek penceresi buysa uygulama da kapanır. Kaydedilmemiş bir değişiklik varsa uygulama önce kaydedip kaydetmeyeceğini sorar.

Sık yapılan hata

Küçültmeyi kapatmakla karıştırmak. Küçültülen pencere ekrandan kaybolur ama uygulama bellekte çalışmayı sürdürür. Onlarca pencere küçültülüp unutulunca bellek dolar ve bilgisayar yavaşlar. Görev çubuğundaki simgelerin altındaki çizgiler hangi uygulamaların hâlâ açık olduğunu gösterir. İşi biten pencere küçültülmez, kapatılır.

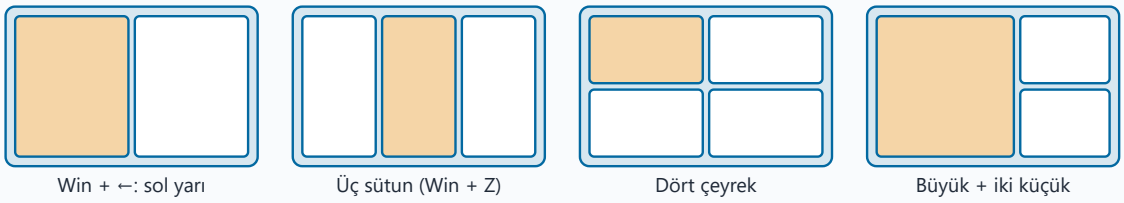
2.2.2.3 Yaslama ve pencereler arasında geçiş

Pencereleri fareyle elle boyutlandırmak yavaştır. Windows, pencereleri ekranın kenarlarına ve köşelerine hızla yerleştiren bir düzenleme özelliği sunar.

Tanım 2.2.7

Yaslama (*snap*), bir pencereyi ekranın yarısına, üçte birine ya da çeyreğine oturtan pencere düzenleme özelliğidir. Pencere başlık çubuğundan ekranın kenarına sürüklenerek, Win + yön tuşlarıyla ya da Win + Z ile açılan yaslama düzenlerinden biri seçilerek yaslanır.

Şekil 2.2.6 yaslama düzenlerinden dördünü gösterir. Win + ← etkin pencereyi ekranın sol yarısına, Win + → sağ yarısına yerleştirir. Win + ↑ pencereyi büyütür, Win + ↓ ise önce eski boyutuna döndürür, bir kez daha basılınca küçültür. Win + Z ya da Büyüt düğmesinin üzerinde fareyi bekletmek, üç sütun, dört çeyrek ya da bir büyük iki küçük pencere gibi hazır düzenleri gösterir.



Şekil 2.2.6: Yaslama düzenleri: Win + ← ile sol yarı, Win + Z ile açılan üç sütun, dört çeyrek ve bir büyük iki küçük pencere düzeni.

Açık pencereler arasında geçmenin en hızlı yolu **Alt + Tab** kısayoludur. Alt tuşu basılı tutulup Tab'a basılınca ekranın ortasında açık pencerelerin küçük resimleri belirir. Alt basılı kaldıkça her Tab bir sonraki küçük resmi seçer; Alt bırakılınca seçili pencere öne gelir. Alt + Tab'a bir kez basıp bırakmak, iki pencere arasında gidip gelmeyi sağlar.

Bu işlemlerin birlikte nasıl çalıştığını bir örnek üzerinde izleyelim. Şekil 2.2.7 sunumdaki adım adım slaytın üç önemli anını, Tablo 2.2.2 ise bütün adımları gösterir.

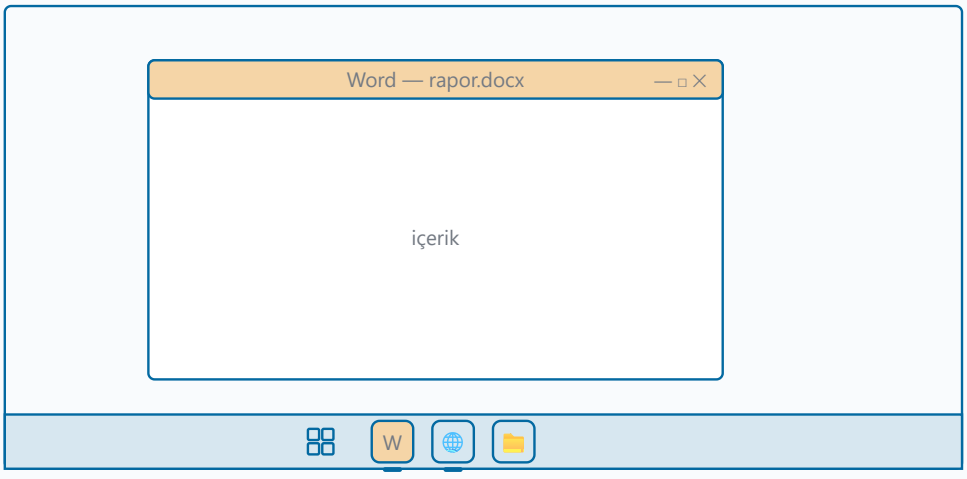
Örnek 2.2.2

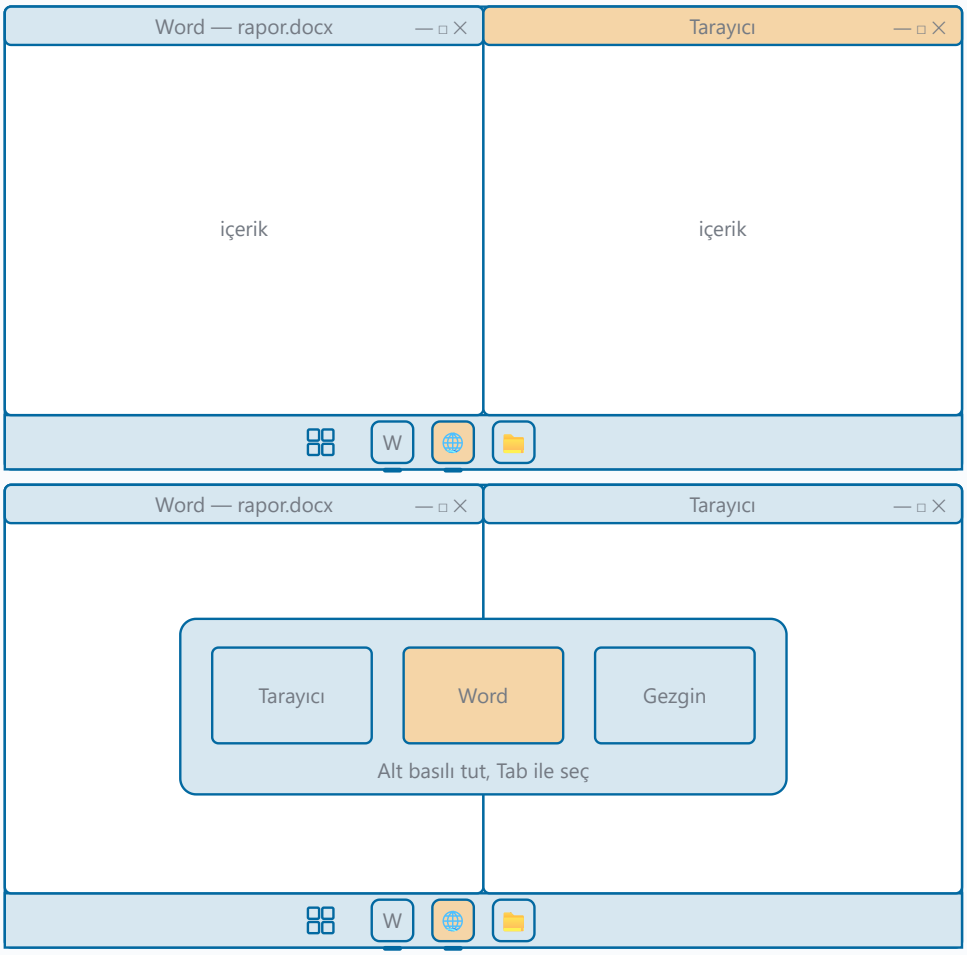
Bir öğrenci rapor yazıyor. Word'de `rapor.docx` açık, tarayıcıda kaynak makale var. Ekranı kaynaktan okuyup rapora yazabileceği biçimde düzenlemek istiyor. Başlangıçta Word normal boyutta ve öndedir. İşlemler sırayla şöyledir:

Adım	İşlem	Word	Tarayıcı	Etkin pencere
0	başlangıç	normal	arkada	Word
1	Word'de Küçült (—)	küçültülmüş (çizgi duruyor)	arkada	—
2	Word simgesine tık, sonra Büyüt (□)	büyütülmüş	arkada	Word
3	Win + ←	sol yarı	arkada	Word
4	tarayıcıya geçip Win + →	sol yarı	sağ yarı	Tarayıcı
5	Alt + Tab, Word seçilir	sol yarı	sağ yarı	Word
6	Word'de Kapat (×)	kapalı (gerekirse kaydetmeyi sorar)	sağ yarı	Tarayıcı

Tablo 2.2.2: Pencere işlemleri ve her işlemden sonraki durum.

Birinci adımda Word kapanmaz; simgesinin altındaki çizgi uygulamanın açık olduğunu gösterir. İkinci adımda Büyüt düğmesinin adı Geri yükle olur. Dördüncü adımdan sonra iki pencere yan yana durur. Beşinci adımda Alt + Tab yerleşimi değiştirmez, yalnızca klavyenin hangi pencereye yazacağını değiştirir. Son adımda yalnızca Word penceresi kapanır, tarayıcı yerinde kalır.





Şekil 2.2.7: Pencere durumlarından üç an. Üstte Word normal boyuttadır. Ortada Word Win + ← ile sol yarıya, tarayıcı Win + → ile sağ yarıya yaslanmıştır. Altta Alt basılı tutulup Tab'a basılmış, açık pencerelerin küçük resimleri görünür.

Sık yapılan hata

Alt + Tab'ın pencereleri yeniden düzenlediğini sanmak. Alt + Tab yalnızca etkin pencereyi değiştirir. Yaslanmış iki pencere arasında geçildiğinde ikisi de yerinde kalır; yalnızca öne gelen ve klavye girdisini alan pencere değişir.

2.2.2.4 Sanal masaüstleri

Aynı anda çok sayıda pencere açıkken ekran kalabalıklaşır. Ders için açılmış Word ve tarayıcı pencereleriyle müzik ve oyun pencereleri birbirine karışır. Windows bu sorunu, pencereleri ayrı çalışma alanlarına bölerek çözer.

Tanım 2.2.8

Sanal masaüstü (*virtual desktop*), açık pencerelerin bir grubunu barındıran ayrı çalışma alanıdır. Ekranda her an yalnızca bir masaüstünün pencereleri görünür; öteki masaüstlerindeki uygulamalar arka planda çalışmaya devam eder.

Sanal masaüstleri dört kısayolla yönetilir (Tablo 2.2.3). Bütün masaüstlerini bir arada görmek için **görev görünümü** (*task view*) kullanılır. Görev görünümü, görev çubuğundaki düğmeyle ya da Win + Tab ile açılır. Üstte etkin masaüstünün açık pencerelerini, altta da masaüstlerinin listesini gösterir.

İşlem	Kısayol	Sonuç
Görev görünümü	Win + Tab	açık pencereleri ve masaüstlerini gösterir
Yeni masaüstü	Win + Ctrl + D	boş bir masaüstü oluşturur ve ona geçer
Geçiş	Win + Ctrl + ← / →	soldaki ya da sağdaki masaüstüne geçer
Kapatma	Win + Ctrl + F4	etkin masaüstünü kapatır; pencereleri soldaki masaüstüne taşınır

Tablo 2.2.3: Sanal masaüstü kısayolları.

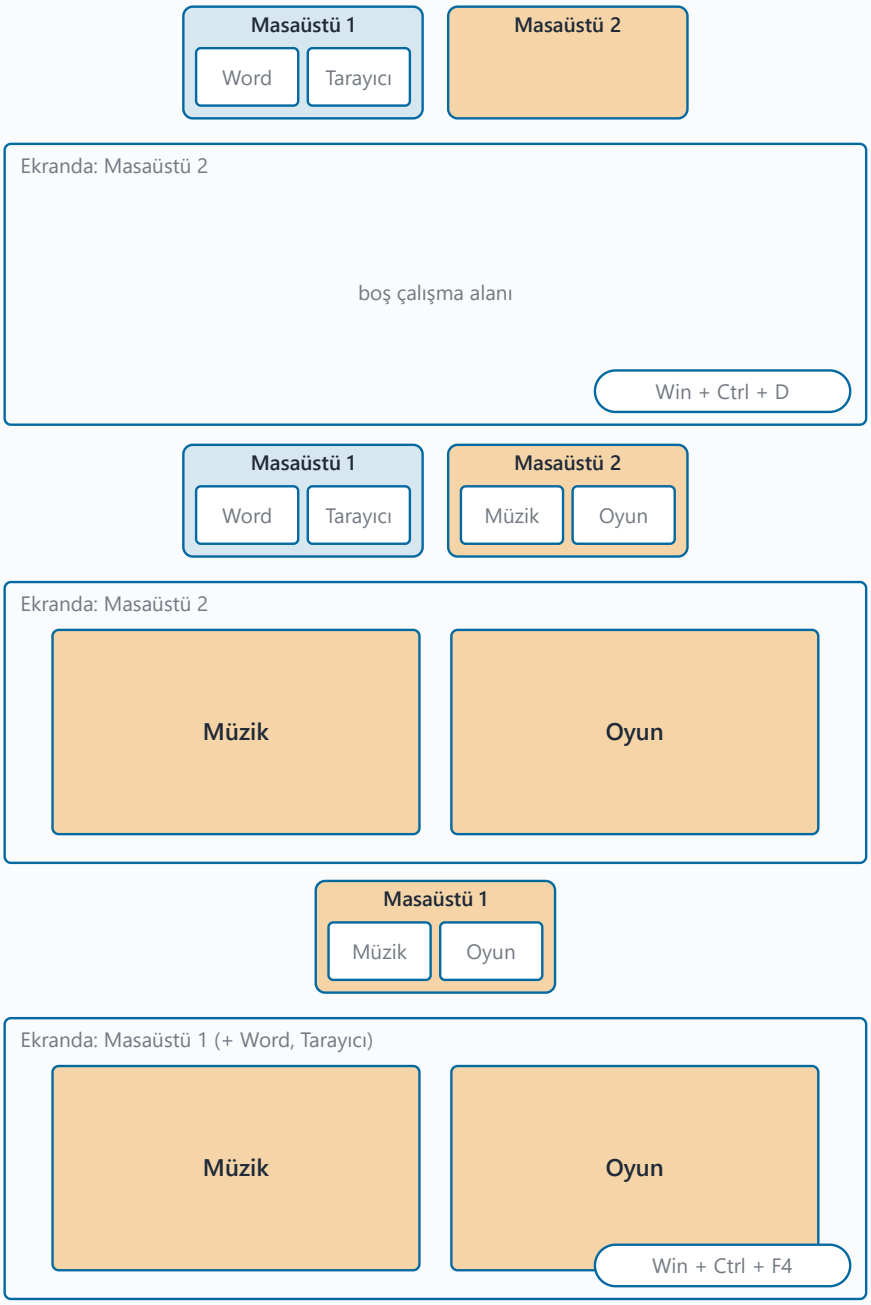
Örnek 2.2.3

Masaüstü 1'de ders için Word ve tarayıcı açık. Öğrenci ara verip müzik dinlemek ve oyun oynamak istiyor, ama ders pencerelerini kapatmak istemiyor. Kısayollar sırayla uygulanıyor:

Adım	İşlem	Masaüstü 1	Masaüstü 2	Ekranda
0	başlangıç	Word, Tarayıcı	—	Masaüstü 1
1	Win + Ctrl + D	Word, Tarayıcı	(boş)	Masaüstü 2
2	Müzik ve Oyun açılır	Word, Tarayıcı	Müzik, Oyun	Masaüstü 2
3	Win + Ctrl + ←	Word, Tarayıcı	Müzik, Oyun	Masaüstü 1
4	Win + Ctrl + →	Word, Tarayıcı	Müzik, Oyun	Masaüstü 2
5	Win + Ctrl + F4	Word, Tarayıcı, Müzik, Oyun	— (kapatıldı)	Masaüstü 1

Tablo 2.2.4: Sanal masaüstlerinin adım adım izlenmesi.

Birinci adımda Masaüstü 1'deki pencereler kapanmaz; yalnızca ekrandan çekilir. Üçüncü adımda ders pencereleri olduğu gibi geri gelir. Bu sırada Masaüstü 2'deki müzik çalmaya devam eder. Son adımda Masaüstü 2 kapanır, ama içindeki Müzik ve Oyun kapanmaz; Masaüstü 1'e taşınır.



Şekil 2.2.8: Sanal masaüstlerinden üç an. Üstte Win + Ctrl + D ile boş Masaüstü 2 oluşturulmuş ve ona geçilmiştir. Ortada Masaüstü 2'de Müzik ve Oyun açılmıştır. Altta Win + Ctrl + F4 ile Masaüstü 2 kapatılmış, pencereleri Masaüstü 1'e taşınmıştır.

Sık yapılan hata

Masaüstünü kapatmanın içindeki uygulamaları da kapattığını sanmak. Win + Ctrl + F4 yalnızca çalışma alanını kaldırır; pencereler soldaki masaüstüne taşınır ve çalışmaya devam eder. Bir uygulamayı kapatmak için penceresindeki ✕ düğmesi kullanılır. Bunun tersi de geçerlidir: sanal masaüstüne geçmek uygulamaları durdurmaz. Ekranda görünmeyen bir masaüstündeki

oyun da bellek ve işlemci kullanmayı sürdürür.

2.2.3 Klavye kısayolları ve ekran görüntüsü

Faredeki her hareket bir menüyü açmayı, bir öğeyi bulmayı ve tıklamayı gerektirir. Aynı işi bir tuş birleşimiyle yapmak çoğu zaman birkaç kat daha hızlıdır. Bu yüzden sık yapılan işlerin kısayollarını bilmek, günlük kullanımda büyük zaman kazandırır.

Tanım 2.2.9

Klavye kısayolu (*keyboard shortcut*), bir komutu menüden seçmek yerine birden çok tuşa birlikte basarak çalıştırma yoludur. “Ctrl + C” yazımı, Ctrl tuşu basılı tutulurken C tuşuna basılıp ikisinin birlikte bırakılması anlamına gelir.

Kısayollarda kullanılan değiştirici tuşların her birinin kabaca bir alanı vardır. **Ctrl** genellikle uygulamanın içindeki komutları (kopyala, yapıştır, kaydet, geri al) çalıştırır. **Win** tuşu Windows'un kendi işlevlerine (Ayarlar, Dosya Gezgini, pencere yerleşimi) ulaşır. **Alt** tuşu pencereler arasında geçişte ve menülere erişimde kullanılır. **Shift** ise bir komutun “daha güçlü” biçimini verir: Delete dosyayı Geri Dönüşüm Kutusu'na gönderir, Shift + Delete kalıcı siler.

2.2.3.1 Temel kısayollar

Tablo 2.2.5 sunumdaki temel kısayolları iki gruba ayırarak verir. İlk dört kısayol hemen her uygulamada aynı işi yapar. Geri kalanlar Windows'un kendi araçlarını açar.

Kısayol	İşlev	Not
Ctrl + C	**Kopyala**	seçili öğe panoya kopyalanır; kaynak değişmez
Ctrl + X	**Kes**	seçili öğe taşınmak üzere panoya alınır
Ctrl + V	**Yapıştır**	panodaki öğe imlecin ya da açık klasörün içine konur
Ctrl + Z	**Geri al**	son işlem geri alınır; Dosya Gezgini'nde de çalışır
Win + E	**Dosya Gezgini**	yeni bir Dosya Gezgini penceresi açar
Win + I	**Ayarlar**	Ayarlar uygulamasını açar
Win + R	**Çalıştır**	bir program ya da klasör adı yazılarak açılan küçük kutu
Win + L	**Ekranı kilitle**	oturum açık kalır, ekran kilitlenir
Win + Shift + S	**Ekran görüntüsü alanı**	seçilen alanın görüntüsünü panoya alır
Ctrl + Shift + Esc	**Görev Yöneticisi**	doğrudan Görev Yöneticisi'ni açar
Win + V	**Pano geçmişi**	son kopyalanan öğelerin listesini gösterir

Kısayol	İşlev	Not
Win + D	**Masaüstünü göster**	bütün pencereleri gizler; ikinci basış geri getirir

Tablo 2.2.5: Temel klavye kısayolları.

Kopyala, kes ve yapıştır komutlarının arkasında **pano** (*clipboard*) bulunur. Pano, kopyalanan ya da kesilen öğenin geçici olarak tutulduğu bellek alanıdır. Normalde panoda tek öğe durur ve her yeni kopyalama bir öncekinin üzerine yazar. Win + V ile açılan **pano geçmişi** bu sınırı kaldırır: son kopyalanan metin ve görüntüler listelenir, istenen öğe tıklanarak yapıştırılır. Pano geçmişi, ilk kullanımda açılması gereken bir özelliktir.

Win + L ile oturumu kilitlemek ile oturumu kapatmak farklı işlerdir. Kilitlemede açık uygulamalar ve belgeler olduğu gibi kalır. Ekrana kilit görüntüsü gelir ve geri dönmek için PIN ya da parola istenir. Oturumu kapatmak ise bütün uygulamaları kapatır. Bilgisayarın başından kısa bir süre için kalkarken kilitlemek yeterlidir.

Örnek 2.2.4

Word ve tarayıcı açıkken sırayla Win + D, Win + E, Win + Shift + S ve Win + L tuşlarına basılıyor. Her adımdan sonra ekranda şunlar görülür:

1. **Win + D.** Masaüstü görünür; Word ve tarayıcı pencereleri gizlenir. Uygulamalar kapanmaz, simgelerinin altındaki çizgi durur.
2. **Win + E.** Dosya Gezgini penceresi açılır ve masaüstünün üstünde görünür.
3. **Win + Shift + S.** Ekran kararır ve üstte bir seçim çubuğu belirir. Fareyle bir alan seçilir; bu alanın görüntüsü panoya alınır.
4. **Win + L.** Kilit ekranı gelir. Word, tarayıcı ve Dosya Gezgini arka planda açık kalır. PIN ya da parola girilince ekran kilitlenmeden önceki hâline döner.

Dört adımın hiçbirinde bir uygulama kapanmaz. Kısayollar yalnızca neyin görüneceğini değiştirir.

2.2.3.2 Diğer kullanışlı kısayollar

Tablo 2.2.6 daha az bilinen ama sık işe yarayan kısayolları toplar. Bunların bir kısmı 2.2.1.3 ve 2.2.1.5'te anlatılan menüleri ve panelleri açar.

Kısayol	İşlev
Win + X	hızlı bağlantı menüsü (Başlat'a sağ tıklamakla aynı)
Win + A	hızlı ayarlar paneli
Win + N	bildirimler ve takvim
Win + Z	yaslama düzenleri

Kısayol	İşlev
Win + . (nokta)	emoji ve simge paneli
Shift + Delete	kalıcı silme (Geri Dönüşüm Kutusu'na gitmez)
Ctrl + T	Dosya Gezgini'nde yeni sekme
sağ tık → Sıkıştır → ZIP dosyası	seçili öğeleri ZIP arşivine sıkıştırma

Tablo 2.2.6: Diğer kullanışlı kısayollar.

Shift + Delete'in geri alınamadığı unutulmamalıdır. Dosya Geri Dönüşüm Kutusu'na gönderilmeden silinir; Ctrl + Z de bu işlemi geri getirmez. Dosya Gezgini'ndeki sekmeler ise güncel Windows 11 sürümlerinde bulunur. Bir tarayıcıdaki gibi, tek pencerede birden çok klasörü sekmelerde açık tutmayı sağlar. Böylece iki klasör arasında dosya taşırken iki ayrı pencere açmaya gerek kalmaz.

Sık yapılan hata

Win + D ile Win + L'yi aynı amaçla kullanmak. Win + D yalnızca pencereleri gizler; bilgisayarın başına gelen herkes bir tıklamayla belgeleri yeniden açabilir. Başkalarının bulunduğu bir ortamda bilgisayardan ayrılırken ekran Win + L ile kilitlenir.

2.2.3.3 Ekran görüntüsü alma

Bir hata iletisini bildirmek, bir ayarın nerede olduğunu göstermek ya da bir grafiği rapora eklemek için ekranın görüntüsünü almak gerekir. Windows'ta bunun birkaç yolu vardır. Bu yollar iki soruya göre ayrılır: *ne* yakalanır ve görüntü *nereye* gider (Tablo 2.2.7).

Tanım 2.2.10

Ekran görüntüsü (*screenshot*), ekranın tamamının ya da bir bölümünün o anki görüntüsünün resim olarak kaydedilmesidir. Windows'ta ekran görüntüleri **Ekran Alıntısı Aracı** (*Snipping Tool*) ve PrtSc (Print Screen) tuşuyla alınır.

Kısayol	Ne yakalanır?	Görüntü nereye gider?
Win + Shift + S	seçilen alan: dikdörtgen, serbest biçim, pencere ya da tam ekran	panoya; bildirimden açılıp düzenlenebilir ve kaydedilebilir
Win + PrtSc	tüm ekran	dosya olarak Resimler\Ekran Görüntüleri klasörüne (PNG)
Alt + PrtSc	yalnızca etkin pencere	panoya; bir belgeye Ctrl + V ile yapıştırılır
PrtSc	güncel Windows 11'de Ekran Alıntısı Aracı'nı açar	bu seçenek kapatılırsa tüm ekran panoya kopyalanır

Tablo 2.2.7: Ekran görüntüsü alma yolları.

Tablodaki en önemli ayrım, görüntünün *dosya* mı yoksa *pano* içeriği mi olduğudur. Win + PrtSc bir PNG dosyası oluşturur; bilgisayar kapatılsa da dosya durur. Öteki yollar ise görüntüyü panoya alır. Panodaki görüntü bir dosya değildir. Bir sonraki kopyalamada üzerine yazılır, bilgisayar yeniden başlatılınca da kaybolur. Saklamak için görüntüyü bir belgeye ya da Paint gibi bir programa yapıştırıp kaydetmek ya da Win + V pano geçmişini kullanmak gerekir. Win + Shift + S ile alınan görüntüden sonra köşede bir bildirim belirir. Bu bildirime tıklamak görüntüyü Ekran Alıntısı Aracı'nda açar; görüntü burada kırpılıp üzerine çizilebilir ve dosya olarak kaydedilebilir.

Sık yapılan hata

Ekran görüntüsünü kontrol etmeden paylaşmak. Görüntüde e-posta adresi, parola ipucu, açık bir sohbet bildirimi ya da başka sekmelerin başlıkları kalabilir. Paylaşmadan önce görüntüye bakılır; gerekirse kişisel bilgiler kırpılarak ya da üzeri boyanarak çıkarılır.

2.2.4 Sürücü, dosya ve klasör

Bilgisayardaki her belge, fotoğraf ve program diskte bir dosya olarak durur. İşletim sistemi bu dosyaları klasörlere, klasörleri de sürücülere yerleştirir. Bu kesim, bu düzenin yapı taşlarını aşağıdan yukarıya, yani fiziksel diskten dosya adına doğru anlatır.

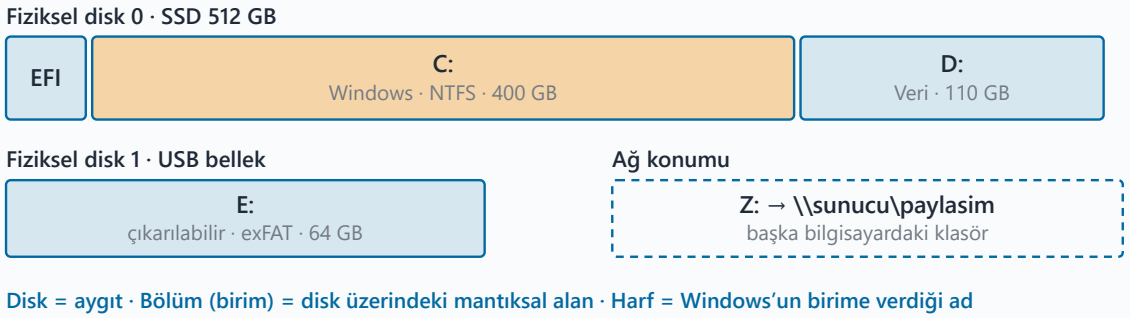
2.2.4.1 Disk, bölüm ve sürücü harfi

Günlük dilde “disk”, “sürücü” ve “C:” sözcükleri birbirinin yerine kullanılır. Oysa bunlar üç ayrı kavramdır.

Tanım 2.2.11

Fiziksel disk, verinin saklandığı aygıtın kendisidir: bir SSD, bir HDD ya da bir USB bellek. **Bölüm** ya da **birim** (*partition / volume*), diskin üzerinde ayrılmış mantıksal bir alandır; bir disk birden çok bölüme ayrılabilir. **Sürücü harfi** ise Windows'un bir birime verdiği addır: C:, D:, E:. Dosya Gezgini'nde her harf ayrı bir **sürücü** (*drive*) olarak görünür.

Şekil 2.2.9 bu üç kavramın ilişkisini gösterir. Fiziksel disk 0, 512 GB'lık bir SSD'dir. Bu disk üç bölüme ayrılmıştır. İlki küçük bir EFI bölümüdür; bilgisayarın açılışında kullanılır ve harfi yoktur. İkincisi Windows'un kurulu olduğu 400 GB'lık C: birimidir. Üçüncüsü veriler için ayrılmış 110 GB'lık D: birimidir. Fiziksel disk 1 ise 64 GB'lık bir USB bellektir ve tek bölümü E: harfini alır. En sağdaki Z: ise bir disk bile değildir: ağdaki başka bir bilgisayarın paylaştığı klasör, bu harfe bağlanmıştır.



Şekil 2.2.9: İki fiziksel disk, bölümleri ve sürücü harfleri. Disk 0 (SSD, 512 GB) üzerinde harfsiz EFI bölümü, Windows'un kurulu olduğu C: (NTFS, 400 GB) ve veri birimi D: (110 GB) vardır. Disk 1 (USB bellek) çıkarılabilir E: birimidir (exFAT, 64 GB). Z: harfi ağdaki \\sunucu\paylasim klasörüne bağlanmıştır.

Şekildeki **NTFS** ve **exFAT** birer dosya sistemi adıdır. Dosya sistemi, bir birimdeki dosyaların adlarını, boyutlarını, konumlarını ve izinlerini tutan düzendir. Windows kendi birimlerinde NTFS kullanır; NTFS dosya izinlerini, şifrelemeyi ve büyük dosyaları destekler. USB bellek ve hafıza kartlarında ise genellikle exFAT kullanılır, çünkü exFAT'ı Windows, macOS ve birçok aygıt okuyup yazabilir.

Windows çoğunlukla C: birimine kurulur ve harfler genellikle sırayla dağıtılır. A: ve B: harfleri tarihsel olarak disket sürücülerine ayrılmıştı; bu yüzden ilk sabit disk birimi bugün de C: harfini alır. Bir USB bellek takıldığında ilk boş harf (bu örnekte E:) ona verilir. Ağ paylaşımına ise kullanıcı genellikle alfabenin sonundan bir harf seçer.

Örnek 2.2.5

Kutusunda "512 GB" yazan bir SSD tek bölüm olarak C: yapılmıştır. Windows bu birimin kapasitesini neden yaklaşık 476 GB gösterir?

1. Disk üreticileri ondalık birim kullanır: 1 GB = 10^9 bayt. Diskte $512 \times 10^9 = 512.000.000.000$ bayt vardır.
2. Windows ise ikili birimle hesaplar: 1 GiB = 2^{30} = 1.073.741.824 bayt. Ekranda "GB" yazsa da kastedilen GiB'dir.
3. $512.000.000.000 \div 1.073.741.824 \approx 476{,}8$. Windows bu sayıyı "476 GB" diye gösterir.

Diskten hiçbir bayt kaybolmamıştır; yalnızca birim farklıdır. EFI ve kurtarma bölümleri de birkaç yüz MB ile birkaç GB arasında yer kaplar. Bu yüzden C: birimi biraz daha küçük görünebilir.

Sık yapılan hata

C: ile D: birimlerinin ayrı diskler olduğunu sanmak. Aynı fiziksel diskin iki bölümü de olabilirler. Bu durumda dosyaları C:'den D:'ye kopyalamak yedek almak sayılmaz: disk bozulursa iki

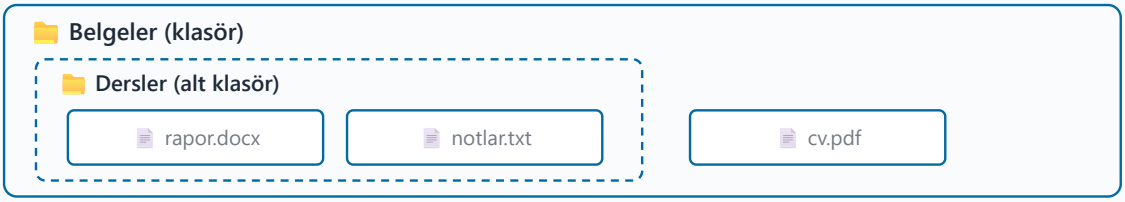
birim birlikte kaybolur. Gerçek bir yedek ayrı bir aygıtta ya da bulutta tutulur.

2.2.4.2 Dosya ve klasör

Tanım 2.2.12

Dosya (*file*), adı, içeriği ve özellikleri olan veri birimidir; örneğin `rapor.docx`. **Klasör** (*folder*), dosyaları ve başka klasörleri bir arada tutan mantıksal kapsayıcıdır; örneğin `Belgeler\Dersler`. Bir klasörün içindeki klasöre **alt klasör** denir.

Klasörler iç içe yerleştirilebildiği için dosyalar bir ağaç biçiminde düzenlenir. Şekil 2.2.10 küçük bir örnek verir. `Belgeler` klasörünün içinde `Dersler` alt klasörü, onun içinde de üç dosya vardır.



Şekil 2.2.10: Belgeler klasörünün içinde Dersler alt klasörü, onun içinde rapor.docx, notlar.txt ve cv.pdf dosyaları.

Klasörün kendisi bir belge içeriği taşımaz. Bir klasör, içindeki dosyaların adlarını ve diskteki yerlerini bilen bir liste gibidir. Bir kitaplıktaki raflara benzer: raf kitap değildir, ama kitapları bulunur kılar. Klasör düzeni iyi kurulursa, yüzlerce dosya arasından aranan dosya birkaç tıklamayla bulunur.

2.2.4.3 Dosya adı ve uzantı

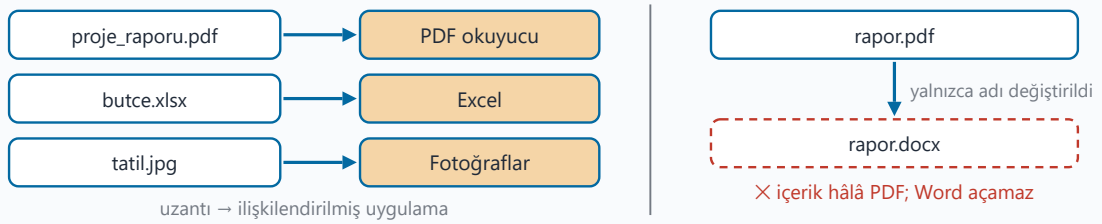
Bir dosyanın tam adı iki parçadan oluşur: `proje_raporu.pdf` adında, son noktaya kadar olan kısım **dosya adı**, son noktadan sonraki kısım **uzantıdır**.

Tanım 2.2.13

Uzantı (*file name extension*), dosya adının son noktadan sonraki kısmıdır ve dosyanın biçimi hakkında ipucu verir: `.pdf`, `.docx`, `.jpg`. Windows, bir dosyaya çift tıklandığında hangi uygulamanın açılacağını uzantıya bakarak belirler. Bir uzantı ile bir uygulama arasındaki bu bağa **ilişkilendirme** (*file association*) denir.

Dosya adını kullanıcı içeriği tanımlamak için verir. Uzantıyı ise genellikle dosyayı oluşturan program koyar: Word `.docx`, Excel `.xlsx`, bir fotoğraf makinesi `.jpg` ekler. Windows uzantıları varsayılan olarak gizler. Dosya Gezgini'nde Görünüm → Göster → Dosya adı uzantıları seçeneği işaretlenince uzantılar görünür olur (2.2.6.2).

Şekil 2.2.11 uzantının iki yüzünü gösterir. Üstte her uzantı ilişkilendirilmiş uygulamaya gider: `.pdf` PDF okuyucuya, `.xlsx` Excel'e, `.jpg` Fotoğraflar uygulamasına. Altta ise bir PDF dosyasının adı `rapor.docx` olarak değiştirilmiştir. Windows artık bu dosyayı Word'le açmaya çalışır, ama içerik hâlâ PDF olduğu için Word dosyayı tanıyamaz.



Şekil 2.2.11: Uzantı dosyanın hangi uygulamayla açılacağını belirler: proje_raporu.pdf → PDF okuyucu, butce.xlsx → Excel, tatil.jpg → Fotoğraflar. rapor.pdf adının rapor.docx yapılması içeriği dönüştürmez; içerik hâlâ PDF'tir ve Word dosyayı açamaz.

Örnek 2.2.6

Uzantılar görünürken `foto.jpg` dosyası F2 ile `foto.png` olarak yeniden adlandırılıyor. Ne olur?

1. Windows, uzantı değişirse dosyanın kullanılamaz hâle gelebileceğini bildiren bir uyarı gösterir. Evet seçilince ad değişir.
2. Dosyanın içindeki baytlar değişmez. İçerik hâlâ JPEG biçiminde sıkıştırılmış bir resimdir; boyutu da aynı kalır.
3. Windows dosyayı artık PNG sanır. Çoğu resim programı içeriğin ilk baytlarına bakıp biçimi yine de tanır, ama bazı programlar ve web siteleri dosyayı reddedebilir.

Gerçek dönüştürme, dosyayı açan bir programda “Farklı Kaydet” ya da “Dışa Aktar” komutuyla yapılır. Örneğin Paint'te resim açılıp PNG olarak kaydedilir. Bu işlem içeriği yeniden kodlar ve yeni bir dosya üretir.

Sık yapılan hata

Uzantılar gizliyken dosyayı yeniden adlandırıp sonuna uzantı eklemek. Gizli uzantılı `odev.docx` dosyası ekranda `odev` görünür. Kullanıcı adı `odev.pdf` yaparsa dosyanın gerçek adı `odev.pdf.docx` olur. Dosya hâlâ Word belgesidir ve PDF bekleyen bir sisteme yüklenince

reddedilir. Benzer bir oyunu kötü amaçlı dosyalar da kullanır: `fatura.pdf.exe` adlı bir program, uzantılar gizliyken `fatura.pdf` gibi görünür. Bu yüzden uzantıları görünür tutmak güvenli bir alışkanlıktır.

2.2.4.4 Dosya adlandırma kuralları

Windows dosya adlarına bazı sınırlar koyar. Bu sınırların bir kısmı zorunludur; Windows kurala uymayan adı kabul etmez. Bir kısmı ise yalnızca iyi alışkanlıktır.

Yasak karakterler. Dokuz karakter dosya adında kullanılamaz: `\ / : * ? " < > |`. Bunların çoğunun yolda ve aramada özel bir anlamı vardır. `\` klasörleri ayırır, `:` sürücü harfinden sonra gelir, `*` ve `?` aramada joker karakterdir. Bu karakterlerden biri yazılmaya çalışıldığında Windows bir uyarı gösterir ve karakteri kabul etmez.

Ayrılmış adlar. `CON`, `PRN`, `AUX`, `NUL`, `COM1` … `COM9` ve `LPT1` … `LPT9` eski aygıt adlarıdır. Bu adlar uzantılı olsa bile dosya adı olamaz; `con.txt` adı da reddedilir.

Büyük ve küçük harf. Windows, `Rapor.docx` ile `rapor.docx` adlarını aynı ad sayar. Bu yüzden ikisi aynı klasörde bulunamaz. Ad yazılırken büyük harf korunur, ama karşılaştırmada harf büyüklüğüne bakılmaz.

Yol uzunluğu. Klasik sınır, sürücü harfinden dosya adının sonuna kadar tam yol için 260 karakterdir. Çok derin klasör yapılarında ya da çok uzun adlarda kopyalama ve açma işlemleri bu sınır yüzünden hata verebilir.

Örnek 2.2.7

Altı dosya adı kurallara göre sınanıyor:

Ad	Geçerli mi?	Neden
`rapor:son.docx`	hayır	`:` yasak karakterdir
`soru?.txt`	hayır	`?` yasak karakterdir
`CON.txt`	hayır	CON ayrılmış addır; uzantı durumu değiştirmez
`console.txt`	evet	ad CON ile başlar ama CON değildir
`Rapor.docx` (klasörde `rapor.docx` varken)	hayır	büyük-küçük harf farkı adları ayırmaz; ad zaten var
`ödev 1 .pdf`	evet	kurala uyar, ama boşluk ve Türkçe karakter sorun çıkarabilir

Son satırdaki ad Windows'ta geçerlidir. Ancak e-posta ekinde, bir web sitesine yüklemede ya da başka bir işletim sisteminde boşluklar ve `ö` gibi harfler bozulabilir. Uzantıdan önceki boşluk da adın yanlış okunmasına yol açar.

Zorunlu kuralların dışında, iyi bir dosya adı içeriği ve sürümü anlatır. Tablo 2.2.8 iki zayıf adı daha iyi karşılıklarıyla karşılaştırır.

Zayıf ad	Daha iyi ad	Neden?
`son hali YENİ (2).docx`	`2026-10-17_rapor_v2.docx`	tarih başta: ada göre sıralama tarihe göre sıralama olur
`ödev 1 .pdf`	`odev1_ad-soyad.pdf`	boşluk ve Türkçe karakter e-posta ve web sistemlerinde sorun çıkarabilir

Tablo 2.2.8: Zayıf ve daha iyi dosya adları.

İlk satırdaki tarih biçimi özellikle yararlıdır. Tarih yıl-ay-gün sırasıyla ve iki basamaklı ay ve günle yazılırsa, adlar alfabetik sıralandığında tarih sırasına da girer. `2026-09-30`, `2026-10-02` ve `2026-10-17` adları alfabetik olarak da bu sırayla dizilir. Gün-ay-yıl biçimi (`17.10.2026`) bu özelliği taşımaz: `02.10.2026` adı `30.09.2026`'dan önce gelir.

> **Sık yapılan hata**
>
> Sürümleri "son", "son hali", "gerçek son" gibi adlarla ayırmak. Birkaç gün sonra hangisinin en yeni olduğu anlaşılmaz. Sürüm numarası (`_v1`, `_v2`) ya da tarih kullanmak sürümleri sıralı ve açık tutar.

2.2.4.5 Yaygın dosya türleri

Uzantıyı tanımak, bir dosyanın ne olduğunu açmadan anlamayı sağlar. Tablo 2.2.9 en sık karşılaşılan türleri kategorilerine göre listeler.

Kategori	Uzantılar	Örnek kullanım
Belge	.docx, .pdf, .txt	metin ve rapor
Tablo	.xlsx, .csv	hesap ve veri
Sunu	.pptx	slayt gösterisi
Görüntü	.jpg, .png, .gif	fotoğraf ve grafik
Ses / video	.mp3, .wav, .mp4	medya
Arşiv	.zip, .7z	sıkıştırılmış dosyalar
Çalıştırılabilir	.exe, .msi	program ya da kurulum

Tablo 2.2.9: Yaygın dosya türleri.

Bazı türler arasındaki fark, ilk bakışta görülmeyen ayrıntılardadır. `.txt` yalnızca düz metin tutar; yazı tipi, kalın yazı ya da resim içermez. `.docx` ise biçimlendirmeyi, resimleri ve tabloları da saklar. `.csv` virgül ya da noktalı virgülle ayrılmış değerlerden oluşan düz metindir; Excel onu tablo olarak açar ama formül ve biçim saklamaz. `.pdf` ise sayfayı her bilgisayarda aynı görünecek biçimde dondurur; bu yüzden teslim edilen ödevler için uygundur.

Görüntü türleri arasında da önemli bir ayrım vardır. `.jpg` (JPEG) fotoğraflar için tasarlanmıştır ve kayıplı sıkıştırma kullanır: dosya küçülür, ama gözün zor fark edeceği ayrıntılar atılır. `.png` kayıpsızdır; ekran görüntüsü, çizim ve keskin kenarlı grafikler için daha uygundur.

Son satırdaki çalıştırılabilir dosyalar özel dikkat ister. Bir `.exe` ya da `.msi` dosyası çift tıklanınca bilgisayarda bir program çalıştırır. Bir belge zarar veremez sanılabilir, ama çalıştırılabilir dosya bilgisayardaki her şeye erişebilir. Kaynağı bilinmeyen çalıştırılabilir dosyalar ve e-posta ekleri açılmamalıdır (2.2.9.3).

2.2.5 Dosya yolları

Klasörler iç içe geçtiği için bir dosyanın yerini tek bir sözcükle söylemek mümkün değildir. "rapor.pdf" adlı onlarca dosya farklı klasörlerde durabilir. Bir dosyayı kesin olarak belirtmek için, sürücüden başlayıp dosyaya kadar inen klasörlerin sırası verilir. Bu sıraya dosyanın yolu denir.

2.2.5.1 Yolun parçaları

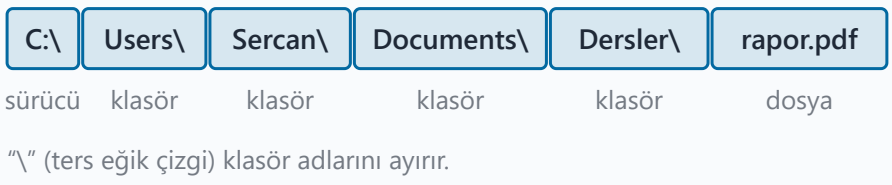
Tanım 2.2.14

Yol (*path*), bir dosyaya ya da klasöre ulaşmak için geçilen klasörlerin, ters eğik çizgiyle (`\`) ayrılmış sırasıdır. **Mutlak yol** (*absolute path*) sürücünün kök klasöründen başlar ve nerede olunursa olunsun aynı dosyayı gösterir. **Göreli yol** (*relative path*) ise o anda içinde bulunulan klasöre, yani **geçerli klasöre** göre yazılır.

Örnek bir mutlak yol şudur:

```
C:\Users\Sercan\Documents\Dersler\rapor.pdf
```

Şekil 2.2.12 bu yolu parçalarına ayırır. `C:\` sürücünün **kök** (*root*) klasörüdür. Sonra sırayla `Users`, `Sercan`, `Documents` ve `Dersler` klasörleri gelir. En sondaki `rapor.pdf` dosyanın kendisidir. Her `\` bir alt düzeye inildiğini gösterir.



Şekil 2.2.12: Dosya yolunun parçaları: sürücü (C:\), dört klasör (Users, Sercan, Documents, Dersler) ve dosya (rapor.pdf). Ters eğik çizgi klasör adlarını ayırır.

Bu yolda dikkat çeken bir ayrıntı var. Türkçe Windows'ta Dosya Gezgini bu klasörü “Belgeler” diye gösterir, ama klasörün diskteki gerçek adı `Documents`'tır. Aynı durum `Desktop` (Masaüstü), `Downloads` (İndirilenler), `Pictures` (Resimler), `Music` (Müzik) ve `Users` (Kullanıcılar) için de geçerlidir. Windows bu klasörlerin adlarını arayüz diline çevirerek gösterir, ama yol yazılırken gerçek adlar kullanılır. Bu yüzden `C:\Users\Sercan\Documents` ile Gezgin'deki “Belgeler” aynı klasördür.

Yolun uzunluğu da sayılabilir. Bu örnekteki yolun 43 karakteri vardır ve içinde 5 ters eğik çizgi bulunur. Klasik 260 karakter sınırına göre çok kısadır. Ancak her klasör adı uzadıkça ve iç içe klasör sayısı arttıkça sınıra yaklaşılır. Örneğin bir ders projesinin klasörü `C:\Users\Sercan\Documents\…` altında on düzey derinlikteyse ve her düzeyin adı 25 karakterse, yol tek başına 250 karakteri aşar.

2.2.5.2 Mutlak yolun çözümlenmesi

Bir yolu çözümlemek, yolun gösterdiği öğeyi bulmak demektir. Mutlak yol soldan sağa okunur ve her parça bir alt düzeye iner. Şekil 2.2.13 bu okumayı klasör ağacı üzerinde gösterir. Tablo 2.2.10'da ise her adımda okunan parça ve varılan konum yazılıdır.

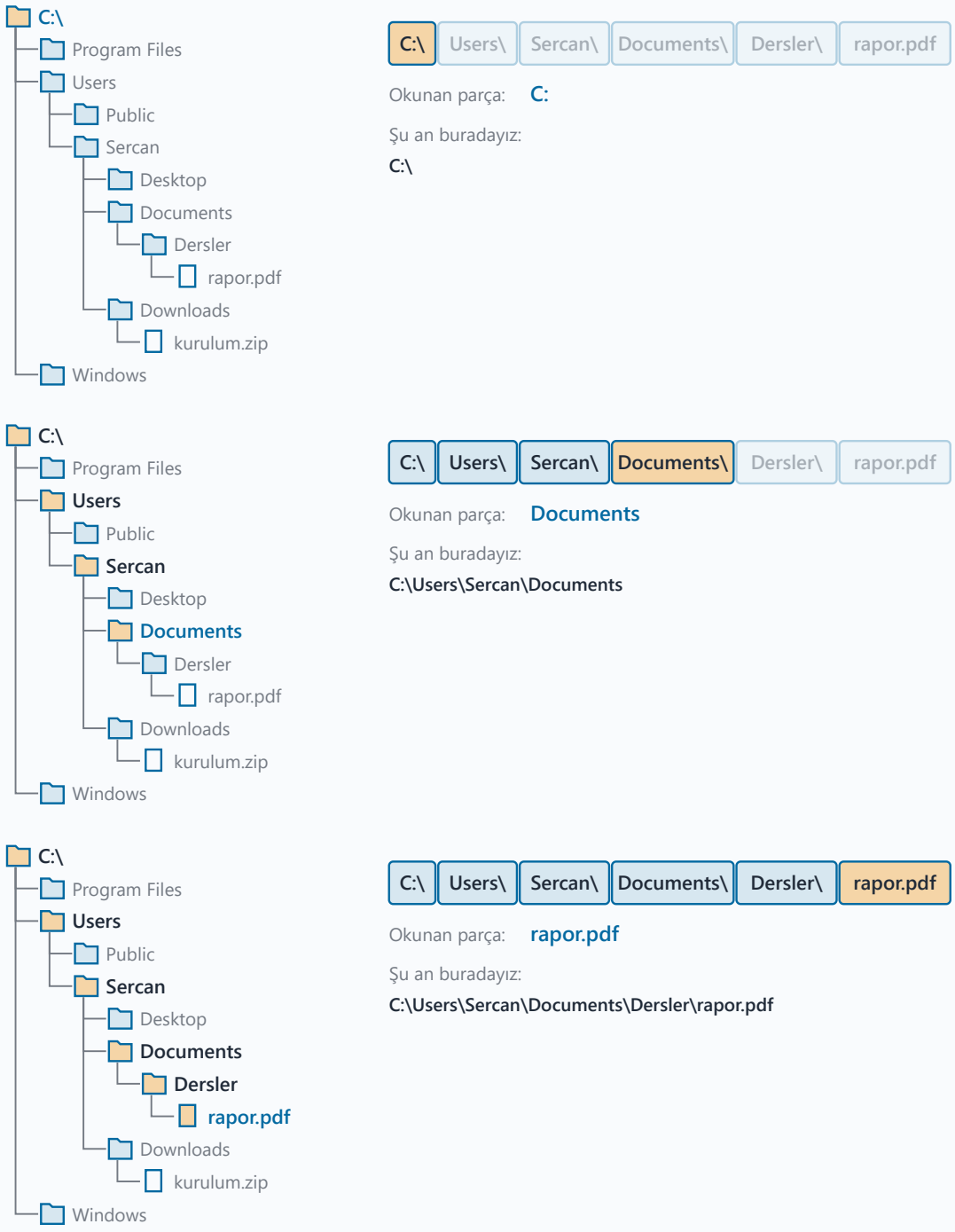
Örnek 2.2.8

`C:\Users\Sercan\Documents\Dersler\rapor.pdf` yolu adım adım çözümleniyor.

Adım	Okunan parça	Konum	Açıklama
1	`C:`	`C:\`	sürücünün kök klasörü; arama buradan başlar
2	`Users`	`C:\Users`	kökteki Program Files ve Windows atlanır
3	`Sercan`	`C:\Users\Sercan`	kullanıcının kendi klasörü
4	`Documents`	`C:\Users\Sercan\Documents`	arayüzde “Belgeler” görünen klasör
5	`Dersler`	`…\Documents\Dersler`	alt klasöre inilir
6	`rapor.pdf`	`…\Dersler\rapor.pdf`	son parça bir dosyadır; bulundu

Tablo 2.2.10: Mutlak yolun çözümlenmesi.

Her adımda yalnızca o anki klasörün içine bakılır. İkinci adımda kökte `Program Files`, `Users` ve `Windows` klasörleri vardır; yol `Users`'ı istediği için öbürleri hiç açılmaz. Yol bir harita gibi çalışır: yanlış klasöre sapma olmaz.



Şekil 2.2.13: Mutlak yolun ağaçta çözümlenmesi. Üstte okuma C:\ kökünde başlar. Ortada Users ve Sercan'dan geçilip Documents'a inilmiştir. Altta Dersler'e inilmiş ve son parça olan rapor.pdf dosyası bulunmuştur.

2.2.5.3 Göreli yolun çözümlenmesi

Göreli yol kökten değil, geçerli klasörden başlar. İki özel ad içerebilir: `..` (iki nokta) **üst klasör** (*parent directory*) demektir, yani bir düzey yukarı çıkılır. `.` (tek nokta) ise geçerli klasörün kendisidir. Göreli yollar özellikle programlamada ve komut satırında sık kullanılır. Bir proje klasörü başka bir bilgisayara taşındığında mutlak yollar bozulur, ama proje içindeki göreli yollar çalışmaya devam eder.

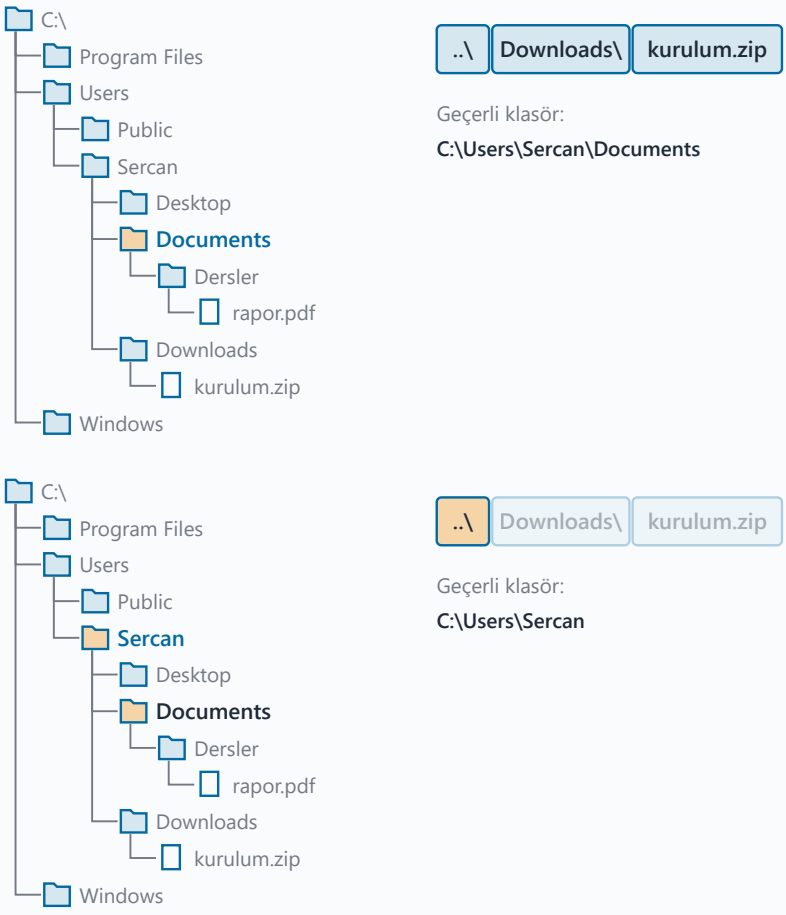
Örnek 2.2.9

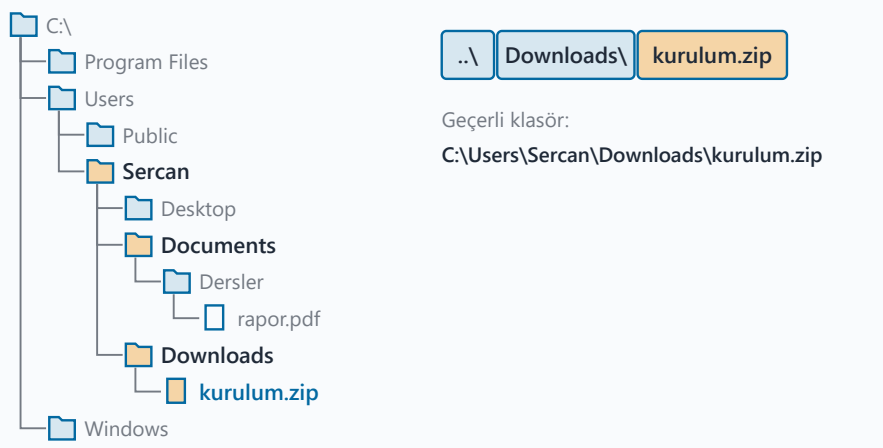
Geçerli klasör `C:\Users\Sercan\Documents`. `..\Downloads\kurulum.zip` göreli yolu çözümleniyor.

Adım	Okunan parça	Geçerli konum
0	—	`C:\Users\Sercan\Documents`
1	`..`	`C:\Users\Sercan` (bir üst klasöre çıkıldı)
2	`Downloads`	`C:\Users\Sercan\Downloads`
3	`kurulum.zip`	`C:\Users\Sercan\Downloads\kurulum.zip` (dosya bulundu)

Tablo 2.2.11: Göreli yolun çözümlenmesi.

Göreli yolun mutlak karşılığı, son satırdaki `C:\Users\Sercan\Downloads\kurulum.zip` yoludur. Bu karşılık şöyle de bulunur: geçerli klasörün yolundan her `..` için son parça silinir, sonra kalan parçalar sırayla eklenir.





Şekil 2.2.14: Göreli yolun ağaçta çözümlenmesi. Üstte geçerli klasör Documents'tır. Ortada .. ile Sercan'a çıkılmıştır. Altta Downloads'a inilmiş ve kurulum.zip dosyası bulunmuştur.

Birden çok `..` art arda gelebilir. Her biri bir düzey daha yukarı çıkar. Bunu daha uzun bir yol üzerinde görelim.

Örnek 2.2.10

Geçerli klasör `C:\Users\Sercan\Downloads`. `..\Documents\Dersler\rapor.pdf` yolunun mutlak karşılığı nedir? Geçerli klasör `C:\Users\Sercan\Documents\Dersler` olsaydı `..\..\Pictures\Tatil\deniz.jpg` neyi gösterirdi?

1. Birinci yol: `..` ile `Downloads`'tan `C:\Users\Sercan`'a çıkılır. Sonra `Documents`, `Dersler` ve `rapor.pdf` eklenir. Sonuç: `C:\Users\Sercan\Documents\Dersler\rapor.pdf`. Bu, 2.2.5.2'deki mutlak yolun ta kendisidir.
2. İkinci yol: ilk `..` ile `Dersler`'den `Documents`'a, ikinci `..` ile `Documents`'tan `Sercan`'a çıkılır.
3. Sonra `Pictures` (arayüzde Resimler) ve `Tatil` klasörlerine inilir. Sonuç: `C:\Users\Sercan\Pictures\Tatil\deniz.jpg`.

Aynı dosyaya farklı geçerli klasörlerden farklı göreli yollarla ulaşılır. Mutlak yol ise her zaman aynıdır.

Sık yapılan hata

Göreli yolun başındaki `..\`'ı unutmak. Geçerli klasör `Documents` iken `Downloads\kurulum.zip` yazılırsa, `Downloads` klasörü `Documents`'ın *içinde* aranır. Böyle bir alt klasör olmadığı için dosya bulunamaz. Göreli yol yazmadan önce geçerli klasörün neresi olduğu kesin olarak bilinmelidir.

2.2.6 Dosya Gezgini

Klasör ağacında gezinmenin, dosyaları görmenin ve düzenlemenin temel aracı Dosya Gezgini'dir. Görev çubuğundaki klasör simgesiyle ya da Win + E ile açılır.

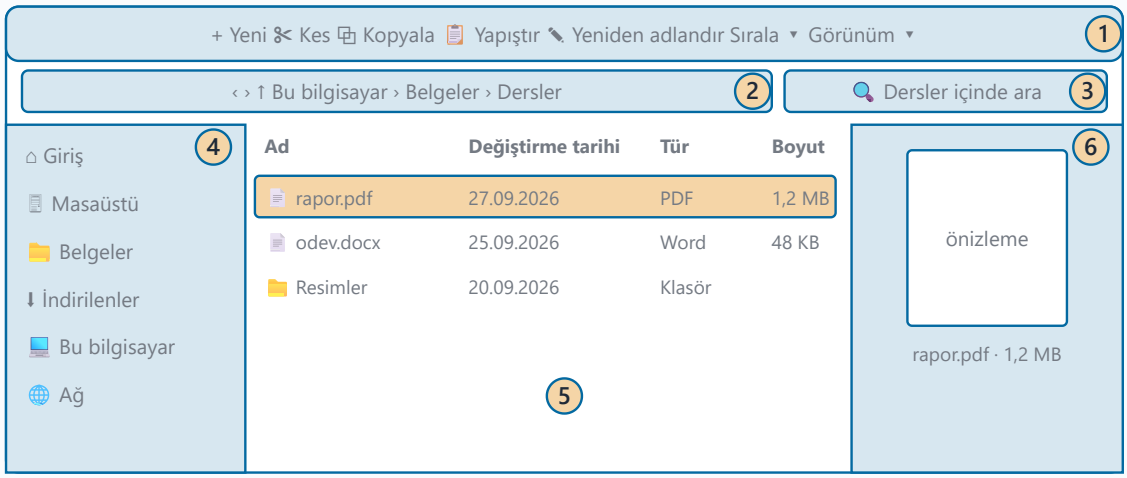
2.2.6.1 Dosya Gezgini'nin bölmeleri

Tanım 2.2.15

Dosya Gezgini (*File Explorer*), sürücüleri, klasörleri ve dosyaları görüntülemeye, açmaya, kopyalamaya, taşımaya, yeniden adlandırmaya, silmeye ve aramaya yarayan Windows uygulamasıdır.

Şekil 2.2.15 Dosya Gezgini penceresini altı bölmeye ayırır:

1. **Komut çubuğu** (*command bar*): Yeni (klasör ya da dosya), Kes, Kopyala, Yapıştır, Yeniden adlandır, Sırala ve Görünüm komutları.
2. **Adres çubuğu** (*address bar*): geçerli klasörün yolu. Yol, tıklanabilir parçalar hâlinde gösterilir: *Bu bilgisayar › Belgeler › Dersler*. Bir parçaya tıklamak o klasöre döner. Çubuğun boş bir yerine tıklanınca tam yol metin olarak görünür; buraya bir yol yazıp Enter'a basmak doğrudan o klasöre gider.
3. **Arama kutusu**: geçerli klasörde ve onun bütün alt klasörlerinde arar.
4. **Gezinti bölmesi** (*navigation pane*): soldaki sütun. Giriş, Masaüstü, Belgeler, İndirilenler, Bu bilgisayar ve Ağ gibi sık kullanılan konumlara tek tıklamayla gider.
5. **Dosya listesi**: geçerli klasörün içeriği. *Ayrıntılar* görünümünde her satırda ad, değiştirme tarihi, tür ve boyut yazar. Sütun başlığına tıklamak listeyi o sütuna göre sıralar; bir kez daha tıklamak sırayı tersine çevirir.
6. **Önizleme / ayrıntı bölmesi**: sağdaki isteğe bağlı sütun. Seçili dosyanın içeriğini açmadan gösterir ya da özelliklerini listeler. Görünüm menüsünden açılıp kapatılır.



Şekil 2.2.15: Dosya Gezgini'nin bölmeleri: (1) komut çubuğu, (2) adres çubuğu, (3) arama, (4) gezinti bölmesi, (5) Ayrıntılar görünümündeki dosya listesi, (6) seçili rapor.pdf dosyasının önizlemesi.

Adres çubuğunun solundaki üç ok gezinmeyi kolaylaştırır. ← (Geri) ve → (İleri), tarayıcıdaki gibi ziyaret edilen klasörler arasında gidip gelir. ↑ (Yukarı) ise geçerli klasörün üst klasörüne çıkar; yani bir göreli yoldaki `..`'ın Gezgin'deki karşılığıdır. Gezinti bölmesindeki *Bu bilgisayar* seçildiğinde dosya listesinde sürücüler ve her birinin doluluk çubuğu görünür.

2.2.6.2 Uzantılar, gizli öğeler, yeni klasör ve arama

Dosya Gezgini'nde en sık yapılan işlerden beşini bir örnek üzerinde izleyelim. Başlangıçta `Belgeler\Dersler` klasöründe dört dosya var: `foto.jpg` (2.410 KB), `odev2.pdf` (820 KB), bir Word belgesi olan `rapor` (480 KB) ve bir PDF belgesi olan `rapor` (610 KB). Ayrıca gizli niteliği verilmiş bir `taslak.txt` (2 KB) var, ama listede görünmüyor. Tablo 2.2.12 her adımın sonucunu, Şekil 2.2.16 ise üç önemli anı gösterir.

Örnek 2.2.11

Adım	İşlem	Listede görünenler
1	Görünüm → Göster → *Dosya adı uzantıları* işaretlenir	foto.jpg, odev2.pdf, rapor.docx, rapor.pdf
2	Görünüm → Göster → *Gizli öğeler* işaretlenir	önceki dört dosya + soluk simgeli taslak.txt
3	Yeni → Klasör, ad olarak `Odevler` yazılıp Enter	Odevler (en üstte), sonra beş dosya
4	rapor.docx, Odevler klasörünün üzerine sürüklenip bırakılır	Odevler, foto.jpg, odev2.pdf, rapor.pdf, taslak.txt
5	arama kutusuna `*.pdf` yazılır	odev2.pdf, rapor.pdf

Tablo 2.2.12: Dosya Gezgini'nde beş işlem ve sonuçları.

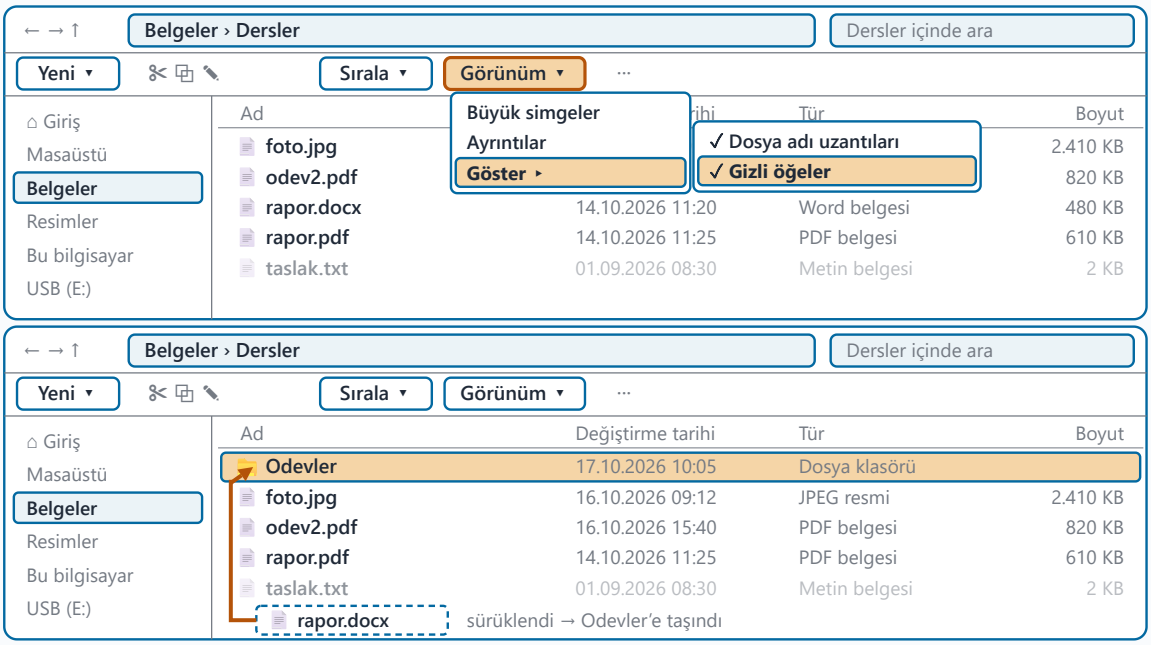
Adım 1. Uzantılar gizliyken iki dosyanın adı da "rapor" görünüyordu. Yalnızca simgeleri ve Tür sütunu (Word belgesi, PDF belgesi) onları ayırıyordu. Uzantılar açılınca adlar `rapor.docx` ve `rapor.pdf` olarak ayrışır.

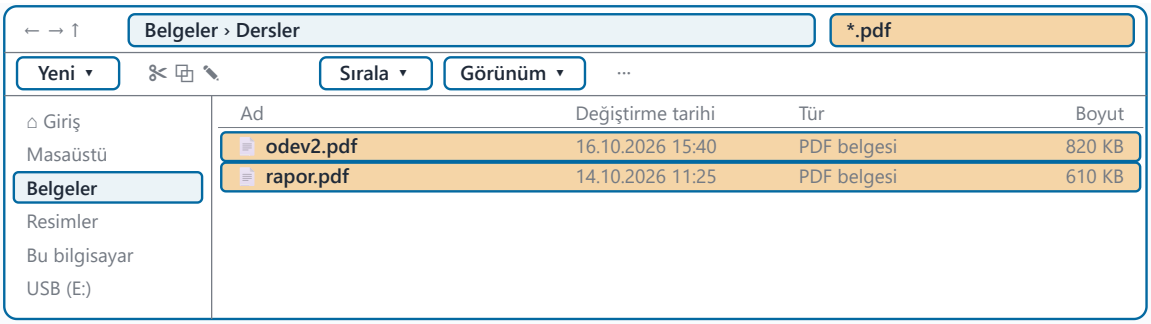
Adım 2. Gizli öğeler açılınca `taslak.txt` soluk bir simgeyle listeye girer. Soluk simge, öğenin gizli niteliği taşıdığını belirtir.

Adım 3. Yeni klasör "Yeni klasör" adıyla oluşur ve adı seçili gelir. Doğrudan `Odevler` yazılıp Enter'a basılır; ayrıca F2'ye basmaya gerek yoktur. Ada göre sıralamada klasörler dosyalardan önce listelendiği için Odevler en üste yerleşir.

Adım 4. `rapor.docx` aynı sürücüdeki bir klasöre sürüklendiği için taşınır. Dosya Dersler'in listesinden kalkar ve Odevler'in içine girer.

Adım 5. `*` bir **joker karakterdir** (*wildcard*) ve "herhangi bir ad" anlamına gelir. `*.pdf` araması Dersler'de ve alt klasörlerinde adı `.pdf` ile biten bütün dosyaları bulur. Arama kutusu temizlenince tam liste geri gelir.





Şekil 2.2.16: Dosya Gezgini'nde üç an. Üstte Dosya adı uzantıları ve Gizli öğeler işaretlenmiştir; gizli taslak.txt listede görünür. Ortada Odevler klasörü oluşturulmuş ve rapor.docx onun içine taşınmıştır. Altta arama kutusuna *.pdf yazılmış, yalnızca iki PDF dosyası listelenmiştir.

Joker karakter farklı biçimlerde de kullanılır. Aynı klasörde `rapor.*` araması, adı "rapor" olan bütün dosyaları türüne bakmadan bulur: `rapor.docx` ve `rapor.pdf`. `odev*` ise adı "odev" ile başlayan her şeyi bulur: bu örnekte `Odevler` klasörü ve `odev2.pdf`. Arama kutusuna joker karakter olmadan bir sözcük yazmak da mümkündür. Bu durumda Windows adında ya da (dizinlenmiş konumlarda) içeriğinde o sözcük geçen dosyaları getirir.

Sık yapılan hata

Gizli niteliğin dosyayı koruduğunu sanmak. Gizli nitelik dosyayı yalnızca listeden saklar. "Gizli öğeler" seçeneğini açan herkes dosyayı görür ve açar. Bir dosyayı başkalarından korumak için ayrı kullanıcı hesapları, dosya izinleri ya da şifreleme kullanılır (2.2.8.3, 2.2.10.5).

2.2.7 Dosya işlemleri

Dosyalar oluşturulduktan sonra yerleri değişir, adları düzeltilir, gereksiz olanlar silinir, paylaşılacak olanlar bir arşivde toplanır. Bu işlemlerin her biri basit görünür, ama sonuçları birbirinden farklıdır. Bir dosyanın kopyası mı oluştu, yoksa dosya mı yer değiştirdi? Silinen dosya geri getirilebilir mi? Bu kesim bu soruları adım adım yanıtlar.

2.2.7.1 Kopyalama ve taşıma

Tanım 2.2.16

Kopyalama, bir öğenin içeriği aynı olan ikinci, bağımsız bir örneğini hedef konumda oluşturur; özgün öğe kaynakta kalır. **Taşıma** ise öğenin konumunu değiştirir; öğe kaynaktan kalkar ve hedefe geçer. Kopyalama Kopyala + Yapıştır (Ctrl + C, Ctrl + V), taşıma Kes + Yapıştır (Ctrl + X, Ctrl + V) ile yapılır.

İşlem	Kaynakta ne olur?	Hedefte ne olur?
Kopyala + Yapıştır	özgün öğe kalır	yeni bir kopya oluşur
Kes + Yapıştır	öğe kaynak konumdan kaldırılır	öğe hedefe taşınır
Sürükle-bırak	aynı sürücüde taşınır, farklı sürücüde kopyalanır	işlem imlecin yanındaki ipucundan kontrol edilir

Tablo 2.2.13: Kopyalama ve taşımanın kaynakta ve hedefte sonuçları.

Kopyalamadan sonra iki ayrı dosya vardır. Biri değiştirildiğinde öteki etkilenmez. Taşımadan sonra ise dosya sayısı değişmez; yalnızca dosyanın konumu değişir. Kes komutu dosyayı hemen yerinden kaldırmaz. Kesilen dosyanın simgesi soluklaşır ve dosya taşınmak üzere işaretlenir. Asıl taşıma, Yapıştır komutu verildiğinde gerçekleşir. Yapıştırmadan başka bir şey kopyalanırsa kesme işlemi iptal olur ve dosya yerinde kalır.

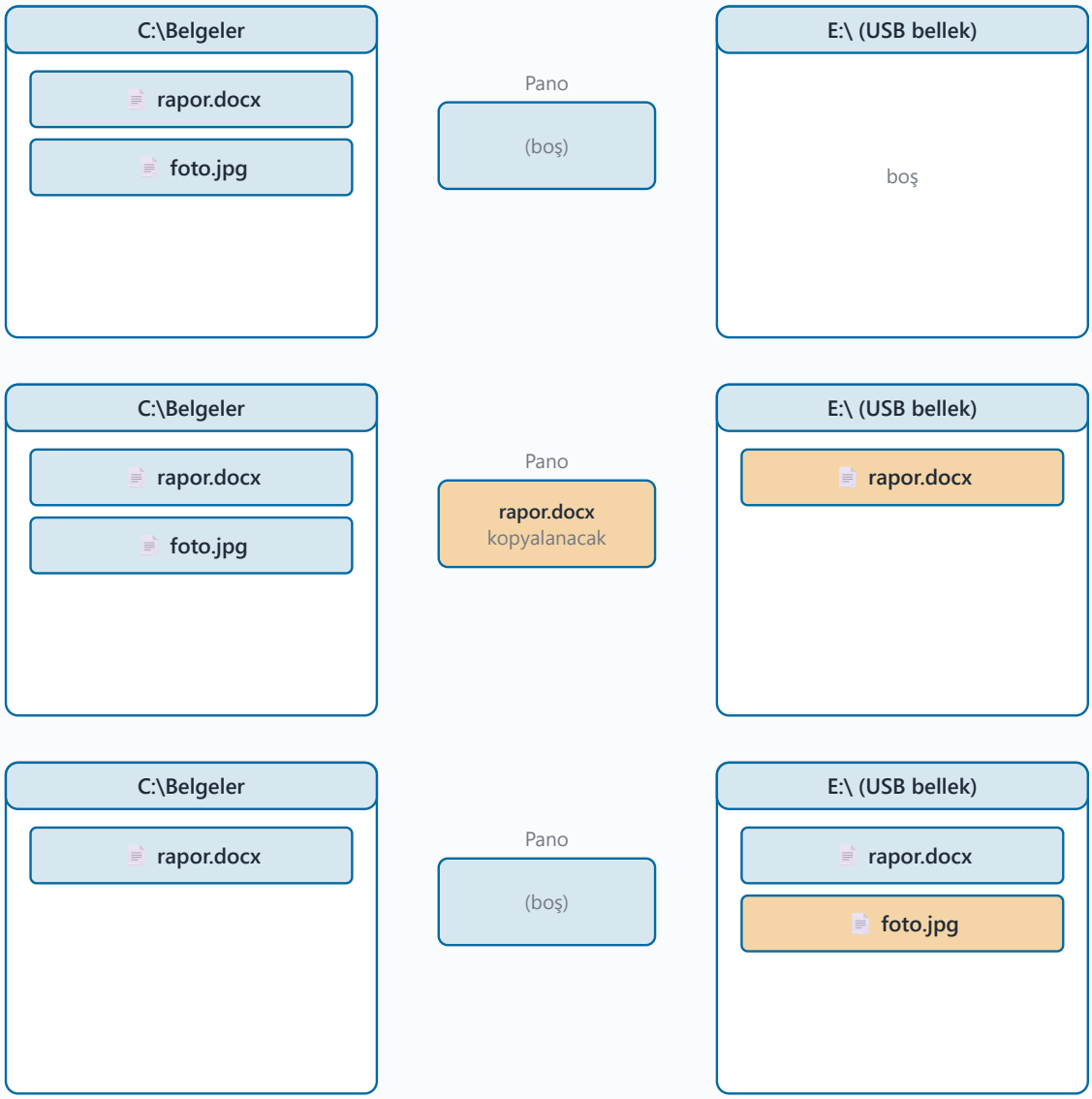
Örnek 2.2.12

`C:\Belgeler` klasöründe `rapor.docx` ve `foto.jpg` var; USB bellek (E:\) boş. Önce rapor USB'ye kopyalanıyor, sonra fotoğraf USB'ye taşınıyor. Tablo 2.2.14 her adımdan sonra iki konumu ve panoyu gösterir.

Adım	İşlem	C:\Belgeler	E:\	Pano	Toplam dosya
0	başlangıç	rapor.docx, foto.jpg	(boş)	(boş)	2
1	rapor.docx seçilir, Ctrl + C	rapor.docx, foto.jpg	(boş)	rapor.docx (kopyalanacak)	2
2	E:\ açılır, Ctrl + V	rapor.docx, foto.jpg	rapor.docx	rapor.docx (kopyalanacak)	3
3	foto.jpg seçilir, Ctrl + X	rapor.docx, foto.jpg (soluk)	rapor.docx	foto.jpg (taşınacak)	3
4	E:\'de Ctrl + V	rapor.docx	rapor.docx, foto.jpg	(boş)	3

Tablo 2.2.14: Kopyalama ve taşımanın adım adım izlenmesi.

Birinci adımda kaynakta hiçbir şey değişmez; pano yalnızca hangi dosyanın kopyalanacağını hatırlar. İkinci adımda toplam dosya sayısı 2'den 3'e çıkar, çünkü yeni bir kopya oluşmuştur. Üçüncü adımda `foto.jpg` henüz yerindedir, yalnızca soluk görünür. Dördüncü adımda fotoğraf hedefe geçer ve kaynaktan kalkar. Toplam dosya sayısı 3'te kalır: taşıma dosya sayısını değiştirmez. Kopyalamadan sonra pano içeriğini korur, aynı dosya başka yerlere de yapıştırılabilir. Kesilen öğe ise tek bir yapıştırmayla taşındığı için pano boşalır.



Şekil 2.2.17: Kopyalama ve taşımadan üç an. Üstte başlangıç durumu: C:\Belgeler'de iki dosya, USB bellek boş. Ortada rapor.docx kopyalanıp yapıştırılmıştır; iki konumda da rapor.docx vardır. Altta foto.jpg kesilip USB'ye yapıştırılmıştır; kaynaktan kalkmış, hedefe geçmiştir.

Kopyalamanın bir sonucu daha vardır. İki dosya bağımsız olduğu için, kaynakta yapılan değişiklik kopyaya yansımaz. Yukarıdaki örnekte `C:\Belgeler\rapor.docx` açılıp değiştirilir ve kaydedilirse, USB'deki kopya eski içeriğini korur. Bu durum yedek alırken işe yarar, ama aynı belgenin iki sürümüyle çalışırken karışıklık yaratır. Hangi kopyanın güncel olduğu bilinmezse yapılan iş kaybolabilir.

Sık yapılan hata

Önemli bir dosyayı yedeği olmadan taşımak ve hedefi kontrol etmemek. Taşıma sırasında USB bellek çekilirse ya da hedef yanlış seçilirse dosya beklenmedik bir yere gidebilir. Önemli dosyalarda taşımadan önce bir yedek bulundurmak ve işlemden sonra hedef klasörü açıp dosya-

nın orada olduğunu doğrulamak güvenlidir.

2.2.7.2 Sürükle-bırak

Bir dosyayı fareyle tutup başka bir klasörün üzerine bırakmak, Kopyala ya da Kes komutlarına gerek bırakmaz. Ancak sürükle-bırakın kopyalama mı taşıma mı yaptığı, kaynağın ve hedefin hangi sürücüde olduğuna bağlıdır.

Tanım 2.2.17

Sürükle-bırak (*drag and drop*), bir öğeyi fare düğmesi basılıyken başka bir konuma götürüp bırakma işlemidir. Varsayılan kural şudur: kaynak ve hedef *aynı sürücüdeyse* öğe taşınır, *farklı sürücülerdeyse* kopyalanır. Bırakırken Shift basılı tutulursa taşıma, Ctrl basılı tutulursa kopyalama zorlanır. Alt basılı tutulursa öğenin kendisi yerine bir kısayolu oluşturulur.

Bu kuralın bir mantığı vardır. Aynı sürücü içindeki taşıma çok hızlıdır, çünkü dosyanın baytları diskte yer değiştirmez; yalnızca dosya sistemindeki kayıt bir klasörden ötekine geçer. Farklı sürücüler arasında ise baytların hepsi okunup yazılmalıdır. Bu durumda Windows özgün dosyaya dokunmamayı, yani kopyalamayı güvenli seçenek olarak kabul eder.

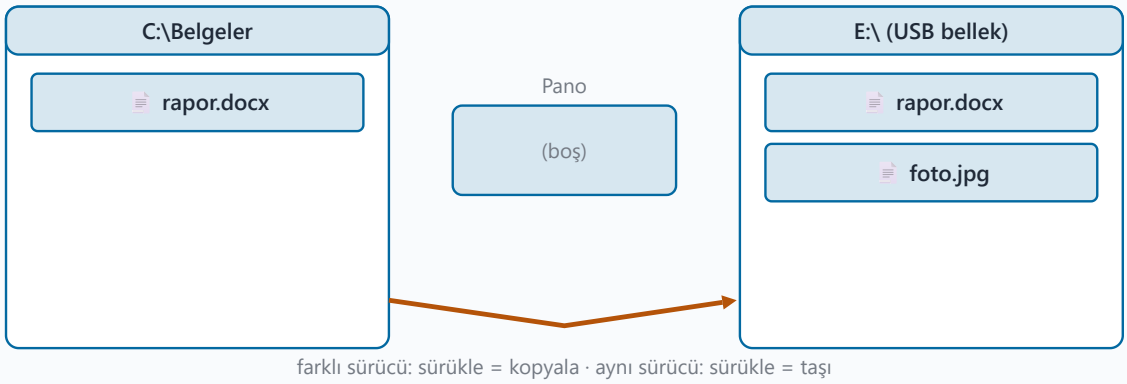
Örnek 2.2.13

Başlangıçta `C:\Belgeler`'de `foto.jpg`, `rapor.docx` ve `not.txt` var; `C:\Arsiv` ve E:\ boş. Dört sürükle-bırak sırayla yapılıyor:

Adım	İşlem	Kural	Sonuç
1	foto.jpg → C:\Arsiv	aynı sürücü	taşıma: foto.jpg Belgeler'den kalkar
2	rapor.docx → E:\	farklı sürücü	kopyalama: rapor.docx iki yerde
3	not.txt → E:\, Shift basılı	farklı sürücü, Shift	taşıma zorlanır: not.txt C:'den kalkar
4	E:\rapor.docx → C:\Arsiv, Ctrl basılı	farklı sürücü, Ctrl	kopyalama: rapor.docx üç yerde

Tablo 2.2.15: Sürükle-bırak kurallarının uygulanması.

Son durumda `C:\Belgeler`'de yalnızca `rapor.docx`, `C:\Arsiv`'de `foto.jpg` ve `rapor.docx`, E:\'de `rapor.docx` ve `not.txt` vardır. Birinci ve üçüncü adımlar taşıma, ikinci ve dördüncü adımlar kopyalamadır. Dördüncü adımda Ctrl basılmasa da sonuç değişmezdi, çünkü E:\ ile C:\ farklı sürücülerdir. Ctrl yalnızca niyeti açık hâle getirir.



Şekil 2.2.18: Sürükle-bırak kuralı: farklı sürücüye sürüklemek kopyalar, aynı sürücüde sürüklemek taşır.

Sık yapılan hata

Sürükle-bırakın her zaman kopyaladığını ya da her zaman taşıdığını sanmak. Sürükleme sırasında imlecin yanında küçük bir ipucu belirir: “Taşı: Arsiv” ya da “Kopyala: E:”. Fare bırakılmadan önce bu ipucuna bakmak, beklenmeyen bir taşımayı önler.

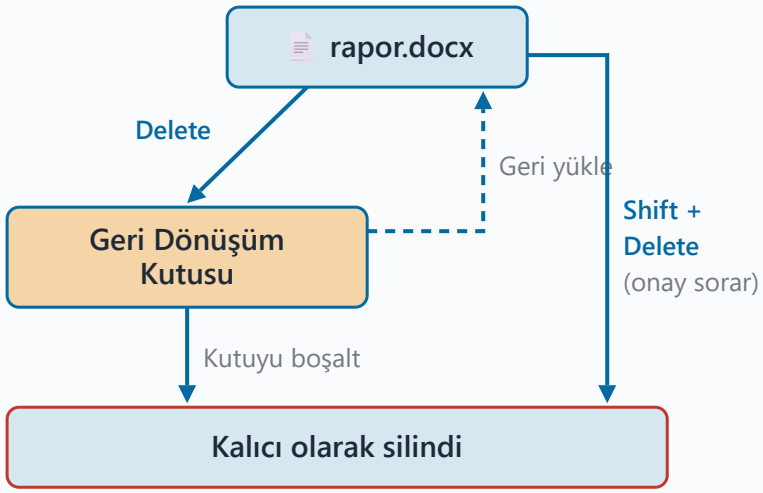
2.2.7.3 Yeniden adlandırma ve silme

Bir dosya ya da klasör seçilip F2'ye basılınca adı düzenlenebilir hâle gelir. Uzantılar görünürken Windows yalnızca adın uzantıdan önceki kısmını seçer; böylece yazılan yeni ad uzantıyı bozmaz. Yine de uzantıyı değiştirmemeye dikkat etmek gerekir (2.2.4.3). Birden çok dosya seçilip F2'ye basılırsa hepsi aynı adla ve sırayla numaralanarak adlandırılır: `tatil (1).jpg`, `tatil (2).jpg`, …

Tanım 2.2.18

Geri Dönüşüm Kutusu (*Recycle Bin*), Delete ile silinen dosyaların kalıcı olarak silinmeden önce bekletildiği özel klasördür. Kutudaki bir öğe *Geri yükle* komutuyla özgün konumuna döner. **Kalıcı silme** ise öğeyi kutuya göndermeden siler; Shift + Delete ile ya da kutu boşaltılarak yapılır.

Şekil 2.2.19 iki silme yolunu yan yana gösterir. Delete ile silinen dosya Geri Dönüşüm Kutusu'na gider. Oradan iki yol çıkar: Geri yükle dosyayı özgün klasörüne döndürür, Kutuyu boşalt ise kalıcı olarak siler. Shift + Delete ise kutuyu atlar: Windows bir onay sorar ve onaylanınca dosya doğrudan kalıcı olarak silinir.



Şekil 2.2.19: Delete dosyayı Geri Dönüşüm Kutusu'na taşır; oradan geri yüklenebilir ya da kutu boşaltılınca kalıcı silinir. Shift + Delete onay sorup dosyayı doğrudan kalıcı olarak siler.

Geri Dönüşüm Kutusu her durumda koruma sağlamaz. Üç önemli istisna vardır:

- **USB bellek ve ağ konumları.** Bu konumlarda silinen dosyalar çoğunlukla kutuya gitmez, doğrudan silinir. Windows bu durumda "kalıcı olarak silinsin mi?" diye sorar.
- **Açık ya da kilitli dosyalar.** Bir uygulamanın açık tuttuğu dosya silinemeyebilir. Windows "dosya başka bir programda açık" uyarısı verir; önce o uygulama kapatılır.
- **Bulut klasörleri.** OneDrive gibi eşitlenen bir klasörde silinen dosya, aynı hesaba bağlı öteki cihazlardan da silinir. Bulutun kendi geri dönüşüm kutusu bir süre için kurtarma imkânı verir.

Örnek 2.2.14

Masaüstünde (C: sürücüsü) `a.txt`, `b.txt` ve `c.txt` var. `a.txt` seçilip Delete'e, `b.txt` seçilip Shift + Delete'e basılıyor ve onay veriliyor. Sonra bir USB bellekteki `d.txt` seçilip Delete'e basılıyor.

1. `a.txt` Geri Dönüşüm Kutusu'na gider. Kutu açılıp Geri yükle seçilirse masaüstüne döner.
2. `b.txt` kutuya gitmez; kalıcı olarak silinir ve Windows'un kendi araçlarıyla geri getirilemez.
3. `c.txt` 'ye dokunulmamıştır, masaüstünde durur.
4. `d.txt` USB bellekte olduğu için Delete ile bile kutuya gitmez. Windows kalıcı silme onayı ister.

Sonuçta Geri Dönüşüm Kutusu'nda yalnızca `a.txt` vardır.

Sık yapılan hata

Geri Dönüşüm Kutusu'nu yedek yerine koymak. Kutu belirli bir doluluğa ulaşınca en eski öğeleri kendiliğinden siler. Depolama Algılaması da belirli bir süreden eski öğeleri temizleyebilir (2.2.10.1). Disk bozulursa kutudaki dosyalar da kaybolur. Kutu yalnızca yanlışlıkla silmeye karşı kısa süreli bir güvencedir.

2.2.7.4 Kısayol ve özgün dosya

Tanım 2.2.19

Kısayol (*shortcut*), başka bir dosyanın, klasörün ya da programın yolunu tutan küçük bir dosyadır. Uzantısı `.lnk` 'dir ve boyutu genellikle 1–2 KB'tır. Kısayola çift tıklamak, tuttuğu yoldaki özgün öğeyi açar. Kısayolun simgesinin sol alt köşesinde küçük bir ok bulunur.

Kısayol, sık açılan bir dosyaya ulaşmayı kolaylaştırır. Derinlerdeki bir klasörde duran bir dosyanın kısayolu masaüstüne konur; dosya yerinden hiç kımıldamaz. Kısayol, sağ tık → *Kısayol oluştur* komutuyla ya da dosya Alt tuşu basılıyken sürüklenerek oluşturulur.

Kısayolun en önemli özelliği, içinde dosyanın kendisinin değil yalnızca yolunun bulunmasıdır. Bu özellik üç sonuç doğurur (Tablo 2.2.16).

İşlem	Kısayola etkisi	Özgün dosyaya etkisi
kısayol silinir	kısayol gider	yok; dosya yerinde durur
özgün dosya başka klasöre taşınır	eski yolu gösterir, açılamayabilir	dosya yeni yerindedir
kısayol USB belleğe kopyalanıp başka bilgisayarda açılır	“hedef bulunamadı” hatası	dosya ilk bilgisayarda kalır

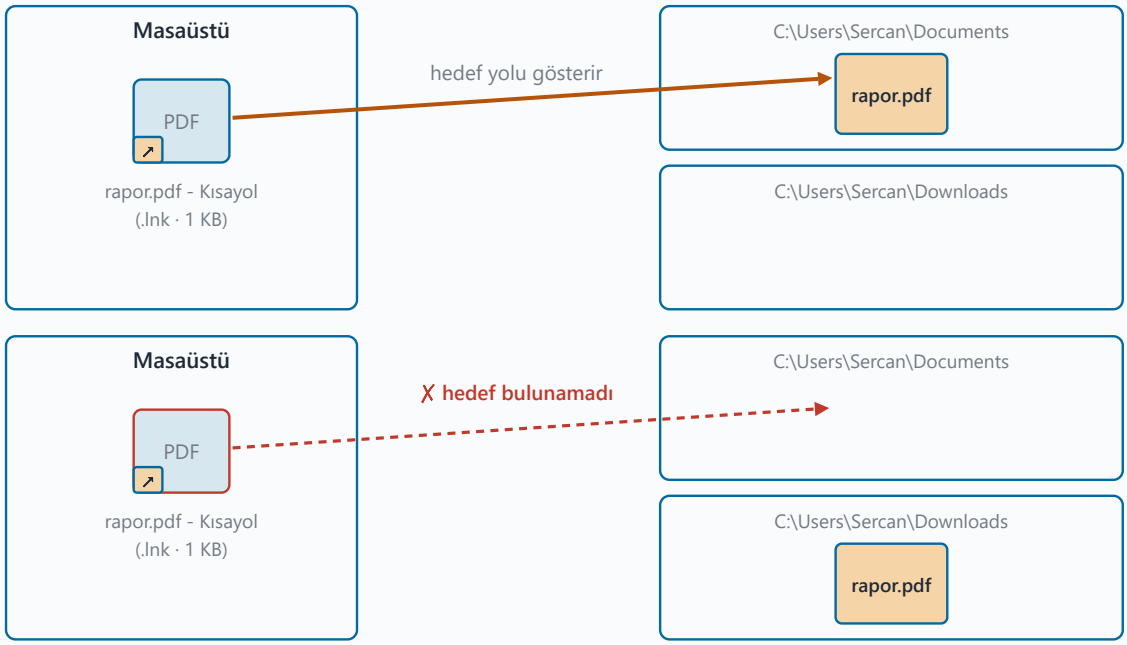
Tablo 2.2.16: Kısayol ve özgün dosya üzerindeki işlemlerin sonuçları.

Örnek 2.2.15

Masaüstünde “rapor.pdf - Kısayol” var ve `C:\Users\Sercan\Documents\rapor.pdf` dosyasını gösteriyor. Üç durum sırayla inceleniyor.

1. **Kısayol silinir.** Masaüstündeki simge kaybolur. `Documents` klasöründeki `rapor.pdf` yerinde durur. Kısayolu silmek dosyayı silmez.
2. **Yeni kısayol oluşturulur, sonra özgün dosya Downloads'a taşınır.** Kısayol hâlâ `…\Documents\rapor.pdf` yolunu tutar. Bu yolda dosya kalmadığı için kısayol açılamayabilir. Windows bazen dosyanın yeni yerini bulup kısayolu onarmayı önerir, ama buna güvenilmez.

3. **Kısayol USB'ye kopyalanır ve başka bir bilgisayarda çift tıklanır.** USB'ye giden şey 1 KB'lık `.lnk` dosyasıdır; PDF'in kendisi değildir. Öteki bilgisayarda `C:\Users\Sercan\Documents\rapor.pdf` yolu yoktur ve "hedef bulunamadı" hatası çıkar.



Şekil 2.2.20: Kısayol ve özgün dosya. Üstte masaüstündeki 1 KB'lık rapor.pdf - Kısayol dosyası C:\Users\Sercan\Documents\rapor.pdf'i gösterir. Altta özgün dosya Downloads'a taşınmıştır; kısayol eski yolu gösterdiği için hedef bulunamaz.

Sık yapılan hata

Masaüstündeki kısayolları USB belleğe kopyalayıp yedek aldığını sanmak. Bilgisayar bozulursa USB'de yalnızca hiçbir yeri göstermeyen küçük `.lnk` dosyaları kalır. Yedek alırken dosyaların kendileri kopyalanır. Bir dosyanın kısayol olup olmadığı simgedeki oktan, Tür sütunundaki "Kısayol" yazısından ya da boyutunun 1–2 KB olmasından anlaşılır.

2.2.7.5 ZIP arşivi

Bir ödevin birkaç dosyasını e-postayla göndermek ya da bir sisteme tek dosya olarak yüklemek gerektiğinde dosyalar bir arşivde toplanır.

Tanım 2.2.20

Arşiv (*archive*), birden çok dosyayı ve klasörü tek bir dosyada toplayan dosyadır. **Sıkıştırma** (*compression*), verideki tekrarları daha kısa biçimde yazarak dosyanın boyutunu küçültme işlemidir. ZIP, hem arşivleme hem kayıpsız sıkıştırma yapan yaygın bir biçimdir; Windows ZIP dosyalarını ek program gerektirmeden oluşturur ve açar. Arşivdeki dosyaların dışarı çıkarılmasına **ayıklama** denir.

Kayıpsız sıkıştırma, ayıklanan dosyanın baytlarının özgün dosyayla birebir aynı olmasını sağlar. Sıkıştırmanın ne kadar küçülteceği ise içeriğe bağlıdır. Düz metinde aynı sözcükler ve harf dizileri çok tekrar eder; bu yüzden metin iyi sıkışır. JPEG fotoğraf ve DOCX belgesi ise zaten sıkıştırılmış biçimlerdir. Bir DOCX dosyası aslında içinde XML dosyaları bulunan bir ZIP arşividir. Sıkıştırılmış veride tekrar azdır; ikinci kez sıkıştırmak neredeyse hiçbir şey kazandırmaz.

Örnek 2.2.16

`Belgeler\Dersler`'deki üç dosya bir ZIP arşivinde toplanıyor: `rapor.docx` (480 KB), `notlar.txt` (120 KB), `foto.jpg` (2.410 KB). Adımlar şöyledir:

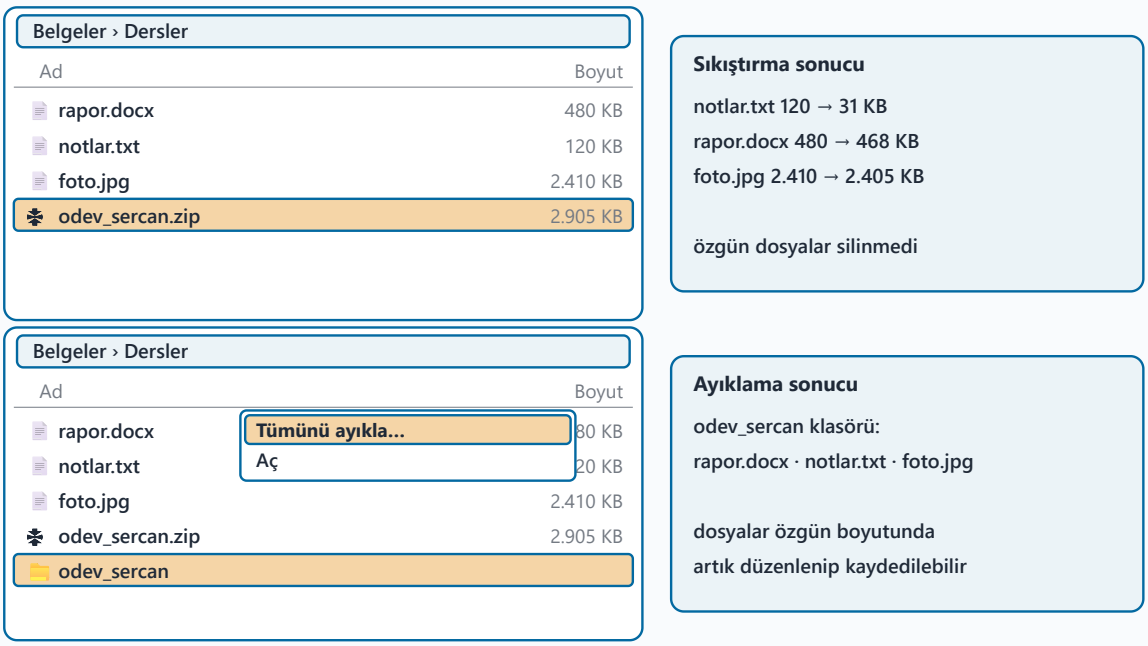
1. **Seçim.** Ctrl basılıyken üç dosyaya tek tek tıklanır. Ctrl + tık seçime tek bir öğe ekler ya da çıkarır; Shift + tık ise iki tıklama arasındaki bütün satırları seçer. Durum çubuğunda "3 öğe seçili" yazar. Seçimin toplam boyutu 480 + 120 + 2.410 = 3.010 KB'tır.
2. **Sıkıştırma.** Seçimin üzerinde sağ tıklanıp *Sıkıştır → ZIP dosyası* seçilir. Windows 11'in eski sürümlerinde bu komutun adı *ZIP dosyasına sıkıştır*'dır; Microsoft'un Türkçe yardım sayfası ise "Daha fazla seçenek" → "Sıkıştırılmış klasöre gönder" yolunu verir. Arşiv aynı klasörde oluşur ve adı seçili gelir; `odev_sercan` yazılıp Enter'a basılır. Özgün dosyalar silinmez.
3. **Sonuç.** `odev_sercan.zip` 2.905 KB'tır. Tablo 2.2.17 her dosyanın ne kadar küçüldüğünü gösterir.
4. **Ayıklama.** Arşive sağ tıklanıp *Tümünü ayıkla…* → *Ayıkla* seçilir. Arşivle aynı adlı `odev_sercan` klasörü oluşur ve üç dosya özgün boyutlarıyla içine çıkar.

Dosya	Özgün	Sıkıştırılmış	Kalan oran	Kazanç
notlar.txt	120 KB	31 KB	%25,8	%74,2
rapor.docx	480 KB	468 KB	%97,5	%2,5
foto.jpg	2.410 KB	2.405 KB	%99,8	%0,2
Toplam	3.010 KB	2.904 KB (+ arşiv bilgisi)	ZIP 2.905 KB; kazanç ≈ %3,5	

Tablo 2.2.17: Dosyaların sıkıştırma sonuçları.

Kalan oran sıkıştırılmış boyutun özgün boyuta bölümüdür: notlar için 31 ÷ 120 ≈ 0,258, yani %25,8. Metin dosyası dörtte birine inmiştir. DOCX ve JPEG ise neredeyse hiç küçülmemiştir. Sıkıştırılmış boyutların toplamı 31 + 468 + 2.405 = 2.904 KB'tır. ZIP dosyası, içindeki dosyala-

rın adlarını ve özelliklerini de tuttuğu için bundan biraz büyüktür. Bütün arşivin kazancı (3.010 − 2.905) ÷ 3.010 ≈ %3,5'tir. Bu örnekte arşivin asıl yararı küçültmek değil, üç dosyayı tek pakette toplamaktır.



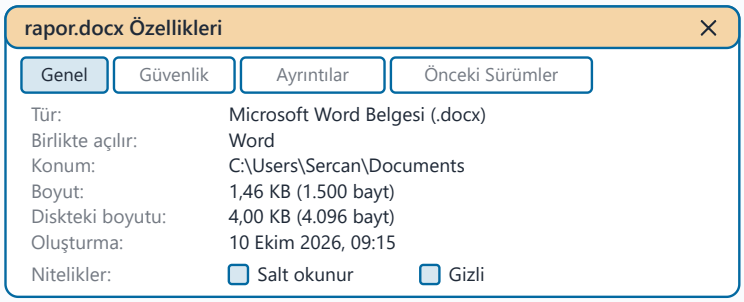
Şekil 2.2.21: ZIP arşivi. Üstte üç dosya odev_sercan.zip (2.905 KB) arşivinde toplanmıştır; notlar.txt 120 KB'tan 31 KB'a inmiş, rapor.docx ve foto.jpg neredeyse küçülmemiştir. Altta Tümünü ayıkla ile odev_sercan klasörü oluşmuş, dosyalar özgün boyutlarıyla çıkarılmıştır.

Sık yapılan hata

ZIP'in içindeki bir belgeyi çift tıklayıp düzenlemek. ZIP'e çift tıklamak arşivin içini bir klasör gibi gösterir, ama buradan açılan belge geçici bir klasördeki kopyadır. Word'de yapılan değişiklik kaydedilse bile geçici kopyaya yazılır; ZIP içindeki dosya değişmez ve emek kolayca kaybolur. Düzenleme her zaman önce ayıklanan kopya üzerinde yapılır.

2.2.7.6 Dosya ve klasör özellikleri

Bir dosyaya sağ tıklanıp *Özellikler* seçilince (ya da dosya seçiliyken Alt + Enter'a basılınca) dosyanın ayrıntılarını gösteren pencere açılır (Şekil 2.2.22).



Şekil 2.2.22: rapor.docx dosyasının Özellikler penceresinin Genel sekmesi: tür, açılacak uygulama, konum, boyut, diskteki boyut, oluşturma tarihi ve Salt okunur / Gizli nitelikleri. Üstte Güvenlik, Ayrıntılar ve Önceki Sürümler sekmeleri vardır.

Genel sekmesinde şu bilgiler bulunur:

- **Tür ve uygulama**: dosyanın biçimi (Microsoft Word Belgesi, .docx) ve çift tıklanınca açılacak uygulama. Yanındaki *Değiştir* düğmesi bu uygulamayı değiştirir.
- **Konum**: dosyanın bulunduğu klasörün tam yolu.
- **Boyut** ve **diskteki boyutu**: içeriğin büyüklüğü ve diskte ayrılan alan. Bu iki sayının neden farklı olduğu aşağıdaki örnekte açıklanır.
- **Tarihler**: oluşturma, değiştirme ve erişim tarihleri.
- **Nitelikler** (*attributes*): *Salt okunur* dosyanın değiştirilmesini engeller; *Gizli* dosyayı listeden saklar (2.2.6.2).

Güvenlik sekmesi dosyanın sahibini ve hangi kullanıcının dosyayı okuyabileceğini ya da değiştirebileceğini gösterir. Önceki Sürümler sekmesi, Dosya Geçmişi ya da geri yükleme noktası açıksa dosyanın eski kopyalarını listeler (2.2.10.6).

Tanım 2.2.21

Dosya sistemi diski sabit boyutlu bloklara böler. NTFS'te bu blokların boyutu genellikle 4 KB'tır (4.096 bayt). Birkaç yüz bayttan büyük her dosya en az bir blok kaplar ve tam sayıda blok kullanır. (Çok küçük dosyalar NTFS'in kendi kayıt tablosunda tutulabilir; onların diskteki boyutu 0 görünür.) **Diskteki boyut** bu yüzden şöyle hesaplanır:

> diskteki boyut = ⌈boyut ÷ 4.096⌉ × 4.096 bayt

Buradaki ⌈ ⌉ işareti yukarı yuvarlamayı gösterir.

Örnek 2.2.17

Şekil 2.2.22'deki dosyanın boyutu 1.500 bayttır. Windows bunu neden "1,46 KB", diskteki boyutunu da neden "4,00 KB" gösterir?

1. Windows KB'ı 1.024 bayt olarak hesaplar: 1.500 ÷ 1.024 ≈ 1,46. Boyut 1,46 KB görünür.
2. Dosya kaç blok kaplar? 1.500 ÷ 4.096 ≈ 0,37; yukarı yuvarlanınca 1 blok.
3. Diskteki boyut 1 × 4.096 = 4.096 bayt = 4,00 KB'tır. Bloğun kalan 2.596 baytı boştur, ama başka bir dosya kullanamaz.

Aynı kural daha büyük dosyalarda da geçerlidir. 10.000 baytlık bir dosya 10.000 ÷ 4.096 ≈ 2,44 → 3 blok kaplar; diskteki boyutu 3 × 4.096 = 12.288 bayttır. Çok sayıda küçük dosyada fark büyür: her biri 1.500 bayt olan 100 dosyanın toplam boyutu 150.000 bayttır, ama diskte 100 × 4.096 = 409.600 bayt (400 KB) yer kaplar.

Sık yapılan hata

Salt okunur niteliğin dosyayı silinmeye karşı da koruduğunu sanmak. Salt okunur, dosyanın içeriğinin değiştirilip aynı adla kaydedilmesini engeller. Dosya yine de kopyalanabilir, farklı adla kaydedilebilir ve silinebilir. Nitelikler bir hatırlatma katmanıdır, güvenlik önlemi değildir.

2.2.8 Ayarlar

Windows'un davranışını belirleyen seçeneklerin çoğu tek bir uygulamada toplanır: Ayarlar. Ekranın çözünürlüğü, bağlanılan Wi-Fi ağı, yüklü uygulamalar, kullanıcı hesapları, klavye düzeni ve güncellemeler buradan yönetilir.

2.2.8.1 Ayarlar uygulaması

Ayarlar, Win + I ile ya da Başlat menüsündeki dişli simgesiyle açılır. Sol sütunda kategoriler, sağda seçilen kategorinin sayfaları bulunur. Kategori adları Windows sürümüne göre biraz değişebilir, ama temel alanlar aynı kalır. Tablo 2.2.18 bu notta en çok kullanılan kategorileri listeler.

Kategori	İçerdiği ayarlar
Sistem	ekran, ses, bildirimler, güç, depolama, kurtarma, hakkında
Bluetooth ve cihazlar	fare, klavye, yazıcı ve bağlı cihazlar
Ağ ve internet	Wi-Fi, Ethernet, VPN ve veri kullanımı
Kişiselleştirme	arka plan, tema, Başlat ve görev çubuğu
Uygulamalar	yüklü uygulamalar, varsayılan uygulamalar, başlangıç
Hesaplar	oturum açma seçenekleri, Windows yedekleme, diğer kullanıcılar

Kategori	İçerdiği ayarlar
Zaman ve dil	tarih ve saat, dil ve bölge, klavye
Erişilebilirlik	metin boyutu, büyüteç, renk filtreleri, ekran okuyucu, altyazı
Gizlilik ve güvenlik	uygulama izinleri ve Windows Güvenliği
Windows Update	güncellemeler, yeniden başlatma ve güncelleme geçmişi

Tablo 2.2.18: Ayarlar uygulamasının başlıca kategorileri.

Aranan ayarın hangi kategoride olduğu bilinmiyorsa, Ayarlar penceresinin sol üstündeki arama kutusu kullanılır. Bu kutu, Başlat menüsündeki aramayla aynı biçimde çalışır, ama yalnızca ayarlar arasında arar. Notta bir ayarın yeri "Ayarlar → Sistem → Ekran" biçiminde yazılır. Bu yazım sırasıyla Ayarlar uygulamasını, Sistem kategorisini ve Ekran sayfasını açmak anlamına gelir.

2.2.8.2 Wi-Fi, yazıcı ve ekran

Yeni bir bilgisayarda ya da yeni bir ortamda ilk yapılan üç ayar genellikle ağa bağlanmak, yazıcı eklemek ve ekranı düzenlemektir.

Wi-Fi'ye bağlanma. Görev çubuğunun sağındaki ağ simgesine tıklanır ya da Win + A'ya basılır; hızlı ayarlar paneli açılır. Wi-Fi düğmesinin yanındaki ok yakındaki ağları listeler. Ağ seçilir, *Bağlan*'a basılır ve ağın parolası girilir. "Otomatik olarak bağlan" işaretlenirse bilgisayar bu ağı bir dahaki sefere kendisi tanır.

Yazıcı ekleme. Ayarlar → Bluetooth ve cihazlar → Yazıcılar ve tarayıcılar → *Cihaz ekle* yolu izlenir. Aynı ağdaki ya da USB ile bağlı bir yazıcı genellikle otomatik bulunur ve sürücüsü kendiliğinden yüklenir. Yazıcının sürücüsü bulunamazsa üreticinin web sitesinden indirilir.

Ekran çözünürlüğü ve ölçek. Ayarlar → Sistem → Ekran sayfasında iki ayrı ayar bulunur. *Ekran çözünürlüğü* görüntünün kaç pikselden oluşacağını belirler. Bu değer ekranın yerel çözünürlüğünde, yani Windows'un "(Önerilen)" diye işaretlediği değerde kalmalıdır. Daha düşük bir çözünürlük seçilirse görüntü bulanıklaşır. *Ölçek* ise yazıların ve simgelerin büyüklüğünü ayarlar. Yazılar küçük geliyorsa çözünürlük düşürülmez, ölçek artırılır.

Tanım 2.2.22

Ölçek (*scale*), Windows'un yazıları, simgeleri ve pencereleri kaç kat büyük çizeceğini belirleyen yüzdedir. %100 ölçekte bir arayüz öğesi 1 piksele 1 piksel çizilir; %150 ölçekte aynı öğe 1,5 kat büyük çizilir. Uygulamaların kullanabildiği *mantıksal* alan, fiziksel çözünürlüğün ölçeğe bölümü kadardır.

Örnek 2.2.18

15,6 inçlik bir dizüstü bilgisayarın ekranı 1920 × 1080 pikseldir. Ölçek %100, %125 ve %150 seçildiğinde uygulamalara kalan mantıksal alan nedir?

1. %100: 1920 ÷ 1 = 1920, 1080 ÷ 1 = 1080. Mantıksal alan 1920 × 1080'dir; yazılar en küçük hâlindedir.
2. %125: 1920 ÷ 1,25 = 1536, 1080 ÷ 1,25 = 864. Mantıksal alan 1536 × 864'tür.
3. %150: 1920 ÷ 1,5 = 1280, 1080 ÷ 1,5 = 720. Mantıksal alan 1280 × 720'dir; yazılar büyük, ekrana sığan içerik azdır.

Her üç durumda da ekranın 1920 × 1080 fiziksel pikselinin hepsi kullanılır; görüntü keskin kalır. Çözünürlüğü 1280 × 720'ye düşürmek ise yazıları aynı oranda büyütür, ama her mantıksal piksel 1,5 fiziksel piksele yayılacağı için görüntü bulanıklaşır. Yazıları büyütmenin doğru yolu ölçektir.

Sık yapılan hata

Herkese açık bir Wi-Fi ağında (kafe, havalimanı) bankacılık ya da okul hesabı gibi önemli hesaplara giriş yapmak. Bu ağlardaki trafiği başkaları izleyebilir ya da sahte bir erişim noktası gerçek ağın adını taklit edebilir. Önemli işlemler güvenilen bir ağda ya da mobil veriyle yapılır.

2.2.8.3 Kullanıcı hesapları

Bir bilgisayarı birden çok kişi kullanabilir. Windows her kişiye ayrı bir **kullanıcı hesabı** (*user account*) açar. Her hesabın kendi masaüstü, Belgeler klasörü, ayarları ve parolası vardır. Ayrı hesaplar, kişisel dosyaları ve ayarları kullanıcılar arasında ayırır. Bir kullanıcının `C:\Users\Ayse` klasörüne öteki standart kullanıcılar erişemez.

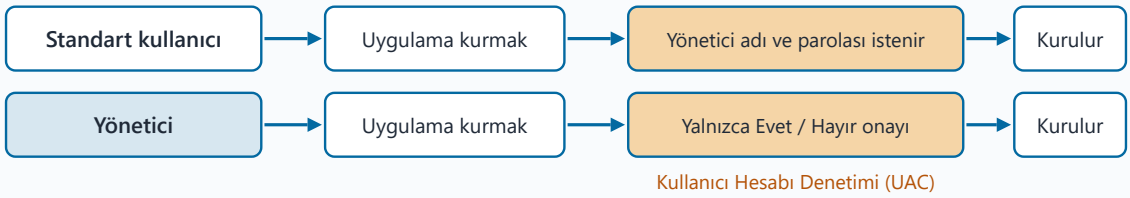
Hesaplar iki ayrı soruya göre sınıflanır. Birinci soru, kimlik bilgilerinin nerede yönetildiğidir:

- **Yerel hesap** (*local account*): kullanıcı adı ve parola yalnızca o bilgisayarda tutulur. Başka bir cihaza bağlanmaz.
- **Microsoft hesabı**: bir e-posta adresiyle açılan çevrim içi hesaptır. Desteklenen ayarları, OneDrive dosyalarını ve Microsoft Store uygulamalarını cihazlar arasında bağlayabilir; parola unutulursa çevrim içi kurtarılabilir.

İkinci soru, hesabın yetkisidir:

- **Standart kullanıcı** (*standard user*): günlük işleri yapar, kendi dosyalarını ve ayarlarını değiştirir. Program kurma ve sistem genelindeki ayarları değiştirme gibi işler için yönetici izni gerekir.
- **Yönetici** (*administrator*): uygulama kurar, kaldırır, öteki hesapları yönetir ve bütün sistem ayarlarını değiştirebilir.

Yönetici izni gerektiren bir iş başlatıldığında Windows ekranı karartır ve bir onay penceresi gösterir. Bu özelliğe **Kullanıcı Hesabı Denetimi** (*User Account Control*, UAC) denir. Şekil 2.2.23 iki hesap türünde UAC'nin ne istediğini karşılaştırır. Standart kullanıcı bir uygulama kurmak isterse, bir yöneticinin adı ve parolası istenir. Yönetici aynı işi yaparsa yalnızca Evet / Hayır onayı istenir.



Şekil 2.2.23: Kullanıcı Hesabı Denetimi. Standart kullanıcı uygulama kurmak istediğinde yönetici adı ve parolası istenir; yönetici aynı işte yalnızca Evet / Hayır onayı verir. İki durumda da onaydan sonra uygulama kurulur.

Yönetici hesabında bile onay istenmesinin bir nedeni vardır. Kötü amaçlı bir program arka planda kendini kurmaya çalışırsa UAC penceresi belirir. Kullanıcı o anda bir şey kurmuyorsa, bu pencere bir uyarı işaretidir ve Hayır seçilir. Bu yüzden günlük işlerde standart hesap kullanmak, yönetici yetkisini yalnızca gerektiğinde açmak güvenli bir ilkedir.

Oturum kapatma, açık uygulamaları kapatıp kullanıcının oturumunu sonlandırır. Başka bir kullanıcı ise aynı bilgisayarda kendi hesabıyla oturum açar. Yeni bir hesap Ayarlar → Hesaplar → Diğer kullanıcılar sayfasından eklenir; hesabın türü de aynı sayfada *Hesap türünü değiştir* ile seçilir.

2.2.8.4 Uygulama yönetimi ve izinler

Bir uygulamanın bilgisayara girişinden çıkışına kadar birkaç karar verilir: nereden kurulacağı, hangi izinlerin verileceği, oturum açılınca çalışıp çalışmayacağı, sorun çıkarınca ne yapılacağı ve sonunda nasıl kaldırılacağı.

- **Kurulum kaynağı.** Uygulamalar Microsoft Store'dan ya da üreticinin resmî sitesinden edinilir. Tanımadık sitelerdeki “ücretsiz” sürümler sık sık kötü amaçlı yazılım taşır.
- **Kaldırma.** Ayarlar → Uygulamalar → Yüklü uygulamalar sayfasında uygulamanın yanındaki üç noktaya tıklanıp *Kaldır* seçilir. Doğru yol budur; bir uygulamanın kısayolunu ya da klasörünü silmek onu kaldırmaz.
- **Başlangıç uygulamaları.** Bazı uygulamalar oturum açılınca kendiliğinden çalışır. Bu liste Ayarlar → Uygulamalar → Başlangıç sayfasından ya da Görev Yöneticisi'nden sınırlandırılır (2.2.10.4).
- **İzinler.** Kamera, mikrofon, konum ve dosya erişimi uygulama bazında denetlenir.
- **Onarım ve sıfırlama.** Desteklenen uygulamalarda, Yüklü uygulamalar → Gelişmiş seçenekler sayfasında *Onar* ve *Sıfırla* düğmeleri bulunur. Onar verilere dokunmadan uygulamayı düzeltmeye çalışır; Sıfırla uygulamanın verilerini siler ve onu ilk kurulduğu hâle döndürür.
- **Güncelleme.** Uygulamalar güvenlik ve uyumluluk için güncel tutulur. Store uygulamaları kendiliğinden güncellenir; öteki uygulamalar genellikle kendi güncelleme bildirimlerini gösterir.

İzinler, bir uygulamanın kişisel verilere ve aygıtlara ne kadar erişebileceğini belirler. Şekil 2.2.24 kamera izni sayfasını gösterir. Kamera uygulamasının ve Teams'in kamerayı kullanmasına izin verilmiş, bir oyunun ise kapatılmıştır. Bir oyunun kameraya ihtiyacı yoktur. Bir uygulamaya yalnızca işini yapmak için gereken izinleri vermek, verilerin gereksiz yere toplanmasını önler.



Şekil 2.2.24: Ayarlar → Gizlilik ve güvenlik → Kamera sayfası: Kamera uygulaması ve Microsoft Teams için izin açık, bir oyun için kapalı.

Sık yapılan hata

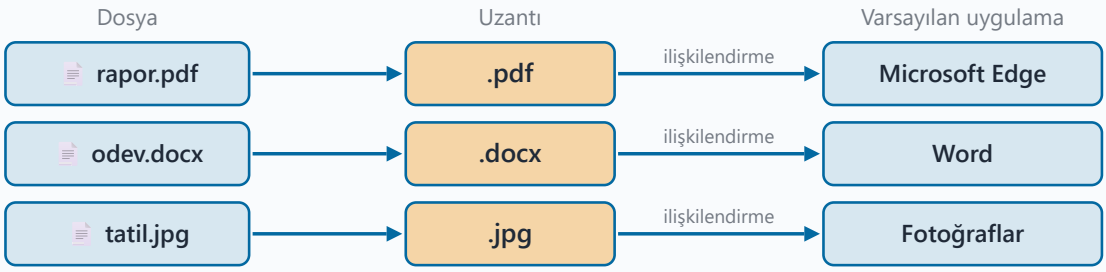
Bir uygulamayı kaldırmak için Program Files'taki klasörünü silmek. Klasör silinse bile uygulamanın kayıtları, kısayolları ve arka plan hizmetleri sistemde kalır; Windows uygulamayı hâlâ yüklü sanır. Uygulamalar Ayarlar'daki Yüklü uygulamalar sayfasından kaldırılır.

2.2.8.5 Varsayılan uygulamalar

Tanım 2.2.23

Varsayılan uygulama (*default app*), belirli bir dosya türü ya da bağlantı türü açıldığında Windows'un kendiliğinden seçtiği programdır. Örneğin `.pdf` dosyalarının varsayılan uygulaması Microsoft Edge olabilir.

Varsayılan uygulama, 2.2.4.3'te anlatılan ilişkilendirmenin kullanıcıya görünen yüzüdür. Şekil 2.2.25 üç dosya için bu eşlemeyi gösterir. Dosyaya çift tıklandığında Windows önce uzantıya bakar, sonra o uzantıyla ilişkilendirilmiş uygulamayı başlatır.



Şekil 2.2.25: Uzantıya göre varsayılan uygulama eşlemesi: rapor.pdf (.pdf) → Microsoft Edge, odev.docx (.docx) → Word, tatil.jpg (.jpg) → Fotoğraflar.

Varsayılanı değiştirmenin iki yolu vardır. Kalıcı değişiklik için Ayarlar → Uygulamalar → Varsayılan uygulamalar sayfasında dosya türü (örneğin `.pdf`) aranır ve yeni uygulama seçilir. Tek seferlik değişiklik için ise dosyaya sağ tıklanıp *Birlikte aç* seçilir ve listeden başka bir uygulama seçilir. Birlikte aç penceresinde "Her zaman" seçilirse bu seçim de kalıcı olur.

Örnek 2.2.19

`.pdf` dosyalarının varsayılan uygulaması Edge'dir. Aşağıdaki işlemlerde `odev.pdf` hangi uygulamada açılır?

Adım	İşlem	Açılan uygulama	Varsayılan
1	odev.pdf'e çift tıklanır	Edge	Edge
2	sağ tık → Birlikte aç → Acrobat Reader ("Her zaman" seçilmez)	Acrobat Reader	Edge
3	yeniden çift tıklanır	Edge	Edge
4	Ayarlar → Uygulamalar → Varsayılan uygulamalar'da .pdf için Acrobat seçilir, sonra çift tıklanır	Acrobat Reader	Acrobat Reader

İkinci adımdaki seçim tek seferliktir ve varsayılanı değiştirmez. Bu yüzden üçüncü adımda dosya yine Edge'de açılır. Dördüncü adımda varsayılan dosya türüne göre değiştirildiği için bundan sonraki bütün çift tıklamalar Acrobat Reader'ı açar.

Sık yapılan hata

Tanınmayan bir dosya türünü "bir programla açılır belki" diye çalıştırmayı denemek. Uzantısı bilinmeyen bir dosya, özellikle e-posta ekinden geldiyse, açılmadan önce kaynağı doğrulanır. Windows "Bu dosyayı nasıl açmak istiyorsunuz?" diye sorduğunda rastgele bir uygulama seçmek yerine dosyanın ne olduğu öğrenilir.

2.2.8.6 Erişilebilirlik

Bilgisayarı kullanan herkes aynı biçimde görmez, duymaz ya da fare kullanamaz. Erişilebilirlik ayarları, Windows'u farklı gereksinimleri olan kullanıcılara uyarlar. Bu ayarlar yalnızca engelli kullanıcılar için değildir. Uzun süre ekran başında çalışan, gürültülü bir ortamda video izleyen ya da bir elini kullanamayan herkes yararlanabilir. Ayarlar Ayarlar → Erişilebilirlik kategorisinde toplanır (Tablo 2.2.19).

Özellik	Ne yapar?	Kimin işine yarar?
Ekran okuyucu (Ekran Okuyucusu)	ekrandaki metni ve denetimleri sesli okur; Win + Ctrl + Enter ile açılıp kapanır	görme engelli kullanıcılar
Büyüteç	ekranın bir bölümünü ya da tamamını büyütür; Win + artı (+) ile açılır	az gören kullanıcılar, küçük ayrıntıyı gösteren eğitmenler
Kontrast ve renk	karşıtlık temaları metni belirginleştirir; renk filtreleri renk ayrımını iyileştirir	az gören ve renk körü kullanıcılar
Canlı açıklamalı altyazı	bilgisayarda çalan ya da mikrofona gelen konuşmayı ekranda metin olarak gösterir	işitme engelli kullanıcılar, sessiz ortamda video izleyenler
Sesli erişim	desteklenen işlemleri sesli komutlarla yönetir	fare ve klavye kullanamayan kullanıcılar
Klavye ve fare	yapışkan tuşlar, filtre tuşları, büyük ve renkli fare işaretçisi	hareket kısıtlılığı olan kullanıcılar

Tablo 2.2.19: Başlıca erişilebilirlik özellikleri.

Yapışkan tuşlar özelliği iyi bir örnektir. Ctrl + Shift + Esc gibi üç tuşlu bir kısayol, tek elle ya da tek parmakla basılamaz. Yapışkan tuşlar açıkken Ctrl, Shift ve Alt gibi değiştirici tuşlar basılı kalmış sayılır. Kullanıcı tuşlara sırayla, birer birer basar; Windows bunları birlikte basılmış gibi yorumlar. Filtre tuşları ise kısa ya da yinelenen istemsiz tuş basışlarını yok sayar. Titreyen bir elin aynı harfi üç kez yazmasını önler.

2.2.8.7 Türkçe klavye ve dil

Klavyenin üzerindeki harfler ile bilgisayarın tuşlara verdiği anlam ayrı şeylerdir. Hangi tuşun hangi harfi üreteceği **klavye düzeni** (*keyboard layout*) ayarıyla belirlenir. Türkiye'de iki düzen kullanılır. **Türkçe Q**, İngilizce klavyenin harf yerleşimine Türkçe harfleri ekleyen ve en yaygın kullanılan düzendir. **Türkçe F** ise Türkçedeki harf sıklıklarına göre tasarlanmış ve hızlı yazım için geliştirilmiş alternatif düzendir. F klavyede en sık kullanılan harfler parmakların durduğu orta sıraya yerleştirilmiştir.

Örnek 2.2.20

Bir bilgisayarda yalnızca Türkçe Q klavye var. Türkçe F eklenip kullanılmak isteniyor. Adımlar şöyledir:

1. Win + I ile Ayarlar açılır. Dil ve klavye ayarları *Zaman ve dil* kategorisindedir; buradan *Dil ve bölge* sayfasına girilir.

2. Listede Türkçe yoksa *Dil ekle* ile Türkçe seçilip yüklenir. Türkçe, ekran dili olarak da kullanılabilir.
3. Türkçenin yanındaki üç nokta → *Dil seçenekleri* açılır. Klavyeler bölümünde *Klavye ekle* ile Türkçe F seçilir. Bu anda etkin klavye değişmez; Türkçe F yalnızca listeye eklenmiştir.
4. Win + Boşluk yüklü klavye düzenleri arasında geçiş yapar. Görev çubuğunun sağında etkin düzen görünür (örneğin TR F).
5. Win + Boşluk'a bir kez daha basılınca Türkçe Q'ya dönülür. Alt + Shift de dil ya da düzen geçişinin klasik kısayoludur ve ayarlardan değiştirilebilir.



Şekil 2.2.26: Türkçe klavye ve dil ekleme yolu: Ayarlar › Zaman ve dil › Dil ve bölge; Dil ekle › Türkçe › Yükle; Türkçe › Dil seçenekleri › Klavye ekle.

Sık yapılan hata

Yanlışlıkla Win + Boşluk ya da Alt + Shift'e basıp düzeni değiştirmek ve klavyenin "bozulduğunu" sanmak. Belirti, tuşların üzerindeki harflerden farklı harfler yazılmasıdır: ı yerine i, ş yerine ; çıkması gibi. Görev çubuğundaki dil göstergesine bakmak ve gerekirse Win + Boşluk ile doğru düzene dönmek sorunu çözer. Parola kutusunda bu durum özellikle can sıkar, çünkü yazılan karakterler görünmez.

2.2.9 Güvenlik

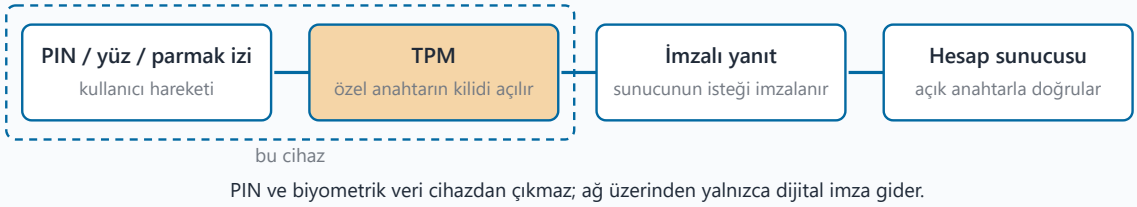
Bir bilgisayarın güvenliği birkaç katmandan oluşur. Oturum açma, bilgisayarı kimin kullanabileceğini denetler. Antivirüs ve güvenlik duvarı kötü amaçlı yazılımları ve ağdan gelen saldırıları durdurur. Kullanıcının alışkanlıkları da bütün bu önlemlerin işe yarayıp yaramayacağını belirler.

2.2.9.1 Windows Hello ve oturum güvenliği

Tanım 2.2.24

Windows Hello, Windows'ta parola yerine PIN, yüz tanıma ya da parmak izi ile oturum açma yöntemidir. Bu yöntemde PIN ya da **biyometrik** veri (yüz, parmak izi) cihazdan çıkmaz. Cihazın içindeki güvenlik yongasında (TPM) saklanan bir özel anahtarın kilidini açar; kimlik doğrulama bu anahtarla yapılır.

Şekil 2.2.27 doğrulamanın adımlarını gösterir. Kullanıcı PIN'ini girer ya da yüzünü kameraya gösterir. Bu hareket, TPM içindeki özel anahtarın kilidini açar. Hesap sunucusu cihaza bir istek gönderir; cihaz bu isteği özel anahtarla imzalar. Sunucu imzayı, daha önce kendisine kaydedilmiş açık anahtarla doğrular. Ağ üzerinden PIN değil, yalnızca dijital imza geçer.



Şekil 2.2.27: Windows Hello. PIN ya da biyometrik hareket bu cihazdaki TPM içinde özel anahtarın kilidini açar; anahtar sunucunun isteğini imzalar, sunucu imzayı açık anahtarla doğrular. PIN ve biyometrik veri cihazdan çıkmaz.

Bu düzen, kısa bir PIN'in neden uzun bir paroladan daha güvenli olabileceğini açıklar. Parola, hesabın kendisine aittir; çalınırsa dünyanın her yerinden hesaba girilebilir. PIN ise tek bir cihaza bağlıdır. Çalınan bir PIN, saldırganın elinde o cihaz yoksa hiçbir işe yaramaz. Üstelik TPM, art arda yanlış PIN denemelerini fark edip cihazı bir süre kilitler.

Örnek 2.2.21

4 haneli bir PIN'in kaç olası değeri vardır? 6 haneli bir PIN'in? Bunlar 8 karakterli, büyük harf, küçük harf ve rakam içerebilen bir parolayla nasıl karşılaştırılır?

1. Her hane 10 rakamdan biridir. 4 hane için $10 \times 10 \times 10 \times 10 = 10^4 = 10.000$ olasılık vardır.
2. 6 hane için $10^6 = 1.000.000$ olasılık vardır.
3. Parolada her karakter 26 küçük harf, 26 büyük harf ve 10 rakamdan, yani 62 karakterden biridir. 8 karakter için $62^8 \approx 2{,}18 \times 10^{14}$ olasılık vardır.

Olasılık sayısına bakınca parola çok daha güçlü görünür. Ancak bir saldırgan PIN'i yalnızca o cihazın başında ve sınırlı sayıda deneyebilir; TPM birkaç yanlış denemeden sonra bekleme süresi koyar. Parola ise sızdırılmış bir listeden çevrim içi denenebilir. PIN'in gücü, olasılık sayı-

sından değil cihaza bağlı olmasından gelir.

Oturum güvenliğinin öteki parçaları şunlardır:

- **Ekran kilidi.** Bilgisayardan ayrılırken Win + L ile oturum kilitlenir.
- **Çok faktörlü doğrulama** (*multi-factor authentication*). Hesaba girişte paroladan başka bir kanıt da istenir: telefona gelen kod ya da bir doğrulama uygulamasındaki onay. Parola çalınsa bile ikinci kanıt olmadan hesaba girilemez.
- **Kurtarma.** PIN unutulursa ya da cihaz değişirse hesaba nasıl dönüleceği önceden düzenlenmelidir: güncel bir telefon numarası, yedek e-posta adresi ya da kurtarma kodları.

Bu seçenekler Ayarlar → Hesaplar → Oturum açma seçenekleri sayfasındadır.

2.2.9.2 Windows Güvenliği

Windows, kötü amaçlı yazılımlara karşı korumayı kendi içinde getirir. Bu korumanın araçları **Windows Güvenliği** uygulamasında toplanır. Uygulama Ayarlar → Gizlilik ve güvenlik → Windows Güvenliği yoluyla ya da Başlat menüsünde "Windows Güvenliği" aranarak açılır. Tablo 2.2.20 başlıca bölümleri özetler.

Bölüm	Görevi
Virüs ve tehdit koruması	kötü amaçlı yazılımları tarar ve engeller; hızlı tarama ve gerçek zamanlı koruma buradadır
Güvenlik duvarı ve ağ koruması	ağdan gelen ve giden bağlantıları denetler; yetkisiz erişimi engeller
Uygulama ve tarayıcı denetimi	SmartScreen ile tehlikeli indirmelere ve kötü amaçlı sitelere karşı uyarır
Cihaz güvenliği	Güvenli Önyükleme ve TPM gibi donanım tabanlı korumaların durumunu gösterir
Hesap koruması	Microsoft hesabı ve Windows Hello oturum açma korumalarını gösterir

Tablo 2.2.20: Windows Güvenliği'nin bölümleri.

Virüs ve tehdit korumasını sağlayan yazılımın adı **Microsoft Defender**'dır. Defender Windows ile birlikte gelir ve ek bir antivirüs kurulmadığı sürece etkin kalır. En önemli özelliği **gerçek zamanlı korumadır**: bir dosya indirildiğinde, açıldığında ya da çalıştırıldığında Defender onu o anda tarar. Bu korumanın açık tutulması ve Defender'ın tanım güncellemelerinin aksatılmaması gerekir. Yeni çıkan kötü amaçlı yazılımlar ancak güncel tanımlarla tanınır.

Güvenlik duvarı (*firewall*), bilgisayarla ağ arasındaki trafiği kurallara göre süzen bir kapı bekçisidir. Örneğin dışarıdan gelen ve hiçbir uygulamanın beklemediği bir bağlantı isteğini engeller. **SmartScreen** ise indirilen dosyaların ve ziyaret edilen sitelerin bilinen tehlikeli listelerde olup olmadığına bakar. Tanınmayan bir program çalıştırılmak istendiğinde "Windows bilgisayarınızı korudu" uyarısını gösteren odur.

Cihaz güvenliği bölümünde adı geçen iki kavram, korumanın donanım katmanını oluşturur. **Güvenli Önyükleme** (*Secure Boot*), bilgisayar açılırken yalnızca imzası doğrulanmış yazılımın yüklenmesine izin verir; işletim sisteminden önce çalışmaya çalışan kötü amaçlı yazılımı durdurur. **TPM** (*Trusted Platform Module*, güvenilir platform modülü) ise anahtarları ve parolaları güvenli biçimde saklayan bir güvenlik yongasıdır. Windows Hello ve BitLocker bu yongayı kullanır.

> **Sık yapılan hata**
>
> Bir programı kurabilmek için gerçek zamanlı korumayı kapatmak. Bir kurulum dosyası "antivirüsü kapatın" diyorsa, büyük olasılıkla kötü amaçlıdır. Güvenilir kaynaktan gelen yazılımlar Defender açıkken kurulabilir. Koruma bir kez kapatılırsa, Windows onu bir süre sonra kendiliğinden yeniden açar; bu aradaki her indirme denetimsiz kalır.

2.2.9.3 Güvenli günlük kullanım

Araçlar ne kadar iyi olursa olsun, güvenliğin büyük kısmı kullanıcının alışkanlıklarına bağlıdır. Altı alışkanlık, sıradan saldırıların çoğunu boşa çıkarır:

- **Kilit.** Bilgisayardan ayrılırken ekran Win + L ile kilitlenir.
- **Güncelleme.** Windows ve uygulamalar güncel tutulur. Güncellemelerin çoğu bilinen güvenlik açıklarını kapatır.
- **Standart hesap.** Günlük işler standart hesapla yapılır; yönetici yetkisi yalnızca gerektiğinde kullanılır.
- **İndirme.** Yazılım Microsoft Store'dan ya da üreticinin resmî sitesinden edinilir.
- **Yedek.** Önemli dosyalar düzenli olarak ve ayrı bir konumda yedeklenir (2.2.10.5).
- **Şüpheli dosya.** Bilinmeyen kaynaktan gelen ekler ve çalıştırılabilir dosyalar açılmaz.

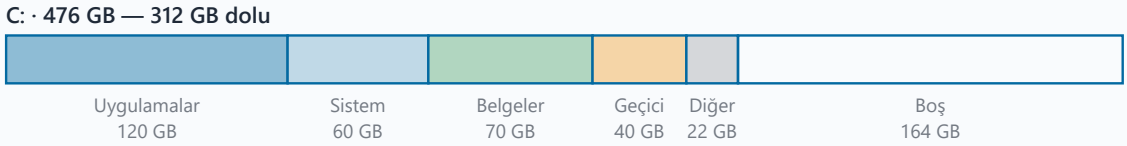
Bu alışkanlıklar birbirini tamamlar. Örneğin kötü amaçlı bir ek açılsa bile, kullanıcı standart hesaptaysa UAC sistem değişikliği için yönetici parolası ister ve saldırı orada durabilir. Saldırı başarılı olup dosyaları şifrelese bile, ayrı bir konumdaki güncel yedek kaybı en aza indirir.

2.2.10 Bakım

Bir bilgisayar zamanla dolar, yavaşlar ve güncelliğini yitirir. Disk geçici dosyalarla şişer, açılışta çalışan program sayısı artar, bir uygulama donar, bir güncelleme sorun çıkarır. Düzenli bakım bu sorunların çoğunu baştan önler. Önlenemeyenler için de Windows'un kurtarma araçları vardır.

2.2.10.1 Depolama yönetimi

Disk dolduğunda Windows güncelleme yükleyemez, uygulamalar geçici dosya yazamaz ve bilgisayar yavaşlar. Diskteki alanın neye harcandığı Ayarlar → Sistem → Depolama sayfasında görülür. Bu sayfa alanı uygulamalar, sistem dosyaları, belgeler ve geçici dosyalar gibi kategorilere ayırır (Şekil 2.2.28).



Şekil 2.2.28: C: sürücüsünde (476 GB, 312 GB dolu) alanın kategorilere göre dağılımı: Uygulamalar 120 GB, Sistem 60 GB, Belgeler 70 GB, Geçici 40 GB, Diğer 22 GB, Boş 164 GB.

Örnek 2.2.22

Şekil 2.2.28'deki sürücü için doluluk oranını, her kategorinin payını ve geçici dosyalar temizlenince boş alanın ne olacağını hesaplayalım.

1. Kategorilerin toplamı: 120 + 60 + 70 + 40 + 22 = 312 GB. Boş alan: 476 − 312 = 164 GB. Sayılar şekille tutarlıdır.
2. Doluluk oranı: 312 ÷ 476 ≈ 0,655, yani %65,5. Boş alan %34,5'tir.
3. Dolu alanın en büyük kısmı uygulamalardır: 120 ÷ 312 ≈ %38,5. Geçici dosyalar 40 ÷ 312 ≈ %12,8 yer tutar.
4. Geçici dosyalar silinirse dolu alan 312 − 40 = 272 GB'a iner, boş alan 164 + 40 = 204 GB'a çıkar. Boş alanın oranı 204 ÷ 476 ≈ %42,9 olur.

Bu sürücüde ilk temizlenecek kategori geçici dosyalardır, çünkü silinmeleri hiçbir veriyi kaybettirmez. Daha fazla alan gerekirse sonraki adım kullanılmayan uygulamaları kaldırmaktır.

Depolama sayfası iki yardımcı araç sunar:

- **Depolama Algılaması** (*Storage Sense*): açıldığında geçici dosyaları ve belirlenen kurallara göre Geri Dönüşüm Kutusu'ndaki eski öğeleri kendiliğinden temizler. Örneğin kutuda 30 günden uzun süre kalan dosyaları silecek biçimde ayarlanabilir. Bu kuralla 1 Ekim 2026'da silinen bir dosya 31 Ekim 2026'dan sonra kutudan da kalkar.
- **Temizleme önerileri**: büyük ya da uzun süredir açılmamış dosyaları, kullanılmayan uygulamaları ve geçici dosyaları listeler. Kullanıcı neyin silineceğine kendisi karar verir.

Sık yapılan hata

Alan açmak için listedeki her büyük dosyayı silmek. Temizleme önerilerindeki bir dosya, uzun süredir açılmadığı için önerilmiş olabilir; bu, gereksiz olduğu anlamına gelmez. Eski bir proje ya da bir dönemin fotoğrafları bu listede çıkabilir. Silmeden önce dosyanın gerekli olmadığı doğrulanır; emin olunamıyorsa dosya harici bir sürücüye taşınır.

2.2.10.2 Sürücüleri iyileştirme

HDD ile SSD veriyi çok farklı yollarla okur. Bu yüzden ikisine uygulanan bakım da farklıdır. Windows'un **Sürücüleri Birleştir ve İyileştir** aracı, Başlat menüsünde "birleştir" aranarak açılır. Araç her sürücünün türünü tanır ve ona uygun işlemi uygular (Tablo 2.2.21).

Sürücü	Windows'un uyguladığı bakım	Neden?
HDD	birleştirme	parçalanmış dosya parçalarını yan yana yerleştirmek
SSD	TRIM / iyileştirme	kullanılmayan blokları SSD denetleyicisine bildirmek

Tablo 2.2.21: Sürücü türüne göre bakım.

Tanım 2.2.25

Bir dosyanın blokları diskte art arda değil, dağınık yerlerde duruyorsa dosya **parçalanmıştır** (*fragmented*). **Birleştirme** (*defragmentation*), her dosyanın bloklarını diskte yan yana gelecek biçimde yeniden yerleştirme işlemidir. **TRIM** ise işletim sisteminin, silinen dosyaların artık kullanılmayan bloklarını SSD'ye bildirmesidir.

Parçalanma, dosyalar silinip yeni dosyalar yazıldıkça kendiliğinden oluşur. Silinen bir dosyanın bıraktığı boşluk küçükse, yeni dosyanın bir kısmı oraya, kalanı başka bir boşluğa yazılır. HDD'de bu bir sorundur, çünkü okuma kafası bir parçadan ötekine giderken mekanik olarak hareket etmek zorundadır. Her geçiş birkaç milisaniye sürer.



Şekil 2.2.29: HDD'de birleştirme öncesi ve sonrası. Önce A, B ve C dosyalarının blokları dağınıktır ve aralarda boş bloklar vardır. Birleştirmeden sonra her dosyanın blokları yan yanadır (AAAA BBB CC) ve boş alan sonda toplanmıştır.

Örnek 2.2.23

Şekil 2.2.29'daki diskte 12 blok vardır. Birleştirmeden önce sıra A A B _ B A _ C A B _ C'dir (_ boş blok). Her dosya kaç parçadır? Her parçaya geçiş ortalama 9,2 ms sürüyorsa, üç dosyayı okumak için konumlanmaya ne kadar zaman harcanır?

1. A dosyası 1., 2., 6. ve 9. bloklardadır. 1. ve 2. bloklar yan yanadır, ötekiler ayrıdır: 3 parça.
2. B dosyası 3., 5. ve 10. bloklardadır: 3 parça. C dosyası 8. ve 12. bloklardadır: 2 parça.
3. Toplam 3 + 3 + 2 = 8 parça vardır. Her parça için kafa yeniden konumlanır: 8 × 9,2 = 73,6 ms.
4. Birleştirmeden sonra her dosya tek parçadır: 3 konumlanma, 3 × 9,2 = 27,6 ms.

Bu küçük örnekte bile konumlanma süresi üçte birin biraz üstüne iner. Binlerce parçalı dosyası olan bir HDD'de fark saniyelere ulaşır.

SSD'de ise hareket eden bir kafa yoktur. Her blok, nerede olursa olsun aynı hızla okunur. Bu yüzden parçalanma SSD'yi yavaşlatmaz ve birleştirme gereksizdir. Üstelik birleştirme sırasında yapılan çok sayıda yazma, sınırlı sayıda yazma ömrü olan SSD'yi gereksiz yere yıpratır. SSD'nin asıl ihtiyacı TRIM'dir. Bir dosya silindiğinde SSD bunu kendiliğinden bilmez. TRIM bildirimi gelmezse SSD, silinmiş verinin bulunduğu blokları hâlâ dolu sanır ve yeni yazmalarda bu blokları önce boşaltmak zorunda kalır; yazma yavaşlar.

Sık yapılan hata

SSD'ye klasik HDD mantığıyla sık sık elle birleştirme uygulamak. Windows sürücü türünü tanır ve uygun işlemi kendiliğinden seçer: HDD'yi birleştirir, SSD'ye TRIM uygular. Araç varsayılan olarak haftalık çalışacak biçimde zamanlanmıştır. Bu ayarı kapatmamak yeterlidir.

2.2.10.3 Sistem Bilgisi ve Görev Yöneticisi

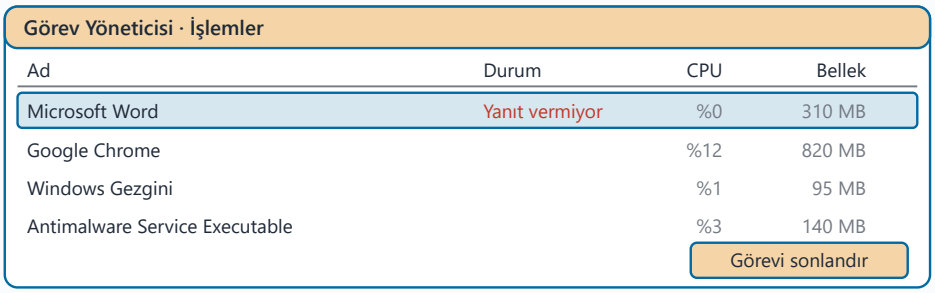
Bir sorunu çözmek için önce bilgisayarda ne olduğunu ve o anda neyin çalıştığını bilmek gerekir. Windows bunun için iki araç sunar.

Sistem Bilgisi, donanımın, işletim sisteminin, bileşenlerin ve kaynakların ayrıntılarını gösterir: işlemcinin modeli, RAM miktarı, Windows sürümü, BIOS/UEFI sürümü, aygıt sürücüleri. Başlat menüsünde “Sistem Bilgisi” aranarak açılır. Daha kısa bir özet Ayarlar → Sistem → Hakkında sayfasındadır. Bilgisayarın bir yazılımın sistem gereksinimlerini karşılayıp karşılamadığı buradan anlaşılır; destek isteyen birine bilgisayarın özellikleri de buradan bakılarak söylenir.

Tanım 2.2.26

Görev Yöneticisi (*Task Manager*), çalışan uygulamaları ve süreçleri, bunların işlemci (CPU), bellek, disk ve ağ kullanımını gösteren ve gerektiğinde bir süreci sonlandıran Windows aracıdır. Ctrl + Shift + Esc ile doğrudan, Win + X menüsünden ya da görev çubuğuna sağ tıklanarak açılır.

Burada geçen **süreç** (*process*), çalışmakta olan bir programdır. Bir uygulama birden çok süreçten oluşabilir; tarayıcılar örneğin her sekme için ayrı süreç açar. Görev Yöneticisi her sürecin ne kadar kaynak kullandığını sütunlarda gösterir (Şekil 2.2.30).



Şekil 2.2.30: Görev Yöneticisi'nin İşlemler sayfası. Durumu "Yanıt vermiyor" olan Microsoft Word seçilidir; CPU ve Bellek sütunları her işlemin kaynak kullanımını gösterir. Sağ üstte Görevi sonlandır düğmesi vardır.

Görev Yöneticisi birkaç sayfadan oluşur. Her sayfa farklı bir soruya yanıt verir (Tablo 2.2.22).

Sayfa	Ne gösterir?	Ne zaman bakılır?
İşlemler	uygulamalar ve arka plan işlemleri; CPU, bellek, disk ve ağ yüzdeleri	bilgisayar yavaşladığında, uygulama donduğunda
Performans	CPU, bellek, disk, ağ ve GPU'nun anlık grafikleri	darboğazın hangi donanımda olduğunu görmek için
Uygulama geçmişi	uygulamaların zaman içindeki kaynak kullanımı	hangi uygulamanın ağı ya da CPU'yu çok tükettiğini bulmak için
Başlangıç uygulamaları	oturum açılınca çalışan programlar ve etkileri	açılış süresi uzadığında
Kullanıcılar	oturum açmış kullanıcılar ve kaynak kullanımları	paylaşılan bilgisayarda
Ayrıntılar / Hizmetler	süreç kimliği (PID) ve Windows hizmetleri	ileri düzey sorun gidermede

Tablo 2.2.22: Görev Yöneticisi sayfaları.

Yüksek kaynak kullanımı her zaman bir sorun değildir. Windows Update, bir virüs taraması ya da arama dizininin oluşturulması kısa süre CPU'yu ve diski yoğun kullanabilir. Bu tür yükler birkaç dakikada geçer. Sürekli %100 disk ya da bellek kullanımı ise yavaşlığın tipik nedenidir. Bellek dolunca

Windows, RAM'deki verilerin bir kısmını diske yazar ve gerektiğinde geri okur; disk RAM'den binlerce kat yavaş olduğu için bilgisayar belirgin biçimde yavaşlar.

Örnek 2.2.24

8 mantıksal işlemcili bir bilgisayarda bir program sonsuz döngüye girmiş ve tek bir çekirdeği tamamen meşgul ediyor. Görev Yöneticisi bu program için CPU sütununda ne gösterir? Aynı durum 4 mantıksal işlemcili bir bilgisayarda olsaydı?

1. CPU yüzdesi bütün işlemci kapasitesine göre hesaplanır. 8 mantıksal işlemcinin her biri kapasitenin 100 ÷ 8 = %12,5'idir.
2. Tek işlemciyi tamamen kullanan program bu yüzden yaklaşık %12,5 görünür.
3. 4 mantıksal işlemcide aynı program 100 ÷ 4 = %25 görünür.

"Program yalnızca %12,5 kullanıyor, sorun değil" yorumu bu yüzden yanıltıcı olabilir. Program kendi çekirdeğini tümüyle tüketir. Performans sayfasındaki çekirdek grafiklerinde bir çekirdeğin sürekli tepede durduğu görülür.

Sık yapılan hata

Adı tanınmayan her süreci kötü amaçlı sanıp sonlandırmak ya da dosyasını silmek. "Antimalware Service Executable" gibi adlar ilk bakışta şüpheli görünür, ama bu süreç Microsoft Defender'ın kendisidir. Bilinmeyen bir süreç, adı güvenilir kaynaklarda araştırılmadan sonlandırılmaz ve silinmez. Yüksek kaynak kullanımı da tek başına kötü amaçlı yazılım kanıtı değildir.

2.2.10.4 Donan uygulama ve başlangıç uygulamaları

Görev Yöneticisi'nin en sık kullanıldığı iki durum, donan bir uygulamayı kapatmak ve açılışı yavaşlatan programları devre dışı bırakmaktır. İkisini bir örnek üzerinde adım adım izleyelim.

Örnek 2.2.25

Word tıklamalara yanıt vermiyor. Ayrıca bilgisayarın açılışı uzun sürüyor. İşlemler Ctrl + Shift + Esc ile açılan Görev Yöneticisi'nde sırayla şöyle yapılır:

Adım	Sayfa	İşlem	Sonuç
1	İşlemler	Görev Yöneticisi açılır	Word'ün durumu "Yanıt vermiyor"; CPU %0, bellek 312,4 MB
2	İşlemler	Word satırı seçilir, birkaç saniye beklenir	Word kendine gelmezse sonraki adıma geçilir

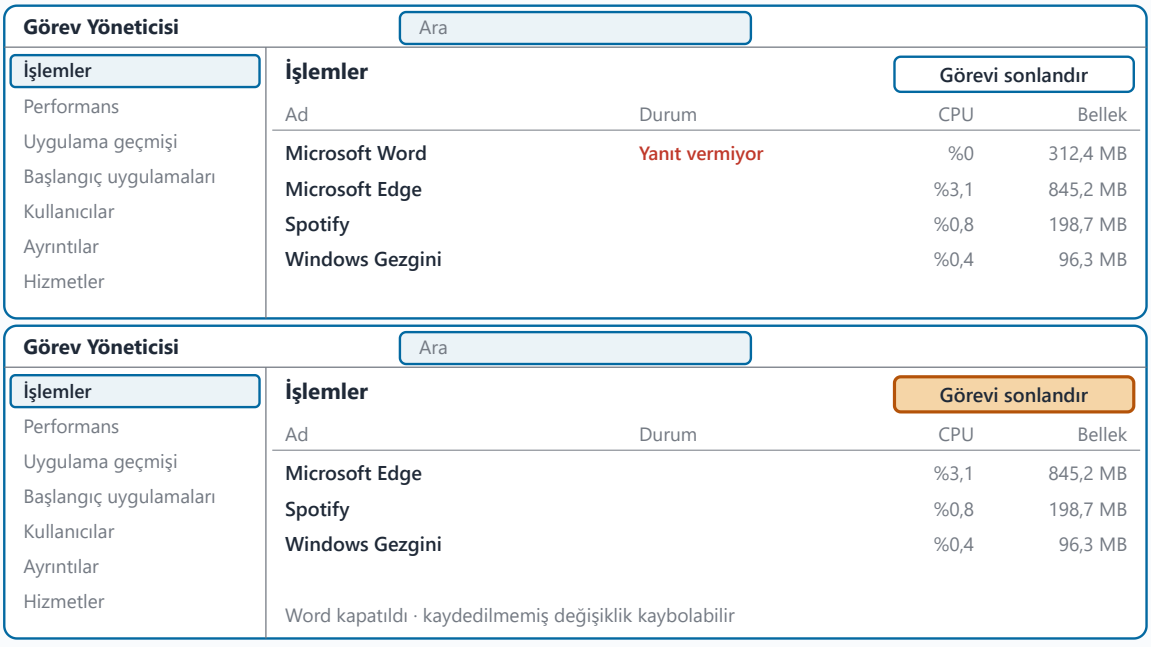
Adım	Sayfa	İşlem	Sonuç
3	İşlemler	*Görevi sonlandır*	Word süreci kapanır, belleği geri verilir; listede Edge, Spotify, Windows Gezgini kalır
4	Başlangıç uygulamaları	sayfa açılır	OneDrive, Spotify, Teams… listelenir; Spotify'ın başlangıç etkisi "Yüksek"
5	Başlangıç uygulamaları	Spotify seçilir, *Devre dışı bırak*	Spotify'ın durumu "Devre dışı"
6	İşlemler	bilgisayar yeniden başlatılır	Spotify kendiliğinden açılmaz, listede yoktur

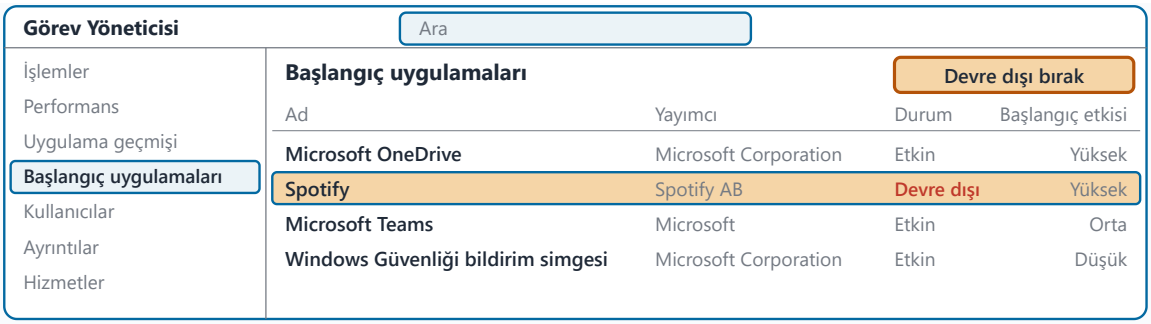
Tablo 2.2.23: Görev Yöneticisi'nde donan uygulama ve başlangıç uygulaması.

İkinci adımdaki bekleme önemlidir. "Yanıt vermiyor" her zaman donmak anlamına gelmez; uygulama uzun bir işlemi, örneğin büyük bir belgenin kaydını, bitirmeye çalışıyor olabilir. Üçüncü adımda Word süreci zorla kapatılır. Son kayıttan sonraki değişiklikler kaybolabilir; Word yeniden açılınca kendi otomatik kurtarma kopyasını önerebilir.

Görev sonlandırılınca listedeki bellek kullanımı da değişir. Dört işlemin toplam belleği 312,4 + 845,2 + 198,7 + 96,3 = 1.452,6 MB'tır. Word'ün 312,4 MB'ı geri verilince kalan üç işlem 1.140,2 MB kullanır.

Beşinci adımda Spotify kaldırılmaz; yalnızca oturum açılışında başlamaz. Gerektiğinde Başlat menüsünden elle açılır. Aynı liste Ayarlar → Uygulamalar → Başlangıç sayfasından da yönetilir.





Şekil 2.2.31: Görev Yöneticisi adım adım. Üstte İşlemler sayfasında Word'ün durumu "Yanıt vermiyor"dur. Ortada Görevi sonlandır ile Word kapatılmıştır. Altta Başlangıç uygulamaları sayfasında Spotify devre dışı bırakılmıştır.

Sık yapılan hata

Başlangıç listesindeki her şeyi devre dışı bırakmak. Antivirüs, dokunmatik yüzey sürücüsünün yardımcı programı ya da bulut eşitleme gibi bazı programların açılışta çalışması gerekir. Devre dışı bırakılacak olanlar, açılışta gerçekten gerekmeyen ve başlangıç etkisi yüksek programlardır: müzik çalar, oyun istemcisi, sohbet uygulaması gibi.

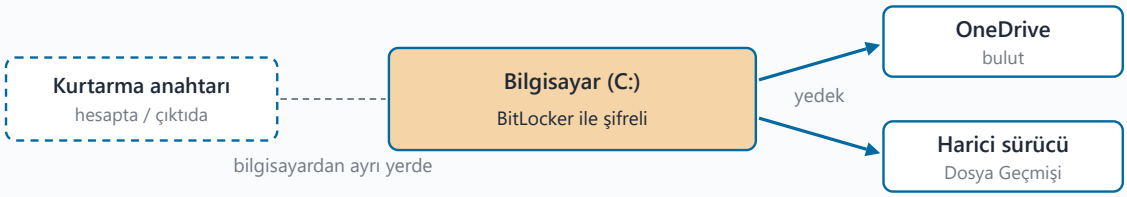
2.2.10.5 Güncelleme, yedekleme ve şifreleme

Windows Update, Windows'a güvenlik ve kalite güncellemelerini getiren hizmettir. Ayarlar → Windows Update sayfasından güncellemeler denetlenir, yüklenir ve geçmişi görülür. Güvenlik güncellemeleri genellikle her ayın ikinci salı günü yayımlanır. Bazı güncellemeler yeniden başlatma ister. *Etkin saatler* ayarı, Windows'un bu yeniden başlatmayı kullanıcının çalıştığı saatlerde yapmamasını sağlar. Güncellemeler kısa bir süre duraklatılabilir, ama tamamen ertelenmemelidir.

Tanım 2.2.27

Yedekleme (*backup*), önemli dosyaların ayrı bir konumda kopyasını tutmaktır. **Geri yükleme** (*restore*) ise dosyaları ya da sistemi daha önceki bir duruma döndürmektir.

Yedeğin işe yaraması için asıl dosyalarla birlikte kaybolmaması gerekir. Aynı diskteki bir kopya, disk bozulduğunda onunla birlikte gider. Bu yüzden yaygın bir kural vardır: **3-2-1 kuralı**. Önemli verinin en az 3 kopyası olmalı, bu kopyalar en az 2 farklı ortamda bulunmalı ve en az 1 kopya başka bir yerde, örneğin bulutta durmalıdır. Şekil 2.2.32 bu düzenin Windows'taki karşılığını gösterir: bilgisayardaki dosyalar hem OneDrive'a hem de harici bir sürücüye yedeklenir.



Şekil 2.2.32: BitLocker ile şifrelenmiş C: sürücüsündeki dosyalar OneDrive'a (bulut) ve Dosya Geçmişi ile harici sürücüye yedeklenir. BitLocker kurtarma anahtarı hesapta ya da çıktı olarak, bilgisayardan ayrı bir yerde saklanır.

Tanım 2.2.28

BitLocker, Windows'un desteklenen sürümlerinde (Pro, Enterprise, Education) bütün sürücüyü şifreleyen özelliktir. Şifreli sürücüdeki veri, doğru anahtar olmadan okunamaz. **Kurtarma anahtarı** (*recovery key*), TPM ya da parola anahtarı açamadığında sürücüye erişmeyi sağlayan 48 haneli sayıdır. Windows Home sürümünde benzer bir özellik "Cihaz şifrelemesi" adıyla bulunur.

Şifreleme, bilgisayar çalındığında veriyi korur. Hırsız diski söküp başka bir bilgisayara taksa bile dosyaları okuyamaz. Ancak aynı koruma sahibine de uygulanır. Anakart değişirse, TPM sıfırlanırsa ya da UEFI ayarları değişirse Windows açılışta kurtarma anahtarını ister. Anahtar bilinmiyorsa veriye bir daha erişilemez. Bu yüzden kurtarma anahtarı bilgisayardan ayrı bir yerde saklanır: Microsoft hesabında, bir çıktıda ya da bir USB bellekte.

Sık yapılan hata

BitLocker kurtarma anahtarını şifreli sürücünün içinde, örneğin Belgeler'deki bir metin dosyasında saklamak. Anahtar tam da sürücü açılamadığında gerekir; o anda sürücünün içindeki dosyaya ulaşılamaz. Anahtar her zaman bilgisayarın dışında tutulur.

2.2.10.6 Kurtarma seçenekleri

Bir şeyler ters gittiğinde başvurulacak araçlar, etkilerine göre hafiften ağıra doğru sıralanır. Hafif araç yalnızca bir dosyayı geri getirir; ağır araç bütün Windows'u yeniden kurar. Doğru seçim, sorunu çözen en hafif araçtır (Şekil 2.2.33, Tablo 2.2.24).



Şekil 2.2.33: Kurtarma seçenekleri hafiften ağıra: OneDrive yedekleme › Dosya Geçmişi › Geri Yükleme Noktası › Bu bilgisayarı sıfırla.

Araç	Neyi korur ya da geri getirir?	Koşul ve sınır
OneDrive klasör yedekleme	Masaüstü, Belgeler ve Resimler klasörlerini buluta yedekler; silinen dosya OneDrive geri dönüşüm kutusundan döner	internet ve OneDrive hesabı gerekir; bulut kutusu öğeleri sınırlı süre tutar
Dosya Geçmişi	dosyaların sürümlerini harici sürücüye düzenli kaydeder; eski ya da silinmiş sürüm geri getirilir	harici sürücü bağlı olmalıdır
Geri Yükleme Noktası	sistem dosyalarını ve ayarları önceki duruma döndürür	kişisel belgeleri yedeklemez; özelliğin önceden açık olması gerekir
Bu bilgisayarı sıfırla	Windows'u yeniden kurar; “Dosyalarımı koru” kişisel dosyaları tutar, uygulamaları kaldırır; “Her şeyi kaldır” hepsini siler	son çaredir; öncesinde yedek alınır

Tablo 2.2.24: Kurtarma araçları.

Dosya Geçmişi, Denetim Masası'ndaki “Dosya Geçmişi” sayfasından açılır ve varsayılan olarak saatte bir kopya alır. Geri yükleme noktası, Başlat menüsünde “Geri yükleme noktası oluştur” aranarak açılan Sistem Koruması sekmesinden etkinleştirilir. Windows, önemli güncellemelerden ve sürücü kurulumlarından önce kendiliğinden nokta oluşturur. Bu bilgisayarı sıfırla ise Ayarlar → Sistem → Kurtarma sayfasındadır.

Örnek 2.2.26

Dört farklı sorun için en uygun kurtarma aracı seçiliyor.

1. **OneDrive ile yedeklenen Belgeler'den dün silinen bir dosya geri isteniyor.** En hafif araç yeterlidir: OneDrive geri dönüşüm kutusundan dosya geri yüklenir.
2. **Harici diske düzenli kaydedilen bir dosyanın geçen haftaki sürümü isteniyor.** Dosya Geçmişi: harici sürücü takılır, dosyanın eski sürümü seçilip geri yüklenir.
3. **Yeni bir aygıt sürücüsü güncellemesinden sonra sistem kararsız, belgeler sağlam.** Sorun sistemdedir, belgelerde değil. Geri Yükleme Noktası sistemi güncellemeden önceki noktaya döndürür; belgelere dokunmaz.
4. **Sistem tamamen bozuk; kişisel dosyalar korunmalı, uygulamalar gidebilir.** Bu bilgisayarı sıfırla → Dosyalarımı koru. Windows yeniden kurulur, kişisel dosyalar kalır, uygulamalar kaldırılır; gerekenler elle yeniden yüklenir.

Her durumda daha ağır bir araç da sorunu çözebilirdi, ama gereksiz kayıplara yol açardı. Örneğin silinen tek bir dosya için bilgisayarı sıfırlamak bütün uygulamaları kaldırır ve dosyayı da geri getirmez.

Sık yapılan hata

Geri yükleme noktasını belge yedeği sanmak. Geri yükleme noktası yalnızca sistem dosyalarını, kayıt defterini ve yüklü programları kapsar. Silinen bir ödev dosyası geri yükleme noktasıyla geri gelmez. Belgeler için OneDrive, Dosya Geçmişi ya da başka bir dosya yedeği gerekir.

2.2.11 Özet

- **Masaüstü** ana çalışma alanıdır. **Başlat menüsü** uygulamalara, aramaya, hesaba ve güç komutlarına ulaştırır; sağ tık ya da Win + X hızlı bağlantı menüsünü açar. **Görev çubuğu** açık ve sabitlenmiş uygulamaları, sistem tepsisini, hızlı ayarları (Win + A) ve bildirimleri (Win + N) taşır.
- Pencere **küçültülünce** uygulama kapanmaz. **Yaslama** (Win + yön tuşları, Win + Z) pencereleri düzenler, Alt + Tab etkin pencereyi değiştirir. **Sanal masaüstleri** (Win + Ctrl + D, ← / →, F4) pencereleri gruplar; masaüstü kapatılınca pencereleri taşınır.
- Kopyala (Ctrl + C), Kes (Ctrl + X), Yapıştır (Ctrl + V) **panoyu** kullanır. Win + PrtSc dosya kaydeder; Win + Shift + S ve Alt + PrtSc panoya alır.
- Bir fiziksel disk birden çok **bölüme** ayrılabilir; her birim bir **sürücü harfi** alır. 512 GB'lık disk Windows'ta ≈ 476,8 GiB görünür.
- Dosya; ad, **uzantı**, yol ve özelliklerle tanımlanır. Uzantı ilişkili uygulamayı belirler; değiştirmek içeriği dönüştürmez. Dokuz karakter (\ / : * ? " < > |) ve CON, PRN gibi ayrılmış adlar kullanılamaz; klasik yol sınırı 260 karakterdir.
- **Mutlak yol** kökten başlar; **göreli yol** geçerli klasörden başlar ve `..` bir üst klasördür. “Belgeler” klasörünün gerçek yolu `C:\Users\<ad>\Documents`'tır.
- **Kopyalama** özgün öğeyi korur ve dosya sayısını artırır; **taşıma** konumu değiştirir. Sürükle-bırak aynı sürücüde taşır, farklı sürücüde kopyalar; Shift taşımayı, Ctrl kopyalamayı, Alt kısayolu zorlar.
- Delete **Geri Dönüşüm Kutusu**'na gönderir, Shift + Delete kalıcı siler; USB ve ağ konumlarında kutu kullanılmaz. **Kısayol** yalnızca yol tutan bir .lnk dosyasıdır; yedek değildir.
- **ZIP** dosyaları tek pakette toplar; metin iyi sıkışır, JPEG ve DOCX neredeyse hiç sıkışmaz. ZIP içindeki belge ayıklanmadan düzenlenmez. Diskteki boyut 4 KB'lık bloklara yuvarlanır.
- **Ayarlar** (Win + I) sistem, cihaz, ağ, hesap, uygulama, dil ve erişilebilirlik seçeneklerini toplar. Yazılar küçükse çözünürlük değil **ölçek** değiştirilir.
- **Standart kullanıcı** günlük işler içindir; **UAC** yönetici işlerinde onay ya da parola ister. **Varsayılan uygulama** kalıcı, Birlikte aç tek seferlik seçimdir.

- **Windows Hello** PIN'i cihaza bağlıdır ve ağdan geçmez. **Windows Güvenliği** virüs koruması, güvenlik duvarı, SmartScreen ve cihaz güvenliğini toplar.
- Depolama Algılaması ve temizleme önerileri alan açar. HDD **birleştirilir**, SSD'ye **TRIM** uygulanır. **Görev Yöneticisi** yanıt vermeyen uygulamayı sonlandırır ve başlangıç uygulamalarını sınırlar.
- Yedek ayrı bir ortamda tutulur (3-2-1). BitLocker kurtarma anahtarı bilgisayardan ayrı saklanır. Kurtarma araçları hafiften ağıra: OneDrive, Dosya Geçmişi, Geri Yükleme Noktası, Bu bilgisayarı sıfırla.

Hesap	Formül	Örnek
GB → GiB	bayt ÷ 2^{30}	512 GB → ≈ 476,8 GiB
Diskteki boyut	⌈boyut ÷ 4.096⌉ × 4.096	1.500 bayt → 4.096 bayt
Sıkıştırma oranı	sıkıştırılmış ÷ özgün	31 ÷ 120 ≈ %25,8
Mantıksal ekran alanı	çözünürlük ÷ ölçek	1920 × 1080, %150 → 1280 × 720
Doluluk oranı	dolu ÷ toplam	312 ÷ 476 ≈ %65,5
Tek çekirdekli yük	100 ÷ mantıksal işlemci sayısı	8 işlemci → %12,5
PIN olasılıkları	$10^{\text{hane sayısı}}$	4 hane → 10.000

Tablo 2.2.25: Notta kullanılan başlıca hesaplar.

2.2.12 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

masaüstü *desktop* · Başlat menüsü *Start menu* · görev çubuğu *taskbar* · sistem tepsisi *system tray* · hızlı ayarlar *quick settings* · bildirim merkezi *notification center* · bağlam menüsü *context menu* · pencere *window* · başlık çubuğu *title bar* · küçült / büyüt / geri yükle / kapat *minimize / maximize / restore down / close* · yaslama *snap* · görev görünümü *task view* · sanal masaüstü *virtual desktop* · klavye kısayolu *keyboard shortcut* · pano *clipboard* · ekran görüntüsü *screenshot* · Ekran Alıntısı Aracı *Snipping Tool* · sürücü *drive* · bölüm (birim) *partition / volume* · dosya *file* · klasör *folder* · uzantı *file name extension* · ilişkilendirme *file association* · ayrılmış ad *reserved name* · yol *path* · mutlak yol *absolute path* · göreli yol *relative path* · geçerli klasör *current folder* · üst klasör *parent directory* · kök *root* · Dosya Gezgini *File Explorer* · gezinti bölmesi *navigation pane* · joker karakter *wildcard* · gizli *hidden* · sürükle-bırak *drag and drop* · Geri Dönüşüm Kutusu *Recycle Bin* · kalıcı silme *permanent delete* · kısayol *shortcut (.lnk)* · arşiv / sıkıştırma *archive / compress* · tümünü ayıkla *extract all* · özellikler *properties* · nitelikler *attributes* · salt okunur *read-only* · ölçek *scale*

kullanıcı hesabı *user account* · yerel hesap *local account* · standart kullanıcı *standard user* · yönetici *administrator* · varsayılan uygulama *default app* · başlangıç uygulamaları *startup apps* · ekran okuyucu *screen reader* · klavye düzeni *keyboard layout* · biyometrik *biometric* · çok faktörlü doğrulama *multi-factor authentication* · güvenlik duvarı *firewall* · gerçek zamanlı koruma *real-time protection* · Güvenli Önyükleme *Secure Boot* · Depolama Algılaması *Storage Sense* · birleştirme *defragmentation* · parçalanmış *fragmented* · TRIM · Görev Yöneticisi *Task Manager* · süreç *process* · yanıt vermiyor *not responding* · görevi sonlandır *end task* · yedekleme *backup* · geri yükleme noktası *restore point* · kurtarma anahtarı *recovery key* · sıfırlama *reset*

2.2.13 Kendini Dene

1. Görev çubuğunda Word simgesinin altında çizgi var ama simge vurgulu değil; Excel simgesinin altında çizgi yok. Bu iki uygulamanın durumu nedir? Excel simgesine tıklanınca ne olur?

   Cevap

   Word açıktır ama etkin pencere değildir (arkada ya da küçültülmüş olabilir). Excel yalnızca sabitlenmiştir, açık değildir. Excel simgesine tıklanınca Excel başlar; simgesinin altında çizgi belirir ve etkin pencere Excel olur. Word açık kalır.

2. Masaüstü 1'de Word, Masaüstü 2'de Müzik açık; ekranda Masaüstü 2 var. Win + Ctrl + D, ardından Win + Ctrl + ← basılıyor. Hangi masaüstündeyiz ve kaç masaüstü vardır? Sonra Win + Ctrl + F4 basılırsa Müzik nereye gider?

   Cevap

   Win + Ctrl + D yeni ve boş Masaüstü 3'ü oluşturup ona geçer. Win + Ctrl + ← bir soldakine, Masaüstü 2'ye döner. Üç masaüstü vardır ve ekranda Masaüstü 2 bulunur. Win + Ctrl + F4 Masaüstü 2'yi kapatır; Müzik kapanmaz, soldaki Masaüstü 1'e taşınır.

3. Kutusunda "1 TB" yazan bir disk tek bölüm olarak D: yapılıyor. Windows bu birimin kapasitesini yaklaşık kaç GB gösterir?

   Cevap

   1 TB = 10^{12} bayt. Windows 2^{30} baytı 1 GB sayar: $10^{12} \div 1.073.741.824 \approx 931{,}3$. Birim yaklaşık 931 GB görünür. Bayt sayısı aynıdır; fark birimden gelir.

4. Geçerli klasör `C:\Users\Ali\Pictures\Tatil`. `..\..\Documents\odev.docx` göreli yolunun mut-

lak karşılığı nedir? Başındaki `..\..\` yazılmasaydı hangi klasörde aranırdı?

Cevap

Birinci `..` Tatil'den Pictures'a, ikinci `..` Pictures'tan Ali'ye çıkar. Sonra Documents'a inilir: `C:\Users\Ali\Documents\odev.docx`. `..\..\` yazılmasaydı `Documents` klasörü Tatil'in içinde, yani `C:\Users\Ali\Pictures\Tatil\Documents` olarak aranırdı ve bulunamazdı.

5. Şu adlardan hangileri Windows'ta dosya adı olamaz: `not|1.txt`, `PRN.docx`, `printer.docx`, `2026-10-23_rapor.pdf`, `a<b.txt`?

Cevap

`not|1.txt` (| yasak karakter), `PRN.docx` (PRN ayrılmış ad; uzantı durumu değiştirmez) ve `a<b.txt` (< yasak karakter) olamaz. `printer.docx` ve `2026-10-23_rapor.pdf` geçerlidir.

6. C:\Belgeler'de `a.docx` ve `b.jpg` var, E:\ (USB) boş. Sırayla: `a.docx` sürüklenip E:\'ye bırakılıyor; `b.jpg` sürüklenip `C:\Arsiv`'e bırakılıyor; E:\'deki `a.docx` Shift basılıyken `C:\Arsiv`'e sürükleniyor. Sonunda hangi konumda hangi dosyalar vardır?

Cevap

Birinci işlem farklı sürücü olduğu için kopyalamadır: `a.docx` hem C:\Belgeler'de hem E:\'de. İkinci işlem aynı sürücü olduğu için taşımadır: `b.jpg` C:\Arsiv'e geçer. Üçüncü işlem farklı sürücü ama Shift basılı olduğu için taşımadır: E:\'deki kopya C:\Arsiv'e geçer. Sonuç: C:\Belgeler'de `a.docx`; C:\Arsiv'de `b.jpg` ve `a.docx`; E:\ boş.

7. Boyutu 9.000 bayt olan bir dosyanın, 4.096 baytlık bloklar kullanan bir NTFS biriminde diskteki boyutu nedir? Windows dosyanın boyutunu yaklaşık kaç KB gösterir?

Cevap

9.000 ÷ 4.096 ≈ 2,2; yukarı yuvarlanınca 3 blok. Diskteki boyut 3 × 4.096 = 12.288 bayt (12,0 KB). Boyut 9.000 ÷ 1.024 ≈ 8,79 KB görünür.

8. Bir ZIP arşivine 600 KB'lık bir metin dosyası ve 3.000 KB'lık bir JPEG fotoğraf ekleniyor. Metin %25'ine, fotoğraf %99'una sıkışıyor. Arşivin yaklaşık boyutu ve toplam kazanç oranı nedir (arşiv bilgisinin payı yok sayılsın)?

Cevap

Metin 600 × 0,25 = 150 KB, fotoğraf 3.000 × 0,99 = 2.970 KB olur. Arşiv ≈ 150 + 2.970 = 3.120 KB'tır. Özgün toplam 3.600 KB; kazanç (3.600 − 3.120) ÷ 3.600 ≈ %13,3. Kazancın neredeyse tamamı metin dosyasından gelir.

9. 2560 × 1600 çözünürlüklü bir ekranda ölçek %200 seçiliyor. Uygulamalara kalan mantıksal alan nedir? Yazıları büyütmek için çözünürlüğü 1280 × 800'e düşürmek neden iyi bir çözüm değildir?

   Cevap

   2560 ÷ 2 = 1280, 1600 ÷ 2 = 800: mantıksal alan 1280 × 800'dür. Ölçekte bütün fiziksel pikseller kullanılır ve görüntü keskin kalır. Çözünürlüğü düşürmek ise her pikseli dörde (2 × 2) yayar; mantıksal alan aynı olur ama görüntü bulanıklaşır.

10. Bir öğrenci ödev dosyasını yanlışlıkla Shift + Delete ile sildi; dosya OneDrive ile yedeklenen Belgeler klasöründeydi. Ayrıca bir sürücü güncellemesinden sonra ses kartı çalışmıyor. Her sorun için hangi kurtarma aracı uygundur?

    Cevap

    Shift + Delete dosyayı Windows'un Geri Dönüşüm Kutusu'na göndermez. Ancak OneDrive silmeyi buluta yansıtırken dosyayı kendi geri dönüşüm kutusunda bir süre tutar; dosya oradan geri yüklenir. Ses kartı sorunu sistemdedir: geri yükleme noktası açıksa sistem güncellemeden önceki noktaya döndürülür. Bilgisayarı sıfırlamak iki sorun için de gereksiz ağırlıktadır.

2.2.14 İleri okuma

- Microsoft Destek, *Windows yardımı ve öğrenimi* (support.microsoft.com/tr-tr/windows): masaüstü, Dosya Gezgini, Ayarlar, yedekleme ve kurtarma konularını Windows 11'in güncel menü adlarıyla anlatan Türkçe resmî yardım sayfaları.
- Microsoft Destek, *Windows'da klavye kısayolları* (Türkçe sayfa): bu notta verilenlerden çok daha uzun, konulara göre düzenlenmiş kısayol listesi.
- Microsoft Learn, *Naming Files, Paths, and Namespaces* (learn.microsoft.com): yasak karakterlerin, ayrılmış adların, mutlak ve göreli yolların ve 260 karakter sınırının teknik açıklaması.
- Microsoft Learn, *Windows Hello for Business overview* (learn.microsoft.com): PIN'in neden cihazdan çıkmadığını ve özel anahtarın TPM'de nasıl korunduğunu anlatır.

2.2.15 Kaynaklar

1. Microsoft Support, “Windows yardımı ve öğrenimi” belgeleri (support.microsoft.com/tr-tr/windows).
2. Microsoft Support, Depolama Algılaması (Storage Sense) ve sürücü iyileştirme belgeleri.
3. Microsoft Learn, Windows Hello ve Windows güvenliği belgeleri (learn.microsoft.com).
4. Microsoft Learn, “Naming Files, Paths, and Namespaces”, Win32 dosya sistemi belgeleri.

KISIM I · BÖLÜM 3

Ofis Uygulamaları

Ofis uygulamaları belge hazırlamayı, veri çözümlemeyi ve bilgi sunmayı kolaylaştırır. Word sunumu stilleri, sayfa düzenini, bölüm sonlarını, içindekiler tablosunu, kaynakçayı ve adres mektup birleştirmeyi anlatır. Excel sunumu hücre başvurularını, formül ve işlevleri, sıralama ve filtrelemeyi, özet tabloyu ve grafikleri ele alır. PowerPoint sunumu ise sunu planlamayı, asıl slaydı, görsel tasarımı, geçiş ve animasyonları ve etkili sunum yapmayı gösterir.

BU BÖLÜMÜN NOTLARI

3.1 Microsoft Word

3.2 Microsoft Excel

3.3 Microsoft PowerPoint

KISIM I · BÖLÜM 3.1

Microsoft Word

Ödev raporu, laboratuvar föyü, dilekçe, proje önerisi, bitirme tezi… Bir mühendislik öğrencisinin yazdığı belgelerin çoğu bir kelime işlemcide hazırlanır. Word'ü yalnızca daktilo gibi kullanmak mümkündür: yazılır, kalın yapılır, Enter'a basılır. Ama belge uzadıkça bu yöntem çöker. Bir başlık eklenince bütün numaralar elle düzeltilir, bir sayfa kayınca içindekiler yanlış kalır, tek bir tablo için bütün belge yatay olur. Bu not, Word'e belgenin *görünüşünü* değil *yapısını* anlatmayı öğretir: stiller, bölümler, alanlar ve otomatik listeler. Konular sırayla ilerler ve her araç küçük bir belge üzerinde adım adım uygulanır. Her adımda belgenin hangi duruma geldiği tablolarla izlenir.

ÖĞRENME ÇIKTILARI

Belge yönetimi
Belge oluşturmak, kaydetmek, şablondan belge üretmek, açmak ve PDF olarak dışa aktarmak; Kaydet, Farklı Kaydet, Otomatik Kaydet ve Otomatik Kurtarma arasındaki farkı açıklamak.

Yapısal biçimlendirme
Karakter, paragraf, bölüm ve belge düzeyindeki biçimleri ayırt etmek; stil ile doğrudan biçimlendirmenin ilişkisini bilerek tutarlı bir belge kurmak.

Uzun belge
Başlık stilleri, çok düzeyli numaralandırma, içindekiler, dipnot, resim yazısı ve çapraz başvuruyu kullanmak; bölüm sonlarıyla sayfa yönünü, üstbilgiyi ve sayfa numarasını bölüm bazında yönetmek.

İş birliği ve paylaşım
Açıklama ve değişiklik izleme araçlarıyla ortak çalışmak; adres mektup birleştirmeyle toplu belge üretmek; belgeyi erişilebilirlik ve gizli içerik açısından denetleyip yazdırmak ya da PDF olarak paylaşmak.

3.1.1 Word ve kelime işlemci

Bilgisayarda yazı yazmanın en basit yolu bir düz metin düzenleyicisidir. Windows'taki Not Defteri bunun örneğidir: harfleri saklar ama kalın yazı, sayfa numarası ya da tablo bilmez. Bir rapor ise yalnızca harflerden oluşmaz. Başlıkları, paragrafları, tabloları, şekilleri, dipnotları ve sayfa düzeni vardır. Bu öğelerin hepsini tek dosyada tutan ve yöneten programlara kelime işlemci denir.

3.1.1.1 Kelime işlemci nedir?

Tanım 3.1.1

Kelime işlemci (*word processor*), metin yazmaya, düzenlemeye, biçimlendirmeye, sayfaya yerleştirmeye ve yazdırmaya yarayan uygulama yazılımıdır. Kelime işlemcide hazırlanan dosyaya **belge** (*document*) denir. Microsoft Word, Microsoft 365 paketinin kelime işlemcisidir; belgelerini varsayılan olarak `.docx` uzantılı dosyalarda saklar.

Bir kelime işlemcinin düz metin düzenleyicisinden farkı, metnin yanında metnin *nasıl görüneceğine* ve *ne anlama geldiğine* dair bilgiyi de saklamasıdır. Bir satırın 14 punto ve mavi olması bir görünüş bilgisidir. Aynı satırın "birinci düzey başlık" olması ise bir yapı bilgisidir. Word ikisini de saklar. Bu notun en önemli fikri de bu ayrımdır: görünüş yapıdan türetilirse belge kolay yönetilir. Ancak her satıra görünüş elle verilirse belge büyüdükçe dağılır.

Word'ün ekranda gösterdiği sayfa, yazdırıldığında kâğıtta çıkacak sayfaya çok yakındır. Bu yaklaşıma "ne görürsen onu alırsın" anlamında WYSIWYG (*what you see is what you get*) denir. Ekrandaki görüntü kâğıda benzediği için kullanıcı biçimi gözüyle ayarlamaya yönelir. Tuzak da buradadır. Boş satırlarla aşağı itilen bir başlık ekranda doğru durur, ama araya iki satır eklenince kayar. Bu yüzden notun büyük kısmı, görüntüyü gözle değil Word'ün araçlarıyla kurmayı anlatır.

3.1.1.2 Word'ün kullanım alanları

Word'ün kullanıldığı işler kabaca altı grupta toplanır. Her grup, Word'ün farklı bir yeteneğini öne çıkarır.

Kullanım	Örnek	Öne çıkan araçlar
Yazışma	dilekçe, mektup, resmî yazı	paragraf hizalama, şablon
Rapor	laboratuvar raporu, proje raporu, tez	başlık stilleri, içindekiler, tablo, resim yazısı, kaynakça
Form ve şablon	başvuru formu, toplantı tutanağı	şablon (`.dotx`), yer tutucular
Toplu belge	yüzlerce kişiye kişisel davet ya da not bildirimi	adres mektup birleştirme
İş birliği	danışmanın düzelttiği tez taslağı, ortak proje raporu	açıklama, değişiklik izleme, ortak düzenleme
Yayınlama	kâğıda basılan ya da e-postayla gönderilen son sürüm	baskı önizleme, PDF olarak dışa aktarma

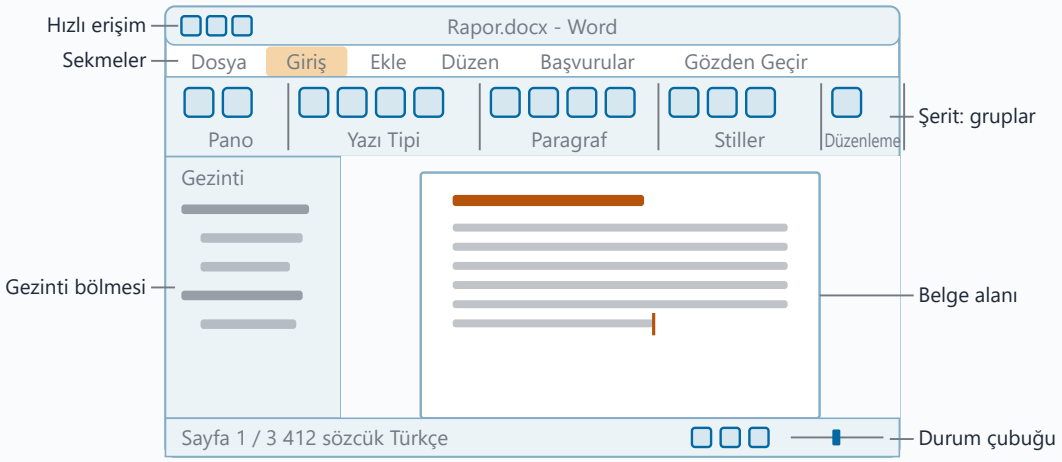
Tablo 3.1.1: Word'ün kullanım alanları ve her birinde öne çıkan araçlar.

Tablo 3.1.1 bir mühendislik öğrencisinin dört yıl boyunca karşılaşacağı belgelerin neredeyse hepsini kapsar. Dilekçe birkaç dakikada yazılır ve basit araçlar yeter. Rapor ise onlarca sayfadır, birkaç kez düzeltilir ve en çok yapısal araçlara ihtiyaç duyar. Bu notun ağırlığı da bu yüzden rapor ve tez gibi uzun belgelerdedir.

3.1.2 Arayüz

Word açıldığında ekranın büyük kısmını belge alanı kaplar. Üstte komutların bulunduğu şerit, altta durum çubuğu, solda isteğe bağlı gezinti bölmesi yer alır. Bu kesim, bu parçaların adlarını ve görevlerini tanıtır. Komutların yerini bilmek, geri kalan bütün işlerin temelidir.

3.1.2.1 Çalışma alanı



Şekil 3.1.1: Word penceresinin bölümleri. En üstte hızlı erişim araç çubuğu ve başlık çubuğu (dosya adı "Rapor.docx"), altında sekmeler ve şerit grupları (Pano, Yazı Tipi, Paragraf, Stiller, Düzenleme), solda gezinti bölmesi, ortada belge alanı, en altta durum çubuğu yer alır.

Şekil 3.1.1 Word penceresini parçalarına ayırır. Her parçanın görevi şöyledir:

- **Başlık çubuğu** açık belgenin adını gösterir. Belge buluta kaydedilmişse Otomatik Kaydet anahtarı da buradadır.
- **Hızlı erişim araç çubuğu**, Kaydet ve Geri Al gibi sık kullanılan birkaç komutu her zaman görünür tutar. Hangi düğmelerin burada duracağını kullanıcı seçebilir.
- **Şerit** (*ribbon*), Word'ün bütün komutlarını sekmelere ve gruplara ayırarak sunan bölgedir.
- **Gezinti bölmesi** (*navigation pane*) belgedeki başlıkları bir ağaç biçiminde listeler; bir başlığa tıklanınca o yere gidilir. Arama sonuçları da burada görünür.
- **Belge alanı** yazının yazıldığı ve sayfaların görüldüğü yerdir. Yanıp sönen dikey çizgi, yani **imleç**, yazılan karakterin nereye ekleneceğini gösterir.
- **Durum çubuğu** sayfa numarasını ("Sayfa 1 / 3"), sözcük sayısını, belge dilini, görünüm düğmelerini ve yakınlaştırma sürgüsünü gösterir.

Tanım 3.1.2

Şerit, komutları üç düzeyde düzenler: **sekme → grup → komut**. Sekme, şeridin üstündeki adlardan biridir (Giriş, Ekle, Düzen…). Her sekme birkaç gruba ayrılır (Giriş sekmesinde Pano, Yazı Tipi, Paragraf, Stiller, Düzenleme). Her grupta da ilgili komut düğmeleri bulunur. **Bağlamsal sekme** (*contextual tab*) ise yalnızca belirli bir nesne seçiliyken görünen ek sekmedir.

Bu notta bir komutun yeri "sekme → grup → komut" biçiminde yazılır. Örneğin "Giriş → Paragraf → Ortala", Giriş sekmesindeki Paragraf grubunda bulunan Ortala düğmesi demektir. Grup adı belirsizlik yoksa atlanır: "Düzen → Kesmeler" gibi.

Bağlamsal sekmeler şeridi sade tutar. Belgede bir tablo seçilince şeridin sağında **Tablo Tasarımı** ve tablo için ayrı bir **Düzen** sekmesi belirir. Bir resim seçilince **Resim Biçimi** sekmesi gelir. Seçim kalkınca bu sekmeler de kaybolur. Bir tablonun kenarlık rengini arayan kullanıcı, önce tabloya tıklamazsa komutu hiçbir sekmede bulamaz.

Sık yapılan hata

Tabloya ya da resme özgü bir komutu, nesne seçili değilken şeritte aramak. Bu komutlar bağlamsal sekmelerdedir ve yalnızca nesne seçiliyken görünür. Önce nesnenin içine ya da üstüne tıklanır, sonra şeridin sağında beliren sekmeye geçilir.

3.1.2.2 Temel sekmeler



Şekil 3.1.2: Giriş sekmesi seçiliyken şerit. Soldan sağa Pano, Yazı Tipi (yazı tipi adı "Calibri", boyut 11), Paragraf, Stiller (Normal, Başlık 1, Başlık 2) ve Düzenleme grupları.

Word açıldığında **Giriş** sekmesi seçilidir, çünkü en sık kullanılan komutlar buradadır (Şekil 3.1.2). Diğer sekmelerin her biri belge hazırlamanın bir aşamasına karşılık gelir. Tablo 3.1.2 bu eşlemeyi verir.

Sekme	Temel görev	Örnek komutlar
Dosya	belgenin bütünüyle ilgili işlemler	Yeni, Aç, Bilgi, Kaydet, Farklı Kaydet, Yazdır, Dışa Aktar
Giriş	metin, paragraf, stil ve pano işlemleri	Kopyala, Yapıştır, Kalın, Ortala, Başlık 1, Bul

Sekme	Temel görev	Örnek komutlar
Ekle	belgeye nesne ekleme	Tablo, Resimler, Şekiller, Bağlantı, Yer İşareti, Üst Bilgi, Sayfa Numarası
Tasarım	belgenin genel görünümü	Temalar, Renkler, Yazı Tipleri, Filigran
Düzen	sayfa yerleşimi	Kenar Boşlukları, Yönlendirme, Boyut, Sütunlar, Kesmeler
Başvurular	uzun belgenin yardımcı öğeleri	İçindekiler, Dipnot Ekle, Alıntı Ekle, Resim Yazısı Ekle, Dizin
Postalar	toplu belge üretimi	Zarflar, Etiketler, Alıcıları Seç, Birleştirme Alanı Ekle
Gözden Geçir	denetim ve iş birliği	Yazım ve Dil Bilgisi, Yeni Açıklama, Değişiklikleri İzle, Kabul Et
Görünüm	belgenin ekranda gösterilişi	Okuma Modu, Yazdırma Düzeni, Gezinti Bölmesi, Cetvel, Yakınlaştır

Tablo 3.1.2: Word'ün temel sekmeleri ve görevleri.

Sekmelerin sırası rastgele değildir; bir belgenin hazırlanma sırasını kabaca izler. Önce metin yazılıp biçimlendirilir (Giriş), sonra tablo ve şekil eklenir (Ekle). Sayfa düzeni ayarlanır (Tasarım, Düzen), içindekiler ve kaynakça kurulur (Başvurular). En sonda belge gözden geçirilir (Gözden Geçir). Postalar sekmesi yalnızca toplu belge üretirken gerekir.

Örnek 3.1.1

Bir raporda yapılacak beş iş için doğru sekme hangisidir?

İş	Sekme → komut	Gerekçe
Sayfanın sağ ve sol boşluğunu 3 cm yapmak	Düzen → Kenar Boşlukları	sayfa yerleşimi
Ölçüm grafiğinin altına “Şekil 1” yazısı eklemek	Başvurular → Resim Yazısı Ekle	numaralı yardımcı öğe
Danışmanın düzeltmelerini kabul etmek	Gözden Geçir → Kabul Et	iş birliği
Kapağa üniversite logosunu koymak	Ekle → Resimler	nesne ekleme
Bütün belgenin renk düzenini değiştirmek	Tasarım → Renkler	belge geneli görünüm

Her satırda karar aynı soruyla verilir: iş metnin kendisiyle mi, sayfayla mı, belge geneliyle mi, yoksa belgeye eklenen bir nesneyle mi ilgili?

3.1.2.3 Belge görünümleri

Aynı belge ekranda farklı biçimlerde gösterilebilir. Görünüm yalnızca ekrandaki sunuşu değiştirir; belgenin içeriğine, biçimine ya da yazdırılan çıktıya dokunmaz. Görünüm, Görünüm sekmesinden ya da durum çubuğunun sağındaki küçük düğmelerden seçilir.

Görünüm	Ne gösterir?	Ne zaman kullanılır?
Yazdırma Düzeni	sayfaları kâğıttaki gibi, kenar boşlukları ve üstbilgilerle	varsayılan; düzen ayarlarken

Görünüm	Ne gösterir?	Ne zaman kullanılır?
Okuma Modu	metni ekrana uyan sütunlarda, şerit olmadan	uzun bir belgeyi yalnızca okurken
Web Düzeni	sayfa sonu olmadan, tarayıcıdaki gibi akan metin	web'de yayımlanacak içerik, geniş tablolar
Anahat	başlıkları düzeylerine göre girintili; alt düzeyler daraltılabilir	uzun belgenin bölümlerini yeniden düzenlerken
Taslak	yalnızca metin; üstbilgi, resim konumu, sayfa kenarı yok	hızlı yazma ve düzenleme

Tablo 3.1.3: Word'ün belge görünümleri.



Şekil 3.1.3: Anahat görünümü. ⊕ işaretli başlıkların altında içerik vardır ve daraltılmıştır; ⊖ işaretli "2.1 Veri" açıktır ve altındaki paragraf (○) görünür.

Anahat görünümü özellikle uzun belgelerde işe yarar (Şekil 3.1.3). Bir başlık sürüklenince altındaki bütün alt başlıklar ve paragraflar da onunla birlikte taşınır. Örneğin "3 Sonuç" bölümü "2 Yöntem"in önüne alınmak istenirse, onlarca paragrafı kesip yapıştırmak yerine Anahat görünümünde tek başlık sürüklenir. Bu görünümün çalışması için başlıkların başlık stilleriyle yazılmış olması gerekir. Stiller 3.1.5.5'te ele alınır.

3.1.3 Belge yönetimi

Bir belgenin yaşamı boş bir sayfayla ya da bir şablonla başlar. İçerik yazılır, kaydedilir, gözden geçirilir ve sonunda yazdırılır ya da PDF olarak paylaşılır. Bu kesim, bu yaşam döngüsünün dosya tarafını anlatır: kaydetmenin türleri, bulutta otomatik kayıt, şablonlar ve bir belgenin içyapısı.

3.1.3.1 Belge oluşturma ve kaydetme

Bir belgenin hazırlanması beş adımlık bir akışla özetlenebilir: yeni belge ya da şablon açılır, içerik oluşturulur, kaydedilir, gözden geçirilir, son olarak PDF'e aktarılır ya da yazdırılır. Kaydetme bu akışın yalnızca bir kez yapılan adımı değildir; çalışma boyunca defalarca tekrarlanır.

Tanım 3.1.3

Kaydet (`Ctrl+S`), pencerede açık olan belgedeki değişiklikleri aynı dosyanın üzerine yazar. **Farklı Kaydet** (`F12`) ise belgeyi yeni bir ad, konum ya da biçimle *ayrı bir dosya* olarak yazar; ardından pencere artık bu yeni dosyayla çalışır. Yeni bir belge ilk kez kaydedilirken Kaydet de Farklı Kaydet gibi davranır ve dosya adı, konum ve biçim sorar.



Şekil 3.1.4: Kaydet ile Farklı Kaydet. Kaydet aynı dosyayı (`rapor.docx`) günceller. Farklı Kaydet yeni bir dosya oluşturur: düzenlenebilir kopya için `rapor_v2.docx` , sabit paylaşım kopyası için `rapor.pdf` .

Şekil 3.1.4 iki komutun farkını gösterir. Farklı Kaydet'in az bilinen sonucu şudur: komut verildikten sonra pencere eski dosyayla değil, yeni oluşan dosyayla çalışır. Bundan sonraki her Kaydet yeni dosyayı günceller, eskisi olduğu gibi kalır. Aşağıdaki örnek bu davranışı adım adım izler.

Örnek 3.1.2

Diskte `deney.docx` var ve içinde yalnızca giriş paragrafı yazılı. Dosya açılıyor ve aşağıdaki işlemler yapılıyor. Her adımdan sonra diskteki dosyaların içeriği nedir?

Adım	İşlem	Pencerede açık dosya	`deney.docx`	`deney_v2.docx`
0	dosya açılır	deney.docx	giriş	yok
1	"Yöntem" paragrafı yazılır	deney.docx	giriş	yok
2	`Ctrl+S`	deney.docx	giriş + yöntem	yok
3	"Sonuç" paragrafı yazılır	deney.docx	giriş + yöntem	yok
4	Farklı Kaydet → `deney_v2.docx`	deney_v2.docx	giriş + yöntem	giriş + yöntem + sonuç
5	"Tartışma" paragrafı yazılır, `Ctrl+S`	deney_v2.docx	giriş + yöntem	giriş + yöntem + sonuç + tartışma

Adım 1 ve 3'te değişiklik yalnızca bellektedir; diske ancak kaydetme komutuyla geçer. Adım 4'te "Sonuç" paragrafı eski dosyaya hiç yazılmamıştır. Adım 5'teki Kaydet ise `deney.docx` 'e değil, pencerede açık olan `deney_v2.docx` 'e yazar. Eski dosya, adım 2'deki hâlinde kalır.

Bir belgenin hangi biçimde kaydedileceği, belgeyle ne yapılacağına bağlıdır. Düzenlemeye devam edilecek belge Word belgesi olarak, değiştirilmeden paylaşılacak son sürüm PDF olarak saklanır. Tablo 3.1.4 Word'ün kullandığı başlıca dosya biçimlerini özetler.

Uzantı	Biçim	Kullanım
`.docx`	Word belgesi	düzenlenebilir belge; varsayılan biçim
`.docm`	makro içerebilen Word belgesi	makrolu belge; açarken dikkat gerekir
`.dotx`	Word şablonu	yeni belgeler için başlangıç dosyası
`.dotm`	makro içerebilen Word şablonu	örneğin boş belgelerin varsayılanlarını tutan `Normal.dotm`
`.pdf`	taşınabilir belge biçimi	düzeni her cihazda aynı kalan paylaşım ve baskı kopyası
`.doc`	eski Word belgesi (2003 ve öncesi)	eski dosyaları açmak; yeni belge için önerilmez

Tablo 3.1.4: Word ile ilgili başlıca dosya biçimleri.

Uzantının sonundaki harf bir ipucu taşır: `x` makro içermeyen, `m` makro içerebilen dosyayı gösterir. Makrolar 3.1.11.3'te ele alınır. PDF dosyası Word'de açılabilir ama Word onu düzenlenebilir bir belgeye dönüştürerek açar; dönüşümde düzen bozulabilir. Bu yüzden PDF'e aktarılan her belgenin `.docx` aslı da saklanır.

Sık yapılan hata

Ödevi yalnızca PDF olarak saklamak. PDF düzeni korur ama düzenlemeye uygun değildir; hocanın istediği küçük bir düzeltme için bile belge yeniden yazılabilir. Doğru alışkanlık, her zaman `.docx` aslını saklamak ve teslim için ondan PDF üretmektir.

3.1.3.2 Otomatik Kaydet, OneDrive ve Otomatik Kurtarma

Kaydetmeyi unutmak, belge hazırlamanın en eski kazalarından biridir. Microsoft 365 bu kaza için iki ayrı önlem sunar. Adları birbirine benzese de işleyişleri farklıdır.

Tanım 3.1.4

Otomatik Kaydet (*AutoSave*), belgedeki değişiklikleri birkaç saniyede bir kendiliğinden buluttaki dosyaya yazan özelliktir. Yalnızca OneDrive ya da SharePoint'te duran dosyalarda çalışır ve başlık çubuğundaki anahtarla açılıp kapanır. **Otomatik Kurtarma** (*AutoRecover*) ise belli aralıklarla ayrı bir kurtarma kopyası tutar. Word beklenmedik biçimde kapanırsa, yeniden açılışta bu kopyayı önerir.

Bilgisayardaki bir klasörde, örneğin masaüstünde duran yerel bir dosyada Otomatik Kaydet anahtarı açılamaz. Böyle bir dosya `Ctrl+S` ile elle kaydedilir. Otomatik Kaydet açık olsa bile `Ctrl+S`'ye basmanın hiçbir zararı yoktur. Bu alışkanlık, belgenin yerel mi bulutta mı olduğunu düşünmeden güvende

kalmayı sağlar.

Bulutta tutulan dosyanın iki ek yararı vardır. Birincisi **sürüm geçmişi**dir (*version history*): dosyanın eski sürümleri tarih ve saatleriyle listelenir. Önceki bir sürüm açılıp incelenebilir ve gerekirse geri yüklenebilir. İkincisi **ortak çalışma**dır: aynı bulut dosyasında birden çok kişi eş zamanlı çalışabilir ve birbirinin değişikliklerini kısa sürede görür.

Özellik	Nereye yazar?	Ne zaman?	Koşul
Kaydet	asıl dosyanın üzerine	kullanıcı istediğinde	her dosyada çalışır
Otomatik Kaydet	buluttaki asıl dosyaya	sürekli, birkaç saniyede bir	dosya OneDrive ya da SharePoint'te olmalı
Otomatik Kurtarma	ayrı bir kurtarma kopyasına	belli aralıklarla	yalnızca beklenmedik kapanmada işe yarar
Sürüm Geçmişi	bulutta eski sürümler	kendiliğinden tutulur	dosya bulutta olmalı

Tablo 3.1.5: Kaydetmeyle ilgili özelliklerin karşılaştırılması.

Örnek 3.1.3

İki öğrenci aynı ödev üzerinde çalışıyor. Elif dosyayı OneDrive klasöründe tutuyor, Can ise masaüstünde. İkisi de bir saat boyunca hiç `Ctrl+S`'ye basmıyor. Sonra elektrik kesiliyor.

1. **Elif.** Dosya bulutta olduğu için Otomatik Kaydet açıktır. Değişiklikler birkaç saniyede bir buluta yazılmıştır. Bilgisayar açılınca dosya son hâliyle gelir; kayıp en fazla son birkaç saniyedir.
2. **Can.** Yerel dosyada Otomatik Kaydet çalışmaz. Asıl dosya bir saat önceki hâlindedir. Word yeniden açılınca Otomatik Kurtarma'nın son kopyasını önerir. Bu kopya son kurtarma anına kadarki değişiklikleri içerir; ondan sonrası kaybolur.
3. **Sonuç.** Otomatik Kurtarma bir emniyet ağıdır, kaydetmenin yerine geçmez. Kurtarılan kopya hemen Farklı Kaydet ile asıl dosya olarak saklanmalıdır.

Otomatik Kaydet düğmesinin adı, yeri ve hatta varlığı Word sürümüne ve güncellemelere göre değişebilir. Örneğin kalıcı lisansla satılan bazı Office sürümlerinde bu özellik bulunmayabilir. Kullanılan sürümde özelliğin olup olmadığı başlık çubuğunda ve Dosya → Bilgi bölümünde görülür.

3.1.3.3 Şablonlar

Her dönem aynı biçimde birkaç dilekçe yazan bir öğrenci düşünelim. Her seferinde kenar boşlukları ayarlanır, tarih sağa yaslanır, imza yeri bırakılır. Bu tekrarı ortadan kaldıran araç şablondur.

Tanım 3.1.5

Şablon (*template*), yeni belgelere başlangıç olan ve `.dotx` uzantısıyla saklanan Word dosyasıdır. Stilleri, kenar boşluklarını, sabit metni ve doldurulacak yerleri gösteren **yer tutucuları** içerir. Bir şablon açıldığında Word şablonun kendisini değil, ondan türetilmiş adsız bir kopyayı açar.



Şekil 3.1.5: Şablondan yeni belge. `Dilekce.dotx` şablonunda [Ad Soyad] ve [Tarih] yer tutucuları var. Dosya → Yeni ile her açılışta ayrı bir belge oluşur: `Belge1.docx` (Ayşe Yılmaz, 28.09.2026) ve `Belge2.docx` (Mehmet Kaya, 30.09.2026). Şablonun kendisi değişmez.

Şekil 3.1.5 bu davranışı gösterir. Şablona çift tıklanıp ad ve tarih doldurulduktan sonra `Ctrl+S`'ye basılırsa Word dosya adı sorar, çünkü açık belge adsız bir kopyadır. Şablon dosyası değişmeden kalır ve aynı şablondan istenildiği kadar dilekçe üretilebilir.

Şablonların iki kaynağı vardır. Hazır şablonlar Dosya → Yeni ekranında listelenir: özgeçmiş, rapor, toplantı notu, takvim gibi. Kendi şablonunu hazırlamak için önce örnek bir belge düzenlenir, sonra Dosya → Farklı Kaydet ile kayıt türü olarak "Word Şablonu (`*.dotx`)" seçilir.

Word'ün özel bir şablonu daha vardır: `Normal.dotm`. Boş belge açıldığında kullanılan varsayılan yazı tipi, boyut, kenar boşlukları ve stiller bu dosyadan gelir. Uzantıdaki `m`, bu şablonun makro da saklayabildiğini gösterir. Normal.dotm'deki bir değişiklik bundan sonra açılan bütün boş belgeleri etkiler. Bu yüzden ona yalnızca bilinçli olarak dokunulur.

Örnek 3.1.4

Laboratuvar raporları için bir şablon hazırlanacak. Adımlar ve her adımın sonucu şöyledir:

1. Boş bir belgede kenar boşlukları 2,5 cm yapılır; Normal ve Başlık 1 stilleri istenen yazı tipine getirilir.
2. Kapağa sabit metin yazılır: ders adı, "Deney No:", "Öğrenci:". Doldurulacak yerlere köşeli ayraçlı yer tutucular konur: [Deney adı], [Ad Soyad], [Tarih].
3. "Giriş", "Yöntem", "Sonuçlar", "Tartışma" başlıkları Başlık 1 stiliyle yazılır; altları boş bırakılır.

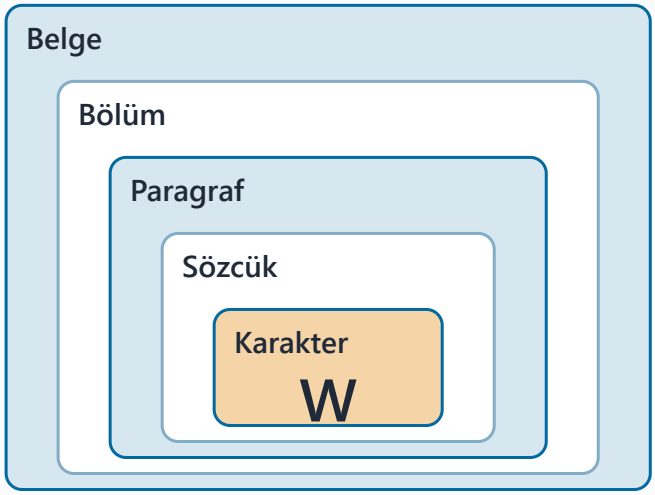
4. Dosya → Farklı Kaydet → kayıt türü “Word Şablonu”: `LabRaporu.dotx` .
5. Her yeni deneyde şablon açılır, yer tutucular doldurulur, rapor `Deney3_Rapor.docx` gibi bir adla kaydedilir. Şablon bir sonraki deney için değişmeden bekler.

Sık yapılan hata

Şablon yerine eski bir raporu açıp içeriğini silerek yeni rapor yazmak. Eski rapordan kalan gizli öğeler (açıklamalar, bekleyen değişiklikler, yazar bilgisi, unutulmuş bir paragraf) yeni belgeye taşınır. Üstelik eski dosyanın üzerine yanlışlıkla kaydetme riski de vardır. Tekrar eden belge türü için bir kez şablon hazırlamak hem daha temiz hem daha güvenlidir.

3.1.3.4 Belgenin yapı düzeyleri

Bir Word belgesi rastgele bir harf yığını değildir; iç içe geçmiş düzeylerden oluşur. Bu düzeyleri bilmek, bir biçimlendirme komutunun belgenin hangi parçasını etkileyeceğini önceden kestirmeyi sağlar.



Şekil 3.1.6: Belgenin yapı düzeyleri iç içe kutular biçiminde: belge bölümlerden, bölüm paragraflardan, paragraf sözcüklerden, sözcük karakterlerden oluşur.

Tanım 3.1.6

Word belgesinin yapı düzeyleri içten dışa şöyledir: **karakter** (tek harf, rakam, boşluk ya da işaret), **sözcük**, **paragraf** (Enter'a basılana kadar yazılan metin), **bölüm** (*section*; kendi sayfa ayarları olan belge parçası) ve **belge**. Her biçimlendirme komutu bu düzeylerden birine uygulanır.

Şekil 3.1.6 ve Tablo 3.1.6 bu düzeyleri ve her düzeyde ayarlanan biçimleri bir arada gösterir. Örneğin kalın yazı karakter düzeyinde, hizalama paragraf düzeyinde, sayfa yönü bölüm düzeyinde, tema ise belge düzeyindedir.

Düzey	Ayarlanan biçimler	Nasıl uygulanır?
Karakter	yazı tipi, boyut, kalın, italik, renk, üst/alt simge	seçili metne; seçim yoksa bundan sonra yazılacak metne
Paragraf	hizalama, girinti, satır ve paragraf aralığı, sekmeler, kenarlık	imlecin bulunduğu paragrafın tamamına; seçim gerekmez
Bölüm	kenar boşlukları, sayfa yönü, sütun sayısı, üstbilgi/altbilgi, sayfa numarası biçimi	imlecin bulunduğu bölümün bütün sayfalarına
Belge	tema, varsayılan stiller, belge özellikleri	bütün belgeye

Tablo 3.1.6: Yapı düzeyleri ve her düzeyde ayarlanan biçimler.

Örnek 3.1.5

Belgede iki paragraf var ve hiçbir şey seçili değil. İmleç 2. paragrafın ortasındayken Ortala ve Satır aralığı 1,5 komutları veriliyor. Ardından imleç 1. paragrafın sonuna götürülüyor, `Ctrl+B`'ye basılıp " Son." yazılıyor. Sonuç nedir?

1. **Ortala ve satır aralığı** paragraf biçimidir. Seçim gerekmez; imlecin bulunduğu 2. paragrafın tamamı ortalanır ve 1,5 satır aralıklı olur. 1. paragraf etkilenmez.
2. **Kalın** karakter biçimidir. Seçim yokken `Ctrl+B` yalnızca bundan sonra yazılacak karakterleri kalın yapar. " Son." kalın yazılır; 1. paragrafın eski metni değişmez.

Aynı tuşa basmanın etkisi, komutun hangi düzeye ait olduğuna göre değişir.

3.1.4 Metin düzenleme

Bir belgeyi yazmak, yazılanı düzeltmekten çok daha kısa sürer. Rapor yazan öğrenci zamanının büyük kısmını cümle taşımaya, sözcük değiştirmeye, gereksiz boşlukları temizlemeye ayırır. Bu kesimdeki araçlar bu işleri hızlandırır: klavyeyle hareket ve seçim, pano, geri alma, bul ve değiştir. Her aracın bir de tuzağı vardır. Tuzaklar da örneklerle birlikte gösterilir.

3.1.4.1 İmleç, hareket ve seçim

Tanım 3.1.7

İmleç (*cursor*), belge alanında yanıp sönen dikey çizgidir ve bir sonraki karakterin ekleneceği yeri gösterir. **Seçim** (*selection*) ise vurgulanmış metin parçasıdır. Biçimlendirme, silme, kesme ve kopyalama gibi komutların çoğu seçime uygulanır.

İmleci fareyle tıklayarak taşımak kısa metinde yeterlidir. Uzun belgede ise klavye çok daha hızlıdır. Ok tuşları imleci birer karakter ya da satır taşır. `Ctrl` tuşu bu adımı büyütür: sözcük, paragraf ya da belge ölçeğine çıkarır. `Shift` tuşu ise hareketi seçime çevirir: imleç nereye giderse, başladığı yerle vardığı yer arasındaki metin seçilir.

Kısayol	Etki
`Ctrl+Home`	imleç belgenin en başına gider
`Ctrl+End`	imleç belgenin en sonuna gider
`Ctrl+→` / `Ctrl+←`	imleç bir sonraki / önceki sözcüğün başına atlar
`Ctrl+Shift+→`	aynı atlamayı yapar ve geçtiği metni seçer
`Shift+↑` / `↓`	seçimi bir satır yukarı / aşağı genişletir
`Shift+End`	imleçten satır sonuna kadar seçer
`Ctrl+A`	bütün belgeyi seçer

Tablo 3.1.7: Sık kullanılan hareket ve seçim kısayolları.

Seçim için fare de kullanılır. Bir sözcüğe çift tıklamak sözcüğü, üç kez tıklamak bütün paragrafı seçer. Satırın solundaki boş kenar alanına bir kez tıklamak o satırı seçer. Seçimi kaldırmak için Shift'siz bir ok tuşuna basmak ya da metnin herhangi bir yerine tıklamak yeter.



Şekil 3.1.7: “Word belgesinde sözcük sözcük ilerlenir.” satırında, iki `Ctrl+→` ve iki `Ctrl+Shift+→` sonrası durum: “sözcük sözcük ” seçili, imleç “ilerlenir.” sözcüğünün başında.

Örnek 3.1.6

Belgede tek satır var: “Word belgesinde sözcük sözcük ilerlenir.” Satır 40 karakterdir. Karakterler 0'dan numaralanırsa sözcükler 0, 5, 16, 23 ve 30. konumlarda başlar. İmleç satırın ortasındayken aşağıdaki tuşlara sırayla basılıyor. Her adımdan sonra imleç nerededir, ne seçilidir? (Şekil 3.1.7 beşinci adımın sonucunu gösterir.)

Adım	Tuş	İmleç konumu	Seçim
1	`Ctrl+Home`	0 (“Word” başı)	yok
2	`Ctrl+→`	5 (“belgesinde” başı)	yok
3	`Ctrl+→`	16 (ilk “sözcük” başı)	yok
4	`Ctrl+Shift+→`	23	“sözcük ” (16–22)
5	`Ctrl+Shift+→`	30 (“ilerlenir.” başı)	“sözcük sözcük ” (16–29)
6	`Ctrl+End`	40 (belge sonu)	yok

Adım 4 ve 5'te imleç, adım 2 ve 3'teki gibi atlar; fark yalnızca geçilen metnin seçilmesidir. Seçim bir sözcüğün sonundaki boşluğu da içerir. Adım 5'ten sonra `Ctrl+B`'ye basılsaydı "sözcük sözcük" kalın olurdu. Adım 6'daki Shift'siz hareket ise seçimi kaldırır.

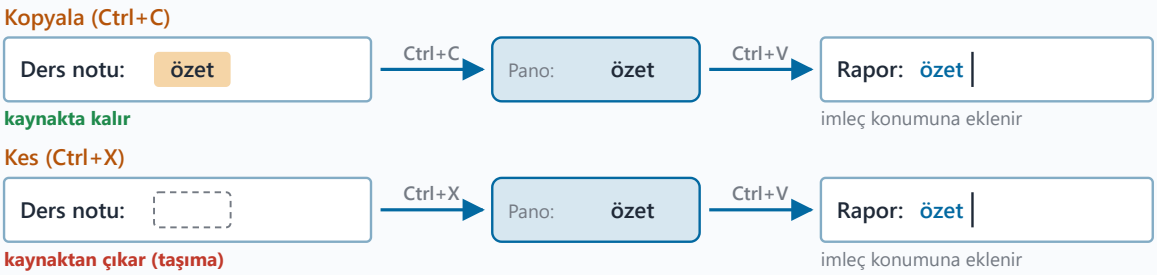
Sık yapılan hata

Seçim varken yanlışlıkla bir harf tuşuna basmak. Word seçili metnin yerine basılan karakteri yazar; bütün paragraf seçiliyse paragraf tek harfe dönüşür. Böyle bir durumda paniğe gerek yoktur: `Ctrl+Z` son işlemi geri alır ve metin geri gelir.

3.1.4.2 Kes, kopyala ve yapıştır

Tanım 3.1.8

Pano (*clipboard*), kesilen ya da kopyalanan içeriğin geçici olarak tutulduğu bellek alanıdır. **Kopyala** (`Ctrl+C`) seçili içeriğin bir kopyasını panoya alır ve kaynağı yerinde bırakır. **Kes** (`Ctrl+X`) seçili içeriği belgeden kaldırıp panoya alır. **Yapıştır** (`Ctrl+V`) panodaki içeriği imlecin bulunduğu yere ekler.



Şekil 3.1.8: Kopyala ile Kes'in farkı. Üstte Kopyala: "özet" sözcüğü ders notunda kalır, kopyası panoya gider ve `Ctrl+V` ile rapora eklenir. Altta Kes: sözcük ders notundan çıkar, yani taşınır.

Şekil 3.1.8 iki işlemin tek farkını gösterir: kaynak yerinde kalıyor mu? Kopyala çoğaltır, Kes taşır. İkisinde de yapıştırma aynı biçimde çalışır. Pano, yapıştırmadan sonra içeriğini korur; aynı içerik istenildiği kadar yeniden yapıştırılabilir. Pano yalnızca son kesilen ya da kopyalanan içeriği tutar. Yeni bir kopyalama eskisinin yerini alır. Office Panosu ise (Giriş → Pano grubunun sağ alt köşesindeki küçük ok) son 24 öğeyi saklayabilir ve bunlardan istenen birini yapıştırmaya izin verir.

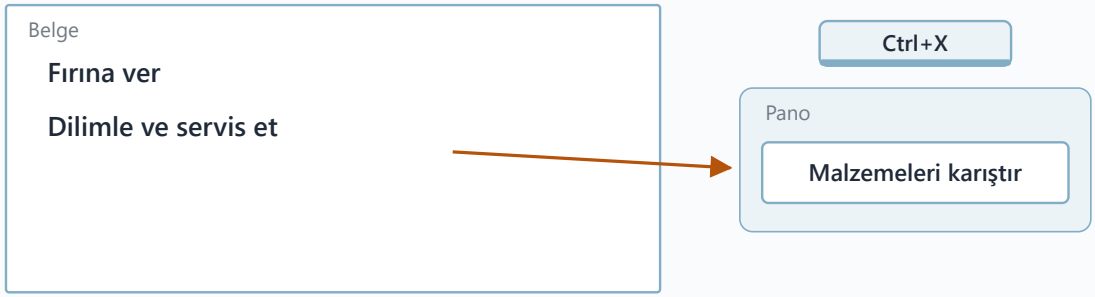
Bir metin başka bir kaynaktan, örneğin bir web sayfasından kopyalandığında rengi, yazı tipi ve boyutu da onunla gelir. Yapıştırırken bu biçimin ne olacağı **yapıştırma seçenekleriyle** belirlenir:

- **Kaynak Biçimlendirmesini Koru**: metin kaynaktaki görünüşüyle gelir. Düz `Ctrl+V` genellikle bunu yapar.

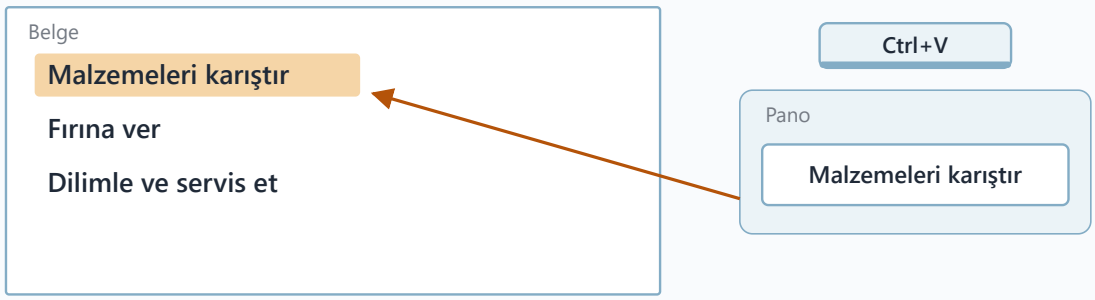
- **Biçimlendirmeyi Birleştir**: metin, yapıştırıldığı yerin biçimine uyum sağlar; kalın, italik gibi vurgular korunur.
- **Yalnızca Metni Koru**: bütün biçim atılır, yalnızca harfler gelir ve hedefin biçimini alır. Güncel Microsoft 365 sürümlerinde kısayolu `Ctrl+Shift+V`'dir; eski sürümlerde bu kısayol farklı olabilir. Her sürümde Giriş → Yapıştır düğmesinin altındaki oktan seçilebilir.

Biçimi metinden ayrı olarak kopyalamak da mümkündür. **Biçim Boyacısı** (*Format Painter*), Giriş → Pano grubundaki fırça düğmesidir. Önce biçimi alınacak metin seçilir ve fırçaya tıklanır. Ardından biçimin uygulanacağı metin seçilir. Fırçaya çift tıklanırsa araç açık kalır ve birden çok yere sırayla uygulanabilir; `Esc` aracı kapatır.

Kes ve yapıştırın gücü en iyi küçük bir düzeltmede görülür. Üç satırlık bir tarif düşünelim. Tarifin sırası yanlıştır: "Malzemeleri karıştır" adımı ikinci satırdadır ama en başta olmalıdır. Satırı silip yeniden yazmak yerine satır taşınır.



Şekil 3.1.9: `Ctrl+X` sonrası: "Malzemeleri karıştır" satırı belgeden kalktı ve panoya alındı. Belgede "Fırına ver" ile "Dilimle ve servis et" kaldı.



Şekil 3.1.10: `Ctrl+Home` ve `Ctrl+V` sonrası: satır belgenin başına eklendi. Pano içeriğini koruyor.

Örnek 3.1.7

Tarifteki satırı taşımanın adımları ve her adımda belgenin ve panonun durumu şöyledir (Şekil 3.1.9 ve Şekil 3.1.10):

Adım	İşlem	Belgedeki satırlar	Pano
0	başlangıç	Fırına ver / Malzemeleri karıştır / Dilimle ve servis et	boş
1	2. satır seçilir (satır başında `Shift+↓` ya da satır solunda tıklama)	aynı; 2. satır seçili	boş
2	`Ctrl+X`	Fırına ver / Dilimle ve servis et	Malzemeleri karıştır
3	`Ctrl+Home`	aynı; imleç belgenin başında	Malzemeleri karıştır
4	`Ctrl+V`	Malzemeleri karıştır / Fırına ver / Dilimle ve servis et	Malzemeleri karıştır

Adım 2'de Kes yerine Kopyala (`Ctrl+C`) kullanılsaydı kaynak satır yerinde kalırdı. Sonuçta "Malzemeleri karıştır" belgede iki kez görünürdü: başta ve üçüncü satırda. Satırla birlikte satır sonundaki paragraf işaretinin de seçilmesi gerekir; aksi hâlde yapıştırılan metin bir sonraki satırla birleşir.

3.1.4.3 Geri al, bul ve değiştir

Word, yapılan işlemleri sırayla bir listede tutar. **Geri Al** (`Ctrl+Z`) bu listedeki son işlemi geri çevirir. Arka arkaya basılırsa işlemler sondan başa doğru teker teker geri alınır. **Yinele** (`Ctrl+Y`) geri alınan son işlemi yeniden yapar. Geri alınacak bir şey yoksa son komutu bir kez daha uygular. Örneğin bir paragraf kalın yapıldıktan sonra başka bir paragraf seçilip `Ctrl+Y`'ye basılırsa o da kalın olur.

Tanım 3.1.9

Bul (`Ctrl+F`), belgede bir metnin geçtiği yerleri arar ve sonuçları gezinti bölmesinde listeler. **Bul ve Değiştir** (*Find and Replace*; `Ctrl+H`) iletişim kutusu, *Aranan* kutusuna yazılan metnin geçtiği yerleri *Yeni değer* kutusundaki metinle değiştirir. **Değiştir** düğmesi o anki eşleşmeyi, **Tümünü Değiştir** ise bütün eşleşmeleri tek seferde değiştirir.



Şekil 3.1.11: Bul ve Değiştir iletişim kutusu. Aranan "ve", yeni değer "ile". Tümünü Değiştir "kitap ve veri" metnini "kitap ile ileri" yapar, çünkü "veri" sözcüğünün içindeki "ve" de değişir. Önlem: Diğer >> düğmesiyle açılan seçeneklerden "Yalnızca tam sözcükleri bul".

Tümünü Değiştir güçlü ama tehlikeli bir komuttur. Arama varsayılan olarak *karakter dizisi* arar; sözcük sınırlarına bakmaz. Şekil 3.1.11 bunun sonucunu gösterir: “ve” aranınca “veri” sözcüğünün ilk iki harfi de eşleşir. Bu tuzaktan korunmanın iki yolu vardır. Birincisi, **Diğer >>** düğmesiyle arama seçeneklerini açıp **Yalnızca tam sözcükleri bul** kutusunu işaretlemektir. İkincisi, Tümünü Değiştir yerine önce **Sonrakini Bul** ile eşleşmeleri tek tek görmek ve yalnızca doğru olanlarda Değiştir'e basmaktır.

Örnek 3.1.8

Belgede tek cümle var: “Deney ve ölçüm verileri evde ve okulda toplandı.” Aranan: `ve` , Yeni değer: `ile` . Tümünü Değiştir'in sonucu iki ayarda nasıl olur?

1. **Eşleşmeleri saymak.** “ve” dizisi cümlede soldan sağa üç yerde geçer: ilk bağlaç “ve”, “verileri” sözcüğünün başı ve ikinci bağlaç “ve”. “Deney” ve “evde” sözcüklerinde v ile e yan yana gelmez, eşleşme yoktur.
2. **Varsayılan ayar.** Üç eşleşme de değişir (3 değiştirme): “Deney ile ölçüm **ileri**leri evde ile okulda toplandı.” “verileri” sözcüğü bozulmuştur.
3. **Yalnızca tam sözcükleri bul.** Önünde ve arkasında harf olmayan eşleşmeler değişir; “verileri” içindeki eşleşme atlanır (2 değiştirme): “Deney ile ölçüm verileri evde ile okulda toplandı.”

Hata fark edilirse tek bir `Ctrl+Z` bütün değiştirmeleri birden geri alır, çünkü Tümünü Değiştir tek bir işlem sayılır.

Bul ve Değiştir yalnızca harfleri değil, belgedeki özel öğeleri de arayabilir: paragraf işaretleri, sekmeler, sayfa sonları ve biçimler. Bunun için gizli işaretleri tanımak gerekir.

Sık yapılan hata

Kısa bir sözcüğü Tümünü Değiştir ile önizlemeden değiştirmek. “not” → “kayıt” değişikliği “notlar”ı “kayıtlar”a, “notu”nu “kayıtu”ya çevirir. Kısa ve başka sözcüklerin içinde geçebilecek dizilerde ya “Yalnızca tam sözcükleri bul” işaretlenir ya da eşleşmeler Sonrakini Bul ile tek tek denetlenir.

3.1.4.4 Gizli biçimlendirme işaretleri

Word belgesinde ekranda görünmeyen ama belgenin yapısını taşıyan karakterler vardır. Her Enter bir paragraf işareti, her boşluk bir boşluk karakteri, her Tab bir sekme karakteri ekler. Bu karakterler yazdırılmaz. Ama belgedeki hataların çoğu onlardan kaynaklanır: üst üste boş paragraflar, çift boşluklar, boşlukla yapılmış hizalamalar.

Tanım 3.1.10

Biçimlendirme işaretleri (*formatting marks*), belgedeki yazdırılmayan karakterlerin ekranda gösterilen simgeleridir. Giriş → Paragraf grubundaki ¶ düğmesiyle (Tümünü Göster/Gizle) açılıp kapanır. Açıkken her paragraf sonu ¶, her boşluk orta nokta (·), her sekme ok (→) olarak görünür.

İşaret	Anlamı	Nasıl oluşur?	Bul ve Değiştir kodu
¶	paragraf sonu	`Enter`	`^p`
↲	satır sonu; aynı paragraf sürer	`Shift+Enter`	`^l`
·	boşluk	boşluk çubuğu	(boşluk)
→	sekme	`Tab`	`^t`
°	bölünmez boşluk: iki yanındaki sözcükler ayrı satırlara düşmez	`Ctrl+Shift+Boşluk`	`^s`
····Sayfa Sonu····	elle eklenen sayfa sonu	`Ctrl+Enter`	`^m`

Tablo 3.1.8: Gizli biçimlendirme işaretleri, nasıl oluştukları ve Bul ve Değiştir'deki kodları.

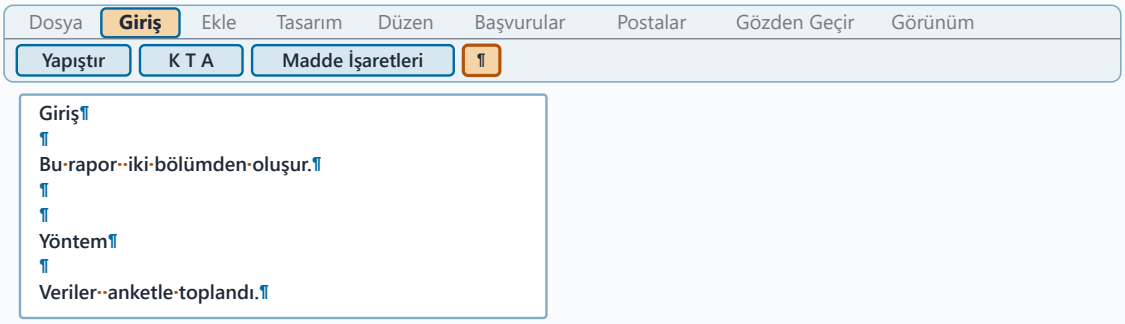
Tablo 3.1.8'deki işaretlerden en önemlisi ¶'dir. Paragrafın hizalama, girinti ve aralık gibi bütün biçim bilgisi bu işarette saklanır. Bir paragrafın sonundaki ¶ silinirse paragraf bir sonrakiyle birleşir. Birleşen paragraf, ikinci paragrafın ¶ işaretindeki biçimi alır. Bu yüzden ¶ silindiğinde metnin hizalaması birden değişebilir.

Satır sonu (↲) ile paragraf sonu (¶) arasındaki fark özellikle başlıklarda önemlidir. Uzun bir başlık iki satıra bölünmek istenirse `Shift+Enter` kullanılır. Böylece başlık tek paragraf olarak kalır ve içindekilerde tek satır görünür. `Enter` kullanılsaydı başlık iki ayrı paragrafa bölünür, içindekilerde de iki ayrı başlık olarak listelenirdi.

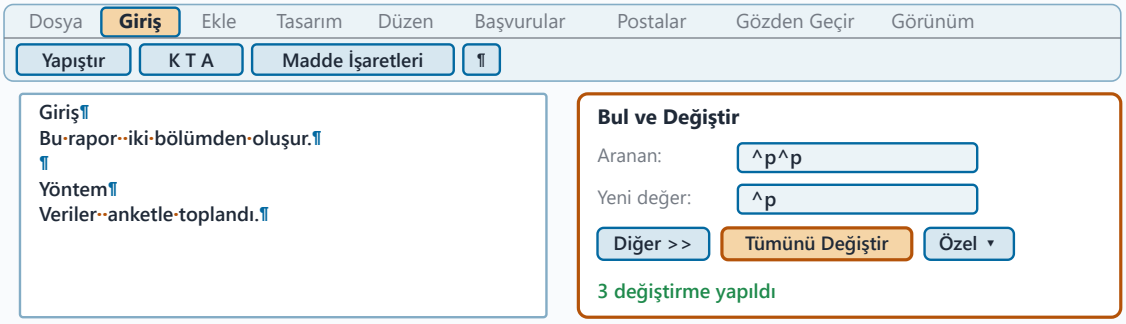
Bölünmez boşluk küçük ama işe yarar bir araçtır. “10 kg”, “Şekil 3” ya da “Prof. Dr.” gibi ifadelerin satır sonunda ikiye bölünmesi okumayı zorlaştırır. Aradaki boşluk `Ctrl+Shift+Boşluk` ile yazılırsa ifade her zaman aynı satırda kalır.

3.1.4.5 Bul ve Değiştir ile belge temizliği

Gizli işaretler ile Bul ve Değiştir birleşince, elle saatler sürecek temizlik birkaç tıklamaya iner. Örnek olarak kısa bir rapor ele alalım. Raporda paragraflar arasındaki boşluk boş paragraflarla verilmiştir ve iki yerde çift boşluk vardır.



Şekil 3.1.12: ¶ düğmesi açıkken belge. “Giriş” ile “Bu rapor…” arasında bir, “oluşur.” ile “Yöntem” arasında iki, “Yöntem” ile “Veriler…” arasında bir boş paragraf var. “rapor··iki” ve “Veriler··anketle” arasında çift boşluk görülüyor.



Şekil 3.1.13: Aranan `^p^p`, yeni değer `^p` ile ilk Tümünü Değiştir sonrası: 3 değiştirme yapıldı. “oluşur.” paragrafından sonra hâlâ bir boş paragraf duruyor.

Boş paragraf, art arda gelen iki paragraf işareti demektir: önceki paragrafın ¶'si ve boş paragrafın kendi ¶'si. Bu yüzden boş paragrafları silmek için Aranan kutusuna `^p^p`, Yeni değer kutusuna `^p` yazılır. Kodlar elle yazılabileceği gibi, Diğer >> → Özel menüsünden “Paragraf İşareti” seçilerek de eklenebilir.

Örnek 3.1.9

Belge, gizli işaretleriyle şöyle bir karakter dizisidir (¶ paragraf sonu, · boşluk):

```
Giriş¶ ¶ Bu·rapor··iki·bölümden·oluşur.¶ ¶ ¶ Yöntem¶ ¶
Veriler··anketle·toplandı.¶
```

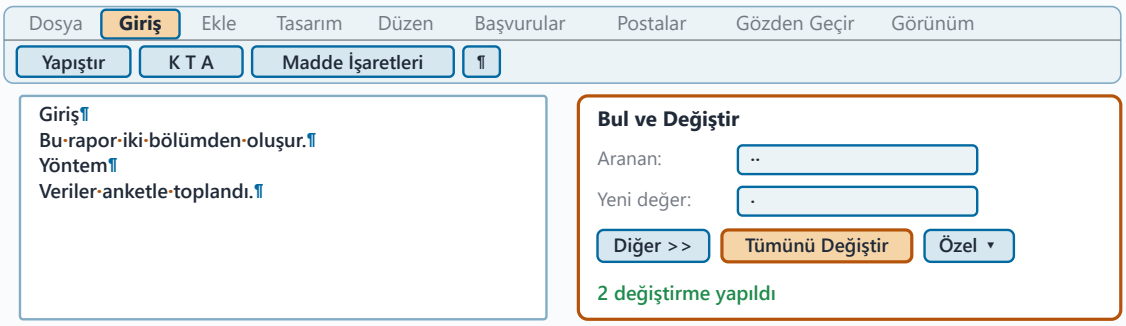
Temizlik dört komutla yapılır. Her komuttan sonra belgenin durumu şöyledir (Şekil 3.1.12 ve Şekil 3.1.13):

Komut	Aranan → Yeni değer	Değiştirme	Belge
1. Tümünü Değiştir	`^p^p` → `^p`	3	Giriş¶ Bu…oluşur.¶ ¶ Yöntem¶ Veriler…toplandı.¶

Komut	Aranan → Yeni değer	Değiştirme	Belge
2. Tümünü Değiştir	`^p^p` → `^p`	1	Giriş¶ Bu…oluşur.¶ Yöntem¶ Veriler…toplandı.¶
3. Tümünü Değiştir	`^p^p` → `^p`	0	değişmedi
4. Tümünü Değiştir	iki boşluk → bir boşluk	2	Bu·rapor·iki…, Veriler·anketle…

Neden ilk komut hepsini temizlemedi? Arama soldan sağa ilerler ve değiştirdiği yere geri dönmez. "oluşur." sonrasında art arda üç ¶ vardır. Word ilk ikisini bulur ve tek ¶'ye indirir, sonra aramayı üçüncü ¶'den sürdürür. Yeni yazılan ¶ ile üçüncü ¶ artık yan yana durur ve bir çift oluşturur. Ama arama geri dönmediği için bu çifti görmez; üçüncü ¶'nin arkasında ise "Yöntem" gelir. Böylece bu noktada iki ¶ kalır, yani bir boş paragraf. İkinci komut bu çifti yakalar. Üçüncü komutun 0 sonuç vermesi işin bittiğini gösterir.

Genel kural şudur: art arda *k* tane boş paragraf varsa, her geçiş bunların yaklaşık yarısını siler. Bu yüzden komut, sonuç 0 olana kadar tekrarlanır. Temizlikten sonra paragraflar arasındaki boşluk boş Enter'larla değil, paragrafın **Sonra** aralığıyla verilmelidir (3.1.5.3).



Şekil 3.1.14: Son durum: boş paragraf ve çift boşluk kalmadı (iki boşluk → bir boşluk, 2 değiştirme).

3.1.4.6 Kısayollar, yer işareti ve köprü

Sık yapılan işlerin klavye kısayolları zamanla el alışkanlığına dönüşür. Tablo 3.1.9 sık kullanılan birkaç önemli kısayolu bir arada toplar.

Kısayol	İşlem
`Ctrl+Enter`	sayfa sonu ekler
`Ctrl+Shift+V`	yalnızca metni yapıştırır (güncel Microsoft 365 sürümlerinde)
`Ctrl+B` / `I` / `U`	kalın / italik / altı çizili
`Ctrl+K`	köprü ekler
`Ctrl+S` / `F12`	kaydet / farklı kaydet

Kısayol	İşlem
F9	seçili alanları günceller

Tablo 3.1.9: Sık kullanılan kısayollar.

Tanım 3.1.11

Yer işareti (*bookmark*), belgede bir noktaya ya da metin parçasına verilen addır; Ekle → Yer İşareti ile eklenir ve adında boşluk bulunamaz. **Köprü** (*hyperlink*; `Ctrl+K`) ise tıklanınca bir hedefe götüren bağlantıdır. Hedef bir web adresi, başka bir dosya ya da belgedeki bir başlık veya yer işareti olabilir.



Şekil 3.1.15: Belge içi köprü. 2. sayfadaki "Yöntem bölümüne" bağlantısı, 5. sayfadaki `Bolum_Yontem` adlı yer işaretine bağlıdır; `Ctrl+tıklama` ile oraya atlanır.

Şekil 3.1.15 belge içi köprünün işleyişini gösterir. Word'de köprüyü izlemek için varsayılan olarak `Ctrl`'ye basılı tutup tıklamak gerekir. Düz tıklama, bağlantı metnini düzenlemek için imleci oraya koyar. Yer işareti, başlık olmayan bir noktayı, örneğin bir tablonun ortasını, hedef yapmak için kullanılır. Hedef bir başlıksa yer işaretine gerek yoktur; köprü doğrudan başlığa bağlanabilir.

Görünüm sekmesini de burada anmak gerekir. Okuma Modu, Gezinti Bölmesi, Cetvel ve Yakınlaştırma buradan açılır. Gezinti bölmesi uzun belgede başlıklar arasında atlamayı sağlar. Cetvel ise paragraf girintilerini görmeyi ve sürükleyerek ayarlamayı kolaylaştırır.

3.1.5 Biçimlendirme

Biçimlendirme, metnin görünüşünü belirleyen ayarların tümüdür. 3.1.3.4'teki düzeyler burada da geçerlidir: bazı ayarlar karaktere, bazıları paragrafa uygulanır. Kısa bir belgede biçimi tek tek vermek sorun çıkarmaz. Uzun belgede ise aynı ayarları onlarca kez tekrarlamak hem zaman alır hem de tutarsızlık doğurur. Bu kesim karakter ve paragraf biçimlerini tanıtır, sonra bu tekrarı ortadan kaldıran aracı, stili anlatır.

3.1.5.1 Karakter biçimlendirme

Tanım 3.1.12

Karakter biçimlendirme, tek tek karakterlerin görünüşünü belirleyen ayarlardır: **yazı tipi** (*font*; karakterlerin görsel ailesi, örneğin Calibri ya da Times New Roman), **boyut** (punto cinsinden), vurgular (kalın, italik, altı çizili, renk), büyük/küçük harf dönüşümü, **üst simge** ve **alt simge**. Bu ayarlar Giriş → Yazı Tipi grubundadır ve seçili metne uygulanır.

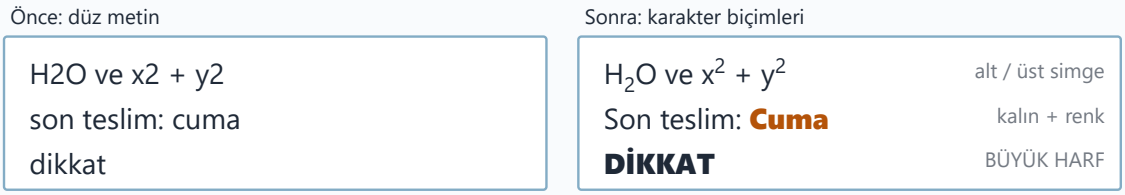
Yazı boyutu **punto** (pt) ile ölçülür. Punto, matbaacılıktan gelen bir uzunluk birimidir: 1 inç 72 punto, 1 inç de 2,54 cm'dir. Buradan bir puntonun milimetre karşılığı bulunur.

Örnek 3.1.10

12 punto ne kadar büyüklüktür?

1. 1 pt = 1/72 inç = 25,4 mm ÷ 72 ≈ 0,353 mm.
2. 12 pt = 12 × 0,353 mm ≈ 4,23 mm.
3. Karşılaştırma: 72 pt tam 1 inçtir, yani 2,54 cm.

Punto değeri harfin kendi yüksekliği değil, yazı tipinin tasarım kutusunun yüksekliğidir. Bu kutu, yukarı en çok uzanan harfin üst ucundan (b, l) aşağı en çok inen harfin alt ucuna (g, p) kadar uzanır. Bu yüzden 12 punto bir "a" harfi 4,23 mm'den kısadır. Aynı puntodaki iki farklı yazı tipi de farklı büyüklükte görünebilir.



Şekil 3.1.16: Karakter biçimleri öncesi ve sonrası. "H2O" alt simgeyle H_2O, "x2 + y2" üst simgeyle $x^2 + y^2$ olur; "Son teslim: Cuma" kalın ve renkli yapılır; "dikkat" büyük harfe dönüştürülür.

Şekil 3.1.16 mühendislik metinlerinde sık geçen biçimleri gösterir. **Alt simge** kimyasal formüllerde (H_2O, CO_2), **üst simge** üslerde ve birimlerde (x^2, m^3, 10^{-6}) kullanılır. Bu biçimler yalnızca görünüş değildir: "m3" ile "m^3" farklı anlamlar taşır. **Büyük/Küçük Harf Değiştir** komutu Giriş → Yazı Tipi grubundaki Aa düğmesidir. Seçili metni yeniden yazmadan BÜYÜK HARF, küçük harf ya da Her Sözcüğün İlk Harfi Büyük biçimine çevirir.

Biçimi Temizle komutu (Giriş → Yazı Tipi grubundaki silgili A düğmesi) seçili metindeki elle verilmiş bütün karakter biçimlerini kaldırır ve metni stilinin varsayılan görünüşüne döndürür. Başka kaynaktan yapıştırılıp tuhaf görünen bir metni düzeltmenin en hızlı yolu budur.

Sık yapılan hata

Uzun bir belgede başlıkları tek tek seçip yazı tipini, boyutunu ve rengini elle değiştirmek. Yirmi başlıktan biri unutulursa ya da bir tanesine 15 yerine 16 punto verilirse belge tutarsız olur. Görünüş sonradan değişecekse yirmi başlığın hepsi yeniden düzeltilir. Tekrarlanan biçimler için stil kullanılır (3.1.5.5).

3.1.5.2 Paragraf biçimlendirme

Tanım 3.1.13

Paragraf, Word'de iki paragraf işareti (¶) arasındaki metindir. **Paragraf biçimlendirme**, bir paragrafın bütününe uygulanan ayarlardır: **hizalama** (*alignment*), **girinti** (*indent*), **satır aralığı**, **paragraf aralığı**, sekme durakları, kenarlık ve gölgelendirme. Bu ayarlar Giriş → Paragraf grubundadır ve paragrafın ¶ işaretinde saklanır.

Paragraf biçimi için seçim yapmak gerekmez; imleç paragrafın herhangi bir yerindeyse komut bütün paragrafa uygulanır. Birden çok paragraf biçimlendirilecekse hepsinden en az birer parça seçilir.



Şekil 3.1.17: Dört hizalama türü: sola, ortala, sağa ve iki yana yaslama. İki yana yaslamada satırların iki ucu da kenar boşluğuna dayanır; son satır sola dayalı kalır.

Şekil 3.1.17 hizalama türlerini karşılaştırır. Türkçe metinlerde gövde metni genellikle **sola** ya da **iki yana yaslı** yazılır. Başlıklar ve kapak sayfası **ortalanır**, dilekçelerdeki tarih ise **sağa** dayanır. İki yana yaslamada Word, satırı kenara yetiştirmek için sözcük aralarını açar. Dar sütunlarda ya da uzun sözcüklerde bu aralıklar göze batabilir; bu durumda heceleme (3.1.6.1) açılabilir.

Sekmeler metni sayfada belirli yatay konumlara dayamaya yarar. Örneğin bir dilekçede "Ad Soyad:" yazısından sonraki değerlerin hepsi aynı hizadan başlasın isteniyorsa Tab tuşu ve bir sekme durağı kullanılır. **Kenarlık ve gölgelendirme** ise bir paragrafı çerçeve ya da arka plan rengiyle çevresinden ayırır; bir uyarı notunu öne çıkarmak için uygundur.

Sık yapılan hata

Metni boşluk tuşuyla hizalamak. Ekranda doğru görünen “Ad Soyad: Ayşe” satırı, yazı tipi ya da boyut değişince kayar, çünkü boşluk karakterlerinin genişliği yazı tipine bağlıdır. Hizalama için sekme, girinti ya da kenarlıksız bir tablo kullanılır.

3.1.5.3 Girinti ve aralık ölçüleri

Girinti, paragrafın kenar boşluğuna göre ne kadar içeride başladığını belirler. Aralık ise satırların ve paragrafların arasında bırakılan dikey boşluktur. Bu dört ölçü, bir belgenin “okunaklı” ya da “sıkışık” görünmesinde en büyük payı taşır.



Şekil 3.1.18: Paragraf ölçüleri. Cetvelde üç işaretçi vardır: üstteki üçgen ilk satır girintisini, alttaki üçgen sol (asılı) girintiyi, sağdaki üçgen sağ girintiyi belirler. 1. paragrafta ilk satır girintisi, 2. paragrafta kaynakçadaki gibi asılı girinti var; satır aralığı (ör. 1,5 satır) ve paragraf sonrası aralık (ör. 12 nk) de gösteriliyor.

Tanım 3.1.14

Sol ve sağ girinti, paragrafın bütün satırlarının sol ve sağ kenar boşluğundan ne kadar içeride duracağıdır. **İlk satır girintisi** (*first-line indent*) yalnızca paragrafın ilk satırını içeri alır. **Asılı girinti** (*hanging indent*) bunun tersidir: ilk satır solda kalır, sonraki satırlar içeri girer. **Satır aralığı**, bir paragrafın satırları arasındaki uzaklıktır. **Paragraf aralığı** ise paragraftan önce (Önce) ve sonra (Sonra) bırakılan boşluktur; Word'de nokta (nk) cinsinden verilir.

Şekil 3.1.18 bu ölçüleri cetvel üzerinde gösterir. Cetveldeki işaretçiler sürüklenerek ayarlanabilir, ama kesin değer için Giriş → Paragraf grubunun sağ alt köşesindeki ok tıklanır. Açılan Paragraf iletişim kutusunda girintiler santimetre, aralıklar nokta cinsinden yazılır. Word'deki “nk” birimi puntoyla aynıdır: 1 nk = 1 pt ≈ 0,353 mm.

İlk satır girintisi Türkçe metinlerde paragraf başını belirtmenin geleneksel yoludur; sık kullanılan değer 1,25 cm'dir. Asılı girinti ise özellikle kaynakçada kullanılır. Her kaynağın ilk satırı solda başlar, devam satırları içeride kalır. Böylece okuyucu kaynakların başlangıcını kolayca ayırt eder.

Örnek 3.1.11

12 punto metinden oluşan bir raporda paragraflar arasında boş satır yerine “Sonra: 12 nk” aralığı kullanılacak. Bu boşluk kaç milimetredir? Satır aralığı “Tam olarak 18 nk” yapılırsa bir satır kaç milimetre yer kaplar?

1. **Paragraf sonrası aralık.** 12 nk × 0,353 mm ≈ 4,23 mm. Bu, 12 puntoluk bir boş satırın yazı tipi kutusuyla aynı yüksekliktir; görünüş boş satıra yakındır, ama belgede fazladan ¶ yoktur.
2. **Satır yüksekliği.** “Tam olarak 18 nk” her satıra 18 × 25,4 ÷ 72 = 6,35 mm ayırır. Bu değer 12 puntonun 1,5 katıdır.

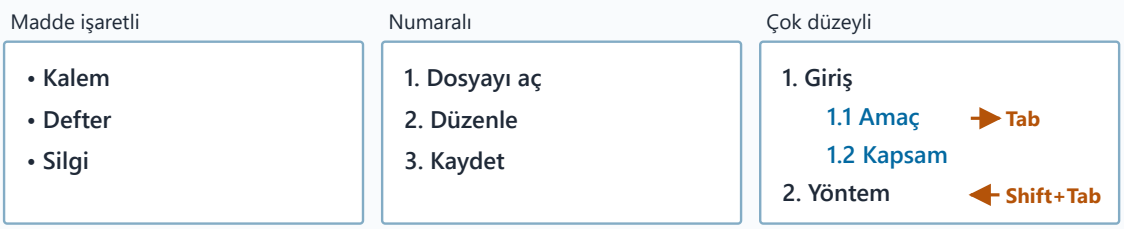
Satır aralığı listesindeki “1,5 satır” seçeneği ise yazı tipine göre hesaplanır. Word, yazı tipinin kendi önerdiği tek satır yüksekliğini 1,5 ile çarpar; bu yükseklik çoğu yazı tipinde puntodan biraz büyüktür. Bu yüzden “1,5 satır” ile “Tam olarak 18 nk” birbirine yakın ama genellikle eşit değildir.

Sık yapılan hata

Paragraflar arasında boşluk bırakmak için Enter'a iki kez basmak. Boş paragraflar sayfa sonuna denk gelince yeni sayfanın başında boşluk bırakır, belge düzeni bozulur ve 3.1.4.5'teki gibi temizlik gerekir. Paragraf aralığı ise yalnızca paragraflar arasında görünür ve paragrafla birlikte taşınır.

3.1.5.4 Listeler

Raporlarda birçok bilgi liste hâlinde verilir: deney malzemeleri, işlem adımları, bölüm başlıkları. Word üç liste türü sunar ve hepsi Giriş → Paragraf grubundadır.



Şekil 3.1.19: Liste türleri. Madde işaretli liste (Kalem, Defter, Silgi), numaralı liste (Dosyayı aç, Düzenle, Kaydet) ve çok düzeyli liste (1. Giriş, 1.1 Amaç, 1.2 Kapsam, 2. Yöntem). `Tab` bir maddeyi bir düzey içeri, `Shift+Tab` bir düzey dışarı alır.

- **Madde işaretli liste** (*bulleted list*), sırası önemli olmayan öğeler içindir: deney malzemeleri gibi.
- **Numaralı liste** (*numbered list*), sırası önemli olan öğeler içindir: işlem adımları, öncelik sırası.
- **Çok düzeyli liste** (*multilevel list*), ana maddeler ve alt maddelerden oluşan bir hiyerarşi kurar: 1, 1.1, 1.1.1 gibi.

Şekil 3.1.19'daki düzey değiştirme kuralı listelerin en kullanışlı özelliğidir. Bir liste maddesinin başındayken `Tab`'a basılırsa madde bir alt düzeye iner; numarası da "3." yerine "a." ya da "2.1" gibi alt düzey numarasına dönüşür. `Shift+Tab` maddeyi bir üst düzeye çıkarır. Listenin sonunda boş bir maddede Enter'a basmak listeyi bitirir.

Örnek 3.1.12

Numaralı bir listede üç madde var: 1. Giriş, 2. Yöntem, 3. Sonuç. "Yöntem"in sonunda Enter'a basılıp yeni satıra "Veri toplama" yazılıyor ve sırayla aşağıdaki işlemler yapılıyor:

İşlem	Liste
Enter, "Veri toplama" yazılır	1. Giriş · 2. Yöntem · 3. Veri toplama · 4. Sonuç
"Veri toplama" başında `Tab`	1. Giriş · 2. Yöntem · a. Veri toplama · 3. Sonuç
aynı satırda `Shift+Tab`	1. Giriş · 2. Yöntem · 3. Veri toplama · 4. Sonuç

Her adımda "Sonuç" maddesinin numarası kendiliğinden güncellenir. Numaralar elle yazılmış olsaydı her ekleme ve düzey değişikliğinde bütün numaralar tek tek düzeltilirdi. Alt düzeyin "a." ya da "2.1" biçiminde görünmesi seçilen liste biçimine bağlıdır.

Sık yapılan hata

Liste işaretlerini elle yazmak: satır başına "1." ya da "-" yazıp ardından boşlukla devam etmek. Word bazen bunu otomatik listeye çevirir, bazen çevirmez. Elle yazılmış bir listede girintiler tutarsız kalır ve araya madde eklenince numaralar kayar. Liste, şeritteki liste düğmeleriyle başlatılır.

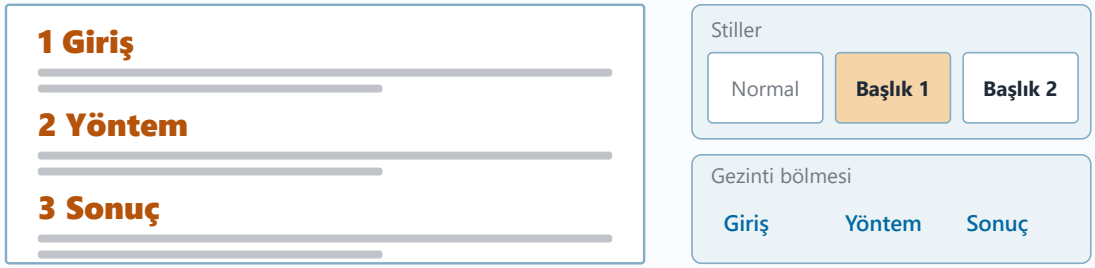
3.1.5.5 Stiller ve temalar

Buraya kadar anlatılan karakter ve paragraf biçimleri tek tek verilebilir. Ama bir raporda aynı görünüş defalarca tekrarlanır: bütün ana başlıklar aynı, bütün gövde paragrafları aynı biçimdedir. Bu tekrarlanan biçimleri bir ad altında toplayan araç stildir.

Tanım 3.1.15

Stil (*style*), adı olan bir biçim paketidir: yazı tipi, boyut, renk, hizalama, girinti ve aralık gibi ayarları tek bir ad altında birlikte tanımlar. Word'ün hazır stilleri arasında **Normal** (gövde metni), **Başlık 1** (ana bölüm başlığı) ve **Başlık 2** (alt bölüm başlığı) vardır. Stiller Giriş → Stiller galerisinden uygulanır. **Tema** (*theme*) ise belgenin genel renk, yazı tipi ve efekt sistemidir; Tasarım sekmesinden seçilir.

Stilin gücü tek bir tanımdan gelir. Bir başlığa Başlık 1 uygulanınca o başlık stilin tanımındaki görünüşü alır. Daha sonra stilin tanımı değiştirilirse, bu stili kullanan bütün başlıklar birlikte güncellenir. Yirmi başlığı tek tek düzeltmek yerine tek bir tanım değiştirilir.



Şekil 3.1.20: Başlık 1 stilini kullanan üç başlık (1 Giriş, 2 Yöntem, 3 Sonuç). Başlık stilleri belge hiyerarşisini de kurar: aynı başlıklar gezinti bölmesinde listelenir.

Başlık stillerinin ikinci görevi, görünüşten daha önemlidir: belgenin yapısını kurarlar (Şekil 3.1.20). Word, Başlık 1, Başlık 2 ve Başlık 3 stillerindeki paragrafları "başlık" olarak tanır. Gezinti bölmesi, içindekiler tablosu, Anahat görünümü ve PDF'teki yer imleri bu bilgiden üretilir. Yalnızca büyük ve kalın yazılmış bir satır ise Word için sıradan bir paragraftır, ne kadar başlığa benzese de.

Tema ile stil arasındaki ilişki katmanlıdır. Tema belgenin renk paletini ve iki temel yazı tipini (başlık ve gövde için) belirler. Stiller de bu paleti ve yazı tiplerini kullanır. Tasarım sekmesinden başka bir tema seçilince stiller tanımlarını korur ama renkleri ve yazı tiplerini yeni temadan alır. Örneğin Başlık 1 "tema rengi 1" ile tanımlıysa, tema değişince bütün Başlık 1'ler yeni temanın birinci rengine geçer. Bir stil değişikliği ise yalnızca o stili kullanan öğeleri etkiler.

Sık yapılan hata

Başlığı Normal stilde bırakıp yalnızca kalın ve büyük yazmak. Görünüş doğrudur ama gezinti bölmesi boş kalır ve otomatik içindekiler bu başlığı görmez. Başlıklar her zaman başlık stilleriyle yazılır; görünüş beğenilmezse stilin tanımı değiştirilir.

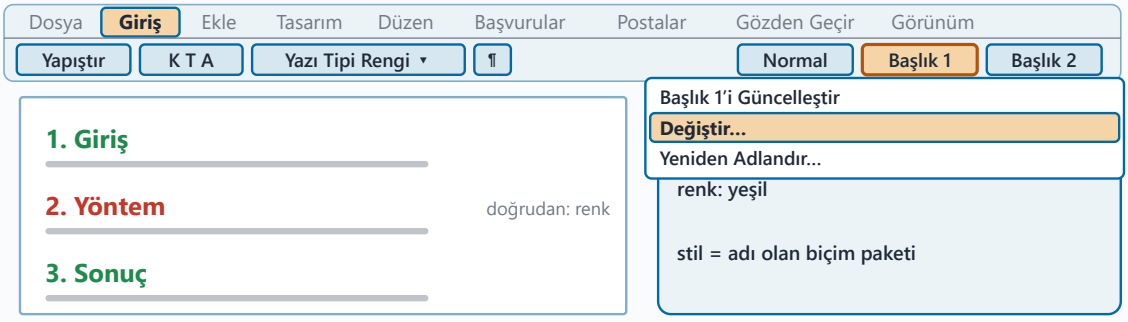
3.1.5.6 Stil tanımı ve doğrudan biçimlendirme

Stil uygulanmış bir metne şeritten ayrıca biçim verilebilir. Örneğin Başlık 1 stilindeki bir başlık şeritteki renk düğmesiyle kırmızı yapılabilir. Bu durumda başlığın görünüşü iki kaynaktan gelir: stil tanımından ve elle verilen biçimden. Hangisinin geçerli olacağını bilmek, stillerle çalışırken yaşanan şaşkınlıkların çoğunu önler.

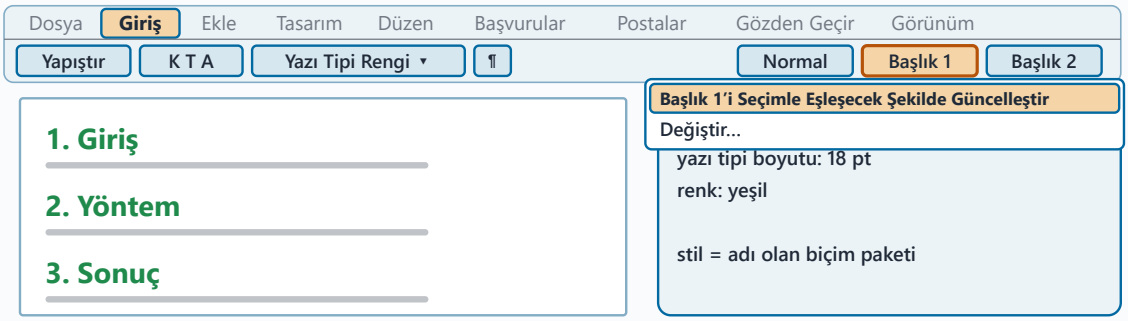
Tanım 3.1.16

Doğrudan biçimlendirme (*direct formatting*), bir metne stil yoluyla değil, şeritten ya da kısayolla tek tek verilen biçimdir. Doğrudan biçim stilin önüne geçer: stilin tanımı değişse bile, elle verilmiş özellik o metinde korunur. `Ctrl+Boşluk` seçili metindeki elle verilmiş karakter biçimlerini kaldırır ve metni yeniden stilin tanımına bağlar.

Stil tanımı iki yolla değiştirilir. Stiller galerisinde stilin adına sağ tıklanınca bir menü açılır. **Değiştir...** seçeneği stilin özelliklerini bir iletişim kutusunda düzenlemeyi sağlar. **Başlık 1'i Seçimle Eşleşecek Şekilde Güncelleştir** ise tanımı, o anda seçili metnin görünüşünden alır. İkinci yol, bir başlığı önce elle denemek ve beğenilen görünüşü bütün başlıklara yaymak için pratiktir.



Şekil 3.1.21: Değiştir... ile Başlık 1'in tanımı 16 pt ve yeşil yapıldı. Üç başlığın da boyutu değişti; ama daha önce elle kırmızı yapılan 2. başlık ("Yöntem") kırmızı kaldı.



Şekil 3.1.22: 3. başlık elle 18 pt yapıldıktan sonra "Başlık 1'i Seçimle Eşleşecek Şekilde Güncelleştir" seçildi. Tanım 3. başlıktan alındı: bütün Başlık 1'ler 18 pt oldu.

Örnek 3.1.13

Üç başlık (1. Giriş, 2. Yöntem, 3. Sonuç) Başlık 1 stilinde. Her adımdan sonra stilin tanımı ve üç başlığın görünüşü şöyledir (Şekil 3.1.21 ve Şekil 3.1.22):

Adım	İşlem	Başlık 1 tanımı	Giriş	Yöntem	Sonuç
1	üç başlığa Başlık 1 uygulanır	14 pt, mavi	14, mavi	14, mavi	14, mavi
2	Yöntem şeritten kırmızı yapılır (doğrudan)	14 pt, mavi	14, mavi	14, **kırmızı**	14, mavi
3	Değiştir... ile tanım 16 pt, yeşil yapılır	16 pt, yeşil	16, yeşil	16, **kırmızı**	16, yeşil
4	Yöntem seçilip `Ctrl+Boşluk`	16 pt, yeşil	16, yeşil	16, yeşil	16, yeşil
5	Sonuç elle 18 pt yapılır (doğrudan)	16 pt, yeşil	16, yeşil	16, yeşil	**18**, yeşil
6	Sonuç seçiliyken Seçimle Eşleşecek Şekilde Güncelleştir	18 pt, yeşil	18, yeşil	18, yeşil	18, yeşil

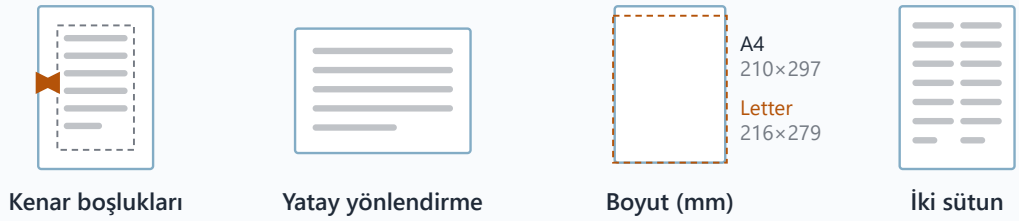
Adım 3 kuralın kendisidir: tanım değişince bütün başlıkların boyutu değişti, çünkü boyut hiçbir başlığa elle verilmemişti. Yöntem'in rengi ise elle verildiği için stilin yeni rengini almadı. Adım 4'te elle verilen renk silinince başlık yeniden stile bağlandı. Adım 6'da tanım bir başlıktan alındı ve yeni tanım üç başlığa birden yayıldı.

Bu örnek, stillerle çalışmanın temel alışkanlığını gösterir: görünüş elle değil, stilin tanımı üzerinden değiştirilir. Elle verilen biçimler zamanla belgede “gizli istisnalar” biriktirir. Görünüşü değişmeyen bir başlıkla karşılaşıldığında ilk bakılacak yer, o başlıkta unutulmuş bir doğrudan biçimdir. Word Stilleri ve İçindekiler simülatörü bu etkileşimi denemeye olanak verir.

3.1.6 Sayfa düzeni

Karakter ve paragraf biçimleri metnin görünüşünü belirler. Sayfa düzeni ise metnin üzerinde durduğu kâğıdı tanımlar: kâğıdın boyutu, yönü, kenar boşlukları, sütun sayısı, üstbilgi ve altbilgi. Bu ayarların hepsi **bölüm** düzeyindedir. Bir belge tek bölümden oluşuyorsa bütün sayfalar aynı düzene sahiptir. Farklı sayfalar için farklı düzen gerektiğinde belge bölüm sonlarıyla parçalara ayrılır.

3.1.6.1 Sayfa ayarları



Şekil 3.1.23: Sayfa düzeni seçenekleri: kenar boşlukları, yatay yönlendirme, A4 (210 × 297 mm) ile Letter (216 × 279 mm) boyut karşılaştırması ve iki sütunlu düzen.

Sayfa ayarları Düzen sekmesindeki Sayfa Yapısı grubunda toplanır (Şekil 3.1.23):

- **Kenar Boşlukları** (*margins*): metin ile kâğıdın kenarı arasındaki boş alandır. Ciltlenecek belgelerde cilt tarafına ek bir **cilt payı** verilebilir.
- **Yönlendirme** (*orientation*): sayfa **dikey** ya da **yatay** durur.
- **Boyut**: kâğıt ölçüsüdür. Türkiye'de ve Avrupa'da A4 (210 × 297 mm), ABD'de Letter (8,5 × 11 inç ≈ 216 × 279 mm) yaygındır.
- **Sütunlar**: metni gazete düzenine benzer sütunlara böler. Metin birinci sütunun sonuna gelince ikinci sütunun başından devam eder.
- **Satır Numaraları**: bazı hukuki ya da hakemli belgelerde satırları numaralandırır; böylece bir düzeltme "12. satır" diye gösterilebilir.
- **Heceleme** (*hyphenation*): uzun sözcükleri satır sonunda kısa çizgiyle bölerek iki yana yaslı metindeki büyük boşlukları azaltır.

Örnek 3.1.14

A4 sayfada dört kenar boşluğu da 2,5 cm. Metin için ne kadar alan kalır? Satır aralığı "Tam olarak 18 nk" ise bir sayfaya en fazla kaç satır sığar?

1. **Genişlik.** 210 mm − 2 × 25 mm = 160 mm.
2. **Yükseklik.** 297 mm − 2 × 25 mm = 247 mm.
3. **Satır yüksekliği.** 18 nk = 18 × 25,4 ÷ 72 mm = 6,35 mm (3.1.5.3).
4. **Satır sayısı.** 247 ÷ 6,35 ≈ 38,9. Yarım satır sayfaya sığmaz; aşağı yuvarlanır: 38 satır.

Aynı belge Letter boyutunda basılırsa sayfa 6 mm daha geniş ama 18 mm daha kısadır. Satır sayısı azalır ve sayfa sonları kayar. Bu yüzden belgenin kâğıt boyutu, yazmaya başlamadan önce teslim edileceği yere göre ayarlanır.

3.1.6.2 Sayfa sonu ve bölüm sonu

Bir sonraki içeriği yeni sayfadan başlatmanın yanlış ve doğru yolları vardır. Yanlış yol, sayfa bitene kadar Enter'a basmaktır. Doğru yol bir **kesme** eklemektir. Kesmeler Düzen → Kesmeler menüsündedir ve iki ana türe ayrılır.

Tanım 3.1.17

Sayfa sonu (*page break*; `Ctrl+Enter`), sonraki içeriği yeni bir sayfadan başlatır; belge aynı bölümde kalır. **Bölüm sonu** (*section break*) ise belgeyi iki bölüme ayırır. Her bölümün kendi kenar boşlukları, sayfa yönü, sütun sayısı, üstbilgisi, altbilgisi ve sayfa numarası biçimi olabilir.

Kesme türü	Ne yapar?	Ne zaman kullanılır?
Sayfa sonu	yeni sayfa; bölüm aynı	yeni bölüm başlığını sayfa başından başlatmak
Sonraki Sayfa bölüm sonu	yeni bölüm ve yeni sayfa	yeni sayfanın düzeni farklı olacaksa (yatay sayfa, farklı numara biçimi)
Sürekli bölüm sonu	yeni bölüm, aynı sayfada	aynı sayfada tek sütundan iki sütuna geçmek
Tek Sayfa / Çift Sayfa bölüm sonu	yeni bölümü bir sonraki tek ya da çift numaralı sayfadan başlatır	kitap ve tezde bölümleri hep sağ (tek) sayfadan başlatmak

Tablo 3.1.10: Kesme türleri ve kullanım yerleri.



Sayfa aynı bölüm, yeni sayfa — **Sonraki Sayfa** yeni bölüm + yeni sayfa — **Sürekli** aynı sayfada 2 sütuna geç — **Tek Sayfa** yeni bölüm tek sayfada

Şekil 3.1.24: Kesme türlerinin etkisi. Sayfa sonu: aynı bölümde yeni sayfa. Sonraki Sayfa: yeni bölüm (B2) yeni sayfada ve yatay olabilir. Sürekli: aynı sayfada iki sütuna geçiş. Tek Sayfa: yeni bölüm tek numaralı sayfada başlar; gerekirse araya boş bir çift sayfa (s.2) girer.

Şekil 3.1.24 dört kesmeyi yan yana gösterir. Kesmeler ¶ düğmesi açıkken "Sayfa Sonu" ya da "Bölüm Sonu (Sonraki Sayfa)" gibi kesikli çizgiler olarak görünür. Bir kesme yanlış yerdeyse bu çizgi seçilip silinir. Bölüm sonunda ince bir nokta vardır: bölümün düzen ayarları, bölümün *sonundaki* bölüm sonu işaretinde saklanır. Bir bölüm sonu silinince iki bölüm birleşir ve birleşen bölüm, sonraki bölümün ayarlarını alır.

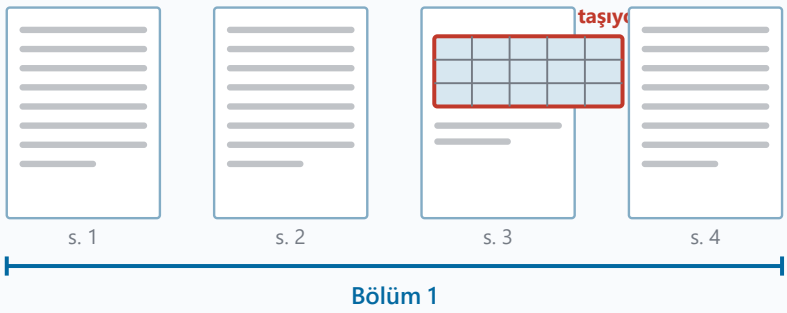
Sık yapılan hata

Yeni sayfaya geçmek için art arda Enter'a basmak. Önceki sayfaya bir satır eklenince bütün boş satırlar bir aşağı kayar; başlık sayfanın ortasına düşer ya da önüne boş bir sayfa açılır. `Ctrl+Enter` ile eklenen sayfa sonu ise metin ne kadar değişirse değişsin sonraki içeriği yeni

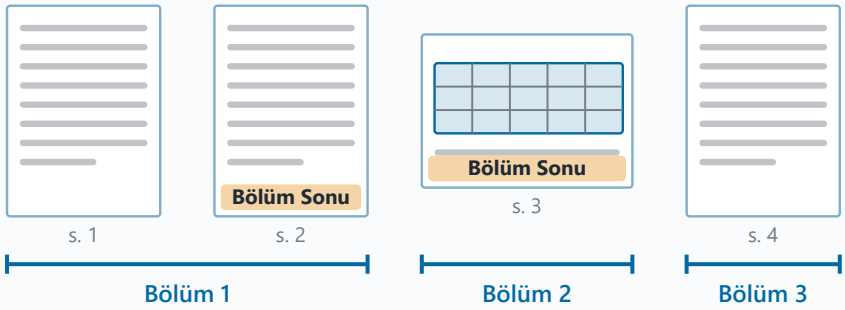
sayfanın başında tutar.

3.1.6.3 Bölüm sonuyla tek sayfayı yatay yapmak

Bölüm sonunun en tipik kullanımı, uzun bir dikey belgenin içinde tek bir sayfayı yatay yapmaktır. Geniş bir ölçüm tablosu dikey sayfaya sığmaz. Yönlendirme komutu ise imlecin bulunduğu bölüme uygulanır. Belge tek bölümse bütün sayfalar yatay olur. Çözüm, tabloyu kendi bölümüne almaktır.



Şekil 3.1.25: Başlangıç: belge tek bölüm (Bölüm 1) ve dört sayfanın hepsi dikey. 3. sayfadaki geniş tablo sağ kenardan taşıyor.



Şekil 3.1.26: Tablonun önüne ve arkasına birer Sonraki Sayfa bölüm sonu eklendi; imleç Bölüm 2'deyken Yönlendirme → Yatay seçildi. Yalnızca 3. sayfa yatay oldu, tablo sığdı.

Örnek 3.1.15

Dört sayfalık belgede geniş tablo 3. sayfada. Amaç yalnızca 3. sayfayı yatay yapmak. Adımlar ve her adımdaki bölüm yapısı şöyledir (Şekil 3.1.25 ve Şekil 3.1.26):

Adım	İşlem	Bölümler	Yatay sayfalar
0	başlangıç	B1: s.1–4	yok

Adım	İşlem	Bölümler	Yatay sayfalar
1	imleç tablonun önüne; Düzen → Kesmeler → Sonraki Sayfa	B1: s.1–2 · B2: s.3–4	yok
2	imleç tablonun arkasına; yine Sonraki Sayfa	B1: s.1–2 · B2: s.3 · B3: s.4	yok
3	imleç tablodayken Düzen → Yönlendirme → Yatay	aynı	yalnızca s.3

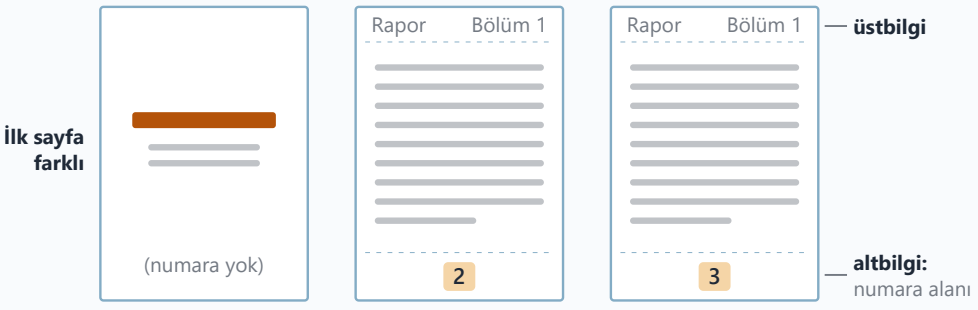
Adım 2 atlanırsa ne olur? Adım 3'teki yönlendirme B2'nin tamamına, yani 3. ve 4. sayfalara uygulanır. Bölüm sonu yerine sayfa sonu kullanılsaydı belge tek bölüm kalır ve yön değişikliği dört sayfanın hepsine uygulanırdı. k tane bölüm sonu belgeyi $k + 1$ bölüme ayırır; tek bir sayfayı yalıtmak için iki bölüm sonu gerekir.

Kenar boşluğu, sütun sayısı, üstbilgi ve sayfa numarası biçimi de aynı yolla bölüm bazında değiştirilir. Bir komutun yalnızca belgenin bir kısmını etkilemesi isteniyorsa ilk soru şudur: o kısım ayrı bir bölüm mü?

3.1.6.4 Üstbilgi, altbilgi ve sayfa numarası

Tanım 3.1.18

Üstbilgi ve **altbilgi** (*header / footer*), her sayfanın üst ve alt kenar boşluğunda tekrarlanan alanlardır. Belge başlığı, bölüm adı, yazar, tarih ve sayfa numarası gibi bilgiler buraya yazılır. Ekle → Üst Bilgi / Alt Bilgi komutlarıyla ya da sayfanın üst veya alt kenarına çift tıklanarak düzenlenir.



Şekil 3.1.27: Üç sayfalık rapor. Kapakta üstbilgi ve numara yok (ilk sayfa farklı); sonraki sayfalarda üstbilgi "Rapor · Bölüm 1", altbilgide otomatik sayfa numarası (2, 3).

Sayfa numarası, Ekle → Sayfa Numarası menüsünden **alan** olarak eklenir (Şekil 3.1.27). Alan, Word'ün o anki duruma göre hesaplayıp gösterdiği bir değerdir. Altbilgiye bir kez eklenen numara alanı her sayfada o sayfanın numarasını gösterir. Sayfalar eklenip silindikçe numaralar kendiliğinden güncelle-

nir. Elle yazılmış bir “3” ise her sayfada “3” olarak kalır.

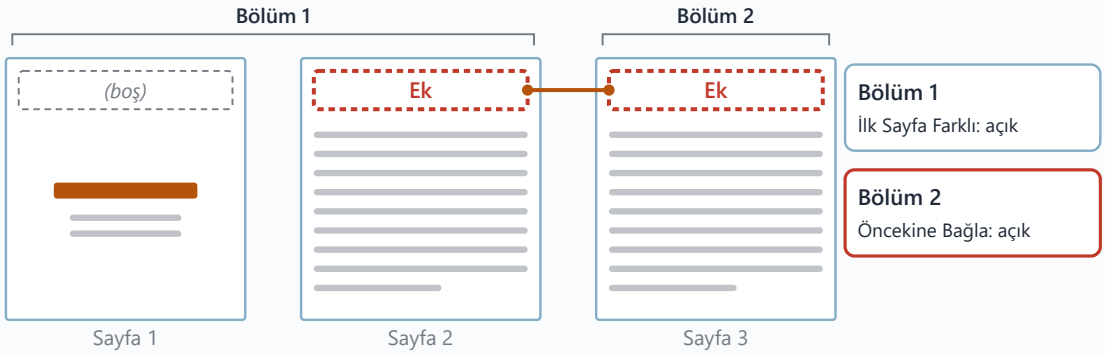
Üstbilgi ve altbilgi için iki seçenek daha vardır. **İlk Sayfa Farklı** işaretlenince bölümün ilk sayfası ayrı bir üstbilgi/altbilgi alanı kazanır; kapakta numara göstermemek için bu alan boş bırakılır. **Tek ve Çift Sayfalar Farklı** ise çift taraflı basılan belgelerde numarayı tek sayfalarda sağa, çift sayfalarda sola koymayı sağlar.

3.1.6.5 İlk Sayfa Farklı ve Öncekine Bağla

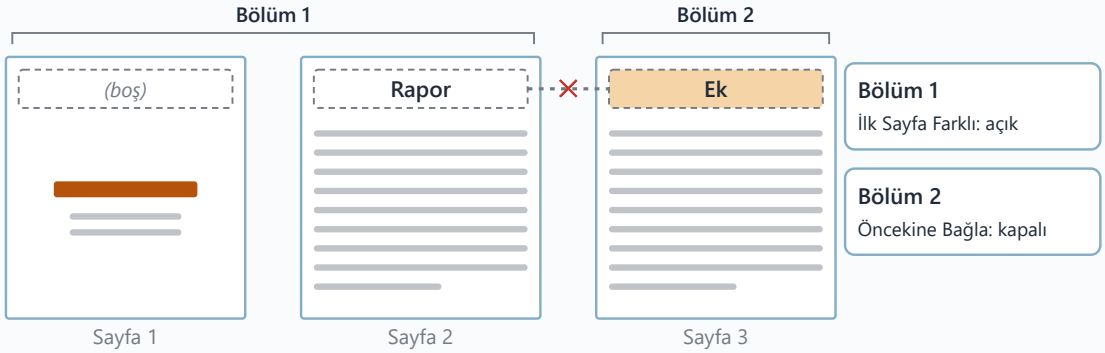
Belge birden çok bölüme ayrılınca üstbilgilerde yeni bir kural devreye girer. Yeni bir bölüm, üstbilgi ve altbilgisini varsayılan olarak bir önceki bölümünkiyle *bağlı* başlatır.

Tanım 3.1.19

Öncekine Bağla, üstbilgi ya da altbilgi düzenlenirken görünen Üst Bilgi ve Alt Bilgi bağlamsal sekmesindeki düğmedir. Açıkken bölümün üstbilgisi önceki bölümünkiyle aynı içeriği paylaşır: birinde yapılan değişiklik ötekinde de görünür. Kapatılınca bölümün kendi bağımsız üstbilgisi olur.



Şekil 3.1.28: Yanlış yol. Öncekine Bağla açıkken Bölüm 2'nin üstbilgisi “Ek” yapıldı. Üstbilgi iki bölümde ortak olduğu için 2. sayfa da “Ek” oldu. Kapak (İlk Sayfa Farklı) boş kaldı.



Şekil 3.1.29: Doğru yol. Önce Bölüm 2'de Öncekine Bağla kapatıldı (✕), sonra "Ek" yazıldı. Yalnızca 3. sayfa "Ek"; 1. sayfa boş, 2. sayfa "Rapor".

Örnek 3.1.16

Üç sayfalık raporun üstbilgisi "Rapor". Kapakta üstbilgi istenmiyor; 3. sayfa bir ek ve üstbilgisi "Ek" olmalı. Adımlar ve her adımda üç sayfanın üstbilgisi şöyledir (Şekil 3.1.28 ve Şekil 3.1.29):

Adım	İşlem	s.1	s.2	s.3
1	tek bölüm; üstbilgiye "Rapor" yazılır	Rapor	Rapor	Rapor
2	İlk Sayfa Farklı işaretlenir, kapak alanı boş bırakılır	(boş)	Rapor	Rapor
3	3. sayfanın başına Sonraki Sayfa bölüm sonu; B2'de Öncekine Bağla açık	(boş)	Rapor	Rapor
4a	yanlış: bağ açıkken B2'de "Ek" yazılır	(boş)	**Ek**	Ek
4b	doğru: 4a geri alınır; B2'de Öncekine Bağla kapatılır, sonra "Ek" yazılır	(boş)	Rapor	Ek

Adım 4a'nın sonucu şaşırtıcı görünür: yalnızca 3. sayfa düzenlendiği hâlde 2. sayfa da değişti. Sebep, bağlı üstbilgilerin tek bir içerik olmasıdır. Kapak ise İlk Sayfa Farklı ile kendi ayrı alanı olduğu için etkilenmedi. Kural kısadır: farklı içerik yazmadan *önce* bağ koparılır.

Sık yapılan hata

Bölüm sonu ekledikten sonra yeni bölümün üstbilgisini hemen değiştirmek. Öncekine Bağla açık olduğu için değişiklik geriye doğru bütün bağlı bölümlere yayılır. Yeni bölümün üstbilgisi düzenlenmeden önce bağlamsal sekmedeki Öncekine Bağla düğmesinin durumuna bakılır.

3.1.6.6 Bölümlere göre sayfa numarası

Tezlerde ve uzun raporlarda sayfa numaralandırması iki parçalıdır. Kapak, özet ve içindekiler gibi ön sayfalar küçük Roma rakamlarıyla (i, ii, iii), ana metin ise Giriş sayfasından başlayarak 1, 2, 3 ile numaralanır. Bu düzen bir bölüm sonu ve bölüme özgü sayfa numarası ayarlarıyla kurulur.

Tanım 3.1.20

Sayfa Numarası Biçimi iletişim kutusu, Ekle → Sayfa Numarası → Sayfa Numaralarını Biçimlendir komutuyla açılır ve iki bölüm ayarını yönetir. **Sayı biçimi** numaranın yazılışıdır: 1, 2, 3 ya da i, ii, iii gibi. **Sayfa numaralandırma** ise numaranın nereden başlayacağıdır: "Önceki bölümden devam et" ya da "Başlangıç" kutusuna yazılan değer. İki ayar da imlecin bulunduğu bölüme uygulanır.

Küçük Roma rakamları için bir hatırlatma yararlı olur: i = 1, v = 5, x = 10. Küçük simge büyüğün önüne gelirse çıkarılır, arkasına gelirse eklenir. Buna göre ii = 2, iii = 3, iv = 5 − 1 = 4, vi = 5 + 1 = 6'dır.



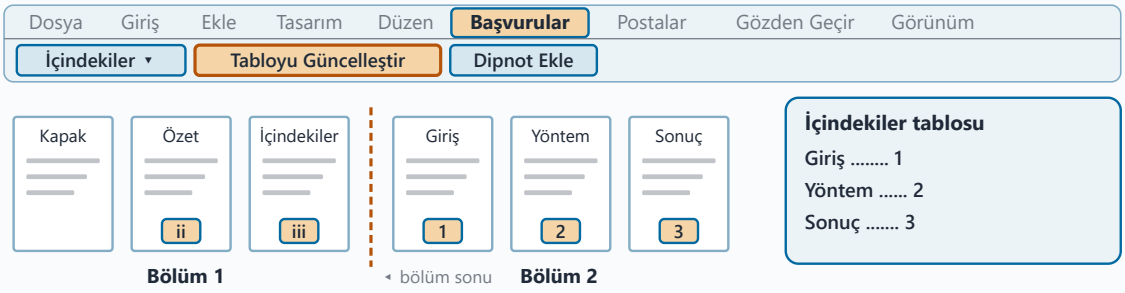
Şekil 3.1.30: İki bölümlü rapor. Bölüm 1 (Kapak, Özet, İçindekiler) küçük Roma rakamlı ve İlk Sayfa Farklı açık: kapakta numara görünmez, ama sayılır; Özet ii, İçindekiler iii. Bölüm 2 (Giriş, Yöntem, Sonuç) 1'den başlar.

Örnek 3.1.17

Altı sayfalık raporda ilk üç sayfa Kapak, Özet ve İçindekiler; son üç sayfa Giriş, Yöntem ve Sonuç. Adımlar ve her adımdan sonra altı sayfanın altbilgisinde görünen numara şöyledir (Şekil 3.1.30):

Adım	İşlem	Kapak	Özet	İçind.	Giriş	Yöntem	Sonuç
1	İçindekiler sonuna Sonraki Sayfa bölüm sonu	–	–	–	–	–	–
2	Sayfa Numarası → Sayfanın Altı	1	2	3	4	5	6
3	imleç B1'de; sayı biçimi i, ii, iii	i	ii	iii	4	5	6
4	imleç B2'de; Başlangıç: 1	i	ii	iii	1	2	3
5	B1'de İlk Sayfa Farklı	(yok)	ii	iii	1	2	3

Adım 3'te Bölüm 2 hâlâ 4'ten sayar, çünkü biçim bölüme aittir ve yalnızca Bölüm 1 değişti. Bölüm 2 "önceki bölümden devam et" ayarındadır. Adım 4 bu bağı koparır. Adım 5'te kapakta numara görünmez ama sayılır: Özet yine ii'dir, i değil.



Şekil 3.1.31: İçindekiler tablosu numaralar değiştikten sonra hâlâ eski değerleri (4, 5, 6) gösteriyordu. Başvurular → Tabloyu Güncelleştir ile Giriş 1, Yöntem 2, Sonuç 3 oldu.

Son bir adım daha gerekir (Şekil 3.1.31). Belgede daha önce eklenmiş bir içindekiler tablosu varsa, numaralandırma değişince tablo kendiliğinden güncellenmez; eski numaraları göstermeye devam eder. İçindekiler tablosu da bir alandır ve Tabloyu Güncelleştir komutuyla yenilenir (3.1.9.1).

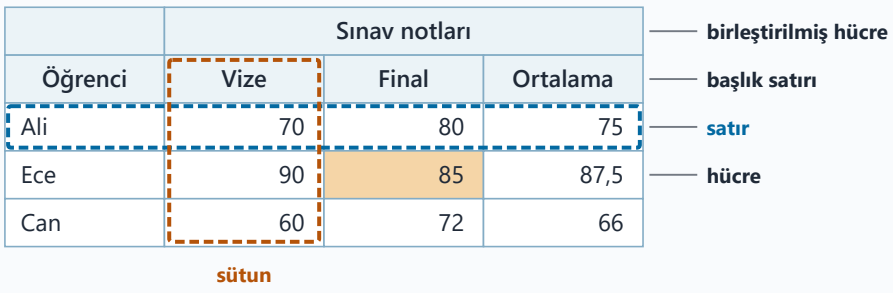
3.1.7 Tablolar ve resimler

Mühendislik raporlarında sayısal sonuçlar tablolarla, düzenekler ve grafikler resimlerle verilir. Word bu nesneleri metnin içine yerleştirir ve her biri için ayrı bağlamsal sekmeler sunar. Bu kesim, tabloların yapısını ve resimlerin metinle ilişkisini anlatır.

3.1.7.1 Tablolar

Tanım 3.1.21

Tablo, bilgiyi satır ve sütunlardan oluşan hücrelerde düzenleyen nesnedir; Ekle → Tablo ile eklenir. **Satır** yatay, **sütun** dikey hücre dizisidir; ikisinin kesiştiği kutuya **hücre** denir. Sütunların anlamını açıklayan ilk satıra **başlık satırı** (*header row*) denir.



Şekil 3.1.32: Tablonun öğeleri. En üstte birleştirilmiş hücre ("Sınav notları"), altında başlık satırı (Öğrenci, Vize, Final, Ortalama) ve üç veri satırı: Ali 70, 80, 75; Ece 90, 85, 87,5; Can 60, 72, 66.

Tablonun düzenini kurarken birkaç ilke işe yarar (Şekil 3.1.32):

- **Başlık satırı** her zaman olmalıdır. Tablo birden çok sayfaya taşarsa başlık satırının her sayfada tekrarlanması seçilebilir (tablo Düzen sekmesi → Başlık Satırlarını Yinele).
- **Hücre birleştirme** (*merge cells*) yalnızca anlamlı bir başlık grubu için kullanılmalıdır. Fazla birleştirilmiş hücre, ekran okuyucuların tabloyu doğru okumasını zorlaştırır ve sıralamayı bozar.
- **Tablo stilleri** (Tablo Tasarımı sekmesi) kenarlık ve renkleri bütün tabloya tutarlı uygular; hücreleri tek tek boyamaya gerek kalmaz.
- **Hizalama**: metin sola, sayılar sağa ya da ondalık virgüle göre hizalanır; böylece basamaklar alt alta gelir.

Örnek 3.1.18

Şekil 3.1.32'deki Ortalama sütunu, vize ve final notlarının aritmetik ortalamasıdır. Değerleri denetleyelim:

1. Ali: (70 + 80) ÷ 2 = 150 ÷ 2 = 75.
2. Ece: (90 + 85) ÷ 2 = 175 ÷ 2 = 87,5.
3. Can: (60 + 72) ÷ 2 = 132 ÷ 2 = 66.

Üç değer de doğrudur. Word tabloları hesap tablosu değildir: bir not değişince ortalama kendiliğinden güncellenmez. Tablo Düzen sekmesindeki Formül komutu basit hesaplar yapabilir, ama hesap ağırlıklı tablolar Excel'de hazırlanıp Word'e eklenir.

3.1.7.2 Resimler

Resimler Ekle → Resimler ile bilgisayardan, çevrim içi kaynaklardan ya da hazır görsellerden eklenir. Bir resim seçilince **Resim Biçimi** bağlamsal sekmesi açılır. Bu sekmedeki üç araç özellikle önemlidir:

- **Alternatif metin** (*alt text*): görselin amacını, onu göremeyen ekran okuyucu kullanıcılarına açıklayan kısa metindir. “resim1.png” gibi bir dosya adı değil, görselin anlattığı bilgi yazılır (3.1.11.1).
- **Kırpma** (*crop*): görselin gereksiz kenar bölümlerini çerçevenin dışında bırakır. Kırpılan kısım silinmez, gizlenir; gerekirse geri açılabilir. Dosya boyutunu küçültmek için kırpılan alanlar ayrıca silinebilir.
- **Düzen Seçenekleri**: resim seçilince sağ üst köşesinde beliren küçük düğmedir. Resmin metinle ilişkisini, yani metin kaydırma türünü değiştirir.

3.1.7.3 Metin kaydırma türleri

Tanım 3.1.22

Metin kaydırma (*text wrapping*), bir resmin çevresindeki metinle nasıl yerleşeceğini belirleyen ayardır. **Satır içi** (Word'deki adı "Metinle Aynı Hizada") resmi metnin bir karakteri gibi satıra yerleştirir. Diğer türlerde resim metinden bağımsız, sayfada serbestçe taşınabilen bir nesne olur.



Şekil 3.1.33: Dört metin kaydırma türü. Satır içi: resim bir karakter gibi. Kare: metin resmin çevresinden akar. Üst ve Alt: resmin yanları boş kalır. Metnin Arkasında: resim metnin altında; okunabilirlik riski.

Tür	Davranış	Uygun kullanım
Metinle Aynı Hizada	satırda bir karakter gibi durur, satır yüksekliğini büyütür; konumu metne bağlıdır	raporlarda kendi paragrafında duran şekiller; en kararlı tür
Kare	metin resmin dikdörtgen çerçevesinin çevresinden akar	broşür, haber bülteni
Sıkı / İçinden	metin resmin gerçek çevresine yakın akar	arka planı saydam, düzensiz biçimli görseller
Üst ve Alt	metin resmin yalnızca üstünde ve altında durur; yanlar boş	sayfa genişliğine yakın görseller
Metnin Arkasında	resim metnin altındaki katmandadır; metin üstten okunur	filigran, soluk arka plan logosu
Metnin Önünde	resim metnin üstündeki katmandadır; altındaki metni örter	nadiren; metni gizleyebilir

Tablo 3.1.11: Metin kaydırma türleri ve kullanım yerleri.



Şekil 3.1.34: Aynı resim dört kaydırma türünde (sol üstten başlayarak satır satır): satır içi (satır yüksekliği büyür), kare (metin çevresinden akar), metnin arkasında (metin üstten okunur), metnin önünde (altındaki metin görünmez).

Şekil 3.1.34 dört türü aynı resim üzerinde karşılaştırır. Raporlarda en güvenli seçim satır içidir. Resim kendi paragrafında durur, ortalanabilir ve altına resim yazısı eklenebilir. Metin eklenip silindikçe resim de metinle birlikte akar. Kare ya da Sıkı gibi serbest türlerde resim sayfaya sabitlenir. Önündeki metin uzayınca resim yerinde kalabilir ve ilgili paragraftan ayrı bir sayfaya düşebilir.

Örnek 3.1.19

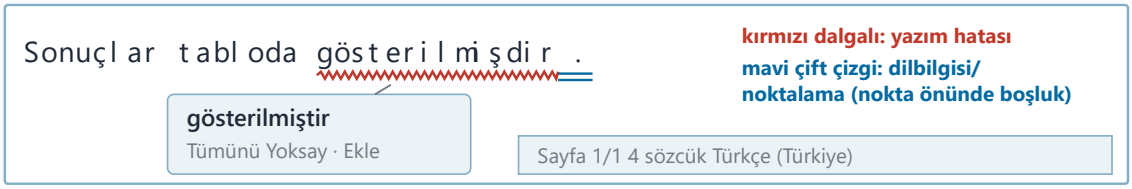
Kulüp duyurusunun arka planına soluk bir logo konacak; metin logonun üzerinden okunabilmeli. Hangi tür seçilir ve logo nasıl hazırlanır?

1. **Tür.** Logo metnin altındaki katmanda olmalıdır: **Metnin Arkasında**. Metnin Önünde metni örter, Kare metni logonun çevresine iter, satır içi ise logoyu bir satıra yerleştirir.
2. **Soluklaştırma.** Resim Biçimi → Renk menüsünden açık bir ton ya da "Soluk" seçilir. Logo soluk değilse metnin okunabilirliği düşer.
3. **Alternatif yol.** Logo her sayfada aynı yerde tekrarlanacaksa Tasarım → Filigran → Özel Filigran ile resim filigranı eklenir. Word bu durumda soluklaştırmayı ve konumlandırmayı kendisi yapar.

3.1.8 Gözden geçirme ve iş birliği

Bir belge yazıldıktan sonra en az bir kez gözden geçirilir: yazım hataları düzeltilir, ifadeler toparlanır, başka birinin önerileri değerlendirilir. Gözden Geçir sekmesi bu işlerin araçlarını toplar. Araçların bir kısmı yazarın kendi denetimine, bir kısmı birden çok kişinin aynı belge üzerinde çalışmasına yöneliktir.

3.1.8.1 Dil, yazım denetimi ve sözcük sayısı



Şekil 3.1.35: Yazım denetimi. "Sonuçlar tabloda gösterilmişdir ." cümlesinde "gösterilmişdir" kırmızı dalgalı çizgiyle (yazım hatası), nokta önündeki boşluk mavi çift çizgiyle (dil bilgisi ya da noktalama) işaretli. Sağ tıklamayla "gösterilmiştir" önerisi çıkar. Durum çubuğunda "4 sözcük" ve "Türkçe (Türkiye)" görünür.

Word yazarken metni arka planda denetler (Şekil 3.1.35). Sözlükte bulunmayan bir sözcüğün altı kırmızı dalgalı çizgiyle, dil bilgisi ve noktalama şüphesi mavi çift çizgiyle işaretlenir. İşaretli sözcüğe sağ tıklanınca öneriler listelenir. Öneri seçilebilir, işaret yok sayılabilir ya da sözcük kullanıcının sözlüğüne eklenebilir.

Denetimin kurallarını **belge dili** belirler. Durum çubuğunda "Türkçe (Türkiye)" yazıyorsa Türkçe sözlük kullanılır. Türkçe bir metin yanlışlıkla İngilizce dilinde işaretlenmişse neredeyse her sözcüğün altı kırmızı olur. Dil, metin seçilip Gözden Geçir → Dil → Yazım Denetleme Dilini Ayarla ile değiştirilir.

Böylece bir belgede Türkçe paragraflar ve İngilizce bir özet yan yana, her biri kendi kurallarıyla denetlenebilir.

Araç	Görevi
Düzenleyici	yazım, dil bilgisi ve ifade önerilerini tek bölmede toplar (*Editor*)
Sözcük Sayısı	sözcük, karakter (boşluklu ve boşluksuz), paragraf, satır ve sayfa sayısını gösterir
Eş Anlamlılar	seçili sözcük için yakın anlamlı seçenekler sunar
Otomatik Düzeltme	sık yapılan yazım hatalarını yazarken düzeltir (ör. “dahada” → “daha da” gibi tanımlı eşleşmeler)

Tablo 3.1.12: Gözden geçirme araçları.

Sözcük sayısı ödevlerde sık istenen bir ölçüttür. Durum çubuğu bütün belgenin sözcük sayısını gösterir; bir metin seçiliyse “120 / 2.350 sözcük” gibi seçimin sayısını da verir. Gözden Geçir → Sözcük Sayısı penceresi ayrıntılı sayımı sunar ve dipnotların sayıma katılıp katılmayacağını sorar.

Sık yapılan hata

Altı çizili olmayan metni hatasız saymak. Denetim yalnızca sözlükte olmayan sözcükleri yakalar. “Devre şemasındaki direnç değeri 10 kΩ'dur” yerine “100 kΩ” yazmak, “ve” yerine “de” kullanmak ya da bir cümleyi yarım bırakmak çoğu zaman işaretlenmez. Otomatik öneriler de bağlamı bilmez; her öneriyi kabul etmeden önce yazar değerlendirir.

3.1.8.2 Açıklamalar ve değişiklikleri izleme

Bir öğrencinin tez taslağını danışmanına gönderdiğini düşünelim. Danışman metne doğrudan müdahale ederse öğrenci neyin değiştiğini göremez. Kenara elle not yazarsa notlar metinle karışır. Word bu iş birliği için iki ayrı araç sunar.

Tanım 3.1.23

Açıklama (*comment*), metni değiştirmeden bir yere bağlanan soru, öneri ya da geri bildirimdir; Gözden Geçir → Yeni Açıklama ile eklenir ve sağ kenarda balon olarak görünür. **Değişiklikleri İzle** (*track changes*) açıkken yapılan her ekleme, silme ve biçim değişikliği, kimin ve ne zaman yaptığıyla birlikte kaydedilir. Kaydedilen değişiklik, yazar **kabul edene** ya da **reddedene** kadar bekleyen durumda kalır.

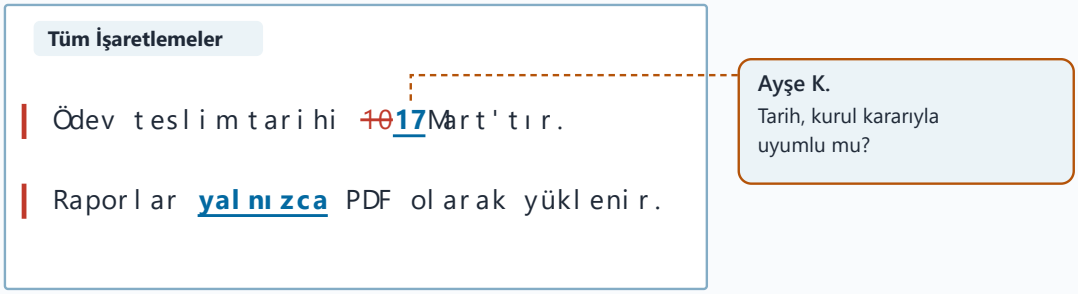


Şekil 3.1.36: Üstte açıklama: "Toplantı Salı günü yapılacak." metni aynı kalır, Ayşe K.'nın "Saati de ekler misin?" sorusu kenarda balondadır. Altta değişiklik izleme: silinen "Salı" üstü çizili, eklenen "Çarşamba" altı çizili görünür. Kabul Et ile "Çarşamba" kalır, Reddet ile "Salı" geri gelir.

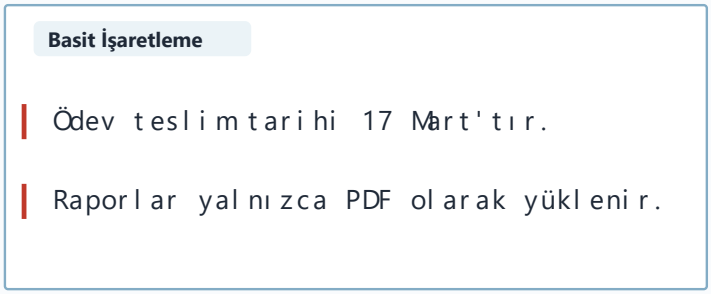
Şekil 3.1.36 iki aracın farkını gösterir. Açıklama bir konuşmadır: yanıtlanabilir, "çözüldü" olarak işaretlenebilir ya da silinebilir. Metne hiçbir zaman dokunmaz. Değişiklik izleme ise metnin kendisinde bir öneridir: kabul edilirse metin kesinleşir, reddedilirse eski hâline döner. Değişiklikler tek tek ya da Kabul Et → Tüm Değişiklikleri Kabul Et ile topluca kabul edilebilir.

3.1.8.3 İşaretleme görünümleri

Değişiklik izlenen bir belge ekranda farklı ayrıntı düzeylerinde gösterilebilir. Bu görünümler Gözden Geçir → İzleme grubundaki listeden seçilir. Görünüm yalnızca gösterimi değiştirir; bekleyen değişikliklerin durumuna dokunmaz.



Şekil 3.1.37: Tüm İşaretlemeler görünümü. "10" silinmiş (üstü çizili), "17" ve "yalnızca " eklenmiş (altı çizili); kenarda değişiklik çubukları ve Ayşe K.'nın açıklaması: "Tarih, kurul kararıyla uyumlu mu?"



Şekil 3.1.38: Aynı belge Basit İşaretleme görünümünde: metin son hâliyle görünür, ama değişiklikler hâlâ bek-

liyor.

Görünüm	Ne gösterir?
Basit İşaretleme	metni değişiklikler uygulanmış gibi gösterir; kenarda yalnızca küçük bir çubuk kalır
Tüm İşaretlemeler	silinenleri üstü çizili, eklenenleri altı çizili ve renkli, açıklamaları balonda gösterir
İşaretleme Yok	metni son hâliyle, hiçbir işaret olmadan gösterir
Özgün	metni değişikliklerden önceki hâliyle gösterir

Tablo 3.1.13: İşaretleme görünümleri.

Örnek 3.1.20

Belgede iki satır var: “Ödev teslim tarihi 10 Mart'tır.” ve “Raporlar PDF olarak yüklenir.” Değişiklikleri İzle açılıyor ve şu işlemler yapılıyor:

Adım	İşlem	Tüm İşaretlemeler'de 1. satır	Bekleyen
1	“10” silinir, “17” yazılır	Ödev teslim tarihi ~~10~~ 17 Mart'tır.	2
2	2. satırda “PDF” önüne “yalnızca ” eklenir	aynı	3
3	tarih seçilip Yeni Açıklama	aynı; kenarda balon	3 + 1 açıklama
4	Basit İşaretleme'ye geçilir	Ödev teslim tarihi 17 Mart'tır.	3 + 1 açıklama
5	Tüm Değişiklikleri Kabul Et; açıklama silinir	Ödev teslim tarihi 17 Mart'tır.	0

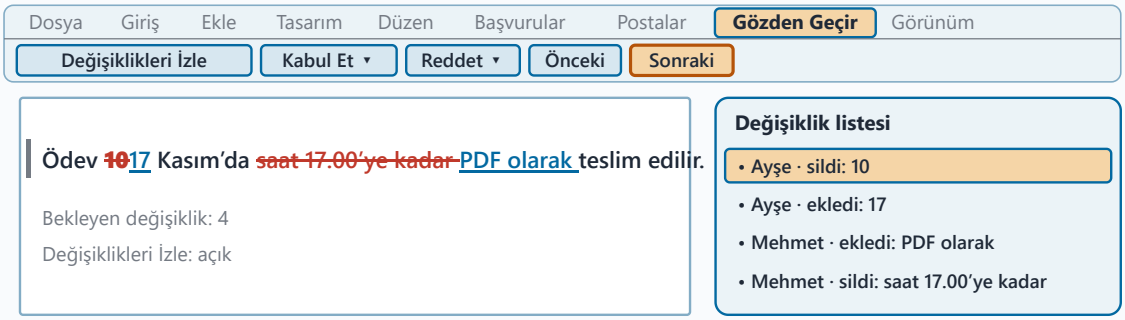
4. ve 5. adımda ekrandaki metin aynıdır, ama belgenin durumu farklıdır. 4. adımda değişiklikler yalnızca gizlenmiştir. Belge bu hâliyle gönderilirse karşı taraf Tüm İşaretlemeler'e geçip silinen “10”u görebilir. 5. adımda ise değişiklikler belgeye işlenmiş, açıklama silinmiştir.

Sık yapılan hata

Basit İşaretleme ya da İşaretleme Yok görünümünde temiz görünen belgeyi son hâl sanıp göndermek. Gizlemek kabul etmek değildir. Paylaşmadan önce Gözden Geçir sekmesinde bekleyen değişiklik ve açıklama kalmadığı denetlenir; Dosya → Bilgi → Sorunları Denetle → Belgeyi İncele de bunları listeler.

3.1.8.4 Değişiklikleri tek tek kabul etmek ve reddetmek

Birden çok kişinin önerisi aynı belgede toplandığında her öneri ayrı ayrı değerlendirilir. Gözden Geçir sekmesindeki **Sonraki** ve **Önceki** düğmeleri bekleyen değişiklikler arasında gezinir. **Kabul Et ve Sonrakine Git** o anki değişikliği kesinleştirip bir sonrakine, **Reddet ve Sonrakine Git** değişikliği geri alıp bir sonrakine geçer. Sıradaki değişiklik metindeki konumuna göre seçilir; kimin yaptığına ya da ne zaman yapıldığına göre değil.



Şekil 3.1.39: Başlangıç: Ayşe ile Mehmet'in dört değişikliği bekliyor. Değişiklik listesi: Ayşe "10"u sildi, Ayşe "17" ekledi, Mehmet "PDF olarak" ekledi, Mehmet "saat 17.00'ye kadar" ifadesini sildi.

Örnek 3.1.21

Metin: "Ödev ~~10~~17 Kasım'da ~~saat 17.00'ye kadar~~ PDF olarak teslim edilir." Saat bilgisinin metinde kalması gerekiyor; diğer üç değişiklik doğru. Sonraki ile başlayıp her adımda şu işlemler yapılıyor (Şekil 3.1.39):

Adım	Seçili değişiklik	Komut	Sonuç	Bekleyen
0	—	—	—	4
1	Ayşe: "10" silindi	Kabul Et ve Sonrakine Git	"10" metinden tamamen çıktı	3
2	Ayşe: "17" eklendi	Kabul Et ve Sonrakine Git	"17" sıradan metin oldu	2
3	Mehmet: "saat 17.00'ye kadar" silindi	Reddet ve Sonrakine Git	silme geri alındı; ifade metinde kaldı	1
4	Mehmet: "PDF olarak" eklendi	Kabul Et	ekleme kesinleşti	0
5	—	Değişiklikleri İzle kapatılır	bundan sonraki düzenlemeler kaydedilmez	0

Son metin: "Ödev 17 Kasım'da saat 17.00'ye kadar PDF olarak teslim edilir." Adım 3'te sıradaki değişiklik Mehmet'in eklemesi değil, metinde daha önce gelen silmesidir. Bir silmeyi *reddetmek* silinen metni geri getirir; bir eklemeyi reddetmek ise eklenen metni çıkarır.

Değişiklikleri İzle'yi kapatmayı unutmak da sık görülür. Düğme açık kaldıkça yazarın kendi düzeltmeleri de birer bekleyen değişiklik olarak birikir. Belgenin son okumasında bunlar da tek tek kabul edilmek zorunda kalır.

3.1.9 Uzun belgeler

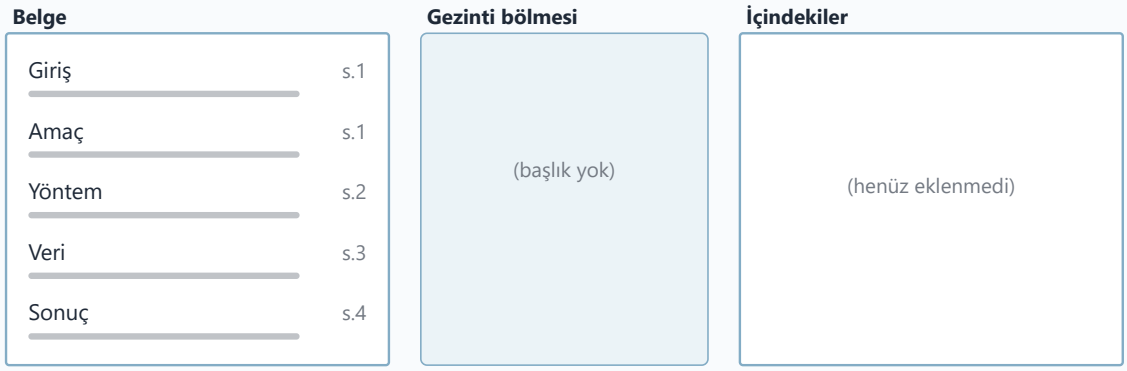
On sayfayı aşan bir belgede okuyucu aradığını bulmak için yol göstericilere ihtiyaç duyar: içindekiler, numaralı başlıklar, numaralı şekiller, dipnotlar ve kaynakça. Bu öğelerin hepsi elle de yazılabilir. Ama belge değiştikçe elle yazılmış her numara bir hata kaynağına dönüşür. Word bu öğelerin hepsini belgenin yapısından otomatik üretir ve güncel tutar.

3.1.9.1 Başlık stilleri ve içindekiler

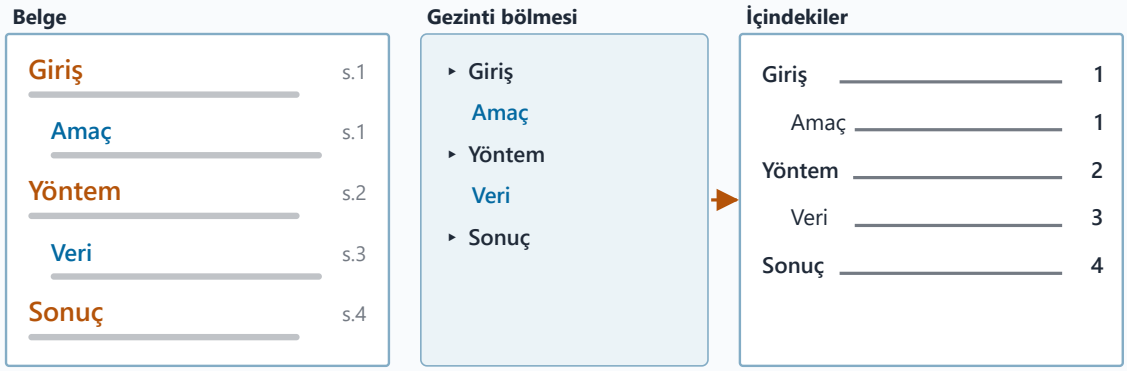
Otomatik içindekiler tablosunun kurulması dört adımlık bir akıştır: başlık stilleri uygulanır, düzeyler gezinti bölmesinde denetlenir, içindekiler eklenir ve belge değiştikçe tablo güncellenir.

Tanım 3.1.24

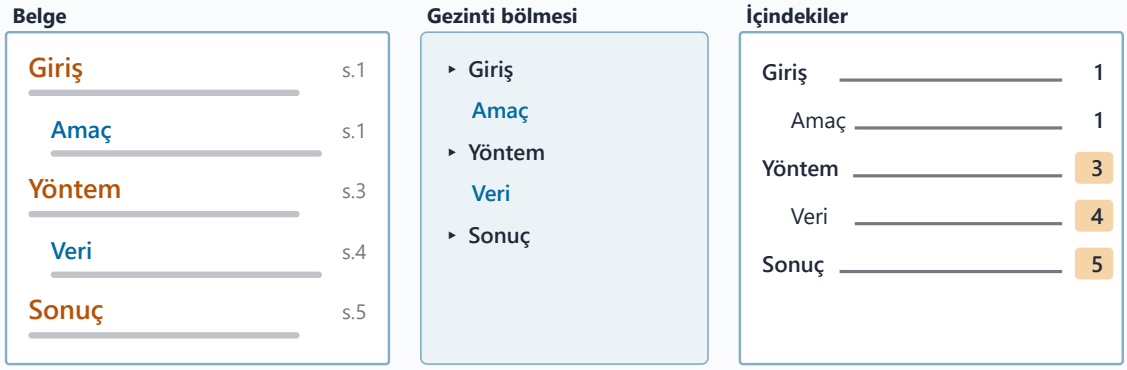
İçindekiler tablosu (*table of contents*), Başvurular → İçindekiler ile eklenen ve belgedeki başlık stillerinden (Başlık 1, Başlık 2, Başlık 3) otomatik üretilen listedir. Her satırda başlığın metni ve sayfa numarası bulunur. Tablo bir **alandır**: belge değiştiğinde kendiliğinden yenilenmez, **Tabloyu Güncelleştir** komutuyla (ya da tablo seçiliyken F9) güncellenir.



Şekil 3.1.40: Başlıklar yalnızca kalın ve büyük yazılmış, Normal stilde. Word bunları başlık olarak tanımaz; gezinti bölmesi boş.



Şekil 3.1.41: Giriş, Yöntem ve Sonuç'a Başlık 1, Amaç ve Veri'ye Başlık 2 uygulandı; gezinti bölmesi iki düzeyli yapıyı gösteriyor. Başvurular → İçindekiler ile tablo otomatik üretildi: Giriş 1, Amaç 1, Yöntem 2, Veri 3, Sonuç 4.



Şekil 3.1.42: Yöntem'den önce bir sayfalık metin eklendi ve Tabloyu Güncelleştir (F9) uygulandı: Yöntem 3, Veri 4, Sonuç 5.

Örnek 3.1.22

Raporda beş başlık var: Giriş ve Amaç 1. sayfada, Yöntem 2., Veri 3., Sonuç 4. sayfada. Adımlar ve her adımda gezinti bölmesi ile içindekiler tablosu şöyledir (Şekil 3.1.40, Şekil 3.1.41 ve Şekil 3.1.42):

Adım	İşlem	Gezinti bölmesi	İçindekiler (Giriş, Amaç, Yöntem, Veri, Sonuç)
1	başlıklar Normal stilde, kalın	boş	yok
2	Giriş, Yöntem, Sonuç'a Başlık 1	Giriş · Yöntem · Sonuç	yok
3	Amaç ve Veri'ye Başlık 2	Giriş › Amaç · Yöntem › Veri · Sonuç	yok
4	Başvurular → İçindekiler	aynı	1, 1, 2, 3, 4
5	Yöntem'den önce bir sayfa metin eklenir	aynı	1, 1, 2, 3, 4 (eski)
6	Tabloyu Güncelleştir	aynı	1, 1, 3, 4, 5

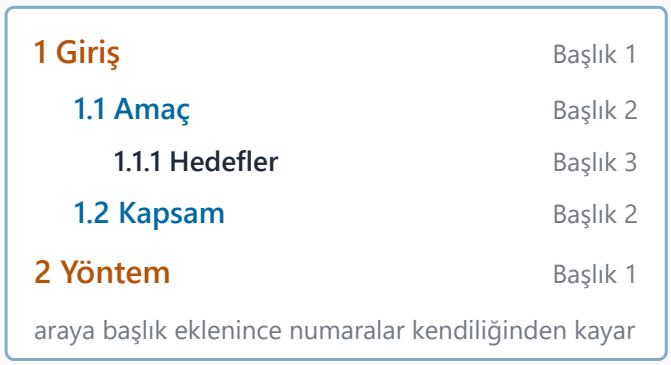
Adım 5'teki tutarsızlık bilinçli bir tasarımdır. Alanlar her tuş basışında yeniden hesaplanmaz; güncelleme kullanıcıya bırakılır. Gezinti bölmesi ise başlıkları anında gösterir. Tabloyu Güncelleştir iki seçenek sunar. **Yalnızca sayfa numaralarını güncelleştir** mevcut satırların numaralarını düzeltir. **Tüm tabloyu güncelleştir** ise yeni eklenen, silinen ya da adı değişen başlıkları da tabloya yansıtır. Belgeye yeni başlık eklendiyse ikinci seçenek gerekir.

Sık yapılan hata

İçindekiler tablosunu elle yazmak ya da otomatik tabloyu ekledikten sonra güncellemeyi unutmak. İkisinde de belge teslim edildiğinde numaralar yanlış olur. PDF'e aktarmadan ya da yazdırmadan önce bütün alanlar güncellenir: `Ctrl+A` ile belge seçilip `F9`'a basılır.

3.1.9.2 Çok düzeyli başlık numaralandırma ve tez biçimi

Tezlerde ve teknik raporlarda başlıklar numaralıdır: 1 Giriş, 1.1 Amaç, 1.1.1 Hedefler. Bu numaralar başlık metnine elle yazılırsa araya bir bölüm eklendiğinde sonraki bütün numaraların elle düzeltilmesi gerekir. Doğru yol, çok düzeyli bir listeyi başlık stillerine bağlamaktır.



Şekil 3.1.43: Başlık stillerine bağlı çok düzeyli numaralandırma: 1 Giriş (Başlık 1), 1.1 Amaç (Başlık 2), 1.1.1 Hedefler (Başlık 3), 1.2 Kapsam (Başlık 2), 2 Yöntem (Başlık 1). Araya başlık eklenince numaralar kendiliğinden kayar.

Bağlama işlemi Giriş → Paragraf → Çok Düzeyli Liste → Yeni Çok Düzeyli Liste Tanımla penceresinde yapılır. Pencerede “Diğer >>” ile ek seçenekler açılır ve her düzey için “Düzeyi stile bağla” alanından bir stil seçilir: 1. düzey Başlık 1'e, 2. düzey Başlık 2'ye, 3. düzey Başlık 3'e. Bundan sonra bir paragrafa Başlık 2 uygulamak, ona hem başlık görünüşünü hem de doğru numarayı birlikte verir (Şekil 3.1.43).

Örnek 3.1.23

Şekil 3.1.43'teki yapıda “1.1 Amaç” ile “1.1.1 Hedefler” arasına değil, “1.1.1 Hedefler”den sonra Başlık 2 stilinde “Yöntemin Sınırları” başlığı ekleniyor. Numaralar nasıl değişir?

1. Yeni başlık 1. bölümün ikinci Başlık 2'sidir: **1.2 Yöntemin Sınırları**.
2. Eski "1.2 Kapsam", aynı bölümün üçüncü Başlık 2'si olur: **1.3 Kapsam**.
3. "2 Yöntem" bir Başlık 1'dir ve değişmez. İçindekiler tablosu güncellenince yeni numaralar oraya da yansır.

Numaralar elle yazılmış olsaydı "1.2 Kapsam" elle "1.3" yapılacak, metindeki "1.2'de anlatıldığı gibi" türünden göndermeler de tek tek aranacaktı.

Tezlerin biçim kuralları kuruma göre değişir. Sık görülen değerler Tablo 3.1.14'te verilmiştir. Bunlar yalnızca örnektir; kenar boşluğu, yazı tipi ve kaynak gösterme biçimi kurumdan kuruma farklıdır. Geçerli kurallar kurumun yazım kılavuzunda yazar.

Konu	Sık görülen örnek	Nerede ayarlanır?
Kenar boşlukları	her kenarda yaklaşık 2,5 cm; cilt payı ayrı olabilir	Düzen → Kenar Boşlukları
Yazı tipi ve boyut	Times New Roman veya benzeri, 12 pt	Normal stilinin tanımı
Satır aralığı	1,5 veya çift; başlıklarda farklı olabilir	Paragraf iletişim kutusu (stil tanımında)

Tablo 3.1.14: Tez ve raporlarda sık görülen biçim değerleri ve Word'deki yerleri (örnek değerler).

Tablodaki "Nerede ayarlanır?" sütunu bir ilkeyi yansıtır: yazı tipi ve aralık gibi değerler tek tek paragraflara değil, *stillerin tanımına* verilir. Kurum kuralı değişirse yalnızca stil tanımı düzeltilir.

3.1.9.3 Dipnot, alıntı ve kaynakça

Tanım 3.1.25

Dipnot (*footnote*), bir ifadeye sayfanın altında verilen ek açıklamadır. **Sonnot** (*endnote*) ise açıklamaları belgenin ya da bölümün sonunda toplar. İkisi de Başvurular sekmesinden eklenir ve otomatik numaralanır. **Alıntı** (*citation*), metindeki bir kaynağa yapılan atıftır. **Kaynakça** (*bibliography*) ise belgede atıf yapılan kaynakların listesidir.



Şekil 3.1.44: Dipnot ve sonnot. Dipnot aynı sayfanın altında, ayırıcı çizginin altında durur (1). Sonnotlar belgenin ya da bölümün sonunda toplanır ve varsayılan olarak küçük Roma rakamlarıyla (i, ii) numaralanır.

Dipnot eklemek için imleç açıklamanın bağlanacağı sözcüğün sonuna konur ve Başvurular → Dipnot Ekle seçilir. Word metne üst simge bir numara koyar ve imleci sayfanın altındaki dipnot alanına taşır (Şekil 3.1.44). Araya yeni bir dipnot eklenince sonraki dipnotların numaraları kendiliğinden kayar. Metin kayıp dipnotun bağlı olduğu sözcük başka sayfaya geçerse dipnot da onunla birlikte o sayfanın altına taşınır.

Word kaynak yönetimini de üstlenebilir. Başvurular → Alıntı Ekle → Yeni Kaynak Ekle ile bir kaynağın yazarı, başlığı, yılı ve yayıncısı bir forma girilir. Bu kaynak metinde istenen yerlere atıf olarak eklenir. **Atıf stili** (APA, IEEE, MLA, Chicago gibi) bu atıfların ve kaynakça satırlarının biçimini belirler. Stil değiştirilince bütün atıflar ve kaynakça yeni biçime geçer. Başvurular → Kaynakça komutu, atıf yapılmış kaynaklardan otomatik bir liste üretir.

Örnek 3.1.24

Bir laboratuvar raporunda iki kaynağa atıf yapılıyor: bir ders kitabı ve bir veri sayfası. Hoca önce APA istiyor, sonra IEEE biçimine geçilmesini söylüyor.

1. Her kaynak bir kez Yeni Kaynak Ekle formuyla girilir. Atıf noktalarında Alıntı Ekle listesinden ilgili kaynak seçilir.
2. Atıf stili APA iken metindeki atıflar yazar ve yıl biçimindedir, örneğin “(Patterson, 2020)”.
3. Raporun sonunda Kaynakça eklenir; iki kaynak APA kurallarıyla sıralanır.
4. Başvurular → Stil listesinden IEEE seçilir. Metindeki atıflar köşeli ayraçlı numaralara, örneğin “[1]”e dönüşür. Kaynakça güncellenince satırlar da numaralı IEEE biçimine geçer.

Elle yazılmış atıflarda bu geçiş, her atfın ve her kaynakça satırının tek tek yeniden yazılması demektir. Yine de Word'ün ürettiği satırlar, forma girilen bilgi kadar doğrudur. Yazar adlarının, yılların ve sayfa numaralarının doğruluğunu belgenin yazarı denetler.

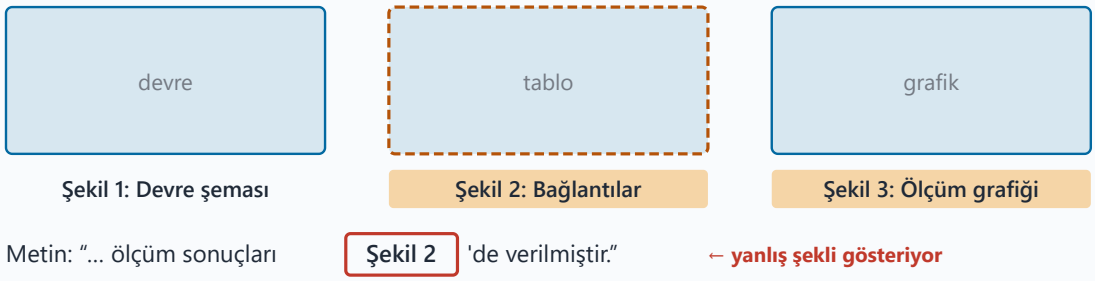
3.1.9.4 Resim yazıları ve çapraz başvurular

Tanım 3.1.26

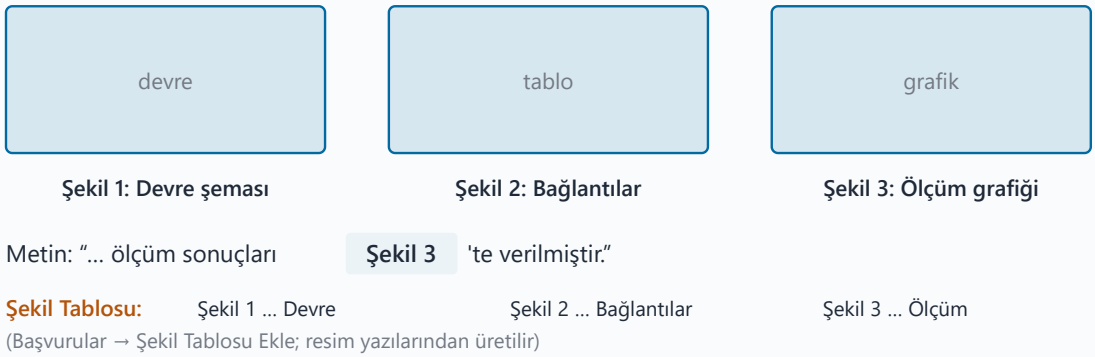
Resim yazısı (*caption*), bir şekli, tabloyu ya da denklemi “Şekil 1: Devre şeması” gibi numaralı bir başlıkla tanımlayan etikettir; Başvurular → Resim Yazısı Ekle ile eklenir. **Çapraz başvuru** (*cross-reference*), metinden belgenin başka bir yerindeki başlığa, şekle, tabloya ya da sayfaya kurulan dinamik bağlantıdır. **Şekil tablosu** (*table of figures*) ise resim yazılarından üretilen şekil listesidir.

Resim yazısındaki numara da bir alandır. Belgeye yeni bir şekil eklenince Word sonraki şekillerin numaralarını kaydırır. Metindeki “Şekil 3'te görüldüğü gibi” ifadesi elle yazılmışsa bu kaymadan haberi olmaz ve yanlış şekli göstermeye başlar. Çapraz başvuru (Başvurular → Çapraz Başvuru) ise numarayı

resim yazısından alır ve güncellenince yeni numarayı gösterir.



Şekil 3.1.45: İki şeklin arasına yeni bir tablo görseli ve resim yazısı (Şekil 2: Bağlantılar) eklendi. Grafiğin numarası Şekil 3 oldu, ama metindeki çapraz başvuru henüz güncellenmediği için "Şekil 2" yazıyor ve yanlış şekli gösteriyor.



Şekil 3.1.46: `Ctrl+A` ve `F9` sonrası çapraz başvuru "Şekil 3" oldu; ek de uyumlu biçimde "'te" yazıldı. Başvurular → Şekil Tablosu Ekle ile resim yazılarından şekil tablosu üretildi.

Örnek 3.1.25

Belgede iki şekil var: "Şekil 1: Devre şeması" ve "Şekil 2: Ölçüm grafiği". Metindeki "ölçüm sonuçları Şekil 2'de verilmiştir" ifadesindeki "Şekil 2" grafiğe çapraz başvurudur. İki şeklin arasına yeni bir şekil ekleniyor (Şekil 3.1.45 ve Şekil 3.1.46):

Adım	İşlem	Resim yazıları	Metindeki başvuru
1	başlangıç	1 Devre · 2 Ölçüm	Şekil 2'de (doğru)
2	araya "Bağlantılar" şekli ve resim yazısı eklenir	1 Devre · 2 Bağlantılar · 3 Ölçüm	Şekil 2'de (yanlış şekli gösteriyor)
3	`Ctrl+A`, `F9` (alanları güncelleştir)	aynı	Şekil 3'de (numara doğru, ek uyumsuz)
4	ek elle düzeltilir	aynı	Şekil 3'te

Adım	İşlem	Resim yazıları	Metindeki başvuru
5	Başvurular → Şekil Tablosu Ekle	aynı	aynı; şekil tablosunda üç satır

Adım 3'te çapraz başvuru yalnızca numarayı günceller. Arkasındaki "'de" eki alanın parçası değil, elle yazılmış sıradan metindir; Word Türkçe ses uyumunu bilmez. "3" sayısı "'te" eki aldığı için ek elle düzeltilir (adım 4). Bu nedenle alanlar güncellendikten sonra çapraz başvuruların çevresi bir kez okunur.

Sık yapılan hata

"Şekil 3" gibi numaraları resim yazısında ya da metinde elle yazmak. Bir şekil eklenince ya da silinince bütün numaralar ve göndermeler tek tek düzeltilir; biri atlanırsa okuyucu yanlış şekle bakar. Resim yazısı ve çapraz başvuru alan olduğu için tek bir `F9` ile hepsi güncellenir.

3.1.10 Adres mektup birleştirme

Bölüm sekreterliği 120 öğrenciye final notlarını bildiren birer mektup gönderecek. Metin herkes için aynı, yalnızca ad, bölüm ve not değişiyor. 120 mektubu tek tek yazmak hem saatler alır hem de yanlış öğrenciye yanlış not yazma riski taşır. Adres mektup birleştirme bu işi bir ana belge ve bir listeyle çözer.

3.1.10.1 Ana belge, veri kaynağı ve alanlar

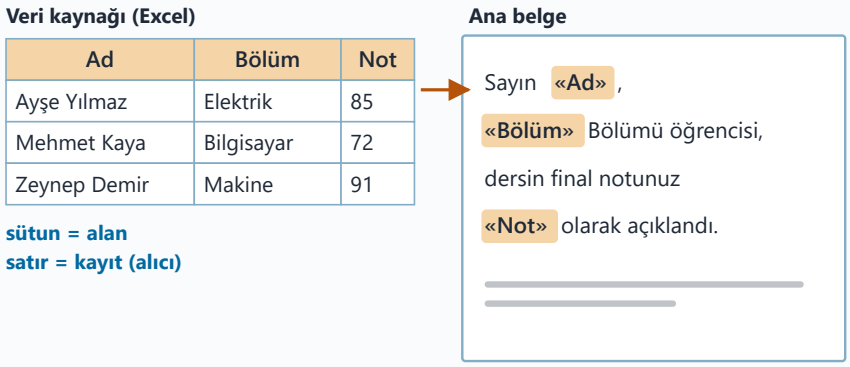
Tanım 3.1.27

Adres mektup birleştirme (*mail merge*), bir **ana belgeyi** bir **veri kaynağıyla** birleştirip her alıcı için kişiselleştirilmiş bir çıktı üreten Postalar sekmesi özelliğidir. Ana belge herkese ortak metni ve yerleşimi içerir. Veri kaynağı (örneğin bir Excel tablosu) alıcıların bilgilerini tutar: her **sütun** bir **alan**, her **satır** bir **kayıt**, yani bir alıcıdır. **Birleştirme alanı** (*merge field*), ana belgede değerin geleceği yeri gösteren «Ad» gibi yer tutucudur.

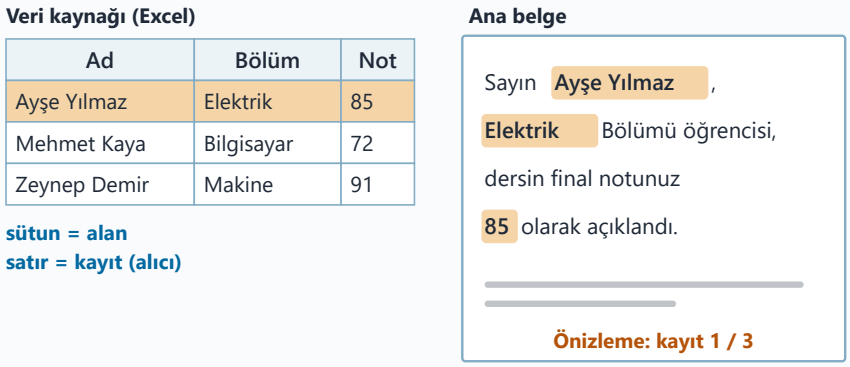
Birleştirme beş aşamada ilerler: ana belge hazırlanır, alıcı listesi seçilir, birleştirme alanları eklenir, sonuçlar önizlenir ve son olarak belgeler ya da e-postalar üretilir. Önizleme aşaması atlanmamalıdır. Sütun adındaki bir yazım hatası ya da boş bir hücre, yüzlerce belgede birden yanlışlık demektir.

Veri kaynağının ilk satırı *alan adlarıdır*: Ad, Bölüm, Not. Bu satır bir alıcı değildir ve mektup üretmez. Bu yüzden ilk satırı başlık olan n satırlık bir listeden $n - 1$ mektup çıkar. Alan adlarının kısa, boşluksuz ve anlamlı olması, ana belgede doğru alanı seçmeyi kolaylaştırır.

3.1.10.2 Birleştirme adım adım



Şekil 3.1.47: Solda veri kaynağı (Excel): Ad, Bölüm, Not sütunları ve üç kayıt (Ayşe Yılmaz, Elektrik, 85; Mehmet Kaya, Bilgisayar, 72; Zeynep Demir, Makine, 91). Sağda ana belge: boşluklara «Ad», «Bölüm» ve «Not» alanları kondu.



Şekil 3.1.48: Sonuçları Önizle, kayıt 1 / 3: alanların yerine 1. kaydın değerleri geldi: "Sayın Ayşe Yılmaz, Elektrik Bölümü öğrencisi, dersin final notunuz 85 olarak açıklandı."

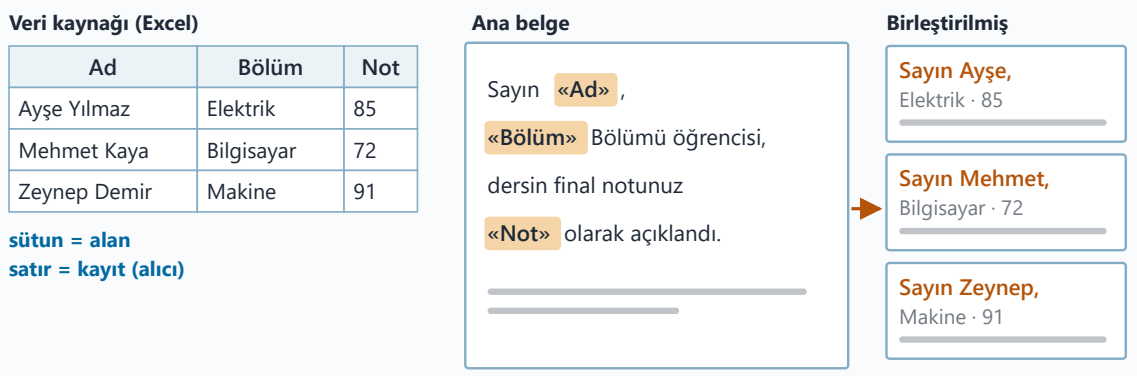
Örnek 3.1.26

Ana belge: "Sayın _____, _____ Bölümü öğrencisi, dersin final notunuz _____ olarak açıklandı." Veri kaynağı Şekil 3.1.47'deki üç kayıtlı Excel tablosudur. Adımlar ve her adımda ana belgenin görünüşü şöyledir:

Adım	Postalar sekmesinde komut	Ana belgede görünen
1	(ana belge yazılır)	Sayın _____, _____ Bölümü … _____ …
2	Alıcıları Seç → Var Olan Listeyi Kullan (Excel dosyası)	aynı; 3 alıcı bağlandı
3	Birleştirme Alanı Ekle: Ad, Bölüm, Not	Sayın «Ad», «Bölüm» Bölümü … «Not» …

Adım	Postalar sekmesinde komut	Ana belgede görünen
4	Sonuçları Önizle, kayıt 1	Sayın Ayşe Yılmaz, Elektrik Bölümü … 85 …
5	sonraki kayıt (2)	Sayın Mehmet Kaya, Bilgisayar Bölümü … 72 …
6	sonraki kayıt (3)	Sayın Zeynep Demir, Makine Bölümü … 91 …
7	Son ve Birleştir → Tek Tek Belgeleri Düzenle	3 mektupluk yeni bir belge oluşur

Excel dosyasında 4 satır vardır: 1 başlık satırı ve 3 kayıt. Çıktı 4 − 1 = 3 mektuptur. Önizleme sırasında ana belgenin ortak metni hiç değişmez; değişen yalnızca alanların yerine gelen değerlerdir. Son ve Birleştir menüsü aynı birleştirmeyi doğrudan yazıcıya ya da e-posta olarak göndermeye de izin verir. Postalar sekmesindeki Etiketler ve Zarflar komutları aynı veri kaynağından adres etiketi ve zarf üretir.



Şekil 3.1.49: Son ve Birleştir: her kayıt için bir mektup üretildi, toplam 3 belge (Ayşe – Elektrik · 85, Mehmet – Bilgisayar · 72, Zeynep – Makine · 91).

Sık yapılan hata

Excel listesinde başlık satırı olmadan birleştirmeye başlamak. Word ilk satırı alan adı sayar; ilk öğrencinin adı "Ayşe Yılmaz" bir alan adına dönüşür ve o öğrenciye mektup üretilmez. Veri kaynağının ilk satırı her zaman alan adlarıdır. Bir başka sık hata, sayıların Excel'deki biçimiyle gelmemesidir. Örneğin 87,5 gibi bir not mektupta farklı basamakla görünebilir. Önizleme bu tür sorunları ilk kayıtlarda ortaya çıkarır.

3.1.11 Kalite ve paylaşım

Bir belge yazıldıktan ve gözden geçirildikten sonra son bir denetimden geçer. Belge onu okuyacak herkes için erişilebilir mi? Yazdırıldığında ya da PDF'e aktarıldığında düzeni korunuyor mu? İçinde unutulmuş açıklama, gizli değişiklik ya da kişisel bilgi var mı? Bu kesim bu soruları ve makroların taşıdığı güvenlik riskini ele alır.

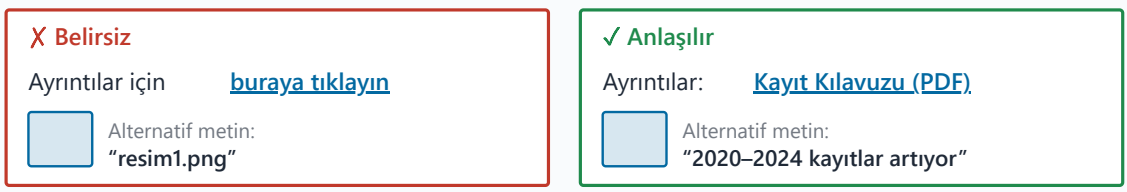
3.1.11.1 Erişilebilir belge

Tanım 3.1.28

Erişilebilir belge, görme, işitme ya da hareket engeli olan kişilerin de yardımcı teknolojilerle (örneğin metni sesli okuyan **ekran okuyucularla**) okuyup gezinebildiği belgedir. **Erişilebilirlik denetleyicisi** (*accessibility checker*; Gözden Geçir → Erişilebilirliği Denetle) belgedeki olası sorunları listeler ve düzeltme önerir.

Bir ekran okuyucu belgeyi yukarıdan aşağı sesli okur. Kullanıcı başlıktan başlığa atlayabilir, bağlantıları listeleyebilir, tabloda hücre hücre gezinebilir. Bu olanakların hepsi belgenin yapısına dayanır. Görünüşü doğru ama yapısı eksik bir belge, ekran okuyucu için anlamsız bir metin yığınıdır. Erişilebilirlik ilkeleri bu yüzden notun başından beri anlatılan iyi alışkanlıklarla örtüşür:

- **Başlık stilleri** belgenin yapısını ekran okuyucuya bildirir; kullanıcı başlıklar arasında atlayabilir.
- **Alternatif metin** anlamlı görsellerin ne anlattığını açıklar. Yalnızca süs amaçlı görseller "dekoratif" olarak işaretlenir ve okunmaz.
- **Bağlantı metni** hedefi anlatır. "buraya tıklayın" bağlamından koparılınca hiçbir şey söylemez; "Kayıt Kılavuzu (PDF)" ise hedefi adlandırır.
- **Tablo yapısı** basit tutulur ve başlık satırı belirgin olur. Boş hücrelerle ya da birleştirilmiş hücrelerle yapılan düzenler, hücreler arası okumayı karıştırır.
- **Renk ve kontrast**: anlam yalnızca renkle verilmez. "Kırmızı satırlar başarısız" yerine satırlara ayrıca bir "başarısız" sözcüğü ya da simgesi eklenir. Metin ile arka plan arasında yeterli karşıtlık bırakılır.



Şekil 3.1.50: Belirsiz ve anlaşılır kullanım. Solda "Ayrıntılar için buraya tıklayın" bağlantısı ve "resim1.png" alternatif metni. Sağda "Ayrıntılar: Kayıt Kılavuzu (PDF)" bağlantısı ve "2020–2024 arasında kayıtlar artıyor" alternatif metni.

Şekil 3.1.50 iki küçük farkı gösterir. İyi bir alternatif metin, görseli görmeyen birine görselin *verdiği bilgiyi* aktarır. Bir grafik için "grafik" yazmak yeterli değildir. Grafiğin gösterdiği eğilim ("2020–2024 arasında kayıtlar artıyor") yazılır. Ayrıntılı sayılar zaten metinde ya da bir tabloda verilmişse alternatif metin kısa tutulabilir.

Sık yapılan hata

Erişilebilirliği yalnızca engelli okuyucuları ilgilendiren ek bir iş saymak. Başlık stilleri, anlamlı bağlantı metinleri ve basit tablolar her okuyucunun işini kolaylaştırır. PDF'e aktarılan belgede yer imlerinin oluşması, arama motorlarının içeriği doğru dizinlemesi ve belgenin telefonda okunabilmesi de aynı yapıya dayanır.

3.1.11.2 Yazdırma ve PDF

Dosya → Yazdır ekranı solda yazdırma ayarlarını, sağda **baskı önizlemeyi** gösterir. Önizleme, belgenin kâğıda nasıl çıkacağının son kontrolüdür. Taşan bir tablo, araya düşmüş boş bir sayfa ya da kenar boşluğuna yapışmış bir resim burada fark edilir.



Şekil 3.1.51: Dosya → Yazdır ekranı. Solda Kopya: 1, Yazıcı: Microsoft Print to PDF, Tüm Sayfaları Yazdır, Sayfalar: 1-3, 5, İki Taraflı Yazdır, Dikey Yönlendirme · A4; sağda 6 sayfalık belgenin baskı önizlemesi.

Şekil 3.1.51'deki ayarların anlamı şöyledir. **Kopya** her sayfanın kaç kez basılacağıdır. **Sayfalar** kutusu yazdırılacak sayfaları belirler: tire bir aralığı, virgül ayrı sayfaları gösterir. Örneğin "1-3, 5", 1, 2, 3 ve 5. sayfalardır; yani 4 sayfa basılır. Tüm Sayfaları Yazdır menüsü seçili metni ya da yalnızca geçerli sayfayı basmaya da izin verir. **İki Taraflı Yazdır**, destekleyen yazıcılarda kâğıdın iki yüzünü de kullanır ve kâğıt tüketimini yaklaşık yarıya indirir.

Örnek 3.1.27

Şekil 3.1.51'deki 6 sayfalık belgede Sayfalar kutusuna "1-3, 5" yazılıyor. Tek taraflı ve iki taraflı yazdırmada kaç sayfa çıkar ve kaç yaprak kâğıt kullanılır?

1. **Basılacak sayfalar.** "1-3" üç sayfadır (1, 2, 3), "5" bir sayfadır. Toplam 4 sayfa; 4. ve 6. sayfa basılmaz.
2. **Tek taraflı.** Her sayfa ayrı bir yaprağa basılır: 4 yaprak.

3. **İki taraflı.** Her yaprağa iki sayfa sığar: 4 ÷ 2 = 2 yaprak. Sayfa sayısı tek olsaydı yukarı yuvarlanırdı; örneğin 5 sayfa 3 yaprak tutardı.
4. **Kopya 3 olsaydı.** İki taraflı yazdırmada 3 × 2 = 6 yaprak kullanılırdı.

Belgeyi değiştirilmeden paylaşmanın standart yolu PDF'tir. **Dışa aktarma** (Dosya → Dışa Aktar → PDF/XPS Belgesi Oluştur ya da Farklı Kaydet → kayıt türü PDF) düzeni koruyan bir kopya üretir. Bu kopya, Word'ü olmayan bir cihazda da aynı görünür. Seçenekler penceresinde başlıklardan PDF yer imleri oluşturulması istenebilir. Böylece PDF okuyucunun kenar çubuğunda tıklanabilir bir içindekiler belirir. Bu da başlık stilleriyle yazılmış belgenin bir başka kazancıdır.

Paylaşmadan önce belgede gizli kalmış içerik de denetlenir. Açıklamalar, bekleyen değişiklikler ve **belge özellikleri** (yazar adı, başlık, etiketler gibi meta veriler) belgeyle birlikte gider. Dosya → Bilgi → Sorunları Denetle → **Belgeyi İncele** komutu bunları listeler ve istenirse topluca kaldırır. Yazar alanında başka birinin adı görünen bir ödev, şablon yerine eski bir dosyanın kullanıldığını ele verir.

Sık yapılan hata

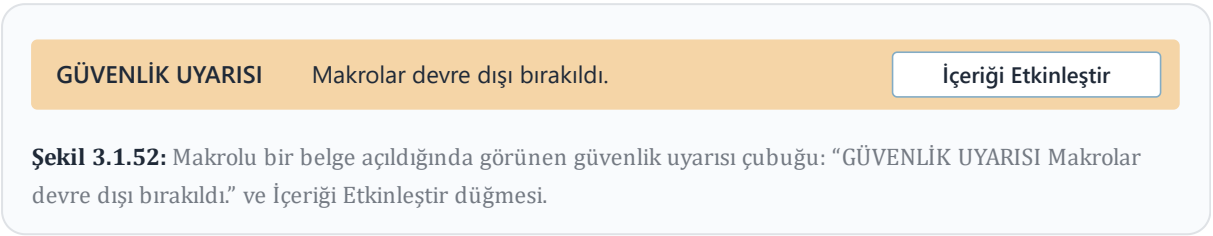
Baskı önizlemeye bakmadan yazdırmak ya da PDF'e aktarmak. Son dakikada eklenen bir paragraf tabloyu bir sonraki sayfaya itebilir. Belgenin sonunda boş bir sayfa kalabilir ya da içindekiler eski numaraları gösterebilir. Önizleme birkaç saniye sürer, yeniden basmaksa kâğıt ve zaman harcar.

3.1.11.3 Makrolar ve güvenlik

Tanım 3.1.29

Makro (*macro*), tekrarlanan bir komut dizisini tek komutla çalıştırılabilir hâle getiren küçük programdır. Kullanıcının eylemleri kaydedilerek (Görünüm → Makrolar → Makro Kaydet) ya da **VBA** (*Visual Basic for Applications*) dilinde kod yazılarak oluşturulur. Makro içeren belgeler `.docm`, makro içeren şablonlar `.dotm` uzantısıyla saklanır.

Makrolar gerçek işlerde zaman kazandırır. Örneğin yüzlerce tablonun hepsine aynı biçimi uygulamak ya da belgedeki bütün resimleri aynı genişliğe getirmek tek bir makroyla yapılabilir. Ancak bir makro bir programdır. Belge açıldığında çalışan bir makro dosya silebilir, başka dosyalar indirebilir ya da bilgisayara zararlı yazılım kurabilir. Saldırganlar bu yüzden zararlı makroları e-posta ekindeki “fatura” ya da “kargo bilgisi” belgelerine gizler.



Şekil 3.1.52: Makrolu bir belge açıldığında görünen güvenlik uyarısı çubuğu: "GÜVENLİK UYARISI Makrolar devre dışı bırakıldı." ve İçeriği Etkinleştir düğmesi.

Word makrolu bir belgeyi açarken makroları varsayılan olarak çalıştırmaz ve Şekil 3.1.52'deki uyarıyı gösterir. İçeriği Etkinleştir düğmesine basmak, belgedeki makrolara bilgisayarda çalışma izni vermek demektir. İnternetten indirilen ya da e-postayla gelen dosyalarda güncel Microsoft 365 sürümleri makroları tamamen engelleyebilir. Bu durumda düğme yerine bir güvenlik riski uyarısı görünür. Kural basittir: kaynağı bilinmeyen ya da beklenmeyen bir belgede içerik etkinleştirilmez. Bilinen bir kişiden gelse bile beklenmedik bir makrolu belge için göndericiye ayrı bir kanaldan sorulur.

3.1.11.4 Belge hazırlama iş akışı ve sık yapılan hatalar

Uzun bir belge altı aşamada hazırlanır: amaç ve hedef kitle belirlenir, taslak yapı (başlıklar) kurulur, içerik yazılır, stiller ve görseller düzenlenir, belge gözden geçirilir ve yayımlanır. Sıranın önemli bir mantığı vardır: önce yapı ve içerik kurulur, ayrıntılı biçimlendirme sona bırakılır. İçerik henüz değişirken yazı tipleriyle uğraşmak boşa emektir. Stiller kullanılıyorsa biçim zaten tek yerden, en sonda ve birkaç dakikada ayarlanır.

Notun bütününe yayılmış sık hatalar Tablo 3.1.15'te bir arada verilmiştir. Hepsinin çözümü aynı ilkeye dayanır: Word'e görünüşü değil yapıyı anlatmak.

Hata	Sonucu	Doğru yol
yeni sayfa için art arda Enter	metin eklenince boş satırlar kayar, sayfa düzeni bozulur	`Ctrl+Enter` ya da Düzen → Kesmeler
boşluk tuşuyla hizalama	yazı tipi değişince hizalar kayar	sekme, girinti ya da tablo
başlığı yalnızca kalın ve büyük yazmak	gezinti bölmesi ve içindekiler başlığı görmez	Başlık 1 / Başlık 2 stili
numaraları elle yazmak (1., 1.1, Şekil 3)	ekleme yapılınca bütün numaralar elle düzeltilir	liste, çok düzeyli liste, resim yazısı
içindekileri ve sayfa numarasını elle yazmak	sayfalar kayınca yanlış numaralar kalır	otomatik içindekiler, sayfa numarası alanı
satır sonunda Enter'a basmak	her satır ayrı paragraf olur, iki yana yaslama bozulur	Word satırı kendisi kaydırır; Enter yalnızca paragraf sonunda

Tablo 3.1.15: Sık yapılan Word hataları, sonuçları ve doğru yollar.

Yapı doğru kurulursa görünüş tek yerden değiştirilebilir: başlıkların görünüşü bir stil tanımından, sayfa düzeni bölüm ayarlarından, numaralar alanlardan gelir. Belge yüz sayfaya da çıksa, son dakikada istenen bir değişiklik birkaç dakikada yapılır.

3.1.12 Özet

- Word bir kelime işlemcidir; belgeyi `.docx` dosyasında saklar. Komutlar şeritte sekme → grup → komut düzeninde durur; tablo ya da resim seçilince bağlamsal sekmeler belirir.
- Kaydet açık dosyayı günceller; Farklı Kaydet yeni bir dosya oluşturur ve pencere artık o dosyayla çalışır. Otomatik Kaydet yalnızca buluttaki dosyada çalışır; Otomatik Kurtarma kaydetmenin yerine geçmez.
- Şablon (`.dotx`) açıldığında ondan türetilmiş adsız bir kopya açılır; şablon değişmez.
- Belgenin yapı düzeyleri karakter, sözcük, paragraf, bölüm ve belgedir. Karakter biçimi seçime, paragraf biçimi imlecin bulunduğu paragrafa, sayfa ayarları imlecin bulunduğu bölüme uygulanır.
- Kes taşır, Kopyala çoğaltır; pano yapıştırmadan sonra içeriğini korur. Tümünü Değiştir sözcük içlerini de değiştirir; "Yalnızca tam sözcükleri bul" ya da Sonrakini Bul ile önizleme gerekir.
- Gizli işaretler (¶, ·, →) ¶ düğmesiyle görünür; Bul ve Değiştir'de `^p` , `^t` , `^l` , `^s` , `^m` kodlarıyla aranır. `^p^p` → `^p` sonuç 0 olana kadar tekrarlanır.
- Stil adı olan bir biçim paketidir; tanımı değişince onu kullanan bütün öğeler güncellenir. Doğrudan biçimlendirme stilin önüne geçer; `Ctrl+Boşluk` elle verilen karakter biçimini kaldırır.
- Sayfa sonu bölüm oluşturmaz; bölüm sonu oluşturur. Tek bir sayfayı yatay yapmak için önüne ve arkasına birer bölüm sonu konur.
- Yeni bölümün üstbilgisi Öncekine Bağla ile öncekine bağlı başlar; farklı içerik yazmadan önce bağ koparılır. Sayfa numarası biçimi ve başlangıcı bölüm bazında ayarlanır.
- Başlık stilleri içindekiler tablosunun, resim yazıları şekil tablosunun ve çapraz başvuruların kaynağıdır. Bunlar alandır; `F9` ya da Tabloyu Güncelleştir ile yenilenir.
- Açıklama metni değiştirmez; değişiklik izleme her düzenlemeyi kabul ya da ret bekleyen bir öneri olarak kaydeder. İşaretlemeyi gizlemek değişiklikleri kabul etmek değildir.
- Adres mektup birleştirmede ana belge ortak metni, veri kaynağı alıcıları tutar; başlık satırı mektup üretmez.
- Paylaşmadan önce baskı önizleme, erişilebilirlik denetimi ve Belgeyi İncele ile gizli içerik denetlenir; kaynağı bilinmeyen belgede makro etkinleştirilmez.

3.1.13 Kavram listesi

belge *document* · şerit *ribbon* · sekme *tab* · bağlamsal sekme *contextual tab* · gezinti bölmesi *navigation pane* · görünüm *view* · kaydet / farklı kaydet *save / save as* · Otomatik Kaydet *AutoSave* · Otomatik Kurtarma *AutoRecover* · sürüm geçmişi *version history* · şablon *template* · imleç *cursor* · seçim *selection* · pano *clipboard* · Biçim Boyacısı *Format Painter* · Bul ve Değiştir *Find and Replace* · biçimlendirme işaretleri *formatting marks*

bölünmez boşluk *non-breaking space* · yer işareti *bookmark* · köprü *hyperlink* · yazı tipi *font* · punto *point* · üst simge / alt simge *superscript / subscript* · hizalama *alignment* · iki yana yaslama *justify* · girinti *indent* · asılı girinti *hanging indent* · satır aralığı *line spacing* · paragraf aralığı *paragraph spacing* · çok düzeyli liste *multilevel list* · stil *style* · tema *theme* · doğrudan biçimlendirme *direct formatting* · kenar boşluğu *margin* · yönlendirme *orientation* · sayfa sonu *page break* · bölüm sonu *section break* · üstbilgi / altbilgi *header / footer* · tablo, hücre, başlık satırı *table, cell, header row* · metin kaydırma *text wrapping* · alternatif metin *alt text* · filigran *watermark* · açıklama *comment* · değişiklik izleme *track changes* · işaretleme *markup* · içindekiler (tablosu) *table of contents* · dipnot / sonnot *footnote / endnote* · alıntı, kaynakça *citation, bibliography* · resim yazısı *caption* · çapraz başvuru *cross-reference* · şekil tablosu *table of figures* · adres mektup birleştirme *mail merge* · birleştirme alanı *merge field* · erişilebilirlik denetleyicisi *accessibility checker* · baskı önizleme *print preview* · makro *macro*

3.1.14 Kendini Dene

1. Diskte `tasarim.docx` açılıyor. Sırayla: A paragrafı yazılır, `Ctrl+S`; B paragrafı yazılır, Farklı Kaydet → `tasarim_son.docx`; C paragrafı yazılır ve Word kaydetmeden kapatılır. Sonunda iki dosyada hangi paragraflar vardır?

   Cevap

   `tasarim.docx`: yalnızca A, çünkü Farklı Kaydet'ten sonra bu dosyaya hiç yazılmadı. `tasarim_son.docx`: A ve B. C hiç kaydedilmediği için kayboldu; Farklı Kaydet'ten sonra pencerede açık olan dosya `tasarim_son.docx` idi.

2. Tek satırlık belge: "Bu not sınavdan önce okunur." İmleç satırın başında. Sırayla `Ctrl+→`, `Ctrl+Shift+→`, `Ctrl+Shift+→`, `Ctrl+I` basılıyor. Hangi sözcükler italik olur, imleç nerede kalır?

   Cevap

   `Ctrl+→` imleci "not" başına götürür. İki `Ctrl+Shift+→` önce "not ", sonra "not sınavdan " seçer. `Ctrl+I` seçime uygulanır: "not sınavdan" italik olur. İmleç "önce" sözcüğünün başındadır ve seçim sürer.

3. "Sonuç" paragrafından sonra art arda dört boş paragraf, ardından "Ekler" paragrafı var. Aranan `^p^p`, Yeni değer `^p` ile Tümünü Değiştir kaç kez çalıştırılır ve her seferinde kaç değiştirme yapı-

lır?

Cevap

“Sonuç.” sonrasında beş ¶ art arda gelir. 1. çalıştırma: soldan ikişerli eşleşir, 2 değiştirme; 3 ¶ kalır (iki boş paragraf). 2. çalıştırma: 1 değiştirme; 2 ¶ kalır (bir boş paragraf). 3. çalıştırma: 1 değiştirme; tek ¶ kalır. 4. çalıştırma 0 sonuç verir ve işin bittiğini gösterir.

4. Amaç, Veri ve Kapsam başlıkları Başlık 2 stilinde (12 pt, mavi). Veri şeritten turuncu yapılıyor. Sonra Başlık 2'nin tanımı Değiştir… ile 13 pt ve mor yapılıyor. Üç başlık nasıl görünür? Ardından Veri seçilip `Ctrl+Boşluk`'a basılırsa ne olur?

Cevap

Amaç ve Kapsam: 13 pt, mor. Veri: 13 pt ama turuncu, çünkü renk ona elle verildi ve doğrudan biçim stilin önüne geçer; boyut elle verilmediği için tanımdan gelir. `Ctrl+Boşluk` elle verilen rengi siler; Veri de 13 pt ve mor olur.

5. 10 sayfalık dikey bir belgede 4. ve 5. sayfalara yayılan geniş bir tablo ile 9. sayfadaki bir şema yatay olmalı. Kaç bölüm sonu gerekir ve belge kaç bölümden oluşur?

Cevap

4. sayfanın önüne, 5. sayfanın arkasına, 9. sayfanın önüne ve arkasına birer bölüm sonu: 4 bölüm sonu. k bölüm sonu k + 1 bölüm yapar: s.1–3, s.4–5, s.6–8, s.9 ve s.10, yani 5 bölüm. Yatay yapılan bölümler 2. ve 4. bölümlerdir.

6. Bir tezde ilk dört sayfa Kapak, Onay, Özet ve İçindekiler; bunlar Bölüm 1'dir ve küçük Roma rakamlıdır, İlk Sayfa Farklı açıktır. Bölüm 2 Giriş sayfasından 1 diye başlar. Özet sayfasında ve belgenin 7. fiziksel sayfasında hangi numara görünür?

Cevap

Bölüm 1'de kapak sayılır ama numarası görünmez: Kapak (i, görünmez), Onay ii, Özet iii, İçindekiler iv. Özet'te “iii” görünür. Bölüm 2 5. fiziksel sayfadan 1 diye başlar; 7. fiziksel sayfa 3 numaradır.

7. Değişiklikleri İzle açıkken “Laboratuvar raporu Cuma günü teslim edilir.” cümlesinde “Cuma” silinip “Pazartesi” yazılıyor, “teslim” önüne “öğlene kadar ” ekleniyor. Yazar silmeyi ve “Pazartesi” eklemesini kabul ediyor, “öğlene kadar ” eklemesini reddediyor. Son cümle nedir? Yazar hiçbir şeyi

kabul etmeden Basit İşaretleme'de belgeyi gönderseydi alıcı ne görebilirdi?

Cevap

Son cümle: “Laboratuvar raporu Pazartesi günü teslim edilir.” Reddedilen ekleme metinden çıkar. Kabul edilmeden gönderilseydi üç değişiklik bekliyor olurdu; alıcı Tüm İşaretlemeler'e geçerek silinen “Cuma”yı ve eklenen ifadeleri görebilirdi. Gizlemek kabul etmek değildir.

8. Adres mektup birleştirmede kullanılan Excel dosyasında 31 dolu satır var; ilk satır alan adlarıdır. İki öğrencinin Not hücresi boş. Son ve Birleştir kaç mektup üretir ve boş hücreler ne olur?

Cevap

Başlık satırı mektup üretmez: 31 − 1 = 30 kayıt, 30 mektup. Boş hücreli kayıtlar da mektup üretir; «Not» alanının yeri boş kalır. Bu yüzden sonuçlar üretilmeden önce kayıtlar önizlenir ve eksik veri tamamlanır.

9. 10 sayfalık bir belgenin Yazdır ekranında Sayfalar kutusuna “2-4, 7, 9-10”, Kopya kutusuna 2 yazılıyor ve İki Taraflı Yazdır seçiliyor. Kaç yaprak kâğıt kullanılır?

Cevap

Basılacak sayfalar 2, 3, 4, 7, 9 ve 10'dur: 6 sayfa. İki taraflı her yaprağa 2 sayfa sığar: bir kopya 6 ÷ 2 = 3 yaprak. İki kopya 2 × 3 = 6 yaprak tutar.

10. A4 sayfada sol kenar boşluğu (cilt payıyla) 3,5 cm, diğer kenar boşlukları 2,5 cm. Metin alanı kaç milimetre genişliktedir? Satır aralığı “Tam olarak 24 nk” ise bir sayfaya en fazla kaç satır sığar?

Cevap

Genişlik 210 − 35 − 25 = 150 mm, yükseklik 297 − 25 − 25 = 247 mm. 24 nk = 24 × 25,4 ÷ 72 ≈ 8,47 mm. 247 ÷ 8,47 ≈ 29,2; aşağı yuvarlanır: 29 satır.

3.1.15 İleri okuma

- Microsoft Destek, *Word yardım ve öğrenme* (support.microsoft.com/tr-tr/word): Word'ün her özelliği için Türkçe arayüz adlarıyla yazılmış kısa makaleler ve videolar sunar; bu notta geçen komutların güncel yerini doğrulamak için en güvenilir kaynaktır.
- Microsoft Destek, *Word belgelerinizi engelli kişiler için erişilebilir hale getirme*: alternatif metin, başlık yapısı, tablo ve renk kullanımını Erişilebilirlik Denetleyicisi ile birlikte adım adım anlatır.
- M. Butterick, *Practical Typography* (practicaltypography.com): bir belgenin okunaklı olması için yazı tipi, satır aralığı, kenar boşluğu ve hizalama seçimlerini gerekçeleriyle anlatan, ücretsiz okunabilen çevrim içi kitap.

3.1.16 Kaynaklar

1. Microsoft Support, *Basic tasks in Word*.
2. Microsoft Support, *Track changes in Word*.
3. Microsoft Support, *Templates, themes and Word styles*.
4. Microsoft Destek, *Word belgelerinizi engelli kişiler için erişilebilir hale getirme*.
5. M. Butterick, *Practical Typography*, çevrim içi kitap, practicaltypography.com.

KISIM I · BÖLÜM 3.2

Microsoft Excel

Bir laboratuvar deneyinin ölçümleri, bir dersin not listesi, bir projenin malzeme bütçesi ya da bir dükkânın stok kaydı aynı biçimde düzenlenebilir: satırlara kayıtlar, sütunlara özellikler yazılır. Elektronik tablo programları bu düzeni hesapla birleştirir. Bir hücreye yazılan formül, başka hücrelerdeki değerlerden sonuç üretir ve o değerler değişince sonucu kendiliğinden günceller. Bu not, Microsoft Excel'in çalışma kitabı, hücre ve aralık gibi temel kavramlarından başlar. Ardından formülleri, hücre başvurularını, sık kullanılan işlevleri, veri düzenleme ve analiz araçlarını ve grafikleri anlatır. Notta Microsoft 365'in Türkçe arayüzündeki sekme ve komut adları ile Türkçe işlev adları kullanılır. Bağımsız değişkenler, Türkçe Excel'de olduğu gibi noktalı virgülle ayrılır. Her formülün sonucu adım adım hesaplanarak gösterilir.

ÖĞRENME ÇIKTILARI

Çalışma kitabı

Çalışma kitabı, çalışma sayfası, hücre, satır, sütun ve aralık kavramlarını doğru kullanmak; hücreye uygun veri tipiyle veri girmek ve sayı biçiminin yalnızca görünümü değiştirdiğini açıklamak.

Formüller

İşleç önceliğine göre formül sonuçlarını hesaplamak; göreli, mutlak ve karma başvuruların kopyalanınca nasıl değiştiğini önceden bulmak; temel, mantıksal, koşullu ve arama işlevleriyle hesap yapmak; hata değerlerinin nedenini bulup düzeltmek.

Veri düzenleme

Tablo, sıralama, filtreleme, ALTTOPLAM, koşullu biçimlendirme, Özet Tablo ve veri doğrulama araçlarını uygulamak.

Görselleştirme

Verinin amacına uygun grafik türünü seçmek ve grafiği yanıltmayacak biçimde düzenlemek; çalışma kitabını yazdırmaya, paylaşmaya ve korumaya hazırlamak.

3.2.1 Excel ve çalışma sayfası

Excel, satır ve sütunlardan oluşan bir ızgara üzerinde çalışır. Izgaranın her gözüne bir sayı, bir metin ya da bir formül yazılabilir. Bu basit yapı, küçük bir alışveriş listesinden binlerce satırlık ölçüm verisine kadar çok farklı işe uyar. Bu kesim, ızgaranın parçalarını ve bu parçaların nasıl adlandırıldığını anlatır.

3.2.1.1 Elektronik tablo ne işe yarar?

Kâğıt üzerinde tutulan bir not çizelgesini düşünelim. Her öğrencinin vize ve final notu yazılır, sonra ortalamalar hesap makinesiyle tek tek bulunur. Bir not yanlış girilmişse o öğrencinin ortalaması yeniden hesaplanır. Sınıf ortalaması da baştan bulunur. Elektronik tablo bu işi tersine çevirir. Ortalama bir kez formül olarak yazılır; notlardan biri değişince sonuç kendiliğinden yenilenir.

Tanım 3.2.1

Elektronik tablo (*spreadsheet*), veriyi satır ve sütunlardan oluşan hücrelerde tutan ve hücrelere yazılan formüllerle bu veriden hesap yapan programdır. Microsoft Excel, LibreOffice Calc ve Google E-Tablolar bu türün yaygın örnekleridir.

Excel'in başlıca kullanım alanları altı başlıkta toplanabilir:

- **Hesaplama.** Formül ve işlevlerle yapılan hesaplar tekrarlanabilir. Aynı hesap yeni veriyle bir kez daha yapılacaksa yalnızca veri değiştirilir.
- **Liste yönetimi.** Öğrenci, ürün, müşteri ya da ölçüm kayıtları satır ve sütunlarda düzenli tutulur.
- **Analiz.** Kayıtlar sıralanır, belirli bir koşula uyanlar süzülür, gruplara göre toplam ve ortalama çıkarılır.
- **Görselleştirme.** Sayılar grafiğe dönüştürülür; eğilim ve karşılaştırma bir bakışta görülür.
- **Planlama.** Bütçe, takvim, stok ve görev takibi gibi işler tablo biçiminde yürütülür.
- **Paylaşım.** Bulutta saklanan bir çalışma kitabını birkaç kişi aynı anda düzenleyebilir, hücrelere açıklama ekleyebilir.

Bir mühendislik öğrencisi için bu alanların en önemlisi çoğu zaman hesaplamadır. Bir devrenin farklı direnç değerlerindeki akımı, bir deneyin ölçüm ortalaması ya da bir malzeme listesinin toplam maliyeti Excel'de birkaç formülle bulunur. Formül bir kez doğru yazıldığında yüzlerce satıra kopyalanabilir.

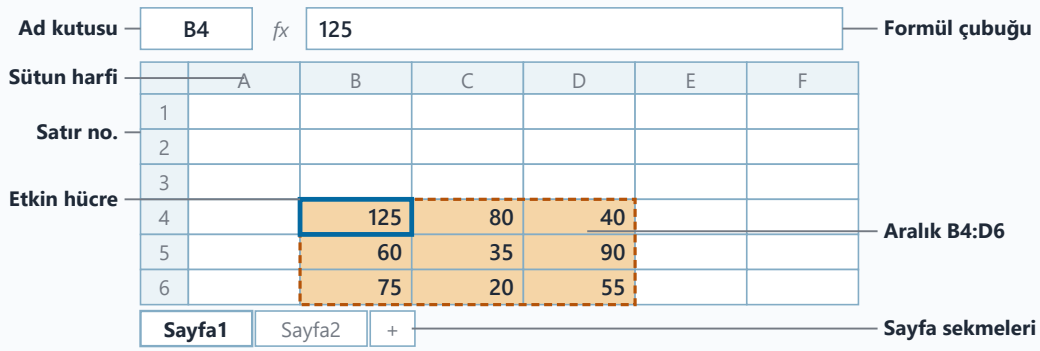
3.2.1.2 Çalışma kitabı, sayfa ve hücre

Excel'de açılan her dosya bir **çalışma kitabıdır**. Kitabın içinde bir ya da daha çok **çalışma sayfası** bulunur. Her sayfa, aynı ızgara yapısında ayrı bir tablodur. Sayfaların adları pencerenin altındaki sekmelerde görünür; yeni bir kitap açıldığında tek sayfa vardır ve adı “Sayfa1”dir.

Tanım 3.2.2

Çalışma kitabı (*workbook*), Excel dosyasının tamamıdır; günümüzde varsayılan uzantısı `.xlsx`'tir. **Çalışma sayfası** (*worksheet*), kitabın içindeki tek bir hücre ızgarasıdır. **Hücre** (*cell*), bir satırla bir sütunun kesiştiği tek gözdür ve tek bir değer ya da formül tutar.

Excel penceresinde ızgaranın çevresinde birkaç önemli öğe vardır. Şekil 3.2.1 bunları bir arada gösterir.



Şekil 3.2.1: Excel penceresinin temel öğeleri. Sol üstteki ad kutusu etkin hücrenin adresini (B4), yanındaki formül çubuğu o hücrenin içeriğini (125) gösterir. Sütunlar harflerle, satırlar sayılarla adlandırılır. B4:D6 aralığı dokuz hücreyi kapsar; sayfa sekmeleri alttadır.

- **Ad kutusu** (*name box*): seçili hücrenin adresini gösterir. Buraya bir adres yazılıp Enter'a basılınca o hücreye gidilir. Örneğin `Z500` yazmak, fareyle uzun süre kaydırmaktan hızlıdır.
- **Formül çubuğu** (*formula bar*): etkin hücrenin gerçek içeriğini gösterir. Hücrede bir formül varsa hücre sonucu, formül çubuğu ise formülün kendisini gösterir.
- **Sütun harfleri ve satır numaraları**: ızgaranın kenarlarındaki başlıklardır. Bir başlığa tıklamak bütün sütunu ya da satırı seçer.
- **Etkin hücre** (*active cell*): o anda seçili olan, yazılan şeyin gireceği hücredir. Kalın bir çerçeveyle gösterilir.
- **Sayfa sekmeleri** (*sheet tab*): kitaptaki sayfaları listeler. Sekmenin yanındaki **+** düğmesi yeni sayfa ekler.

Pencerenin üst kısmında ise **şerit** bulunur. Şerit, komutları sekmelere ayırır: **Dosya**, **Giriş**, **Ekle**, **Sayfa Düzeni**, **Formüller**, **Veri**, **Gözden Geçir** ve **Görünüm**. Notta bir komutun yeri "Veri → Sırala" biçiminde yazılır. Bu yazım, önce Veri sekmesinin, sonra o sekmedeki Sırala düğmesinin seçildiğini anlatır.

3.2.1.3 Hücre adresi ve aralık

Her hücrenin bir adresi vardır. Adres önce sütun harfi, sonra satır numarasıyla yazılır: C3, C sütunuyla 3. satırın kesiştiği hücredir. Sütun harfleri A'dan Z'ye gider; Z'den sonra iki harfli adlar başlar: AA, AB, …, AZ, BA. Microsoft 365 Excel'de bir sayfada 1.048.576 satır ve 16.384 sütun bulunur. Son sütunun adı XFD'dir.

Örnek 3.2.1

AB ve XFD sütunları baştan kaçıncı sütundur? Bir sayfada toplam kaç hücre vardır?

1. **Tek harf.** A = 1, B = 2, …, Z = 26'dır. Sütun harfleri, 26 harfli bir sayı sistemine benzer.
2. **AB.** İlk harf 26'lık bir "tur" sayar, ikinci harf turun içindeki sırayı verir: A × 26 + B = 1 × 26 + 2 = **28**. Demek ki AA 27., AB 28. sütundur.
3. **XFD.** X = 24, F = 6, D = 4. Üç harfte ilk harf 26 × 26 = 676'lık turları sayar: 24 × 676 + 6 × 26 + 4 = 16.224 + 156 + 4 = **16.384**. Bu sayı 2^{14}'tür.
4. **Hücre sayısı.** 1.048.576 satır (2^{20}) × 16.384 sütun = **17.179.869.184** hücre, yani yaklaşık 17 milyar hücre.

Bu sınırlar pratikte nadiren zorlanır. Yine de milyonlarca satırlık bir veri bir sayfaya sığmaz. O boyuttaki veriler için veritabanı ya da programlama dili gibi başka araçlar kullanılır.

Çoğu hesap tek bir hücre yerine bir hücre grubuyla yapılır. Bitişik hücrelerden oluşan dikdörtgen bir bölgeye **aralık** denir.

Tanım 3.2.3

Aralık (*range*), sol üst ve sağ alt köşe hücreleriyle belirlenen dikdörtgen hücre bloğudur ve iki adres arasına iki nokta konarak yazılır: `B4:D6`, B4'ten D6'ya kadar olan dokuz hücreyi kapsar.

Örnek 3.2.2

`B4:D6`, `A1:A10` ve `C3:F10` aralıkları kaçar hücre içerir?

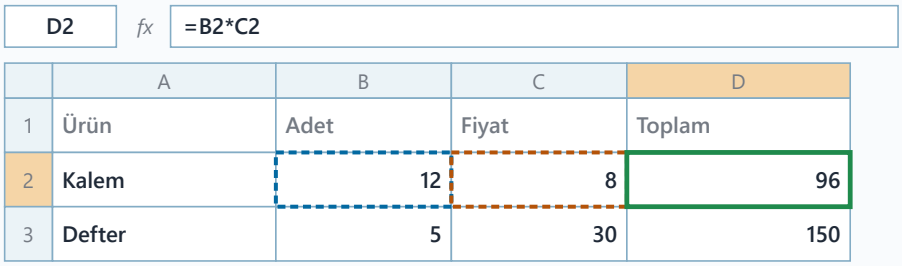
1. **Formül.** Sütun sayısı = son sütun − ilk sütun + 1; satır sayısı = son satır − ilk satır + 1. Hücre sayısı bu ikisinin çarpımıdır. "+ 1" iki uç hücrenin de aralığa dahil olmasından gelir.
2. **B4:D6.** Sütunlar B, C, D: 4 − 2 + 1 = 3. Satırlar 4, 5, 6: 6 − 4 + 1 = 3. Hücre sayısı 3 × 3 = **9**.

3. **A1:A10.** Tek sütun, 10 − 1 + 1 = 10 satır: **10** hücre.
4. **C3:F10.** C, D, E, F: 4 sütun. 10 − 3 + 1 = 8 satır. 4 × 8 = **32** hücre.

Bir aralığı seçmenin birkaç yolu vardır. Fare sol üst hücreden sağ alt hücreye sürüklenebilir. Ya da ilk hücre seçilir, Shift basılı tutularak son hücreye tıklanır. Ad kutusuna `B4:D6` yazmak da aynı sonucu verir. Bitişik olmayan hücreler Ctrl tuşu basılıyken tek tek seçilir.

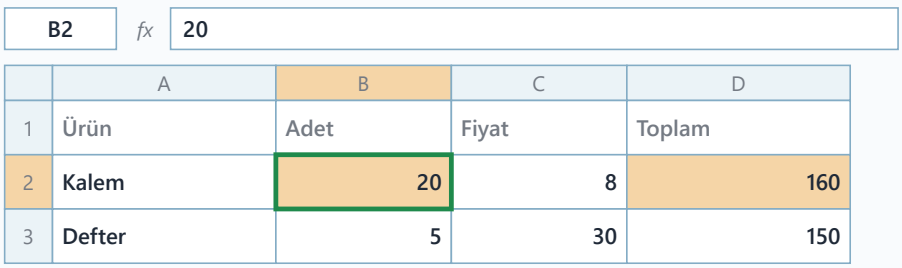
3.2.1.4 Hücrede değer, hücrede formül

Bir hücrede iki tür içerik bulunabilir. Birincisi doğrudan yazılmış bir değerdir: 12 sayısı ya da "Kalem" sözcüğü gibi. İkincisi ise eşittir işaretiyle başlayan bir formüldür. Formül başka hücrelerin adreslerini kullanır ve sonucu o hücrelerdeki değerlerden hesaplar. Şekil 3.2.2 küçük bir alışveriş tablosunu gösterir.



Şekil 3.2.2: Ürün, adet ve fiyat bilgisini tutan bir tablo. D2 seçilidir: hücrede sonuç (96), formül çubuğunda formülün kendisi (`=B2*C2`) görünür.

D2 hücresine `=B2*C2` yazılmıştır. Excel bu formülü şöyle okur: "B2'deki değeri C2'deki değerle çarp." B2'de 12, C2'de 8 olduğu için D2'de 12 × 8 = 96 görünür. Aynı mantıkla D3'teki `=B3*C3` formülü 5 × 30 = 150 verir. Formül sayıların kendisini değil, sayıların bulunduğu hücrelerin adreslerini kullanır. Formülün asıl gücü bu ayrımdan gelir.



Şekil 3.2.3: B2'deki adet 12'den 20'ye çıkarıldı. D2'deki formüle dokunulmadığı hâlde sonuç kendiliğinden 160 oldu.

Şekil 3.2.3'te kalem sayısı 12'den 20'ye çıkarılmıştır. D2'deki formül hâlâ `=B2*C2`'dir, ama B2 değiştiği için sonuç 20 × 8 = 160 olur. Formüle 12 × 8 yerine `=12*8` yazılsaydı, adet değişince formülün de elle düzeltilmesi gerekirdi. Bu yüzden formüllerde sabit sayılar yerine mümkün olduğunca hücre adresleri kullanılır.

Sık yapılan hata

Hesaplanan sonucu formül yerine elle yazmak. Örneğin D2'ye 96 sayısı doğrudan yazılırsa, B2 değiştiğinde D2 eski değerde kalır. Hücre aynı görünür, ama tablo artık yanlıştır. Bir hücrenin değer mi formül mü tuttuğu formül çubuğuna bakılarak anlaşılır.

3.2.2 Veri girişi

Excel bir hücreye yazılanı yorumlar: “12” sayı, “Kalem” metin, “01.03.2024” tarih olarak algılanır. Bu yorum, hesapların doğru yapılıp yapılmayacağını belirler. Bu kesim, veri tiplerini, hücreye veri girmenin pratik yollarını ve sayfanın satır ve sütunlarıyla yapılan işlemleri anlatır.

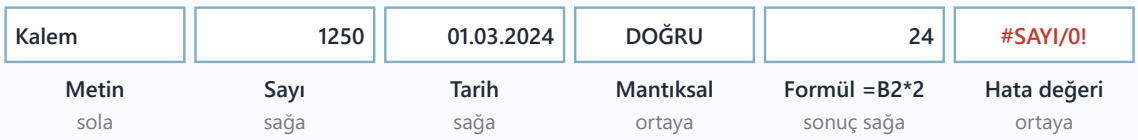
3.2.2.1 Veri tipleri

Tanım 3.2.4

Veri tipi (*data type*), bir hücredeki değerin türüdür ve o değerle hangi işlemlerin yapılabileceğini belirler. Excel'deki başlıca türler metin, sayı, tarih ve saat, mantıksal değer, formül ve hata değeridir.

- **Metin**: ad, açıklama, kategori gibi hesaba katılmayan içerik. “Kalem”, “Elektrik”, “A-101” birer metindir.
- **Sayı**: hesaba katılabilen nicelik. Türkçe ayarlarda ondalık ayırıcı virgüldür: 35,5 bir sayıdır. 35.5 yazılırsa Türkçe Excel bunu sayı olarak tanımaz ve metin olarak saklar; 3.5 gibi bir giriş ise 3 Mayıs tarihine bile dönüşebilir.
- **Tarih ve saat**: Excel tarihleri arka planda sayı olarak saklar. Bu sayede iki tarih arasındaki gün farkı çıkarma işlemiyle bulunur.
- **Mantıksal değer**: yalnızca `DOĞRU` ya da `YANLIŞ` olabilir. Karşılaştırma formüllerinin sonucu bu türdendir: `=5>3` formülü DOĞRU verir.
- **Formül**: eşittir işaretiyle başlayan hesaplama. Hücre formülün kendisini değil, sonucunu gösterir.
- **Hata değeri**: formül hesaplanamayınca çıkan özel sonuç. `#SAYI/0!`, `#AD?`, `#DEĞER!` bu türün örnekleridir.

Excel, bir hücreye girilenin türünü hizalamayla belli eder. Hücreye özel bir hizalama verilmemişse metin sola, sayı ve tarih sağa, mantıksal değer ve hata değeri ortaya yaslanır. Şekil 3.2.4 bu varsayılan hizalamaları gösterir.



Şekil 3.2.4: Veri tiplerinin hücredeki varsayılan hizalaması. Metin sola, sayı ve tarih sağa yaslanır; formülün sonucu da sayıysa sağa yaslanır. Mantıksal değer ve hata değeri ortalanır.

Bu kural küçük ama değerli bir denetim aracıdır. Bir sütundaki sayılardan biri sola yaslı duruyorsa, o hücre büyük olasılıkla metin olarak saklanmıştır. Öyle bir hücre TOPLA gibi işlevlerin hesabına girmez. Yanlış bir toplamın nedeni çoğu zaman böyle bir hücredir.

Örnek 3.2.3

Türkçe ayarlı bir Excel'de A1:A5 hücrelerine sırasıyla `250`, `12,5`, `01.03.2024`, `12 TL` ve `DOĞRU` yazılıyor. Her hücrenin türü ve hizası nedir? `=A1+A2` ne verir?

1. **A1 = 250.** Yalnızca rakamlardan oluşur: sayı, sağa yaslı.
2. **A2 = 12,5.** Ondalık ayırıcı virgül olduğu için sayı olarak tanınır, sağa yaslanır.
3. **A3 = 01.03.2024.** Gün.ay.yıl kalıbı tarih olarak tanınır; tarih de bir sayı olduğu için sağa yaslanır.
4. **A4 = 12 TL.** Sonda harfler vardır; Excel bunu sayıya çeviremez ve metin olarak saklar. Sola yaslanır.
5. **A5 = DOĞRU.** Mantıksal değer olarak tanınır ve ortalanır.
6. **Formül.** `=A1+A2` = 250 + 12,5 = **262,5**.

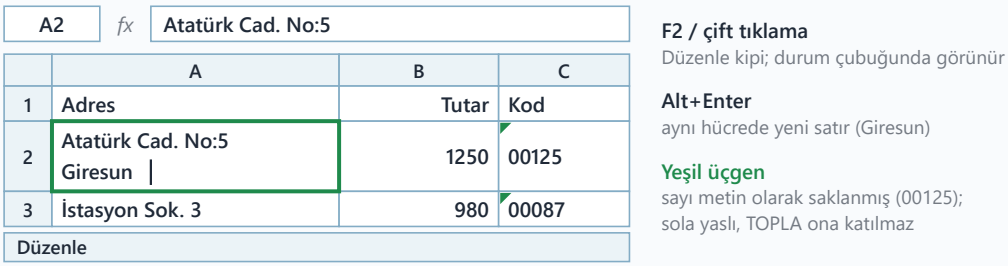
A4'teki "12 TL" ile hesap yapılmak istenirse `#DEĞER!` hatası çıkar. Para birimini göstermenin doğru yolu, hücreye yalnız 12 yazıp birimi sayı biçimiyle eklemektir.

3.2.2.2 Veri girme ve düzenleme

Bir hücreye veri girmek için hücre seçilip yazılır ve Enter'a basılır. Enter girişi onaylar ve bir alttaki hücreye geçer; Tab ise sağdaki hücreye geçer. Hücrenin içeriğini silmeden düzeltmek için **Düzenle** kipine girilir. Bu kipe F2 tuşuyla ya da hücreye çift tıklanarak girilir. Düzenle kipinde imleç hücrenin içindedir ve ok tuşları hücre değiştirmek yerine imleci metin içinde gezdirir. Tablo 3.2.1 veri girişinde en sık kullanılan tuşları özetler.

İşlem	Tuş	Açıklama
Hücreyi düzenle	`F2`	Düzenle kipine geçer; içerik silinmeden düzeltilir.
Girişi onayla	`Enter`	Değeri yazar, alttaki hücreye geçer.
Hücre içinde yeni satır	`Alt+Enter`	Aynı hücrede ikinci bir satır başlatır.
Girişten vazgeç	`Esc`	Yazılanı iptal eder, eski içerik kalır.
İçeriği temizle	`Delete`	Değeri siler; hücrenin biçimi kalır.
Satır seç	`Shift+Space`	Etkin hücrenin bütün satırını seçer.
Sütun seç	`Ctrl+Space`	Etkin hücrenin bütün sütununu seçer.

Tablo 3.2.1: Veri girişinde sık kullanılan tuşlar.



Şekil 3.2.5: A2 hücresi Düzenle kipindedir; Alt+Enter ile adres iki satıra bölünmüştür. C sütunundaki 00125 ve 00087 kodları metin olarak saklanmıştır. Excel bu hücreleri sol üst köşedeki yeşil üçgenle işaretler.

Şekil 3.2.5'teki C sütunu önemli bir durumu gösterir. Öğrenci numarası, posta kodu ya da ürün kodu gibi değerler rakamlardan oluşur, ama başlarındaki sıfırlar anlamlıdır. Excel 00125'i sayı olarak alırsa baştaki sıfırları atar ve 125 yazar. Sıfırları korumak için değer metin olarak saklanır: ya hücreye önce kesme işareti yazılır (`'00125`) ya da hücre önceden **Metin** biçimine getirilir. Excel bu durumda hücrenin sol üst köşesine yeşil bir üçgen koyar ve "sayı metin olarak saklanmış" uyarısı verir.

Kod gibi hesaba girmeyecek değerler için metin olarak saklamak doğrudur. Hesaba girecek sayılar ise asla metin olarak saklanmamalıdır. Bu tür hücreler sıralamada da yanlış yere düşer, çünkü metinler sayılardan ayrı sıralanır.

Sık yapılan hata

Başka bir programdan kopyalanan sayıları kontrol etmeden kullanmak. Web sayfalarından ya da metin dosyalarından gelen sayılar çoğu zaman metin olarak yapışır; bazen sonlarında görünmeyen bir boşluk da bulunur. Böyle bir sütunun toplamı eksik çıkar ve hata iletisi de görünmez. Sola yaslı sayılar ve yeşil üçgenler bu sorunun işaretidir. Yeşil üçgenin yanındaki uyarı düğmesinden **Sayıya Dönüştür** seçilerek düzeltilir.

Veri girerken iki ilkeye uyulmaya değer. Birincisi, bir hücreye tek bir gözlem yazılır: “12 kg” yerine yalnızca 12 yazılır, birim sütun başlığına taşınır. İkincisi, bir sütun tek bir özelliği tutar: aynı sütuna hem tarih hem açıklama yazılmaz. Bu iki ilke, sıralama, filtreleme ve formüllerin sorunsuz çalışmasının ön koşuludur.

3.2.2.3 Satır, sütun ve sayfalar

Tablo büyüdükçe satır ve sütunlar üzerinde işlem yapmak gerekir. Bu işlemler hem sağ tık menüsünden hem de Giriş sekmesindeki **Hücreler** grubundan (Ekle, Sil, Biçim) yapılır.

- **Ekleme ve silme.** Bir satır eklenince altındaki satırlar aşağı kayar; bir sütun silinince sağındaki sütunlar sola kayar. Formüllerdeki adresler bu kaymaya göre kendiliğinden güncellenir.
- **Boyut.** Satır yüksekliği ve sütun genişliği başlıkların arasındaki çizgi sürüklenerek ayarlanır. Çizgiye çift tıklamak sütunu en uzun içeriğe göre otomatik genişletir.
- **Gizleme.** Gizlenen satır ya da sütun silinmez, yalnızca görünümden kalkar. Formüller gizli hücreleri de kullanmaya devam eder.
- **Dondurma.** Uzun bir listede aşağı kaydırılınca başlık satırı ekrandan çıkar ve hangi sütunun ne olduğu unutulur. **Görünüm → Bölmeleri Dondur → Üst Satırı Dondur** komutu 1. satırı ekranda sabit tutar.
- **Sayfa adı ve düzeni.** “Sayfa1” gibi adlar yerine içeriği anlatan kısa adlar kullanılır: “Veri”, “Hesap”, “Rapor”. Ham veri, hesaplama ve rapor ayrı sayfalarda tutulabilir.

Kaydırmadan önce

	A	B	C
1	Ad	Vize	Final
2	Ali	70	80
3	Ece	90	85
4	Can	60	72
5	Deniz	55	64

Aşağı kaydırınca: 1. satır sabit

	A	B	C
1	Ad	Vize	Final
41	Selin	88	91
42	Burak	76	70
43	Mert	65	81
44	Zehra	93	90

Görünüm → Bölmeleri Dondur → Üst Satırı Dondur

Şekil 3.2.6: Bölmeleri dondurma. Solda listenin başı görünür. Sağda 41.–44. satırlara kaydırılmıştır, ama donmuş 1. satır (Ad, Vize, Final) ekranda kalır.

Dondurma yalnızca ilk satırla sınırlı değildir. **İlk Sütunu Dondur** komutu A sütununu sabitler; bu, satır başlıkları sütunda duran geniş tablolarda işe yarar. Hem satır hem sütun dondurmak için önce sabit kalacak bölümün sağ altındaki ilk hücre seçilir, sonra **Bölmeleri Dondur** komutu verilir. Örneğin B2 seçilirse 1. satır ve A sütunu birlikte donar.

3.2.3 Biçimlendirme

Aynı veri, biçimlendirilmeden önce ve sonra çok farklı okunur. Başlığı belirgin, sayıları hizalı ve birimleri açık bir tablo hem daha kolay okunur hem de daha az yanlış anlaşılır. Excel'de biçimlendirmenin iki ayrı katmanı vardır: hücrenin görünüşü (yazı tipi, renk, kenarlık, hizalama) ve sayının gösterilme biçimi. İkinci katman, hesaplar açısından ayrıca önemlidir.

3.2.3.1 Hücre biçimlendirme

Hücre biçimlendirme komutları Giriş sekmesindeki **Yazı Tipi**, **Hizalama**, **Sayı** ve **Stiller** gruplarında toplanır. Bütün seçenekler **Hücreleri Biçimlendir** iletişim kutusunda da bulunur; bu kutu `Ctrl+1` ile açılır. Kutunun sekmeleri Sayı, Hizalama, Yazı Tipi, Kenarlık, Dolgu ve Koruma'dır.

Önce

	A	B	C
1	Ürün	Fiyat	Adet
2	Kalem	1250	12
3	Defter	35,5	40
4	Silgi	8	100

Sonra

	A	B	C
1	**Ürün**	**Fiyat**	**Adet**
2	Kalem	₺1.250,00	12
3	Defter	₺35,50	40
4	Silgi	₺8,00	100

Şekil 3.2.7: Biçimlendirmeden önce ve sonra aynı tablo. Sağda başlık satırı dolguyla ayrılmış, fiyatlar para birimi biçiminde ve iki ondalıkla gösterilmiş, tabloya kenarlık eklenmiştir.

- **Yazı ve dolgu.** Kalın yazı ve açık bir dolgu (*fill*) rengi başlık satırını verinin geri kalanından ayırır.
- **Hizalama.** Metnin hücre içindeki yatay ve dikey konumunu belirler. Sayıların sağa yaslı kalması, basamakların alt alta gelmesini sağlar.
- **Kenarlık** (*border*). Tablonun sınırlarını gösterir. Her hücreye kalın kenarlık eklemek ise tabloyu kalabalıklaştırır ve okumayı zorlaştırır.
- **Metni kaydır** (*wrap text*). Uzun bir metni sütunu genişletmeden hücre içinde birkaç satıra böler.
- **Birleştirme. Birleştir ve Ortala** komutu birkaç hücreyi tek hücre gibi gösterir. Rapor başlığında kullanılabilir, ama veri aralığında birleştirilmiş hücreler sıralamayı ve filtrelemeyi bozar.
- **Hücre stili.** Önceden tanımlı biçim takımlarıdır. Aynı biçim bir kez tanımlanır ve bütün kitapta tutarlı uygulanır.

Sık yapılan hata

Veri tablosunun ortasında hücreleri birleştirmek. Örneğin aynı bölümdeki üç öğrencinin bölüm adı tek bir birleştirilmiş hücreye yazılırsa, Excel ikinci ve üçüncü satırın bölümünü boş görür. Sıralama “aynı boyutta birleştirilmiş hücreler gerekir” uyarısıyla durur; filtre de yalnız ilk satırı bulur. Her satıra bölüm adı ayrı ayrı yazılmalıdır.

3.2.3.2 Sayı biçimleri

Bir hücredeki sayı farklı biçimlerde gösterilebilir. 0,25 değeri yüzde biçiminde %25, 45292 değeri tarih biçiminde 01.01.2024 olarak görünür. Biçim değiştirmek hücrede saklanan sayıyı değiştirmez; yalnızca o sayının ekrana nasıl yazılacağını belirler.

Tanım 3.2.5

Sayı biçimi (*number format*), bir hücrede saklanan değerin ekranda nasıl gösterileceğini belirleyen kuraldır. Biçim değeri değiştirmez: hesaplarda her zaman formül çubuğunda görünen temel değer kullanılır.

Saklanan değer	Biçim	Hücrede görünen
0,25	Yüzde	%25
1250	Para Birimi	₺1.250,00
45292	Tarih	01.01.2024
0,5	Saat	12:00
123456789	Bilimsel	1,23E+08

Tablo 3.2.2: Aynı türden değerlerin farklı sayı biçimlerinde görünümü.

Tablo 3.2.2'deki satırların her biri bir kuralı gösterir. Yüzde biçimi değeri 100 ile çarpıp sonuna yüzde işareti ekleyerek gösterir; hücrede yine 0,25 durur. Para birimi biçimi binlik ayırıcı ve iki ondalık basamak ekler. Tarih ve saat değerlerinin neden sayı olduğu 3.2.8.1'de açıklanır. Kısaca, 1 gün 1 sayısına karşılık gelir; 0,5 de yarım gün, yani 12 saattir. Bilimsel biçimde 1,23E+08, $1{,}23 \times 10^{8}$ anlamına gelir.



Şekil 3.2.8: Formül çubuğu saklanan değeri, hücre ise biçimlenmiş görünümü gösterir. B2'de 0,25 saklanır ve %25 görünür; B3'te 45292 saklanır ve 01.01.2024 görünür.

Görünen değerle saklanan değer arasındaki fark en çok ondalık basamaklarda kafa karıştırır. **Ondalık Azalt** düğmesi bir sayıyı daha az basamakla gösterir, ama yuvarlamaz. Hesap, her zaman formül çubuğundaki tam değerle yapılır.

Örnek 3.2.4

A1, A2 ve A3 hücrelerine sırasıyla 1,4, 1,4 ve 1,4 yazılmış ve üçüne de 0 ondalık basamaklı sayı biçimi verilmiştir. Hücrelerde 1 görünür. A4'teki `=TOPLA(A1:A3)` formülü ne gösterir?

1. **Saklanan değerler.** Biçim yalnızca görünümü etkilediği için hücrelerde 1,4 durur.
2. **Toplam.** 1,4 + 1,4 + 1,4 = 4,2.
3. **Gösterim.** A4 de 0 ondalık biçimdeyse 4,2, 4 olarak görünür.

Tabloyu okuyan kişi 1 + 1 + 1 = 3 bekler, ama 4 görür. Görünen değerlerle hesap yapılması isteniyorsa değer gerçekten yuvarlanmalıdır: `=YUVARLA(A1;0)` gibi bir formül 1,4'ü 1'e çevirir ve sonraki hesaplar bu yuvarlanmış değerle yapılır.

Sayı biçimlerinin hazır seçenekleri Giriş → Sayı grubundaki açılır listede bulunur: Genel, Sayı, Para Birimi, Finansal, Kısa Tarih, Uzun Tarih, Saat, Yüzde, Kesir, Bilimsel ve Metin. Hücreleri Biçimlendir kutusundaki **İsteğe Uyarlanmış** kategorisi ise kendi biçim kodunu yazmaya izin verir. Örneğin `0,0 "kg"` kodu 12 değerini “12,0 kg” olarak gösterir. Hücrede yine 12 sayısı saklandığı için bu değer hesaplara katılır.

Sık yapılan hata

Yüzde değerini yanlış ölçekte girmek. Yüzde biçimli bir hücreye 20 yazıldığında Excel bunu %20 olarak alır ve 0,2 saklar. Biçimsiz bir hücreye 20 yazılıp sonra yüzde biçimi verilirse hücre %2000 gösterir, çünkü saklanan değer 20'dir. Bir oran hücresinin değeri her zaman formül çubuğundan denetlenmelidir: %20 için orada 0,2 ya da %20 görünmelidir.

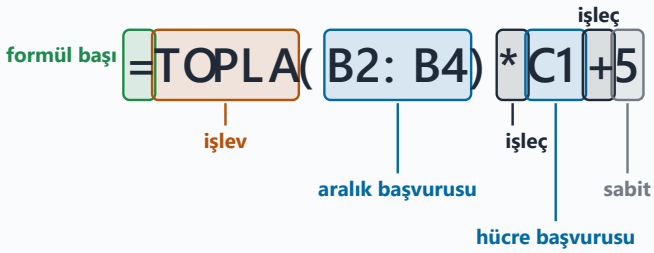
3.2.4 Formüller ve işleç önceliği

Excel'in hesap gücü formüllerden gelir. Bir formül, bir hesap makinesine yazılan işlemin hücre adresleriyle zenginleştirilmiş biçimidir. Bu kesim formülün parçalarını, Excel'in işlemleri hangi sırayla yaptığını ve bu sıradan doğan şaşırtıcı bir durumu anlatır.

3.2.4.1 Formülün parçaları

Tanım 3.2.6

Formül (*formula*), bir hücreye yazılan ve eşittir işaretiyle (=) başlayan hesaplama ifadesidir. Formülde dört tür öğe bulunabilir: **sabit** (5 gibi doğrudan yazılmış değer), **hücre başvurusu** (C1 gibi başka bir hücrenin adresi), **işleç** (+, −, *, / gibi işlem işareti) ve **işlev** (TOPLA gibi hazır hesap).



Şekil 3.2.9: `=TOPLA(B2:B4)*C1+5` formülünün parçaları: formül başındaki eşittir işareti, TOPLA işlevi, B2:B4 aralık başvurusu, çarpma ve toplama işleçleri, C1 hücre başvurusu ve 5 sabiti.

Şekil 3.2.9'daki formül şöyle okunur: "B2, B3 ve B4'ün toplamını C1 ile çarp, sonuca 5 ekle." B2:B4'te 10, 20 ve 30, C1'de 2 bulunuyorsa sonuç (10 + 20 + 30) × 2 + 5 = 60 × 2 + 5 = 125 olur.

Eşittir işareti unutulursa Excel yazılanı formül olarak değil, metin olarak saklar. `B2*C2` yazılan bir hücrede hesap yapılmaz; hücrede bu yazı aynen görünür. Excel'deki işleçler şunlardır:

- **Aritmetik:** `+` toplama, `-` çıkarma ve olumsuzlama, `*` çarpma, `/` bölme, `^` üs alma, `%` yüzde.
- **Karşılaştırma:** `=`, `>`, `<`, `>=`, `<=` ve eşit değil anlamında `<>`. Sonuç DOĞRU ya da YANLIŞ'tır.
- **Birleştirme:** `&` iki metni yan yana ekler. `=A1&" TL"`, A1'de 1250 varsa "1250 TL" metnini üretir. Birleştirmede sayının biçimi kullanılmaz; A1 para birimi biçiminde görünse bile sonuçta binlik ayırıcı yer almaz.
- **Başvuru:** iki nokta (`:`) bir aralık kurar: `B2:B4`.

İyi bir formül yazarken üç ilke yol gösterir. Birincisi, işlem sırası şüpheliyse parantez kullanılır: `=(B2+C2)/2` iki notun ortalamasıdır. İkincisi, sonuç başvurulan hücreler değiştiğinde yeniden hesaplanır; bu yüzden formüller değerler yerine adreslerle yazılır. Üçüncüsü, KDV oranı ya da indirim oranı gibi sabitler formülün içine gömülmez, ayrı bir hücreye yazılır. Oran değişince yalnızca o hücre güncellenir.

3.2.4.2 İşleç önceliği

Bir formülde birden çok işleç varsa Excel bunları belirli bir sırayla uygular. Bu sıra, matematikteki işlem önceliğine büyük ölçüde benzer: önce parantez, sonra üs, sonra çarpma ve bölme, en son toplama ve çıkarma. Excel'in sırasında iki ek basamak daha vardır: olumsuzlama (eksi işareti) ve yüzde.

Tanım 3.2.7

İşleç önceliği (*operator precedence*), bir formüldeki işleçlerin hangi sırayla uygulanacağını belirleyen kurallar dizisidir. Aynı öncelikteki işleçler soldan sağa uygulanır.

Öncelik	İşlem	Örnek
1	Parantez	`=(A1+B1)*2`
2	Olumsuzlama (işaret)	`=-A1`
3	Yüzde	`=A1*10%`
4	Üs	`=A1^2`
5	Çarpma ve bölme	`=A1*B1/C1`
6	Toplama ve çıkarma	`=A1+B1-C1`
7	Birleştirme	`=A1&" TL"`
8	Karşılaştırma	`=A1>B1`

Tablo 3.2.3: Excel'de işleç önceliği (1 en yüksek).

"Soldan sağa" kuralı, aynı basamaktaki işlemler için geçerlidir. `=12/3*2` formülünde bölme ve çarpma aynı önceliktedir. Excel önce soldaki işlemi yapar: 12 / 3 = 4, sonra 4 × 2 = **8**. Önce çarpma yapılsaydı 12 / 6 = 2 bulunurdu; bu yanlıştır.

Karşılaştırmanın en sonda olması da işe yarar. `=A1+B1>100` formülünde önce toplama yapılır, sonra toplam 100 ile karşılaştırılır. Böylece "iki notun toplamı 100'ü geçiyor mu?" sorusu parantezsiz yazılabilir.

Örnek 3.2.5

`=5+3*2^2-8/4` formülünün sonucu nedir? Başa parantez eklenip `=(5+3)*2^2-8/4` yazılırsa sonuç ne olur?

1. **Üs.** Formülde parantez ve olumsuzlama yok; en yüksek öncelik üstedir: 2^2 = 4. Formül `5+3*4-8/4` olur.
2. **Çarpma ve bölme, soldan sağa.** 3 × 4 = 12, sonra 8 / 4 = 2. Formül `5+12-2` olur.
3. **Toplama ve çıkarma, soldan sağa.** 5 + 12 = 17, 17 − 2 = **15**.
4. **Parantezli biçim.** Önce parantez: 5 + 3 = 8. Sonra üs: 2^2 = 4. Sonra çarpma ve bölme: 8 × 4 = 32 ve 8 / 4 = 2. En son 32 − 2 = **30**.

Bir çift parantez sonucu iki katına çıkardı. Parantez, işlem sırasını değiştirmenin tek yoludur.

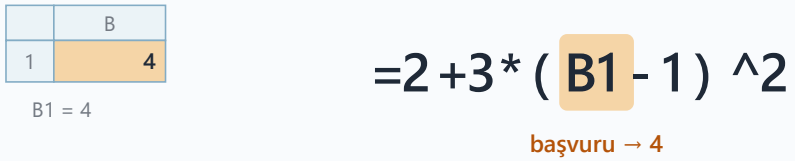
3.2.4.3 Bir formülün adım adım hesaplanması

Excel bir formülü hesaplarken önce hücre başvurularını o hücrelerdeki değerlerle değiştirir. Sonra işlemleri öncelik sırasına göre birer birer yapar. Her adımda formül biraz kısalır, sonunda tek bir değer kalır. Bu süreç **Formüller → Formül Denetleme → Formülü Değerlendir** komutuyla ekranda adım adım izlenebilir. Tablo 3.2.4, B1'de 4 varken `=2+3*(B1-1)^2` formülünün hesabını gösterir.

Adım	Yapılan işlem	Formülün yeni hâli
0	Başlangıç	`=2+3*(B1-1)^2`
1	Başvuru değerle değişir: B1 → 4	`=2+3*(4-1)^2`
2	Parantez içi: 4 − 1 = 3	`=2+3*3^2`
3	Üs: 3^2 = 9	`=2+3*9`
4	Çarpma: 3 × 9 = 27	`=2+27`
5	Toplama: 2 + 27 = 29	`=29`

Tablo 3.2.4: `=2+3*(B1-1)^2` formülünün hesaplanması (B1 = 4).

Sonuç **29**'dur. Şekil 3.2.10 bu sürecin ilk adımını gösterir. Parantezler kaldırılıp `=2+3*B1-1^2` yazılsaydı hesap bambaşka giderdi: üs 1^2 = 1, çarpma 3 × 4 = 12, toplama ve çıkarma 2 + 12 − 1 = 13. Parantezler, "önce farkı al, sonra karesini al" isteğini Excel'e iletmenin yoludur.



Şekil 3.2.10: Hesabın ilk adımı. Excel formüldeki B1 başvurusunu B1 hücresindeki değerle (4) değiştirir; işlemler bundan sonra öncelik sırasıyla yapılır.

3.2.4.4 Olumsuzlama ve üs

Tablo 3.2.3'te olumsuzlama ikinci, üs dördüncü sıradadır. Yani Excel'de eksi işareti üsten önce uygulanır. Matematikte ise -2^2 ifadesi $-(2^2) = -4$ olarak okunur; üs önce alınır, eksi sonra uygulanır. Bu fark Excel'de şaşırtıcı bir sonuç doğurur.

Örnek 3.2.6

`=-2^2` , `=0-2^2` ve `=-(2^2)` formülleri Excel'de ne verir?

1. `=-2^2` **.** Baştaki eksi olumsuzlamadır ve üsten önce gelir. Önce −2 oluşur, sonra (−2)^2 = **4** hesaplanır.
2. `=0-2^2` **.** Buradaki eksi olumsuzlama değil, çıkarmadır; çıkarma üsten sonra gelir. Önce 2^2 = 4, sonra 0 − 4 = **−4**.
3. `=-(2^2)` **.** Parantez en önce: 2^2 = 4. Sonra olumsuzlama: **−4**.

Matematikteki $-2^2 = -4$ sonucunu almak için ikinci ya da üçüncü yazım kullanılır. Aynı durum hücre başvurularında da geçerlidir: A1'de 3 varsa `=-A1^2` formülü 9 verir, −9 değil.

Üs işlecinin bir başka özelliği de soldan sağa uygulanmasıdır. `=2^3^2` formülünü Excel (2^3)^2 = 8^2 = 64 olarak hesaplar. Matematikte ise üs sağdan sola okunur: 2^(3^2) = 2^9 = 512. Bu tür ifadelerde parantez yazmak, okuyanın ve Excel'in aynı şeyi anlamasını sağlar.

Sık yapılan hata

Bir mühendislik formülünü Excel'e parantezsiz aktarmak. Örneğin çan eğrisinde geçen e^{-x^2} terimi `=ÜS(-A1^2)` biçiminde yazılırsa Excel $e^{(-x)^2} = e^{x^2}$ hesaplar. A1 = 2 için beklenen değer yaklaşık 0,018 iken sonuç yaklaşık 54,6 çıkar. Doğru yazım `=ÜS(-(A1^2))` ya da `=ÜS(0-A1^2)` 'dir. Eksi işaretli her üs ifadesinde parantezin yeri düşünülmeli, sonuç elle hesaplanan bir değerle karşılaştırılmalıdır.

3.2.5 Hücre başvuruları

Bir formül çoğu zaman tek bir hücrede kalmaz. Aynı hesap bir sütunun bütün satırları için yapılacaksa formül bir kez yazılır ve aşağı kopyalanır. Kopyalanan formüldeki adresler yeni konuma göre kendiliğinden değişir. Bu değişimin kuralını bilmek, Excel'i doğru kullanmanın belki de en önemli adımıdır. Bu kesim üç başvuru türünü, kopyalandıklarında nasıl davrandıklarını ve en sık yapılan başvuru hatasını anlatır.

3.2.5.1 Başvuru türleri

Tanım 3.2.8

Hücre başvurusu (*cell reference*), bir formülün başka bir hücrenin değerini kullanmak için yazdığı adrestir. Üç türü vardır:

- **Göreli başvuru** (*relative reference*), `A1`: formül kopyalanınca hem sütun harfi hem satır numarası formülün kaydığı kadar kayar.
- **Mutlak başvuru** (*absolute reference*), `$A$1`: formül nereye kopyalanırsa kopyalansın aynı hücreyi gösterir.
- **Karma başvuru** (*mixed reference*), `$A1` ya da `A$1`: yalnızca sütun ya da yalnızca satır sabittir; öteki parça kayar.

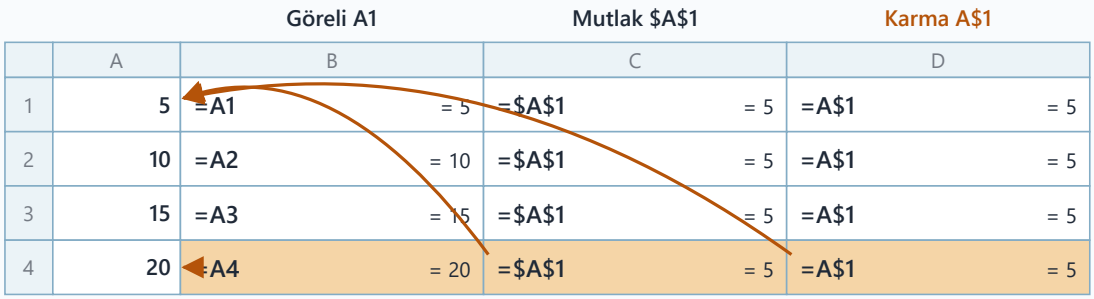
Dolar işaretinin anlamı basittir: `$`, hemen ardından gelen parçayı kilitler. `$A$1`'de iki işaret vardır; sütun da satır da kilitlidir. `$A1`'de yalnızca A harfinin önünde işaret vardır; sütun kilitli, satır serbesttir. `A$1`'de ise satır kilitli, sütun serbesttir. Dolar işaretinin para birimiyle bir ilgisi yoktur; yalnızca bir kilit simgesidir.

Başvurular yalnızca aynı sayfadaki tek bir hücreyle sınırlı değildir. Bir aralık iki adresle (`A1:A10`), başka bir sayfadaki hücre sayfa adıyla (`Satışlar!B2`), anlamlı bir ad verilmiş hücre de o adla (`KDV_Orani`) yazılabilir. Bu iki biçim 3.2.5.8'de ele alınır.

3.2.5.2 Aşağı doldururken başvurular

Bir formülü kopyalamanın en hızlı yolu **doldurma tutamacıdır** (*fill handle*). Tutamaç, seçili hücrenin sağ alt köşesindeki küçük karedir. Fare bu kareye getirilince imleç ince bir artıya dönüşür. Tutamaç aşağı sürüklenince formül alttaki hücrelere kopyalanır; sağa sürüklenince sağdaki hücrelere kopyalanır.

Başvuru türlerinin farkını görmek için küçük bir deney yapalım. A1:A4 hücrelerinde 5, 10, 15 ve 20 olsun. B1'e `=A1`, C1'e `=$A$1`, D1'e `=A$1` yazılır. Üç formül de A1'i gösterir ve üçü de 5 verir. Sonra B1:D1 seçilip doldurma tutamacıyla 4. satıra kadar aşağı sürüklenir. Şekil 3.2.11 sonucu gösterir.



Şekil 3.2.11: Üç başvuru türü aşağı doldurulduktan sonra. Göreli `A1` her satırda bir aşağı kayar (A2, A3, A4) ve 5, 10, 15, 20 verir. `$A$1` ve `A$1` satırı kilitli tuttuğu için her satırda A1'i gösterir ve hep 5 verir.

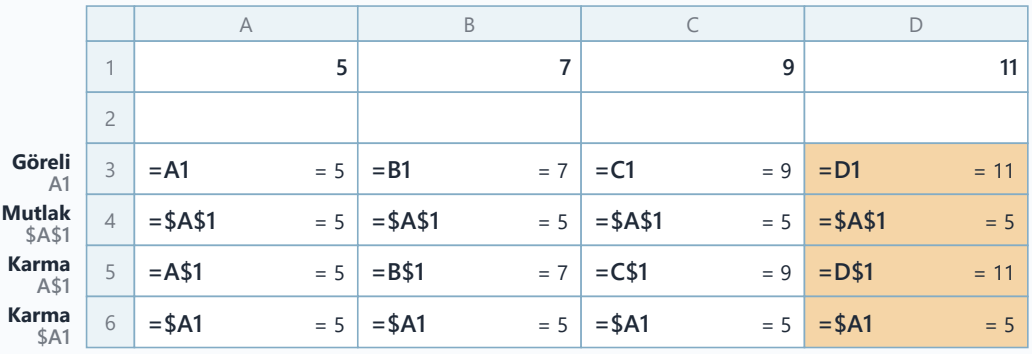
Satır	B (göreli)	C (mutlak)	D (karma, satır kilitli)
1	`=A1` → 5	`=$A$1` → 5	`=A$1` → 5
2	`=A2` → 10	`=$A$1` → 5	`=A$1` → 5
3	`=A3` → 15	`=$A$1` → 5	`=A$1` → 5
4	`=A4` → 20	`=$A$1` → 5	`=A$1` → 5

Tablo 3.2.5: Aşağı doldurmada formüllerin satır satır hâli (A1:A4 = 5, 10, 15, 20).

Tablo ilginç bir sonucu gösterir: aşağı doldurulurken `A$1`, `$A$1` ile aynı davranır. Bunun nedeni, aşağı doldurmada yalnızca satır numarasının kaymak istemesidir. Satır kilitliyse adres değişmez. Sütun kilidinin bir etkisi yoktur, çünkü sütun zaten kaymaz. İki başvurunun farkı ancak formül sağa doldurulunca ortaya çıkar.

3.2.5.3 Sağa doldururken başvurular

Bu kez veriler 1. satırda olsun: A1:D1'de 5, 7, 9 ve 11. A3:A6 hücrelerine dört başvuru türü yazılır: `=A1`, `=$A$1`, `=A$1` ve `=$A1`. Dördü de 5 verir. Sonra A3:A6 seçilip D sütununa kadar sağa doldurulur.



Şekil 3.2.12: Dört başvuru türü sağa doldurulduktan sonra. Göreli başvuru ve `A$1` sütun boyunca kayar (B1, C1, D1) ve 7, 9, 11 verir. `$A$1` ve `$A1` sütunu kilitli tuttuğu için hep A1'i gösterir.

Sağa doldurmada yalnızca sütun harfi kaymak ister. `A1` ve `A$1` 'de sütun serbesttir: B1, C1, D1'e kayarlar. `$A$1` ve `$A1` 'de sütun kilitlidir: hep A1'de kalırlar. Bu kez `$A1` , mutlak başvuru gibi davranmıştır. İki deneyden çıkan kural şudur:

- `$A$1` hiçbir yönde kaymaz.
- `A$1` aşağı doldurulunca kaymaz, sağa doldurulunca kayar.
- `$A1` sağa doldurulunca kaymaz, aşağı doldurulunca kayar.
- `A1` her iki yönde de kayar.

3.2.5.4 Kopyalama kuralı

Doldurma tutamacı formülü bitişik hücrelere taşır. Formül `Ctrl+C` ve `Ctrl+V` ile uzak bir hücreye de kopyalanabilir. Kural aynıdır: Excel, kaynak hücreyle hedef hücre arasındaki sütun ve satır farkını bulur ve bu farkı formüldeki kilitsiz parçalara ekler.

Tanım 3.2.9

Kopyalama kuralı. Bir formül, sütun farkı Δs ve satır farkı Δr olan bir hücreye kopyalanıyor olsun. Formüldeki her başvuruda önünde `$` olmayan sütun harfi Δs kadar kayar. Önünde `$` olmayan satır numarası da Δr kadar kayar. `$` işaretli parçalar değişmez.

Örnek 3.2.7

C3 hücresinde `=A1+$B$2+B$1+$A2` formülü var. Formül E7'ye kopyalanıyor. E7'deki formül nedir?

1. **Fark.** C'den E'ye 2 sütun sağa ($\Delta s = 2$), 3'ten 7'ye 4 satır aşağı ($\Delta r = 4$).
2. `A1` **.** İki parça da serbest: A + 2 = C, 1 + 4 = 5 → `C5` .
3. `$B$2` **.** İki parça da kilitli → `$B$2` olduğu gibi kalır.
4. `B$1` **.** Sütun serbest, satır kilitli: B + 2 = D, satır 1 → `D$1` .
5. `$A2` **.** Sütun kilitli, satır serbest: A kalır, 2 + 4 = 6 → `$A6` .

Sonuç: `=C5+$B$2+D$1+$A6` . Her başvuru ayrı ayrı ele alınır; formülün bütünü için tek bir kural uygulanmaz.

Bu kural ters yönde de işler. Bir formül yukarı ya da sola kopyalanınca farklar negatiftir; adresler de geriye kayar. Kayma sayfanın dışına taşıyorsa, örneğin `=A1` bir satır yukarı kopyalanırsa, Excel geçerli bir adres bulamaz ve `#BAŞV!` hatası verir.

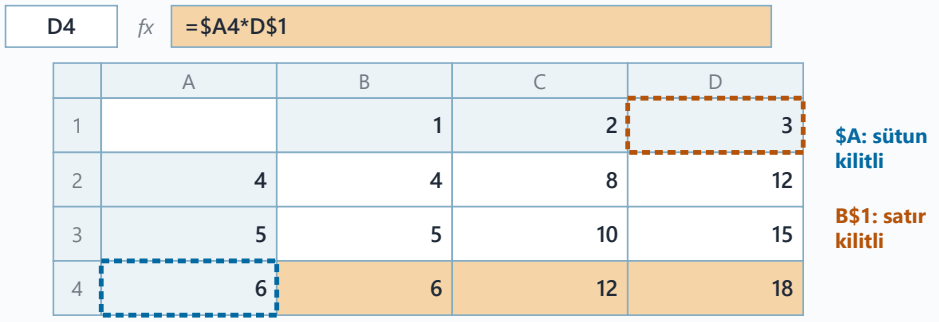
Kesip yapıştırmak (`Ctrl+X`) ise kopyalamadan farklıdır. Taşınan formüldeki başvurular değişmez; formül aynı hücreleri göstermeye devam eder. Kural, yalnızca kopyalanan formüller içindir.

Başvuru türlerinin davranışını denemek için ders sayfasındaki Excel hücre başvuruları simülatörü kullanılabilir. Simülatörde bir formül seçilip doldurma tutamacıyla sürüklenir ve her hücrede oluşan formül ile sonuç aynı anda görülür.

3.2.5.5 Karma başvuru: çarpım tablosu

Karma başvurunun en açık kullanım alanı, iki yönde birden doldurulan tablolardır. Bir çarpım tablosunda her hücre, kendi satırının başındaki sayıyla kendi sütununun başındaki sayının çarpımıdır. A2:A4'te 4, 5, 6; B1:D1'de 1, 2, 3 olsun. B2'ye yazılacak tek bir formül bütün tabloyu doldurabilir mi?

Formülün iki başvurusu olacaktır. Birincisi satır başındaki sayıyı gösterir. Bu sayı hep A sütunundadır, ama satırı hücreyle birlikte değişmelidir: `$A2`. İkincisi sütun başındaki sayıyı gösterir. Bu sayı hep 1. satırdadır, ama sütunu hücreyle birlikte değişmelidir: `B$1`. Formül `=$A2*B$1` 'dir.



Şekil 3.2.13: B2'ye yazılan `=$A2*B$1` formülü B2:D4'e doldurulmuştur. Son doldurulan D4 hücresinde formül `=$A4*D$1`, sonuç 6 × 3 = 18'dir. `$A` sütunu, `$1` satırı kilitler.

Satır	B	C	D
2	`=$A2*B$1` → 4	`=$A2*C$1` → 8	`=$A2*D$1` → 12
3	`=$A3*B$1` → 5	`=$A3*C$1` → 10	`=$A3*D$1` → 15
4	`=$A4*B$1` → 6	`=$A4*C$1` → 12	`=$A4*D$1` → 18

Tablo 3.2.6: Çarpım tablosunda her hücrenin formülü ve sonucu.

Tablo 3.2.6'ya satır satır bakınca kural açıkça görülür. Sağa gidildikçe yalnızca B$1'in sütun harfi değişir (B → C → D). Aşağı gidildikçe yalnızca $A2'nin satır numarası değişir (2 → 3 → 4). Tek bir formül, dokuz hücrenin hepsinde doğru sonucu verir.

Sık yapılan hata

Çarpım tablosunda `=$A$2*$B$1` gibi tam mutlak başvuru kullanmak. Bu formül bütün tabloya kopyalanınca her hücre 4 × 1 = 4 verir. Tersine, `=A2*B1` gibi tam göreli bir formül de yanlıştır: C3'e kopyalanınca `=B3*C2` olur. B3 ve C2'de de birer formül durduğu için hata katlanarak

büyür: B3 = 5 × 4 = 20, C2 = 4 × 2 = 8 ve C3 = 20 × 8 = 160 çıkar. Doğru değer 10'dur. Doğru başvuru türü şu soruyla bulunur: "Bu değer hangi yönde sabit kalmalı?"

3.2.5.6 F4 tuşu

Dolar işaretlerini elle yazmak yerine F4 tuşu kullanılabilir. Formül düzenlenirken imleç bir başvurunun üzerindeyken F4'e her basış, başvuru türünü sıradaki türe çevirir. Dört basışta döngü tamamlanır ve başvuru ilk hâline döner. Tablo 3.2.7 `=B2*E1` formülündeki E1 başvurusu için döngüyü gösterir.

F4 basışı	Formül	Sütun (E)	Satır (1)
Basılmadı	`=B2*E1`	kayar	kayar
1.	`=B2*$E$1`	sabit	sabit
2.	`=B2*E$1`	kayar	sabit
3.	`=B2*$E1`	sabit	kayar
4.	`=B2*E1`	kayar	kayar

Tablo 3.2.7: F4 tuşuyla başvuru türünün değişmesi.

F4 yalnızca formül düzenlenirken ve imleç bir başvurunun içinde ya da hemen yanındayken çalışır. Pek çok dizüstü bilgisayarda işlev tuşları ses ya da parlaklık gibi işlere ayrıldığı için `Fn+F4` gerekir. Mac'te kısayol farklıdır. `$` işareti her zaman elle de yazılabilir.

3.2.5.7 Kopyalanan formülde göreli başvuru hatası

Göreli başvurunun yanlış yerde kullanılması, Excel'deki en sinsi hatalardan birine yol açar. Sinsidir, çünkü Excel hiçbir hata iletisi vermez; sonuçlar sayı olduğu için ilk bakışta doğru görünür. Bir örnekle izleyelim.

Üç ürünün fiyatı B2:B4'te: defter 200, kalem 150, silgi 80. İndirim oranı tek bir hücrede tutuluyor: E1 = %20. C2'ye indirimli fiyat formülü yazılacak ve aşağı kopyalanacak. İlk deneme `=B2*(1-E1)`'dir. C2'de sonuç 200 × (1 − 0,2) = 200 × 0,8 = 160 olur; bu doğrudur. Formül aşağı kopyalanınca ne olur?

	A	B	C	D	E
1	Ürün	Fiyat	İndirimli	İndirim	%20
2	Defter	200	160		
3	Kalem	150	150		
4	Silgi	80			

=B3*(1-E2)

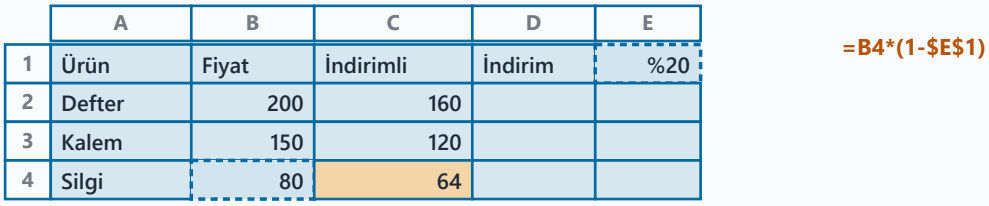
Şekil 3.2.14: C2'deki `=B2*(1-E1)` aşağı kopyalanınca C3'te `=B3*(1-E2)` olur. E2 boş olduğu için indirim uygulanmaz ve C3'te fiyatın kendisi (150) görünür.

Hücre	Göreli: formül	Sonuç	Mutlak: formül	Sonuç
C2	`=B2*(1-E1)`	200 × 0,8 = 160	`=B2*(1-$E$1)`	200 × 0,8 = 160
C3	`=B3*(1-E2)`	150 × 1 = 150	`=B3*(1-$E$1)`	150 × 0,8 = 120
C4	`=B4*(1-E3)`	80 × 1 = 80	`=B4*(1-$E$1)`	80 × 0,8 = 64

Tablo 3.2.8: İndirim formülünün iki biçimi aşağı kopyalanınca (E1 = %20, E2 ve E3 boş).

Göreli biçimde iki başvuru da bir satır kaymıştır. B2'nin B3 olması istenen şeydir, çünkü her satır kendi fiyatını almalıdır. E1'in E2 olması ise hatadır. E2 boştur ve Excel boş hücreyi 0 sayar. Formül 150 × (1 − 0) = 150 hesaplar; indirim hiç uygulanmaz. C4'te de aynı şey olur.

Düzeltme, oran hücresini kilitlemektir. C2'deki formülde imleç E1'in üzerine getirilir ve F4'e bir kez basılır: `=B2*(1-$E$1)`. Fiyat başvurusu göreli kalır, oran başvurusu mutlak olur. Formül aşağı kopyalanınca B3, B4'e kayar ama $E$1 yerinde kalır. Sonuçlar 160, 120 ve 64 olur.



Şekil 3.2.15: Düzeltilmiş formül. C4'te `=B4*(1-$E$1)` → 80 × 0,8 = 64. E1'deki oran %30 yapılırsa üç sonuç birden güncellenir.

Mutlak başvurunun bir yararı daha vardır. Oran %30'a çıkarılırsa yalnızca E1 değiştirilir; üç sonuç birden 140, 105 ve 56 olur. Oran her formüle ayrı ayrı yazılsaydı, üç formülün de tek tek düzeltilmesi gerekirdi.

Örnek 3.2.8

Bir kırtasiye siparişinde B2:B6'da adet, C2:C6'da birim fiyat (₺) var: kalem 12 adet × 8; defter 5 × 30; silgi 20 × 4,5; cetvel 6 × 12,75; dosya 10 × 22. KDV oranı F1'de %20. D sütununa tutar, E sütununa KDV'li tutar yazılacak ve formüller 6. satıra kadar kopyalanacak.

1. **Tutar.** D2'ye `=B2*C2`. İki başvuru da her satırda kendi satırına kaymalıdır; ikisi de göreli kalır. Sonuçlar: 12 × 8 = 96; 5 × 30 = 150; 20 × 4,5 = 90; 6 × 12,75 = 76,5; 10 × 22 = 220.
2. **KDV'li tutar.** E2'ye `=D2*(1+$F$1)`. Tutar satırla birlikte kayar, oran sabittir. Sonuçlar: 96 × 1,2 = 115,2; 150 × 1,2 = 180; 90 × 1,2 = 108; 76,5 × 1,2 = 91,8; 220 × 1,2 = 264.
3. **Denetim.** E6'daki formül `=D6*(1+$F$1)` olmalıdır. Formül çubuğunda bu görülüyorsa kopyalama doğrudur.
4. **Toplamlar.** `=TOPLA(D2:D6)` = 96 + 150 + 90 + 76,5 + 220 = 632,5 ₺. `=TOPLA(E2:E6)` = 759 ₺. Aradaki fark 759 − 632,5 = 126,5 ₺ KDV'dir; bu da 632,5 × 0,2'ye eşittir.

Bu tür hataları yakalamanın kolay bir yolu, formülleri sonuç yerine görünür kılmaktır. **Formüller → Formül Denetleme → Formülleri Göster** komutu ya da `Ctrl+`` kısayolu, her hücrede sonucun yerine formülün kendisini gösterir. Bir sütundaki formüllerin hepsi tek bir oran hücresini göstermeliyse, o başvurunun her satırda aynı kalıp kalmadığına bakılır. Aynı komut bir kez daha verilince görünüm eski hâline döner.

Sık yapılan hata

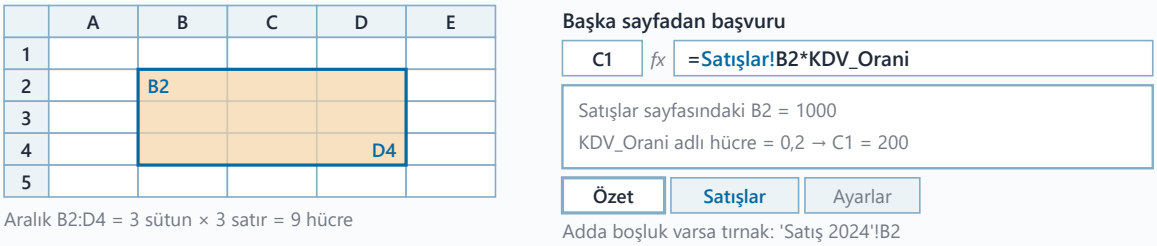
Kopyalanan formülün yalnız ilk satırını denetlemek. Göreli başvuru hatası ilk satırda hiç görünmez; C2'deki sonuç her iki biçimde de 160'tır. Hata ancak ikinci ve sonraki satırlarda ortaya çıkar. Bir formül kopyalandıktan sonra en az bir ara satırın ve son satırın formülü de açılıp bakılmalıdır.

3.2.5.8 Başka sayfaya başvuru ve adlandırılmış aralık

Büyük çalışma kitaplarında veri, hesap ve rapor çoğu zaman ayrı sayfalarda durur. Bir formül başka bir sayfadaki hücreyi kullanacaksa sayfa adı adresin önüne yazılır ve arasına ünlem işareti konur: `Satışlar!B2`, Satışlar sayfasının B2 hücresidir. Sayfa adında boşluk varsa ad tek tırnak içine alınır: `'Satış 2024'!B2`. Formül yazılırken diğer sayfanın sekmesine tıklayıp hücreyi seçmek, bu adresi Excel'in kendisinin yazmasını sağlar.

Tanım 3.2.10

Adlandırılmış aralık (*named range*), bir hücreye ya da aralığa verilen anlamlı addır. Ad, formülde adresin yerine kullanılır ve her zaman aynı hücreyi gösterir; yani mutlak başvuru gibi davranır.



Şekil 3.2.16: Solda B2:D4 aralığı: 3 sütun × 3 satır = 9 hücre. Sağda Özet sayfasındaki C1 hücresi, Satışlar sayfasının B2 hücresini (1000) KDV_Orani adlı hücreyle (0,2) çarpar ve 200 verir.

Bir hücreye ad vermenin en kısa yolu, hücreyi seçip ad kutusuna adı yazmak ve Enter'a basmaktır. **Formüller → Tanımlı Adlar → Ad Tanımla** komutu da aynı işi yapar; **Ad Yöneticisi** ise kitaptaki bütün adları listeler. Adlarda boşluk kullanılamaz, bu yüzden sözcükler alt çizgiyle bağlanır:

`KDV_Orani`. Ad bir harf ya da alt çizgiyle başlamalı ve bir hücre adresine benzememelidir; örneğin `KDV1` bir sütun adı ve satır numarasına benzediği için kabul edilmez.

Şekil 3.2.16'daki formül `=Satışlar!B2*KDV_Orani`'dır. Satışlar sayfasının B2 hücresinde 1000, KDV_Orani adlı hücrede 0,2 olduğu için sonuç 1000 × 0,2 = 200'dür. `=Satışlar!B2*Ayarlar!$B$1` ile aynı sonucu verir, ama okuyan kişi neyin neyle çarpıldığını hemen anlar. Ad mutlak başvuru gibi davrandığı için formül kopyalanınca da KDV_Orani yerinde kalır.

3.2.6 Temel ve mantıksal işlevler

Bir sütundaki yüz sayıyı toplamak için `=B2+B3+…+B101` yazmak hem yorucu hem de hataya açıktır. Excel'de bu gibi sık yapılan hesaplar için hazır işlevler vardır. Bu kesim işlevin yapısını, en sık kullanılan beş işlevi ve karar vermeye yarayan mantıksal işlevleri anlatır.

3.2.6.1 İşlevin yapısı ve temel işlevler

Tanım 3.2.11

İşlev (*function*), belirli bir hesabı yapan ve adıyla çağrılan hazır formüldür. İşlev adından sonra parantez açılır ve hesaba girecek değerler, yani **bağımsız değişkenler** (*arguments*), parantezin içine yazılır. Türkçe Excel'de işlev adları Türkçedir ve bağımsız değişkenler noktalı virgülle ayrılır: `=EĞER(B2>=60;"Geçti";"Kaldı")`.

İngilizce Excel'de aynı formül `=IF(B2>=60,"Geçti","Kaldı")` biçiminde yazılır. Ayırıcının virgül değil noktalı virgül olmasının nedeni, Türkçe ayarlarda virgülün ondalık ayırıcı olmasıdır. `=TOPLA(1,5;2)` formülü 1,5 ile 2'yi toplar ve 3,5 verir. Bir dosya İngilizce Excel'de açılınca işlev adları ve ayırıcılar kendiliğinden çevrilir; dosyanın içinde formül dilden bağımsız saklanır. İnternetteki bir İngilizce formül örneği Türkçe Excel'e ise elle uyarlanmalıdır.

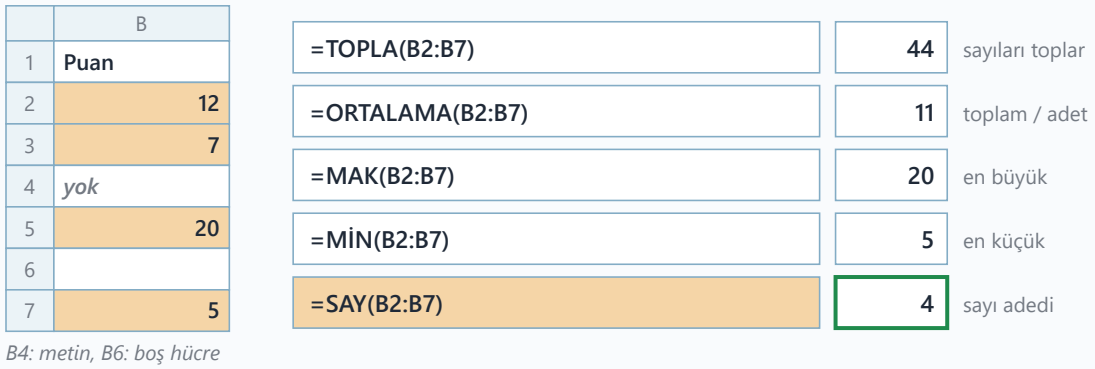
İşlev	Örnek	Ne yapar?
TOPLA	`=TOPLA(B2:B10)`	Aralıktaki sayıları toplar.
ORTALAMA	`=ORTALAMA(B2:B10)`	Sayıların aritmetik ortalamasını verir.
MAK	`=MAK(B2:B10)`	En büyük sayıyı verir.
MİN	`=MİN(B2:B10)`	En küçük sayıyı verir.
SAY	`=SAY(B2:B10)`	Sayı içeren hücreleri sayar.
BAĞ_DEĞ_DOLU_SAY	`=BAĞ_DEĞ_DOLU_SAY(B2:B10)`	Boş olmayan bütün hücreleri sayar.
EĞERSAY / ETOPLA	`=EĞERSAY(B2:B10;">=50")`	Koşula uyan hücreleri sayar ya da toplar.
BUGÜN	`=BUGÜN()`	Bugünün tarihini verir.

Tablo 3.2.9: Sık kullanılan temel işlevler.

Bir işlev üç yoldan yazılabilir. Birincisi, adını doğrudan yazmaktır: `=TO` yazılınca Excel TO ile başlayan işlevleri listeler ve Tab tuşu seçileni tamamlar. İkincisi, **Formüller → İşlev Ekle** komutuyla işlevi listeden seçmektir; açılan kutu her bağımsız değişkenin ne olduğunu açıklar. Üçüncüsü, **Giriş → Otomatik Toplam** düğmesidir. Bu düğme seçili hücrenin üstündeki sayıları bulup `=TOPLA(…)` formülünü kendisi yazar. Kısayolu `Alt+=`'dir.

3.2.6.2 İşlevler aynı veride

Temel işlevlerin metin ve boş hücrelere karşı nasıl davrandığı, sonuçları doğru yorumlamak için önemlidir. B2:B7 aralığında 12, 7, "yok" metni, 20, boş bir hücre ve 5 bulunsun. Beş işlev aynı aralığa uygulanınca Şekil 3.2.17 ortaya çıkar.



Şekil 3.2.17: Beş temel işlev aynı aralıkta. B4'teki "yok" metni ve B6'daki boş hücre bütün işlevlerce yok sayılır: TOPLA 44, ORTALAMA 11, MAK 20, MİN 5, SAY 4.

Örnek 3.2.9

B2:B7 = 12, 7, "yok", 20, boş, 5 iken beş işlevin sonucunu adım adım bulalım.

1. **Hangi hücreler sayı?** 12, 7, 20 ve 5. "yok" metindir, B6 boştur; ikisi de hesaba girmez.
2. **TOPLA.** 12 + 7 + 20 + 5 = **44**.
3. **ORTALAMA.** Toplam, sayı adedine bölünür: 44 / 4 = **11**. Bölen 6 değil 4'tür; boş hücre 0 kabul edilseydi bölen 5 olur ve sonuç 8,8 çıkardı.
4. **MAK ve MİN.** En büyük sayı **20**, en küçük sayı **5**.
5. **SAY.** Sayı içeren hücre adedi: **4**. Boş olmayan bütün hücreleri sayan BAĞ_DEĞ_DOLU_SAY ise metni de sayar ve **5** verir.

Boş hücre ile 0 içeren hücre arasındaki fark ortalamada belirleyicidir. Bir öğrenci sınava girmemişse hücre boş bırakılır ve ortalamaya katılmaz. Öğrenci sınava girip 0 almışsa hücreye 0 yazılır ve ortalamayı düşürür. İki durumda tablo aynı görünebilir, ama sınıf ortalaması farklı çıkar.

Sık yapılan hata

SAY işlevini “dolu hücre sayısı” sanmak. SAY yalnızca sayıları sayar; ad listesindeki öğrenci sayısını bulmak için SAY kullanılırsa sonuç 0 çıkar. Metinli bir sütundaki kayıt sayısı için BAĞ_DEĞ_DOLU_SAY kullanılır.

3.2.6.3 EĞER işlevi

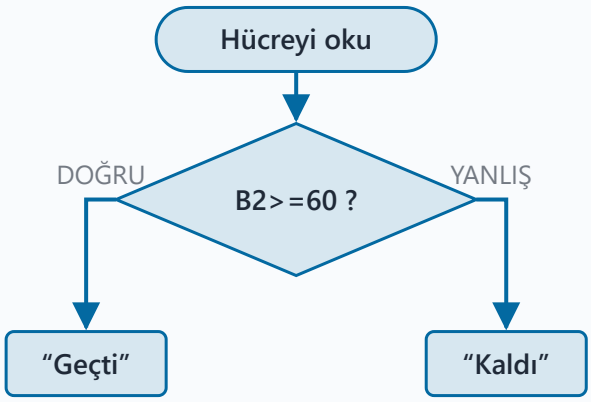
Bazı hesaplarda sonuç bir koşula bağlıdır: not 60 ve üstündeyse “Geçti”, değilse “Kaldı” yazılır. Bu tür kararlar EĞER işleviyle verilir.

Tanım 3.2.12

EĞER işlevi üç bağımsız değişken alır: `=EĞER(mantıksal_sınama; doğruysa_değer; yanlışsa_değer)`. **Mantıksal sınama** (*logical test*), sonucu DOĞRU ya da YANLIŞ olan bir ifadedir. Sınama DOĞRU ise ikinci, YANLIŞ ise üçüncü bağımsız değişken döner.

```
=EĞER(B2>=60; "Geçti"; "Kaldı")
```

Bu formülde sınama `B2>=60`'tır. Doğruysa “Geçti” metni, yanlışsa “Kaldı” metni döner. Metinler formülde çift tırnak içinde yazılır. Tırnak unutulursa Excel `Geçti` sözcüğünü bir ad sanar ve `#AD?` hatası verir. Şekil 3.2.18 işlevin karar yapısını bir akış diyagramıyla gösterir.

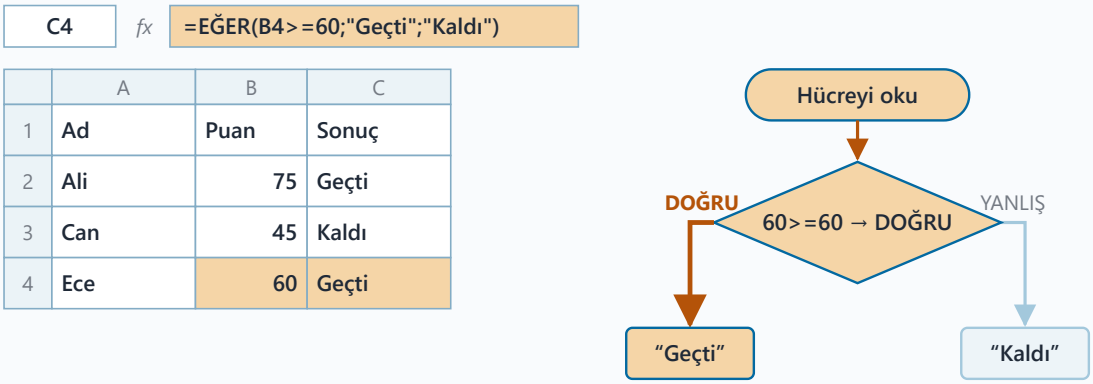


Şekil 3.2.18: EĞER işlevinin akış diyagramı. Hücre okunur, `B2>=60` sınanır; sınama doğruysa “Geçti”, yanlışsa “Kaldı” döner.

Formül C2'ye yazılıp aşağı kopyalandığında her satır kendi puanını sınar. A2:B4'te Ali 75, Can 45 ve Ece 60 puan almış olsun. Tablo 3.2.10 her satırın değerlendirilmesini gösterir.

Hücre	Okunan puan	Sınama	Sonuç
C2	B2 = 75	75 >= 60 → DOĞRU	Geçti
C3	B3 = 45	45 >= 60 → YANLIŞ	Kaldı
C4	B4 = 60	60 >= 60 → DOĞRU	Geçti

Tablo 3.2.10: `=EĞER(B2>=60;"Geçti";"Kaldı")` formülünün satır satır değerlendirilmesi.



Şekil 3.2.19: Son satırın değerlendirilmesi. C4'te B4 = 60 okunur; 60 >= 60 doğru olduğu için ikinci bağımsız değişken ("Geçti") döner.

Üçüncü satır, karşılaştırma işleci seçiminin önemini gösterir. `>=` eşitliği de kapsar; 60 puan alan Ece geçer. Formülde `>` yazılsaydı sınama 60 > 60 → YANLIŞ olur ve Ece "Kaldı" görünürdü. Geçme sınırındaki puanlar, formülün doğru yazılıp yazılmadığını denetlemek için en iyi deneme değerleridir.

EĞER'in döndürdüğü değerler metin olmak zorunda değildir. Sayı ya da başka bir formül de dönebilir. Örneğin `=EĞER(B2>=1000;B2*0,1;0)` formülü 1000 ₺ ve üstü alışverişte %10 indirim tutarını, altında 0'ı verir. B2 = 1250 ise sonuç 125, B2 = 800 ise 0'dır.

3.2.6.4 VE, YADA, DEĞİL ve iç içe EĞER

Bazı kararlar birden çok koşula bağlıdır. "Not en az 60 *ve* devam en az %70 ise geçer" gibi bir kural tek bir karşılaştırmayla yazılamaz. Excel'de koşulları birleştirmek için üç mantıksal işlev vardır:

- **VE**: bütün koşullar DOĞRU ise DOĞRU, en az biri YANLIŞ ise YANLIŞ verir.
- **YADA**: en az bir koşul DOĞRU ise DOĞRU, hepsi YANLIŞ ise YANLIŞ verir.
- **DEĞİL**: tek bir koşulu tersine çevirir; DOĞRU'yu YANLIŞ, YANLIŞ'ı DOĞRU yapar.

Bir öğrencinin notu B2 = 72, devamı C2 = 65 (yüzde olarak), D2 hücresinde de 0 olsun. Tablo 3.2.11 çeşitli formüllerin sonuçlarını gösterir.

Formül	Sonuç	Açıklama
`=VE(B2>=60;C2>=70)`	YANLIŞ	72 >= 60 doğru, 65 >= 70 yanlış; ikisi birden doğru değil.

Formül	Sonuç	Açıklama
`=YADA(B2>=60;C2>=70)`	DOĞRU	En az biri doğru.
`=DEĞİL(B2>=60)`	YANLIŞ	DOĞRU tersine çevrilir.
`=EĞERHATA(B2/D2;"Hesaplanamadı")`	Hesaplanamadı	72 / 0 hata verir; yerine metin gösterilir.
`=EĞER(B2>=85;"AA";EĞER(B2>=70;"BB";"CC"))`	BB	İç içe EĞER: ilk doğru koşulun değeri döner.
`=EĞER(VE(B2>=60;C2>=70);"Geçti";"Kaldı")`	Kaldı	VE, EĞER'in sınaması olarak kullanılır.

Tablo 3.2.11: Mantıksal işlevler (B2 = 72, C2 = 65, D2 = 0).

Son satır en sık kullanılan kalıptır: VE ya da YADA tek başına bir sonuç vermek için değil, EĞER'in sınaması olarak kullanılır. Öğrencinin notu yeterlidir, ama devamı yetersizdir; VE YANLIŞ verdiği için EĞER üçüncü bağımsız değişkeni döndürür.

İkiden çok sonuç gerektiğinde EĞER'ler iç içe yazılır. Bir EĞER'in yanlışsa_değer bölümüne ikinci bir EĞER konur. Excel koşulları soldan sağa sırayla dener ve ilk doğru koşulun değerini döndürür; sonraki koşullara hiç bakmaz.

Örnek 3.2.10

`=EĞER(B2>=85;"AA";EĞER(B2>=70;"BB";"CC"))` formülü B2 = 72, 90 ve 69 için ne döndürür?

1. **B2 = 72.** Dış EĞER: 72 >= 85 → YANLIŞ; üçüncü bağımsız değişkene, yani iç EĞER'e geçilir. İç EĞER: 72 >= 70 → DOĞRU → **BB**.
2. **B2 = 90.** Dış EĞER: 90 >= 85 → DOĞRU → **AA**. İç EĞER hiç hesaplanmaz.
3. **B2 = 69.** 69 >= 85 → YANLIŞ; 69 >= 70 → YANLIŞ → **CC**.

Parantezler sayılarak denetlenir: her EĞER bir parantez açar ve kendi parantezini kapatır. Formülde iki EĞER olduğu için sonda iki kapanış parantezi vardır.

Sık yapılan hata

İç içe EĞER'de koşulları düşük eşikten başlatmak. `=EĞER(B2>=70;"BB";EĞER(B2>=85;"AA";"CC"))` formülünde 90 puan alan öğrenci de ilk sınamayı geçer ve "BB" alır; "AA" koşuluna hiç sıra gelmez. Koşullar en yüksek eşikten başlayarak aşağı doğru yazılmalıdır.

Çok sayıda kademe varsa iç içe EĞER uzar ve okunması zorlaşır. Microsoft 365'te bu iş için `ÇOKEĞER` işlevi de vardır: `=ÇOKEĞER(B2>=85;"AA";B2>=70;"BB";DOĞRU;"CC")`. Bu işlev sınama–değer çiftlerini sırayla dener. Son çiftteki `DOĞRU` sınaması, önceki koşulların hiçbiri tutmazsa kullanılacak değeri belirler. Kademeli eşikler için bir başka yol, 3.2.7.4'te anlatılan yaklaşık eşleşmeli aramadır.

3.2.6.5 EĞERHATA

Bazı formüller belirli girdilerde hata verir. Bir bölme işleminde payda 0 olabilir ya da bir arama, listede olmayan bir değeri arıyor olabilir. EĞERHATA işlevi, böyle durumlarda hata değeri yerine anlamlı bir sonuç göstermeye yarar.

Tanım 3.2.13

EĞERHATA işlevi iki bağımsız değişken alır: `=EĞERHATA(değer; hata_ise_değer)`. Birinci bağımsız değişken hata vermiyorsa onun sonucu, hata veriyorsa ikinci bağımsız değişken döner.

Tablo 3.2.11'deki örnekte `B2/D2` = 72 / 0 hesaplanamaz ve `#SAYI/0!` verir. EĞERHATA bu hatayı yakalar ve "Hesaplanamadı" metnini gösterir. D2'de 4 olsaydı 72 / 4 = 18 hata vermez; EĞERHATA bu durumda 18'i olduğu gibi döndürürdü.

EĞERHATA hatayı gizler, ama nedenini ortadan kaldırmaz. Paydanın neden 0 olduğu yine incelenmelidir. Ayrıca bu işlev bütün hata türlerini yakalar. Yazım yanlışından doğan bir `#AD?` hatası da "Hesaplanamadı" olarak görünür ve gerçek sorun gözden kaçar. Bu yüzden EĞERHATA, hatanın beklendiği ve anlamı bilinen yerlerde kullanılır.

3.2.7 Koşullu sayma, toplama ve arama

Gerçek tablolarda soruların çoğu bir koşul içerir. "Elektrik bölümünden kaç öğrenci var?", "Bu öğrencilerin puanları toplamı nedir?", "Ece'nin puanı kaç?" Bu kesim, bu tür soruları yanıtlayan koşullu sayma ve toplama işlevlerini ve tablodan değer getiren arama işlevlerini anlatır. Kesim boyunca aynı örnek liste kullanılır: Ad, Bölüm ve Puan sütunlu altı öğrenci. Ali (Elektrik, 72), Ece (Bilgisayar, 91), Can (Elektrik, 85), Deniz (Bilgisayar, 64), Burak (Makine, 78) ve Selin (Elektrik, 58). Liste A1:C7'dedir; 1. satır başlıktır.

3.2.7.1 EĞERSAY ve ETOPLA

Tanım 3.2.14

EĞERSAY, bir aralıkta belirli bir **ölçüte** (*criteria*) uyan hücreleri sayar: `=EĞERSAY(aralık; ölçüt)`. **ETOPLA**, ölçüte uyan satırlardaki değerleri toplar: `=ETOPLA(ölçüt_aralığı; ölçüt; toplam_aralığı)`. ETOPLA'da ölçüt bir sütunda sınanır, toplanan değerler aynı satırların başka bir sütunundan alınır.

	A	B	C
1	Ad	Bölüm	Puan
2	Ali	Elektrik	72
3	Ece	Bilgisayar	91
4	Can	Elektrik	85
5	Deniz	Bilgisayar	64
6	Burak	Makine	78
7	Selin	Elektrik	58

=EĞERSAY(B2:B7;"Elektrik")
Bölüm = "Elektrik" olan satırları sayar = 3

=ETOPLA(B2:B7;"Elektrik";C2:C7)
koşulu sağlayan satırların puanlarını toplar = 215

=EĞERSAY(C2:C7;">=80")
80 ve üstü puanları sayar = 2

Şekil 3.2.20: Öğrenci listesinde üç koşullu formül. Bölüm sütununda "Elektrik" 3 kez geçer; Elektrik öğrencilerinin puanları toplamı 215'tir; 80 ve üstü puan alan 2 öğrenci vardır.

Örnek 3.2.11

`=ETOPLA(B2:B7;"Elektrik";C2:C7)` formülü satırları nasıl gezer?

1. **Satır 2.** B2 = Elektrik → ölçüte uyuyor; C2 = 72 toplama eklenir. Ara toplam 72.
2. **Satır 3.** B3 = Bilgisayar → uymuyor; C3 atlanır.
3. **Satır 4.** B4 = Elektrik → uyuyor; 72 + 85 = 157.
4. **Satır 5 ve 6.** Bilgisayar ve Makine → ikisi de atlanır.
5. **Satır 7.** B7 = Elektrik → uyuyor; 157 + 58 = **215**.

Aynı ölçütle `=EĞERSAY(B2:B7;"Elektrik")` 3 verir. İkisinin oranı Elektrik öğrencilerinin ortalamasıdır: 215 / 3 ≈ 71,7. Bu ortalama doğrudan `=EĞERORTALAMA(B2:B7;"Elektrik";C2:C7)` ile de bulunur.

ETOPLA'da iki aralığın aynı boyutta ve aynı satırlarda olması gerekir. Ölçüt B2:B7'de sınanıyorsa toplam aralığı da C2:C7 olmalıdır; C3:C8 yazılırsa her ölçüt bir alt satırdaki puanla eşleşir ve sonuç sessizce yanlış çıkar. Toplam aralığı hiç yazılmazsa ölçüt aralığının kendisi toplanır. `=ETOPLA(C2:C7;">=70")` puanı 70 ve üstü olanları toplar: 72 + 91 + 85 + 78 = 326.

3.2.7.2 Ölçüt yazımı ve çok koşullu işlevler

Ölçüt bir metin, bir sayı ya da bir karşılaştırma olabilir. Karşılaştırma ölçütleri çift tırnak içinde yazılır. Tablo 3.2.12 sık kullanılan biçimleri örnek listeyle birlikte gösterir.

Formül	Anlamı	Sonuç
`=EĞERSAY(B2:B7;"Elektrik")`	Elektrik'e eşit olanlar	3
`=EĞERSAY(B2:B7;"<>Elektrik")`	Elektrik dışındakiler	3
`=EĞERSAY(C2:C7;">=80")`	80 ve üstü puanlar	2
`=EĞERSAY(C2:C7;"<60")`	60'ın altındaki puanlar	1

Formül	Anlamı	Sonuç
`=EĞERSAY(B2:B7;"E*")`	E ile başlayan bölümler (yıldız: herhangi bir dizi)	3
`=EĞERSAY(C2:C7;">="&F1)`	Eşik F1 hücresinden alınır (F1 = 80 ise)	2

Tablo 3.2.12: Ölçüt yazımı ve örnek listede sonuçları.

Son satırdaki yazım önemlidir. Eşik bir hücredeyse karşılaştırma işareti tırnak içinde yazılır ve hücre adresiyle `&` işleci üzerinden birleştirilir. `">=F1"` yazılırsa Excel "F1" sözcüğünü arar ve hiçbir sayıyla eşleşmez. Ölçütlerde büyük–küçük harf ayrımı yapılmaz: "elektrik" ile "Elektrik" aynı sayılır.

Birden çok koşul aynı anda aranıyorsa çok koşullu işlevler kullanılır. `ÇOKEĞERSAY` aralık–ölçüt çiftlerini alır ve hepsine birden uyan satırları sayar. `ÇOKETOPLA`'da ise toplam aralığı en başa yazılır, ardından aralık–ölçüt çiftleri gelir.

Örnek 3.2.12

Elektrik bölümünden puanı 70 ve üstü olan kaç öğrenci vardır? Bu öğrencilerin puanları toplamı nedir?

1. **Formüller.** `=ÇOKEĞERSAY(B2:B7;"Elektrik";C2:C7;">=70")` ve `=ÇOKETOPLA(C2:C7;B2:B7;"Elektrik";C2:C7;">=70")`.
2. **Birinci koşul.** Elektrik satırları: Ali 72, Can 85, Selin 58.
3. **İkinci koşul.** Bunlardan 70 ve üstü: Ali 72 ve Can 85. Selin elenir.
4. **Sonuçlar.** Sayı **2**, toplam 72 + 85 = **157**.

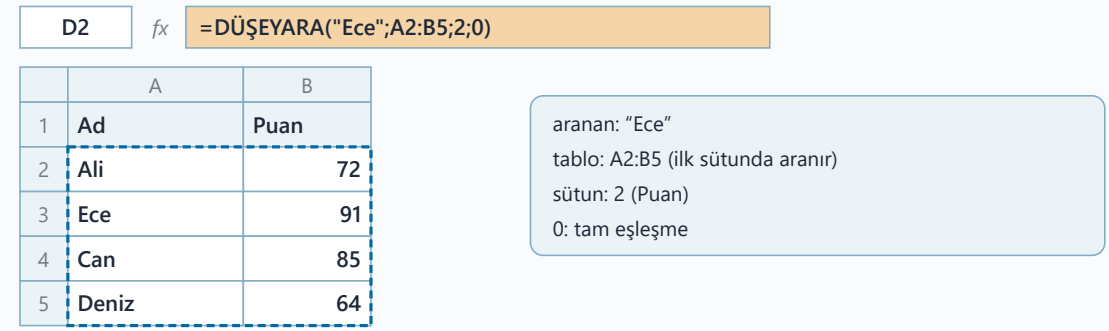
Çok koşullu işlevlerde koşullar VE ile bağlanır: bir satırın sayılması için bütün koşullara uyması gerekir.

3.2.7.3 DÜŞEYARA ile tam eşleşme

Bir tablodan belirli bir kaydın bilgisini getirmek sık yapılan bir iştir. Öğrenci numarasından adı, ürün kodundan fiyatı ya da öğrenci adından puanı bulmak gibi. DÜŞEYARA, bu işi tablonun ilk sütununda arama yaparak çözer.

Tanım 3.2.15

DÜŞEYARA dört bağımsız değişken alır: `=DÜŞEYARA(aranan; tablo; sütun_no; aralık_bak)`. Aranan değeri tablonun *ilk sütununda* yukarıdan aşağı arar; bulduğu satırda tablonun `sütun_no`'ncu sütunundaki değeri döndürür. Son bağımsız değişken 0 (YANLIŞ) ise **tam eşleşme** (*exact match*), 1 (DOĞRU) ise **yaklaşık eşleşme** (*approximate match*) yapılır.

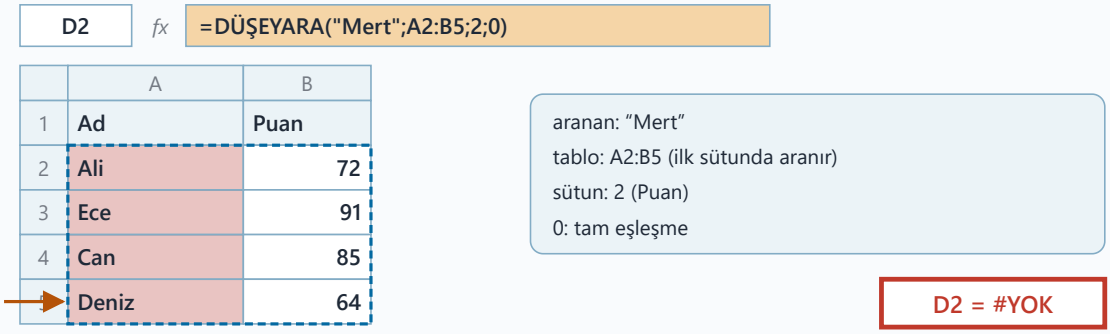


Şekil 3.2.21: `=DÜŞEYARA("Ece";A2:B5;2;0)` formülünün parçaları: aranan "Ece", tablo A2:B5 (arama ilk sütunda yapılır), sütun numarası 2 (Puan) ve tam eşleşme için 0.

Örnek 3.2.13

A2:B5'te Ali 72, Ece 91, Can 85, Deniz 64 var. `=DÜŞEYARA("Ece";A2:B5;2;0)` ve `=DÜŞEYARA("Mert";A2:B5;2;0)` ne döndürür?

1. **A2.** "Ali" ≠ "Ece" → bir alt satıra geçilir.
2. **A3.** "Ece" = "Ece" → bulundu; arama durur.
3. **Sonuç.** Bulunan satır tablonun 2. satırıdır. Tablonun 2. sütunu B olduğu için B3 döner: **91**.
4. **"Mert".** A2, A3, A4 ve A5'in hiçbiri "Mert" değildir. Bütün satırlar denendikten sonra sonuç **#YOK** (değer bulunamadı) olur.



Şekil 3.2.22: "Mert" ilk sütunun hiçbir satırında bulunmaz; tam eşleşmeli DÜŞEYARA #YOK döndürür.

Sütun numarası, sayfanın sütun harfine göre değil tablonun kendi sütunlarına göre sayılır. Tablo C5:F20 ise C 1., D 2., E 3. sütundur. Bu numara formüle sabit yazıldığı için tablonun ortasına yeni bir sütun eklenince numara artık yanlış sütunu gösterir. DÜŞEYARA'nın bir başka sınırı da yalnızca sağa bakabilmesidir. Aranan sütun her zaman tablonun ilk sütunudur; ilk sütunun solundaki bir değer getirilemez.

Sık yapılan hata

Dördüncü bağımsız değişkeni yazmamak. Bu bağımsız değişken atlanırsa Excel yaklaşık eşleşme yapar. Sırasız bir ad listesinde yaklaşık eşleşme hata vermeden yanlış bir kişinin puanını döndürebilir. Ad, kod ya da numara aramalarında son bağımsız değişken her zaman 0 yazılmalıdır.

Tam eşleşmede de aranan değerin yazımı birebir tutmalıdır. Sondaki görünmez bir boşluk (“Ece ”) eşleşmeyi bozar ve sonuç #YOK olur. Büyük–küçük harf ise fark etmez: “ece” araması da 91 döndürür. Listede olmayan değerler için hata yerine bir ileti göstermek istenirse `EĞERYOKSA` kullanılır: `=EĞERYOKSA(DÜŞEYARA(D2;A2:B5;2;0);"Kayıt yok")`. EĞERYOKSA yalnızca #YOK hatasını yakalar; başka hataları gizlemediği için EĞERHATA'dan daha güvenlidir.

3.2.7.4 DÜŞEYARA ile yaklaşık eşleşme

Yaklaşık eşleşmenin doğru kullanım alanı, değerin bir aralığa düştüğü durumlardır: not baremi, vergi dilimi, kargo ücreti tablosu gibi. Bu tablolarda aranan değer çoğu zaman tabloda birebir bulunmaz; hangi dilime düştüğü bulunur.

D2:E6'da bir not baremi olsun. İlk sütunda alt eşikler küçükten büyüğe sıralıdır: 0 → F, 50 → D, 60 → C, 70 → B, 85 → A. Bir puanın harfi şu formülle bulunur:

```
=DÜŞEYARA(B2; $D$2:$E$6; 2; DOĞRU)
```

Barem $ ile sabitlenmiştir, çünkü formül aşağı kopyalanınca her satır aynı bareme bakmalıdır. Yaklaşık eşleşmenin kuralı şudur: ilk sütunda *aranan değerden küçük ya da ona eşit olan en büyük eşik* bulunur ve o satırın değeri döner.

Örnek 3.2.14

Puanlar 72, 85, 49 ve −5 için harf notları nedir?

1. **72.** 0, 50, 60 ve 70 eşikleri 72'den küçük ya da eşittir; 85 değildir. En büyük uygun eşik 70 → **B**.
2. **85.** Eşitlik de sayılır: 85 ≤ 85. En büyük uygun eşik 85 → **A**.
3. **49.** Yalnızca 0 uyar, çünkü 50 > 49 → **F**.
4. **−5.** Hiçbir eşik −5'ten küçük ya da eşit değildir → **#YOK**.

Tam eşleşme kullanılsaydı 72 ve 49 gibi tabloda bulunmayan puanların hepsi #YOK verirdi. −5 sonucunun öğrettiği şey de şudur: baremin en küçük eşiği, olası en küçük değer (burada 0) olmalıdır. Geçersiz puanlar ise 3.2.11.4'te anlatılan veri doğrulamayla baştan engellenebilir.

	A	B	C	D	E
1	Ad	Puan	Harf	Eşik	Harf
2	Ece	72	B	0	F
3	Can	85		50	D
4	Mert	49		60	C
5	Ada	-5		70	B
6				85	A

0 ≤ 72 ? evet
50 ≤ 72 ? evet
60 ≤ 72 ? evet
70 ≤ 72 ? evet
85 ≤ 72 ? hayır

Şekil 3.2.23: 72 puanının barem üzerinde aranması. 0, 50, 60 ve 70 eşikleri 72'den küçük ya da eşittir, 85 değildir; en büyük uygun eşik 70'tir ve harf B olur.

Excel yaklaşık eşleşmede tabloyu satır satır taramaz, **ikili arama** yapar. Tablonun ortasındaki değere bakar, aranan değer daha küçükse üst yarıda, değilse alt yarıda aramayı sürdürür. Bu yöntem büyük tablolarda çok hızlıdır, ama ilk sütunun küçükten büyüğe sıralı olmasına dayanır. Sırasız bir baremde Excel hata vermez; yanlış bir harf döndürebilir. Bu yüzden yaklaşık eşleşme kullanılmadan önce barem sıralanır.

3.2.7.5 ÇAPRAZARA ve FİLTRE

Microsoft 365'te ve Excel 2021'de DÜŞEYARA'nın sınırlarını kaldıran yeni işlevler vardır. Bu işlevler Excel 2019 ve daha eski sürümlerde çalışmaz; bu nedenle dosyayı açacak kişilerin sürümü önemlidir.

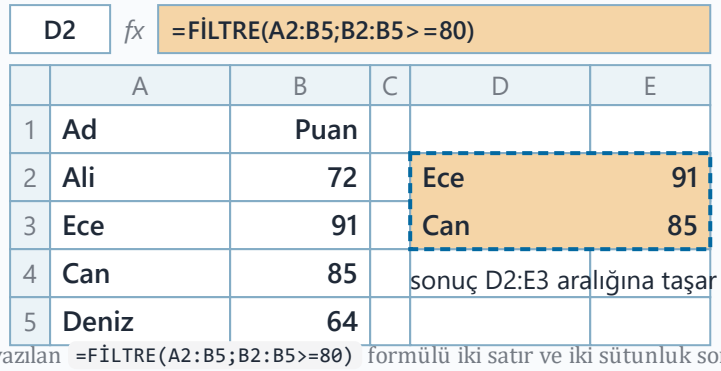
Tanım 3.2.16

ÇAPRAZARA, arama yapılacak diziyle sonucu döndürülecek diziyi ayrı ayrı alır: `=ÇAPRAZARA(aranan; arama_dizisi; döndürülen_dizi)`. Varsayılan olarak tam eşleşme yapar. **FİLTRE**, bir aralıktan koşula uyan satırların tamamını döndürür: `=FİLTRE(dizi; koşul)`.

```
=ÇAPRAZARA("Ece";A2:A5;B2:B5)     → 91
=FİLTRE(A2:B5;B2:B5>=80)        → Ece 91 / Can 85
```

ÇAPRAZARA'da sütun numarası yoktur; döndürülecek sütun doğrudan gösterilir. Tabloya sütun eklenince formül bozulmaz. Arama dizisi döndürülen dizinin sağında da olabilir: adlar B sütununda, puanlar A sütunundaysa `=ÇAPRAZARA("Ece";B2:B5;A2:A5)` yine 91 verir. DÜŞEYARA bu aramayı yapamaz. Aranan bulunamazsa ÇAPRAZARA da #YOK verir; ama dördüncü bağımsız değişkene bir ileti yazılabilir: `=ÇAPRAZARA("Mert";A2:A5;B2:B5;"Kayıt yok")`.

FİLTRE ise tek bir değer değil, bir sonuç tablosu döndürür. Sonuç, formülün yazıldığı hücreden başlayarak gerektiği kadar satıra ve sütuna yayılır. Bu davranışa **taşma** (*spill*) denir.



Şekil 3.2.24: D2'ye yazılan `=FİLTRE(A2:B5;B2:B5>=80)` formülü iki satır ve iki sütunluk sonucu D2:E3 aralığına taşırır: Ece 91 ve Can 85.

Taşan sonucun yolunda dolu bir hücre varsa Excel sonucu yerleştiremez ve `#TAŞMA!` hatası verir. Yoldaki hücre boşaltılınca sonuç kendiliğinden yerine oturur. Taşan aralığın yalnızca ilk hücresinde formül vardır; öteki hücreler formül çubuğunda soluk renkle aynı formülü gösterir ve tek tek düzenlenemez. Kaynak veri değişince taşan sonuç da güncellenir: Deniz'in puanı 64'ten 82'ye çıkarsa FİLTRE sonucu üç satıra genişler.

Sık yapılan hata

Taşan bir sonucun altına başka veri yazmak. FİLTRE bugün iki satır döndürüyor olabilir, ama veri değişince beş satıra çıkabilir. Hemen altına yazılmış bir toplam satırı bu durumda sonucun yolunu keser ve bütün sonuç #TAŞMA! olur. Taşan formüllerin altında ve sağında yeterli boş alan bırakılmalıdır.

3.2.8 Tarih, saat ve formül hataları

Tarih ve saat hesapları ilk bakışta karışık görünür, ama Excel'in onları nasıl sakladığı bilinince basit aritmetiğe dönüşür. Formül hataları da benzer biçimde rastgele değildir; her hata değeri belirli bir nedeni haber verir. Bu kesim önce tarih ve saatin sayı olarak saklanmasını, sonra hata değerlerini ve düzeltme yollarını anlatır.

3.2.8.1 Tarih ve saat

Tanım 3.2.17

Excel her tarihi bir **seri değer** (*date serial number*) olarak saklar: 01.01.1900 tarihi 1'dir ve sonraki her gün bir fazlasıdır. Saat, günün kesri olarak saklanır: 0,5 yarım gün, yani 12:00'dir.



Şekil 3.2.25: Tarihler seri sayı olarak saklanır. 01.03.2024 = 45352, 15.03.2024 = 45366; ikisinin farkı `=B2-A2` ile 14 gün bulunur. Saat günün kesridir: 0,5 → 12:00.

Bu gösterimin yararı açıktır: tarihler sayı olduğu için aritmetik işlemlerle hesaplanır. İki tarihin farkı aradaki gün sayısını verir. Bir tarihe sayı eklemek ileri bir tarihi verir: 15.03.2024 + 30 = 14.04.2024. Tarih hücresi Genel biçime çevrilince saklanan seri değer görünür; bu, bir hücrede gerçekten tarih mi metin mi bulunduğunu anlamanın kolay bir yoludur.

Örnek 3.2.15

A2'de 01.02.2024, B2'de 01.03.2024 var. `=B2-A2` ne verir? Aynı ay aralığı 2023'te olsaydı sonuç ne olurdu?

1. **Seri değerler.** 01.02.2024 = 45323, 01.03.2024 = 45352.
2. **Fark.** 45352 − 45323 = **29** gün. 2024 artık yıldır; şubat 29 gün çeker.
3. **2023.** 2023 artık yıl değildir; aynı fark **28** gündür.

Sonuç hücresi bazen kendiliğinden tarih biçimini alır ve 29 yerine 29.01.1900 gibi anlamsız bir tarih görünür. Bu durumda yalnızca hücrenin sayı biçimi Genel ya da Sayı yapılır; hesap doğrudur.

Tarih ve saatle ilgili başlıca işlevler ve kurallar şunlardır:

- `=BUGÜN()` bugünün tarihini, `=ŞİMDİ()` tarih ve saati verir. İkisi de her yeniden hesaplamada güncellenir. Bir gecikme takibinde `=BUGÜN()-C2` formülü teslim tarihinden bu yana geçen günü her gün yeniden bulur.
- `=TARİH(2024;3;15)` yıl, ay ve günden bir tarih kurar. Ayrı sütunlarda duran yıl, ay ve gün bilgisini birleştirmek için kullanılır.
- Metin olarak saklanmış bir tarih (“15.03.2024” sola yaslı duruyorsa) hesaba katılmaz; çıkarmada `#DEĞER!` hatası verebilir. Böyle veriler genellikle başka bir sistemden aktarılırken oluşur.
- Saatler de günün kesri olduğu için toplanabilir. 09:00 ile 11:45 arasındaki süre `=B2-A2` ile bulunur ve 02:45 görünür; bu, 2,75 saat ya da 165 dakikadır.

Örnek 3.2.16

Bir laboratuvar asistanı beş gün boyunca her gün 9 saat çalışmıştır. Günlük süreler B2:B6'da 09:00 olarak yazılıdır. `=TOPLA(B2:B6)` ne gösterir? Saat ücreti 150 ₺ ise haftalık ücret nasıl hesaplanır?

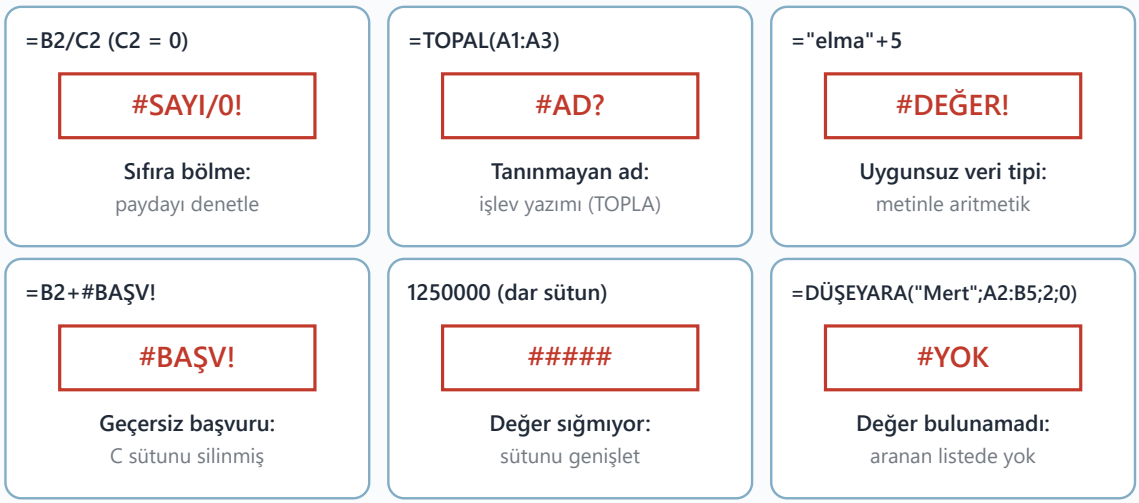
1. **Toplam.** 5 × 9 = 45 saat. Saklanan değer 45 / 24 = 1,875 gündür.
2. **Standart saat biçimi.** `ss:dd` biçimi tam günleri atar ve yalnızca kalan saati gösterir: 45 saat = 1 gün + 21 saat → hücrede **21:00** görünür.
3. **Doğru biçim.** İsteğe Uyarlanmış biçim kodu `[s]:dd` yapılır. Köşeli parantez, saatlerin 24'te sıfırlanmamasını sağlar: **45:00**.
4. **Ücret.** Saklanan değer gün cinsindendir. Saate çevirmek için 24 ile çarpılır: `=B7*24*150` = 1,875 × 24 × 150 = **6750 ₺**.

Saat toplamını doğrudan 150 ile çarpmak (1,875 × 150 = 281,25) yanlış sonuç verir. Gün cinsinden saklanan süre önce saate çevrilmelidir.

3.2.8.2 Hata değerleri

Tanım 3.2.18

Hata değeri (*error value*), bir formül hesaplanamadığında hücrede görünen ve `#` ile başlayan özel sonuçtur. Her hata değeri farklı bir sorun türünü gösterir.



Şekil 3.2.26: Sık görülen hata değerleri ve nedenleri: sıfıra bölme (#SAYI/0!), tanınmayan ad (#AD?), uygunsuz veri tipi (#DEĞER!), geçersiz başvuru (#BAŞV!), sütuna sığmayan değer (#####) ve bulunamayan değer (#YOK).

Hata	Örnek	Neden	Düzeltme
`#SAYI/0!`	`=B2/C2`, C2 = 0	Sıfıra bölme; boş hücre de 0 sayılır.	Paydanın denetimi ya da `=EĞER(C2=0;"";B2/C2)`.
`#AD?`	`=TOPAL(B2:B4)`	Excel adı tanımıyor: yazım yanlışı ya da tırnaksız metin.	`=TOPLA(B2:B4)`; metinler tırnak içinde.
`#DEĞER!`	`=B2*2`, B2 = "12 TL"	Metinle aritmetik işlem.	Hücrede yalnız 12; birim sayı biçimiyle gösterilir.
`#BAŞV!`	`=B2+#BAŞV!`	Formülün kullandığı hücre ya da sütun silinmiş.	Ctrl+Z ile geri alma ya da başvuruyu yeniden yazma.
`#YOK`	`=DÜŞEYARA("Selin";A2:B4;2;0)`	Aranan değer listede yok.	Yazımın denetimi, kaydın eklenmesi ya da EĞERYOKSA.
`#####`	Dar sütunda 1250000	Değer sütuna sığmıyor; gerçek bir hata değil.	Sütun genişletilir.
`#TAŞMA!`	`=FİLTRE(…)`	Taşan sonucun yolunda dolu hücre var.	Yoldaki hücre boşaltılır.

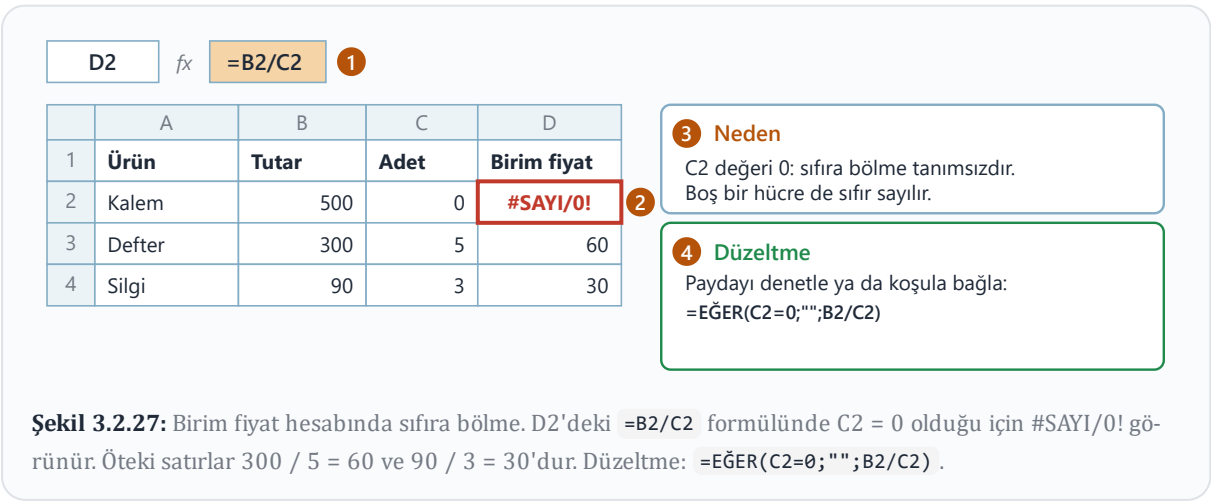
Tablo 3.2.13: Hata değerleri, örnek formüller ve düzeltme yolları.

Bu listeye iki hata daha eklenebilir. `#SAYI!`, sayısal olarak geçersiz bir işlemde görülür; örneğin `=KAREKÖK(-4)` negatif sayının karekökünü alamaz. `#BOŞ!` ise iki aralığın kesişmesi beklenen yerde kesişim olmadığında çıkar ve genellikle iki nokta yerine boşluk yazılmasından kaynaklanır.

Hata değerlerinin önemli bir özelliği yayılmalarıdır. Hatalı bir hücreyi kullanan her formül de hata verir. B2'deki bir `#DEĞER!`, B2'yi toplayan sütun toplamını da, o toplamı kullanan rapor hücresini de hatalı yapar. Bu yüzden bir hata zincirinde düzeltme, zincirin başındaki hücreden başlar.

3.2.8.3 Hataları bulma ve düzeltme

Hatalı bir hücre seçilince yanında küçük bir uyarı simgesi belirir. Simgenin menüsü hatanın türünü açıklar ve hesabı adım adım izlemeye yarayan bir seçenek sunar. Formüller sekmesindeki **Formül Denetleme** grubu da hatanın kaynağını bulmaya yardım eder: **Etkileyenleri İzle** komutu, seçili formülün hangi hücreleri kullandığını oklarla gösterir. Zincirin başına bu oklar izlenerek ulaşılır.



Şekil 3.2.27: Birim fiyat hesabında sıfıra bölme. D2'deki `=B2/C2` formülünde C2 = 0 olduğu için #SAYI/0! görünür. Öteki satırlar 300 / 5 = 60 ve 90 / 3 = 30'dur. Düzeltme: `=EĞER(C2=0;"";B2/C2)`.

Örnek 3.2.17

Şekil 3.2.27'deki tablonun altında D5'e `=ORTALAMA(D2:D4)` yazılıyor. Ne görünür? Hata nasıl giderilir?

1. **Yayılma.** D2'de #SAYI/0! vardır. ORTALAMA, aralıktaki hata değerini atlayamaz; D5'te de **#SAYI/0!** görünür.
2. **Kaynak.** D5 seçilip Etkileyenleri İzle denince oklar D2:D4'ü, D2'den bir kez daha denince B2 ve C2'yi gösterir. Sorun C2'deki 0'dır.
3. **Düzeltme.** D2 `=EĞER(C2=0;"";B2/C2)` yapılır ve aşağı kopyalanır. D2 artık boş bir metin döndürür.
4. **Yeni sonuç.** ORTALAMA metni yok sayar ve yalnız 60 ile 30'un ortalamasını alır: (60 + 30) / 2 = **45**.

EĞER'in boş metin döndürmesi bilinçli bir seçimdir. 0 döndürseydi ORTALAMA bu 0'ı da sayar ve (0 + 60 + 30) / 3 = 30 gibi yanıltıcı bir sonuç verirdi.

#BAŞV! hatasının en yaygın nedeni, formüllerin kullandığı bir sütunun silinmesidir. Excel silinen hücrenin yerine başka bir hücre koyamaz ve başvuruyu formülün içinde #BAŞV! olarak işaretler. Bu hata, silme işlemi hemen `Ctrl+Z` ile geri alınarak giderilir. Bir sütundaki değerlere artık ihtiyaç yoksa, sütunu silmek yerine içeriğini temizlemek başvuruları korur.

Sık yapılan hata

görünen bir hücreyi hatalı sanmak. Bu bir hata değeri değildir; sayı ya da tarih sütuna sığmadığı için Excel onu kesmek yerine işaretlerle doldurur. Değer doğru saklanmıştır ve formüller onu kullanmaya devam eder. Sütun genişletilince değer görünür. Tek istisna negatif bir tarih ya da saat sonucudur; o durumda sütun ne kadar genişletilirse genişletilsin ##### kalır.

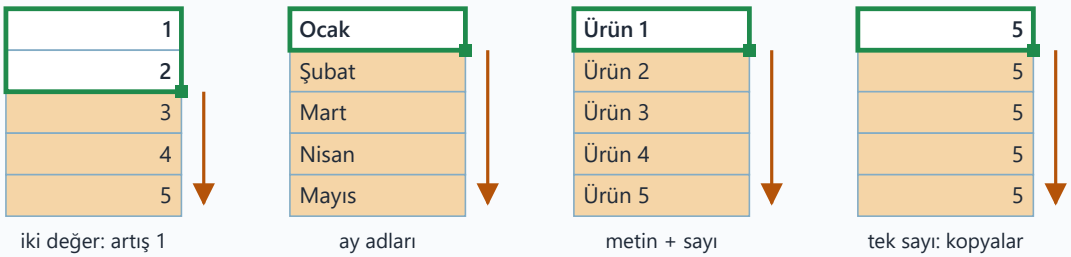
3.2.9 Veri düzenleme araçları

Veri girişini hızlandıran ve tekrarlı işleri azaltan araçlar, Excel'i verimli kullanmanın ikinci ayağıdır. Bu kesim otomatik doldurmayı, sık kullanılan kısayolları, Hızlı Doldurma'yı ve Excel tablosunu anlatır.

3.2.9.1 Otomatik doldurma

Tanım 3.2.19

Otomatik doldurma (*AutoFill*), doldurma tutamacı sürüklenince seçili hücrelerdeki değerin, formülün ya da örüntünün bitişik hücrelere uzatılmasıdır. Excel seçimdeki örüntüyü tanırsa bir **seri** (*data series*) üretir; tanımazsa değeri kopyalar.



Şekil 3.2.28: Doldurma tutamacıyla seriler. 1 ve 2 seçilince artış 1 olarak algılanır ve 3, 4, 5 gelir. "Ocak" ay adlarını, "Ürün 1" metin ve sayı örüntüsünü sürdürür. Tek başına seçilen 5 sayısı ise kopyalanır.

Excel seriyi seçimin ilk iki değerinden çıkarır. 2 ve 5 seçilip sürüklenirse artış 5 − 2 = 3 olarak hesaplanır ve 8, 11, 14 gelir. 10 ve 20 seçilirse 30, 40, 50 gelir. Tarihlerde her sürükleme bir gün ilerler; Pazartesi gibi gün adları ve Ocak gibi ay adları da tanınır. Tek bir sayı seçilip sürüklenirse kopyalanır; sürüklerken Ctrl basılı tutulursa ise 1'er artan seri oluşur.

Sürükleme bitince sağ altta **Otomatik Doldurma Seçenekleri** düğmesi belirir. Bu düğmeyle sonuç değiştirilebilir: Hücreleri Kopyala, Seri Doldur, Yalnızca Biçimlendirmeyi Doldur ya da Biçimlendirmeden Doldur. Bir formül sürüklendiğinde göreli başvurular 3.2.5.2'de anlatılan kurala göre değişir. Uzun bir sütunda sürüklemek yerine doldurma tutamacına çift tıklamak da yeterlidir: formül, solundaki sütunun dolu olduğu son satıra kadar kendiliğinden kopyalanır.

Sık yapılan hata

Doldurulan serinin ilk ve son değerlerini denetlememek. Örneğin "BIL-101" gibi bir ders kodu sürüklenince Excel sondaki sayıyı bir örüntü sanır ve "BIL-102", "BIL-103" üretir; oysa amaç kodun kopyalanmasıydı. Doldurmadan sonra son hücreye bakmak, örüntünün doğru anlaşılıp anlaşılmadığını hemen gösterir.

3.2.9.2 Kısayollar ve Hızlı Doldurma

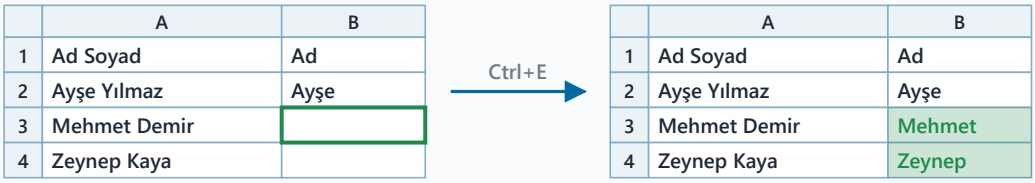
İşlem	Kısayol
Geri al	`Ctrl+Z`

İşlem	Kısayol
Hücreleri Biçimlendir	`Ctrl+1`
Tablo oluştur	`Ctrl+T`
Filtreyi aç / kapat	`Ctrl+Shift+L`
Hızlı Doldurma	`Ctrl+E`
Bul ve Değiştir	`Ctrl+H`
Veri bloğunun sonuna git	`Ctrl+ok tuşu`

Tablo 3.2.14: Veri düzenlemede sık kullanılan kısayollar.

Son satırdaki kısayol uzun listelerde çok zaman kazandırır. `Ctrl+↓`, etkin hücreden başlayarak verinin bittiği son dolu hücreye atlar. `Ctrl+Shift+↓` aynı yolu seçerek gider; binlerce satırlık bir sütun böylece tek tuşla seçilir.

Hızlı Doldurma (*Flash Fill*), bir sütundaki örüntüyü örnekten öğrenir. A sütununda "Ayşe Yılmaz", "Mehmet Demir" ve "Zeynep Kaya" gibi ad soyadlar olsun. B2'ye elle "Ayşe" yazılır, sonra B3'te `Ctrl+E`'ye basılır. Excel örüntüyü "ilk boşluğa kadar olan kısım" diye tahmin eder ve B sütununu Mehmet ve Zeynep ile doldurur. Komut **Veri → Veri Araçları → Hızlı Doldurma** yolundan da verilir.



1) B2'ye örnek yaz, B3'te Ctrl+E

2) Örüntü tahmin edilir: değer, formül değil

Şekil 3.2.29: Hızlı Doldurma. B2'ye örnek olarak "Ayşe" yazılır, B3'te Ctrl+E'ye basılır; Excel örüntüyü tahmin ederek Mehmet ve Zeynep adlarını kendisi doldurur. Sonuçlar formül değil, değerdir.

Hızlı Doldurma'nın iki sınırı vardır. Birincisi, sonuçlar formül değil sabit değerdir; A sütunundaki bir ad değişince B sütunu güncellenmez. İkincisi, Excel örüntüyü tahmin eder ve bu tahmin her satırda doğru olmayabilir. "Ali Rıza Can" gibi iki adlı bir kişide yalnızca "Ali" gelebilir. Doldurulan satırlar bu yüzden tek tek gözden geçirilmelidir.

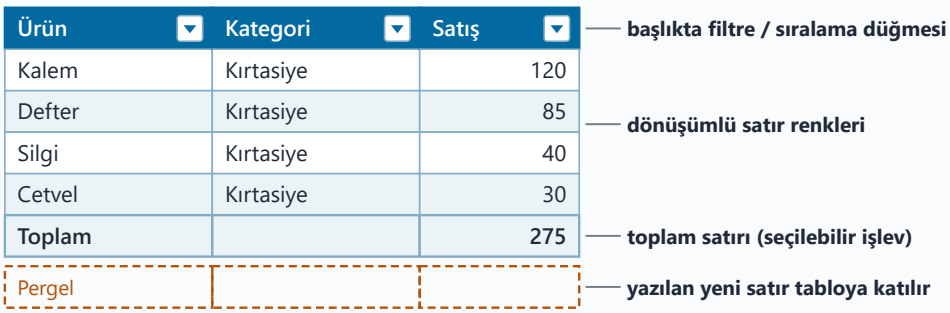
Bul ve Değiştir (`Ctrl+H`) ise bir metni bütün sayfada başka bir metinle değiştirir. Toplu değişiklik yapmadan önce **Seçenekler** düğmesiyle aramanın kapsamı denetlenmelidir: yalnız seçili alanda mı, bütün sayfada mı, bütün kitapta mı? "Tüm hücre içeriğini eşleştir" işaretlenmezse "Ali" araması "Alican"ı da bulur ve değiştirir.

3.2.9.3 Excel tablosu

Başlıklı bir veri aralığı, Excel'in **tablo** özelliğiyle akıllı bir yapıya dönüştürülebilir. Aralığın içinde bir hücre seçilip **Ekle → Tablo** komutu verilir ya da `Ctrl+T`'ye basılır. Açılan kutuda ilk satırın başlık olduğunu belirten seçenek işaretlenir ve Tamam denir.

Tanım 3.2.20

Excel tablosu (*table*), başlıklı sütunlardan oluşan bir aralığın adlandırılmış, kendiliğinden genişleyen bir veri yapısına dönüştürülmüş hâlidir. Tablonun sütunlarına formüllerde adresle değil adla başvurulabilir; buna **yapılandırılmış başvuru** (*structured reference*) denir.



Şekil 3.2.30: Excel tablosu. Başlıklarda filtre ve sıralama düğmeleri, dönüşümlü satır renkleri ve satış toplamını (275) gösteren toplam satırı vardır. Tablonun hemen altına yazılan yeni satır (Pergel) tabloya kendiliğinden katılır.

Tabloya dönüştürmenin yararları şunlardır:

- **Otomatik genişleme.** Tablonun hemen altına ya da sağına yazılan yeni satır ve sütun tabloya katılır. Tabloya başvuran formüller, grafikler ve Özet Tablolar yeni veriyi de kapsar.
- **Filtre ve sıralama.** Her başlık hücresine bir açılır düğme eklenir.
- **Yapılandırılmış başvuru.** Tablonun adı Tablo1, sütunlarından biri Satış ise `=TOPLA(Tablo1[Satış])` bütün satış sütununu toplar. Bir satırın içinde aynı satırdaki değere `=[@Adet]*[@Fiyat]` biçiminde başvurulur. Formül bir hücreye yazılınca bütün sütuna kendiliğinden kopyalanır.
- **Toplam satırı. Tablo Tasarımı** sekmesindeki Toplam Satırı kutusu işaretlenince tablonun altına bir özet satırı eklenir. Her sütun için toplam, ortalama, sayı gibi işlevlerden biri seçilir.
- **Stil.** Bantlı satırlar ve tutarlı renkler okumayı kolaylaştırır.

Şekil 3.2.30'daki satış değerleri 120, 85, 40 ve 30'dur; toplam satırı 120 + 85 + 40 + 30 = 275 gösterir. Pergel satırına bir satış değeri yazıldığı anda toplam satırı da güncellenir. Aynı tablo düz bir aralık olsaydı toplam formülünün aralığının elle genişletilmesi gerekirdi.

3.2.10 Sıralama ve filtreleme

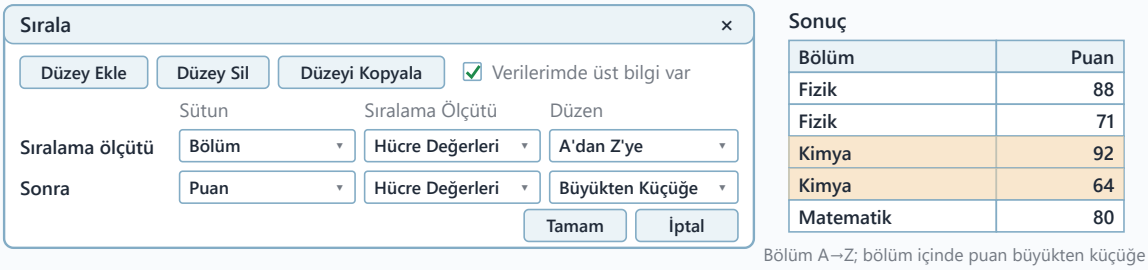
Bir listeden bilgi çıkarmanın ilk adımı, onu bakılan soruya göre düzenlemektir. Sıralama kayıtların sırasını değiştirir, filtreleme yalnızca belirli kayıtları gösterir. İkisi de verinin kendisini değiştirmez; ama sıralama yanlış yapılırsa veriyi kalıcı olarak bozabilir.

3.2.10.1 Sıralama

Tanım 3.2.21

Sıralama (*sort*), bir listedeki kayıtların, satır bütünlüğü korunarak bir ya da birkaç sütunun değerlerine göre yeniden düzenlenmesidir. Birden çok sütuna göre yapılan sıralamaya **çok düzeyli sıralama** (*multi-level sort*) denir.

Sıralama yönü veri tipine göre adlandırılır. Metinler A'dan Z'ye ya da Z'den A'ya, sayılar küçükten büyüğe ya da büyükten küçüğe, tarihler eskiden yeniye ya da yeniden eskiye sıralanır. Tek bir sütuna göre sıralamak için o sütunda bir hücre seçilip Veri sekmesindeki A→Z ya da Z→A düğmesine basılır. Çok düzeyli sıralama için **Veri → Sırala** iletişim kutusu açılır.



Şekil 3.2.31: Sırala iletişim kutusu. Birinci düzey Bölüm, A'dan Z'ye; **Düzey Ekle** ile eklenen ikinci düzey Puan, büyükten küçüğe. Sağda sonuç: Fizik 88, 71; Kimya 92, 64; Matematik 80.

Sırala kutusunda **Verilerimde üst bilgi var** seçeneği işaretli olmalıdır. Bu seçenek başlık satırını sıralamanın dışında tutar. İşaret kalkarsa "Ad, Bölüm, Puan" başlıkları da veri gibi sıralanır ve listenin ortasına karışabilir. Excel ayrıca aynı sütunda sayılarla metin olarak saklanmış sayıları ayrı gruplar hâlinde sıralar; tutarsız veri tipi sıralamayı bozar.

Türkçe dil ayarında metinler Türk alfabesine göre sıralanır: Ç, C'den; Ğ, G'den; I, H'den; İ, I'dan; Ö, O'dan; Ş, S'den; Ü, U'dan sonra gelir. Selin, Can, Çağla, Ali, İrem ve Burak adları A'dan Z'ye sıralanınca Ali, Burak, Can, Çağla, İrem, Selin elde edilir. Harfleri bilgisayardaki kod sırasına göre sıralayan bir program ise Ç ve İ'yi alfabenin sonuna atar ve Ali, Burak, Can, Selin, Çağla, İrem sırasını verir.

3.2.10.2 Sıralamada satır bütünlüğü

Bir listede her satır bir kaydın bütün bilgisini taşır. Sıralamada bu satırlar bir bütün olarak yer değiştirmelidir. Ali'nin bölümü ve puanı, sıralamadan sonra da Ali'nin satırında kalmalıdır. Şekil 3.2.32 beş öğrencilik bir listenin puana göre doğru sıralanmasını gösterir.

	A	B	C	
1	Ad ▾	Bölüm ▾	Puan ↓	
2	Ece	Bilgisayar	91	(eski 3. satır)
3	Can	Elektrik	85	(eski 4. satır)
4	Burak	Makine	78	(eski 6. satır)
5	Ali	Elektrik	72	(eski 2. satır)
6	Deniz	Bilgisayar	64	(eski 5. satır)

Sırala: Puan
Büyükten Küçüğe
satırlar bütün taşınır

Şekil 3.2.32: Veri → Sırala ile Puan'a göre büyükten küçüğe sıralama. Satırlar bir bütün olarak taşınmıştır; her ad kendi bölümü ve puanıyla kalır. Sağdaki notlar her satırın eski yerini gösterir.

Örnek 3.2.18

Ali (Elektrik, 72), Ece (Bilgisayar, 91), Can (Elektrik, 85), Deniz (Bilgisayar, 64) ve Burak (Makine, 78) önce Bölüm'e göre A'dan Z'ye, aynı bölümdekiler Puan'a göre büyükten küçüğe sıralanıyor. Sonuç nedir?

1. **Birinci düzey.** Bölümler alfabetik olarak Bilgisayar < Elektrik < Makine sırasındadır. Önce Bilgisayar öğrencileri, sonra Elektrik, en son Makine gelir.
2. **İkinci düzey, yalnız aynı bölüm içinde.** Bilgisayar: Ece 91 > Deniz 64. Elektrik: Can 85 > Ali 72. Makine'de tek kişi vardır: Burak 78.
3. **Sonuç.** Ece, Deniz, Can, Ali, Burak.

İkinci düzey yalnızca birinci düzeyde eşit olan satırlar arasında sıra belirler. Burak'ın puanı Deniz'den ve Ali'den yüksektir, ama Makine alfabede sonra geldiği için listenin sonunda kalır.

Sıralamanın en tehlikeli hatası, yalnızca bir sütunu seçip sıralamaktır. Excel bu durumda bir **Sıralama Uyarısı** gösterir ve iki seçenek sunar: **Seçimi genişlet** ve **Geçerli seçimle devam et**. İkinci seçenek seçilirse yalnızca o sütun sıralanır; öteki sütunlar yerinde kalır.

	A	B	C
1	Ad ▾	Bölüm ▾	Puan ↓
2	Ali	Elektrik	91
3	Ece	Bilgisayar	85
4	Can	Elektrik	78
5	Deniz	Bilgisayar	72
6	Burak	Makine	64

HATA: satırlar bozuldu
Yalnızca Puan sütunu seçildi,
"Geçerli seçimle devam et"
denildi

Şekil 3.2.33: Yanlış kullanım. Yalnızca Puan sütunu seçilip "Geçerli seçimle devam et" denmiştir. Puanlar büyükten küçüğe dizilmiş, adlar yerinde kalmıştır: Ali artık 91, Ece 85 görünür. Ad–puan eşleşmesi bozulmuştur.

Sık yapılan hata

Sıralama uyarısında “Geçerli seçimle devam et” seçeneğini seçmek. Ortaya çıkan tablo düzgün görünür ve hiçbir hata iletisi yoktur; ama kayıtlar birbirine karışmıştır. Dosya kaydedilip kapatılırsa doğru eşleşme bir daha bulunamaz. Uyarıda her zaman **Seçimi genişlet** seçilmeli, hata fark edilirse hemen Ctrl+Z ile geri alınmalıdır.

3.2.10.3 Filtreleme

Tanım 3.2.22

Filtreleme (*filtering*), bir listede yalnızca belirli ölçütlere uyan satırları gösterip ötekileri geçici olarak gizlemektir. Filtre (*filter*) veriyi silmez ve değiştirmez; yalnızca görünümü daraltır.

Filtre, listenin içinde bir hücre seçilip **Veri → Filtre** komutuyla ya da `Ctrl+Shift+L` ile açılır. Her başlık hücresinin sağında bir açılır düğme belirir. Düğmeye tıklanınca o sütundaki değerlerin listesi açılır. İşareti kaldırılan değerlerin satırları gizlenir.



Şekil 3.2.34: Şehir sütununun filtre menüsü. İzmir'in işareti kaldırılınca 3. ve 6. satırlar gizlenir, silinmez. Filtre açıkken görünen satır numaraları mavi olur, sütun başlığında huni simgesi belirir.

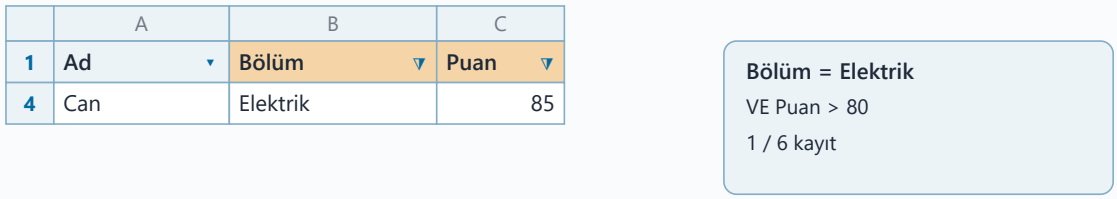
Filtre menüsünde değer listesinin yanında koşul filtreleri de vardır. Sütunun veri tipine göre menüde **Metin Filtreleri** (İçerir, İle Başlar, Eşittir), **Sayı Filtreleri** (Büyüktür, Küçüktür, Arasında, İlk 10) ya da **Tarih Filtreleri** (Bu Ay, Geçen Yıl, Arasında) görünür. Birden çok sütuna filtre uygulanırsa koşullar birlikte daraltır: bir satırın görünmesi için bütün sütunlardaki koşullara uyması gerekir.

Örnek 3.2.19

Altı öğrencilik listede (Ali Elektrik 72, Ece Bilgisayar 91, Can Elektrik 85, Deniz Bilgisayar 64, Burak Makine 78, Selin Elektrik 58) önce Bölüm = Elektrik, sonra Puan > 80 filtresi uygulanıyor. Her adımda ne görünür?

1. **Başlangıç.** Filtre açılmıştır, ama koşul yoktur: 6 kaydın hepsi görünür.

2. **Bölüm = Elektrik.** Ali 72, Can 85 ve Selin 58 görünür; 3 / 6 kayıt. Satır numaraları 1, 2, 4, 7 olur ve atlamalı ilerler.
3. **Puan > 80.** Sayı Filtreleri → Büyüktür → 80. Kalan üç kayıttan yalnız Can (85) koşula uyar: 1 / 6 kayıt.
4. **Temizle. Veri → Temizle** bütün filtreleri kaldırır; 6 kaydın hepsi geri gelir.



Şekil 3.2.35: İki koşul birlikte: Bölüm = Elektrik ve Puan > 80. Yalnızca 4. satırdaki Can görünür; iki başlıkta da huni simgesi vardır.

Filtrelenmiş bir listede kopyalama, silme ve biçimlendirme yalnızca görünen satırları etkiler. Görünen satırlar seçilip kopyalanırsa filtreyle gizlenen satırlar kopyalanmaz; filtre açıkken silinen satırlar da yalnızca görünenlerdir. Elle gizlenmiş satırlarda ise durum farklıdır: böyle bir aralık kopyalanınca gizli satırlar da kopyalanır.

3.2.10.4 Filtre ve ALTTOPLAM

Filtrelenmiş bir listenin altındaki TOPLA formülü ne verir? Beklenen, görünen satırların toplamıdır. Oysa TOPLA gizli satırları da toplar. Filtreye duyarlı özet için ALTTOPLAM işlevi kullanılır.

Tanım 3.2.23

ALTTOPLAM (*SUBTOTAL*), filtreyle gizlenen satırları hesaba katmadan özet değer hesaplayan işlevdir: `=ALTTOPLAM(işlev_sayısı; aralık)`. İşlev sayısı hangi hesabın yapılacağını seçer: 1 ortalama, 2 sayı, 4 en büyük, 5 en küçük, 9 toplam.

Altı kayıtlık yeni bir liste ele alalım: Selin (Bilgisayar, 78), Can (Elektrik, 85), Çağla (Bilgisayar, 91), Ali (Elektrik, 64), İrem (Makine, 70), Burak (Bilgisayar, 55). Puanlar C2:C7'dedir. Listenin altına üç formül yazılır:

```
C9:  =TOPLA(C2:C7)
C10: =ALTTOPLAM(9;C2:C7)
C11: =ALTTOPLAM(1;C2:C7)
```

	A	B	C
1	**Ad**	**Bölüm**	**Puan**
2	Selin	Bilgisayar	78
3	Can	Elektrik	85
4	Çağla	Bilgisayar	91
5	Ali	Elektrik	64
6	İrem	Makine	70
7	Burak	Bilgisayar	55

C9: =TOPLA(C2:C7) **443**
C10: =ALTTOPLAM(9;C2:C7) **443**
C11: =ALTTOPLAM(1;C2:C7) **73,8**

Şekil 3.2.36: Filtre uygulanmadan önce. Üç formül de bütün satırları kapsar: TOPLA ve ALTTOPLAM(9) 443, ALTTOPLAM(1) 73,8.

Adım	Görünen kayıtlar	TOPLA	ALTTOPLAM(9)	ALTTOPLAM(1)
Başlangıç	6 kaydın hepsi	443	443	443 / 6 ≈ 73,8
Sırala: Bölüm A→Z, Puan büyükten küçüğe	Çağla 91, Selin 78, Burak 55, Can 85, Ali 64, İrem 70	443	443	73,8
Filtre: Bölüm = Bilgisayar	Çağla 91, Selin 78, Burak 55	443	91 + 78 + 55 = 224	224 / 3 ≈ 74,7
İkinci filtre: Puan > 75	Çağla 91, Selin 78	443	91 + 78 = 169	169 / 2 = 84,5
Temizle	6 kaydın hepsi	443	443	73,8

Tablo 3.2.15: Sıralama ve filtre adımlarında üç formülün değeri.

Tablonun her satırı bir kuralı doğrular. Sıralama sonuçları değiştirmez, çünkü formüller C2:C7 aralığını gösterir ve aralıktaki değerler yalnızca yer değiştirmiştir. Filtre TOPLA'yı da değiştirmez; TOPLA gizli satırları da toplar ve hep 443 verir. ALTTOPLAM ise yalnız görünenleri hesaplar. Filtre temizlenince bütün satırlar geri gelir ve ALTTOPLAM yeniden 443 olur.

	A	B	C	
1	Ad ▾	Bölüm ▾	Puan ▾	ve Puan > 75
2	Çağla	Bilgisayar	91	
3	Selin	Bilgisayar	78	

C9: =TOPLA(C2:C7)	443
C10: =ALTTOPLAM(9;C2:C7)	169
C11: =ALTTOPLAM(1;C2:C7)	84,5

Şekil 3.2.37: İki filtre birlikte: Bölüm = Bilgisayar ve Puan > 75. Yalnız Çağla ve Selin görünür. TOPLA hâlâ 443'tür; ALTTOPLAM(9) 169, ALTTOPLAM(1) 84,5 verir.

ALTTOPLAM'ın işlev sayıları 1–11 ve 101–111 olmak üzere iki gruptur. 9 yalnız filtreyle gizlenen satırları atlar. 109 ise ayrıca elle gizlenen satırları da atlar. Excel tablosunun toplam satırı arka planda ALTTOPLAM'ın 109 sürümünü kullanır; bu yüzden tablo filtrelenince toplam satırı da yalnız görünenleri gösterir.

3.2.11 Koşullu biçimlendirme, Özet Tablo ve veri doğrulama

Bir listenin içindeki önemli değerleri göze görünür kılmak, listeyi gruplara göre özetlemek ve listeye yanlış veri girilmesini önlemek, veri analizinin üç tamamlayıcı işidir. Excel bu işler için sırasıyla koşullu biçimlendirme, Özet Tablo ve veri doğrulama araçlarını sunar.

3.2.11.1 Koşullu biçimlendirme

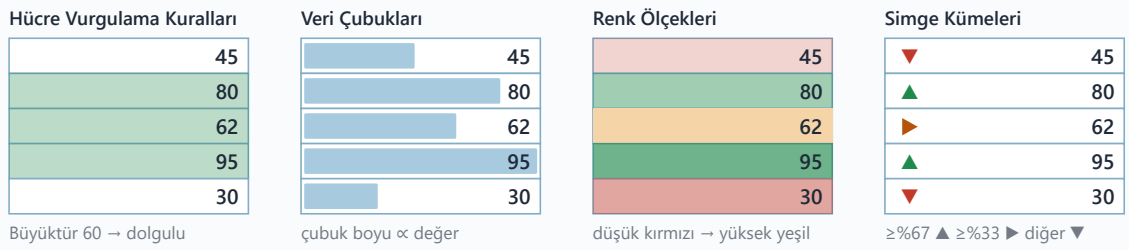
Tanım 3.2.24

Koşullu biçimlendirme (*conditional formatting*), bir hücrenin biçimini o hücrenin değerine bağlayan kuraldır. Değer değişince biçim de kendiliğinden değişir.

Komutlar **Giriş → Koşullu Biçimlendirme** menüsünde toplanır. Başlıca kural türleri şunlardır:

- **Hücre Vurgulama Kuralları.** Belirli bir eşiğe göre biçim uygular: Büyüktür, Küçüktür, Arasında, Eşittir, Metin İçerir, Yinelenen Değerler.
- **İlk/Son Kuralları.** En büyük ya da en küçük birkaç değeri ya da ortalamanın üstündekileri işaretler.
- **Veri çubukları** (*data bars*). Hücrenin içine değerle orantılı bir çubuk çizer.
- **Renk ölçekleri** (*color scales*). Düşük ve yüksek değerleri bir renk geçişiyle gösterir.

- **Simge kümeleri** (*icon sets*). Değerin aralıktaki konumunu ok, daire ya da bayrak gibi simgelerle belirtir.
- **Formül kuralı. Yeni Kural → Biçimlendirilecek hücreleri belirlemek için formül kullan** seçeneğiyle DOĞRU ya da YANLIŞ veren herhangi bir formül koşul olarak yazılabilir.



Şekil 3.2.38: Aynı beş değere (45, 80, 62, 95, 30) dört kural türü uygulanmış: 60'tan büyükleri vurgulama, veri çubukları, renk ölçeği ve simge kümesi.

Formül kuralı en esnek türdür. Örneğin bir öğrenci listesinin *bütün satırını* boyamak için A2:C7 seçilir ve kural formülü `=$C2<60` yazılır. Formül seçimin sol üst hücresine göre yazılır ve öteki hücrelere kopyalanmış gibi uygulanır. `$C` sütunu kilitler, böylece her hücre kendi satırının C sütunundaki puana bakar. Satır numarası serbest olduğu için her satır kendi puanını sınar. Bu, 3.2.5.1'deki karma başvurunun bir başka kullanımıdır.

Koşullu biçimlendirmede ölçülü olmak gerekir. Aynı aralığa beş kural ve on renk uygulanırsa göz önemli değeri seçemez. Renk tek başına anlam taşımamalıdır; renk körlüğü olan okurlar için kırmızı ile yeşili ayırmak güçtür. Bir kuralın yanına açıklama yazmak ya da simge kullanmak bu sorunu azaltır.

3.2.11.2 Simge kümelerinin eşikleri

Yedi öğrencinin puanları B2:B8'de olsun: 72, 91, 85, 64, 78, 58, 45. Bu puanlara dört farklı kural uygulanabilir. Hücre kuralı “Küçüktür 60” seçilirse 58 ve 45 kırmızı dolgu alır. Veri çubuklarında en uzun çubuk en büyük değere (91) aittir. Renk ölçeğinde en düşük değer (45) en açık, en yüksek değer (91) en koyu tonu alır. Simge kümesinde ise her değer üç gruptan birine düşer. Peki gruplar nasıl belirlenir?

Üç simgeli bir kümenin varsayılan eşikleri %67 ve %33'tür. Bu yüzdeler değerlerin kendisine değil, en küçük ile en büyük değer arasındaki aralığa uygulanır:

> eşik = en küçük + yüzde × (en büyük − en küçük)

Örnek 3.2.20

72, 91, 85, 64, 78, 58, 45 puanlarına üç simgeli küme (▲ %67 ve üstü, ● %33–%67, ▼ %33'ün altı) uygulanıyor. Eşikleri ve her puanın simgesini bulalım.

1. **Aralık.** En küçük 45, en büyük 91; aralık 91 − 45 = 46.
2. **Üst eşik.** 45 + 0,67 × 46 = 45 + 30,82 = **75,82** ≈ 75,8.
3. **Alt eşik.** 45 + 0,33 × 46 = 45 + 15,18 = **60,18** ≈ 60,2.
4. **▲ alanlar.** 75,82 ve üstü: 91, 85, 78.
5. **● alanlar.** 60,18 ile 75,82 arası: 72 ve 64.
6. **▼ alanlar.** 60,18'in altı: 58 ve 45.

Eşikler veriyle birlikte değişir. En düşük puan 45'ten 30'a inerse aralık 91 − 30 = 61'e genişler. Üst eşik 30 + 0,67 × 61 ≈ 70,9'a iner ve 72 puan ●'den ▲'ya geçer; 72'nin kendisi değişmediği hâlde simgesi değişmiştir. Sabit eşik isteniyorsa **Kuralları Yönet → Kuralı Düzenle** kutusunda tür "Yüzde" yerine "Sayı" yapılır ve 75 ile 60 gibi sabit değerler yazılır.

	A	B
1	Ad	Puan
2	Ali	● 72
3	Ece	▲ 91
4	Can	▲ 85
5	Deniz	● 64
6	Burak	▲ 78
7	Selin	▼ 58
8	Mert	▼ 45

Simge kümesi
▲ ≥ %67 · ● ≥ %33 · ▼ diğer
eşikler: 75,8 ve 60,2

Şekil 3.2.39: Simge kümesi uygulanmış puanlar. Aralığın (45–91) üst üçte biri ▲, ortası ●, alt kısmı ▼ alır; varsayılan %67 ve %33 eşikleri 75,8 ve 60,2'ye karşılık gelir.

Sık yapılan hata

Simge kümesindeki %67'yi "puanların en iyi %67'si" sanmak. Eşik, değerlerin sıralamasına göre değil, en küçük ile en büyük değer arasındaki uzaklığa göre hesaplanır. Örnekte yedi puandan üçü ▲ almıştır; bu oran %67 ile ilgili değildir. Sıralamaya göre bir eşik isteniyorsa kural türü "Yüzde birlik" seçilir.

3.2.11.3 Özet Tablo

Bir listeyi gruplara göre özetlemek için her grup için ayrı bir ETOPLA ya da EĞERSAY formülü yazılabilir. Grup sayısı arttıkça bu yol yorucu olur. **Özet Tablo** aynı işi formül yazmadan, alanları sürükleyerek yapar.

Tanım 3.2.25

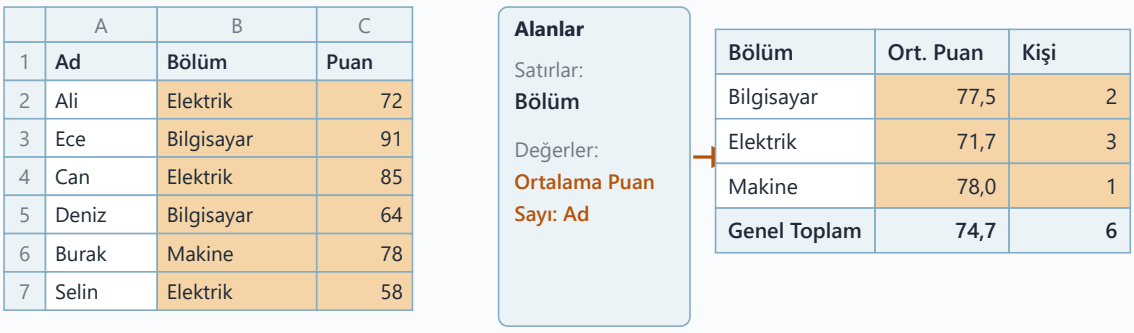
Özet Tablo (*PivotTable*), bir listenin sütunlarını satır, sütun, filtre ve değer alanlarına yerleştirerek gruplu toplam, ortalama, sayı gibi özetler üreten etkileşimli tablodur. Kaynak verinin kendisini değiştirmez.

Özet Tablo oluşturmak için listenin içinde bir hücre seçilir ve **Ekle → Özet Tablo** komutu verilir. Kaynak liste başlıklı olmalı ve içinde boş satır bulunmamalıdır. Excel yeni bir sayfada boş bir özet tablo ve sağda **Özet Tablo Alanları** bölmesini açar. Bölmede kaynak listenin başlıkları alan olarak listelenir; altında dört kutu vardır: Filtreler, Sütunlar, Satırlar ve Değerler.

Örnek 3.2.21

Altı öğrencilik liste (Ali Elektrik 72, Ece Bilgisayar 91, Can Elektrik 85, Deniz Bilgisayar 64, Burak Makine 78, Selin Elektrik 58) bölümlere göre özetleniyor.

1. **Bölüm → Satırlar.** Her bölüm bir kez listelenir: Bilgisayar, Elektrik, Makine ve en altta Genel Toplam.
2. **Puan → Değerler.** Sayısal bir alanın varsayılan özeti toplamdır. Bilgisayar 91 + 64 = 155, Elektrik 72 + 85 + 58 = 215, Makine 78; Genel Toplam 155 + 215 + 78 = 448.
3. **Değer Alanı Ayarları → Ortalama.** Bilgisayar 155 / 2 = 77,5; Elektrik 215 / 3 ≈ 71,7; Makine 78,0; Genel Toplam 448 / 6 ≈ 74,7.
4. **Ad → Değerler.** Metin alanının varsayılan özeti saymaktır: Bilgisayar 2, Elektrik 3, Makine 1; toplam 6 kişi.



Şekil 3.2.40: Özet Tablo'nun son hâli. Satırlarda Bölüm, değerlerde Ortalama Puan ve Sayı: Ad vardır. Elektrik ortalaması 71,7, kişi sayısı 3'tür. Genel Toplam satırında ortalama 74,7, kişi sayısı 6'dır.

Genel Toplam satırındaki ortalama dikkat ister. 74,7, altı puanın ortalamasıdır: 448 / 6. Üç bölüm ortalamasının ortalaması ise (77,5 + 71,7 + 78) / 3 ≈ 75,7'dir. Bu hesapta kalabalık Elektrik grubu ile tek kişilik Makine grubu aynı ağırlıkla sayılır. Özet Tablo doğru ağırlığı kendisi kullanır.

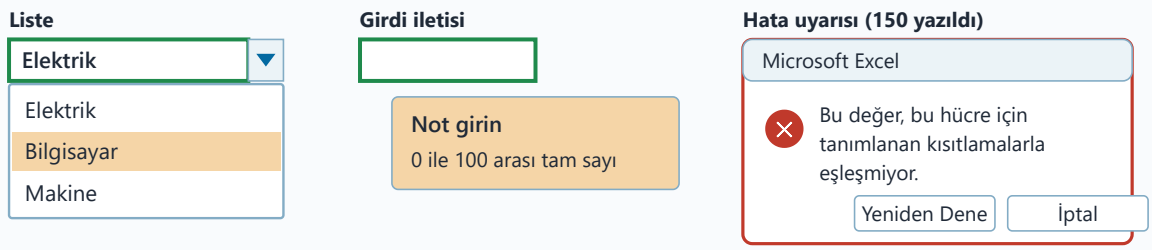
Özet Tablo ile formüller aynı sonucu verir: Elektrik toplamı `=ETOPLA(B2:B7;"Elektrik";C2:C7)` ile de 215 bulunur. Farkı şudur: formüller kaynak veri değişince kendiliğinden güncellenir, Özet Tablo ise güncellenmez. Kaynağa yeni bir kayıt eklenince Özet Tablo'nun üzerinde sağ tıklanıp **Yenile** denmelidir. Kaynak bir Excel tablosu olarak tanımlanmışsa yeni satırlar yenilemede kendiliğinden kapsanır.

3.2.11.4 Veri doğrulama

Analizin kalitesi, girilen verinin kalitesine bağlıdır. Bir not sütununa 150, bir bölüm sütununa "Elektirik" yazılırsa sonraki bütün hesaplar etkilenir. Veri doğrulama, yanlış verinin girildiği anda yakalanmasını sağlar.

Tanım 3.2.26

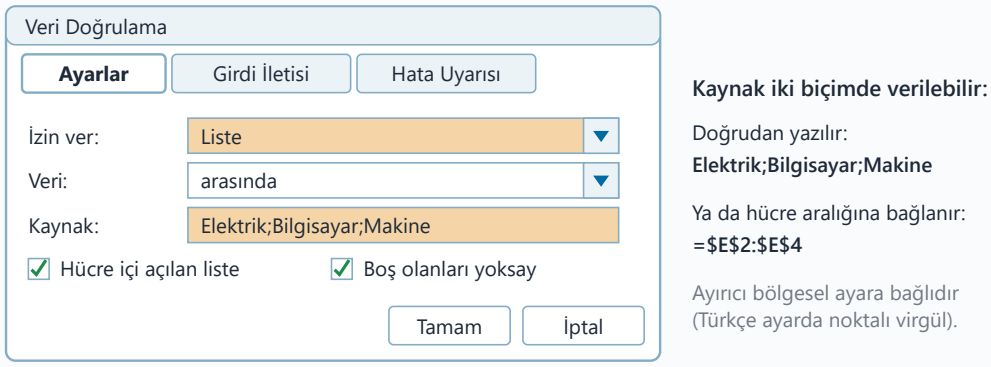
Veri doğrulama (*data validation*), bir hücreye girilebilecek değerleri bir kuralla sınırlar. Kural; bir liste, bir sayı ya da tarih aralığı, bir metin uzunluğu ya da bir formül olabilir. Hücre seçilince görünen açıklamaya **girdi iletisi** (*input message*), kurala uymayan girişte açılan uyarıya **hata uyarısı** (*error alert*) denir.



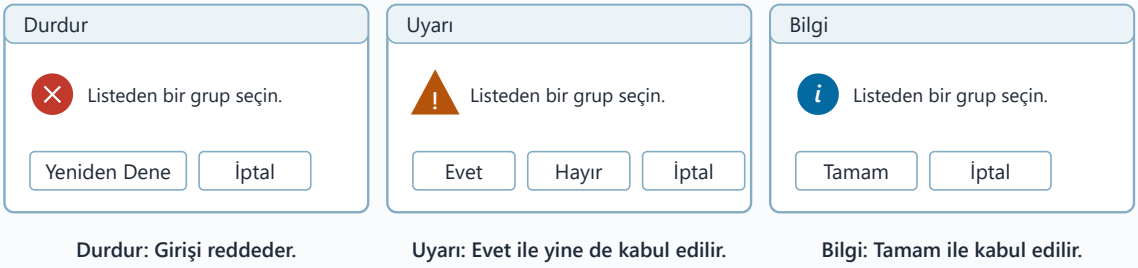
Şekil 3.2.41: Veri doğrulamanın üç yüzü: bölüm hücresinde açılır liste, not hücresinde "0 ile 100 arası tam sayı" girdi iletisi ve 150 yazılınca açılan hata uyarısı.

Bir bölüm sütununa açılır liste eklemek altı adımda yapılır:

1. **Aralığın seçimi.** Doğrulanacak hücreler (örneğin B2:B4) seçilir; **Veri → Veri Araçları → Veri Doğrulama** açılır.
2. **Liste kuralı. Ayarlar** sekmesinde **İzin ver** kutusundan **Liste** seçilir. **Kaynak** kutusuna seçenekler noktalı virgülle yazılır (`Elektrik;Bilgisayar;Makine`) ya da seçeneklerin bulunduğu bir aralık gösterilir (`=$E$2:$E$4`).
3. **Sonuç.** Hücre seçilince sağında bir ok belirir; kullanıcı yalnızca listedeki değerlerden birini seçer.
4. **Hata uyarısı. Hata Uyarısı** sekmesinde stil, başlık ve ileti yazılır: Stil Durdur, başlık "Geçersiz grup", ileti "Listeden bir grup seçin."
5. **Geçersiz giriş.** Listede olmayan "Muz" yazılırsa Durdur stili girişi reddeder; **Yeniden Dene** hücreyi düzenlemeye döndürür.
6. **Stilin seçimi.** Uyarının ne kadar katı olacağı stile bağlıdır.



Şekil 3.2.42: Veri Doğrulama kutusunun Ayarlar sekmesi. İzin ver: Liste; Kaynak: Elektrik;Bilgisayar;Makine. Kaynak bir hücre aralığına da bağlanabilir (`=$E$2:$E$4`). Ayırıcı bölgesel ayara bağlıdır; Türkçe ayarda noktalı virgüldür.



Şekil 3.2.43: Hata uyarısının üç stili. Durdur girişi reddeder. Uyarı, Evet ile yine de kabul eder. Bilgi yalnızca bilgilendirir; Tamam ile değer kabul edilir.

Kaynağı bir hücre aralığına bağlamak, listenin bakımını kolaylaştırır. Yeni bir bölüm eklenince yalnızca E sütunundaki liste güncellenir. Aralık $ işaretleriyle sabitlenir; böylece kural birden çok hücreye uygulanınca her hücre aynı listeye bakar.

Veri doğrulamanın bir sınırı vardır: kural yalnızca elle yazılan girişleri denetler. Başka bir yerden kopyalanıp yapıştırılan değerler kuralı aşabilir. Daha önce girilmiş geçersiz değerler de kendiliğinden işaretlenmez. Bunları bulmak için **Veri Doğrulama → Geçersiz Veriyi Daire İçine Al** komutu kullanılır.

3.2.12 Grafikler

Bir tablo kesin sayıları verir; bir grafik ise eğilimi, karşılaştırmayı ve aykırı değeri bir bakışta gösterir. Ama yanlış türde ya da özensiz çizilmiş bir grafik veriyi gizleyebilir, hatta yanlış bir izlenim bırakabilir. Bu kesim, verinin amacına uygun grafiği seçmeyi ve grafiği anlaşılır kılmayı anlatır.

3.2.12.1 Grafik türünü seçmek

Grafik türü, verinin görünüşüne göre değil, grafiğin yanıtlayacağı soruya göre seçilir. Tablo 3.2.16 temel soruları ve onlara uyan grafik türlerini eşleştirir.

Amaç	Uygun grafik	Örnek soru
Kategorileri karşılaştırmak	**Sütun ya da çubuk grafik**	Hangi bölümün ortalaması daha yüksek?
Zamana göre değişimi göstermek	**Çizgi grafik**	Aylık satışlar nasıl değişti?
Parça–bütün ilişkisi	Az kategoriyle **pasta grafik** ya da yüzde çubuk	Bütçenin yüzde kaçı malzemeye gitti?
İki sayısal değişkenin ilişkisi	**Dağılım (XY) grafiği**	Çalışma süresi arttıkça not artıyor mu?
Bir değişkenin dağılım biçimi	**Histogram**	Puanlar hangi aralıklarda yoğunlaşıyor?

Tablo 3.2.16: Amaca göre grafik seçimi.

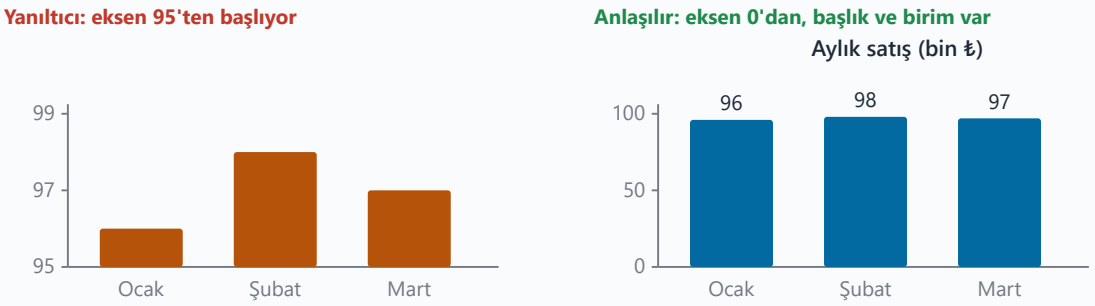
Grafik, verinin bulunduğu aralık seçilip **Ekle** sekmesinin Grafikler grubundan bir tür seçilerek eklenir. **Önerilen Grafikler** düğmesi, Excel'in veriye bakarak önerdiği türleri gösterir. Sütun grafikte kategoriler yatay eksene, değerler dikey eksene yerleşir. Çubuk grafik aynı şeyi yatay çubuklarla gösterir ve uzun kategori adlarında daha okunur.

Pasta grafik en çok yanlış kullanılan türdür. Parçaların toplamı anlamlı bir bütün oluşturmuyorsa (örneğin beş öğrencinin puanları) pasta grafik anlamsızdır. Kategori sayısı beşi altıyı geçerse dilimlerin büyüklüğünü gözle karşılaştırmak zorlaşır; bu durumda sıralı bir çubuk grafik daha iyidir. Dağılım grafiği ile çizgi grafik de karıştırılır. Çizgi grafikte yatay eksen kategoridir ve noktalar eşit aralıklı dizilir. Dağılım grafiğinde ise yatay eksen gerçek bir sayı eksenidir. Ölçümler eşit aralıklı değilse, örneğin 1, 2, 5 ve 10 saniyede alınmışsa, çizgi grafik zaman eksenini çarpıtır.

3.2.12.2 Anlaşılır ve dürüst grafik

İyi bir grafikte okur bir bakışta neyin gösterildiğini anlar. Bunun için grafiğin şu öğeleri gözden geçirilir:

- **Başlık.** Grafiğin neyi gösterdiğini söyler: “Grafik1” değil, “Aylık satış (bin ₺)”.
- **Eksen ve birim.** Eksen başlıkları değerlerin anlamını ve birimini belirtir.
- **Kaynak veri.** Doğru aralığın ve serinin seçildiği denetlenir; toplam satırı yanlışlıkla bir kategori gibi çizilmemelidir.
- **Ölçek.** Eksen sınırları farkları olduğundan büyük ya da küçük göstermeyecek biçimde ayarlanır.
- **Renk.** Serileri ayırmaya yarar, ama tek başına anlam taşımamalıdır.
- **Sadelik.** Üç boyutlu efektler, gölgeler ve gereksiz süsler kaldırılır; bunlar değerleri okumayı zorlaştırır.



Şekil 3.2.44: Aynı veri iki grafikte. Solda eksen 95'ten başlar ve küçük farklar çok büyük görünür. Sağda eksen 0'dan başlar, başlık ve birim vardır: Ocak 96, Şubat 98, Mart 97 bin ₺.

Örnek 3.2.22

Ocak, şubat ve mart satışları 96, 98 ve 97 bin ₺'dir. Dikey eksen 95'ten başlatılırsa şubat sütunu ocak sütununun kaç katı uzunlukta görünür? Gerçek fark yüzde kaçtır?

1. **Görünen boylar.** Eksen 95'ten başladığı için sütun boyu değerin 95'i aşan kısmıdır: ocak 96 − 95 = 1, şubat 98 − 95 = 3, mart 97 − 95 = 2.
2. **Görünen oran.** Şubat sütunu ocak sütununun 3 / 1 = **3 katı** görünür.
3. **Gerçek oran.** 98 / 96 ≈ 1,021; şubat satışı ocaktan yalnızca yaklaşık **%2,1** fazladır.

Sütun ve çubuk grafiklerde eksen 0'dan başlamalıdır, çünkü okur değeri sütunun boyuyla karşılaştırır. Çizgi grafikte ise eğimi göstermek amaçlandığı için eksenin 0'dan başlaması şart değildir. Excel ekseni bazen kendiliğinden 0'dan başka bir değerden başlatır; bu, **Ekseni Biçimlendir → Sınırlar → En Küçük** ayarından düzeltilir.

3.2.13 Raporlama, çalışma kitabı yönetimi ve veri düzeni

Bir çalışma kitabı çoğu zaman başkalarıyla paylaşılır, yazdırılır ya da yıllarca kullanılır. Bu kesim kitabı yazdırmaya hazırlamayı, sayfaları yönetmeyi, değişiklikten korumayı ve en baştan analize uygun bir veri düzeni kurmayı anlatır.

3.2.13.1 Yazdırma ve sayfa düzeni

Ekranda düzgün görünen bir tablo kâğıda basılınca birkaç sayfaya bölünebilir; son sütun ayrı bir sayfaya düşebilir. Yazdırmadan önce çıktı **Dosya → Yazdır** ekranındaki önizlemede denetlenir. Sayfa düzeniyle ilgili ayarlar **Sayfa Düzeni** sekmesinde toplanır:

- **Yazdırma alanı** (*print area*). Yalnızca seçili aralığın basılmasını sağlar: Sayfa Düzeni → Yazdırma Alanı → Yazdırma Alanını Belirle.
- **Yönlendirme.** Geniş tablolar için yatay sayfa seçilir.

- **Ölçekleme. Sığacak Şekilde Ölçeklendir** grubundaki Genişlik kutusu “1 sayfa” yapılırsa tablo sayfa genişliğine sığacak kadar küçültülür.
- **Başlıkları yinele. Başlıkları Yazdır** düğmesiyle açılan kutuda **Üstte yinelenecek satırlar** olarak 1. satır seçilir; çok sayfalı çıktının her sayfasında başlık satırı basılır.
- **Sayfa sonları. Görünüm → Sayfa Sonu Önizleme** sayfaların nerede bölündüğünü mavi çizgilerle gösterir; çizgiler sürüklenerek bölünme yeri değiştirilebilir.

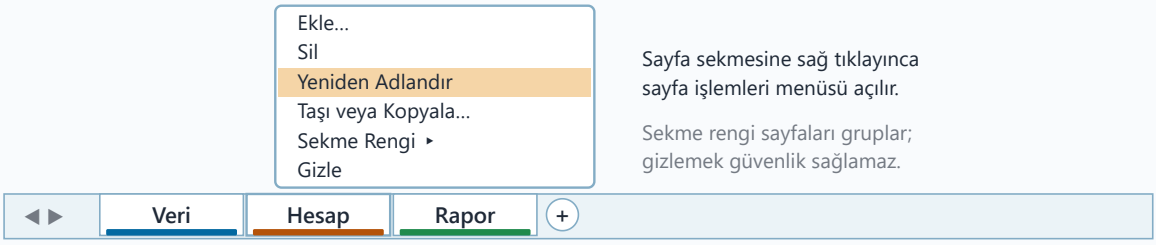


Şekil 3.2.45: Sayfa Sonu Önizleme. Mavi kesikli çizgiler sayfaların bölündüğü yeri gösterir. Başlıkları yinele ayarıyla 1. satır her sayfanın başında basılır.

Ölçekleme küçültme demektir ve küçültmenin bir sınırı vardır. Yatay bir A4 sayfasının basılabilir genişliği yaklaşık 27,7 cm olsun. 35 cm genişliğindeki bir tablo bu genişliğe sığdırılınca ölçek 27,7 / 35 ≈ %79'a düşer. Yazı da aynı oranda küçülür; 11 punto yazı yaklaşık 8,7 puntoya iner. Ölçek %60'ın altına inerse çıktı okunmaz hâle gelir; bu durumda tabloyu bölmek ya da gereksiz sütunları yazdırma alanından çıkarmak daha iyidir.

3.2.13.2 Çalışma kitabı yönetimi

Sayfa işlemleri, sayfa sekmesine sağ tıklanınca açılan menüden yapılır: Ekle, Sil, Yeniden Adlandır, Taşı veya Kopyala, Sekme Rengi, Gizle ve Göster.



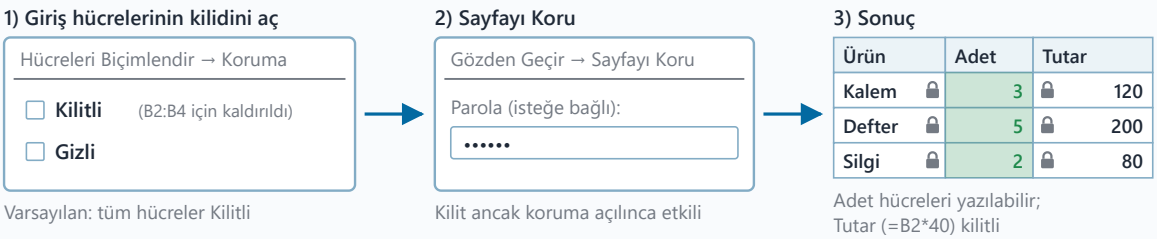
Şekil 3.2.46: Sayfa sekmeleri (Veri, Hesap, Rapor) ve sağ tık menüsü. Sekme rengi sayfaları gruplar; gizlemek güvenlik sağlamaz.

- **Yeniden adlandırma.** Sekmeye çift tıklanır ve içeriği anlatan kısa bir ad yazılır. Formüllerdeki sayfa başvuruları yeni ada göre kendiliğinden güncellenir.

- **Taşı veya Kopyala.** Sayfayı kitabın içinde başka bir konuma ya da başka bir kitaba aktarır. "Kopya oluştur" işaretlenirse sayfa taşınmaz, kopyalanır.
- **Sekme rengi.** Veri, hesap ve rapor sayfalarını renkle gruplar.
- **Gizleme ve gösterme.** Yardımcı sayfaları görünümden kaldırır. Gizli sayfayı herkes sağ tık → Göster ile geri getirebilir; bu yüzden gizlemek bir güvenlik önlemi değildir.
- **Bağlantılar.** Başka bir çalışma kitabına başvuran formüller o dosyanın konumuna bağlıdır. Kaynak dosya taşınır ya da yeniden adlandırılırsa bağlantı kopar. Bu tür başvurular dikkatle ve az kullanılır.

3.2.13.3 Koruma ve iş birliği

Formüllerin yanlışlıkla silinmesini önlemek için sayfa korunabilir. Sayfa koruması (*sheet protection*) iki adımlıdır. Bunun nedeni, Excel'de her hücrenin varsayılan olarak **Kilitli** işaretli olmasıdır; ama bu kilit, koruma açılmadıkça etkili değildir.



Şekil 3.2.47: Sayfa koruması iki adımdır. Önce kullanıcının yazacağı giriş hücrelerinin (B2:B4) Kilitli işareti kaldırılır; sonra Gözden Geçir → Sayfayı Koru açılır. Sonuçta yalnızca Adet hücreleri yazılabilir, Tutar formülleri kilitli kalır.

1. **Giriş hücrelerinin kilidi.** Kullanıcının değiştirebileceği hücreler (örneğin adetler) seçilir. **Hücreleri Biçimlendir → Koruma** sekmesinde **Kilitli** işareti kaldırılır.
2. **Sayfanın korunması. Gözden Geçir → Sayfayı Koru** komutu verilir; isteğe bağlı bir parola yazılır. Artık kilitli hücreler değiştirilemez, kilidi açılmış hücrelere yazılabilir.

Şekil 3.2.47'deki örnekte tutarlar `=B2*40` gibi formüllerle hesaplanır: 3 × 40 = 120, 5 × 40 = 200, 2 × 40 = 80. Kullanıcı adetleri değiştirebilir, ama formüllere dokunamaz.

Excel'de koruma düzeyleri birbirinden farklıdır:

- **Sayfa koruması** hücrelerin ve bazı işlemlerin değiştirilmesini engeller. Amacı yanlışlıkla yapılan değişiklikleri önlemektir; güçlü bir güvenlik önlemi değildir.
- **Kitap yapısının korunması** (**Gözden Geçir → Çalışma Kitabını Koru**) sayfa eklemeyi, silmeyi, taşımayı ve gizli sayfaları göstermeyi engeller.
- **Dosya şifreleme** (**Dosya → Bilgi → Çalışma Kitabını Koru → Parolayla Şifrele**) dosyanın açılması için parola ister. Gizli bilgi taşıyan dosyalar için gerçek koruma budur. Parola unutulursa dosya kurtarılamaz.

İş birliği için de araçlar vardır. **Açıklamalar** (Gözden Geçir → Yeni Açıklama), bir hücre üzerinde soru sormaya ve yanıtlamaya yarar. Dosya OneDrive ya da SharePoint gibi bir bulut konumunda saklanırsa birkaç kişi aynı anda düzenleyebilir. **Sürüm Geçmişi** de dosyanın önceki hâllerine dönmeyi sağlar.

3.2.13.4 İyi veri düzeni

Bu notta anlatılan araçların çoğu — sıralama, filtre, ETOPLA, DÜŞEYARA, Özet Tablo — verinin belirli bir düzende tutulduğunu varsayar. Düzen bozuksa araçlar ya çalışmaz ya da sessizce yanlış sonuç verir. Analize uygun bir veri sayfası dört kurala uyar:

- Her satır tek bir kaydı, her sütun tek bir değişkeni temsil eder.
- Başlıklar tek satırdadır; kısa, benzersiz ve anlamlıdır. Birim başlıkta belirtilir: "Miktar (kg)".
- Ham veride boş satır, birleştirilmiş hücre ve ara toplam bulunmaz.
- Ham veri, hesaplamalar ve rapor çıktıları ayrı sayfalarda tutulabilir.

✗ Analize uygun değil

	A	B	C
1	2024 Satışları (birleşik)		
2	Ürün	Miktar	
3	Elma	12 kg	
4	(boş satır)		
5	Ara toplam	12 kg	

✓ Düzenli

	A	B	C
1	Tarih	Ürün	Miktar (kg)
2	05.01.2024	Elma	12
3	05.01.2024	Armut	8
4	06.01.2024	Elma	15

toplamlar ayrı hücrede / rapor sayfasında

Şekil 3.2.48: Solda analize uygun olmayan düzen: birleştirilmiş başlık, boş satır, birimli metin ("12 kg") ve ara toplam. Sağda düzenli veri: tek başlık satırı, sayısal değerler, birim başlıkta; toplamlar ayrı hücrede ya da rapor sayfasında.

Örnek 3.2.23

Şekil 3.2.48'deki soldaki tabloda hangi Excel araçları bozulur? Sağdaki düzende aynı sorular nasıl yanıtlanır?

1. **Birleştirilmiş başlık.** Sıralama ve filtre birleştirilmiş hücre nedeniyle çalışmaz; Excel tablosuna dönüştürme de başlıkları tanımaz.
2. **Boş satır.** Excel bitişik veriyi bir bütün sayar. Boş satır listeyi ikiye böler; Ctrl+T ya da Özet Tablo yalnızca üst parçayı alır.
3. **"12 kg" metni.** Metin olduğu için TOPLA'ya girmez, aritmetikte #DEĞER! verir.
4. **Ara toplam satırı.** Sütun toplanınca ara toplam bir kez daha eklenir ve sonuç iki kat şişer.
5. **Düzenli tabloda.** Elma'nın toplamı `=ETOPLA(B2:B4;"Elma";C2:C4)` = 12 + 15 = 27 kg; bütün miktarlar `=TOPLA(C2:C4)` = 12 + 8 + 15 = 35 kg. Toplamlar verinin içinde değil, ayrı bir hücrede durur.

Bu kurallar başta zahmetli görünebilir. Ama düzenli bir veri sayfası, her yeni soruda baştan uğraşmayı önler: aynı liste bugün sıralanır, yarın filtrelenir, sonra bir Özet Tablo'ya kaynak olur. Düzenli verinin bir yararı da başka araçlara kolayca aktarılmasıdır; bir istatistik programı ya da bir programlama dili bu tabloyu olduğu gibi okuyabilir.

3.2.14 Özet

- Bir **çalışma kitabı** bir ya da daha çok **çalışma sayfasından** oluşur. Hücre adresi sütun harfi ve satır numarasıyla yazılır (C3); **aralık** iki köşe hücresiyle belirtilir (B4:D6 = 3 × 3 = 9 hücre). Bir sayfada 1.048.576 satır ve 16.384 sütun vardır.
- Excel metni sola, sayı ve tarihi sağa yaslar; sola yaslı bir sayı metin olarak saklanmıştır ve hesaba girmez. **Sayı biçimi** yalnızca görünümü değiştirir; hesapta formül çubuğundaki değer kullanılır.
- **Formül** eşittir işaretiyle başlar ve sabit, başvuru, işleç ve işlev içerebilir. **İşleç önceliği**: parantez, olumsuzlama, yüzde, üs, çarpma–bölme, toplama–çıkarma, birleştirme, karşılaştırma; aynı öncelikte soldan sağa. Excel'de `=-2^2` = 4'tür.
- **Göreli** başvuru (A1) kopyalanınca kayar, **mutlak** başvuru ($A$1) sabit kalır, **karma** başvuruda ($A1, A$1) yalnızca $ işaretli parça sabittir. F4 türler arasında döner. Tek bir oran hücresine başvuran formüllerde o başvuru mutlak olmalıdır; aksi hâlde Excel hata vermeden yanlış sonuç üretir.
- TOPLA, ORTALAMA, MAK, MİN ve SAY metin ve boş hücreleri yok sayar. **EĞER** bir sınamaya göre iki değerden birini döndürür; VE, YADA ve DEĞİL koşulları birleştirir; iç içe EĞER'de koşullar yüksek eşikten başlar. EĞERHATA hatayı gizler ama nedenini gidermez.
- **EĞERSAY** ve **ETOPLA** bir ölçüte göre sayar ve toplar; çok koşul için ÇOKEĞERSAY ve ÇOKETOPLA kullanılır. **DÜŞEYARA** ilk sütunda arar ve sağdaki bir sütunu döndürür; tam eşleşmede son bağımsız değişken 0'dır. Yaklaşık eşleşme sıralı barem tablolarında kullanılır. **ÇAPRAZARA** iki yöne de bakar; **FİLTRE** sonucu taşırır.
- Tarihler **seri değer**, saatler günün kesridir; tarih farkı çıkarma ile bulunur. Her **hata değeri** bir nedeni gösterir: #SAYI/0!, #AD?, #DEĞER!, #BAŞV!, #YOK, #TAŞMA!. Hatalar kendilerini kullanan formüllere yayılır.
- **Otomatik doldurma** seriyi ilk iki değerden çıkarır; **Hızlı Doldurma** örnekten öğrenir ama değer üretir. **Excel tablosu** kendiliğinden genişler ve yapılandırılmış başvuruya izin verir.
- **Sıralamada** satırlar bütün olarak taşınmalıdır; çok düzeyli sıralamada ikinci ölçüt yalnızca eşit gruplar içinde çalışır. **Filtre** satırları gizler, silmez; koşullar birlikte daraltır. TOPLA gizli satırları da toplar, **ALTTOPLAM** yalnız görünenleri hesaplar.
- **Koşullu biçimlendirme** biçimi değere bağlar; simge kümesi eşikleri en küçük ile en büyük değer arasındaki aralığa göre hesaplanır. **Özet Tablo** formülsüz gruplu özet verir ve kaynak değişince yenilenmelidir. **Veri doğrulama** geçersiz girişi baştan engeller.
- Grafik türü sorulan soruya göre seçilir: karşılaştırma için sütun, zaman için çizgi, parça–bütün için az dilimli pasta, ilişki için dağılım, dağılım biçimi için histogram. Sütun grafiğin ekseni 0'dan başlamalıdır.

- Yazdırmadan önce yazdırma alanı, ölçek ve yinelenen başlıklar ayarlanır. Sayfa koruması yanlışlıkla değişikliği önler, gizlilik sağlamaz; gizlilik için dosya şifrelenir. İyi bir veri düzeninde her satır bir kayıt, her sütun bir değişkendir; boş satır, birleştirilmiş hücre ve ara toplam bulunmaz.

Hesap	Kural	Örnek
Aralıktaki hücre sayısı	(sütun farkı + 1) × (satır farkı + 1)	C3:F10 → 4 × 8 = 32
İşleç önceliği	parantez, işaret, %, ^, * /, + −	`=2+3*(4-1)^2` → 29
Formül kopyalama	$ işaretsiz parçalar Δs, Δr kadar kayar	C3 → E7: `=A1+$B$2+B$1+$A2` → `=C5+$B$2+D$1+$A6`
Ortalama	sayıların toplamı ÷ sayı adedi	12, 7, "yok", 20, boş, 5 → 44 / 4 = 11
Yaklaşık eşleşme	aranandan küçük ya da eşit en büyük eşik	72 → eşik 70 → B
Tarih farkı	seri değerlerin farkı	01.02.2024 → 01.03.2024: 29 gün
Saat ücreti	süre (gün) × 24 × saat ücreti	1,875 × 24 × 150 = 6750 ₺
Simge eşiği	en küçük + yüzde × (en büyük − en küçük)	45 + 0,67 × 46 ≈ 75,8

Tablo 3.2.17: Notta kullanılan başlıca hesaplar.

3.2.15 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

çalışma kitabı *workbook* · çalışma sayfası *worksheet* · sayfa sekmesi *sheet tab* · etkin hücre *active cell* · ad kutusu *name box* · formül çubuğu *formula bar* · aralık *range* · veri tipi *data type* · mantıksal değer (DOĞRU/YANLIŞ) *logical value* · dondurma (Bölmeleri Dondur) *freeze panes* · kenarlık *border* · dolgu *fill* · metni kaydır *wrap text* · sayı biçimi *number format* · formül *formula* · sabit *constant* · işleç *operator* · işleç önceliği *operator precedence* · olumsuzlama *negation* · üs *exponent* · birleştirme (&) *concatenation* · hücre başvurusu *cell reference* · göreli başvuru *relative reference* · mutlak başvuru *absolute reference* · karma başvuru *mixed reference* · adlandırılmış aralık *named range* · doldurma tutamacı *fill handle* · Formülleri Göster *Show Formulas* · işlev *function* · bağımsız değişken *argument* · mantıksal sınama *logical test* · ölçüt *criteria* · tam eşleşme *exact match* · yaklaşık eşleşme *approximate match* · taşma *spill* · seri değer *date serial number* · hata değeri *error value* · otomatik doldurma *AutoFill* · seri *data series* · Hızlı Doldurma *Flash Fill* · tablo *table* · yapılandırılmış başvuru *structured reference* · toplam satırı *total row* · sıralama *sort* · çok düzeyli sıralama *multi-level sort* · filtreleme *filtering* · ALTTOPLAM *SUBTOTAL*

koşullu biçimlendirme *conditional formatting* · veri çubukları *data bars* · renk ölçekleri *color scales* · simge kümeleri *icon sets* · Özet Tablo *PivotTable* · veri doğrulama *data validation* · açılır liste *drop-down list* · girdi iletisi *input message* · hata uyarısı *error alert* · sütun / çubuk grafik *column / bar chart* · dağılım (XY) grafiği *scatter chart* · histogram *histogram* · yazdırma alanı *print area* · sayfa koruması *sheet protection*

3.2.16 Kendini Dene

1. C3 hücresindeki `=A1+$B$2+B$1+$A2` formülü kopyalanıp D5'e yapıştırılıyor. D5'teki formül nedir?

   Cevap

   C'den D'ye 1 sütun sağa, 3'ten 5'e 2 satır aşağı: Δs = 1, Δr = 2. `A1` → B3; `$B$2` değişmez; `B$1` → C$1 (yalnız sütun kayar); `$A2` → $A4 (yalnız satır kayar). Sonuç: `=B3+$B$2+C$1+$A4`.

2. Excel'de `=-3^2+10` ve `=10-3^2` formülleri ne verir? Neden farklıdır?

   Cevap

   Birincide baştaki eksi olumsuzlamadır ve üsten önce uygulanır: (−3)^2 = 9, 9 + 10 = 19. İkincide eksi çıkarmadır ve üsten sonra gelir: 3^2 = 9, 10 − 9 = 1. Matematikteki $-3^2 + 10 = 1$ sonucu için `=-(3^2)+10` yazılmalıdır.

3. B2:B6'da 40, "girmedi", 70, boş, 90 var. `=ORTALAMA(B2:B6)`, `=SAY(B2:B6)` ve `=BAĞ_DEĞ_DOLU_SAY(B2:B6)` ne verir? "girmedi" yerine 0 yazılsaydı ortalama ne olurdu?

   Cevap

   Yalnızca 40, 70 ve 90 sayıdır: ORTALAMA = 200 / 3 ≈ 66,7; SAY = 3; BAĞ_DEĞ_DOLU_SAY metni de sayar: 4. 0 yazılırsa 0 bir sayı olarak hesaba girer: 200 / 4 = 50.

4. 3.2.7'deki altı öğrencilik listede (B2:B7 bölüm, C2:C7 puan) `=EĞERSAY(B2:B7;"<>Elektrik")` ve `=ETOPLA(C2:C7;">=70")` ne verir?

   Cevap

   Elektrik dışındakiler Ece, Deniz ve Burak'tır: 3. Puanı 70 ve üstü olanlar 72, 91, 85 ve 78'dir; toplam aralığı verilmediği için puanların kendisi toplanır: 72 + 91 + 85 + 78 = 326.

5. 0 → F, 50 → D, 60 → C, 70 → B, 85 → A baremiyle `=DÜŞEYARA(B2;$D$2:$E$6;2;DOĞRU)` formülü 59

ve 84 puanları için ne döndürür? Son bağımsız değişken 0 olsaydı ne olurdu?

Cevap

59: küçük ya da eşit en büyük eşik 50 → D. 84: en büyük uygun eşik 70 (85 > 84) → B. Tam eşleşmede ikisi de tabloda birebir bulunmadığı için #YOK döner.

6. Bir proje haftada beş gün, günde 08:30–17:30 arası çalışılarak yürütülüyor. Günlük süre `=B2-A2` ile bulunup beş gün toplanıyor. Toplam hücresi `ss:dd` biçimindeyse ne görünür? Doğru gösterim için ne yapılır?

Cevap

Günlük süre 17:30 − 08:30 = 9 saat; haftalık 45 saat = 1,875 gün. `ss:dd` tam günü atar ve 45 − 24 = 21 saati gösterir: 21:00. Biçim kodu `[s]:dd` yapılınca 45:00 görünür.

7. 3.2.10.4'teki listede (Selin Bilgisayar 78, Can Elektrik 85, Çağla Bilgisayar 91, Ali Elektrik 64, İrem Makine 70, Burak Bilgisayar 55) Bölüm = Elektrik filtresi uygulanıyor. TOPLA, ALTTOPLAM(9) ve ALTTOPLAM(1) ne verir?

Cevap

TOPLA gizli satırları da toplar: 443. Görünen satırlar Can 85 ve Ali 64'tür: ALTTOPLAM(9) = 149, ALTTOPLAM(1) = 149 / 2 = 74,5.

8. Puanlar 35, 50, 65, 80, 95, 100. Varsayılan üç simgeli küme (%67 ve %33) uygulanıyor. Eşikler nedir; 65 ve 80 hangi simgeyi alır?

Cevap

Aralık 100 − 35 = 65. Üst eşik 35 + 0,67 × 65 = 78,55; alt eşik 35 + 0,33 × 65 = 56,45. 80 ≥ 78,55 → ▲; 65 iki eşiğin arasında → ●. 35 ve 50 ▼, 95 ve 100 ▲ alır.

9. Bir Özet Tablo'da bölüm ortalamaları Bilgisayar 77,5 (2 kişi), Elektrik 71,7 (3 kişi), Makine 78 (1 kişi) çıkmıştır. Genel Toplam satırındaki ortalama neden 75,7 değil de 74,7'dir?

Cevap

Genel ortalama altı puanın toplamının altıya bölümüdür: 448 / 6 ≈ 74,7. Üç ortalamanın basit ortalaması (77,5 + 71,7 + 78) / 3 ≈ 75,7 ise her bölüme kişi sayısından bağımsız olarak aynı ağırlığı verir. Tek kişilik Makine grubu fazla, üç kişilik Elektrik grubu az sayılmış olur.

10. Bir öğrenci ad–puan listesinde yalnızca Puan sütununu seçip büyükten küçüğe sıralıyor ve uyarıda

"Geçerli seçimle devam et" diyor. Ne olur, nasıl düzeltilir?

Cevap

Yalnızca puanlar yer değiştirir, adlar yerinde kalır; her ad başka birinin puanıyla eşleşir. Excel hata vermez. Hemen Ctrl+Z ile geri alınır; sıralama uyarısında "Seçimi genişlet" seçilerek satırların bir bütün olarak taşınması sağlanır.

3.2.17 İleri okuma

- Microsoft Destek, Excel eğitim sayfaları (*support.microsoft.com*): formüller, işlevler, tablolar, Özet Tablo ve grafikler için Türkçe dahil birçok dilde kısa, örnekli ve güncel anlatımlar sunar.
- M. Alexander ve D. Kusleika, *Microsoft Excel 365 Bible* (Wiley, 2022): bu notta kısaca değinilen işlevleri, dinamik dizi formüllerini, Özet Tabloları ve grafikleri başvuru kitabı ayrıntısında işler.
- K. W. Broman ve K. H. Woo, "Data Organization in Spreadsheets", *The American Statistician* 72(1), 2018: elektronik tabloda veriyi tutarlı, dikdörtgen ve analize hazır biçimde düzenlemenin kurallarını kısa ve uygulamalı olarak anlatır.
- C. Nussbaumer Knaflic, *Storytelling with Data* (Wiley, 2015): grafik türünü seçmeyi, gereksiz öğeleri ayıklamayı ve okurun dikkatini doğru yere yöneltmeyi örneklerle gösterir.

3.2.18 Kaynaklar

1. Microsoft Support, *Basic tasks in Excel*.
2. Microsoft Support, *Overview of formulas in Excel*.
3. Microsoft Support, *Sort data in a range or table*.
4. M. Alexander, D. Kusleika, *Microsoft Excel 365 Bible*, Wiley, 2022.
5. K. W. Broman, K. H. Woo, "Data Organization in Spreadsheets", *The American Statistician*, 72(1), 2–10, 2018.
6. C. Nussbaumer Knaflic, *Storytelling with Data: A Data Visualization Guide for Business Professionals*, Wiley, 2015.

KISIM I · BÖLÜM 3.3

Microsoft PowerPoint

Bir proje önerisi, bir laboratuvar sonucu ya da bitirme çalışması çoğu zaman birkaç dakikalık bir konuşmayla anlatılır. Bu konuşmanın arkasındaki ekranda genellikle bir PowerPoint sunusu vardır. Bu not, sununun planlanmasından paylaşılmasına kadar bütün aşamaları anlatır: çalışma alanı ve görünümler, süreye göre plan ve anahat, dosya biçimleri, slayt düzeni ve bölümler, tema ve Asıl Slayt, görsel hiyerarşi, resim, şekil, tablo ve grafik, hizalama, geçiş ve animasyon, konuşmacı notları ve Sunucu Görünümü, zamanlama provası, erişilebilirlik, denetim ve yazdırma. Her konu şekillerle ve adım adım işlem tablolarıyla desteklenir. Animasyon zamanları, resim boyutu, sıkıştırma, hizalama koordinatları ve sayfa sayısı gibi sayılar küçük hesaplarla açıklanır.

ÖĞRENME ÇIKTILARI

Planlama
Sununun amacını, hedef kitlesini ve ana mesajını belirlemek; konuşma süresine göre slayt sayısını hesaplamak; içeriği Anahat Görünümü'nde kurmak.

Slayt yönetimi
Düzen, tema, Asıl Slayt, bölüm, gizli slayt ve yakınlaştırma araçlarını amacına uygun kullanmak; elle biçim ile asıl slayt kuralı arasındaki öncelik ilişkisini açıklamak.

Görsel anlatım
Metin, resim, şekil, tablo ve grafikleri okunaklı yerleştirmek; resim boyutunu ve sıkıştırma kazancını hesaplamak; nesneleri hizalayıp eşit aralıkla dağıtmak.

Sunum yapma
Geçiş ile animasyonu ayırt etmek, animasyonların başlangıç ve bitiş zamanlarını hesaplamak; konuşmacı notları, Sunucu Görünümü, zamanlama provası ve gösteri kısayollarını kullanmak; sunuyu paylaşmadan önce erişilebilirlik ve teknik denetimleri yapmak.

3.3.1 PowerPoint ve çalışma alanı

Microsoft PowerPoint, Microsoft 365 paketindeki sunu programıdır. Word metin belgesi, Excel hesap tablosu üretir; PowerPoint ise sırayla ekrana gelen sayfalardan oluşan bir dosya üretir. Bu sayfaların her birine slayt denir. Not, Microsoft 365 PowerPoint'in Windows'taki Türkçe arayüzüne göre yazıldı. PowerPoint'in web, Mac ve mobil sürümlerinde menülerin yeri biraz farklı olabilir, ama kavramlar aynıdır.

3.3.1.1 Sunu programı ne işe yarar?

Türkçede iki yakın sözcük ayrı anlamlarda kullanılır. Dosyanın kendisine *sunu*, onu izleyici önünde anlatma eylemine *sunum* denir. Örneğin "sunu dosyası 20 slayttan oluşuyor" ama "sunum 10 dakika sürdü" denir. Not boyunca bu ayrım korunur.

Tanım 3.3.1

Slayt (*slide*), ekranı tek seferde dolduran sayfadır; metin, resim, şekil, tablo, grafik, ses ve video taşıyabilir. **Sunu** (*presentation file*), belirli bir sırayla gösterilecek slaytlardan oluşan dosyadır. **Sunum** (*presentation*) ise bir konuşmacının bu slaytlar eşliğinde yaptığı anlatımdır.

PowerPoint en çok canlı bir konuşmayı desteklemek için kullanılır. Ama kullanım alanları bununla sınırlı değildir (Tablo 3.3.1).

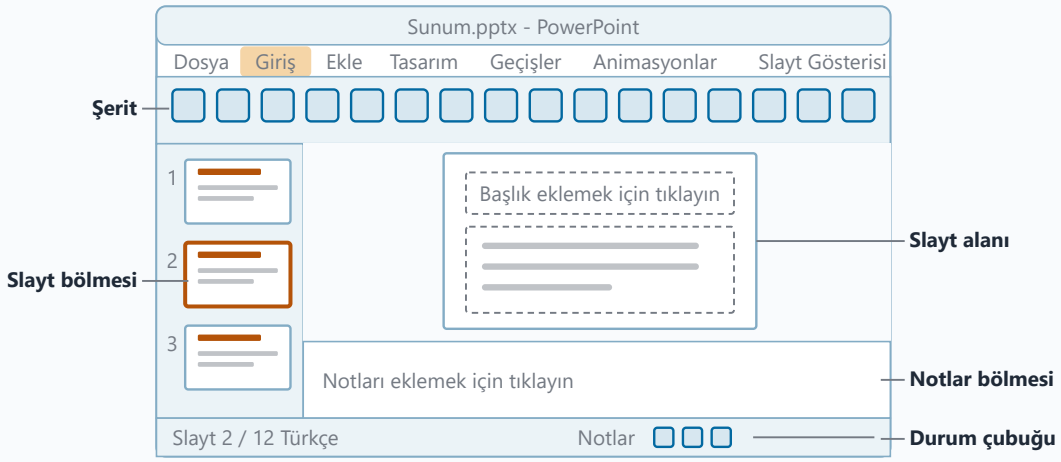
Kullanım	Sununun görevi	Örnek
Ders ve konuşma	Konuşmacının anlatımını görsel olarak destekler	Bir dersin anlatımı, seminer konuşması
Proje sunumu	Amaç, yöntem, bulgu ve sonucu belirli bir sırayla yapılandırır	Bitirme projesi savunması
Raporlama	Veri ve bulguları grafiklerle özetler	Aylık enerji tüketimi raporu
Eğitim materyali	Adımları, örnekleri ve alıştırmaları düzenler	Laboratuvar cihazının kullanım eğitimi
Bilgi ekranı (kiosk)	Kimse dokunmadan kendi kendine ilerler ve başa döner	Fakülte girişindeki duyuru ekranı
Kayıt ve video	Sesli anlatımla birlikte izlenebilir bir çıktı üretir	Uzaktan eğitim için ders videosu

Tablo 3.3.1: PowerPoint'in başlıca kullanım alanları.

Bu kullanımların ortak noktası şudur: slayt, anlatılan şeyin yerini almaz; onu destekler. İyi bir sununun slaytlarında ana noktalar, sayılar ve görseller bulunur. Ayrıntı ise konuşmacının sözünde, konuşmacı notlarında ya da ayrı bir belgededir.

3.3.1.2 Çalışma alanı

PowerPoint açılıp yeni bir sunu oluşturulduğunda ekranı beş ana bölge paylaşır (Şekil 3.3.1). En üstte şerit, solda slayt bölmesi, ortada slayt alanı, onun altında notlar bölmesi ve en altta durum çubuğu bulunur.



Şekil 3.3.1: PowerPoint penceresi. Üstte sekmeli şerit, solda slaytların küçük resimlerini gösteren slayt bölmesi, ortada düzenlenen slayt, altında notlar bölmesi ve en altta durum çubuğu ("Slayt 2 / 12").

Tanım 3.3.2

Şerit (*ribbon*), komutları Dosya, Giriş, Ekle, Tasarım, Geçişler, Animasyonlar, Slayt Gösterisi, Gözden Geçir ve Görünüm gibi sekmelere bölerek gösteren araç alanıdır. Her sekmede komutlar işlevlerine göre gruplanır. **Bağlamsal sekme** ise yalnızca belirli bir nesne seçiliyken görünen sekmedir; örneğin bir resim seçilince Resim Biçimi, bir tablo seçilince Tablo Tasarımı sekmesi çıkar.

Bölgelerin görevleri Tablo 3.3.2'de özetlenir. Bunlardan ikisi özellikle önemlidir. Slayt bölmesi sununun bütününü yönetmek için, notlar bölmesi ise konuşmacının kendisi için kullanılır.

Bölge	Ne gösterir?	Ne için kullanılır?
Şerit	Sekmeler ve komut grupları	Ekleme, biçimlendirme, geçiş, animasyon, gösteri komutları
Slayt bölmesi	Slaytların numaralı küçük resimleri	Slayt seçmek, sıralamak, çoğaltmak, silmek, gizlemek
Slayt alanı	Seçili slaydın büyük hâli	İçeriği yazmak ve düzenlemek
Notlar bölmesi	Seçili slayda ait konuşmacı notu	Slayda sığmayan açıklamayı, süreyi, hatırlatıcıyı yazmak

Bölge	Ne gösterir?	Ne için kullanılır?
Durum çubuğu	Slayt numarası, dil, görünüm düğmeleri, yakınlaştırma	Konumu görmek, görünümler arasında hızlı geçiş

Tablo 3.3.2: PowerPoint çalışma alanının bölgeleri.

Slayt alanındaki kesikli kutularda “Başlık eklemek için tıklayın” gibi yazılar görünür. Bu kutulara yer tutucu denir ve gösteride görünmezler; yalnızca nereye ne yazılacağını gösterirler. Her kutunun türünü ve yerini slaydın düzeni belirler.

Sık yapılan hata

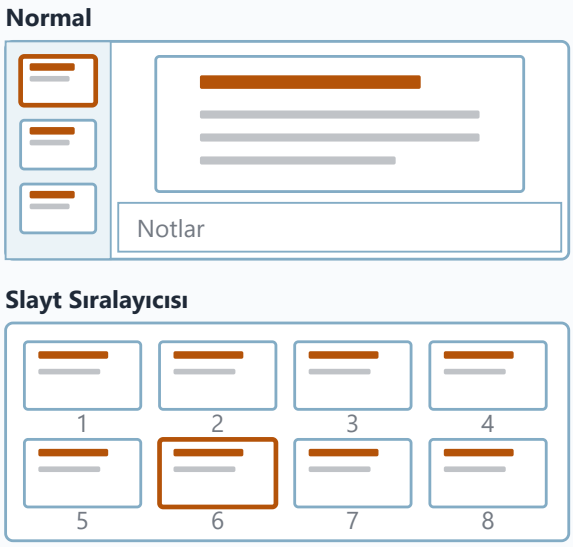
Bir resmi düzenlemek için gereken komutları Giriş sekmesinde aramak. Kırpma, sıkıştırma, arka plan kaldırma gibi komutlar Resim Biçimi bağlamsal sekmesindedir ve bu sekme ancak resim seçiliyken görünür. Aranan komut şeritte yoksa önce ilgili nesnenin seçili olup olmadığına bakılır.

3.3.1.3 Sunu görünümleri

Aynı sunu farklı işler için farklı biçimlerde gösterilebilir. PowerPoint bunlara görünüm der. Görünümler Görünüm sekmesinden ya da durum çubuğundaki düğmelerden seçilir. Görünüm değiştirmek dosyanın içeriğini değiştirmez; yalnızca ekranda nasıl gösterildiğini değiştirir.

Görünüm	Ekranda ne var?	Ne zaman kullanılır?
Normal	Slayt bölmesi, slayt ve notlar	Slayt içeriğini düzenlerken; varsayılan görünüm
Anahat Görünümü	Başlıklar ve maddeler metin olarak	İçeriği tasarımdan önce, yazı olarak kurarken
Slayt Sıralayıcısı	Bütün slaytlar ızgara hâlinde	Akışı topluca görmek, sıralamak, bölümlemek
Not Sayfası	Üstte slayt, altta not, sayfa düzeninde	Konuşmacı notlarını yazdırılacak biçimde hazırlarken
Okuma Görünümü	Sunu pencere içinde oynar	Gösteriyi tam ekrana geçmeden denemek
Asıl Slayt	Asıl slayt ve ona bağlı düzenler	Bütün sunuyu etkileyen ortak tasarımı yönetmek
Slayt Gösterisi	Slaytlar tam ekran, birer birer	Sunuyu izleyiciye sunarken

Tablo 3.3.3: PowerPoint görünümleri ve kullanım amaçları.



Şekil 3.3.2: Aynı sununun iki görünümü. Normal görünümde solda küçük resimler, sağda düzenlenen slayt ve altta notlar vardır. Slayt Sıralayıcısı'nda sekiz slaydın hepsi ızgarada görünür.

Şekil 3.3.2, iki görünüm arasındaki farkı gösterir. Normal görünüm tek bir slayt üzerinde çalışmaya uygundur. Slayt Sıralayıcısı ise sununun bütününü gösterir: hangi bölümün kaç slayt tuttuğu, hangi slaytların birbirine benzediği ve akışın nerede koptuğu bu görünümde kolayca fark edilir.

Örnek 3.3.1

Bir öğrenci 35 slaytlık bitirme sunusu üzerinde çalışıyor. Aşağıdaki işlerin her biri için en uygun görünüm hangisidir?

1. Danışman, “Yöntem slaytları bulgulardan önce gelsin” dedi. Birkaç slaydın yeri değişecek. Bütün slaytları aynı anda görmek ve sürükleyerek taşımak gerekir: **Slayt Sıralayıcısı**.
2. Henüz hiçbir slayt yok; yalnızca başlıklar ve maddeler belli. İçerik hızlıca, yazı olarak girilecek: **Anahat Görünümü**.
3. Bütün slaytların sağ alt köşesine üniversite logosu konacak: tek tek slayt yerine ortak tasarım değişir, **Asıl Slayt** görünümü.
4. Konuşmacı notları kâğıda basılıp provada elde tutulacak: notların sayfadaki yerleşimini görmek için **Not Sayfası**.

Dört işin hiçbiri Normal görünümde imkânsız değildir; ama her biri uygun görünümde birkaç kat daha hızlı yapılır.

3.3.2 Sunum planı

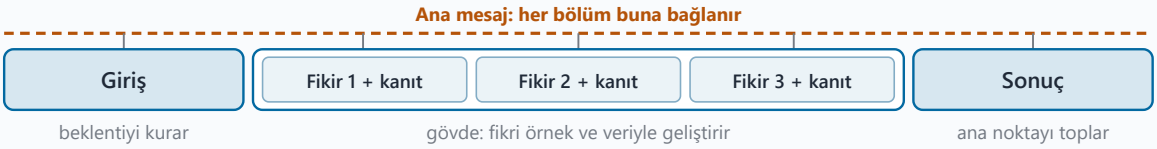
Sunu hazırlamaya PowerPoint'i açarak başlamak yaygın bir alışkanlıktır. Oysa iyi bir sunumun en önemli kısmı, ilk slayt çizilmeden önce yapılan plandır. Plan üç soruya cevap verir: Bu sunum ne için yapılıyor? Kime yapılıyor? İzleyici salondan çıkarken aklında tek bir cümle kalacaksa bu hangi cümledir?

3.3.2.1 Amaç, hedef kitle ve ana mesaj

Planın öğeleri sırasıyla şunlardır: amaç, hedef kitle, ana mesaj, kanıt ve örnek, akış ve kapanış. Amaç sununun neden yapıldığını söyler: bilgilendirmek, ikna etmek, bir karar aldırmak ya da öğretmek. Hedef kitle aynı konunun nasıl anlatılacağını belirler. Örneğin bir güneş paneli projesi, jüri üyesi öğretim elemanlarına teknik ayrıntıyla, lise öğrencilerine ise günlük hayattan örneklerle anlatılır.

Tanım 3.3.3

Ana mesaj, sununun bütün bölümlerinin desteklediği tek cümlelik sonuçtur. "Sensörlü aydınlatma laboratuvarın elektrik tüketimini yaklaşık üçte bir azaltır" bir ana mesajdır; "Aydınlatma sistemleri" ise yalnızca bir konu başlığıdır.



Şekil 3.3.3: Sunum akışı. Giriş beklentiyi kurar, gövdede her fikir bir kanıtla geliştirilir, sonuç ana noktayı toplar. Ana mesaj bütün bölümleri birbirine bağlar.

Akış genellikle üç parçadan oluşur (Şekil 3.3.3). Giriş, konunun neden önemli olduğunu söyleyerek izleyicide bir beklenti kurar. Gövde, ana mesajı destekleyen birkaç fikri sırayla geliştirir; her fikrin yanında bir kanıt, bir sayı ya da bir örnek bulunur. Sonuç, ana mesajı yeniden ve açıkça söyler. Bu yapıdan üç pratik kural çıkar:

- Her slaydın tek ve açık bir amacı olur. Bir slaytta iki ayrı fikir varsa genellikle iki slayt gerekir.
- Slaytlar konuşmanın metin dökümü değildir. Konuşmacının söyleyeceği her cümle slayda yazılırsa izleyici dinlemeyi bırakıp okumaya başlar.
- Kapanış, yeni bilgi vermez; ana mesajı toplar. "Teşekkürler" slaydı tek başına bir kapanış değildir.

Sık yapılan hata

Sununun başlığını ana mesaj sanmak. “Laboratuvarda Enerji Tüketimi” başlığı izleyiciye ne düşünmesi gerektiğini söylemez. Ana mesaj bir yargı içerir ve sayıyla desteklenebilir: “Laboratuvarın gece tüketiminin yarısı açık unutulan cihazlardan geliyor.”

3.3.2.2 Süreye göre slayt planı

Sunumlar çoğu zaman belirli bir süreye sığmak zorundadır. Bu yüzden slayt sayısı içerikten önce süreden hesaplanır. Deneyimle bulunmuş bir ölçü vardır: bir içerik slaydı ortalama 1–1,5 dakikalık konuşma taşır. Başlık slaytı, kaynak slaytı gibi üzerinde konuşulmayan slaytlar bu hesaba katılmaz.

Örnek 3.3.2

Bir proje sunumu için 15 dakikalık oturum verildi. Oturumun bir kısmı soru-cevaba ayrılacak. Slayt planı çıkarılacak.

1. Soru-cevap süresi plana baştan yazılır: 15 dakikalık oturum = 10 dakika sunum + 5 dakika soru.
2. 10 dakika bölümlere paylaştırılır: giriş ve problem 2 dk, yöntem 3 dk, bulgular 4 dk, sonuç 1 dk. Toplam 2 + 3 + 4 + 1 = 10 dakikadır.
3. Her bölüme süresine uygun sayıda slayt verilir (Tablo 3.3.4). İçerik slaytlarının toplamı 2 + 3 + 3 + 1 = 9'dur.
4. Ölçüyle karşılaştırılır: 10 ÷ 9 ≈ 1,1 dakika/slayt. Bu değer 1–1,5 aralığındadır; plan gerçekçidir.
5. Başlık ve kaynak slaytı eklenir. Dosyada 11 slayt olur, ama süre hesabına 9'u girer.

Bölüm	Süre	Slayt	İçerik
Giriş ve problem	2 dk	2	Konu ve neden önemli olduğu
Yöntem	3 dk	3	Her adım için bir slayt
Bulgular	4 dk	3	Her slaytta tek bulgu ve bir grafik
Sonuç	1 dk	1	Ana mesajın tekrarı
Toplam	10 dk	9	+ başlık ve kaynak slaytı

Tablo 3.3.4: 10 dakikalık sunum için süreye göre slayt planı.

Bulgular bölümünde dört dakikaya üç slayt düşer; bu bölümdeki slaytlarda grafik yorumlandığı için slayt başına süre biraz daha uzundur. Ölçü katı bir kural değil, bir uyarı eşiğidir. Asıl yararı, gerçekçi olmayan planları hemen yakalamasıdır. Örneğin 10 dakikaya 30 slayt hazırlanmışsa her slayda 600 ÷ 30 = 20 saniye düşer. Bu sürede bir grafik ancak gösterilir, açıklanamaz.

Bazı ayrıntılar soru gelirse gerekebilir: uzun bir sonuç tablosu ya da bir hesaplamanın ara adımları gibi. Bunlar sununun sonuna gizli slayt olarak eklenir. Gizli slaytlar normal gösteride atlanır, gerekince açılır.

Sık yapılan hata

Süreyi yalnızca slayt sayısına bakarak tahmin etmek ve hiç prova yapmamak. Ölçü yalnızca başlangıç noktasıdır. Gerçek süre ancak sesli provayla ölçülür; PowerPoint'in Zamanlamaları Prova Et aracı bu ölçümü slayt slayt kaydeder.

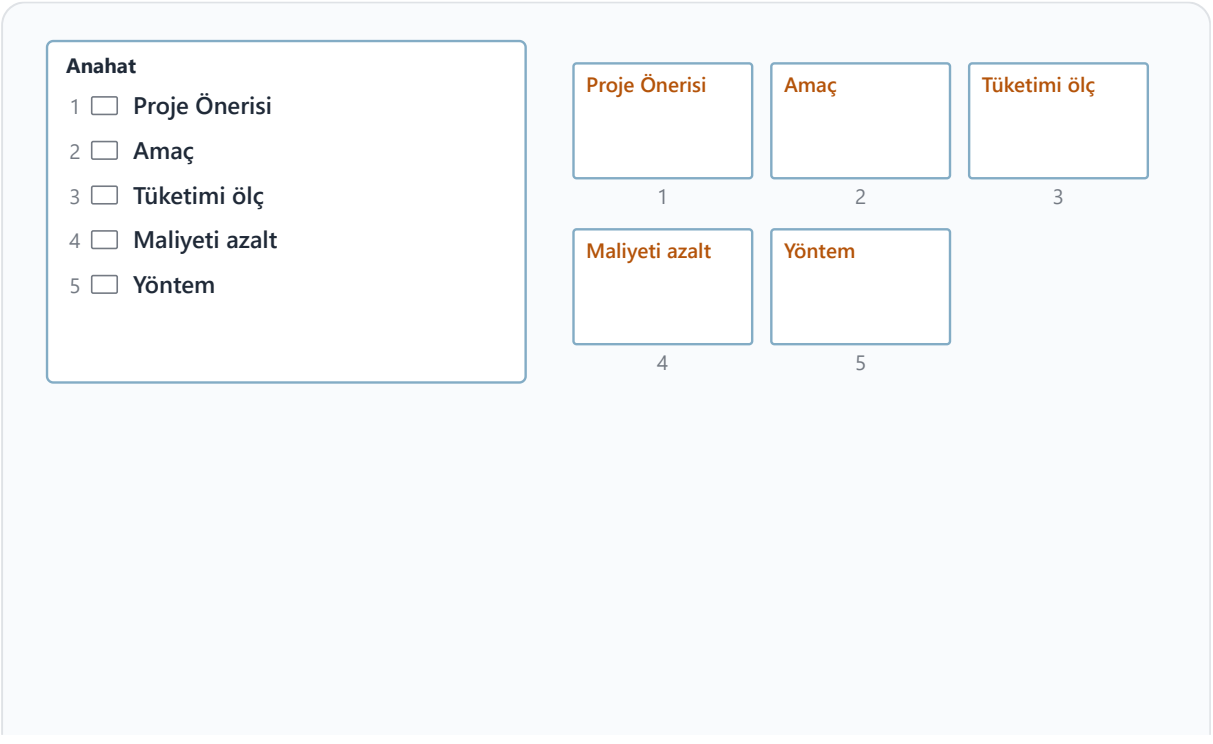
3.3.2.3 Anahat Görünümü

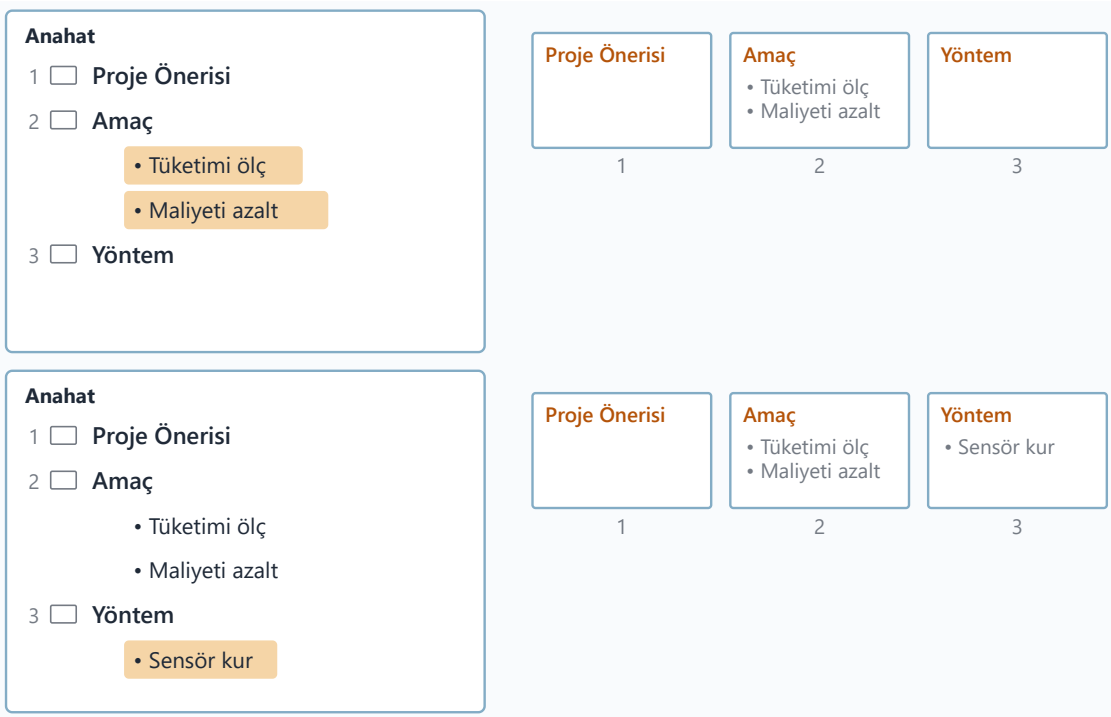
Planı slayda dökmenin en hızlı yolu Görünüm → Anahat Görünümü'dür. Bu görünümde sunu, girintili bir metin listesi gibi görünür. Her satırın bir düzeyi vardır. En üst düzeydeki satır bir slaydın başlığıdır; bir alt düzeydeki satır o slaydın maddesidir.

Tanım 3.3.4

Anahat (*outline*), sununun başlık ve maddelerinin düzeyleriyle birlikte metin olarak gösterilmesidir. Anahatta Enter aynı düzeyde yeni bir satır açar; Tab satırın düzeyini bir düşürür (başlığı maddeye çevirir), Shift + Tab bir yükseltir (maddeyi başlığa çevirir).

Anahattın asıl gücü şudur: düzey değiştikçe slayt sayısı da kendiliğinden değişir. Şekil 3.3.4, beş satırlık bir anahattın üç adımda nasıl biçimlendiğini gösterir. İşlemler Tablo 3.3.5'te sırayla izlenir.





Şekil 3.3.4: Anahat Görünümü'nde düzeyler. Üstte beş satırın her biri bir slayt başlığıdır: beş slayt. Ortada 3. ve 4. satırda Tab'a basıldı; bu satırlar "Amaç" slaydının maddeleri oldu ve slayt sayısı üçe indi. Altta "Yöntem"den sonra Enter + Tab ile bir madde eklendi.

Adım	İşlem	Anahat	Slayt sayısı
1	Beş satır yazılır, her birinden sonra Enter	Proje Önerisi / Amaç / Tüketimi ölç / Maliyeti azalt / Yöntem	5
2	"Tüketimi ölç" satırında Tab	"Tüketimi ölç", Amaç'ın maddesi olur	4
3	"Maliyeti azalt" satırında Tab	"Maliyeti azalt" da Amaç'ın maddesi olur	3
4	"Yöntem" satırının sonunda Enter, ardından Tab	Yeni satır Yöntem'in maddesi olur; "Sensör kur" yazılır	3

Tablo 3.3.5: Anahat Görünümü'nde düzey işlemleri ve slayt sayısı.

Dördüncü adım dikkat ister. Başlık satırının sonunda Enter'a basılınca yeni satır başlıkla aynı düzeyde açılır; yani o an yeni ve boş bir slayt oluşur. Tab'a basılınca bu boş slayt bir önceki slaydın maddesine dönüşür ve slayt sayısı yeniden üçe iner. Anahatta slayt sayısını belirleyen tek şey, en üst düzeydeki satırların sayısıdır.

Anahat önce içerik, sonra tasarım ilkesini uygular. Başlık ve maddeler yazılırken yazı tipi, renk ya da resim düşünülmez. Akış oturduktan sonra slaytlara düzen ve tema uygulanır. Böylece tasarıma harcanan emek, sonradan silinecek slaytlara gitmez.

Sık yapılan hata

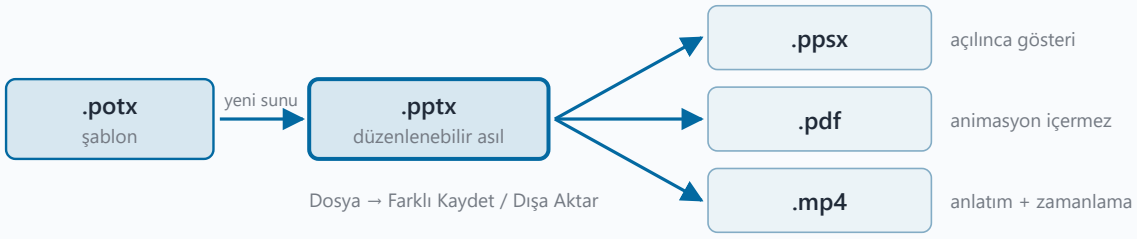
Anahatta madde yazmak isterken Tab'a basmayı unutmak. Enter ile açılan her yeni satır başlık düzeyindedir; madde sanılarak yazılan beş satır beş ayrı slayt oluşturur. Slayt bölmesinde beklenmedik sayıda boş görünümlü slayt varsa ilk bakılacak yer anahattaki düzeylerdir.

3.3.3 Dosya ve slayt yönetimi

Plan hazır olunca sıra dosyaya ve slaytlara gelir. Bu kesim sununun hangi biçimde kaydedileceğini, slaytların hangi kalıplarla oluşturulacağını ve uzun bir sunuda slaytların nasıl düzenli tutulacağını anlatır.

3.3.3.1 Yeni sunu ve dosya biçimleri

Yeni bir sunu Dosya → Yeni ile boş olarak ya da bir şablondan başlatılır. Sunu kaydedilirken seçilen dosya biçimi, dosyanın başka bir bilgisayarda nasıl açılacağını belirler. Aynı içerik farklı biçimlerde çok farklı davranır (Şekil 3.3.5).



Şekil 3.3.5: Sunu dosya biçimleri arasındaki ilişki. Bir .potx şablonundan yeni bir .pptx sunu oluşur. Düzenlenebilir asıl dosya .pptx'tir; ondan Farklı Kaydet ya da Dışa Aktar ile .ppsx gösteri dosyası, .pdf ve .mp4 video üretilir.

Uzantı	Ne tür dosya?	Çift tıklanınca	Animasyon
.pptx	Standart, düzenlenebilir sunu	PowerPoint Normal görünümde açılır	Var
.ppsx	Gösteri dosyası	Doğrudan slayt gösterisi başlar	Var
.potx	Şablon	Şablondan yeni, adsız bir sunu açılır	Var
.pdf	Düzeni sabit belge	PDF okuyucuda sayfalar görünür	Yok; slaydın son hâli
.mp4	Video	Video oynatıcıda oynar	Zamanlamaya göre oynar

Tablo 3.3.6: PowerPoint'in başlıca dosya biçimleri.

Tanım 3.3.5

Şablon (*template*), yeni sunuların başlangıç noktası olarak kullanılan, tema, düzen ve örnek içerik taşıyan dosyadır. Şablon açıldığında şablonun kendisi değil, ondan türetilmiş adsız bir kopya düzenlenir; böylece şablon bozulmaz.

Kaydet ile Farklı Kaydet arasındaki fark da önemlidir. Kaydet, açık olan dosyanın üzerine yazar. Farklı Kaydet ise yeni bir dosya oluşturur; özgün dosya değişmeden kalır ve PowerPoint artık yeni dosya üzerinde çalışır. Bir sunuyu .ppsx ya da .pdf olarak kaydetmek, asıl .pptx dosyasını silmez ya da dönüştürmez.

Örnek 3.3.3

Bir öğrenci bitirme sunusunu (sunu.pptx) hazırladı. Sunu üç yere gidecek. Her biri için hangi biçim uygundur?

1. Danışman sunuyu okuyup slaytlara düzeltme yapacak: düzenlenebilir olması gerekir, **.pptx** gönderilir.
2. Jüri üyelerine basılı kopya verilecek: sayfa düzeni her yazıcıda aynı kalmalıdır, **.pdf**. PDF'te animasyon yoktur; her slayt son hâliyle basılır.
3. Fakülte girişindeki ekranda tanıtım olarak dönecek: açılınca doğrudan gösteri başlamalı ve kimse düzenleme görünümünü görmemelidir, **.ppsx**. Bilgisayarda PowerPoint yoksa ya da yalnızca video oynatılabiliyorsa **.mp4** seçilir.

Üç çıktı da Farklı Kaydet ya da Dışa Aktar ile aynı .pptx dosyasından üretilir. Sonradan bir düzeltme gerekirse yalnızca .pptx düzeltilir ve öteki biçimler yeniden üretilir.

Sık yapılan hata

Sunuyu yalnızca PDF olarak saklamak. PDF'ten slayt düzenlemek zahmetlidir, animasyonlar ve konuşmacı notları da kaybolur. Paylaşım biçimi ne olursa olsun, düzenlenebilir .pptx dosyası her zaman ayrıca saklanır.

3.3.3.2 Slayt düzeni ve yer tutucu

Bir slayt boş bir sayfadan da başlatılabilir, ama PowerPoint bunun yerine hazır kalıplar sunar. Bu kalıplara düzen denir. Düzen, slaytta hangi kutuların bulunacağını ve bu kutuların nerede duracağını belirler.

Tanım 3.3.6

Yer tutucu (*placeholder*), slayt üzerinde belirli bir içerik türü için ayrılmış kesikli çerçeveli kutudur: başlık, metin, içerik (metin, tablo, grafik ya da resim), resim gibi. **Slayt düzeni** (*slide layout*), yer tutucuların türünü, sayısını ve konumunu tanımlayan kalıptır.



Şekil 3.3.6: Altı yaygın slayt düzeni: Başlık Slaytı, Başlık ve İçerik, İki İçerik, Karşılaştırma, Yalnızca Başlık ve Boş. Kesikli kutular yer tutuculardır.

Şekil 3.3.6, en sık kullanılan altı düzeni gösterir. Başlık Slaytı sununun açılışı içindir. Başlık ve İçerik, bir başlık ve madde listesi ya da tek bir grafik taşıyan sıradan slayttır. İki İçerik iki şeyi yan yana koyar; Karşılaştırma bunlara birer başlık da ekler. Yalnızca Başlık, altını serbestçe doldurmak için boş alan bırakır. Boş düzende hiç yer tutucu yoktur.

Bir slaydın düzeni Giriş → Düzen ile sonradan da değiştirilebilir. Bu durumda slayttaki metin kaybolmaz; yeni düzenin uygun yer tutucularına taşınır. Örneğin Başlık ve İçerik düzenindeki bir slayt İki İçerik'e çevrilirse maddeler sol kutuya geçer ve sağ kutu boş kalır.

Yer tutucu kullanmanın üç yararı vardır. Birincisi, başlıklar her slaytta aynı yerde ve aynı biçimde durur; izleyicinin gözü her slaytta başlığı aramaz. İkincisi, yer tutucudaki metin Anahat Görünümü'nde görünür; elle çizilmiş metin kutusundaki metin görünmez. Üçüncüsü, ekran okuyucular slayt başlığını yer tutucudan okur. Başlığı serbest metin kutusuna yazılmış bir slayt, ekran okuyucuya “başlıksız” görünür.

Sık yapılan hata

Her slaydı Boş düzenle açıp başlığı ve maddeleri tek tek metin kutusuyla eklemek. Slaytlar ilk bakışta düzgün görünse de başlıkların konumu slayttan slayda birkaç piksel kayar. Anahatta bu slaytlar başlıksız görünür ve tema değişince metin kutuları yeni tasarıma uymaz.

3.3.3.3 Slayt ekleme, çoğaltma, taşıma ve gizleme

Slaytlarla ilgili temel işlemler slayt bölmesinde, küçük resimlere sağ tıklanarak ya da Giriş sekmesinden yapılır (Tablo 3.3.7).

İşlem	Nasıl yapılır?	Sonuç
Yeni slayt	Giriş → Yeni Slayt ya da Ctrl + M	Seçili slayttan sonra, seçilen düzenle eklenir
Çoğalt	Sağ tık → Slaydı Çoğalt ya da Ctrl + D	Seçili slaydın kopyası hemen arkasına eklenir
Sil	Küçük resim seçilip Delete	Slayt sunudan kaldırılır, numaralar yeniden verilir
Taşı	Küçük resim slayt bölmesinde ya da Slayt Sıralayıcısı'nda sürüklenir	Sıra değişir, numaralar yeniden verilir
Bölüm	Giriş → Bölüm → Bölüm Ekle	Slaytlar adlandırılmış gruplara ayrılır
Gizle	Sağ tık → Slaydı Gizle	Slayt dosyada kalır, normal gösteride atlanır

Tablo 3.3.7: Temel slayt işlemleri.



Şekil 3.3.7: Slayt Sıralayıcısı'nda iki bölüm (Giriş ve Yöntem) ve gizli 4. slayt. Gizli slaydın küçük resmi soluk, numarası üstü çizili görünür.

Slayt numaraları slayda bağlı değildir; slaydın o anki sırasını gösterir. Bir slayt silinince ya da taşınınca arkasındaki bütün numaralar kayar. Gizli slayt ise numarasını korur, ama gösteride atlanır (Şekil 3.3.7).

Örnek 3.3.4

Sunuda sırayla P, Q, R, S slaytları var. Slayt bölmesinde aşağıdaki işlemler yapılıyor. Her adımdan sonra slayt sırası nedir ve F5 ile başlatılan gösteride hangi slaytlar görünür?

Adım	İşlem	Dosyadaki sıra	Slayt sayısı
0	Başlangıç	P Q R S	4
1	S seçilip Ctrl + D	P Q R S S′	5
2	Q seçilip Delete	P R S S′	4
3	P seçilip Ctrl + M (yeni slayt Y)	P Y R S S′	5
4	S′ sürüklenip P'nin önüne bırakılır	S′ P Y R S	5

Adım	İşlem	Dosyadaki sıra	Slayt sayısı
5	R seçilip Slaydı Gizle	S′ P Y R(gizli) S	5

Tablo 3.3.8: Slayt bölmesinde işlem izlemesi.

Gösteride S′, P, Y, S sırasıyla dört slayt görünür; R atlanır. Dosyada ise beş slayt vardır. 5. adımdan sonra R'nin numarası 4, S'ninki 5 olarak kalır. R'nin içeriği silinmedi; slaydın gizliliği yeniden Slaydı Gizle ile kaldırılabilir.

Sık yapılan hata

Gizli slaydı silinmiş saymak. Gizli slayt dosyanın içindedir: dosyayı alan herkes onu açıp görebilir. Paylaşılmaması gereken bir içerik gizlenerek değil, silinerek çıkarılır.

3.3.3.4 Bölümler

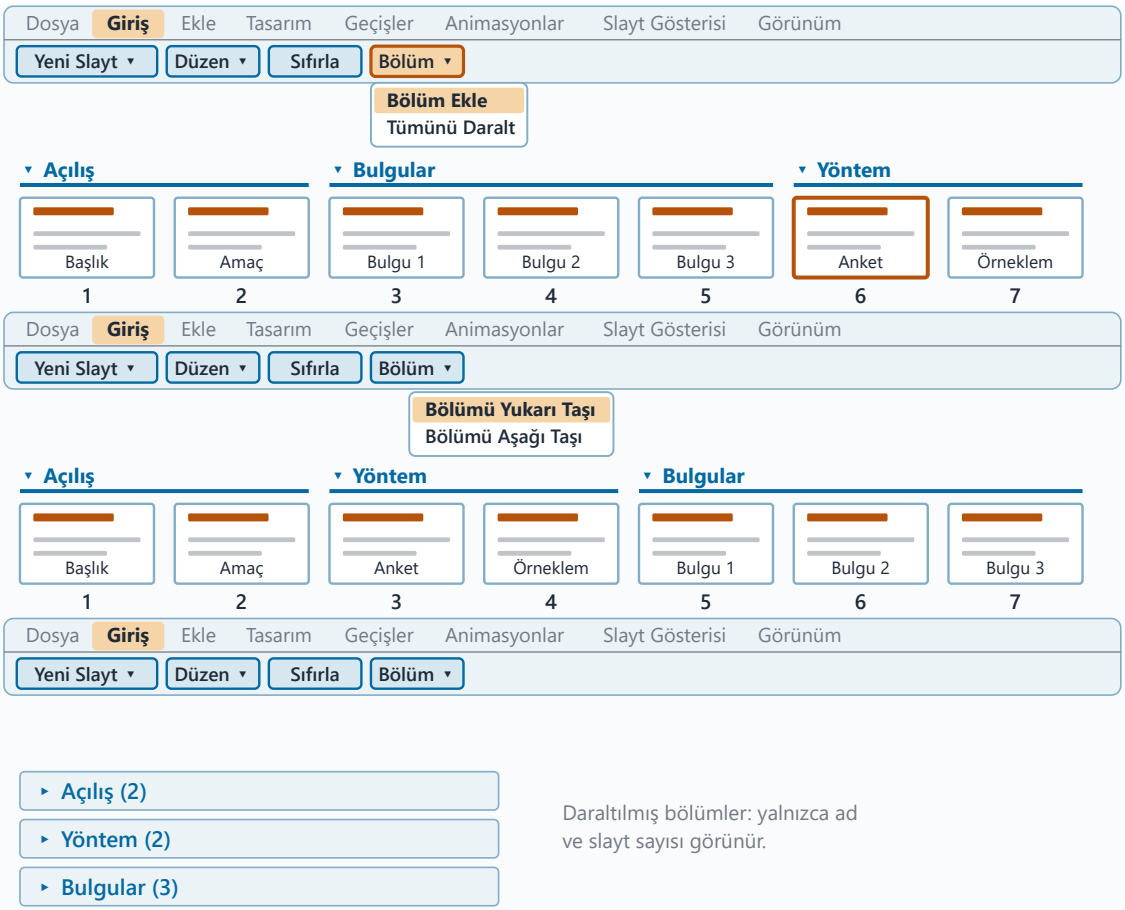
Kırk ya da elli slaytlık bir sunuda slayt bölmesi uzun bir şerit hâline gelir ve gezinmek zorlaşır. Bölümler bu sorunu çözer. Bir bölüm, ardışık slaytlardan oluşan ve bir adı olan gruptur. Bölümler yalnızca düzenleme ve gezinme içindir; gösteride görünmezler.

Tanım 3.3.7

Bölüm (*section*), slayt bölmesinde ve Slayt Sıralayıcısı'nda bir adla gösterilen ardışık slayt grubudur. Bir bölüm seçili slayttan başlar ve bir sonraki bölümün başına kadar sürer. Bölüm bir bütün olarak taşınabilir, daraltılabilir, yeniden adlandırılabilir ve silinebilir.

Bölümlerin en büyük yararı toplu taşımadır. Yanlış yerde kalmış bir slayt grubu, slaytlar tek tek sürüklenmeden yerine konur. Şekil 3.3.8 ve Tablo 3.3.9, bir örneği adım adım verir. Yedi slaytlık sunuda yöntem slaytları (Anket, Örneklem) yanlışlıkla bulguların arkasında kalmıştır.





Şekil 3.3.8: Bölüm ekleme ve yeniden sıralama. Birinci karede bölümsüz yedi slayt var. İkinci karede Açılış, Bulgular ve Yöntem bölümleri açılmıştır. Üçüncü karede Bölümü Yukarı Taşı ile Yöntem bölümü bulguların önüne geçmiştir. Dördüncü karede Tümünü Daralt ile yalnızca bölüm adları ve slayt sayıları görünür.

Adım	İşlem	Bölümler (slayt numaraları)	Slayt sırası
1	Başlangıç	Bölüm yok	Başlık, Amaç, Bulgu 1, Bulgu 2, Bulgu 3, Anket, Örneklem
2	Slayt 1 seçili, Bölüm Ekle, ad “Açılış”	Açılış (1–7)	Değişmedi
3	Slayt 3 seçili, Bölüm Ekle, ad “Bulgular”	Açılış (1–2), Bulgular (3–7)	Değişmedi
4	Slayt 6 seçili, Bölüm Ekle, ad “Yöntem”	Açılış (1–2), Bulgular (3–5), Yöntem (6–7)	Değişmedi
5	Yöntem bölümünde sağ tık → Bölümü Yukarı Taşı	Açılış (1–2), Yöntem (3–4), Bulgular (5–7)	Başlık, Amaç, Anket, Örneklem, Bulgu 1, Bulgu 2, Bulgu 3
6	Sağ tık → Tümünü Daralt	Açılış (2), Yöntem (2), Bulgular (3)	Değişmedi

Tablo 3.3.9: Bölüm ekleme ve bölüm taşıma işlemleri.

İkinci ile dördüncü adımlar arasında slayt sırası hiç değişmez; bölüm eklemek yalnızca gruplar. Asıl değişiklik beşinci adımdadır. Yöntem bölümü içindeki iki slaytla birlikte bir üste geçer ve numaralar yeniden verilir: Anket'in numarası 6'dan 3'e iner, Bulgu 1'inki 3'ten 5'e çıkar. Aynı işi bölümsüz yapmak için iki slaydı ayrı ayrı sürüklemek ve her seferinde doğru yere bırakmak gerekirdi.

Sık yapılan hata

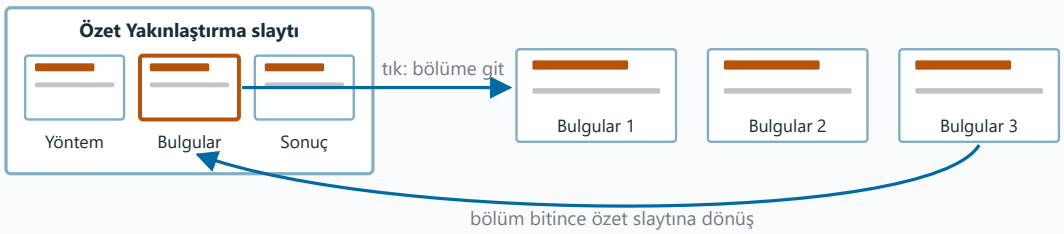
Bölüm adlarının gösteride bir ara başlık olarak görüneceğini sanmak. Bölüm adı yalnızca düzenleme ekranında görünür. İzleyicinin bölüm geçişini görmesi isteniyorsa bölümün başına ayrı bir başlık slaytı konur.

3.3.3.5 Yakınlaştırma ve köprüler

Gösteri normalde doğrusal ilerler: 1, 2, 3, … Bazen bu sıra yetmez. Soru-cevapta bir ayrıntıya hızla dönmek ya da izleyicinin ilgisine göre bölümlerden birini seçmek gerekebilir. PowerPoint'te bunun iki aracı vardır: yakınlaştırma ve köprü.

Tanım 3.3.8

Özet Yakınlaştırma (*Summary Zoom*), seçilen bölümlerin tıklanabilir küçük resimlerinden oluşan bir içindekiler slaytıdır. Küçük resme tıklanınca o bölüme geçilir; bölüm bitince gösteri özet slaytına döner. **Bölüm Yakınlaştırma** ve **Slayt Yakınlaştırma** tek bir bölüme ya da slayda atlar. Hepsi Ekle → Yakınlaştırma ile eklenir.



Şekil 3.3.9: Özet Yakınlaştırma slaytı. “Bulgular” küçük resmine tıklanınca Bulgular bölümünün üç slaydı gösterilir; bölüm bitince gösteri özet slaytına geri döner.

Köprü (*hyperlink*) ise bir nesneyi ya da metni başka bir hedefe bağlar: aynı sunudaki bir slayda, başka bir dosyaya ya da bir web adresine. Köprü Ekle → Bağlantı ile eklenir. Eylem düğmeleri (Ekle → Şekiller → Eylem Düğmeleri) “sonraki”, “önceki”, “ilk slayt”, “ana sayfa” gibi hazır hedefleri olan şekillerdir. Tıklamayla ilerlemeyen bilgi noktası (kiosk) gösterisinde izleyicinin gezinmesini sağlayan tek araçlar köprüler ve eylem düğmeleridir.

Yakınlaştırma güncel PowerPoint sürümlerinde bulunur. Sunu PDF'e aktarılınca yakınlaştırmanın küçük resimleri durağan resim olarak kalır ve hareket görünmez. Bu yüzden yakınlaştırmayla kurulmuş doğrusal olmayan bir akış PDF'te kendiliğinden doğrusal hâle gelir.

Sık yapılan hata

Web köprüsüne güvenip sunumda çevrim dışı bir yedek hazırlamamak. Salonun internet bağlantısı yoksa köprü hiçbir yere gitmez. Köprünün açacağı sayfanın ekran görüntüsü yedek bir slayda konur.

3.3.4 Tema, Asıl Slayt ve tasarım

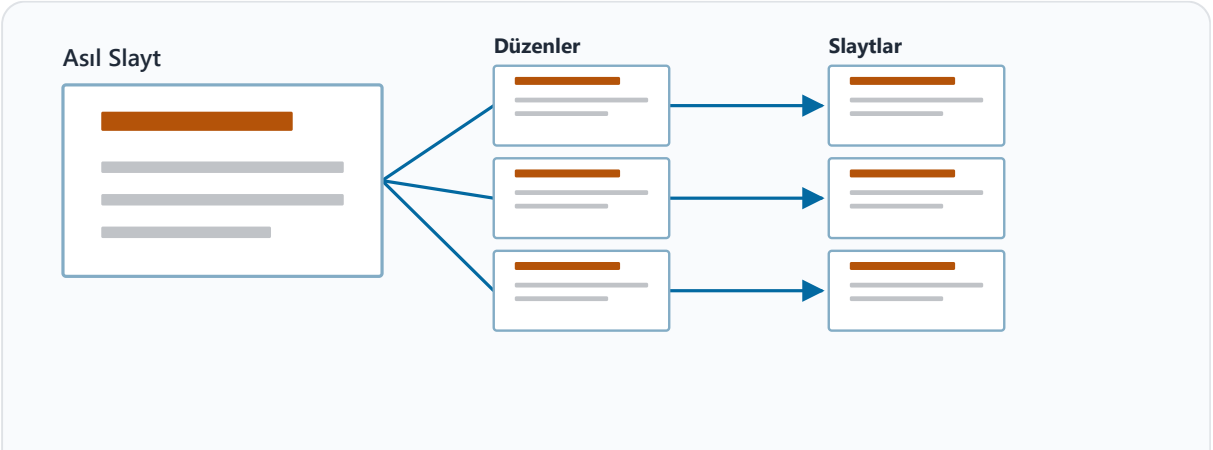
Bir sununun profesyonel görünmesinin sırrı süs değil, tutarlılıktır. Bütün başlıklar aynı yerde, aynı yazı tipinde ve aynı renkte durur. Aynı renkler her slaytta aynı anlama gelir. PowerPoint bu tutarlılığı iki araçla kurar: tema ve Asıl Slayt.

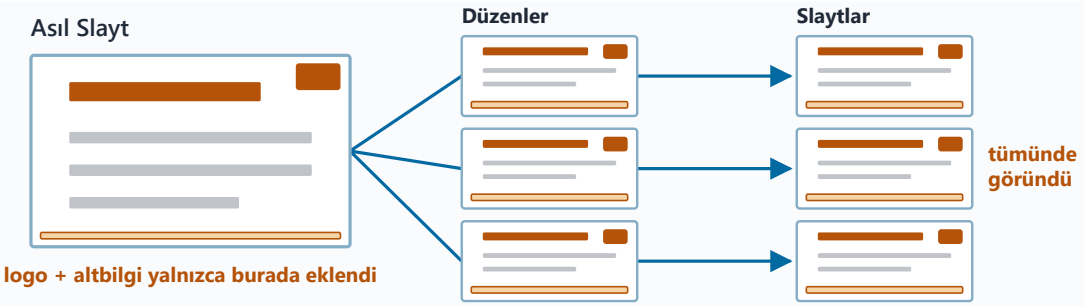
3.3.4.1 Tema ve Asıl Slayt

Tanım 3.3.9

Tema (*theme*), sununun renk paletini, başlık ve gövde yazı tiplerini, efektlerini ve arka planını bir arada tanımlayan tasarım kümesidir; Tasarım sekmesinden seçilir. **Asıl Slayt** (*slide master*), bütün düzenlerin ortak kurallarını taşıyan en üst slayttır: yer tutucuların konumu, yazı tipi, rengi, logo ve altbilgi gibi.

Asıl Slayt'ı anlamanın yolu, kuralların bir hiyerarşi içinde aşağı aktığını görmektir (Şekil 3.3.10). En üstte asıl slayt vardır. Altında ona bağlı düzenler (Başlık Slaytı, Başlık ve İçerik, İki İçerik, ...) bulunur. En altta ise bu düzenleri kullanan gerçek slaytlar yer alır. Asıl slayttaki bir kural düzenlere, düzenlerdeki kural da slaytlara aktarılır.





Şekil 3.3.10: Asıl Slayt, düzenler ve slaytlar. Üstte kuralların asıl slayttan düzenlere, düzenlerden slaytlara aktarılışı görülür. Altta logo ve altbilgi yalnızca asıl slayda eklenmiş, bütün düzenlerde ve slaytlarda aynı yerde belirmiştir.

Asıl Slayt'a Görünüm → Asıl Slayt ile girilir. Bu görünümde sol bölmede en üstte büyük küçük resimli asıl slayt, altında daha küçük düzen küçük resimleri görünür. Logo, kurum adı ya da slayt numarası gibi her slaytta bulunması gereken öğeler buraya bir kez eklenir. Sonradan eklenen slaytlarda da kendiliğinden görünürler.

Tema ile Asıl Slayt birlikte çalışır. Tema renk ve yazı tipini seçer; Asıl Slayt bu seçimlerin slaytta nereye ve nasıl uygulanacağını belirler. Tema değiştirildiğinde asıl slayt ve bütün düzenler yeni temaya göre yeniden çizilir.

Sık yapılan hata

Logoyu ilk slayda ekleyip o slaydı gerektiği kadar çoğaltmak. Bu yolla oluşan her logo ayrı bir nesnedir. Logo değişince bütün slaytlarda tek tek değiştirilmesi gerekir; yeni düzenle eklenen slaytlarda ise logo hiç görünmez. Her slaytta bulunacak bir öğenin yeri asıl slayttır.

3.3.4.2 Asıl Slayt'ta değişiklik ve elle biçim

Asıl slayt kuralları her zaman kazanmaz. Bir slayttaki nesneye doğrudan, elle verilen biçim, asıl slayttan gelen kuraldan önce gelir. Bu öncelik bilinmezse Asıl Slayt'ta yapılan bir değişikliğin bazı slaytlarda neden görünmediği anlaşılmaz.

Tanım 3.3.10

Elle biçim, tek bir slayttaki nesneye Giriş sekmesindeki yazı tipi, renk, boyut gibi komutlarla doğrudan verilen biçimdir. Elle biçim yalnızca değiştirilen özelliği kilitler; öteki özellikler asıl slayttan gelmeye devam eder. Giriş → Sıfırla, seçili slaydın yer tutucularını düzendeki konum, boyut ve biçime döndürür ve elle biçimi siler.

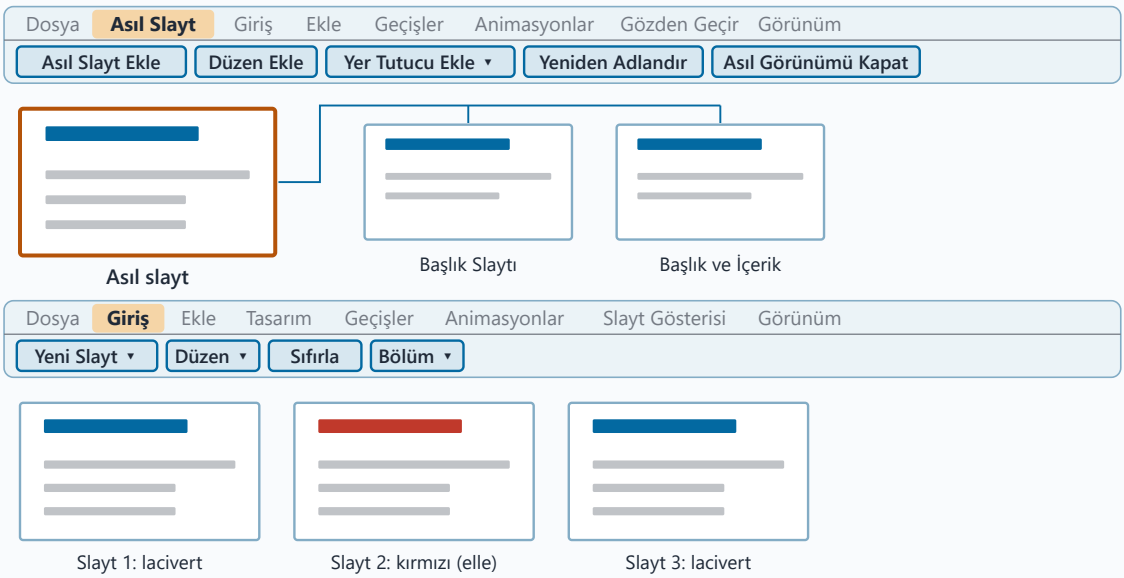
Tablo 3.3.10, bir örneği adım adım izler. Üç slaytlık bir sunuda Slayt 2'nin başlığı daha önce elle kırmızı yapılmıştır. Asıl Slayt'ta başlık rengi değiştirilecek, ayrıca resim alanlı yeni bir düzen tanımlanacaktır.

Örnek 3.3.5

Adım	Komut	Ne olur?	Slayt 1 / 2 / 3 başlığı
1	Görünüm → Asıl Slayt	Asıl Slayt sekmesi açılır	tema rengi / kırmızı (elle) / tema rengi
2	Asıl slaydın başlık yer tutucusuna lacivert renk	Bütün düzenlerin başlığı laciverte döner	(henüz görünmüyor)
3	Düzen Ekle, Yer Tutucu Ekle → Resim	Başlıklı yeni düzene resim alanı çizilir	(henüz görünmüyor)
4	Yeniden Adlandır: “Başlık ve Resim”	Düzen bu adla listelenecek	(henüz görünmüyor)
5	Asıl Görünümü Kapat	Normal görünüme dönülür	lacivert / kırmızı / lacivert
6	Slayt 2 seçili, Giriş → Sıfırla	Elle verilen kırmızı silinir	lacivert / lacivert / lacivert
7	Yeni Slayt → Başlık ve Resim	Slayt 4 resim alanıyla gelir	Slayt 4 de lacivert

Tablo 3.3.10: Asıl Slayt'ta değişiklik ve elle biçimin önceliği.

Beşinci adımdaki sonuç öncelik kuralının kendisidir. Slayt 1 ve 3'ün başlığına elle renk verilmemişti; renkleri asıl slayttan gelir ve lacivert olur. Slayt 2'nin başlık rengi ise elle verilmişti; asıl slayttaki değişiklik bu özelliği ezmez ve kırmızı kalır. Altıncı adımda Sıfırla bu kilidi kaldırır. Yedinci adım, yeni düzenin de asıl slayttaki lacivert kuralı aldığını gösterir.





Şekil 3.3.11: Asıl Slayt adım adım. Üstte asıl slaydın başlığı lacivert yapılmış, bütün düzenlere yansımıştır. Ortada Asıl Görünümü Kapat sonrası Slayt 2'nin elle verilmiş rengi korunur. Altta Sıfırla'dan sonra üç başlık da laciverttir ve yeni "Başlık ve Resim" düzeniyle Slayt 4 eklenmiştir.

Elle biçim yalnızca değiştirilen özelliği kilitler. Diyelim ki bir başlığa elle yalnızca başka bir yazı tipi verildi, rengine dokunulmadı. Asıl slaytta hem yazı tipi hem renk değiştirilirse bu başlıkta yazı tipi elle verildiği gibi kalır, ama renk yeni asıl slayt rengine döner.

Sık yapılan hata

Asıl Slayt'taki bir değişiklik bazı slaytlara yansımayınca değişikliği o slaytlarda elle tekrarlamak. Bu, kilidi kaldırmak yerine yeni bir kilit ekler. Doğru çözüm, yansımayan slaytları seçip Giriş → Sıfırla ile düzene döndürmektir.

3.3.4.3 Biçimi kopyalama ve toplu değiştirme

Tema ve Asıl Slayt'ın kapsamadığı durumlar için PowerPoint'te biçimi kopyalayan ya da toplu değiştiren araçlar da vardır (Tablo 3.3.11).

Araç	Sekme	Ne yapar?
Biçim Boyacısı	Giriş	Seçili nesnenin yazı tipi, renk ve kenarlık biçimini başka nesneye kopyalar; çift tıklanırsa birden çok nesneye uygulanır, Esc ile bırakılır.
Animasyon Boyacısı	Animasyonlar	Bir nesnenin animasyon efektini ve zamanlamasını başka nesneye kopyalar.
Tümüne Uygula	Geçişler	Seçili geçişi ve süresini bütün slaytlara verir.
Sıfırla	Giriş	Yer tutucuların konum, boyut ve biçimini düzene döndürür.
Yazı Tiplerini Değiştir	Giriş → Değiştir	Sunuda kullanılan bir yazı tipini her yerde başka bir yazı tipiyle değiştirir.

Tablo 3.3.11: Biçimi kopyalama ve toplu değiştirme araçları.

Bu araçların arkasındaki düşünce aynıdır. Elle biçim her nesnede ayrı durur; değişiklik gerekince her nesne tek tek düzeltilir. Tema, Asıl Slayt ve toplu araçlar ise bir kez değiştirilir ve değişiklik her yere yansır. Örneğin 40 slaytlık bir sunuda başlıkların yazı tipini değiştirmek, elle biçimde 40 ayrı işlem, Asıl Slayt'ta ise tek işlemdir.

Sık yapılan hata

Biçim Boyacısı'nı tek tıklayıp birden çok nesneye uygulamaya çalışmak. Tek tık yalnızca bir sonraki nesneye uygular ve araç kapanır. Birden çok nesne için düğmeye çift tıklanır; iş bitince Esc'ye basılır.

3.3.4.4 Görsel hiyerarşi

İzleyici bir slayda ilk baktığında gözü bir yere gider. Bu ilk bakış birkaç saniye sürer. Tasarımcının işi, gözün önce en önemli bilgiye gitmesini sağlamaktır.

Tanım 3.3.11

Görsel hiyerarşi, slayttaki öğelerin önem sırasının boyut, kalınlık, renk, konum ve boşlukla gösterilmesidir. En önemli öğe en büyük ya da en belirgin olandır; ayrıntı daha küçük, kaynak bilgisi en soluk yazılır.



Şekil 3.3.12: Görsel hiyerarşi. Üstte başlık, bulgu ve kaynak aynı boyut ve ağırlıktadır. Altta başlık sonucu söyler, %38 en büyük öğedir, ayrıntı küçük ve kaynak soluk yazılmıştır.

Örnek 3.3.6

Şekil 3.3.12'deki slaytta 2023'te 1.200, 2024'te 1.656 öğrenci kayıt olmuştur. Artış oranını bulalım ve slaydı yeniden düzenleyelim.

1. Artış miktarı: 1.656 − 1.200 = 456 öğrenci.

2. Artış oranı: 456 ÷ 1.200 = 0,38, yani %38.
3. Slaydın ana bulgusu bu orandır. Başlık konuyu değil sonucu söyler: “Kayıt sonuçları” yerine “Kayıtlar %38 arttı”.
4. %38 slaydın en büyük öğesi olur. Altında küçük yazıyla 1.200 → 1.656 öğrenci ve kısa bir neden (yeni bölümler ve burslar) yer alır.
5. Kaynak (Öğrenci İşleri, 2024) en küçük ve en soluk yazılır; gerektiğinde okunur ama dikkat çekmez.

Önceki slayttaki beş cümle aynı ağırlıktadır. İzleyici en önemli bilgiyi bulmak için hepsini okumak zorundadır. Düzenlenmiş slaytta ise göz önce %38'e gider; geri kalanı gerekirse okunur.

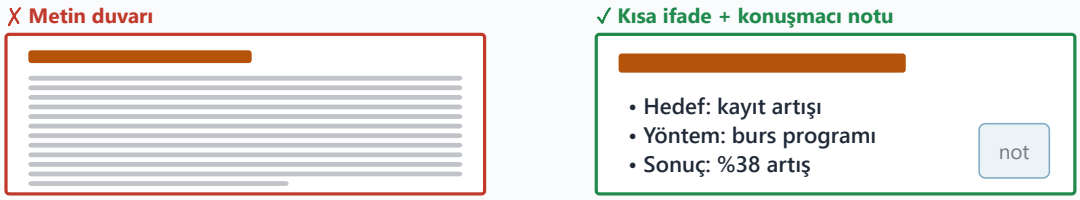
Görsel hiyerarşinin üç aracı vardır. Başlık konuyu ya da ana sonucu doğrudan söyler. Boyut ve renk önem sırasını gösterir; süs için kullanılmaz. Boşluk ve hizalama da birbirine ait öğeleri gruplar ve göze bir yol çizer. Birbirine yakın duran öğeler birbiriyle ilgili algılanır; ilgisiz öğeler arasına boşluk bırakılır.

Sık yapılan hata

Önemli bilgiyi vurgulamak için her şeyi kalın ve renkli yapmak. Her şey vurgulanırsa hiçbir şey vurgulanmış olmaz. Bir slaytta genellikle tek bir öğe öne çıkarılır.

3.3.4.5 Slaytta metin

Sunum slaydı bir belge sayfası değildir. İzleyici aynı anda hem uzun bir metni okuyup hem konuşmacıyı dinleyemez; ikisinden birini seçer ve genellikle okumayı seçer. Bu yüzden slayttaki metin kısa tutulur (Şekil 3.3.13).



Şekil 3.3.13: Solda paragraflarla dolu bir “metin duvarı”. Sağda üç kısa madde (hedef, yöntem, sonuç) ve slayda sığmayan anlatımın taşındığı konuşmacı notu.

Slayt metni için altı ilke vardır:

- **Kısa ifade:** Slaytta konuşmanın tamamı değil, ana nokta yer alır. Tam cümle yerine anahtar sözcük öbekleri çoğu zaman yeter.

- **Okunabilir boyut:** Salonun en arkasındaki izleyici düşünülür. Arka sıradan okunamayan bir yazı slayda konmamış sayılır.
- **Tutarlı dil:** Bir listedeki maddeler aynı dil yapısıyla yazılır. Biri "Verilerin toplanması", öteki "Analiz ettik" biçiminde olursa liste dağınık görünür.
- **Kontrast:** Metin rengi arka plandan belirgin biçimde ayrılır. Projeksiyon cihazı renkleri bilgisayar ekranından daha soluk gösterir.
- **Satır uzunluğu:** Satır, gözün tek bakışta izleyebileceği kadar kısa tutulur.
- **Konuşmacı notu:** Slayda sığmayan açıklama silinmez, notlar bölmesine taşınır.

> **Sık yapılan hata**
>
> Metin sığmayınca yazı boyutunu küçültmek. PowerPoint taşan metni kendiliğinden küçültebilir; ama küçük yazı sorunu çözmez, yalnızca gizler. Metin sığmıyorsa ya kısaltılır ya da iki slayda bölünür.

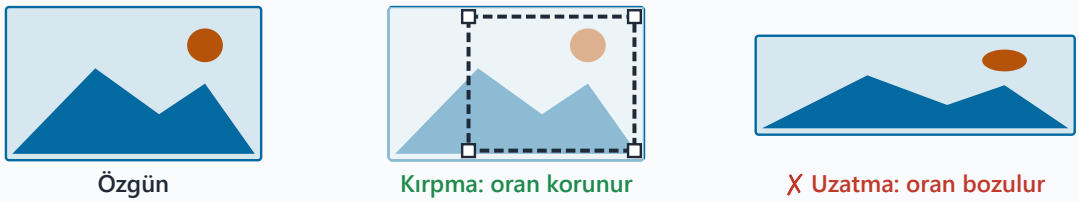
3.3.5 Görseller, tablolar ve grafikler

"Bir resim bin kelimeye bedeldir" sözü sunumlar için büyük ölçüde doğrudur, ama yalnızca doğru resim için. Konuyu açıklamayan bir görsel, slayda yer kaplayan bir süsten öteye geçmez. Bu kesim resim, şekil, tablo ve grafiklerin ne zaman ve nasıl kullanılacağını anlatır.

3.3.5.1 Resim kullanımı

Slayda resim Ekle → Resimler ile eklenir. Resim seçilince Resim Biçimi bağlamsal sekmesi açılır; kırpma, boyut, sıkıştırma ve alternatif metin komutları buradadır. Resim kullanırken altı noktaya dikkat edilir:

- **Amaç:** Görsel konuyu açıklar ya da bir iddiayı kanıtlar. Ölçüm düzeneğinin fotoğrafı yöntemi açıklar; rastgele bir "bilgisayar başında çalışan insan" fotoğrafı hiçbir şey açıklamaz.
- **Çözünürlük:** Resim, sunum ekranında net görünecek kadar ayrıntılı olur. Çok küçük bir resim büyütülünce bulanıklaşır.
- **Kırpma:** Önemli bölgeyi öne çıkarır. Kırparken oran korunur; resim yalnızca bir yönde çekilerek uzatılmaz.
- **Alternatif metin:** Görselin ne anlattığını ekran okuyucu kullanan izleyiciye söyler.
- **Kaynak:** Başkasına ait görselin lisansı ve atıf koşulu kontrol edilir.
- **Tutarlılık:** Benzer görseller aynı oran ve yerleşimle kullanılır.



Şekil 3.3.14: Aynı resmin üç hâli. Solda özgün resim. Ortada oran korunarak kırpılmış hâli. Sağda yalnızca yatay yönde çekilip uzatılmış hâli; dağlar basıklaşmış, güneş elipse dönmüştür.

Tanım 3.3.12

En-boy oranı (*aspect ratio*), bir resmin genişliğinin yüksekliğine oranıdır. 4000 × 3000 piksellik bir fotoğrafın oranı 4000 ÷ 3000 = 4 : 3, 1920 × 1080 piksellik bir ekranın oranı 16 : 9'dur. Resim yeniden boyutlandırılırken oran korunmazsa görüntü basıklaşır ya da uzar.

Örnek 3.3.7

4000 × 3000 piksellik (4 : 3) bir fotoğraf, 16 : 9 bir slaydı tam olarak kaplayacak. Fotoğraf uzatılmadan bu orana nasıl getirilir ve resmin ne kadarı kalır?

1. Genişlik korunur: 4000 piksel. 16 : 9 oranında yükseklik 4000 × 9 ÷ 16 = 2250 piksel olmalıdır.
2. Yükseklikten 3000 − 2250 = 750 piksel kırpılır. Konuya göre bu satırlar üstten, alttan ya da ikisinden paylaştırılarak atılır.
3. Kalan piksel sayısı 4000 × 2250 = 9.000.000'dur. Özgün 4000 × 3000 = 12.000.000 pikselin 9.000.000 ÷ 12.000.000 = 0,75'i, yani %75'i kalır.

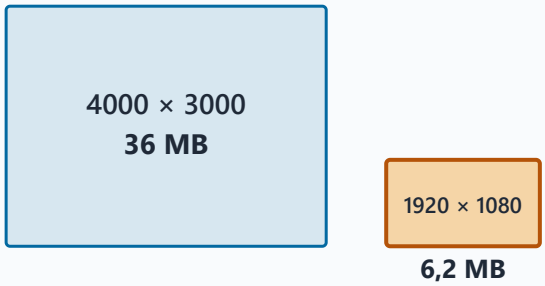
Aynı sonuca uzatarak ulaşmak, yani 4 : 3 resmi yatayda 16/9 ÷ 4/3 = 4/3 ≈ 1,33 kat çekmek de mümkündür. Ama bu durumda resimdeki her daire elipse, her kare dikdörtgene döner (Şekil 3.3.14).

Sık yapılan hata

Resmi kenar ortasındaki tutamaçtan çekerek büyütmek. Kenar ortasındaki tutamaç yalnızca bir boyutu değiştirir ve oranı bozar. Oranı korumak için köşe tutamacı kullanılır ya da Shift basılı tutulur.

3.3.5.2 Resim boyutu ve sıkıştırma

Telefonlar bile artık 12 megapikselin üzerinde fotoğraf çeker. Bu fotoğraflar slayda olduğu gibi eklenirse sunu dosyası hızla yüzlerce megabayta ulaşır. Oysa ekran bu piksellerin yalnızca bir kısmını gösterebilir.



Şekil 3.3.15: 4000 × 3000 piksellik fotoğraf ile 1920 × 1080 piksellik ekranın boyut karşılaştırması. Sıkıştırılmamış hâlde fotoğraf 36 MB, ekranı dolduran görüntü yaklaşık 6,2 MB'tır.

Örnek 3.3.8

4000 × 3000 piksellik bir fotoğraf 16 : 9 slaytta tam ekran gösterilecek. Projeksiyon 1920 × 1080 piksellik görüntü veriyor. Her piksel sıkıştırılmadan 3 baytla (kırmızı, yeşil, mavi için birer bayt) saklanırsa fotoğrafın gereksiz kısmı ne kadardır?

1. Fotoğrafın ham boyutu: 4000 × 3000 × 3 = 36.000.000 bayt, yani 36 MB.
2. Ekranı dolduran görüntünün ham boyutu: 1920 × 1080 × 3 = 6.220.800 bayt, yani yaklaşık 6,2 MB.
3. Oran: 36.000.000 ÷ 6.220.800 ≈ 5,8. Fotoğraf, ekranın gösterebileceğinden yaklaşık 5,8 kat fazla veri taşır.

Fazladan pikseller ekranda görünmez; yalnızca dosyayı büyütür, açılışı ve kaydı yavaşlatır, e-postayla gönderimi zorlaştırır. Gerçek dosyada resimler JPEG ya da PNG olarak sıkıştırılmış durur, bu yüzden boyutlar daha küçüktür. Ama oran aşağı yukarı korunur.

Tanım 3.3.13

ppi (*pixels per inch*, inç başına piksel), bir resmin bir inçlik (2,54 cm) uzunluğa kaç piksel düştüğünü gösterir. Resmin slayttaki fiziksel boyutu sabitken ppi yükselirse piksel sayısı, dolayısıyla dosya boyutu da artar.

Resim Biçimi → Resimleri Sıkıştır komutu, resmi hedeflenen ppi değerine indirir. Hedef kullanım için uygun değer seçilir: baskı için 220 ppi, web ve ekran için 150 ppi, e-posta için 96 ppi gibi. Aynı pencerede “Resimlerin kırpılan alanlarını sil” seçeneği de bulunur.

Örnek 3.3.9

16 : 9 bir slayt 13,333 × 7,5 inç (33,867 × 19,05 cm) boyutundadır. Slaydı tam kaplayan bir resim farklı ppi değerlerine sıkıştırılırsa kaç piksel olur?

Hedef	ppi	Genişlik × yükseklik	Ham boyut
Baskı	220	2933 × 1650	≈ 14,5 MB
Web ve ekran	150	2000 × 1125	6,75 MB
E-posta	96	1280 × 720	≈ 2,8 MB

Tablo 3.3.12: Tam slayt resminin piksel boyutu ve ham boyutu (piksel başına 3 bayt).

Hesap her satırda aynıdır: genişlik 13,333 × ppi, yükseklik 7,5 × ppi. Örneğin 150 ppi'de 13,333 × 150 ≈ 2000 ve 7,5 × 150 = 1125 piksel bulunur. Bu değer 1920 × 1080'lik projeksiyonu karşılar. 4000 × 3000'lik fotoğraf 150 ppi'ye sıkıştırıldığında ekranda bir kayıp olmaz, ama dosya belirgin biçimde küçülür.

Kırpma ile sıkıştırma arasında önemli bir ilişki vardır. Resim Biçimi → Kırp ile gizlenen bölüm aslında silinmez; dosyada kalır ve kırpma her an geri alınabilir. Sıkıştırırken "kırpılan alanları sil" seçilmezse dosya boyutu azalmaz. Resim Biçimi → Arka Planı Kaldır ise resmin arka planını saydam yapar; konu nesnesi slaydın rengine oturur.

Sık yapılan hata

Fotoğrafı kırpınca dosyanın küçüldüğünü sanmak. Kırpılan alan sıkıştırma sırasında silinene kadar dosyada durur. Kırpılan alanlar silindikten sonra ise kırpma artık geri alınamaz; bu adım sunu bittikten sonra yapılır.

3.3.5.3 Şekiller, simgeler ve SmartArt

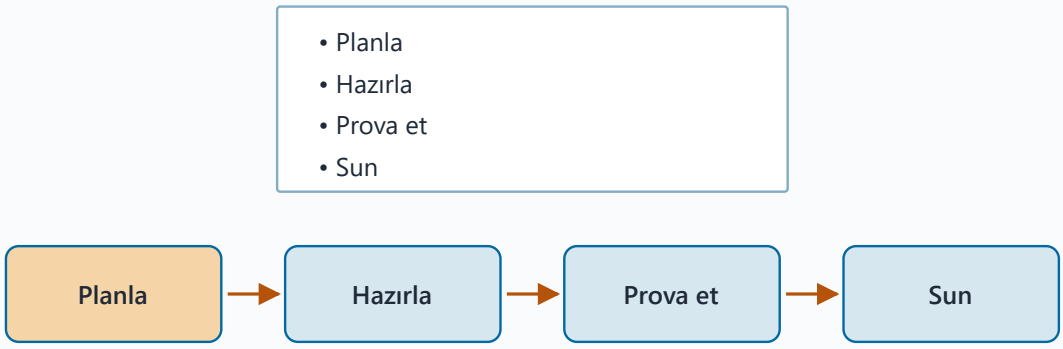
Her görsel bir fotoğraf değildir. Bir süreç, bir sıralama ya da iki şey arasındaki ilişki çoğu zaman basit şekillerle daha iyi gösterilir. PowerPoint'te bunun için birkaç araç vardır (Tablo 3.3.13).

Araç	Ne için?
Şekil	Basit vurgu, süreç ve ilişki göstermek (Ekle → Şekiller)
Simge	Tanıdık bir kavramı küçük alanda temsil etmek (Ekle → Simgeler)
SmartArt	Liste ya da süreci hazır bir düzenle görselleştirmek
Metin kutusu	Yer tutucu dışında sınırlı metni konumlandırmak
Bağlayıcı	Şekiller taşındığında da bağlı kalan çizgi ya da ok

Tablo 3.3.13: Şekil türündeki görsel araçlar.

Tanım 3.3.14

SmartArt, metni süreç, döngü, hiyerarşi ya da liste gibi hazır bir grafik düzenine yerleştiren araçtır. Metin, SmartArt'ın içeriği olarak kalır ve düzenlenebilir; düzen değişince kutular ve oklar kendiliğinden yeniden çizilir.



Şekil 3.3.16: Üstte sıralı dört adım düz madde listesi olarak yazılmıştır. Altta Giriş → SmartArt'a Dönüştür → Temel Süreç ile aynı metin oklarla bağlı kutulara dönüşmüştür.

Şekil 3.3.16, madde listesinin bir eksiğini gösterir: düz liste sırayı göstermez. “Planla, Hazırla, Prova et, Sun” maddeleri alt alta yazıldığında bunların bir süreç olduğu ancak okununca anlaşılır. Aynı metin SmartArt'a dönüştürülünce oklar sıra ilişkisini ilk bakışta verir. Metin kaybolmaz; SmartArt'ın içine taşınır.

Şekillerle çalışırken bir ilke daha vardır: renk, çizgi kalınlığı ve stil bütün sunuda tutarlı tutulur. Bir slaytta mavi kutu “girdi”, kırmızı kutu “çıktı” anlamına geliyorsa aynı renkler öteki slaytlarda da aynı anlamı taşır.

Sık yapılan hata

Şekilleri düz çizgilerle birbirine bağlamak. Şekil taşınınca düz çizgi yerinde kalır ve bağlantı kopar. Bağlayıcı ise uçlarını şekillerin bağlantı noktalarına tutturur; şekil taşınınca onunla birlikte uzar ya da kısalır.

3.3.5.4 Tablolar

Tablo, sayıları ya da özellikleri satır ve sütunlarda karşılaştırmanın en açık yoludur. Ama sunumdaki tablo bir rapordaki tablodan farklıdır. Raporu okuyan kişi istediği kadar zaman harcayabilir; sunumu izleyen kişinin ise birkaç saniyesi vardır.

✗ Yoğun: 6 sütun, küçük sayılar

S1	S2	S3	S4	S5	S6
47	10	63	26	79	42
84	47	10	63	26	79
31	84	47	10	63	26
68	31	84	47	10	63
15	68	31	84	47	10

✓ Sade: yalnızca mesaj için gerekenler

Yıl	Kayıt
2023	1.200
2024	1.656

Şekil 3.3.17: Solda altı sütunlu, küçük sayılarla dolu yoğun bir tablo. Sağda yalnızca mesaj için gereken iki satır: 2023'te 1.200, 2024'te 1.656 kayıt; önemli hücre vurgulanmış.

Sunumda tablo hazırlanırken şunlara dikkat edilir:

- Başlık satırı her sütunun anlamını ve birimini açıklar.
- Yalnızca karşılaştırma için gereken sütunlar gösterilir. Altı sütunun ikisi mesajı taşıyorsa öteki dördü slayttan çıkarılır, gerekirse yedek slayda konur.
- Önemli hücre ölçülü bir renk ya da kalınlıkla belirginleştirilir.
- Metin sola, sayılar sağa hizalanır. Sağa hizalı sayılarda birler, onlar, yüzler basamağı alt alta gelir ve büyüklükler göz ucuyla karşılaştırılır.
- Verinin kaynağı ve tarihi gerektiğinde tablonun altına küçük yazıyla eklenir.
- Tablo hâlâ yoğunsa bir grafik ya da birkaç ana sayı düşünülür.

Sık yapılan hata

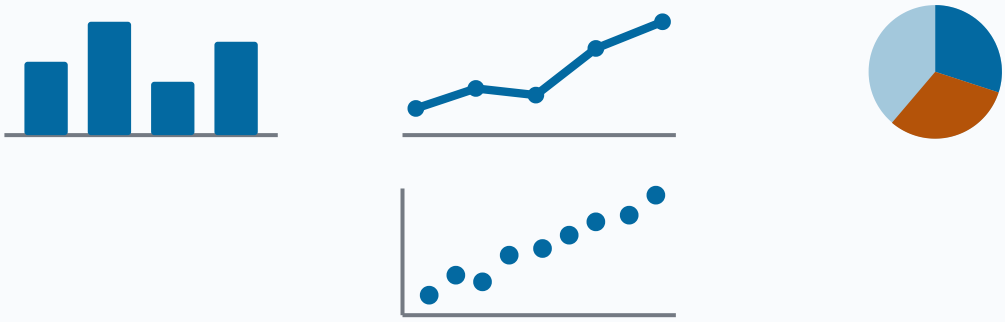
Excel'deki tabloyu bütün satır ve sütunlarıyla slayda yapıştırmak. Yazı boyutu küçülür, arka sıralar hiçbir sayıyı okuyamaz. Slayttaki tablo, mesajı taşıyan birkaç satır ve sütuna indirgenir.

3.3.5.5 Grafikler

Grafik, sayılar arasındaki ilişkiyi şekle çevirir. Hangi grafiğin seçileceği verinin türüne değil, iletilmek istenen mesaja bağlıdır (Tablo 3.3.14).

Mesaj	Grafik	Örnek
Kategorileri karşılaştırmak	Çubuk ya da sütun	Bölümlerin öğrenci sayıları
Zaman içindeki değişimi göstermek	Çizgi	Aylık elektrik tüketimi
Parçanın bütüne oranını göstermek	Az kategorili pasta ya da yüzde çubuk	Bütçenin kalemlere dağılımı
İki değişken arasındaki ilişkiyi göstermek	Dağılım	Çalışma saati ile sınav notu

Tablo 3.3.14: Mesaja göre grafik türü.



Şekil 3.3.18: Soldan sağa sütun, çizgi, pasta ve dağılım grafiklerinin küçük örnekleri.

Örnek 3.3.10

Bir öğrencinin aylık bütçesi üç kaleme ayrılıyor: kira 4.500 TL, yemek 3.000 TL, ulaşım 1.500 TL. "Bütçenin yarısı kiraya gidiyor" mesajı için hangi grafik uygundur ve grafikteki paylar nedir?

1. Mesaj parça-bütün ilişkisidir ve yalnızca üç kategori vardır: pasta grafik uygundur.
2. Toplam: 4.500 + 3.000 + 1.500 = 9.000 TL.
3. Paylar: kira 4.500 ÷ 9.000 = %50, yemek 3.000 ÷ 9.000 ≈ %33,3, ulaşım 1.500 ÷ 9.000 ≈ %16,7.
4. Pasta dilimlerinin açıları payların 360°'deki karşılığıdır: kira 0,5 × 360 = 180°, yemek 120°, ulaşım 60°. Kira dilimi dairenin tam yarısını kaplar; mesaj ilk bakışta görülür.
5. Grafik başlığı mesajı söyler: "Bütçenin yarısı kiraya gidiyor".

Kalem sayısı yediye, sekize çıkarsa dilimler incelir ve birbirine benzer. Bu durumda yüzde çubuk ya da sıralı bir çubuk grafik daha okunaklıdır.

Her grafik türü için geçerli üç kural vardır. Grafik başlığı ana sonucu açıkça söyler. Eksenler ve birimler görünür. Üç boyutlu efektler kullanılmaz; çünkü perspektif, öndeki çubuğu ya da dilimi olduğundan büyük gösterir ve karşılaştırmayı bozar.

Sık yapılan hata

Zaman içindeki değişimi pasta grafikle göstermek. Pasta grafik tek bir anın paylarını gösterir; iki yılın pastası yan yana konunca dilimler karşılaştırılamaz. Zaman ekseni olan veri çizgi grafikle verilir.

3.3.5.6 Gömülü ve bağlı grafik

Grafikler çoğu zaman Excel'de hazırlanır ve sunuya taşınır. Taşımanın üç yolu vardır ve her birinde verinin nerede durduğu farklıdır (Tablo 3.3.15).

Tanım 3.3.15

Gömülü nesne (*embedded object*), verisiyle birlikte sunu dosyasının içine kopyalanmış nesnedir; kaynak dosyadan bağımsızdır. **Bağlı nesne** (*linked object*) ise verisini kaynak dosyadan okur; sunu yalnızca kaynağın yolunu tutar.

Tür	Nasıl oluşur?	Veri nerede?	Excel dosyası değişince
Gömülü grafik	Ekle → Grafik; ya da yapıştırırken çalışma kitabını ekleme seçeneği	Sunu dosyasının içinde	Grafik değişmez; veri Verileri Düzenle ile sunu içinde güncellenir
Bağlı grafik	Yapıştırırken veriyi bağlama seçeneği	Kaynak Excel dosyasında	Grafik güncellenebilir; dosya taşınır ya da silinirse bağlantı kopar
Resim	Yapıştırırken resim seçeneği	Veri yok, yalnızca görüntü	Değişmez; grafik artık düzenlenemez

Tablo 3.3.15: Excel grafiğini sunuya taşımanın üç yolu.

Örnek 3.3.11

İki durum için uygun yöntemi seçelim.

1. Sunum bir konferans salonundaki başka bir bilgisayarda yapılacak ve yalnızca .pptx dosyası USB bellekle götürülecek. Excel dosyası yanında olmayacak. Grafiğin verisi dosyanın içinde olmalıdır: **gömülü grafik**. Bağlantı kopma riski yoktur.
2. Her ay aynı düzende bir tüketim raporu sunuluyor ve rakamlar her ay Excel'de güncelleniyor. Rakamları her ay sunuya tek tek kopyalamak yerine grafiğin Excel'den beslenmesi istenir: **bağlı grafik**. Sunu ve Excel dosyası aynı klasörde tutulur ve klasör birlikte taşınır.

Sık yapılan hata

Bağlı grafikli sunuyu başka bilgisayara yalnız götürmek. Grafik son kaydedilen görüntüsüyle görünür, ama güncellenmek istendiğinde PowerPoint kaynak dosyayı bulamaz. Ya Excel dosyası da götürülür ya da grafik gömülü hâle getirilir.

3.3.6 Hizalama ve yerleşim

İzleyici, aralarında birkaç piksellik fark olan iki kutunun neden aynı hizada durmadığını bilinçli olarak fark etmeyebilir. Ama slaydın "dağınık" göründüğünü hisseder. Gözle yerleştirme bu yüzden yetersizdir. PowerPoint, nesneleri sayısal olarak hizalayan araçlar sunar.

3.3.6.1 Yerleşim araçları

Tanım 3.3.16

Hizala (*align*), seçili nesnelerin kenarlarını ya da merkezlerini ortak bir çizgiye getirir: sola, ortaya, sağa, üste, merkeze, alta. **Dağıt** (*distribute*), en dıştaki iki nesneyi yerinde bırakıp aradaki boşlukları eşitler. İkisi de Şekil Biçimi → Hizala menüsündedir.

Araç	Ne yapar?
Hizala	Nesnelerin kenar ya da merkezlerini ortak çizgiye getirir
Dağıt	Nesneler arasındaki boşluğu eşitler
Kılavuz ve ızgara	Slayt boyunca ve slaytlar arasında aynı yerleşim çizgilerini sağlar (Görünüm sekmesi)
Grupla	Birden fazla nesneyi tek nesne gibi taşır ve boyutlandırır (Ctrl + G)
Öne Getir / Arkaya Gönder	Üst üste binen nesnelerin sırasını değiştirir
Seçim Bölmesi	Nesneleri adıyla listeler; görünürlüğü ve katman sırasını yönetir

Tablo 3.3.16: Yerleşim araçları.



Şekil 3.3.19: Üst üste binen Başlık, Daire ve Fotoğraf nesneleri ile Seçim Bölmesi. Listede en üstteki nesne slaytta en öndedir; göz simgesi nesneyi gizler ya da gösterir.

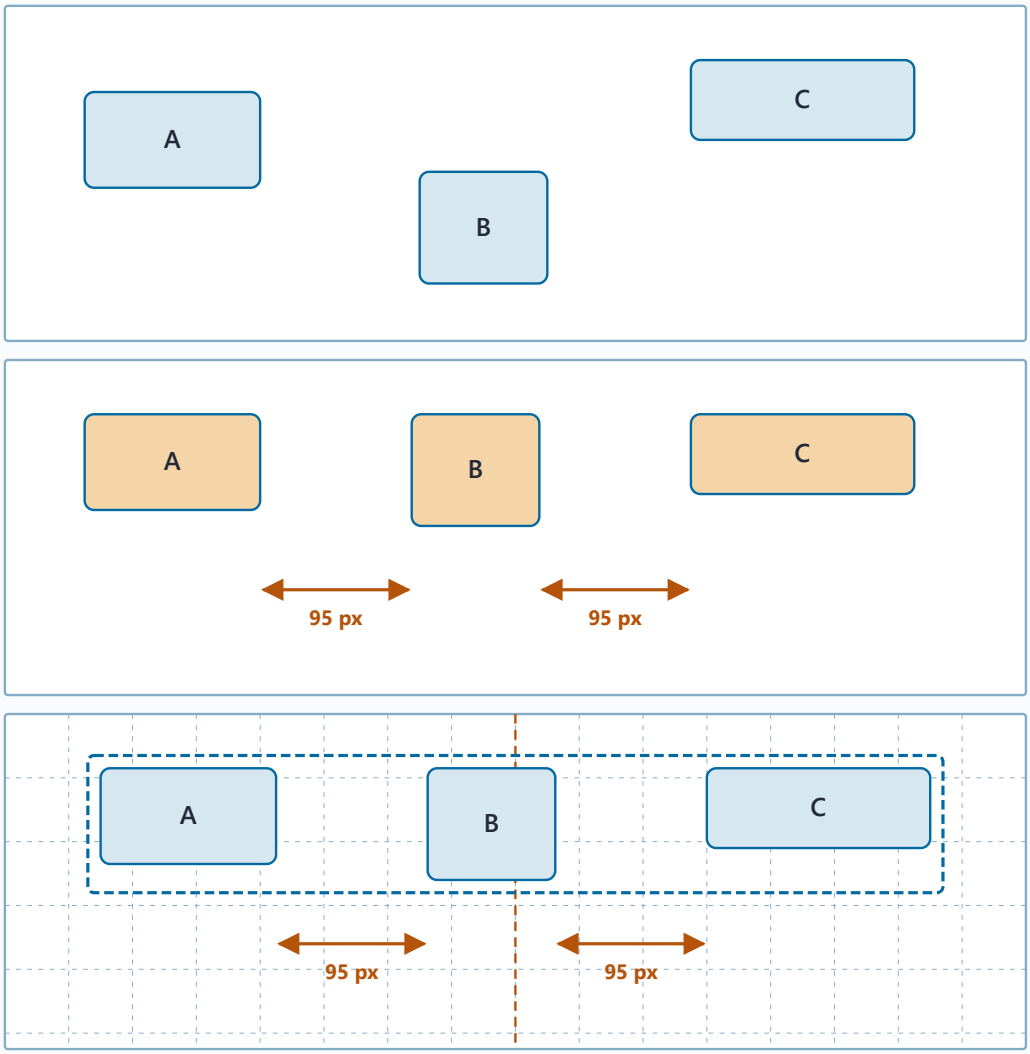
Seçim Bölmesi kalabalık slaytlarda özellikle işe yarar (Şekil 3.3.19). Üst üste binen nesnelerden arkadakini fareyle seçmek zordur; bölmede adına tıklamak yeter. Listede yukarıdaki nesne slaytta öndedir. Öne Getir ve Arkaya Gönder listedeki sırayı, dolayısıyla slayttaki katman sırasını değiştirir. Aynı bölmede nesnelere anlamlı adlar verilebilir. Bu adlar Animasyon Bölmesi'nde de görünür; "Resim 7" yerine "Grafik" yazan bir animasyon listesi çok daha kolay okunur.

Sık yapılan hata

Hizalamadan sonra nesneleri tek tek taşıyıp düzeni bozmak. Birlikte kalması gereken nesneler hizalandıktan sonra gruplanır. Grup tek nesne gibi taşınır ve içindeki aralıklar korunur.

3.3.6.2 Hizala ve dağıt adım adım

Hizala ve dağıt komutları sayısal olarak çalışır. Bu yüzden sonuçları önceden hesaplanabilir. Şekil 3.3.20 ve Tablo 3.3.17, üç şekli adım adım izler. Slayt çerçevesi yatayda 40'tan 680'e kadar uzanır; şekillerin konumu sol üst köşelerinin koordinatıyla verilir.



Şekil 3.3.20: Hizala ve dağıt. Üstte üç şekil gözle yerleştirilmiştir. Ortada Üste Hizala ve Yatay Dağıt sonrası üst kenarlar aynı çizgidedir, aralıklar 95 px'tir. Altta şekiller gruplanıp slayt ortasına alınmıştır.

Örnek 3.3.12

Başlangıçta A (x = 90, y = 60, genişlik 110), B (x = 300, y = 110, genişlik 80) ve C (x = 470, y = 40, genişlik 140) şekilleri var.

Adım	Komut	A (x, y)	B (x, y)	C (x, y)
0	Gözle yerleştirme	90, 60	300, 110	470, 40
1	Üçü seçili, Hizala → Üste Hizala	90, 40	300, 40	470, 40
2	Hizala → Yatay Dağıt	90, 40	295, 40	470, 40
3	Grupla, Hizala → Ortala	100, 40	305, 40	480, 40

Tablo 3.3.17: Hizala, dağıt ve grupla işlemlerinde koordinatlar.

1. **Üste Hizala:** En üstteki şeklin üst kenarı C'ninkidir (y = 40). A ve B'nin y değeri 40'a çekilir; x değerleri değişmez.
2. **Yatay Dağıt:** En soldaki A ile en sağdaki C yerinde kalır. A'nın sol kenarı 90, C'nin sağ kenarı 470 + 140 = 610'dur. Bu aralığın uzunluğu 610 − 90 = 520'dir. Şekillerin genişlikleri toplamı 110 + 80 + 140 = 330 olduğundan boş alan 520 − 330 = 190'dır. İki aralık eşit olacağı için her biri 190 ÷ 2 = 95 olur. B'nin yeni x değeri 90 + 110 + 95 = 295'tir. Denetim: B'nin sağ kenarı 295 + 80 = 375, C'ye uzaklığı 470 − 375 = 95.
3. **Grupla ve ortala:** Grup 90'dan 610'a uzanır; merkezi (90 + 610) ÷ 2 = 350'dir. Slayt çerçevesinin merkezi (40 + 680) ÷ 2 = 360'tır. Grup 360 − 350 = 10 birim sağa kayar. Her şekil aynı miktarda kaydığı için aralıklar 95'te kalır.

Dağıt komutunun en dıştaki nesnelere dokunmadığı unutulmamalıdır. Dağıtılan nesnelerin slaytta tam ortada durması isteniyorsa önce dağıtılır, sonra gruplanıp ortalanır.

Görünüm sekmesindeki kılavuzlar ve ızgara çizgileri bu hesapları slaytlar arasında taşır. Örneğin bütün slaytlarda içerik soldan aynı kılavuz çizgisinde başlarsa slayt değiştiğinde metin “zıplamaz”. Ortadaki kesikli çizgi slaydın merkezini gösterir.

Sık yapılan hata

Yalnızca iki nesne seçiliyken Dağıt'ı denemek. Dağıtılacak bir ara nesne olmadığı için hiçbir şey değişmez. Dağıt en az üç nesneyle anlam kazanır; iki nesne yalnızca hizalanır.

3.3.7 Geçiş ve animasyon

PowerPoint'te iki ayrı hareket türü vardır ve ikisi sık karıştırılır. Biri bir slayttan ötekine geçerken ekranın nasıl değiştiğini, öteki bir slaydın içindeki nesnelerin nasıl belirdiğini ya da kaybolduğunu belirler. Hareket iyi kullanılınca dikkati yönetir ve bir süreci adım adım açar. Kötü kullanılınca ise içerikten

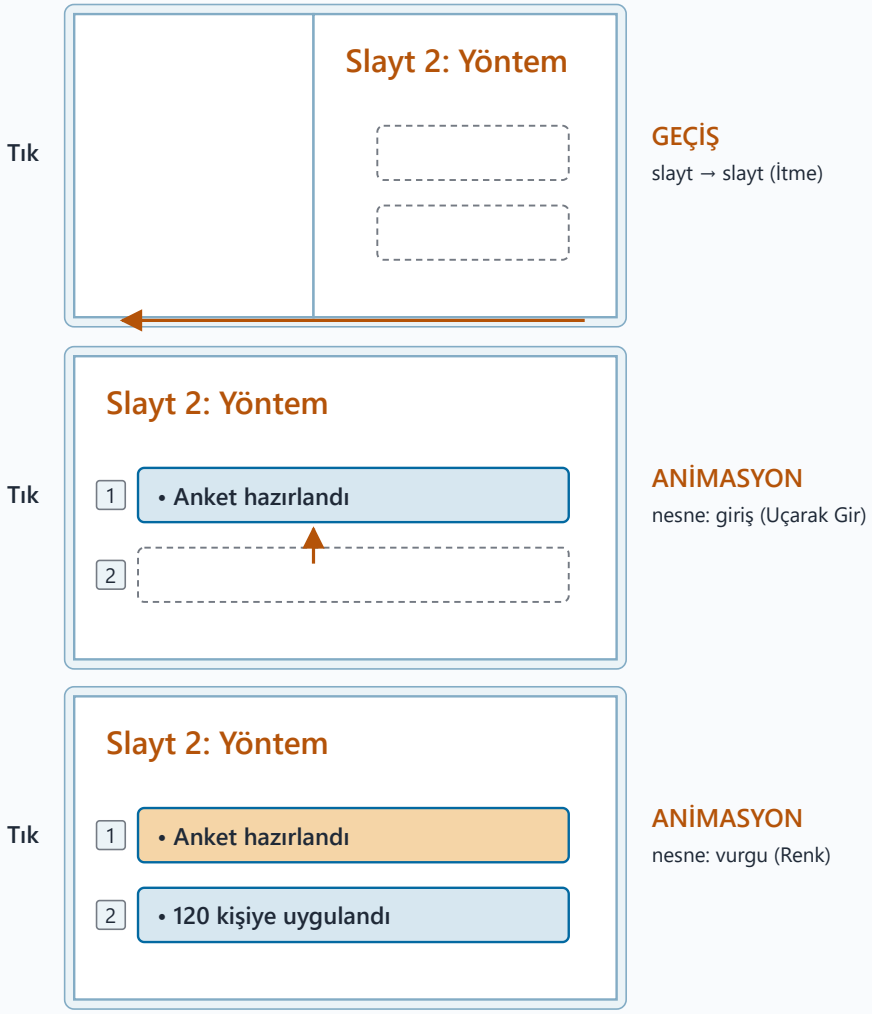
daha çok dikkat çeker.

3.3.7.1 Geçiş ile animasyonun farkı

Tanım 3.3.17

Geçiş (*transition*), bir slayttan sonraki slayda geçilirken ekranın değişme biçimidir ve slaydın bütününe uygulanır; Geçişler sekmesinden verilir. **Animasyon** (*animation*), slayt içindeki tek bir nesnenin (başlık, madde, resim, grafik) belirmesi, vurgulanması, kaybolması ya da hareket etmesidir; Animasyonlar sekmesinden verilir.

Farkı görmenin en iyi yolu, gösteride her tıklamanın ne yaptığını izlemektir. Şekil 3.3.21 ve Tablo 3.3.18, iki slaytlık bir gösteriyi tıklama tıklama verir. Slayt 2'ye İtme geçişi verilmiştir. Slayt 2'nin iki maddesine giriş animasyonu, birinci maddeye ayrıca bir vurgu animasyonu eklenmiştir.



Şekil 3.3.21: Geçiş ile animasyon. Üstte tıklamayla Slayt 2 sağdan gelip Slayt 1'i iter: geçiş slaydın bütününe uygulanır. Ortada slayt değişmeden birinci madde giriş animasyonuyla belirir. Altta zaten görünen birinci madde-

ye vurgu animasyonu uygulanır.

Tıklama	Ne olur?	Türü	Ekrandaki slayt
Başlangıç	Gösteri başlar	—	Slayt 1
1	Slayt 2 sağdan gelip Slayt 1'i iter; maddeler henüz görünmez	Geçiş (İtme)	Slayt 2
2	“Anket hazırlandı” maddesi belirir	Giriş animasyonu	Slayt 2
3	“120 kişiye uygulandı” maddesi belirir	Giriş animasyonu	Slayt 2
4	Birinci maddenin rengi değişir	Vurgu animasyonu	Slayt 2
5	Sonraki slayda geçilir	Geçiş	Slayt 3

Tablo 3.3.18: İki slaytlık gösteride her tıklamanın etkisi.

Tabloda geçiş yalnızca slayt değiştiğinde görülür. Animasyonlar ise slayt yerinde dururken çalışır. Slayt 2'de üç animasyon bulunduğu için Slayt 2'den çıkmak dört tıklama ister: üçü animasyonlar, dördüncüsü geçiş içindir. PowerPoint'in kuralı şudur: bir slaydın tıklamayla başlayan bütün animasyonları bitmeden tıklama sonraki slayda geçirmez; önce sıradaki animasyonu başlatır.

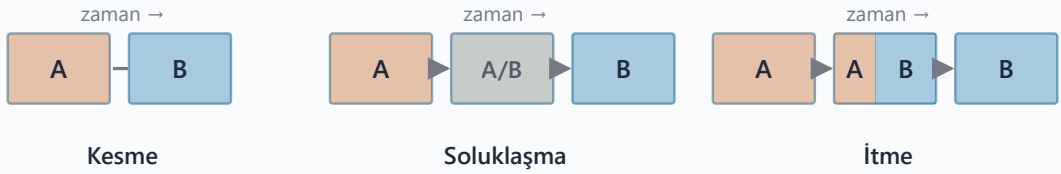
Maddelerin tek tek gelmesi anlatımı izleyicinin hızına bağlar. İzleyici henüz konuşulmayan maddeyi okumaz; dikkati konuşmacının o anda anlattığı noktada kalır. Düzenleme görünümünde nesnelerin yanında görünen küçük numaralar bu sırayı gösterir. Bu numaralar gösteride görünmez.

Sık yapılan hata

Maddelerin tek tek gelmesi için her maddeyi ayrı bir slayda koyup geçiş vermek. Bu, slayt sayısını gereksiz yere artırır ve her tıklamada bütün ekran değişir. Aynı slayttaki maddelerin sırayla gelmesi animasyonun işidir.

3.3.7.2 Geçişler

Geçiş, seçili slayda Geçişler sekmesinden verilir ve o slayt ekrana gelirken oynar. Şekil 3.3.22, üç yaygın geçiş türünü zaman içinde gösterir.



Şekil 3.3.22: Üç geçiş türü. Kesmede A bir anda B'ye döner. Soluklaşmada A ile B kısa bir süre üst üste görünür. İtmede B gelirken A'yı ekrandan dışarı iter.

Ayar	Ne belirler?
Efekt	Slaydın ekrana nasıl geldiğini
Süre	Geçişin kaç saniye süreceğini
Efekt Seçenekleri	Yönü ya da çeşidi (ör. İtme'nin sağdan mı alttan mı geleceğini)
Tümüne Uygula	Seçili geçişi ve süresini bütün slaytlara vermeyi
Slayt ilerletme	Sonraki slayda tıklamayla mı, belirli bir süre sonra mı geçileceğini

Tablo 3.3.19: Geçiş ayarları.

Geçişte asıl ilke tutarlılıktır. Bir sunuda az sayıda geçiş türü kullanılır; çoğu zaman tek bir sade geçiş yeter. Her slayda başka bir geçiş verilirse izleyici her slayt değişiminde "bu sefer ne olacak" diye bekler ve dikkat içerikten kayar. Daha belirgin bir geçiş, yalnızca anlatımda bir dönüm noktasını, örneğin yeni bir bölümün başını işaretlemek için kullanılabilir.

> **Sık yapılan hata**
>
> Geçiş süresini uzun tutmak. İki saniyelik bir geçiş 30 slaytlık bir sunumda bir dakikayı yalnızca ekran değişimine harcar. Geçiş süresi kısa tutulur; izleyici geçişi fark etmeden içeriğe odaklanır.

3.3.7.3 Tasarım Önerileri, Dönüşüm ve Copilot

PowerPoint'in güncel sürümlerinde tasarım ve hareket işini kolaylaştıran üç araç vardır. Bu araçların varlığı ve adı PowerPoint sürümüne, lisansa ve güncelleme düzeyine göre değişebilir; sekme ve düğme adları kullanılan sürümde biraz farklı görünebilir.

- **Tasarım Önerileri** (*Designer*): Slayttaki içeriğe bakarak hazır düzen önerileri sunar; Tasarım sekmesindedir. Öneriler çevrim içi üretildiği için internet bağlantısı ve masaüstü sürümde Microsoft 365 aboneliği gerektirir.
- **Dönüşüm** (*Morph*): Geçişler sekmesindeki bir geçiş türüdür. Aynı nesne iki ardışık slaytta farklı konum, boyut ya da renkte durursa aradaki hareketi PowerPoint kendisi üretir.
- **Copilot**: Taslak slayt oluşturma, sunuyu özetleme gibi yapay zekâ yardımı sunar. Bireysel Microsoft 365 aboneliğinde aylık yapay zekâ kredisiyle, kurumsal hesapta ise ayrı bir Copilot lisansıyla çalışır.



Şekil 3.3.23: Dönüşüm geçişi. Aynı daire Slayt 1'de küçük ve solda, Slayt 2'de büyük ve sağdadır. Slayt 2'ye Dönüşüm verilince aradaki kareleri PowerPoint üretir; daire büyüyerek sağa kayar.

Tanım 3.3.18

Dönüşüm (*Morph*), iki ardışık slaytta ortak olan nesnelerin konum, boyut, dönüş ve renk farklarını bularak aradaki hareketi kendiliğinden canlandıran geçiştir. Geçiş ikinci slayda verilir.

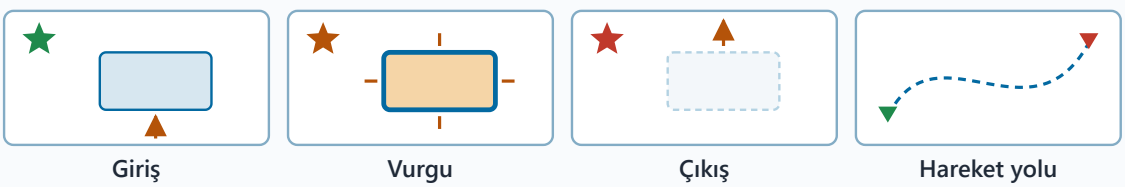
Dönüşümün kolay yolu şudur: birinci slayt çoğaltılır, kopyadaki nesneler yeni konumlarına taşınır ya da boyutlandırılır, ikinci slayda Dönüşüm geçişi verilir. Slayt çoğaltıldığı için iki slayttaki nesneler PowerPoint'e göre "aynı" nesnedir. Nesne ayrı ayrı çizilmiş olursa PowerPoint onları eşleyemeyebilir; bu durumda biri solarak kaybolur, öteki solarak belirir.

Sık yapılan hata

Tasarım Önerileri bölmesindeki bir düzeni düşünmeden kabul etmek. Öneri estetik açıdan iyi görünebilir, ama görsel hiyerarşiyi bozabilir: örneğin ana sayıyı küçültüp süs bir resmi büyütebilir. Öneri, slaydın tek mesajını öne çıkarıyorsa kullanılır.

3.3.7.4 Animasyon türleri

Animasyonlar dört türe ayrılır (Şekil 3.3.24). Animasyonlar sekmesindeki galeride her tür ayrı bir grupta listelenir ve simgelerinin rengi farklıdır.



Şekil 3.3.24: Animasyon türleri. Giriş nesneyi slayda getirir; vurgu zaten görünen nesneye dikkat çeker; çıkış nesneyi slayttan kaldırır; hareket yolu nesneyi kesikli çizgiyle gösterilen rota üzerinde taşır.

Tür ya da araç	Ne yapar?	Örnek
Giriş	Nesnenin slaytta görünmesini sağlar	Maddenin soldan uçarak gelmesi

Tür ya da araç	Ne yapar?	Örnek
Vurgu	Zaten görünen nesneye dikkat çeker	Bir sayının renginin değişmesi
Çıkış	Nesneyi slayttan kaldırır	Eski değerin solarak kaybolması
Hareket yolu	Nesneyi belirli bir rota üzerinde taşır	Bir paketin ağ şemasında ilerlemesi
Tetikleyici	Animasyonu belirli bir nesneye tıklanınca başlatır	Bir düğmeye tıklanınca cevabın görünmesi
Animasyon Bölmesi	Bütün efektlerin sırasını, başlangıcını, süresini ve gecikmesini gösterir ve yönetir	—

Tablo 3.3.20: Animasyon türleri ve animasyon araçları.

Bir nesneye birden fazla animasyon verilebilir. Bunun için Animasyonlar → Animasyon Ekle kullanılır. Galeriden doğrudan bir efekt seçmek, nesnenin var olan efektinin yerine yenisini koyar. Örneğin bir madde önce girişle gelip sonra vurgulanacaksa vurgu Animasyon Ekle ile eklenir.

Sık yapılan hata

Görünen bir nesneye dikkat çekmek için giriş animasyonu vermek. Giriş animasyonu olan nesne slayt açıldığında görünmez; ancak animasyonu çalışınca belirir. Zaten görünmesi gereken bir nesne için vurgu türü seçilir.

3.3.7.5 Başlangıç, süre ve gecikme

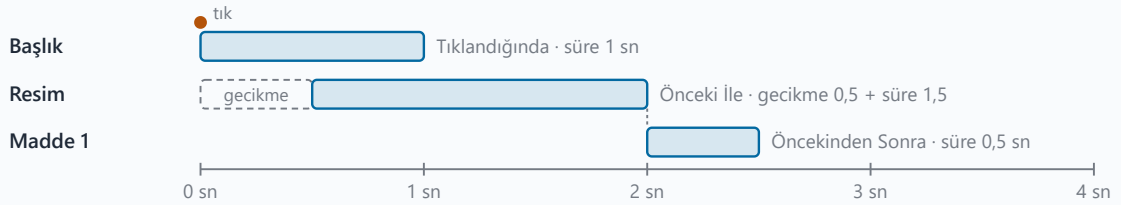
Her animasyonun üç zaman ayarı vardır: ne zaman başlayacağı, ne kadar süreceği ve başlamadan önce ne kadar bekleyeceği. Bu ayarlar Animasyonlar sekmesinin Zamanlama grubunda ya da Animasyon Bölmesi'ndedir.

Tanım 3.3.19

Animasyonun **başlangıcı** üç biçimde olabilir. **Tıklandığında**: sunucu tıklayınca ya da bir tuşa basınca başlar. **Önceki İle**: listedeki önceki efektle aynı anda başlar. **Öncekinden Sonra**: önceki efekt bitince kendiliğinden başlar. **Süre** efektin ne kadar sürdüğünü, **gecikme** ise başlangıç anından sonra efektin ne kadar bekleyeceğini belirler.

Bu ayarlardan bir efektin başlangıç ve bitiş anı basit bir kuralla hesaplanır:

- Başlangıç noktası: Tıklandığında için tıklama anı, Önceki İle için önceki efektin başlangıç noktası, Öncekinden Sonra için önceki efekt(ler)in bittiği an.
- Başlangıç = başlangıç noktası + gecikme.
- Bitiş = başlangıç + süre.



Şekil 3.3.25: Zaman çizelgesi. Başlık tıklamayla başlar ve 1 sn sürer. Resim Önceki İle, 0,5 sn gecikme ve 1,5 sn süreyle 0,5–2 sn arasında oynar. Madde 1 Öncekinden Sonra, 0,5 sn süreyle 2–2,5 sn arasında oynar.

Örnek 3.3.13

Şekil 3.3.25'teki üç efektin zamanlarını kurala göre hesaplayalım. Sunucu t = 0'da tıklıyor.

1. **Başlık** (Tıklandığında, gecikme 0, süre 1 sn): başlangıç noktası tıklama anı, yani 0. Başlangıç 0 + 0 = 0, bitiş 0 + 1 = 1 sn.
2. **Resim** (Önceki İle, gecikme 0,5, süre 1,5 sn): başlangıç noktası Başlık'ın başlangıç noktasıdır, yani 0. Başlangıç 0 + 0,5 = 0,5 sn, bitiş 0,5 + 1,5 = 2 sn.
3. **Madde 1** (Öncekinden Sonra, gecikme 0, süre 0,5 sn): önceki efektlerin en geç biteni Resim'dir (2 sn). Başlangıç 2 sn, bitiş 2 + 0,5 = 2,5 sn.

Üç efekt tek bir tıklamayla 2,5 saniyede tamamlanır. Başlık ile Resim 0,5 ile 1 sn arasında aynı anda oynar.

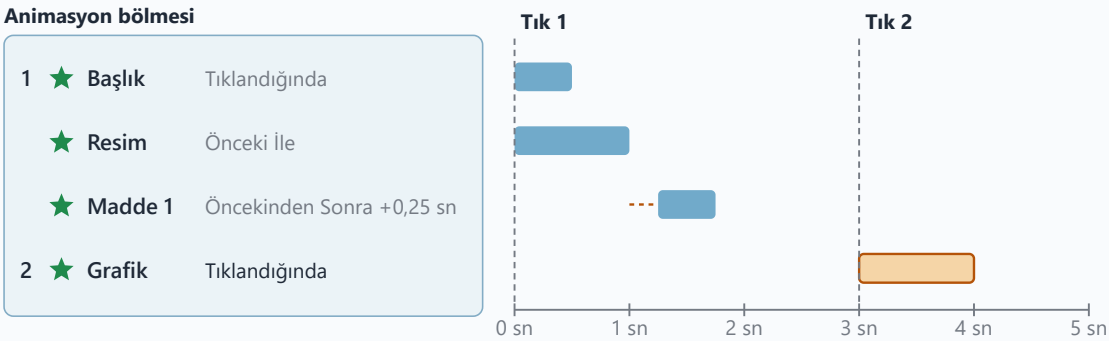
Animasyonlar her zaman gösteri görünümünde prova edilir. Düzenleme ekranında birbirinden bağımsız görünen efektler gösteride beklenmedik biçimde üst üste binebilir. Erişilebilirlik için de hızlı yanıp sönen ve gereksiz yere tekrar eden hareketlerden kaçınılır. Bu tür hareketler bazı izleyicilerde rahatsızlık ve dikkat dağınıklığı yaratır.

Sık yapılan hata

Gecikmeyi "önceki efektin bitişinden sonra bekleme" sanmak. Önceki İle başlayan bir efektin gecikmesi önceki efektin başlangıcından sayılır, bitişinden değil. Resim'in 0,5 sn gecikmesi Başlık bittikten değil, Başlık başladıktan 0,5 sn sonra dolar.

3.3.7.6 Animasyon zaman çizelgesi

Birden çok tıklamaya yayılan animasyonlarda Animasyon Bölmesi'ndeki numaralar önem kazanır. Bölmede bir efektin yanındaki numara, o efektin hangi tıklamayla başladığını gösterir. Önceki İle ve Öncekinden Sonra başlayan efektler kendi numarasını almaz; bağlı oldukları tıklamanın grubuna girer.



Şekil 3.3.26: Animasyon Bölmesi ve zaman çizelgesi. 1 numaralı tıklama Başlık, Resim ve Madde 1'i başlatır. Grafik 2 numaralı tıklamayla, sunucunun 3. saniyedeki tıklamasıyla 3–4 sn arasında oynar.

Örnek 3.3.14

Şekil 3.3.26'daki bölmede dört efekt var. Sunucu t = 0'da ve t = 3'te tıklıyor. Her efektin başlangıç ve bitiş anı nedir?

Efekt	Başlangıç türü	Gecikme	Süre	Başlangıç noktası	Oynadığı aralık
Başlık	Tıklandığında (1)	0	0,5 sn	Tık 1: 0	0–0,5 sn
Resim	Önceki İle	0	1 sn	Başlık'ın başlangıcı: 0	0–1 sn
Madde 1	Öncekinden Sonra	0,25 sn	0,5 sn	Önceki efektlerin bitişi: 1	1,25–1,75 sn
Grafik	Tıklandığında (2)	0	1 sn	Tık 2: 3	3–4 sn

Tablo 3.3.21: Dört efektin zaman çizelgesi.

1. Başlık ve Resim aynı anda başlar; Başlık 0,5'te, Resim 1'de biter.
2. Madde 1, önceki efektlerin en geç bittiği ana kadar bekler: Başlık 0,5'te bitse de Resim 1'de bittiği için başlangıç noktası 1'dir. Üzerine 0,25 sn gecikme eklenir: 1,25 sn'de başlar, 0,5 sn sürer, 1,75 sn'de biter.
3. Otomatik zincir 1,75 sn'de biter. Sonraki efekt Tıklandığında başladığı için PowerPoint sunucunun tıklamasını bekler. 1,75 ile 3 sn arasında ekranda hiçbir şey hareket etmez.
4. Sunucu t = 3'te tıklar; Grafik 3'te başlar, 1 sn sürer ve 4'te biter.

Bölmedeki numaralar yalnızca 1 ve 2'dir. Bu slaydın bütün animasyonları için iki tıklama yeter; üçüncü tıklama sonraki slayda geçer.

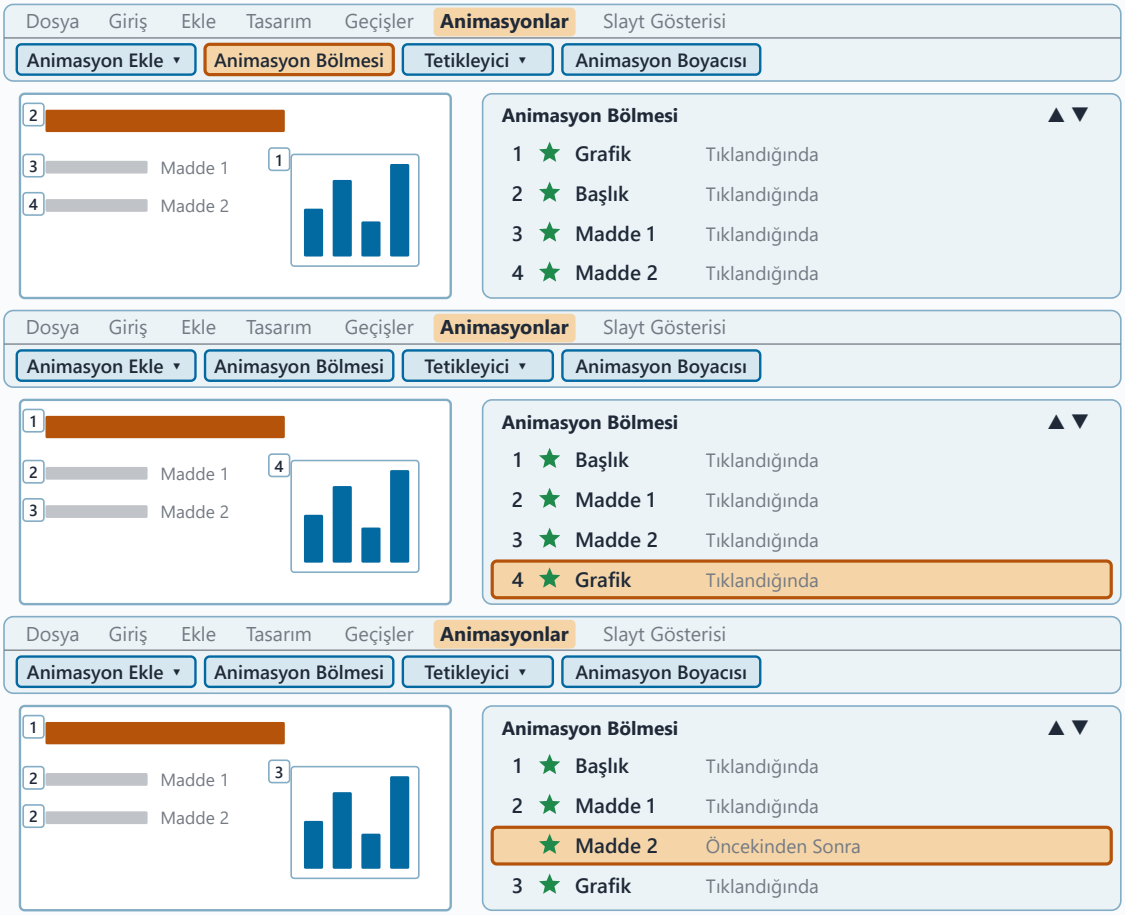
Sık yapılan hata

Öncekinden Sonra'yı "listede hemen üstteki efekt bitince" diye okumak ve yanlış efekte bakmak. Üstteki efekt, kendinden önce başlamış daha uzun bir efektten önce bitebilir. Madde 1 örneğinde hemen üstteki Resim'dir ve doğru sonuç verir; ama Resim 0,3 sn sürseydi Madde 1

yine de Başlık'ın 0,5'teki bitişini beklerdi.

3.3.7.7 Animasyon sırasını düzenleme

Efektler Animasyon Bölmesi'nde eklendikleri sırayla listelenir. Önce grafiğe, sonra başlığa animasyon verilmişse gösteride grafik başlıktan önce gelir. Sıra, bölmedeki ▲ ve ▼ düğmeleriyle ya da efekt sürüklenerek değiştirilir.



Şekil 3.3.27: Animasyon sırasını düzenleme. Üstte grafik yanlışlıkla ilk sıradadır. Ortada ▲ ve ▼ ile sıra Başlık, Madde 1, Madde 2, Grafik olmuştur. Altta Madde 2'nin başlangıcı Öncekinden Sonra yapılmış; Madde 2 kendi numarasını kaybetmiş ve Madde 1 ile aynı numarayı (2) almıştır.

Örnek 3.3.15

Adım	İşlem	Bölmedeki sıra (numara)	Tıklama sayısı
1	Başlangıç	Grafik (1), Başlık (2), Madde 1 (3), Madde 2 (4)	4

Adım	İşlem	Bölmedeki sıra (numara)	Tıklama sayısı
2	Başlık seçilip ▲	Başlık (1), Grafik (2), Madde 1 (3), Madde 2 (4)	4
3	Grafik seçilip iki kez ▼	Başlık (1), Madde 1 (2), Madde 2 (3), Grafik (4)	4
4	Madde 2'nin başlangıcı Öncekinden Sonra	Başlık (1), Madde 1 (2), Madde 2 (—), Grafik (3)	3

Tablo 3.3.22: Animasyon Bölmesi'nde sıra değişikliği ve tıklama sayısı.

İlk üç adım yalnızca sırayı değiştirir; her efekt kendi tıklamasıyla başladığı için tıklama sayısı 4'te kalır. Numaralar her adımda yeniden verilir. Dördüncü adımda Madde 2 ayrı bir tıklama almaz; Madde 1 bitince kendiliğinden gelir. Gösteride üç tıklama yeter: Tık 1 Başlık'ı, Tık 2 Madde 1'i ve ardından Madde 2'yi, Tık 3 Grafik'i getirir. Dördüncü tıklama sonraki slayda geçer.

Sık yapılan hata

Sırayı düzeltmek için animasyonları silip doğru sırayla yeniden eklemek. Bu, süre, gecikme ve efekt seçeneklerini de siler. Sıra Animasyon Bölmesi'nde ▲ ve ▼ ile değiştirilir; efektlerin ayarları korunur.

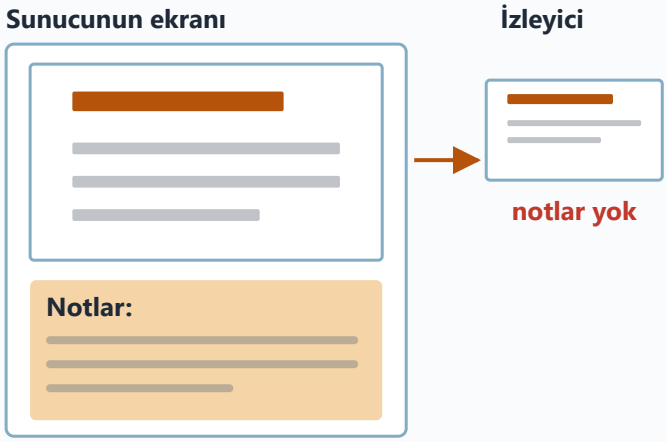
3.3.8 Sunum yapma

Sunu hazırlandıktan sonra asıl iş başlar: izleyicinin önünde konuşmak. PowerPoint bu aşamada konuşmacıya yalnızca slaytları göstermekle kalmaz. Notları, zamanlayıcıyı ve sıradaki slaydı yalnızca konuşmacının göreceği bir ekranda toplar, gösteriyi klavyeden yönetmeyi sağlar ve provayı ölçer.

3.3.8.1 Konuşmacı notları

Tanım 3.3.20

Konuşmacı notu (*speaker notes*), bir slayda bağlı olarak notlar bölmesine yazılan ve yalnızca konuşmacı için olan metindir. Normal gösteride izleyiciye görünmez; Sunucu Görünümü'nde konuşmacıya gösterilir ve Not Sayfaları olarak yazdırılabilir.



Şekil 3.3.28: Konuşmacının ekranında slaydın altında notlar bulunur. Projeksiyonda izleyici yalnızca slaydı görür.

Notlara dört tür bilgi yazılır:

- **İçerik:** Slaytta yer almayan anlatım ayrıntısı. Slayttaki "%38 artış" yazısının arkasındaki açıklama notta durur.
- **Hatırlatıcı:** Verilecek örnek, izleyiciye sorulacak soru ya da bir kaynak.
- **Süre:** Bu bölüme ayrılan zaman, örneğin "Yöntem bölümü burada bitmeli: 5. dakika".
- **Geçiş cümlesi:** Bir sonraki konuya bağlanan cümle. İki slayt arasında akışın kopmaması büyük ölçüde bu cümleye bağlıdır.

Notlar, slayttaki metni kısaltmanın da güvencesidir. Slayttan çıkarılan cümle kaybolmaz; konuşmacının gözünün önünde durur. Notların bir kâğıt kopyası da Dosya → Yazdır → Not Sayfaları ile alınabilir.

Sık yapılan hata

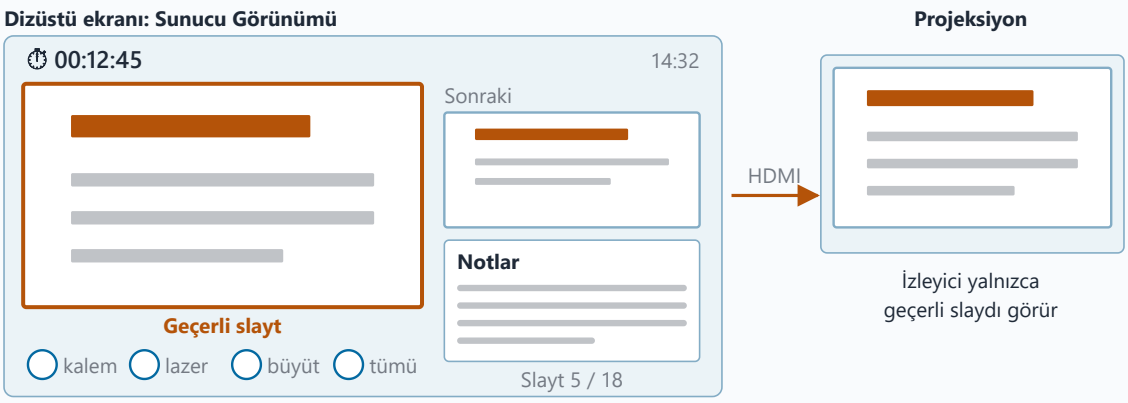
Notları cümle cümle okumak. Not bir senaryo değil, hatırlatıcıdır. Kısa anahtar sözcüklerle yazılan not bir bakışta okunur; uzun paragraf ise konuşmacıyı ekrana kilitler ve izleyiciyle göz temasını keser.

3.3.8.2 Sunucu Görünümü

Dizüstü bilgisayar bir projeksiyon cihazına ya da ikinci bir ekrana bağlanınca iki ekran oluşur. PowerPoint gösteri başlayınca bu iki ekrana farklı şeyler gönderir: projeksiyona yalnızca geçerli slaydı, dizüstü ekranına ise Sunucu Görünümü'nü.

Tanım 3.3.21

Sunucu Görünümü (*Presenter View*), yalnızca konuşmacının ekranında görünen gösteri görünümüdür. Geçerli slaydı, sıradaki slaydı, konuşmacı notlarını, geçen süreyi, saati ve gösteri araçlarını (kalem, lazer işaretçi, yakınlaştırma, bütün slaytlar) bir arada gösterir.



Şekil 3.3.29: Dizüstü ekranında Sunucu Görünümü: solda geçerli slayt ve altında araçlar, sağda sonraki slayt ve notlar, üstte zamanlayıcı (00:12:45) ve saat. HDMI ile bağlı projeksiyonda izleyici yalnızca geçerli slaydı görür.

Sunucu Görünümü'nün araçları sunumu akıcı tutar (Şekil 3.3.29):

- **Zamanlayıcı** gösteri başladığından beri geçen süreyi gösterir; duraklatılabilir ve sıfırlanabilir. Yanında gerçek saat de vardır.
- **Sonraki slayt** önizlemesi, konuşmacının bir sonraki konuya geçiş cümlesini hazırlamasını sağlar.
- **Kalem ve lazer işaretçi**, slaytta belirli bir alanı göstermek ya da üzerine kısa bir işaret çizmek için kullanılır.
- **Bütün slaytlar** düğmesi, küçük resimler arasından seçilen slayda atlar. İzleyici bu arama ekranını görmez; projeksiyonda geçerli slayt kalır.

Sunucu Görünümü ikinci ekran bağlıyken gösteri başlayınca kendiliğinden açılır. Alt + F5, gösteriyi doğrudan Sunucu Görünümü ile başlatır. Tek ekranda prova yaparken de bu kısayol kullanılabilir.

Sık yapılan hata

Ekranlar karışınca Sunucu Görünümü'nü projeksiyona yansıtmak. Bu durumda izleyici konuşmacının notlarını ve sıradaki slaydı görür. Sunucu Görünümü'nün üst kısmındaki ekran ayarlarından iki ekranın yeri değiştirilir. Salonda sunumdan önce bu ayar denenir.

3.3.8.3 Gösteri kısayolları

Gösteri sırasında fareyle menü açmak hem zaman alır hem izleyicinin dikkatini dağıtır. Gösteri, klavyeden birkaç tuşla yönetilir (Tablo 3.3.23).

İşlem	Tuş
Gösteriyi baştan başlat	F5
Gösteriyi geçerli slayttan başlat	Shift + F5
Sunucu Görünümü ile başlat	Alt + F5
Sonraki animasyon ya da slayt	→, Boşluk ya da Enter
Önceki animasyon ya da slayt	←
Ekranı karart / geri aç	B ya da . (nokta)
Ekranı beyaza çevir / geri aç	W ya da , (virgül)
Belirli bir slayda git	Slayt numarası + Enter
Kalem / lazer işaretçi	Ctrl + P / Ctrl + L
Gösteriyi bitir	Esc

Tablo 3.3.23: Slayt gösterisi kısayolları.

Örnek 3.3.16

12 slaytlık animasyonsuz bir sunu gösteride, ekranda 7. slayt var. Konuşmacı sırayla aşağıdaki tuşlara basıyor. Her tuştan sonra ekranda ne görünür?

Adım	Tuş	Ekran	Açıklama
1	W	Beyaz ekran	Dikkat konuşmacıya ya da tahtaya çekilir
2	W	Slayt 7	Kaldığı slayda döner
3	1, 1, Enter	Slayt 11	Numara ve Enter doğrudan o slayda atlar
4	←	Slayt 10	Önceki slayt
5	Esc	Normal görünüm, Slayt 10 seçili	Gösteri biter
6	Shift + F5	Slayt 10	Gösteri geçerli slayttan başlar
7	Esc, sonra F5	Slayt 1	F5 her zaman baştan başlatır

Tablo 3.3.24: Gösteri kısayollarının izlemesi.

Üçüncü adımda “1” ve “1” tuşlarına sırayla basılıp ardından Enter'a basılır; ekrana 11 sayısı yazılmaz ama PowerPoint numarayı toplar. Soru-cevap sırasında belirli bir slayda dönmenin en hızlı yolu budur; konuşmacının slayt numaralarını notlarında tutması bu yüzden işe yarar.

Sık yapılan hata

Gösterinin ortasında Esc'ye basıp yeniden F5 ile başlatmak. F5 gösteriyi 1. slayttan başlatır; konuşmacı kaldığı yere ulaşmak için bütün slaytları yeniden geçer. Kaldığı yerden devam etmek için Shift + F5 kullanılır.

3.3.8.4 Slayt boyutu, altbilgi, ses ve video

Sunumun teknik ayarlarından bazıları sunu hazırlanmadan önce verilmelidir. Bunların başında slayt boyutu gelir.

Tanım 3.3.22

Slayt boyutu (*slide size*), slaydın en-boy oranı ve fiziksel ölçüsüdür; Tasarım → Slayt Boyutu ile seçilir. Geniş ekran (16 : 9) 33,867 × 19,05 cm, standart (4 : 3) 25,4 × 19,05 cm'dir. Günümüz ekranları ve projeksiyon cihazlarının çoğu 16 : 9'dur; 4 : 3 eski projeksiyonlar içindir.

Örnek 3.3.17

16 : 9 olarak hazırlanmış bir sununun boyutu sonradan 4 : 3'e çevriliyor ve PowerPoint içeriği sığacak biçimde küçültüyor. 32 punto başlık kaç puntoya iner?

1. İki boyutun yüksekliği aynıdır (19,05 cm); genişlik 33,867 cm'den 25,4 cm'ye iner.
2. İçeriğin yeni genişliğe sığması için ölçek oranı 25,4 ÷ 33,867 ≈ 0,75'tir.
3. Bütün içerik bu oranla küçülür: 32 × 0,75 = 24 punto. Resimler ve şekiller de dörtte bir oranında küçülür; slaydın üstünde ve altında boş bantlar kalır.

Öteki seçenek, içeriği küçültmeden yeni boyuta yerleştirmektir; bu durumda sağdaki ve soldaki nesneler slayttan taşabilir. İki seçenekte de yerleşim bozulur ve her slaydın elden geçirilmesi gerekir. Bu yüzden slayt boyutu sunuya başlamadan seçilir.

Öteki teknik ayarlar şunlardır:

- **Altbilgi ve slayt numarası:** Ekle → Üstbilgi ve Altbilgi ile slayt numarası, tarih ve altbilgi metni eklenir; Tümüne Uygula ile bütün slaytlara yayılır. Slayt numarası soru-cevapta izleyicinin “12. slayttaki grafik” diye bir slayda işaret edebilmesini sağlar.
- **Ses ve video:** Ekle → Video ya da Ekle → Ses ile eklenir. Eklenen ortam seçilince açılan Kayıttan Yürüt sekmesinden başlatma biçimi (tıklayınca ya da kendiliğinden), kırpma ve ses düzeyi ayarlanır.

- **Yazı tipi gömme:** Dosya → Seçenekler → Kaydet → Dosyaya yazı tiplerini ekle. Sunuda kullanılan yazı tipleri dosyanın içine konur; sunu o yazı tipinin kurulu olmadığı bir bilgisayarda açıldığında yazılar bozulmaz. Bazı yazı tiplerinin lisansı gömülmeye izin vermez.
- **Çoğaltma:** Ctrl + D seçili slaydı ya da nesneyi hemen yanında kopyalar.
- **Kalem ve lazer:** Gösteri sırasında Ctrl + P kalemi, Ctrl + L lazer işaretçiyi açar.

Sık yapılan hata

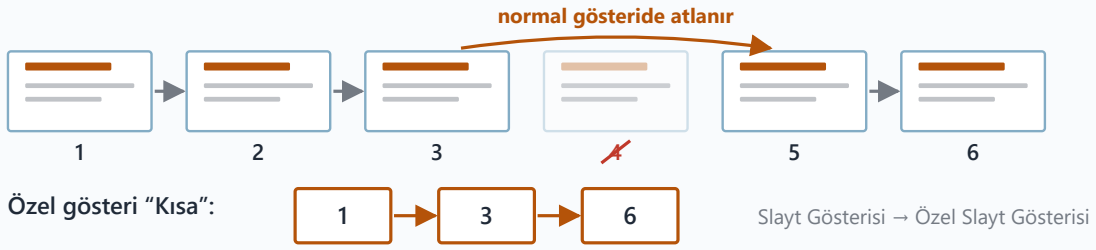
Sunuyu hazırlayan bilgisayarda düzgün görünen bir yazı tipine güvenmek. Salondaki bilgisayarda o yazı tipi yoksa PowerPoint yerine başka bir yazı tipi koyar; satırlar uzar, metin kutulardan taşar. Yazı tipi gömülür ya da sunumdan önce salondaki bilgisayarda denenir.

3.3.8.5 Gizli slayt ve özel gösteri

Aynı sunu farklı izleyicilere farklı uzunluklarda sunulabilir. Örneğin bir proje jüriye 20 dakikada, bölüm tanıtım gününde 5 dakikada anlatılır. Her biri için ayrı dosya tutmak yerine iki araç kullanılır.

Tanım 3.3.23

Gizli slayt (*hidden slide*), dosyada kalan ama normal gösteride atlanan slayttır. **Özel gösteri** (*custom show*), aynı sunudaki slaytlardan seçilip adlandırılmış ayrı bir gösteri dizisidir; Slayt Gösterisi → Özel Slayt Gösterisi ile oluşturulur ve oynatılır.



Şekil 3.3.30: Altı slaytlık sunuda 4. slayt gizlidir ve normal gösteride atlanır. “Kısa” adlı özel gösteri yalnızca 1, 3 ve 6. slaytları içerir.

Örnek 3.3.18

Şekil 3.3.30'daki sunu için üç gösteriyi karşılaştıralım.

1. **F5 ile normal gösteri:** 1, 2, 3, 5, 6. Gizli 4. slayt atlanır; beş slayt gösterilir.
2. **“Kısa” özel gösterisi:** 1, 3, 6. Özel gösteri kendi listesini kullanır; listede olmayan 2 ve 5 de gösterilmez.
3. **Soru gelirse:** Gösteri sırasında 4 yazılıp Enter'a basılır. Gizli slayt bu yolla gösterilebilir.

Gizli slayt soru-cevap için yedek ayrıntıyı hazır tutar. Özel gösteri ise aynı dosyadan farklı uzunlukta sunumlar üretir; bir slayt düzeltilince bütün özel gösterilerde düzeltilmiş olur.

Sık yapılan hata

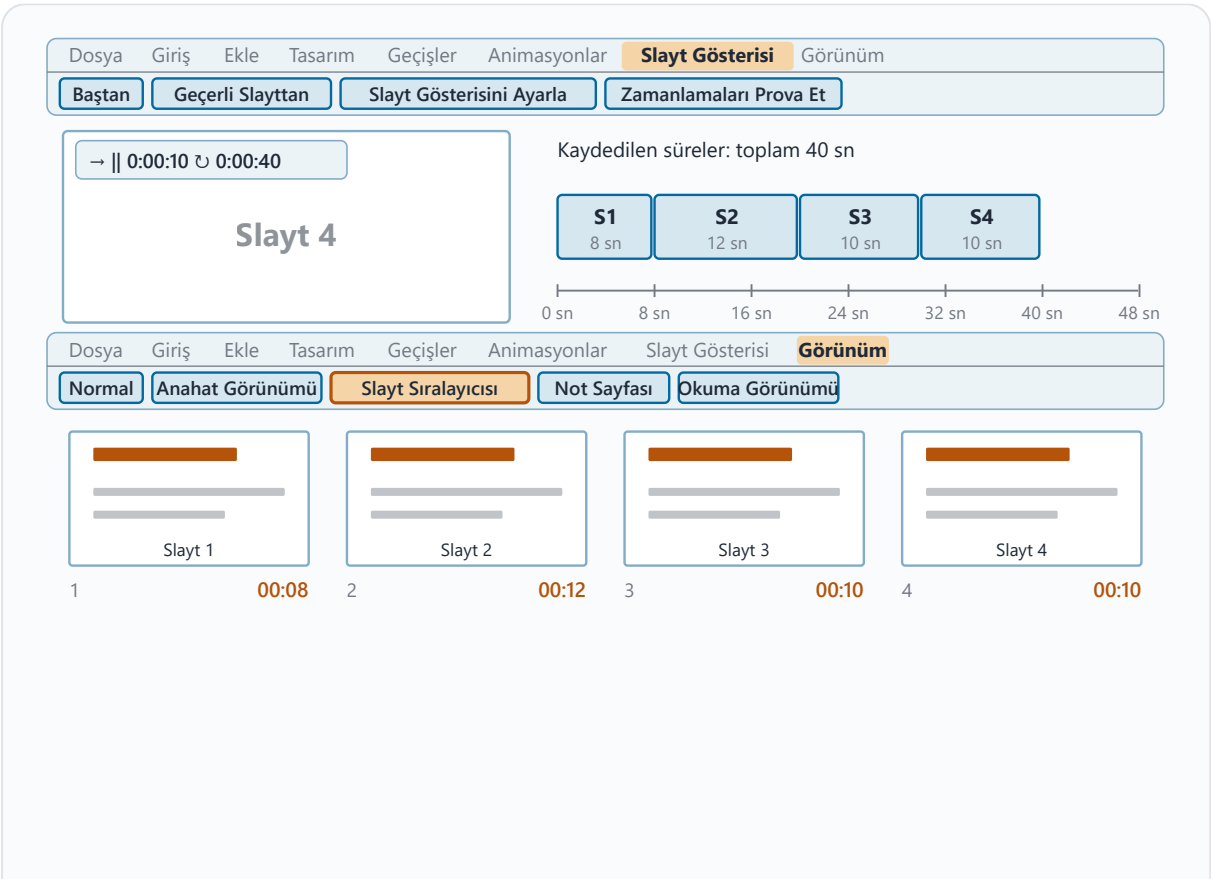
Hassas bir içeriği gizleyip dosyayı paylaşmak. Gizli slayt dosyada durur; dosyayı açan herkes onu slayt bölmesinde görür. Hassas içerik paylaşılmadan önce silinir.

3.3.8.6 Zamanlama provası ve otomatik gösteri

Gösteri her zaman konuşmacının tıklamasıyla ilerlemez. Bir bilgi ekranında ya da sesli bir kayıtta slaytların belirli sürelerde kendiliğinden ilerlemesi gerekir. Bu sürelerin en kolay belirlenme yolu, sunuyu bir kez gerçek hızında prova etmektir.

Tanım 3.3.24

Zamanlamaları Prova Et, gösteriyi başlatıp her slaytta geçen süreyi ölçen ve kaydeden Slayt Gösterisi komutudur. Kaydedilen süreler her slaydın geçiş ayarına yazılır. **Bilgi noktası** (*kiosk*) türündeki gösteri ise tam ekran açılır, tıklamayla ilerlemez ve Esc'ye basılana kadar döngüde kalır.





Şekil 3.3.31: Zamanlama provası. Üstte dört slayt için kaydedilen süreler: 8, 12, 10 ve 10 sn, toplam 40 sn. Ortada süreler Slayt Sıralayıcısı'nda küçük resimlerin altında görünür. Altta bilgi noktası gösterisi son slayttan sonra başa döner; t = 46 sn'de ekranda yine Slayt 1 vardır.

Adım	İşlem	Sonuç
1	Slayt Gösterisi → Zamanlamaları Prova Et	Gösteri başlar; köşedeki araç çubuğu slayt süresini ve toplam süreyi sayar
2	Slayt 1 gerçek hızında anlatılıp ilerlenir	Slayt 1 için 8 sn kaydedilir
3	Öteki slaytlar anlatılır	12, 10, 10 sn; toplam 8 + 12 + 10 + 10 = 40 sn
4	Prova biter, zamanlamalar kaydedilir	Her slaydın Geçişler sekmesindeki “Sonra” kutusuna kendi süresi yazılır
5	Slayt Gösterisini Ayarla → bilgi noktası türü	Sürekli döngü kendiliğinden işaretlenir; gösteri tıklamayla ilerlemez

Tablo 3.3.25: Zamanlama provası ve otomatik gösteri adımları.

Örnek 3.3.19

Dört slaytlık sunu bilgi noktası gösterisi olarak dönüyor (süreler 8, 12, 10, 10 sn). Gösteri başladıktan t = 46 sn ve t = 75 sn sonra ekranda hangi slayt vardır?

1. Bir turun süresi 8 + 12 + 10 + 10 = 40 sn'dir. Slaytların tur içindeki aralıkları: Slayt 1 0–8, Slayt 2 8–20, Slayt 3 20–30, Slayt 4 30–40 sn.
2. t = 46: tam turlar atılır, kalan bulunur: 46 − 40 = 6. Altıncı saniye Slayt 1'in aralığındadır (0–8). Ekranda ikinci turun Slayt 1'i vardır.
3. t = 75: 75 − 40 = 35. 35 saniye Slayt 4'ün aralığındadır (30–40). Ekranda Slayt 4 vardır; bu slaytta 35 − 30 = 5 sn geçmiştir.

Genel kural: t'nin tur süresine bölümünden kalan bulunur ve hangi slaydın aralığına düştüğüne bakılır. Örneğin 10 dakikada (600 sn) gösteri 600 ÷ 40 = 15 tur atar.

Bilgi noktası gösterisinde tıklama işe yaramadığı için zamanlama şarttır. Zamanlamasız bir bilgi noktası gösterisi ilk slaytta takılı kalır. İzleyicinin ekrana dokunarak gezinmesi isteniyorsa yalnızca köprüler ve eylem düğmeleri çalışır.

Sık yapılan hata

Prova sırasında kaydedilen zamanlamaları canlı sunumda unutmak. Kaydedilen süreler konuşmacının tıklamasını beklemeden slaytları ilerletir; konuşmacı daha uzun konuşursa slayt onu beklemeden değişir. Canlı sunumdan önce Slayt Gösterisini Ayarla'da ilerlemenin elle yapılacağı seçilir.

3.3.8.7 Etkili sunum yapma

En iyi hazırlanmış sunu bile konuşmacının tutumuyla güçlenir ya da zayıflar. Sunumun kendisi için altı ilke vardır:

- Sunum, gerçek ekran ve bağlantılarla önceden prova edilir. Projeksiyon cihazının çözünürlüğü, renkleri ve bağlantı kablosu sunum gününden önce denenir.
- Slayt okunmaz; izleyiciyle göz teması ve iletişim kurulur. Konuşmacı slayda dönüp okuyorsa izleyici de okur ve konuşmacıyı dinlemez.
- Her bölüm açıkça ana mesaja bağlanır. Bölüm sonunda “bu sonuç ana sorumuza şunu söylüyor” gibi bir bağlantı cümlesi kurulur.
- İşaretçi ve animasyon yalnızca dikkati yönlendirmek için kullanılır. Lazer işaretçiyle slaytta daireler çizmek dikkati dağıtır.
- Süreye uyulur; soru-cevap için pay bırakılır. Süre aşılıyorsa son slaytlar hızlandırılmaz, önceden belirlenmiş bir ayrıntı atlanır.
- PDF yedeği, ekran adaptörü ve çevrim dışı kopya gibi alternatifler hazır tutulur. Bilgisayar, yazılım ya da internet bağlantısı sorun çıkarsa sunum PDF'ten sürdürülür.

Sık yapılan hata

Süre azalınca slaytları hızla geçip “bunlara zaman kalmadı” demek. İzleyici atlanan slaytları önemli sanır ve kaçırdığını düşünür. Zaman yetmezse sona gelinmeden kapanış slaydına atlanır ve ana mesaj söylenir.

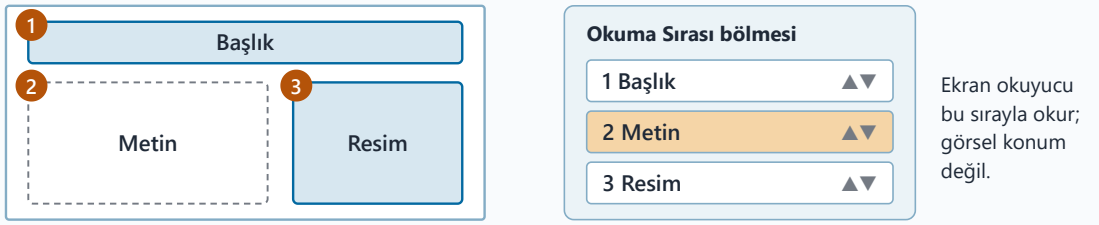
3.3.9 Kalite ve paylaşım

Sunu bittiğinde iş bitmiş sayılmaz. Paylaşmadan önce sununun herkesçe okunabildiği, istenmeyen bilgiler taşımadığı ve başka bir bilgisayarda doğru çalıştığı denetlenir.

3.3.9.1 Erişilebilir sunu

Tanım 3.3.25

Erişilebilir sunu, görme, işitme ya da renk algısı farklı olan izleyicilerin ve ekran okuyucu kullananların da içeriği eksiksiz alabildiği sunudur. **Alternatif metin** (*alt text*), bir görselin ne anlattığını ekran okuyucuya söyleyen kısa açıklamadır.



Şekil 3.3.32: Okuma sırası. Slayttaki Başlık, Metin ve Resim'i ekran okuyucunun hangi sırayla okuyacağı Okuma Sırası bölmesinde düzenlenir. Ekran okuyucu nesneleri görsel konumlarına göre değil, bu listedeki sıraya göre okur.

Erişilebilir bir sunuda altı noktaya bakılır:

- **Okuma sırası:** Ekran okuyucu nesneleri ekleniş sırasıyla okur; bu sıra görsel sıradan farklı olabilir (Şekil 3.3.32). Sıra Okuma Sırası bölmesinde düzeltilir.
- **Alternatif metin:** Anlamlı her görsel kısaca açıklanır. “Resim” değil, “2023'ten 2024'e kayıtların %38 arttığını gösteren sütun grafik” yazılır. Yalnızca süs olan görseller “dekoratif” olarak işaretlenir.
- **Başlık:** Her slaydın benzersiz ve açıklayıcı bir başlığı olur. Ekran okuyucu kullanan izleyici slaytlar arasında başlıklara bakarak gezinir.
- **Kontrast:** Metin ile arka plan arasında yeterli ayrım bulunur.
- **Renk:** Bilgi yalnızca renk farkıyla verilmez; renge bir etiket ya da simge eşlik eder.
- **Altyazı:** Video, ses ya da canlı anlatım için uygun olduğunda altyazı kullanılır.

Kontrast ölçülebilir bir büyüklüktür. Web erişilebilirlik yönergeleri (WCAG), metin rengi ile arka plan rengi arasındaki oranın normal boyutlu yazıda en az 4,5 : 1 olmasını önerir. Oran, iki rengin göreli parlaklığından hesaplanır: (açık rengin parlaklığı + 0,05) ÷ (koyu rengin parlaklığı + 0,05). Parlaklık beyazda 1, siyahta 0'dır.

Örnek 3.3.20

Beyaz arka plan üzerinde iki metin rengi deneniyor: siyah ve RGB değerleri (119, 119, 119) olan orta gri. Kontrast oranları yeterli mi?

1. Siyah üzerinde beyaz: (1 + 0,05) ÷ (0 + 0,05) = 1,05 ÷ 0,05 = 21. Oran 21 : 1'dir; olabilecek en yüksek değerdir.
2. Gri için önce her kanal 0–1 arasına getirilir: 119 ÷ 255 ≈ 0,467. Ekran renkleri doğrusal olmadığı için bu değer $((0{,}467 + 0{,}055) \div 1{,}055)^{2{,}4} \approx 0{,}184$ parlaklığına çevrilir. Üç kanal eşit olduğundan grinin göreli parlaklığı da 0,184'tür.
3. Oran: (1 + 0,05) ÷ (0,184 + 0,05) ≈ 1,05 ÷ 0,234 ≈ 4,48. Oran 4,5'in hemen altında kalır; yönergeye göre bu gri normal metin için yetersizdir.

Bir ton koyu gri (118, 118, 118) yaklaşık 4,54 : 1 verir ve sınırı geçer. Projeksiyon cihazları renkleri ekrandan daha soluk gösterdiği için sunumda sınırda kalan renkler yerine belirgin biçimde daha koyu renkler seçilir.

Sık yapılan hata

Bir grafikte başarılı grubu yeşil, başarısız grubu kırmızıyla gösterip başka hiçbir ayrım koymamak. Kırmızı-yeşil renk körlüğü olan izleyici iki grubu ayırt edemez; ekran okuyucu da rengi okumaz. Renge bir etiket (“başarılı”, “başarısız”) ya da farklı bir desen eklenir.

3.3.9.2 Paylaşmadan önce denetleyiciler

PowerPoint, paylaşmadan önce sunuyu tarayan dört denetleyici sunar (Tablo 3.3.26).

Araç	Yeri	Neyi bulur?
Erişilebilirliği Denetle	Gözden Geçir	Alternatif metni olmayan görseller, başlıksız slaytlar, düşük kontrast, karışık okuma sırası
Belgeyi İncele	Dosya → Bilgi → Sorunları Denetle	Açıklamalar, yazar gibi kişisel bilgiler, konuşmacı notları, slayt dışında kalan nesneler
Uyumluluğu Denetle	Dosya → Bilgi → Sorunları Denetle	Eski PowerPoint sürümlerinde çalışmayacak özellikler
Yazım Denetimi	Gözden Geçir	Dil ayarına göre yazım hataları

Tablo 3.3.26: Paylaşmadan önce kullanılan denetleyiciler.

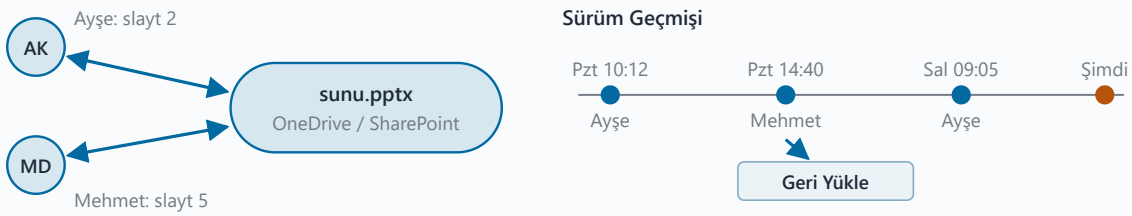
Denetleyiciler sorunu bulur, ama düzeltmez. Erişilebilirlik denetleyicisi örneğin “Resim 3'te alternatif metin yok” der ve nedenini açıklar. Alternatif metni yazmak ya da okuma sırasını düzeltmek sununun sahibinin işidir. Belgeyi İncele ise bulduklarını kaldırabilir; ama kaldırılan notlar ve açıklamalar geri getirilemez. Bu yüzden Belgeyi İncele, sununun bir kopyası üzerinde çalıştırılır.

Sık yapılan hata

Belgeyi İncele'yi asıl dosyada çalıştırıp konuşmacı notlarını silmek. Notlar bu işlemle geri dönmeyecek biçimde kaybolur. Önce Farklı Kaydet ile paylaşılacak bir kopya alınır, inceleme o kopyada yapılır.

3.3.9.3 İş birliği ve gözden geçirme

Sunular çoğu zaman bir ekipçe hazırlanır. Sunu OneDrive ya da SharePoint gibi desteklenen bir bulut konumunda saklanırsa birden çok kişi aynı dosya üzerinde aynı anda çalışabilir (Şekil 3.3.33).



Şekil 3.3.33: Ortak yazarlık. Ayşe slayt 2'de, Mehmet slayt 5'te buluttaki aynı sunu üzerinde aynı anda çalışır. Sürüm Geçmişi önceki kayıtları listeler ve seçilen sürüm Geri Yükle ile geri getirilir.

Araç	Ne yapar?
Açıklamalar	Bir slayda ya da nesneye geri bildirim yazılır; açıklamalar yanıtlanabilir ve çözüldü olarak işaretlenebilir
Ortak yazarlık	Bulut konumundaki dosyada eş zamanlı düzenleme; kimin hangi slaytta olduğu görünür
Sürüm Geçmişi	Önceki kayıtlı sürümlere erişim ve geri yükleme
Karşılaştır	Sununun iki ayrı kopyasındaki farkları gösterir
Yazım denetimi	Dil ayarına göre olası yazım hatalarını gösterir
Son kontrol	Açıklama, gizli slayt ve kişisel bilgiler paylaşım öncesi incelenir

Tablo 3.3.27: İş birliği ve gözden geçirme araçları.

Sık yapılan hata

Ekip üyelerinin sununun kopyalarını e-postayla dolaştırıp her birinde ayrı değişiklik yapması. Sonunda “sunu_son_v3_gercekson.pptx” gibi birbirinden farklı dosyalar ortaya çıkar ve değişiklikleri birleştirmek saatler alır. Tek bir bulut dosyasında ortak yazarlık bu sorunu baştan önler.

3.3.9.4 Kayıt ve dışa aktarma

Sunu yalnızca canlı anlatılmaz; kaydedilip video olarak paylaşılabilir ya da sabit bir belge olarak dağıtılabilir.

- **Sunuyu kaydet:** Slayt süreleri, sesli anlatım, kamera görüntüsü ve işaretçi hareketleri slayt slayt kaydedilir.
- **Video dışa aktar:** Dosya → Dışa Aktar → Video Oluştur, kayıtlı anlatımı ve zamanlamaları izlenebilir bir video dosyasına (.mp4) dönüştürür. Zamanlaması olmayan slaytlar için varsayılan bir süre kullanılır.
- **PDF:** Sabit düzenli paylaşım ya da baskı sağlar; animasyonları ve geçişleri içermez.
- **Dinleyici notları:** Bir sayfaya birden fazla slayt basılır; izleyiciye dağıtılabilir.
- **Paketleme:** Sunu başka bilgisayara taşınırken bağlı medya ve yazı tipleri de denetlenir.
- **Dosya boyutu:** Görseller ve videolar uygun biçimde sıkıştırılarak dosya küçültülür.



Şekil 3.3.34: Dinleyici notları çıktısı. 3 Slayt düzeninde her slaydın yanında not satırları vardır; 6 Slayt düzeninde sayfaya daha çok slayt sığar. Sağda Yazdır ayarlarındaki yazdırma düzenleri listelenir.

Sık yapılan hata

Videoyu bağlantı olarak ekleyip sunu dosyasını tek başına başka bilgisayara kopyalamak. Bağlı video, sunu dosyasının içinde değildir; yeni bilgisayarda bulunamaz ve gösteride açılmaz. Video sunuya gömülür ya da sunu medyasıyla birlikte paketlenir.

3.3.9.5 Yazdırma seçenekleri

Dosya → Yazdır ekranında “Tam Sayfa Slaytlar” yazan menü, sayfaya neyin basılacağını belirler (Tablo 3.3.28).

Yazdırma düzeni	Sayfada ne var?	Kim için?
Tam Sayfa Slaytlar	Her sayfada bir slayt	Slaytların kendisi
Not Sayfaları	Üstte slayt, altta konuşmacı notu	Konuşmacı
Anahat	Yalnızca başlık ve madde metni	Metni gözden geçiren

Yazdırma düzeni	Sayfada ne var?	Kim için?
Dinleyici Notları	Sayfa başına 1, 2, 3, 4, 6 ya da 9 slayt; 3'lü düzende yanında not satırları	İzleyici

Tablo 3.3.28: Yazdırma düzenleri.

Örnek 3.3.21

30 slaytlık bir sunu yazdırılacak. Farklı düzenlerde kaç sayfa tutar?

Düzen	Hesap	Sayfa	Çift taraflı yaprak
Tam sayfa	30 ÷ 1	30	15
3'lü dinleyici notu	30 ÷ 3	10	5
6'lı dinleyici notu	30 ÷ 6	5	3
9'lu dinleyici notu	30 ÷ 9 ≈ 3,3 → 4	4	2

Tablo 3.3.29: 30 slaytlık sununun sayfa ve yaprak sayısı.

1. Sayfa sayısı, slayt sayısının sayfa başına slayda bölümünün yukarı yuvarlanmasıdır. 30 ÷ 9 ≈ 3,3 çıkar; son sayfa eksik dolu da olsa basılacağı için 4 sayfa gerekir.
2. Çift taraflı baskıda her yaprağa iki sayfa düşer; sayfa sayısı ikiye bölünüp yukarı yuvarlanır. 6'lı düzende 5 ÷ 2 = 2,5 → 3 yaprak.

9'lu düzen kâğıt açısından en tasarruflusudur. Ama A4 sayfaya dokuz slayt sığınca her slayt kabaca bir kibrit kutusu boyutuna iner ve küçük yazılar okunamaz. İzleyiciye dağıtılacak çıktıda çoğu zaman 3'lü ya da 6'lı düzen seçilir.

3.3.9.6 Son kontrol ve sık yapılan hatalar

Paylaşmadan ya da sunum gününden önce sunu altı başlık altında son kez gözden geçirilir (Tablo 3.3.30).

Başlık	Denetlenecekler
İçerik	Amaç, doğruluk, kaynak ve anlatı akışı
Tasarım	Başlık, yazı boyutu, kontrast, hizalama ve boşluk
Teknik	Yazı tipleri, medya, bağlantılar ve animasyonlar
Erişilebilirlik	Başlık, okuma sırası, alternatif metin ve altyazı
Sunum	Notlar, zamanlama, gösteri ve ekran ayarları
Paylaşım	Dosya adı, biçim, gizli slayt ve kişisel bilgiler

Tablo 3.3.30: Son kontrol listesi.

Sunumlarda en sık görülen hatalar ve çözümleri Tablo 3.3.31'de toplanmıştır.

Sık yapılan hata	Sonuç	Çözüm
Konuşmanın tamamını slayda yazmak	İzleyici okur, dinlemez	Ana nokta slaytta, ayrıntı notlarda
Her slaytta başka yazı tipi ve renk	Dağınık, özensiz görünüm	Tema ve Asıl Slayt
Her nesneye farklı animasyon	Dikkat içerikten harekete kayar	Az sayıda, amaçlı efekt
Küçük yazı ve soluk renk	Arka sıralar okuyamaz	Büyük punto, yüksek kontrast
Slayt boyutunu en sonda değiştirmek	Nesneler ölçeklenir, yerleşim bozulur	16 : 9 ya da 4 : 3 baştan seçilir
Bağlı videolu dosyayı yalnız kopyalamak	Gösteride video açılmaz	Gömülü medya ya da paketleme
Bilgiyi yalnızca renkle vermek	Renk ayırt edemeyen izleyici kaçırır	Renk + etiket ya da simge

Tablo 3.3.31: Sık yapılan sunum hataları.

3.3.10 Özet

- **Sunu** dosyadır, **sunum** anlatımdır. Slayt konuşmayı destekler, onun yerini almaz. Çalışma alanı şerit, slayt bölmesi, slayt alanı, notlar bölmesi ve durum çubuğundan oluşur; **bağlamsal sekmeler** nesne seçiliyken görünür.
- Görünümler içeriği değiştirmez: Normal düzenleme, **Anahat** metin kurma, **Slayt Sıralayıcısı** akışı görme ve sıralama, **Asıl Slayt** ortak tasarım içindir.
- Plan amaç, hedef kitle ve **ana mesajla** başlar. Bir içerik slaydı yaklaşık 1–1,5 dakika taşır; soru-cevap süresi plana baştan yazılır.
- Anahatta en üst düzeydeki her satır bir slayttır; Tab düzeyi düşürür, Shift + Tab yükseltir.
- .pptx düzenlenebilir, .ppsx doğrudan gösteri, .potx şablon, .pdf sabit ve animasyonsuz, .mp4 videodur. Farklı Kaydet yeni dosya oluşturur.
- **Düzen**, yer tutucuların türünü ve konumunu belirler. Gizli slayt dosyada kalır, gösteride atlanır. **Bölümler** slaytları gruplar ve grup grup taşır; gösteride görünmez.
- **Tema** renk ve yazı tipini, **Asıl Slayt** bunların yerleşimini bütün düzenlere ve slaytlara aktarır. Elle biçim yalnızca değiştirilen özelliği kilitler ve asıl slayttan önce gelir; Sıfırla bu kilidi kaldırır.
- Görsel hiyerarşide en önemli öğe en büyük, kaynak en soluk olur; başlık sonucu söyler. Slayt metni kısa tutulur, ayrıntı notlara taşınır.
- Resim kırpılırken oran korunur. Ekranın gösterebileceğinden fazla piksel yalnızca dosyayı büyütür; Resimleri Sıkıştır hedef ppi'ye indirir ve kırpılan alanları silebilir.
- Grafik mesaja göre seçilir: karşılaştırma için sütun, zaman için çizgi, az kategorili parça-bütün için pasta, ilişki için dağılım. **Gömülü** grafik dosyayla taşınır, **bağlı** grafik Excel'den güncellenir.
- **Hizala** kenarları ortak çizgiye getirir; **Dağıt** dıştaki iki nesneyi yerinde bırakıp boşluğu eşitler: aralık = (toplam aralık − genişlikler toplamı) ÷ (nesne sayısı − 1).
- **Geçiş** slaytlar arasında, **animasyon** slayt içindeki nesnelerde çalışır. Animasyon türleri giriş, vurgu, çıkış ve hareket yoludur.

- Başlangıç = başlangıç noktası + gecikme, bitiş = başlangıç + süre. Tıklandığında tıklamayı, Önceki İle önceki efektin başlangıcını, Öncekinden Sonra önceki efektlerin bitişini başlangıç noktası alır. Bölmedeki numaralar tıklama sayısını verir.
- Konuşmacı notları izleyiciye görünmez; **Sunucu Görünümü** notları, sonraki slaydı ve zamanlayıcıyı konuşmacıya gösterir. F5 baştan, Shift + F5 geçerli slayttan başlatır; B/W ekranı karartır ya da beyazlatır.
- Slayt boyutu baştan seçilir; 16 : 9'dan 4 : 3'e geçişte içerik yaklaşık 0,75 oranında küçülür. Yazı tipleri dosyaya eklenir.
- **Zamanlamaları Prova Et** slayt sürelerini kaydeder; bilgi noktası gösterisi döngüde kalır, t anındaki slayt tur süresine bölümden kalanla bulunur.
- Erişilebilirlik için başlık, okuma sırası, alternatif metin, yeterli kontrast (normal metinde en az 4,5 : 1) ve renk dışı ipucu gerekir. Denetleyiciler sorunu bulur, düzeltmeyi kullanıcı yapar; Belgeyi İncele bir kopyada çalıştırılır.
- Yazdırmada sayfa sayısı slayt sayısının sayfa başına slayda bölümünün yukarı yuvarlanmasıdır; çift taraflı baskıda yaprak sayısı bunun yarısının yukarı yuvarlanmasıdır.

Hesap	Formül	Örnek
Slayt başına süre	sunum süresi ÷ içerik slaydı sayısı	10 dk ÷ 9 ≈ 1,1 dk
Ham resim boyutu	genişlik × yükseklik × 3 bayt	4000 × 3000 → 36 MB
Hedef piksel	slayt ölçüsü (inç) × ppi	13,333 × 150 ≈ 2000
Dağıt aralığı	(dış kenarlar arası − genişlikler) ÷ (n − 1)	(520 − 330) ÷ 2 = 95
Animasyon zamanı	başlangıç noktası + gecikme; + süre	1 + 0,25 = 1,25; 1,25 + 0,5 = 1,75 sn
Döngüdeki slayt	t'nin tur süresine bölümünden kalan	46 − 40 = 6 → Slayt 1
Kontrast oranı	$(L_{açık} + 0{,}05) \div (L_{koyu} + 0{,}05)$	1,05 ÷ 0,234 ≈ 4,48
Yazdırma sayfası	⌈slayt ÷ sayfa başına slayt⌉	⌈30 ÷ 9⌉ = 4

Tablo 3.3.32: Notta kullanılan başlıca hesaplar.

3.3.11 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

slayt *slide* · sunu *presentation (file)* · sunum *presentation* · şerit *ribbon* · sekme *tab* · slayt bölmesi / notlar bölmesi *slide pane / notes pane* · anahat *outline* · Slayt Sıralayıcısı *slide sorter* · şablon *template* · yer tutucu *placeholder* · slayt düzeni *slide layout* · gizli slayt *hidden slide* · Özet / Bölüm / Slayt Yakınlaştırma *Summary / Section / Slide Zoom* · köprü (bağlantı) *hyperlink* · tema *theme* · Asıl Slayt *slide master* · kontrast *contrast* · piksel *pixel* · çözünürlük *resolution* · sıkıştırma *compress (media)* · SmartArt *SmartArt* · bağlı / gömülü nesne *linked / embedded object*

hizala / dağıt *align / distribute* · grup *group* · geçiş *transition* · animasyon *animation* · animasyon bölmesi *animation pane* · Animasyon Boyacısı *Animation Painter* · Tasarım Önerileri *Designer* · Dönüşüm *Morph* · Copilot *Copilot* · sunucu *presenter* · Sunucu Görünümü *Presenter View* · slayt gösterisi *slide show* · slayt boyutu *slide size* · üstbilgi / altbilgi *header / footer* · bilgi noktası (kiosk) *browsed at a kiosk* · erişilebilirlik *accessibility* · erişilebilirlik denetleyicisi *accessibility checker* · alternatif metin *alt text*

3.3.12 Kendini Dene

1. Anahat Görünümü'nde sırayla şu altı satır yazılıyor: Bütçe, Gelir, Burs, Kira, Gider, Sonuç. Sonra “Burs” ve “Kira” satırlarında Tab'a, ardından “Kira” satırında Shift + Tab'a basılıyor. Her aşamada kaç slayt vardır ve son durumda “Gelir” slaydının maddeleri nelerdir?

   Cevap

   Altı satırın hepsi başlık düzeyindedir: 6 slayt. Burs ve Kira'da Tab'a basılınca ikisi de Gelir'in maddesi olur: Bütçe, Gelir, Gider, Sonuç, yani 4 slayt. Kira'da Shift + Tab düzeyi yeniden yükseltir ve Kira ayrı bir slayt olur: Bütçe, Gelir, Kira, Gider, Sonuç, yani 5 slayt. Gelir slaydının tek maddesi Burs'tur.

2. 20 dakikalık bir oturumun 6 dakikası soru-cevaba ayrılıyor. Bir içerik slaydı 1–1,5 dakika taşıyorsa kaç içerik slaydı planlanmalıdır? Aynı sürede 35 slayt hazırlayan bir öğrenciye slayt başına kaç saniye düşer?

   Cevap

   Sunuma 20 − 6 = 14 dakika kalır. 14 ÷ 1,5 ≈ 9,3 ve 14 ÷ 1 = 14 olduğundan yaklaşık 10–14 içerik slaydı uygundur. 35 slaytta her slayda 14 × 60 ÷ 35 = 24 saniye düşer; bu süre bir grafiği açıklamaya yetmez.

3. Asıl slaytta başlık boyutu 40 punto, rengi lacivert. Slayt 2'nin başlığına daha önce elle yalnızca 28 punto boyut verilmişti. Asıl slaytta başlık boyutu 44 punto ve rengi bordo yapılıyor. Asıl görünüm kapatılınca Slayt 1 ve Slayt 2'nin başlıkları nasıl görünür? Slayt 2'ye Sıfırla uygulanırsa ne olur?

   Cevap

   Slayt 1: 44 punto, bordo; iki kural da asıl slayttan gelir. Slayt 2: boyut elle verildiği için 28 punto kalır, renk elle değiştirilmediği için bordo olur. Sıfırla elle verilen boyutu siler; Slayt 2'nin başlığı da 44 punto ve bordo olur.

4. 6000 × 4000 piksellik (3 : 2) bir fotoğraf, 1920 × 1080 piksellik bir projeksiyonda 16 : 9 slaytta tam ekran gösterilecek. Piksel başına 3 bayt sayılırsa fotoğrafın ham boyutu ekranın gösterebildiğinin kaç katıdır? Genişlik korunarak 16 : 9'a kırpılırsa piksellerin yüzde kaçı kalır?

   Cevap

   Fotoğraf 6000 × 4000 × 3 = 72.000.000 bayt (72 MB), ekran görüntüsü 1920 × 1080 × 3 = 6.220.800 bayttır. Oran 72.000.000 ÷ 6.220.800 ≈ 11,6'dır. Kırpmada yükseklik 6000 × 9 ÷ 16 = 3375 piksel olur; 4000 − 3375 = 625 satır atılır. Kalan oran 3375 ÷ 4000 ≈ %84,4'tür.

5. Yatayda 0'dan 960'a uzanan bir slaytta üç şekil var: A (x = 40, genişlik 120), B (x = 300, genişlik 100), C (x = 700, genişlik 200). Üçü seçilip Yatay Dağıt uygulanıyor; sonra gruplanıp yatayda ortalanıyor. Her adımdan sonra şekillerin x değerleri nedir?

   Cevap

   Dış kenarlar 40 ve 700 + 200 = 900'dür; aralık 860. Genişlikler toplamı 120 + 100 + 200 = 420, boş alan 860 − 420 = 440, her aralık 220. Dağıt sonrası A = 40, B = 40 + 120 + 220 = 380, C = 700. Grup 860 genişliğindedir; ortalanınca sol kenarı (960 − 860) ÷ 2 = 50 olur, yani grup 10 birim sağa kayar: A = 50, B = 390, C = 710.

6. Animasyon Bölmesi'nde sırayla şu efektler var: Başlık (Tıklandığında, süre 1 sn), Alt başlık (Önceki İle, gecikme 0,5 sn, süre 1 sn), Madde (Öncekinden Sonra, süre 0,5 sn), Logo (Öncekinden Sonra, gecikme 1 sn, süre 2 sn), Grafik (Tıklandığında, süre 1,5 sn). Sunucu t = 0 ve t = 8'de tıklıyor. Her efekt ne zaman oynar? Bölmede hangi numaralar görünür?

   Cevap

   Başlık 0–1 sn. Alt başlık Başlık'ın başlangıcından 0,5 sn sonra: 0,5–1,5 sn. Madde, önceki efektlerin en geç bitişinden (1,5) başlar: 1,5–2 sn. Logo, Madde'nin bitişinden 1 sn sonra: 3–5 sn. Grafik ikinci tıklamayla: 8–9,5 sn. Bölmede yalnızca Başlık'ın yanında 1, Grafik'in yanında 2 görünür; slaydın animasyonları için iki tıklama yeter.

7. Slayt 1 animasyonsuz. Slayt 2'ye Soluklaşma geçişi verilmiş. Madde 1 ve Madde 3 Tıklandığında, Madde 2 Önceki İle başlıyor; Madde 3'e ayrıca Öncekinden Sonra başlayan bir vurgu eklenmiş. Slayt 1'den başlayan gösteride Slayt 3'ü görmek için kaç tıklama gerekir?

   Cevap

   Dört tıklama. Tık 1 Soluklaşma ile Slayt 2'ye geçer. Tık 2 Madde 1 ile Madde 2'yi birlikte getirir. Tık 3 Madde 3'ü getirir; vurgu ardından kendiliğinden oynar. Tık 4 Slayt 3'e geçer.

8. Üç slaytlık bir bilgi noktası gösterisinde zamanlamalar 15, 20 ve 25 sn'dir. Gösteri başladıktan 100

sn ve 200 sn sonra ekranda hangi slayt vardır ve o slaytta kaç saniye geçmiştir?

Cevap

Tur süresi 15 + 20 + 25 = 60 sn; aralıklar Slayt 1 0–15, Slayt 2 15–35, Slayt 3 35–60. t = 100: 100 − 60 = 40, Slayt 3'ün aralığı; slaytta 40 − 35 = 5 sn geçmiştir. t = 200: 200 − 3 × 60 = 20, Slayt 2'nin aralığı; slaytta 20 − 15 = 5 sn geçmiştir.

9. 45 slaytlık bir sunu 6'lı dinleyici notu olarak çift taraflı basılacak. Kaç sayfa ve kaç yaprak gerekir? 4'lü düzende sayfa sayısı ne olur?

Cevap

45 ÷ 6 = 7,5 → 8 sayfa; çift taraflı baskıda 8 ÷ 2 = 4 yaprak. 4'lü düzende 45 ÷ 4 = 11,25 → 12 sayfa.

10. Sekiz slaytlık bir sunuda 7. slayt gizli. “Jüri” adlı özel gösteri 1, 2, 5 ve 8. slaytlardan oluşuyor. F5 ile kaç slayt gösterilir? “Jüri” gösterisi hangi slaytları gösterir? Gizli slayt için dosyanın paylaşılması güvenli midir?

Cevap

F5 ile 1, 2, 3, 4, 5, 6, 8 sırasıyla yedi slayt gösterilir; 7 atlanır. “Jüri” gösterisi yalnızca 1, 2, 5, 8'i gösterir. Gizli slayt dosyada kaldığı için dosyayı alan herkes görebilir; hassas bir içerik varsa paylaşmadan önce silinir.

3.3.13 İleri okuma

- Microsoft Destek, *PowerPoint yardımı ve öğrenme* (support.microsoft.com/powerpoint): slayt düzeni, Asıl Slayt, geçiş, animasyon, Sunucu Görünümü ve dışa aktarma konularını güncel menü adlarıyla ve kısa videolarla anlatan resmî yardım sayfaları.
- Microsoft Destek, *Engelli kişiler için erişilebilir PowerPoint sunuları oluşturma*: alternatif metin, okuma sırası, slayt başlıkları ve Erişilebilirlik Denetleyicisi'nin kullanımını adım adım verir.
- W3C, *Web Content Accessibility Guidelines (WCAG) 2.2*: kontrast oranının ve renk dışı ipucu kuralının dayandığı uluslararası erişilebilirlik yönergesi; göreli parlaklık formülünün tanımı buradadır.

3.3.14 Kaynaklar

1. Microsoft Support, PowerPoint help and learning (support.microsoft.com/powerpoint).
2. Microsoft Support, Add, change or remove transitions between slides.
3. Microsoft Support, Give a presentation in PowerPoint (Presenter View).

4. Microsoft Support, Benefits of embedding custom fonts (Türkçe sürüm: Dosyaya yazı tiplerini ekle).

5. Microsoft Support, Frequently asked questions about Copilot in Microsoft 365 subscriptions.

6. W3C, Web Content Accessibility Guidelines (WCAG) 2.2, 2023.

KISIM II

Algoritmadan Programa

Bölüm 4–5

KISIM II · BÖLÜM 4

Algoritmalar ve Programlamaya Giriş

Programlama, problem çözme adımlarını bilgisayarın yürütebileceği talimatlara dönüştürmektir. Algoritmalar sunumu problem tanımını, sözde kodu, iyi bir algoritmanın özelliklerini, faktöriyel, Fibonacci ve asal sayı gibi örnekleri, doğrusal ve ikili aramayı, seçimli, kabarcık, eklemeli ve birleştirme sıralamasını izleme tablolarıyla anlatır. Akış Diyagramları sıralı, koşullu ve tekrarlı akışı, sayaç ve biriktiriciyi gösterir. Programlamaya Giriş ise programlama dillerini, derleyici ve yorumlayıcıyı, programlama paradigmalarını, yazılım geliştirme sürecini ve hata ayıklamayı tanıtır.

Bu bölümün kodu: `github.com/sercankulcu/computer-programming-java/Bolum04/src`

BU BÖLÜMÜN NOTLARI

4.1 Algoritmalar

4.2 Akış Diyagramları

4.3 Programlamaya Giriş

KISIM II · BÖLÜM 4.1

Algoritmalar

Bir programın doğru çalışıp çalışmayacağı çoğu zaman ilk kod satırı yazılmadan belli olur. Problem açıkça tanımlanmış, çözüm yolu adım adım düşünülmüş ve küçük örneklerle denenmişse kodlama kolay bir çeviri işine dönüşür. Bu not, problemi tanımlamaktan başlar ve iyi bir algoritmanın özelliklerini, algoritma tasarlama döngüsünü ve sözde kodu anlatır. Ardından beş küçük algoritma satır satır izlenir: iki sayıyı sıralama, faktöriyel, Fibonacci dizisi, palindrom denetimi ve asal sayı denetimi. Algoritmanın doğruluğu izleme tablosu ve test durumlarıyla sınanır, verimliliği işlem sayısıyla ölçülür. Son iki kesim, bilgisayar biliminin en temel iki işini ele alır: doğrusal ve ikili arama ile seçimli, kabarcık, eklemeli ve birleştirme sıralaması. Her yöntem bir örnek dizi üzerinde adım adım çalıştırılır ve yaptığı karşılaştırmalar tek tek sayılır.

ÖĞRENME ÇIKTILARI

Tanımlama
Bir problemin amacını, girdisini, çıktısını, kısıtlarını, doğrulama örneğini ve hata durumunu belirlemek.

Tasarım
Çözümü açık, sıralı, uygulanabilir ve sonlu adımlara ayırmak; algoritmayı sözde kodla yazmak.

Doğrulama
Algoritmayı izleme tablosuyla elle çalıştırmak; normal, sınır, geçersiz ve büyük girdilerden oluşan test durumları seçmek.

Karşılaştırma
Doğrusal ve ikili aramayı, seçimli, kabarcık, eklemeli ve birleştirme sıralamasını karşılaştırma sayısı, zaman ve alan karmaşıklığı açısından değerlendirmek.

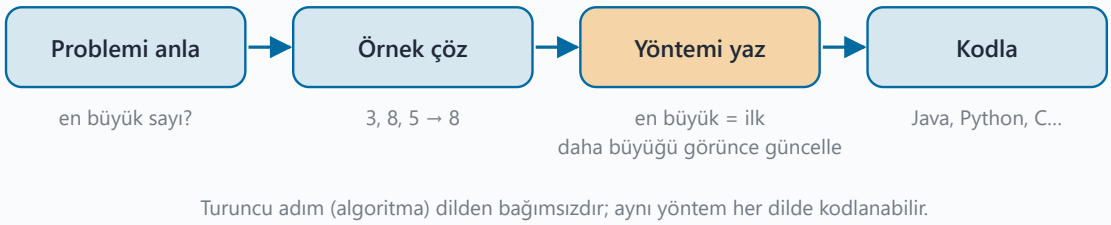
4.1.1 Problem çözme

Programlamaya yeni başlayan biri, bir ödevi okur okumaz bilgisayarın başına geçip kod yazmaya başlamak ister. Kısa ödevlerde bu yol bazen işe yarar. Problem biraz büyüdüğünde ise kod denemelerle şişer, bir hatayı düzeltmek başka bir hatayı doğurur ve programın neden çalıştığı ya da neden çalış-

madığı anlaşılmaz hâle gelir. Deneyimli programcılar tersini yapar: önce problemi anlar, sonra çözümü kâğıt üzerinde kurar, en son kodu yazar. **Problem çözme** (*problem solving*) bu düzenli yaklaşımın adıdır.

4.1.1.1 Önce algoritma, sonra kod

Kod, bir çözüm fikrinin belirli bir programlama dilindeki yazılışıdır. Aynı fikir Java ile de, Python ile de, C ile de yazılabilir; değişen yalnızca yazım kurallarıdır. Fikrin kendisi, yani problemi çözen adımlar dizisi, dilden bağımsızdır. Bu adımlar dizisine **algoritma** denir. Algoritma kurulmadan yazılan kod iki sıkıntı doğurur. Birincisi, programcı ne yapacağını yazarken düşündüğü için gereksiz denemeler yapar. İkincisi, bazı durumlar hiç düşünülmediği için gizli hatalar oluşur: program örnek girdide doğru çalışır, başka bir girdide yanlış sonuç verir.



Şekil 4.1.1: Problem çözmenin doğru sırası: problemi anla, küçük bir örneği elle çöz, yöntemi yaz, sonra kodla. Örnek problem bir listedeki en büyük sayıyı bulmaktır. Turuncu adım (algoritma) dilden bağımsızdır; aynı yöntem her dilde kodlanabilir.

Şekil 4.1.1 bu sırayı "en büyük sayıyı bul" problemi üzerinde gösterir. İlk adım **problemi anlamaktır**: liste boş olabilir mi, sayılar negatif olabilir mi, aynı sayı birden çok kez geçebilir mi? İkinci adımda **küçük bir örnek elle çözülür**. 3, 8, 5 listesinde en büyük sayı 8'dir. Bunu bulurken insan farkında olmadan bir yöntem izler: sayılara soldan sağa bakar ve o ana kadar gördüğü en büyük değeri aklında tutar. Üçüncü adımda bu yöntem açıkça **yazılır**: "En büyük değer başta listenin ilk öğesidir; sıradaki öğe daha büyükse en büyük değeri güncelle." Dördüncü adım **kodlamadır**; yöntem artık bellidir, geriye seçilen dilin yazım kurallarını uygulamak kalır.

İkinci adım çoğu zaman atlanır ama en öğretici adımdır. Elle çözülen örnek, algoritmanın taslağını verir. Örneği çözerken yapılan her işlem algoritmanın bir adımına, akılda tutulan her bilgi ise bir değişkene dönüşür. En büyük sayı örneğinde akılda tutulan bilgi "şimdiye kadarki en büyük değer"dir; programda bu, `enBuyuk` adlı bir değişken olur.

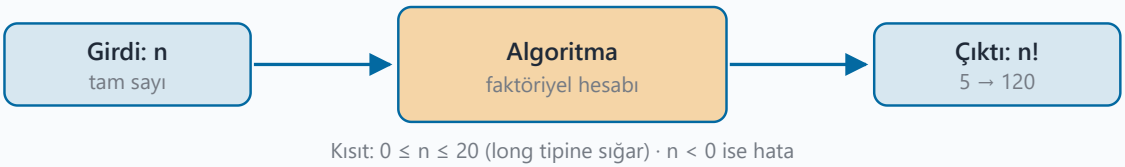
4.1.1.2 Problem tanımı

Bir problemi anlamak, onu birkaç soruya cevap verecek biçimde yazmak demektir. Bu sorulara verilen cevapların bütününe **problem tanımı** (*problem definition*) denir.

Tanım 4.1.1

Problem tanımı, çözülecek problemin altı öğesini açıkça yazar: **amaç** (hangi sorun çözülecek), **girdi** (programa hangi bilgiler verilecek), **çıktı** (hangi sonuç üretilecek), **kısıt** (*constraint*; değerlerin sınırı ve geçerlilik koşulu), **örnek** (beklenen sonucun hangi küçük örnekle doğrulanacağı) ve **hata durumu** (geçersiz ya da eksik girdide ne yapılacağı).

Bu altı öğenin her biri, kod yazılırken verilecek bir kararı önceden verir. Amaç, programın ne zaman "bitmiş" sayılacağını belirler. Girdi ve çıktı, programın dış dünyayla nasıl konuşacağını söyler. Kısıt, programın hangi değerler için doğru çalışmak zorunda olduğunu sınırlar. Örnek, program yazıldıktan sonra yapılacak ilk denetimi hazırlar. Hata durumu ise en çok unutulan öğedir: kullanıcı beklenmeyen bir değer girdiğinde programın çökmesi yerine anlamlı bir mesaj vermesi bu kararla sağlanır.



Şekil 4.1.2: Faktöriyel hesabı için problem tanımının özeti. Girdi bir tam sayı n'dir, çıktı n!'dir; kısıt 0 ≤ n ≤ 20'dir (sonuç 64 bitlik long tipine sığar) ve n < 0 ise hata verilir.

Örnek 4.1.1

"Bir sayının faktöriyelini hesapla" isteği için tam bir problem tanımı nasıl yazılır?

Faktöriyel, 1'den n'ye kadar olan tam sayıların çarpımıdır: *5! = 1 × 2 × 3 × 4 × 5 = 120*. Tanım gereği *0! = 1*'dir. Problem tanımının altı öğesi şöyle doldurulur:

Öğe	Faktöriyel problemi için cevap
Amaç	Verilen n sayısının faktöriyelini hesaplamak
Girdi	Bir tam sayı n
Çıktı	n! değeri
Kısıt	0 ≤ n ≤ 20 (sonuç 64 bitlik long tipinde tutulacak)
Örnek	n = 5 için 120; n = 0 için 1
Hata durumu	n < 0 ya da n > 20 ise "hata" yazılır

Kısıt satırındaki 20 sınırı keyfî değildir. Java'daki `long` tipi en çok 9.223.372.036.854.775.807 değerini tutar. *20! = 2.432.902.008.176.640.000* bu sınırın altında kalır, *21! = 51.090.942.171.709.440.000* ise üstüne çıkar.

Sık yapılan hata

Problem tanımında yalnızca girdi ve çıktıyı yazıp kısıtı ve hata durumunu atlamak. "n sayısının faktöriyelini hesapla" cümlesi, n = −3 girildiğinde ne olacağını söylemez. Bu soru tasarım aşamasında cevaplanmazsa program ya anlamsız bir sonuç verir ya da hiç durmayan bir döngüye girer. Kısıt ve hata durumu, en az girdi ve çıktı kadar önemlidir.

4.1.2 Algoritma

4.1.2.1 Algoritma nedir?

Günlük hayat algoritmalarla doludur. Bir yemek tarifi, bir mobilyanın kurulum kılavuzu, bir bankamatikten para çekme talimatı birer algoritma sayılabilir. Hepsinde belirli bir sonuca ulaşmak için sırayla uygulanacak adımlar vardır. Bilgisayar biliminde algoritma kavramı daha kesin tanımlanır, çünkü adımları uygulayacak olan bir insan değil, kendi başına yorum yapamayan bir makinedir.

Tanım 4.1.2

Algoritma (*algorithm*), belirli bir problemi çözmek için tanımlanmış sonlu adımlar dizisidir. Bir algoritma sıfır ya da daha fazla girdi alır ve en az bir çıktı üretir.

Tanımdaki "sıfır ya da daha fazla girdi" ifadesi ilk bakışta tuhaf görünebilir. Hiç girdi almayan bir algoritma da vardır: "1'den 100'e kadar olan sayıları yaz" işi dışarıdan hiçbir bilgi istemez. Buna karşılık çıktısı olmayan bir algoritma işe yaramaz; sonucu hiçbir yere yansımayan bir hesap, yapılmamış bir hesaptan farksızdır. Çıktı ekrana yazılan bir sayı olabileceği gibi bir dosyaya kaydedilen veri ya da bir motoru çalıştıran bir sinyal de olabilir.

4.1.2.2 İyi algoritmanın özellikleri

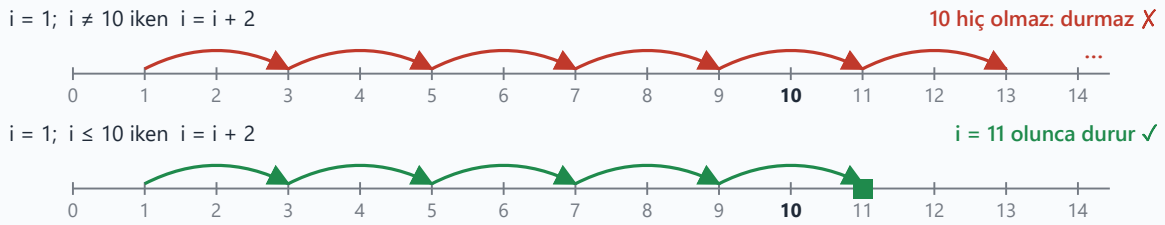
Her adımlar dizisi iyi bir algoritma değildir. "Sayıları uygun biçimde düzenle" cümlesi bir adım gibi görünür ama "uygun" sözcüğü ne yapılacağını söylemez. Bir algoritmanın bilgisayarda güvenle çalışması için adımlarının bazı özellikleri taşıması gerekir.

Özellik	Anlamı	Bozulduğunda ne olur?
Kesinlik	Her adım açık ve tek anlamlıdır	"Biraz büyüt" gibi bir adımı farklı kişiler farklı uygular
Sıralılık	Adımların hangi sırayla yapılacağı bellidir	Takasta atamaların sırası değişirse bir değer kaybolur
Uygulanabilirlik	Her adım gerçekten yapılabilir	"Bütün asal sayıları listele" adımı hiçbir zaman tamamlanamaz
Sonluluk	Algoritma sonlu sayıda adımdan sonra durur	Döngü koşulu hiç yanlış olmaz, program durmaz

Özellik	Anlamı	Bozulduğunda ne olur?
Doğruluk	Geçerli her girdi için doğru sonucu üretir	Program bazı girdilerde yanlış sonuç verir
Verimlilik	Girdi büyüdüğünde kaynak kullanımı kabul edilebilir kalır	Küçük veride hızlı olan program büyük veride saatlerce çalışır

Tablo 4.1.1: İyi bir algoritmanın özellikleri.

Tablo 4.1.1 altı özelliği ve her biri bozulduğunda görülen belirtiyi listeler. Bunlardan **kesinlik** (*definiteness*) ve **sonluluk** (*finiteness*) en sık gözden kaçanlardır. Sonluluğun ne kadar ince bir koşula bağlı olabileceğini küçük bir örnek gösterir.



Şekil 4.1.3: Sonluluk. Üstteki döngüde i, 1'den başlayıp ikişer artar: 1, 3, 5, ... Değer hiçbir zaman tam 10 olmadığı için i ≠ 10 koşulu hep doğru kalır ve döngü durmaz. Alttaki döngüde koşul i ≤ 10'dur; i = 11 olunca koşul yanlış olur ve döngü durur.

Şekil 4.1.3 iki döngüyü karşılaştırır. İkisinde de sayaç 1'den başlar ve her turda 2 artar. Üstteki döngü "i, 10'a eşit olmadığı sürece devam et" der. Sayaç 1, 3, 5, 7, 9, 11, 13 ... diye ilerler ve 10'u atlar. Koşul hiçbir zaman yanlış olmadığından döngü sonsuza dek sürer. Alttaki döngü "i, 10'dan küçük ya da 10'a eşit olduğu sürece devam et" der. Sayaç 11 olduğunda *11 ≤ 10* yanlış olur ve döngü biter. İki koşul arasındaki fark tek bir simgedir ama birinde algoritma vardır, ötekinde yoktur.

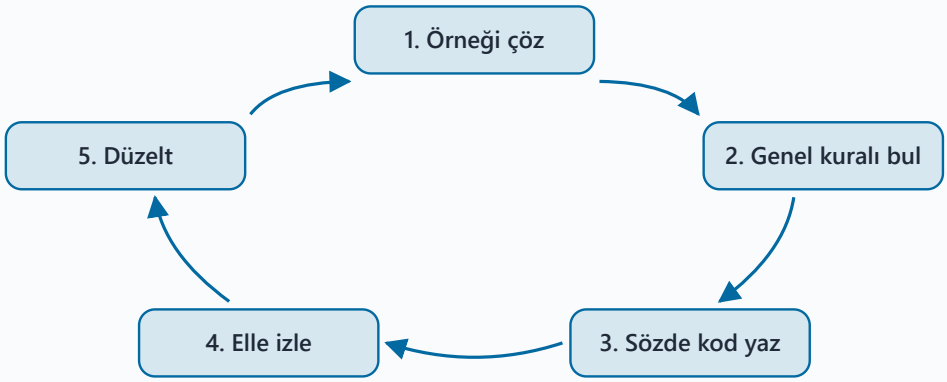
Sık yapılan hata

Döngüyü bitirmek için eşitlik (= ya da ≠) koşulu kullanmak. Sayaç tam o değere hiç uğramazsa döngü durmaz. Sayaç bir sınıra doğru ilerliyorsa "küçük ya da eşit" (≤) ya da "küçük" (<) koşulu daha güvenlidir; sayaç sınırı geçtiği anda döngü kesinlikle biter.

Doğruluk (*correctness*) ile **verimlilik** ayrı özelliklerdir. Doğru bir algoritma yavaş olabilir, hızlı bir algoritma bazı girdilerde yanlış sonuç verebilir. Önce doğruluk sağlanır; hızlı ama yanlış bir sonucun hiçbir değeri yoktur. Verimlilik ise girdi büyüdükçe önem kazanır.

4.1.2.3 Algoritma tasarlama döngüsü

İyi bir algoritma nadiren ilk denemede ortaya çıkar. Tasarım, birkaç kez tekrarlanan bir döngüdür. Her turda algoritma biraz daha netleşir ve gözden kaçan durumlar yakalanır.



Şekil 4.1.4: Algoritma tasarlama döngüsü: örneği çöz, genel kuralı bul, sözde kod yaz, elle izle, düzelt. Elle izlemede hata çıkarsa düzeltilir ve döngü yeniden döner; hata kalmayınca kodlamaya geçilir.

Şekil 4.1.4'teki beş adım şöyle işler:

1. **Örneği çöz.** Problemin küçük bir örneği elle çözülür. Faktöriyel için *4! = 1 × 2 × 3 × 4 = 24* hesaplanır.
2. **Genel kuralı bul.** Elle yapılan işlemlerdeki tekrar fark edilir: her adımda o ana kadarki çarpım bir sonraki sayıyla çarpılıyor. Kural "çarpımı 1 ile başlat, 2'den n'ye kadar her sayıyla çarp" biçiminde yazılır.
3. **Sözde kod yaz.** Kural, programlama diline yakın ama ona bağlı olmayan açık adımlara dönüştürülür.
4. **Elle izle.** Sözde kod birkaç girdiyle, bilgisayar gibi satır satır çalıştırılır; her değişkenin değeri bir tabloya yazılır.
5. **Düzelt.** İzlemede beklenen sonuçla bulunan sonuç ayrışırsa hata bulunur ve sözde kod düzeltilir. Sonra döngü yeniden döner: düzeltilmiş sözde kod yeniden izlenir.

Döngü, elle izlemede artık hata çıkmayınca biter ve kodlamaya geçilir. Aynı problem için birden fazla çözüm bulunmuşsa bu aşamada doğruluk, okunabilirlik ve verimlilik bakımından karşılaştırılır. Okunabilirlik küçümsenmemelidir: bir algoritmayı en çok okuyan kişi, birkaç ay sonra onu değiştirmek zorunda kalan programcının kendisidir.

4.1.2.4 Sözde kod

Algoritmayı yazmanın birkaç yolu vardır. Günlük dille yazılan adımlar kolay anlaşılır ama belirsizliğe açıktır. Programlama diliyle yazılan adımlar kesindir ama yazım kurallarıyla doludur; noktalı virgüller ve parantezler düşüncenin önüne geçer. **Sözde kod** bu iki ucun arasında durur.

Tanım 4.1.3

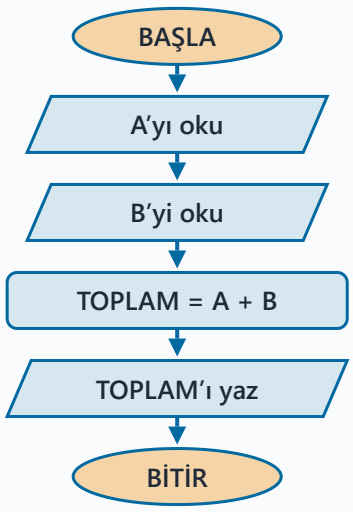
Sözde kod (*pseudocode*), algoritmanın mantığını belirli bir programlama diline bağlı kalmadan, doğal dile yakın, kısa ve tutarlı ifadelerle anlatan yazım biçimidir.

Sözde kodun herkesçe kabul edilmiş tek bir yazım kuralı yoktur. Önemli olan, bir algoritma içinde aynı ifadenin hep aynı anlamda kullanılmasıdır. Bu notta kullanılan ifadeler şunlardır: `oku` bir girdiyi alır, `yaz` bir çıktı üretir, `=` sağdaki değeri soldaki değişkene atar, `eğer … ise` bir koşulu denetler, `… iken` koşul doğru olduğu sürece altındaki satırları tekrarlar, `… kadar tekrarla` belirli sayıda tekrar yapar. **Girinti** (*indentation*), yani satır başındaki boşluk, bir koşulun ya da döngünün hangi satırları kapsadığını gösterir.

En basit örnek, iki sayının toplamını yazan algoritmadır:

```
BAŞLA
  A değerini oku
  B değerini oku
  TOPLAM = A + B
  TOPLAM değerini yaz
BİTİR
```

Aynı algoritma bir **akış diyagramıyla** da çizilebilir. Şekil 4.1.5 her sözde kod satırının diyagramdaki karşılığını gösterir: başlangıç ve bitiş oval, girdi ve çıktı paralelkenar, işlem dikdörtgen kutudur. Oklar adımların sırasını belirtir.



Şekil 4.1.5: Toplama algoritmasının akış diyagramı: başla, A'yı oku, B'yi oku, TOPLAM = A + B, TOPLAM'ı yaz, bitir.

Sözde kodun amacı yazım kurallarına uymak değil, algoritmayı anlaşılır kılmaktır. Bu yüzden sözde kodda “dizi[i] ile dizi[j] yer değiştir” gibi bir satır rahatlıkla yazılabilir. Java'da aynı iş üç atama ister ama algoritmanın mantığı açısından tek bir adımdır. Sözde kod hazır olunca koda çevirmek çoğu zaman satır satır bir iştir: `eğer … ise` satırı `if`'e, `… iken` satırı `while`'a dönüşür.

> **Sık yapılan hata**
>
> Sözde kodu ya çok belirsiz ya da gereğinden fazla ayrıntılı yazmak. “Listeyi işle ve sonucu bul” satırı hiçbir şey anlatmaz. Öte uçta, her satırın sonuna noktalı virgül koyup değişken tiplerini tek tek bildiren sözde kod, aslında Java kodudur ve sözde kodun kolaylığını yitirir. Ölçü şudur: sözde kodu okuyan biri, hangi dili bilirse bilsin, algoritmayı o dilde yazabilmelidir.

4.1.3 Örnek algoritmalar

Algoritma kurmayı öğrenmenin en iyi yolu, küçük ve iyi bilinen problemleri çözmektir. Bu kesimdeki beş örnek giderek zorlaşır. İlki yalnızca bir koşul ve üç atama içerir. Faktöriyel ve Fibonacci örnekleri döngü ve biriktirme fikrini getirir. Palindrom ve asal sayı örneklerinde ise döngü, sonuç belli olur olmaz erken biter. Her örnekte önce sözde kod verilir, sonra algoritma bir girdi için adım adım izlenir.

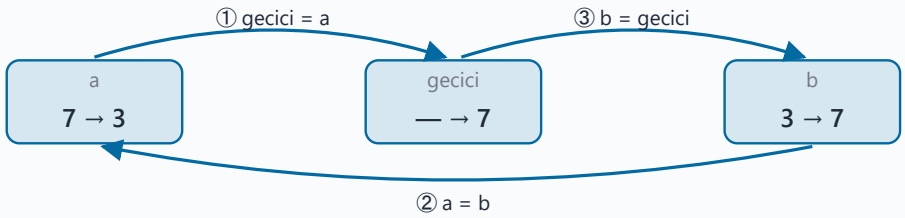
4.1.3.1 İki sayıyı sıralama ve takas

Problem şudur: iki sayı okunacak ve küçük olan önce yazılacak. a = 7, b = 3 girilirse çıktı “3 7” olmalıdır; a = 2, b = 9 girilirse “2 9”. Çözüm fikri basittir. a, b'den büyükse iki değişkenin değerleri yer değiştirir; değilse zaten doğru sıradadır. İki değişkenin değerlerini yer değiştirmeye **takas** (*swap*) denir.

```
oku a, b
eğer a > b ise
    gecici = a
    a = b
    b = gecici
yaz a, b
```

Algoritmanın 2. satırı koşulu denetler. Koşul doğruysa girintili üç satır çalışır, yanlışsa atlanır. Eşit değerlerde (a = b) koşul yanlıştır ve takas yapılmaz; zaten gerekmez. Son satır her iki durumda da çalışır ve a ≤ b koşulunu sağlayan çıktıyı yazar.

Takasın neden üç atama gerektirdiği, algoritmanın en öğretici yanıdır. Bir değişken aynı anda yalnızca bir değer tutabilir. `a = b` atamasıyla a'ya yeni değer yazıldığı anda eski değeri silinir. Eski değer önceden bir yere kopyalanmamışsa bir daha geri getirilemez. Bu yüzden önce a'nın değeri **geçici değişkene** (*temporary variable*) kopyalanır.



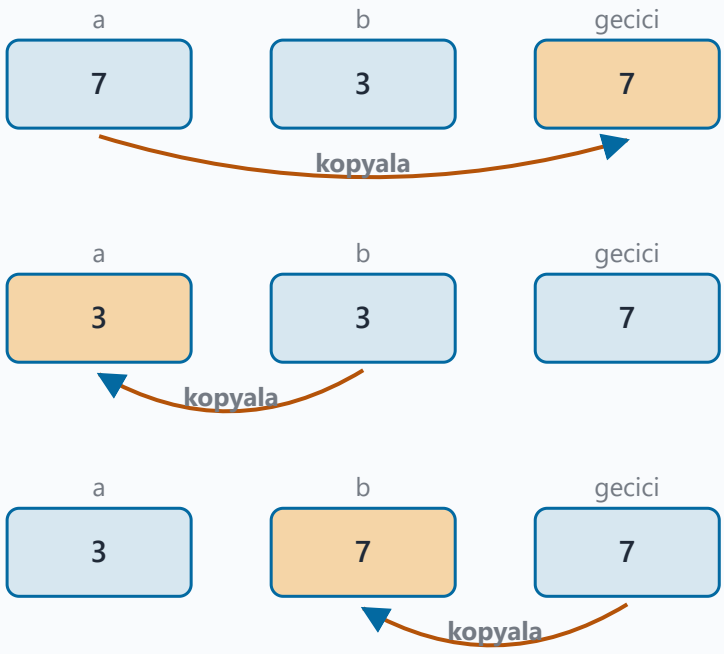
Şekil 4.1.6: Takasın üç ataması. ① gecici, a'nın 7 değerini saklar. ② a, b'den 3 değerini alır; a'nın eski değeri artık yalnızca gecici'dedir. ③ b, gecici'den 7 değerini alır.

Şekil 4.1.6 üç atamayı oklarla gösterir. Atamalar bir döngü oluşturur: a'dan gecici'ye, b'den a'ya, gecici'den b'ye. Algoritmanın a = 7, b = 3 için adım adım çalışması Tablo 4.1.2'de verilmiştir.

Adım	Çalışan satır	a	b	gecici
1	`oku a, b`	7	3	—
2	`eğer 7 > 3 ise`	7	3	—
3	`gecici = a`	7	3	7
4	`a = b`	3	3	7
5	`b = gecici`	3	7	7
6	`yaz a, b → 3 7`	3	7	7

Tablo 4.1.2: İki sayıyı sıralama algoritmasının a = 7, b = 3 için izleme tablosu. Her satır, o satır çalıştıktan sonraki değerleri gösterir; "—" değişkenin henüz değer almadığını belirtir.

Tablonun 4. satırında a ve b'nin ikisi de 3'tür. Algoritmanın en kırılgan anı budur: 7 değeri yalnızca gecici'de yaşar. 5. satırda gecici'deki 7, b'ye kopyalanır ve takas tamamlanır. a = 2, b = 9 girilseydi 2. satırdaki *2 > 9* koşulu yanlış olur, 3–5. satırlar hiç çalışmaz ve doğrudan "2 9" yazılırdı.



Şekil 4.1.7: Takasın üç anı (sırasıyla): gecici = a sonrası (a = 7, b = 3, gecici = 7); a = b sonrası (a = 3, b = 3, gecici = 7); b = gecici sonrası (a = 3, b = 7, gecici = 7).

Sık yapılan hata

Takası geçici değişken olmadan, `a = b` ve `b = a` diye iki atamayla yazmak. a = 7, b = 3 için ilk atama a'yı 3 yapar ve 7'yi siler. İkinci atama b'ye a'nın yeni değerini, yani 3'ü yazar. Sonuç "3 3" olur; 7 kaybolmuştur. Atama bir kopyalama işlemidir, iki kutunun içeriğini aynı anda değiştirmez.

İki sayıyı sıralama fikri, daha fazla sayıya genişletilebilir. Üç sayı için iki sayıyı sıralama adımı üç kez uygulanır: önce a ile b, sonra b ile c, en son yeniden a ile b karşılaştırılır.

Örnek 4.1.2

Aşağıdaki algoritma a = 8, b = 5, c = 1 için nasıl çalışır? Neden üç karşılaştırma yeterlidir?

```
oku a, b, c
eğer a > b ise a ile b yer değiştir
eğer b > c ise b ile c yer değiştir
eğer a > b ise a ile b yer değiştir
yaz a, b, c
```

Adım	Koşul	Sonuç	a	b	c
başlangıç	—	—	8	5	1
1. karşılaştırma	a > b	doğru → a ile b yer değiştirir	5	8	1
2. karşılaştırma	b > c	doğru → b ile c yer değiştirir	5	1	8
3. karşılaştırma	a > b	doğru → a ile b yer değiştirir	1	5	8

Çıktı “1 5 8”dir. Girdi ters sıralı olduğu için üç karşılaştırmanın üçünde de takas yapıldı. İlk iki karşılaştırma, kabarcık gibi, en büyük değeri en sağa taşır: 1. adımdan sonra 8, b'ye; 2. adımdan sonra c'ye geçer. c artık kesin olarak en büyük değeri tutar. Geriye a ile b'yi sıralamak kalır; bunu 3. karşılaştırma yapar. Bu yüzden üç sayı için üç karşılaştırma her zaman yeterlidir.

4.1.3.2 Faktöriyel

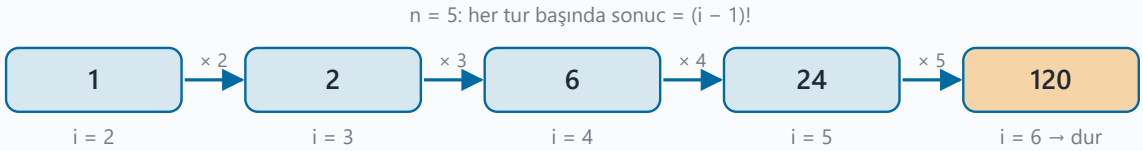
n'nin **faktöriyeli** (*factorial*), 1'den n'ye kadar olan tam sayıların çarpımıdır ve n! diye yazılır. *3! = 1 × 2 × 3 = 6*, *5! = 120*'dir. 0! tanım gereği 1'dir. Faktöriyelin problem tanımı 4.1.1.2'de yazılmıştı; algoritma bu tanımı adım adım izler.

```
oku n
eğer n < 0 ise "hata" yaz ve dur
sonuc = 1
i = 2
i ≤ n iken
    sonuc = sonuc × i
    i = i + 1
yaz sonuc
```

Satırlar şu işleri yapar:

1. `oku n` : faktöriyeli hesaplanacak sayı okunur.
2. `eğer n < 0 ise "hata" yaz ve dur` : negatif sayıların faktöriyeli tanımlı değildir. Bu satır problem tanımındaki hata durumunu karşılar.
3. `sonuc = 1` : çarpımın başlangıç değeri 1'dir. Değişken çarpımı adım adım biriktirir.
4. `i = 2` : sayaç 2'den başlar. 1 ile çarpmak değeri değiştirmediği için 1'den başlamaya gerek yoktur.
5. `i ≤ n iken` : i, n'yi geçmediği sürece altındaki iki satır tekrarlanır.
6. `sonuc = sonuc × i` : o ana kadarki çarpım i ile çarpılır.
7. `i = i + 1` : sayaç bir artar; sonra 5. satıra dönülür ve koşul yeniden denetlenir.
8. `yaz sonuc` : döngü bittiğinde sonuç yazılır.

3. satırdaki gibi, bir döngü boyunca bir sonucu adım adım biriktiren değişkene **biriktirici** (*accumulator*) denir. Biriktiricinin başlangıç değeri yapılan işleme göre seçilir. Toplama için 0 seçilir, çünkü 0 eklemek toplamı değiştirmez. Çarpma için 1 seçilir, çünkü 1 ile çarpmak çarpımı değiştirmez.



Şekil 4.1.8: n = 5 için faktöriyel zinciri. Her tur başında sonuc = (i − 1)!'dir: i = 2'de 1, i = 3'te 2, i = 4'te 6, i = 5'te 24. Son turda 24 × 5 = 120 bulunur; i = 6 olunca döngü durur.

Şekil 4.1.8 çarpımın nasıl büyüdüğünü gösterir. Algoritmanın n = 5 için tam izlemesi Tablo 4.1.3'te verilmiştir.

An	Koşul i ≤ n	Çarpma	sonuc	Yeni i
başlangıç	—	—	1	2
1. tur	2 ≤ 5 → doğru	1 × 2 = 2	2	3
2. tur	3 ≤ 5 → doğru	2 × 3 = 6	6	4
3. tur	4 ≤ 5 → doğru	6 × 4 = 24	24	5
4. tur	5 ≤ 5 → doğru	24 × 5 = 120	120	6
çıkış	6 ≤ 5 → yanlış	—	120	6

Tablo 4.1.3: Faktöriyel algoritmasının n = 5 için izleme tablosu. "Koşul" sütunu turun başında denetlenen i ≤ n koşuludur.

Tabloda döngünün 4 kez döndüğü görülür: i, 2, 3, 4 ve 5 değerlerini alır. Genel olarak n ≥ 2 için döngü $n - 1$ kez döner. i = 6 olduğunda $6 \le 5$ koşulu yanlıştır ve döngü biter; 120 yazılır.

Tablonun sonuc sütununda bir düzen göze çarpar. Her turun başında sonuc, i'nin bir eksiğinin faktöriyeline eşittir: i = 2 iken 1 = 1!, i = 3 iken 2 = 2!, i = 4 iken 6 = 3!. Döngünün her turunda doğru kalan böyle bir ifadeye **döngü değişmezi** denir.

Tanım 4.1.4

Döngü değişmezi (*loop invariant*), bir döngünün her turunun başında doğru olan bir ifadedir. Faktöriyel algoritmasının döngü değişmezi "her tur başında *sonuc = (i − 1)!*" ifadesidir.

Döngü değişmezi, algoritmanın neden doğru olduğunu gösterir. Döngü ilk kez başlarken i = 2 ve sonuc = 1 = 1!'dir; ifade doğrudur. Bir turda sonuc i ile çarpılır ve i bir artar. Turdan önce *sonuc = (i − 1)!* ise turdan sonra *sonuc = (i − 1)! × i = i!* olur. Yeni i, eski i'nin bir fazlası olduğundan ifade yeniden doğrudur. Döngü *i = n + 1* iken biter; bu anda *sonuc = n!*'dir. Tam istenen sonuç budur.

Algoritmanın küçük girdilerdeki davranışı da değişmezle açıklanır. n = 0 ya da n = 1 için ilk denetimde $2 \le n$ koşulu yanlıştır. Döngü hiç dönmez ve sonuc = 1 kalır; 0! ve 1! zaten 1'dir. Algoritma bu iki durum için ayrı bir satıra ihtiyaç duymaz.

Sık yapılan hata

Biriktiriciye yanlış başlangıç değeri vermek ya da döngü sınırını bir eksik yazmak. `sonuc = 0` yazılırsa her çarpım 0 çıkar ve algoritma her n için 0 yazar. Koşul `i ≤ n` yerine `i < n` yazılırsa son çarpma yapılmaz; n = 5 için 120 yerine 24 bulunur. İkinci hataya **sınır hatası** (*off-by-one error*) denir. Sınır hatası n = 0 ve n = 1 girdilerinde görünmez, çünkü orada döngü zaten dönmez. Bu yüzden algoritma tek bir girdiyle denenip bırakılmamalıdır.

Biriktirici fikri yalnızca faktöriyelde değil, toplam, sayaç ve ortalama gerektiren her problemde kullanılır. Aşağıdaki örnek, aynı kalıbı toplama için uygular.

Örnek 4.1.3

1'den n'ye kadar olan çift sayıların toplamını bulan algoritma nasıl yazılır? n = 10 için nasıl çalışır?

Toplama yapılacağı için biriktiricinin başlangıç değeri 0'dır. Çift sayılar 2'den başlar ve ikişer artar; sayaç da 2'den başlayıp her turda 2 artırılır. Böylece tek sayılara hiç bakılmaz.

```
oku n
toplam = 0
i = 2
i ≤ n iken
    toplam = toplam + i
    i = i + 2
yaz toplam
```

An	Koşul i ≤ n	toplam	Yeni i
başlangıç	—	0	2
1. tur	2 ≤ 10 → doğru	2	4
2. tur	4 ≤ 10 → doğru	6	6
3. tur	6 ≤ 10 → doğru	12	8
4. tur	8 ≤ 10 → doğru	20	10
5. tur	10 ≤ 10 → doğru	30	12
çıkış	12 ≤ 10 → yanlış	30	12

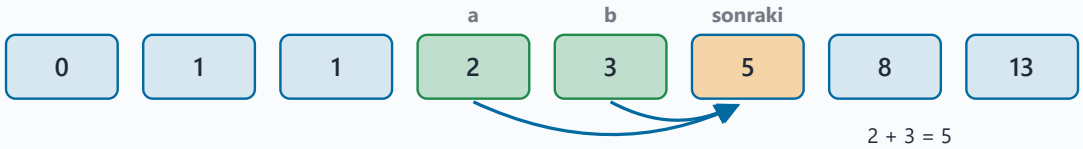
Sonuç *2 + 4 + 6 + 8 + 10 = 30*'dur. Döngü 5 kez döner; sayaç 12 olunca koşul yanlış olur. n = 0 ya da n = 1 için koşul ilk denetimde yanlıştır ve 0 yazılır; 1'den küçük ya da 1'e eşit hiçbir çift pozitif sayı olmadığı için bu doğrudur. Döngü koşulunun ≤ olması önemlidir: n = 10 için 10'un kendisi de toplama girmelidir.

4.1.3.3 Fibonacci dizisi

Fibonacci dizisi 0 ve 1 ile başlar; sonraki her terim, kendinden önceki iki terimin toplamıdır: 0, 1, 1, 2, 3, 5, 8, 13, 21, … Problem, dizinin ilk n terimini yazmaktır. Faktöriyelde her turda tek bir değeri hatırlamak yetiyordu. Burada yeni terimi hesaplamak için iki önceki değerin birlikte saklanması gerekir.

```
oku n
a = 0, b = 1
0'dan n-1'e kadar tekrarla
    yaz a
    sonraki = a + b
    a = b
    b = sonraki
```

a ve b, dizinin art arda gelen iki terimini tutar. Her turda a yazılır, sonra a ile b toplanarak bir sonraki terim bulunur. Ardından iki değişken birer terim ilerler: a, b'nin değerini; b, yeni terimin değerini alır. a ile b, dizinin üzerinde her turda bir sağa kayan iki hücrelik bir pencere gibidir. Bu yönteme **kayan pencere** (*sliding window*) denir.



Tur 3: a (2) yazılır, sonraki = a + b; sonra a = 3, b = 5 olur ve pencere bir sağa kayar.

Şekil 4.1.9: Fibonacci dizisinde kayan pencere. Tur 3'te a = 2 yazılır ve sonraki = a + b = 2 + 3 = 5 hesaplanır; sonra a = 3, b = 5 olur ve pencere bir sağa kayar.

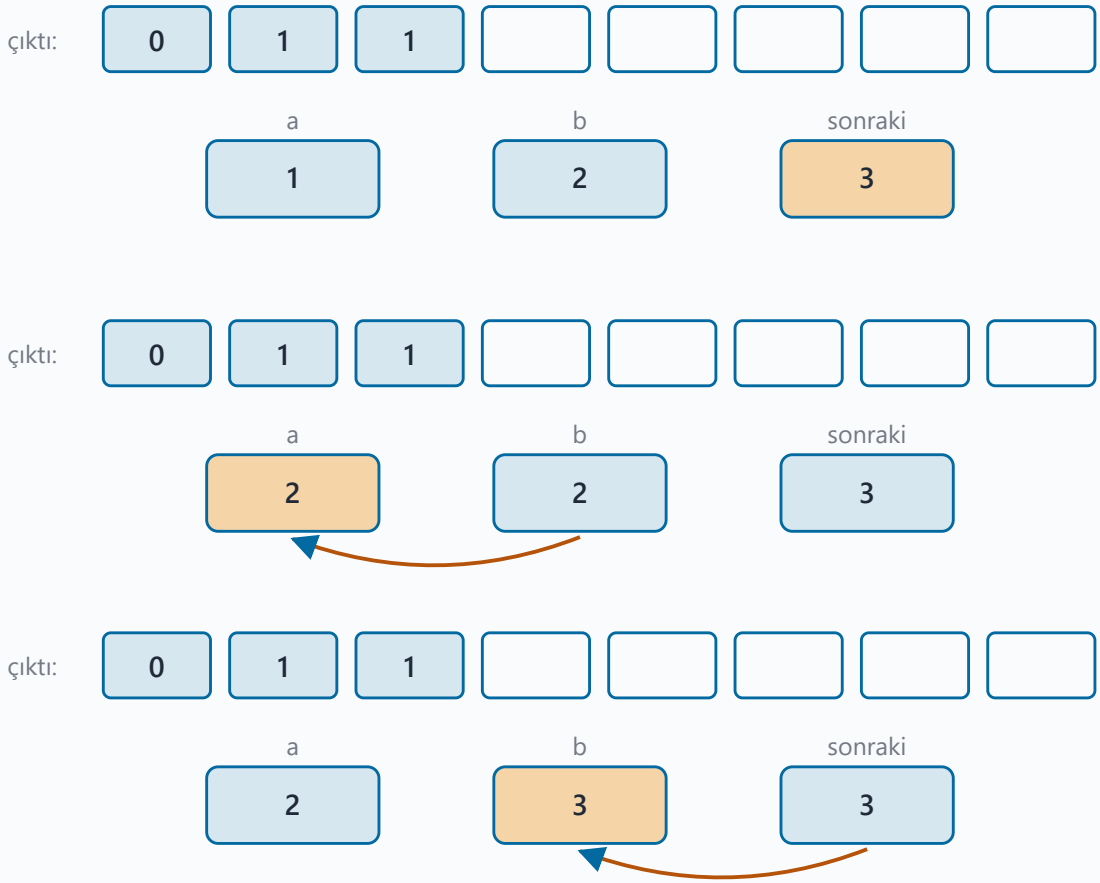
Şekil 4.1.9 üçüncü turdaki durumu gösterir. Algoritmanın n = 8 için bütün turları Tablo 4.1.4'te izlenmiştir.

Tur	Yazılan a	sonraki = a + b	Yeni a	Yeni b
başlangıç	—	—	0	1
0	0	0 + 1 = 1	1	1
1	1	1 + 1 = 2	1	2
2	1	1 + 2 = 3	2	3
3	2	2 + 3 = 5	3	5
4	3	3 + 5 = 8	5	8
5	5	5 + 8 = 13	8	13
6	8	8 + 13 = 21	13	21

Tur	Yazılan a	sonraki = a + b	Yeni a	Yeni b
7	13	13 + 21 = 34	21	34

Tablo 4.1.4: Fibonacci algoritmasının n = 8 için izleme tablosu. Turlar 0'dan n − 1 = 7'ye kadar numaralanır; "Yeni a" ve "Yeni b" turun sonundaki değerlerdir.

Çıktı, "Yazılan a" sütununun sırasıdır: 0 1 1 2 3 5 8 13. Son turda sonraki = 34 hesaplanır ama yazılmaz; döngü 8 turdan sonra biter. Bu küçük israf algoritmanın sadeliği için ödenen bedeldir. Algoritma n = 0 için hiçbir şey yazmaz, n = 1 için yalnızca 0 yazar; ikisi de beklenen sonuçtur.



Şekil 4.1.10: Fibonacci, tur 2'nin son üç adımı (sırasıyla): sonraki = 1 + 2 = 3 hesaplanır; a = b ile a, 2 olur; b = sonraki ile b, 3 olur. Üst satır o ana kadar yazılan terimlerdir.

Son iki atamanın sırası, takastaki sorunun benzerini taşır. Şekil 4.1.10'daki gibi önce `a = b`, sonra `b = sonraki` yapılır. Sıra ters çevrilirse `b = sonraki` ataması b'nin eski değerini siler. Ardından `a = b`, a'ya eski b'yi değil yeni terimi yazar. a ile b eşit olur ve her tur yeni terim *a + a = 2a* diye hesaplanır. Dizi 0, 1, 2, 4, 8 diye ikiye katlanarak ilerler.

Sık yapılan hata

Birbirine bağlı atamaların sırasını önemsememek. Bir değişkene yeni değer yazılmadan önce, eski değerine ihtiyaç duyan bütün işlemler bitmiş olmalıdır. Fibonacci'de eski b'ye a için ihtiyaç vardır; bu yüzden önce `a = b` yazılır. Geçici değişken (burada sonraki), yeni değeri hesaplandığı anda saklayarak bu sıranın korunmasını sağlar.

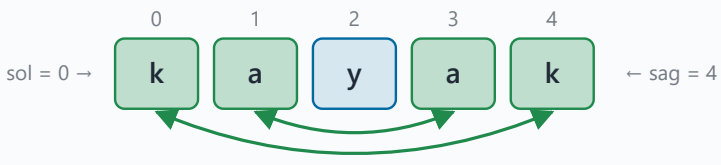
4.1.3.4 Palindrom

Tersten okunuşu düz okunuşuyla aynı olan sözcük ya da cümleye **palindrom** (*palindrome*) denir. “kayak”, “ses”, “kazak” birer palindromdur; “kayık” değildir. Problem, verilen bir **metnin** palindrom olup olmadığını doğru ya da yanlış diye bildirmektir.

İlk akla gelen çözüm, metni ters çevirip kendisiyle karşılaştırmaktır. Bu çözüm doğrudur ama metnin bir kopyasını oluşturur. Daha az iş yapan çözüm, metnin iki ucundan başlayıp ortaya doğru ilerlemektir. İlk harf son harfle, ikinci harf sondan ikinci harfle karşılaştırılır. Farklı bir çift bulunursa metin palindrom değildir ve hemen yanlış döndürülür. Uçlar ortada buluşana kadar farklı çift çıkmazsa metin palindromdur.

```
metni oku
metni kurala göre temizle
sol = 0, sag = uzunluk - 1
sol < sag iken
    karakterler farklıysa yanlış döndür
    sol = sol + 1, sag = sag - 1
doğru döndür
```

İkinci satırdaki “kurala göre temizle” adımı problem tanımının bir parçasıdır. “Kayak” sözcüğünde ilk harf büyük K'dir; harfler olduğu gibi karşılaştırılırsa K ≠ k sanılır ve doğru cevap kaçırılır. Bu yüzden algoritmadan önce bir kural belirlenir. Örnekteki kural şudur: bütün harfler küçük harfe çevrilir, boşluklar ve noktalama işaretleri atılır. “Kayak” böylece “kayak” olur.



Dıştan içe: k = k, a = a; ortadaki y ile karşılaştırma gerekmez → ⌊5 / 2⌋ = 2 karşılaştırma

Şekil 4.1.11: “kayak” sözcüğünde iki uçtan merkeze karşılaştırma: k ile k, a ile a. Ortadaki y kendisiyle karşılaştırılmaz; 5 harf için ⌊5 / 2⌋ = 2 karşılaştırma yeterlidir.

Şekil 4.1.11 karşılaştırılan çiftleri yaylarla gösterir. sol ve sag, karşılaştırılacak iki harfin **indisidir**, yani metindeki sıra numarasıdır. İndisler 0'dan başlar: “kayak” sözcüğünde metin[0] = 'k', metin[4] = 'k'dir. Algoritmanın izlemesi Tablo 4.1.5'tedir.

Karşılaştırma	sol	sag	sol < sag	metin[sol]	metin[sag]	Karar
1	0	4	0 < 4 → doğru	'k'	'k'	aynı, içe ilerle
2	1	3	1 < 3 → doğru	'a'	'a'	aynı, içe ilerle
—	2	2	2 < 2 → yanlış	—	—	döngü biter → doğru döndür

Tablo 4.1.5: Palindrom algoritmasının “Kayak” için izleme tablosu. Temizlenmiş metin “kayak”, uzunluk 5; başlangıçta sol = 0, sag = 4.

Üçüncü denetimde sol ve sag ikisi de 2'dir; ortadaki harfe gelinmiştir. $2 < 2$ yanlış olduğu için döngü biter ve doğru döndürülür. Uzunluğu n olan bir metin için döngü en fazla $\lfloor n / 2 \rfloor$ kez döner. $\lfloor\ \rfloor$ işareti sayının aşağı yuvarlandığını gösterir: $\lfloor 5 / 2 \rfloor = \lfloor 2{,}5 \rfloor = 2$. Çift uzunluktaki bir metinde, örneğin “abba” sözcüğünde, sol ve sag ortada birbirini geçer: 2 karşılaştırmadan sonra sol = 2, sag = 1 olur ve döngü yine biter.

Algoritma palindrom olmayan metinlerde daha da erken durur. “Kayık” için ilk karşılaştırmada k ile k aynıdır. İkinci karşılaştırmada a ile ı farklıdır ve yanlış döndürülür; ortadaki y'ye hiç bakılmaz.

Örnek 4.1.4

“Ey Edip Adana'da pide ye” cümlesi palindrom mudur? Kaç karşılaştırma yapılır?

Önce temizleme kuralı uygulanır: harfler küçültülür, boşluklar ve kesme işareti atılır. Sonuç 19 harflik “eyedipadanadapideye” metnidir. Başlangıçta sol = 0, sag = 18'dir.

Karşılaştırma	sol	sag	sol < sag	metin[sol]	metin[sag]	Karar
1	0	18	0 < 18 → doğru	'e'	'e'	aynı, içe ilerle
2	1	17	1 < 17 → doğru	'y'	'y'	aynı, içe ilerle
3	2	16	2 < 16 → doğru	'e'	'e'	aynı, içe ilerle
4	3	15	3 < 15 → doğru	'd'	'd'	aynı, içe ilerle
5	4	14	4 < 14 → doğru	'i'	'i'	aynı, içe ilerle
6	5	13	5 < 13 → doğru	'p'	'p'	aynı, içe ilerle
7	6	12	6 < 12 → doğru	'a'	'a'	aynı, içe ilerle
8	7	11	7 < 11 → doğru	'd'	'd'	aynı, içe ilerle
9	8	10	8 < 10 → doğru	'a'	'a'	aynı, içe ilerle
—	9	9	9 < 9 → yanlış	—	—	döngü biter → doğru döndür

Dokuz karşılaştırmanın hepsinde harfler aynı çıkar; sol ve sag ortadaki 9 numaralı 'n' harfinde buluşur. Cümle palindromdur ve *⌊19 / 2⌋ = 9* karşılaştırma yapılmıştır. Temizleme yapılmasaydı ilk karşılaştırmada büyük 'E' ile küçük 'e' farklı sayılır ve cevap yanlış çıkardı.

Sık yapılan hata

Türkçe metinlerde büyük harfi küçültürken I ve İ harflerini unutmak. Birçok programlama dilinin küçük harfe çevirme işlemi, dil ayarı belirtilmezse İngilizce kuralını uygular ve "I" harfini "i" yapar. Türkçede ise "I"nın küçüğü "ı", "İ"nin küçüğü "i"dir. "Irmak" sözcüğü İngilizce kuralla "irmak" olur ve palindrom, arama ya da sıralama sonuçları yanlış çıkar. Temizleme kuralı yazılırken metnin dili de göz önüne alınmalıdır.

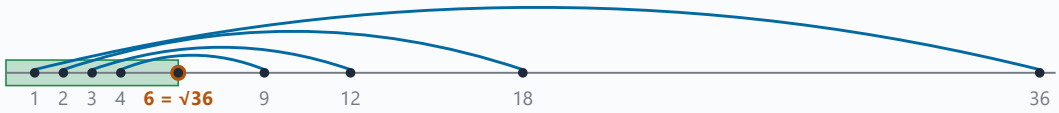
4.1.3.5 Asal sayı

1'den büyük olup 1'den ve kendisinden başka pozitif **böleni** (*divisor*) olmayan sayıya **asal sayı** (*prime number*) denir. 2, 3, 5, 7, 11, 13 asaldır; 9 = 3 × 3 ve 21 = 3 × 7 asal değildir. Problem, verilen bir n sayısının asal olup olmadığını yazmaktır.

Tanım doğrudan bir algoritma verir: 2'den n − 1'e kadar her sayıyla n bölünmeye çalışılır. Kalansız bölen bir sayı bulunursa n asal değildir. Kalan, `%` işlemiyle hesaplanır. *29 % 5 = 4*'tür, çünkü *29 = 5 × 5 + 4*'tür. *n % i = 0* ise i, n'yi tam böler. Bu yöntem doğrudur ama gereğinden fazla çalışır. Sunumdaki algoritma aynı soruyu çok daha az denemeyle cevaplar.

```
oku n
eğer n < 2 ise "asal değil" yaz ve dur
i = 2
i × i ≤ n iken
    eğer n % i == 0 ise "asal değil" yaz ve dur
    i = i + 1
"asal" yaz
```

Algoritmanın iki önemli özelliği vardır. Birincisi, bir bölen bulununca arama hemen durur; geri kalan adaylara bakmaya gerek yoktur. İkincisi, döngü *i × i ≤ n* koşulu sağlandığı sürece, yani i, √n'yi geçmediği sürece döner. Bu **karekök sınırının** nedeni şudur: n, a × b diye iki çarpana ayrılıyorsa a ile b'nin ikisi birden √n'den büyük olamaz. İkisi de büyük olsaydı çarpımları n'den büyük çıkardı. Öyleyse n'nin 1'den büyük bir böleni varsa, √n'den küçük ya da ona eşit bir böleni de vardır.



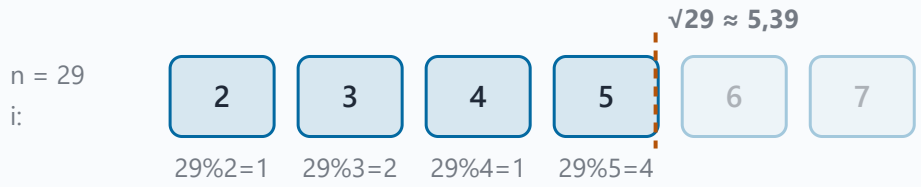
36 = 1×36 = 2×18 = 3×12 = 4×9 = 6×6: her çiftin küçüğü ≤ √36 → i = 2…6 denemek yeter

Şekil 4.1.12: 36'nın bölen çiftleri: 1 × 36, 2 × 18, 3 × 12, 4 × 9, 6 × 6. Her çiftin küçük öğesi √36 = 6'dan küçük ya da ona eşittir; bu yüzden i = 2, …, 6 aralığını denemek yeter.

Şekil 4.1.12 bu gözlemi 36 için gösterir. Her yay bir bölen çiftini bağlar. Yayların sol uçları hep yeşil bölgede, yani 6'nın solunda ya da üstündedir. Koşulun karekök yerine $i \times i \le n$ diye yazılması bilinçli bir seçimdir: kesirli karekök hesabı yerine tam sayı çarpması kullanılır ve yuvarlama hatası olmaz.

i	i × i ≤ n	n % i	Karar
2	4 ≤ 29 → doğru	29 % 2 = 1	bölen değil
3	9 ≤ 29 → doğru	29 % 3 = 2	bölen değil
4	16 ≤ 29 → doğru	29 % 4 = 1	bölen değil
5	25 ≤ 29 → doğru	29 % 5 = 4	bölen değil
6	36 > 29 → yanlış	—	döngü biter → asal

Tablo 4.1.6: Asal sayı algoritmasının n = 29 için izleme tablosu. Her satır bir döngü denetimidir.



Şekil 4.1.13: n = 29 için denenen bölenler. Kesikli çizgi √29 ≈ 5,39 sınırıdır: yalnızca 2, 3, 4 ve 5 denenir, 6 ve sonrası gereksizdir.

Tablo 4.1.6 ve Şekil 4.1.13 aynı çalışmayı iki biçimde gösterir. 29, dört adayın hiçbirine tam bölünmez. i = 6 olduğunda *36 ≤ 29* yanlıştır ve döngü biter. Bölen bulunmadığı için “asal” yazılır. Tanımdan gelen yöntem 2'den 28'e kadar 27 aday denerdi; karekök sınırı bu sayıyı 4'e indirir. n = 21 olsaydı i = 3'te *21 % 3 = 0* bulunur, “asal değil” yazılır ve algoritma iki denemede biterdi.

Örnek 4.1.5

221 asal mıdır? Algoritma kaç kez `%` işlemi yapar?

221 tek sayıdır, rakamlarının toplamı 5 olduğu için 3'e bölünmez ve 5 ile bitmez. İlk bakışta asal sanılabilir. √221 ≈ 14,87 olduğundan algoritma en çok i = 14'e kadar dener.

i	i × i ≤ n	n % i	Karar
2	4 ≤ 221 → doğru	221 % 2 = 1	bölen değil
3	9 ≤ 221 → doğru	221 % 3 = 2	bölen değil
4	16 ≤ 221 → doğru	221 % 4 = 1	bölen değil
5	25 ≤ 221 → doğru	221 % 5 = 1	bölen değil
6	36 ≤ 221 → doğru	221 % 6 = 5	bölen değil
7	49 ≤ 221 → doğru	221 % 7 = 4	bölen değil
8	64 ≤ 221 → doğru	221 % 8 = 5	bölen değil
9	81 ≤ 221 → doğru	221 % 9 = 5	bölen değil
10	100 ≤ 221 → doğru	221 % 10 = 1	bölen değil
11	121 ≤ 221 → doğru	221 % 11 = 1	bölen değil
12	144 ≤ 221 → doğru	221 % 12 = 5	bölen değil
13	169 ≤ 221 → doğru	221 % 13 = 0	bölen bulundu → asal değil

i = 13'te kalan 0 çıkar: *221 = 13 × 17*. Algoritma "asal değil" yazar ve durur. 2'den 13'e kadar 12 aday denenmiş, 12 kez `%` işlemi yapılmıştır. Çarpanlardan küçüğü olan 13, √221 sınırının altında kaldığı için yakalanmıştır; büyüğü olan 17'ye hiç gelinmez.

Sık yapılan hata

Karekök sınırını `i × i < n` diye, eşitliği dışarıda bırakarak yazmak. Bu hata tam karelerde ortaya çıkar. n = 49 için i = 7 olduğunda *49 < 49* yanlıştır; döngü 7'yi denemeden biter. 2'den 6'ya kadar hiçbir sayı 49'u bölmediği için algoritma 49'a yanlışlıkla "asal" der. 4, 9, 25, 49 gibi tam kareler bu sınırı sınayan iyi test girdileridir.

4.1.4 Algoritmayı doğrulama

Bir algoritmanın yazılmış olması doğru çalıştığını göstermez. Doğruluk iki yolla sınanır. Birincisi algoritmayı küçük girdilerle, bilgisayarın yapacağı gibi satır satır çalıştırmaktır. İkincisi, algoritmanın davranışını farklı türden girdilerle denemektir. İki yöntem birbirini tamamlar.

4.1.4.1 Elle izleme ve izleme tablosu

Algoritmayı kâğıt üzerinde, her satırı sırayla uygulayarak çalıştırmaya **elle izleme** (*hand tracing*) denir. İzleme sırasında değişkenlerin değerleri akılda tutulmaz, bir tabloya yazılır. Satırlar zamanı, sütunlar değişkenleri gösterir. Böyle bir tabloya **izleme tablosu** (*trace table*) denir. Önceki kesimdeki bütün tablolar birer izleme tablosudur.

Tanım 4.1.5

İzleme tablosu, bir algoritma elle çalıştırılırken değişkenlerin her adımdaki değerlerinin yazıldığı tablodur. Her sütun bir değişkene ya da denetlenen bir koşula, her satır bir adıma ya da döngü turuna karşılık gelir.

İzleme tablosu hazırlanırken en önemli karar, bir satırın neyi temsil ettiğidir. Değişkenin turun başındaki değeri mi yazılıyor, sonundaki değeri mi? İkisi karıştırılırsa tablo yanıltıcı olur. Bu yüzden tablonun başlığında ya da altında bu karar açıkça belirtilir.

Örnek 4.1.6

Faktöriyel algoritması n = 4 için nasıl izlenir? Tabloda i sütunu turun başındaki, sonuc sütunu turun sonundaki değeri gösterir.

An	i	sonuc	i ≤ n
başlangıç	2	1	—
tur 1	2	2	doğru
tur 2	3	6	doğru
tur 3	4	24	doğru
çıkış	5	24	5 ≤ 4 yanlış

Döngü 3 kez döner ve 24 yazılır; *4! = 24* olduğu için sonuç doğrudur. Tablodaki her satırda döngü değişmezi de denetlenebilir: tur 1'in başında i = 2 iken sonuc 1 = 1!'dir, tur 2'nin başında i = 3 iken sonuc 2 = 2!'dir.

İzleme tablosu yalnızca doğruluğu göstermek için değil, hata bulmak için de kullanılır. Algoritma yanlış sonuç verdiğinde beklenen ve gerçek değerler satır satır karşılaştırılır. İki değerin ilk ayrıldığı satır, hatanın yerini gösterir. Örneğin faktöriyel algoritmasında koşul yanlışlıkla `i < n` yazılmışsa n = 4 için tablo tur 2'ye kadar doğru ilerler. i = 4 olduğunda koşul yanlış çıkar ve tur 3 hiç yapılmaz. Hatanın yeri döngü koşuludur.

Sık yapılan hata

İzleme sırasında algoritmanın yazdığını değil, yapması gerektiğini düşünüleni uygulamak. İzleyen kişi algoritmanın amacını bildiği için gözden kaçan bir satırı farkında olmadan "düzeltir" ve hatayı göremez. Elle izleme, satırlar tam yazıldığı gibi, mekanik olarak uygulandığında işe yarar.

4.1.4.2 Test durumları

İzleme tablosu algoritmanın bir girdi için doğru çalıştığını gösterir. Tek bir başarılı deneme ise algoritmanın her girdi için doğru olduğunu kanıtlamaz. Faktöriyel algoritmasındaki sınır hatası n = 1 ile denendiğinde görünmez; algoritma doğru cevap olan 1'i verir. Bu yüzden algoritma, özenle seçilmiş birkaç girdiyle sınanır. Bir girdi ile o girdi için beklenen çıktının ikilisine **test durumu** (*test case*) denir.

Tanım 4.1.6

Test durumu, bir algoritmaya verilecek girdi ile bu girdi için beklenen çıktıdan oluşan ikilidir. Test durumları dört türe ayrılır: **normal** (yaygın kullanım), **sınır** (kısıtın kenarındaki değerler), **geçersiz** (kısıtın dışındaki değerler; *invalid input*) ve **büyük** (taşma ya da yavaşlık gibi ölçek sorunlarını ortaya çıkaran değerler).

Test durumunda beklenen çıktı, algoritma çalıştırılmadan önce yazılır. Önce algoritmayı çalıştırıp çıkan sonucu "beklenen" sütununa yazmak, testi anlamsız kılar. Beklenen değer problem tanımından ve elle yapılan hesaptan gelmelidir.



Şekil 4.1.14: Faktöriyel girdisi için sayı doğrusu. Negatif değerler geçersizdir (−3 → "hata"); 0 alt sınırdır (0! = 1); 5 normal bir girdidir (120); 0 ile 20 arası geçerlidir; 21 ve üstünde sonuç long tipine sığmaz.

Şekil 4.1.14 faktöriyel girdisinin bölgelerini gösterir. Her bölgeden en az bir test seçilir; bölgelerin sınırları ayrıca denenir, çünkü hatalar çoğunlukla sınırlarda saklanır. Tablo 4.1.7 böyle bir test kümesini verir.

Tür	Amaç	Girdi	Beklenen
Normal	Yaygın kullanımı doğrulamak	n = 5	120 (döngü 4 kez döner)
Sınır	Alt sınırı denemek	n = 0	1 (döngü hiç dönmez)
Sınır	Döngünün hiç dönmediği ikinci durum	n = 1	1 (döngü hiç dönmez)
Sınır	Üst sınırı denemek	n = 20	2.432.902.008.176.640.000 (döngü 19 kez döner)
Geçersiz	Hata davranışını görmek	n = −3	"hata"

Tür	Amaç	Girdi	Beklenen
Büyük	Taşmayı görmek	n = 21	51.090.942.171.709.440.000 → long'a sığmaz; kısıt denetimi "hata" vermeli

Tablo 4.1.7: Faktöriyel algoritması (kısıt 0 ≤ n ≤ 20) için test durumları. Beklenen değerler Python'un sınırsız tam sayılarıyla hesaplanmıştır.

Tablodaki son satır özel bir durumu gösterir. 21! değerinin kendisi doğrudur ama Java'nın 64 bitlik `long` tipi en çok 9.223.372.036.854.775.807 değerini tutabilir. 21! bu sınırı yaklaşık 5,5 kat aşar. Sınırı aşan bir hesapta Java hata vermez; sonuç sessizce başa sarar. Algoritma Java'da `long` ile çalıştırıldığında 21! için −4.249.290.049.419.214.848 gibi negatif ve anlamsız bir sayı çıkar. Bir değerin, tutulduğu veri tipinin sınırını aşmasına **taşma** (*overflow*) denir.

Bu test, algoritmada bir eksiklik olduğunu gösterir: kısıtın üst sınırı hiç denetlenmiyordur. Problem tanımında yazılan "0 ≤ n ≤ 20" kısıtı algoritmaya da yansıtılmalıdır. Düzeltilmiş algoritma, girdiyi işlemeye başlamadan önce iki sınırı birlikte denetler:

```
oku n
eğer n < 0 veya n > 20 ise "hata" yaz ve dur
sonuc = 1
i = 2
i ≤ n iken
    sonuc = sonuc × i
    i = i + 1
yaz sonuc
```

Bu sürümde n = 21 girildiğinde döngüye hiç girilmez ve "hata" yazılır. Görüldüğü gibi test, yalnızca hatayı bulmakla kalmaz, algoritmanın kendisini de iyileştirir.

> **Sık yapılan hata**
>
> Yalnızca normal girdilerle test yapmak. n = 5, n = 6 ve n = 7 için yapılan üç test aynı bölgeyi üç kez dener ve yeni bir şey göstermez. n = 0, n = 20, n = −3 ve n = 21 ise her biri farklı bir soruyu cevaplar. İyi bir test kümesi, testlerin sayısıyla değil, kapsadığı bölgelerle ölçülür.

4.1.5 Algoritma analizi

Bir problem için çoğu zaman birden fazla doğru algoritma vardır. Bunlar arasında seçim yapmak için algoritmaların ne kadar iş yaptığını ölçmek gerekir. Bu ölçmeye **algoritma analizi** denir. Analizde amaç, saniye cinsinden süre ölçmek değildir; o süre bilgisayara, programlama diline ve o anda çalışan başka programlara göre değişir. Amaç, yapılan işin girdinin büyüklüğüyle nasıl değiştiğini bulmaktır.

4.1.5.1 Aynı problem, iki algoritma

Problem, 1'den n'ye kadar olan tam sayıların toplamını bulmaktır: *1 + 2 + … + n*. İlk akla gelen algoritma (A), bir döngüyle sayıları tek tek toplar:

```
toplam = 0
i = 1
while i <= n
    toplam = toplam + i
    i = i + 1
```

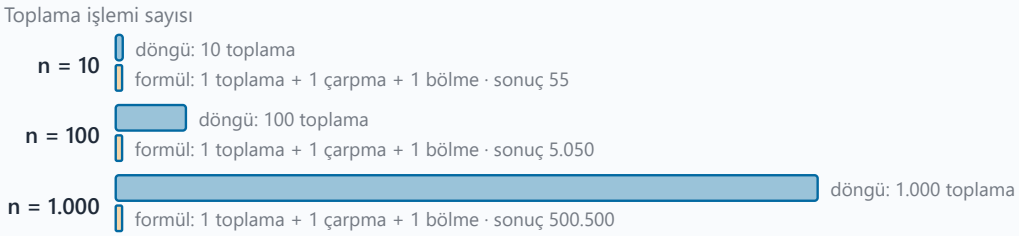
İkinci algoritma (B), Gauss'a atfedilen bir formülü kullanır ve sonucu tek adımda bulur:

```
toplam = n * (n + 1) / 2
```

Formülün nereden geldiği kısa bir hesapla görülür. Toplam iki kez yazılır; biri baştan sona, öteki sondan başa:

> T = 1 + 2 + 3 + … + n
>
> T = n + (n − 1) + (n − 2) + … + 1

Alt alta gelen sayılar toplanırsa her çift *n + 1* eder: *1 + n*, *2 + (n − 1)*, … Toplam n çift vardır. Öyleyse *2T = n × (n + 1)*, yani *T = n × (n + 1) / 2*'dir. n = 10 için *10 × 11 / 2 = 55* bulunur; döngüyle hesaplanan *1 + 2 + … + 10* da 55'tir.



Şekil 4.1.15: İki algoritmanın yaptığı işlem sayısı. n büyüdükçe döngünün toplama sayısı n ile birlikte artar (10, 100, 1.000); formülün işlem sayısı n'den bağımsız olarak sabit kalır.

n	Sonuç	A: döngüdeki toplama sayısı	B: formüldeki işlemler
10	55	10	1 çarpma, 1 toplama, 1 bölme
100	5.050	100	1 çarpma, 1 toplama, 1 bölme
1.000	500.500	1.000	1 çarpma, 1 toplama, 1 bölme

n	Sonuç	A: döngüdeki toplama sayısı	B: formüldeki işlemler
1.000.000	500.000.500.000	1.000.000	1 çarpma, 1 toplama, 1 bölme

Tablo 4.1.8: 1 + 2 + … + n için iki algoritmanın sonucu ve yaptığı işlem sayısı. Sonuçlar iki yöntemle de hesaplanmış ve eşit çıkmıştır.

Şekil 4.1.15 ve Tablo 4.1.8 iki algoritmanın da doğru olduğunu, ama yaptıkları işin çok farklı büyüdüğünü gösterir. A'da döngü n kez döner; her turda bir toplama ve bir sayaç artırma yapılır. n bin katına çıkınca A'nın işi de bin katına çıkar. B ise n ne olursa olsun bir toplama (*n + 1*), bir çarpma ve bir bölme yapar. Bir milyon sayının toplamı için A bir milyon tur döner; B yine üç işlemle biter.

4.1.5.2 Doğruluk ve verimlilik

Toplama örneği, algoritmaları değerlendirirken iki ayrı ölçüt kullanıldığını gösterir. **Doğruluk**, algoritmanın beklenen sonucu üretmesidir. **Verimlilik** ise algoritmanın bu sonuca ulaşırken ne kadar kaynak kullandığıdır. En önemli iki kaynak zaman ve bellektir.

Tanım 4.1.7

Zaman karmaşıklığı (*time complexity*), bir algoritmanın yaptığı temel işlem sayısının **girdi boyutuyla** (*input size*) nasıl değiştiğini anlatır. **Alan karmaşıklığı** (*space complexity*), algoritmanın girdinin kendisi dışında kullandığı ek bellek miktarının girdi boyutuyla nasıl değiştiğini anlatır.

Girdi boyutu, genellikle n harfiyle gösterilir ve problemin doğasına göre seçilir. Bir dizide arama ya da sıralama yapan algoritmada n, dizinin öğe sayısıdır. Faktöriyel ya da toplam hesaplayan algoritmada n, girilen sayının kendisidir. Temel işlem de probleme göre seçilir. Arama ve sıralama algoritmalarında genellikle iki öğenin **karşılaştırma** sayısı sayılır, çünkü işin büyük kısmını karşılaştırmalar oluşturur.

Aynı girdi boyutundaki farklı girdiler, farklı miktarda iş gerektirebilir. Bir dizide aranan değer ilk öğe ise arama hemen biter; son öğe ise bütün diziye bakılır. Bu yüzden üç durum ayrı ayrı ele alınır. **En iyi durum** (*best case*), algoritmanın en az iş yaptığı girdidir. **En kötü durum** (*worst case*), en çok iş yaptığı girdidir. Ortalama durum ise rastgele girdilerde beklenen iş miktarıdır. Uygulamada en çok en kötü durum kullanılır: algoritmanın hiçbir girdide bu sınırdan daha yavaş olmayacağı güvencesini verir.

4.1.5.3 Büyük O gösterimi

İşlem sayısını tam olarak saymak çoğu zaman gereksizdir. 3n + 2 işlem yapan algoritma ile 5n işlem yapan algoritma aynı biçimde büyür: n iki katına çıkınca ikisinin işi de yaklaşık iki katına çıkar. Önemli olan katsayılar değil, büyümenin biçimidir. **Büyük O gösterimi** bu biçimi kısaca yazmanın yoludur.

Tanım 4.1.8

Büyük O gösterimi (*Big O notation*), bir algoritmanın işlem sayısının n büyüdükçe en hızlı büyüyen terimini, katsayısını atarak yazar. 3n + 2 işlem yapan algoritma O(n), *n(n − 1)/2* karşılaştırma yapan algoritma O(n^2) karmaşıklıktadır. n'den bağımsız, sabit sayıda işlem yapan algoritma O(1) karmaşıklıktadır.

Bu notta en sık karşılaşılan karmaşıklıklar şunlardır. **O(1)**, sabit zamandır: Gauss formülü n ne olursa olsun üç işlem yapar. **O(log n)**, logaritmik zamandır: her adımda problem yarıya iner; ikili arama böyle çalışır. **O(n)**, doğrusal zamandır: her öğeye bir kez bakılır; döngüyle toplama ve doğrusal arama böyledir. **O(n log n)**, birleştirme sıralamasının karmaşıklığıdır. **O(n^2)**, karesel zamandır: her öğe öteki öğelerin çoğuyla karşılaştırılır; seçimli, kabarcık ve eklemeli sıralama en kötü durumda böyledir.

$\log_2$ n ifadesi, "2'yi kaçıncı kuvvete yükseltince n bulunur?" sorusunun cevabıdır. $2^{10} = 1.024$ olduğu için $\log_2$ 1.024 = 10'dur. Daha sezgisel bir okuma da vardır: $\log_2$ n, n'yi 1'e inene kadar kaç kez ikiye bölmek gerektiğini söyler. 1.000.000 sayısı 20 kez ikiye bölününce 1'in altına iner. Bu yüzden logaritmik algoritmalar çok büyük girdilerde bile az iş yapar.

n	İkili arama ($\lfloor \log_2 n \rfloor + 1$ tur)	Doğrusal arama (n)	Seçimli sıralama (n(n − 1)/2)	n^2
10	4	10	45	100
100	7	100	4.950	10.000
1.000	10	1.000	499.500	1.000.000
1.000.000	20	1.000.000	499.999.500.000	1.000.000.000.000

Tablo 4.1.9: Girdi boyutu n büyüdükçe farklı karmaşıklıkların işlem sayısı. İkili arama sütunu, sıralı n öğeli dizide bulunamayan bir değer için yapılan tur sayısıdır ve algoritma çalıştırılarak sayılmıştır.

Tablo 4.1.9 farkı sayılarla gösterir. n = 10'da üç sütun arasındaki fark küçüktür; hangi yöntemin seçildiği pek önemli değildir. n = 1.000.000'da ikili arama 20 turda biter, doğrusal arama bir milyon karşılaştırma yapar. Seçimli sıralama ise yaklaşık beş yüz milyar karşılaştırma yapar. Saniyede bir milyar karşılaştırma yapan bir bilgisayar, ikili aramayı bir saniyenin milyonda birinden kısa sürede, seçimli sıralamayı ise yaklaşık sekiz dakikada bitirir.

Verimlilik her zaman tek ölçüt değildir. Küçük veride en hızlı görünen yöntem her zaman en anlaşılır yöntem olmayabilir. On öğelik bir listeyi sıralamak için karmaşık bir algoritma yazmak yerine anlaşılması ve doğrulanması kolay basit bir yöntem seçmek çoğu zaman daha akıllıcadır. Girdi büyüdükçe denge verimlilik lehine döner.

Sık yapılan hata

Büyük O gösterimini kesin süre sanmak. O(n) bir algoritmanın her zaman O(n^2) bir algoritmadan hızlı olduğu söylenemez. Gösterim katsayıları atar: 1.000n işlem yapan bir algoritma O(n)'dir ama n = 100 için 100.000 işlem yapar. n^2 işlem yapan algoritma aynı n için 10.000 iş-

lemle biter. Büyük O, n yeterince büyüdüğünde hangi algoritmanın öne geçeceğini söyler.

4.1.6 Arama

Bir veri topluluğunda belirli bir değerin olup olmadığını ve varsa nerede olduğunu bulma işine **arama** (*search*) denir. Rehberde bir telefon numarasını, bir öğrenci listesinde bir numarayı, bir sözlükte bir kelimeyi bulmak birer aramadır. Aranan değere **hedef** (*target*) denir. Buradaki algoritmalar bir tam sayı dizisinde hedefi arar. Hedef bulunursa onun indisini, bulunamazsa −1 döndürür. −1 hiçbir zaman geçerli bir indis olmadığı için "yok" anlamına gelen güvenli bir işarettir.

Arama algoritmaları Java ile yazılmıştır. Java'da `int[] dizi` bir tam sayı dizisidir; `dizi[i]`, i numaralı öğedir ve `dizi.length` öğe sayısını verir. İndisler 0'dan başlar; n öğeli bir dizinin son indisi $n - 1$'dir. `return` deyimi metodu hemen bitirir ve yanındaki değeri, metodu çağıran yere geri gönderir.

4.1.6.1 Doğrusal arama

En basit arama yöntemi, öğelere baştan sona sırayla bakmaktır. Her öğe hedefle karşılaştırılır; eşleşme bulunursa arama biter. Kitaplıkta sırasız dizilmiş kitaplar arasında bir kitabı bulmak için rafları tek tek taramak bu yönteme benzer.

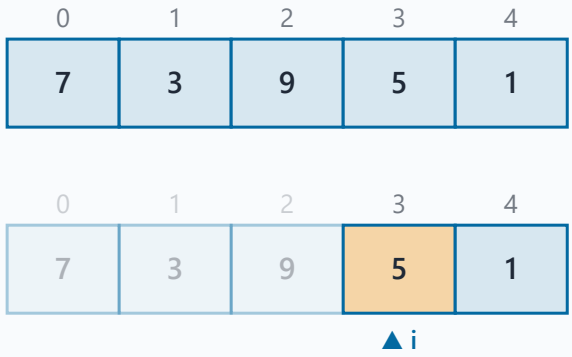
Tanım 4.1.9

Doğrusal arama (*linear search*), dizinin öğelerini ilk öğeden başlayarak sırayla hedefle karşılaştıran ve ilk eşleşmede o öğenin indisini döndüren arama yöntemidir. Hiçbir öğe eşleşmezse −1 döndürülür.

```
int ara(int[] dizi, int hedef) {
    for (int i = 0; i < dizi.length; i++) {
        if (dizi[i] == hedef) return i;
    }
    return -1;
}
```

Metot iki parametre alır: aranacak dizi ve hedef. `for` döngüsü i'yi 0'dan *dizi.length − 1*'e kadar artırır. Her turda `dizi[i] == hedef` karşılaştırması yapılır. Eşitse `return i` metodu hemen bitirir; döngünün geri kalanı çalışmaz. Döngü hiç eşleşme bulmadan biterse son satıra gelinir ve −1 döndürülür.

Doğrusal aramanın iki önemli özelliği vardır. Birincisi, dizinin **sıralı** (*sorted*) olması gerekmez; öğeler hangi düzende olursa olsun hepsine bakılır. İkincisi, ilk eşleşmede durur. Hedef dizide birden çok kez geçiyorsa en küçük indis döndürülür.



Şekil 4.1.16: {7, 3, 9, 5, 1} dizisinde 5'i doğrusal aramak. Solda başlangıç durumu; sağda i = 3'te 5 bulunmuştur. Soluk hücreler hedefle karşılaştırılıp eşleşmeyen öğelerdir.

i	i < dizi.length	dizi[i]	dizi[i] == hedef	Sonuç
0	0 < 5 → doğru	7	7 == 5 → yanlış	sonraki öğe
1	1 < 5 → doğru	3	3 == 5 → yanlış	sonraki öğe
2	2 < 5 → doğru	9	9 == 5 → yanlış	sonraki öğe
3	3 < 5 → doğru	5	5 == 5 → doğru	return 3

Tablo 4.1.10: `ara(dizi, 5)` çağrısının izleme tablosu; dizi = {7, 3, 9, 5, 1}.

Tablo 4.1.10'da görüldüğü gibi 5, dördüncü öğede bulunur ve metot 3 döndürür; 4 karşılaştırma yapılmıştır. Hedef 4 olsaydı beş öğenin hepsi denenir, beşinin de eşleşmediği görülür ve 5 karşılaştırmadan sonra −1 döndürülürdü. JDK 21 ile derlenip çalıştırılan metot da `ara(dizi, 5)` için 3, `ara(dizi, 4)` için −1 verir.

Doğrusal aramanın maliyeti hedefin yerine bağlıdır. En iyi durumda hedef ilk öğedir ve tek karşılaştırma yeter. En kötü durumda hedef son öğedir ya da dizide hiç yoktur; n karşılaştırma yapılır. Bu yüzden doğrusal aramanın zaman karmaşıklığı O(n)'dir. Ek bellek olarak yalnızca i değişkeni kullanılır; alan karmaşıklığı O(1)'dir.

Sık yapılan hata

"Bulunamadı" sonucunu döngünün içinde vermek. Döngünün içine `else return -1;` eklenirse metot ilk öğe hedef değilse hemen −1 döndürür ve geri kalan öğelere hiç bakmaz. −1, ancak bütün öğeler denendikten sonra, yani döngü bittikten sonra döndürülebilir.

4.1.6.2 İkili arama

Telefon rehberinde bir isim aranırken rehber ilk sayfadan başlanıp tek tek çevrilmez. Ortalarda bir sayfa açılır. Aranan isim o sayfadakinden alfabede önce geliyorsa ön yarıya, sonra geliyorsa arka yarıya geçilir. Her bakışta rehberin yarısı elenir. Bu yöntem rehberin alfabetik sırada olmasına dayanır. Bilgisayardaki karşılığı ikili aramadır.

Tanım 4.1.10

İkili arama (*binary search*), sıralı bir dizide hedefi **arama aralığının** (*search range*) ortasındaki öğeyle karşılaştıran ve her adımda aralığın, hedefin bulunamayacağı yarısını eleyen arama yöntemidir. Aralık boşalırsa hedef dizide yoktur.

```
int ikiliAra(int[] dizi, int hedef) {
    int sol = 0, sag = dizi.length - 1;
    while (sol <= sag) {
        int orta = sol + (sag - sol) / 2;
        if (dizi[orta] == hedef) return orta;
        if (dizi[orta] < hedef) sol = orta + 1;
        else sag = orta - 1;
    }
    return -1;
}
```

Metodun satırları şöyle çalışır:

1. `int sol = 0, sag = dizi.length - 1;` : arama aralığı başta bütün dizidir. sol aralığın ilk, sag son indisidir.
2. `while (sol <= sag)` : aralıkta en az bir öğe kaldığı sürece döngü sürer. sol, sag'ı geçerse aralık boştur.
3. `int orta = sol + (sag - sol) / 2;` : aralığın ortasındaki indis hesaplanır. Java'da iki tam sayının bölümü tam sayıdır; kesirli kısım atılır: *7 / 2 = 3*.
4. `if (dizi[orta] == hedef) return orta;` : ortadaki öğe hedefse arama biter.
5. `if (dizi[orta] < hedef) sol = orta + 1;` : ortadaki öğe hedeften küçükse, dizi sıralı olduğu için ortanın solundaki bütün öğeler de hedeften küçüktür. Sol yarı ortayla birlikte elenir.
6. `else sag = orta - 1;` : aksi hâlde ortadaki öğe hedeften büyüktür; sağ yarı ortayla birlikte elenir.
7. `return -1;` : aralık boşaldıysa hedef dizide yoktur.

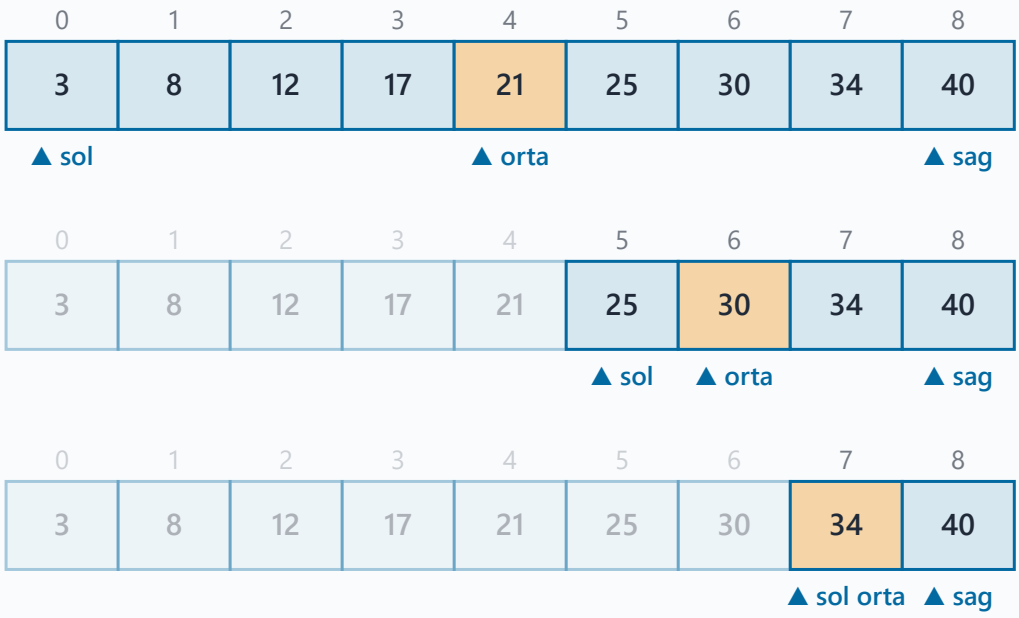
6. ve 7. satırdaki eleme kararı, algoritmanın kalbidir ve tamamen sıralılığa dayanır. Sıralı bir dizide *dizi[orta] < hedef* ise *dizi[0], ..., dizi[orta]* öğelerinin hepsi hedeften küçüktür; hiçbiri hedef olamaz. Bu yüzden ortanın kendisi de elenir: sol, orta değil *orta + 1* olur.

Orta indisin `(sol + sag) / 2` yerine `sol + (sag - sol) / 2` diye hesaplanmasının bir nedeni vardır. İki ifade matematikte aynı sonucu verir. Bilgisayarda ise `int` tipi en çok 2.147.483.647 değerini tutar. Çok büyük bir dizide sol = 1.500.000.000 ve sag = 2.000.000.000 olsaydı `sol + sag` bu sınırı aşar ve taşma olurdu. JDK 21 ile denendiğinde `(sol + sag) / 2` ifadesi −397.483.648 gibi negatif bir indis verir. `sol + (sag - sol) / 2` ise doğru sonuç olan 1.750.000.000'u bulur, çünkü önce iki indisin farkı alınır ve fark hiçbir zaman sınırı aşmaz.

Sunumdaki örnekte dizi dokuz öğelidir ve hedef 34'tür. Algoritmanın her turu Tablo 4.1.11'de ve Şekil 4.1.17'de gösterilmiştir.

Tur	sol	sag	orta	dizi[orta]	Karar
1	0	8	0 + (8 − 0) / 2 = 4	21	21 < 34 → sol = 5
2	5	8	5 + (8 − 5) / 2 = 6	30	30 < 34 → sol = 7
3	7	8	7 + (8 − 7) / 2 = 7	34	34 == 34 → return 7

Tablo 4.1.11: `ikiliAra(dizi, 34)` çağrısının izleme tablosu; dizi = {3, 8, 12, 17, 21, 25, 30, 34, 40}.



Şekil 4.1.17: İkili aramada 34'ün bulunması, üç tur (sırasıyla). 1. tur: aralık 0…8, orta = 4, dizi[4] = 21. 2. tur: aralık 5…8, orta = 6, dizi[6] = 30. 3. tur: aralık 7…8, orta = 7, dizi[7] = 34; hedef bulunur. Soluk hücreler elenmiş öğelerdir.

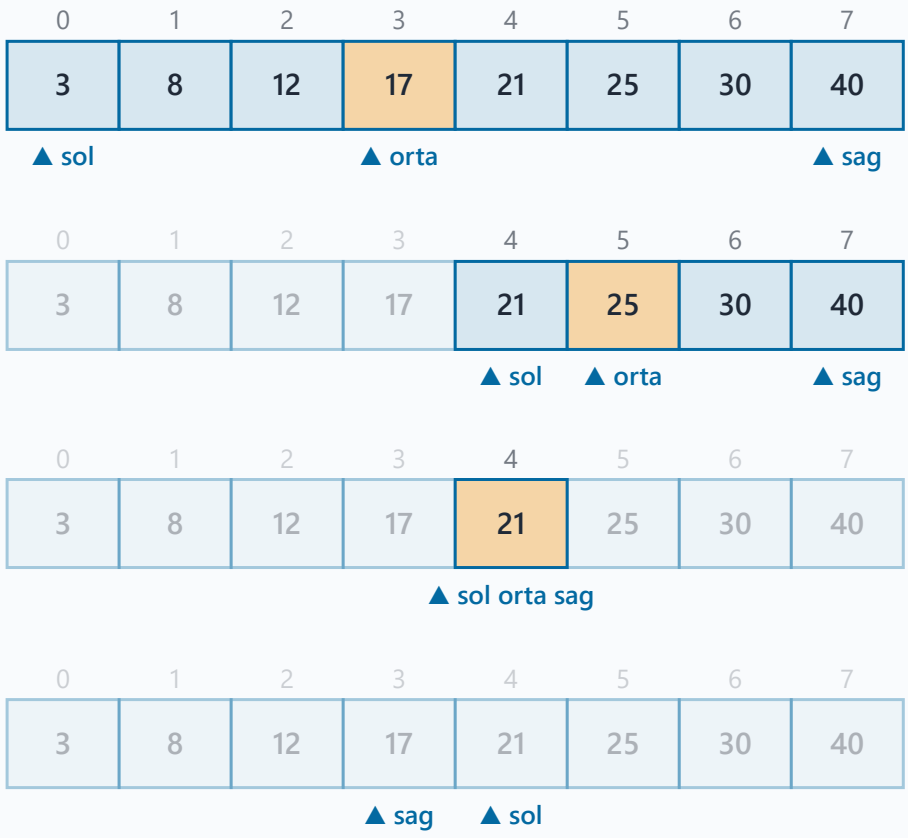
İlk turda ortadaki 21, 34'ten küçüktür; 21 ve solundaki dört öğe elenir. İkinci turda kalan dört öğenin ortası 30'dur ve yine hedeften küçüktür; 30 ve solundaki 25 elenir. Üçüncü turda iki öğe kalmıştır. *7 + (8 − 7) / 2 = 7*'dir ve dizi[7] = 34 bulunur. İkili arama hedefi 3 turda bulmuştur. Doğrusal arama aynı dizide 34'e ulaşmak için 8 karşılaştırma yapardı.

4.1.6.3 İkili arama: hedef bulunamadığında

İkili aramanın en kolay yanlış anlaşılan yanı, hedef dizide olmadığında nasıl durduğudur. Bunu görmek için sekiz öğeli {3, 8, 12, 17, 21, 25, 30, 40} dizisinde 19 aranır. 19 dizide yoktur; olsaydı 17 ile 21 arasında, 4 numaralı konumda olurdu.

Tur	sol	sag	orta	dizi[orta]	Karar
1	0	7	0 + (7 − 0) / 2 = 3	17	17 < 19 → sol = 4
2	4	7	4 + (7 − 4) / 2 = 5	25	25 > 19 → sag = 4
3	4	4	4 + (4 − 4) / 2 = 4	21	21 > 19 → sag = 3
—	4	3	—	—	4 ≤ 3 → yanlış: aralık boş, return −1

Tablo 4.1.12: `ikiliAra(dizi, 19)` çağrısının izleme tablosu; dizi = {3, 8, 12, 17, 21, 25, 30, 40}.



Şekil 4.1.18: İkili aramada 19'un bulunamaması. 1. tur: orta = 3, dizi[3] = 17 < 19, sol yarı elenir. 2. tur: orta = 5, dizi[5] = 25 > 19, sağ yarı elenir. 3. tur: aralık tek öğe (4), dizi[4] = 21 > 19, sag = 3 olur. Son durumda sag, sol'un solundadır; aralık boştur.

Üçüncü turdan sonra sol = 4, sag = 3'tür. sol ile sag yer değiştirmiş, yani sol, sag'ı geçmiştir. *4 ≤ 3* koşulu yanlış olduğu için döngü biter ve −1 döndürülür. Algoritma 3 turda "yok" cevabını vermiştir; sekiz öğenin hepsine tek tek bakılmamıştır. Döngünün bittiği anda sol'un değeri de anlamlıdır: 4, hedefin dizide olsaydı bulunacağı konumdur.

Java'nın kendi kütüphanesi bu bilgiyi kullanır. `java.util.Arrays.binarySearch` metodu, hedef bulunamadığında −1 yerine *−(ekleme noktası) − 1* değerini döndürür. Aynı dizide 19 için JDK 21'in döndürdüğü değer −5'tir: *−4 − 1 = −5*. Değer negatif olduğu için "bulunamadı" anlamı korunur; ayrıca hedefin hangi konuma eklenmesi gerektiği de öğrenilir.

Örnek 4.1.7

İkili arama sıralı olmayan {7, 3, 9, 5, 1} dizisinde 5 için çalıştırılırsa ne olur?

1. tur: sol = 0, sag = 4, orta = 2, dizi[2] = 9. 9 > 5 olduğu için sağ yarı elenir ve sag = 1 olur. 5, aslında elenen tarafta, 3 numaralı konumdadır. 2. tur: sol = 0, sag = 1, orta = 0, dizi[0] = 7. 7 > 5 olduğu için sag = −1 olur. *0 ≤ −1* yanlıştır ve −1 döndürülür. 5 dizide olduğu hâlde bulunamamıştır. JDK 21 ile çalıştırılan `ikiliAra` da bu dizi için −1 verir.

Algoritma hata vermez ve sonsuz döngüye de girmez; yalnızca yanlış cevap verir. Bu, hatayı daha tehlikeli kılar. Yarıyı elemek ancak dizi sıralıysa güvenlidir.

İkili aramanın maliyeti tur sayısıyla ölçülür. Her turda aralıktaki öğe sayısı yarıdan biraz daha aza iner, çünkü orta da elenir. n öğeli bir dizide aralık en çok $\lfloor \log_2 n \rfloor + 1$ turda boşalır. Sekiz öğeli dizide $\lfloor \log_2 8 \rfloor + 1 = 3 + 1 = 4$ tur sınırı vardır; 19 araması 3 turda bitmiştir. Bir milyon öğeli dizide sınır 20 turdur. Bu yüzden ikili aramanın zaman karmaşıklığı O(log n)'dir. Ek bellek gereksinimi sol, sag ve orta değişkenlerinden ibarettir; alan karmaşıklığı O(1)'dir.

Sık yapılan hata

Döngü koşulunu `sol < sag` diye yazmak ya da sınırları `sol = orta`, `sag = orta` diye güncellemek. Birinci hata, aralıkta tek öğe kaldığında (sol = sag) o öğeye hiç bakılmamasına yol açar; 9 öğeli dizide 3 araması sonunda böyle bir durum oluşur ve hedef kaçırılır. İkinci hata, orta elenmediği için aralığın bazen hiç küçülmemesine ve döngünün sonsuza dek sürmesine yol açar. Doğru biçim `sol <= sag` koşulu ile `orta + 1` ve `orta - 1` güncellemeleridir.

4.1.6.4 Arama yöntemlerinin karşılaştırılması

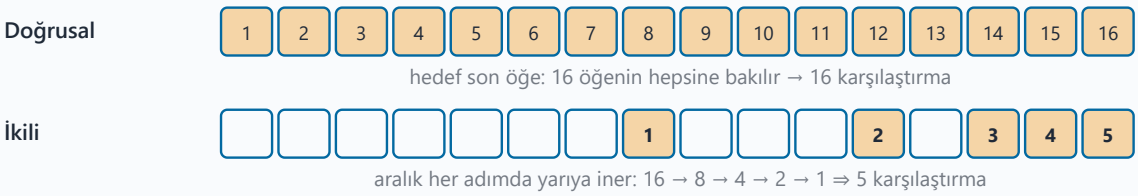
İki arama yöntemi farklı koşullar için uygundur. Tablo 4.1.13 farkları özetler.

Özellik	Doğrusal arama	İkili arama
Sıralı veri	Gerekmez	Gerekir
En iyi durum	1 karşılaştırma (hedef ilk öğe)	1 tur (hedef ortada)
En kötü zaman	O(n)	O(log n)
Ek bellek	O(1)	O(1)
Yapı	Dizi ya da baştan sona okunan veri akışı	Rastgele erişimli sıralı yapı

Özellik	Doğrusal arama	İkili arama
Uygun kullanım	Az ya da sırasız veri, tek arama	Aynı veride çok sayıda arama

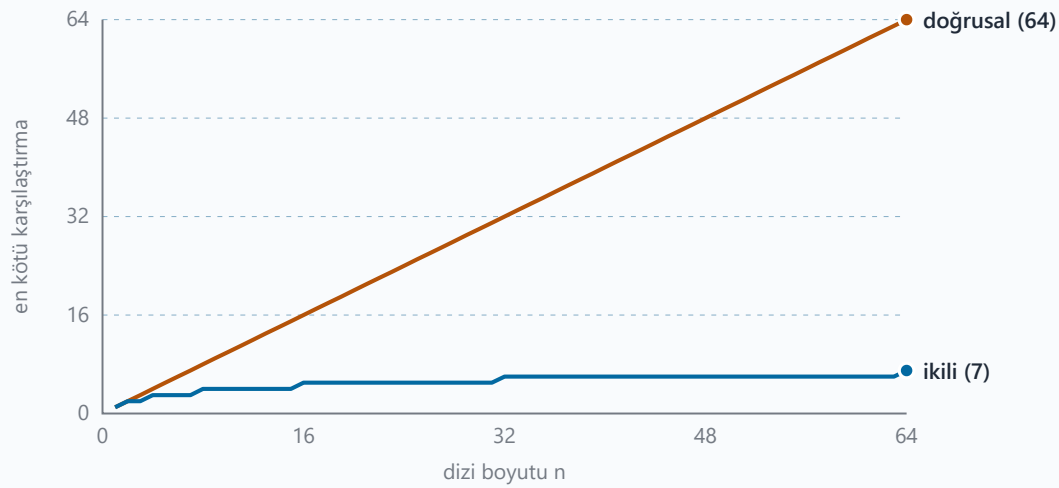
Tablo 4.1.13: Doğrusal ve ikili aramanın karşılaştırması.

Tablodaki "yapı" satırı bir ayrıntıya dikkat çeker. İkili arama, her turda dizinin ortasındaki öğeye doğrudan ulaşabilmeye dayanır. Bir öğeye indisiyle, öncekilerden geçmeden tek adımda ulaşmaya **rastgele erişim** (*random access*) denir. Diziler rastgele erişimlidir. Bir dosyadan ya da ağdan sırayla gelen veride ise ortadaki öğeye ulaşmak için önceki bütün öğelerin okunması gerekir; bu tür verilerde doğrusal arama tek seçenektir.



Şekil 4.1.19: 16 öğeli sıralı dizide son öğeyi aramak. Doğrusal arama 16 öğenin hepsine bakar. İkili aramanın baktığı öğeler sırasıyla numaralanmıştır: aralık 16 → 8 → 4 → 2 → 1 diye daralır ve 5 karşılaştırma yeter.

Şekil 4.1.19 iki yöntemin aynı hedef için baktığı öğeleri gösterir. Dizi 16 öğeli ve hedef son öğe olduğunda ikili arama sırasıyla 8., 12., 14., 15. ve 16. öğelere bakar. Bu küçük örnekte bile fark üç kattan fazladır. Fark, dizi büyüdükçe hızla açılır.



Şekil 4.1.20: Bulunamayan bir hedef için en kötü durumdaki karşılaştırma sayısı. Doğrusal arama n karşılaştırma yapar (n = 64'te 64); ikili arama $\lfloor \log_2 n \rfloor + 1$ tur yapar (n = 64'te 7). Değerler iki algoritma gerçekten çalıştırılarak sayılmıştır.

Şekil 4.1.20 farkın nasıl büyüdüğünü çizer. Doğrusal aramanın eğrisi düz bir çizgidir; n iki katına çıkınca karşılaştırma sayısı da iki katına çıkar. İkili aramanın eğrisi ise basamak basamak ve çok yavaş yükselir: n iki katına çıkınca tur sayısı yalnızca bir artar. n = 1.000.000 için doğrusal arama bir milyon, ikili arama yalnızca 20 karşılaştırma yapar.

Bu üstünlüğün bir bedeli vardır: ikili arama sıralı veri ister. Veri sıralı değilse önce sıralanması gerekir. Sıralama, aramanın kendisinden çok daha pahalı bir iştir; en iyi genel amaçlı yöntemler bile O(n log n) zaman alır. Tek bir arama için önce sıralayıp sonra ikili arama yapmak, doğrudan doğrusal arama yapmaktan daha pahalıdır. Aynı veride binlerce arama yapılacaksa ise bir kez sıralamak kendini hızla öder.

Örnek 4.1.8

1.000 öğeli sırasız bir dizide 500 arama yapılacak. Doğrusal arama ile "önce seçimli sıralama, sonra ikili arama" yolunun en kötü durumdaki karşılaştırma sayıları nedir?

Doğrusal arama her aramada en çok 1.000 karşılaştırma yapar: $500 \times 1.000 = 500.000$. Seçimli sıralama 1.000 öğe için $1.000 \times 999 / 2 = 499.500$ karşılaştırma yapar. Ardından her ikili arama en çok $\lfloor \log_2 1.000 \rfloor + 1 = 10$ tur sürer: $500 \times 10 = 5.000$. Toplam $499.500 + 5.000 = 504.500$ eder. Bu sayıda aramada iki yol neredeyse eşittir. Arama sayısı artarsa sıralama yolu öne geçer; daha hızlı bir sıralama yöntemi seçilirse fark çok daha erken açılır.

4.1.7 Sıralama

Bir veri topluluğunun öğelerini belirli bir düzene, örneğin küçükten büyüğe dizmeye **sıralama** (*sorting*) denir. Sıralama, bilgisayarların en çok yaptığı işlerden biridir. Sınav sonuçları puana göre, e-postalar tarihe göre, ürünler fiyata göre sıralanır. Sıralamanın bir başka değeri de sonraki işleri kolaylaştırmasıdır: ikili arama yalnızca sıralı veride çalışır.

Burada dört yöntem karşılaştırılır. İlk üçü, seçimli, kabarcık ve eklemeli sıralama, kolay anlaşılır ve elle izlenebilir. Üçü de en kötü durumda $O(n^2)$ karşılaştırma yapar. Dördüncüsü olan birleştirme sıralaması farklı bir fikre dayanır ve büyük dizilerde çok daha hızlıdır. Bütün yöntemler diziyi küçükten büyüğe sıralar. Dizide yan yana duran iki öğeye kısaca **komşu** (*adjacent*) öğeler denir.

4.1.7.1 Seçimli sıralama

Bir kişi elindeki kâğıtları sıralarken önce en küçüğünü bulup en başa koyabilir, sonra kalanlar arasında en küçüğü bulup ikinci sıraya koyabilir. Bu doğal yöntemin adı seçimli sıralamadır.

Tanım 4.1.11

Seçimli sıralama (*selection sort*), sıralanmamış bölümdeki en küçük öğeyi bulup bölümün ilk konumundaki öğeyle yer değiştiren sıralama yöntemidir. Bu adım, sıralanmamış bölümde tek öğe kalana kadar tekrarlanır.

```
i = 0'dan n-2'ye kadar
    enK = i
    j = i+1'den n-1'e kadar
        eğer dizi[j] < dizi[enK] ise enK = j
    dizi[i] ile dizi[enK] yer değiştir
```

Dış döngünün her turuna bir **geçiş** (*pass*) denir. i'nci geçişin başında dizinin 0…i − 1 bölümü sıralıdır ve en küçük i öğeyi tutar; i…n − 1 bölümü sıralanmamıştır. `enK` değişkeni, o ana kadar bulunan en küçük öğenin indisidir. Başta sıralanmamış bölümün ilk öğesi en küçük kabul edilir. İç döngü bölümün geri kalan öğelerini tek tek enK'deki öğeyle karşılaştırır; daha küçüğü bulununca enK güncellenir. İç döngü bitince en küçük öğe, i konumundaki öğeyle yer değiştirir ve sıralı bölüm bir öğe büyür.



Şekil 4.1.21: Seçimli sıralamanın ilk geçişi: {29, 10, 14, 37, 13} dizisinde en küçük öğe 10'dur ve 0 konumundaki 29 ile yer değiştirir. Geçiş sonunda 10 sıralı bölümdedir; geri kalan dört öğe sıralanmamıştır.

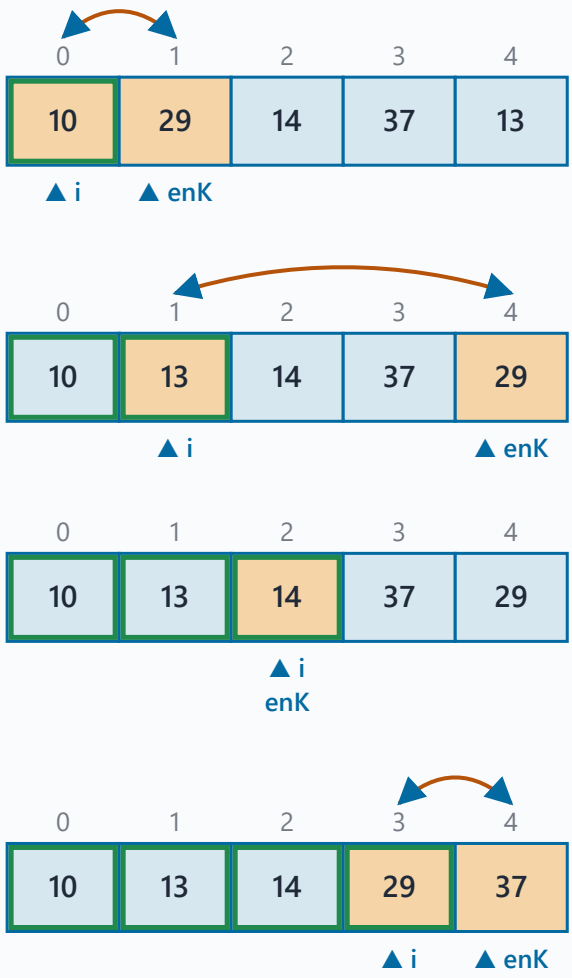
Sunumdaki örnek dizi {29, 10, 14, 37, 13}'tür. Her geçişin sonucu Tablo 4.1.14'te verilmiştir.

Geçiş	Sıralanmamış bölüm	Karşılaştırma	enK'nın aldığı yeni değerler	Takas	Geçiş sonunda dizi
i = 0	0…4	4	10 (dizi[1])	dizi[0] ↔ dizi[1]	[10, 29, 14, 37, 13]
i = 1	1…4	3	14 (dizi[2]), 13 (dizi[4])	dizi[1] ↔ dizi[4]	[10, 13, 14, 37, 29]
i = 2	2…4	2	değişmez	yerinde (etkisiz)	[10, 13, 14, 37, 29]
i = 3	3…4	1	29 (dizi[4])	dizi[3] ↔ dizi[4]	[10, 13, 14, 29, 37]

Tablo 4.1.14: Seçimli sıralamanın {29, 10, 14, 37, 13} için izleme tablosu. "enK'nın aldığı yeni değerler" sütunu iç döngüde daha küçük bulunan öğeleri sırasıyla gösterir.

Tablonun satırları şöyle okunur. i = 0'da aday en küçük 29'dur. dizi[1] = 10 daha küçüktür ve enK = 1 olur; 14, 37 ve 13, 10'dan küçük değildir. 10 başa taşınır. i = 1'de aday 29'dur. Önce 14 bulunur (enK = 2), sonra 13 daha da küçüktür (enK = 4). 13 ile 29 yer değiştirir. i = 2'de aday 14'tür ve kalan 37 ile

29'dan küçüktür; en küçük zaten yerindedir. Algoritma bu geçişte dizi[2]'yi kendisiyle "yer değiştirir"; işlem etkisizdir. i = 3'te 29 ile 37 yer değiştirir ve dizi sıralanır.



Şekil 4.1.22: Seçimli sıralamanın dört geçişinin sonu. i = 0: dizi[0] ile dizi[1] yer değiştirir. i = 1: dizi[1] ile dizi[4] yer değiştirir. i = 2: en küçük zaten yerindedir (enK = i). i = 3: dizi[3] ile dizi[4] yer değiştirir. Yeşil çerçeveli hücreler sıralı bölümdür.

Karşılaştırma sayısı dizinin içeriğine hiç bağlı değildir. i = 0'da iç döngü $n - 1$, i = 1'de $n - 2$, …, son geçişte 1 karşılaştırma yapar. Toplam:

> $(n - 1) + (n - 2) + \ldots + 1 = n(n - 1) / 2$

Bu, Gauss formülünün n yerine $n - 1$ konmuş hâlidir. n = 5 için $4 + 3 + 2 + 1 = 10$ karşılaştırma eder; sunumdaki izlemede de 10 karşılaştırma sayılmıştır. Dizi zaten sıralı olsa bile algoritma her geçişte en küçüğü yeniden arar ve yine 10 karşılaştırma yapar. Bu yüzden seçimli sıralama en iyi durumda da en kötü durumda da $O(n^2)$'dir. Buna karşılık takas sayısı azdır: her geçişte en çok bir takas yapılır, toplam en çok $n - 1$. Yalnızca birkaç yardımcı değişken gerekir; alan karmaşıklığı O(1)'dir.

Sık yapılan hata

İç döngüde en küçük öğenin indisi yerine değerini tutmak. Değer bilinir ama dizideki yeri bilinmezse takas yapılamaz. `enK` bir indistir; karşılaştırma `dizi[j] < dizi[enK]` biçiminde yapılır. Bir başka hata da takası iç döngünün içine koymaktır: bu durumda daha küçük bir öğe her bulunduğunda takas yapılır, takas sayısı gereksiz yere artar.

4.1.7.2 Kabarcık sıralaması

Kabarcık sıralaması yalnızca yan yana duran öğeleri karşılaştırır. Bir geçişte dizi soldan sağa taranır ve sırası yanlış olan her komşu çift yer değiştirir. Büyük değerler, sudaki kabarcıklar gibi her geçişte sağa doğru yükselir.

Tanım 4.1.12

Kabarcık sıralaması (*bubble sort*), komşu öğeleri soldan sağa karşılaştırıp yanlış sıradaki her çifti yer değiştiren sıralama yöntemidir. Geçişler, bir geçişte hiç değişim olmayana kadar tekrarlanır.

```
tur = 0
tekrarla
    degisti = yanlış
    j = 0'dan n-2-tur'a kadar
        eğer dizi[j] > dizi[j+1] ise
            dizi[j] ile dizi[j+1] yer değiştir
            degisti = doğru
    tur = tur + 1
degisti yanlış olana kadar
```

Bir geçişin sonunda, kalan öğelerin en büyüğü kesinlikle en sağa ulaşmıştır. Bunun nedeni şudur: en büyük öğe hangi konumda olursa olsun, sağındaki komşusuyla her karşılaştırmada kazanır ve bir sağa kayar. Bu yüzden her geçiş bir öncekinden bir öğe kısadır. `tur` değişkeni tamamlanan geçiş sayısını tutar ve iç döngünün sınırını $n - 2 - tur$ olarak kısaltır.

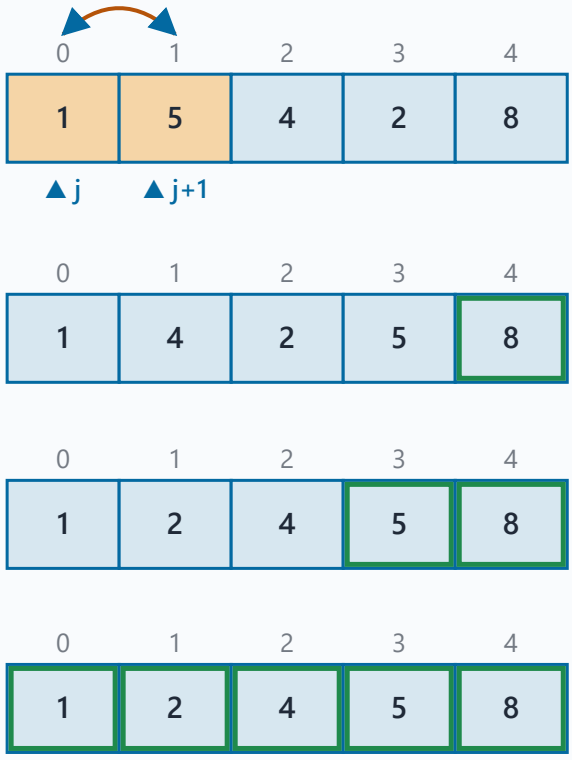
`degisti` değişkeni algoritmanın **erken durma** özelliğini sağlar. Her geçişin başında yanlış yapılır; geçişte bir takas olursa doğru olur. Bir geçişte hiç takas olmamışsa bütün komşu çiftler doğru sıradadır, yani dizi sıralıdır. Bu durumda geçiş sürdürülmez, algoritma durur. Döngünün “tekrarla … olana kadar” biçiminde yazılması, koşulun gövdeden sonra denetlendiğini gösterir: gövde en az bir kez çalışır.

Geçiş	j	Karşılaştırılan çiftler (↔ takas)	Karşılaştırma	Geçiş sonunda dizi	degisti
1. geçiş	0…3	(5, 1)↔ (5, 4)↔ (5, 2)↔ (5, 8)	4	[1, 4, 2, 5, 8]	doğru

Geçiş	j	Karşılaştırılan çiftler (↔ takas)	Karşılaştırma	Geçiş sonunda dizi	degisti
2. geçiş	0…2	(1, 4) (4, 2)↔ (4, 5)	3	[1, 2, 4, 5, 8]	doğru
3. geçiş	0…1	(1, 2) (2, 4)	2	[1, 2, 4, 5, 8]	yanlış

Tablo 4.1.15: Kabarcık sıralamasının {5, 1, 4, 2, 8} için izleme tablosu. ↔ işareti, o çiftin yer değiştirdiğini gösterir.

Tablo 4.1.15 sunumdaki {5, 1, 4, 2, 8} dizisinin üç geçişini gösterir. 1. geçişte 5, sırasıyla 1, 4 ve 2 ile yer değiştirerek 3 numaralı konuma ilerler. 8 ile karşılaştırıldığında sıra doğrudur; 8 zaten en büyüktür ve en sağdadır. Geçiş sonunda 8 kesin yerindedir. 2. geçişte yalnızca 4 ile 2 yer değiştirir ve dizi sıralanır. Bunu algoritma henüz bilmez: degisti doğrudur, bir geçiş daha yapılır. 3. geçişte hiç takas olmaz; degisti yanlış kalır ve algoritma durur. Toplam *4 + 3 + 2 = 9* karşılaştırma yapılmıştır.



Şekil 4.1.23: Kabarcık sıralamasından dört an: 1. geçişte ilk takas (5 ile 1); 1. geçiş sonu (8 yerinde); 2. geçiş sonu (5 de yerinde); 3. geçiş sonu (hiç takas olmadı, dizi sıralı). Yeşil çerçeveli hücreler kesin yerine oturmuş öğelerdir.

Kabarcık sıralamasının maliyeti girdiye bağlıdır. En kötü durumda, dizi ters sıralıyken, her geçişte bütün çiftler yer değiştirir ve *n − 1* geçiş gerekir. Karşılaştırma sayısı seçimli sıralamadaki gibi *n(n − 1)/2*'dir; zaman karmaşıklığı $O(n^2)$'dir. En iyi durumda dizi zaten sıralıdır. İlk geçişte *n − 1* karşılaştırma yapılır, hiç takas olmaz ve algoritma durur: O(n). Erken durma olmasaydı algoritma sıralı dizide de bütün geçişleri boşuna yapardı. Ek bellek O(1)'dir.

Sık yapılan hata

İç döngüyü `n - 1 - tur`'a kadar çalıştırmak. Karşılaştırma `dizi[j]` ile `dizi[j+1]` arasında yapıldığı için j'nin en büyük değeri, karşılaştırılacak son çiftin sol öğesidir. İlk geçişte bu indis $n - 2$'dir. j, $n - 1$'e kadar giderse `dizi[j+1]` dizinin dışına taşar; Java bu durumda `ArrayIndexOutOfBoundsException` hatası verir.

4.1.7.3 Eklemeli sıralama

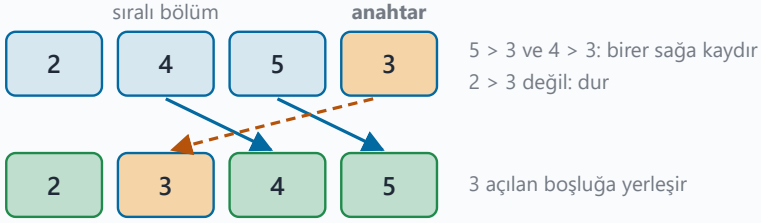
Kâğıt oyunlarında oyuncular eldeki kâğıtları genellikle şöyle dizer: yeni bir kâğıt alındığında, elde zaten sıralı duran kâğıtların arasında doğru yer bulunur ve kâğıt oraya sokulur. Eklemeli sıralama bu yöntemin aynısıdır.

Tanım 4.1.13

Eklemeli sıralama (*insertion sort*), dizinin sol bölümünü her zaman sıralı tutan ve sağdaki sıradaki öğeyi (**anahtar**; *key*) alıp, ondan büyük öğeleri birer sağa kaydırarak açılan boşluğa yerleştiren sıralama yöntemidir.

```
i = 1'den n-1'e kadar
    anahtar = dizi[i]
    j = i - 1
    j >= 0 ve dizi[j] > anahtar iken
        dizi[j+1] = dizi[j]
        j = j - 1
    dizi[j+1] = anahtar
```

Dış döngü i'yi 1'den başlatır, çünkü tek öğeli bir bölüm (0…0) zaten sıralıdır. Her turda anahtar, dizi[i]'den alınır; anahtarın yeri artık "boşluk" sayılabilir. j, anahtarın solundaki öğeden başlar. İç döngü iki koşulla sürer: j dizinin içinde olmalı ($j \geq 0$) ve dizi[j] anahtardan büyük olmalıdır. Koşul sağlandıkça dizi[j] bir sağa kaydırılır ve j bir azalır; boşluk sola doğru ilerler. Döngü durduğunda boşluk $j + 1$ konumundadır ve anahtar oraya yazılır.



Şekil 4.1.24: Eklemeli sıralamada bir adım. Sıralı bölüm {2, 4, 5}, anahtar 3'tür. 5 > 3 ve 4 > 3 olduğu için ikisi

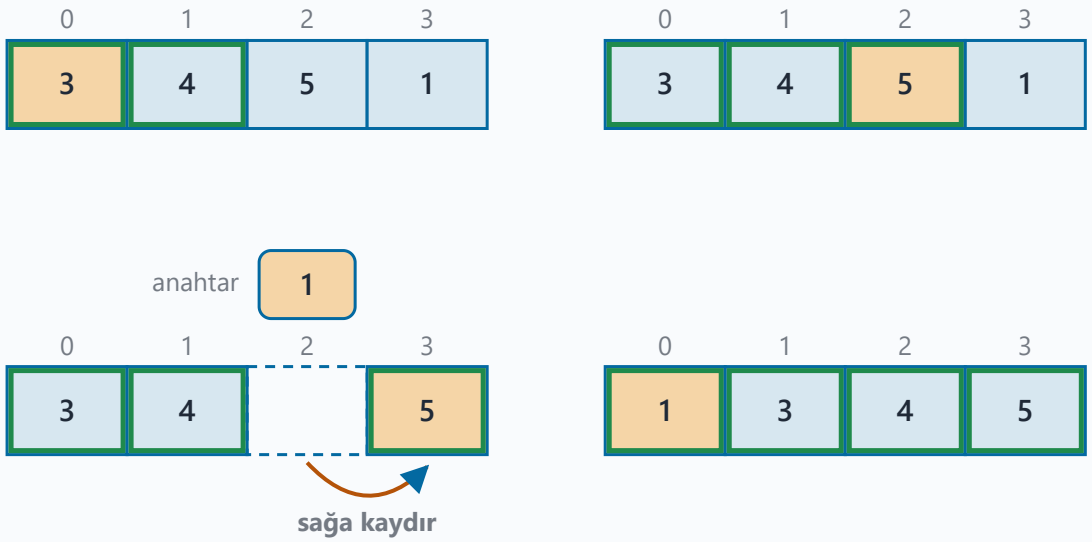
birer sağa kayar; 2 > 3 yanlış olduğu için durulur. 3, açılan boşluğa yerleşir: {2, 3, 4, 5}.

Koşuldaki `j >= 0` denetiminin önce yazılması önemlidir. j = −1 olduğunda bu koşul yanlıştır. "ve" bağlacı, ilk koşul yanlışsa ikinciye hiç bakmaz; dizi[−1] okunmaya çalışılmaz. İki koşulun sırası değiştirilirse anahtar en başa yerleşeceği zaman dizi[−1] okunur ve program hata verir.

Adım	anahtar	Karşılaştırmalar	Sayı	Kaydırma	Yerleştiği indis	Sonra dizi
i = 1	3	4 > 3 ✓ (j = −1, durur)	1	1	0	[3, 4, 5, 1]
i = 2	5	4 > 5 ✗	1	0	2	[3, 4, 5, 1]
i = 3	1	5 > 1 ✓, 4 > 1 ✓, 3 > 1 ✓ (j = −1, durur)	3	3	0	[1, 3, 4, 5]

Tablo 4.1.16: Eklemeli sıralamanın {4, 3, 5, 1} için izleme tablosu. ✓, dizi[j] > anahtar koşulunun doğru çıktığı (öğe kaydırılır), ✗ yanlış çıktığı (döngü durur) karşılaştırmadır.

Tablo 4.1.16 sunumdaki {4, 3, 5, 1} dizisinin izlemesidir. i = 1'de anahtar 3'tür; 4 > 3 olduğu için 4 sağa kayar, j = −1 olur ve döngü sol uca geldiği için durur. 3, 0 konumuna yerleşir. i = 2'de anahtar 5'tir; solundaki 4 ondan küçüktür ve tek karşılaştırmayla yeri bulunur, hiç kaydırma yapılmaz. i = 3'te anahtar 1 bütün öğelerden küçüktür. 5, 4 ve 3 sırayla sağa kayar ve 1 en başa yerleşir. Toplam *1 + 1 + 3 = 5* karşılaştırma yapılmıştır.



Şekil 4.1.25: Eklemeli sıralamadan dört an: i = 1 sonunda 3 yerleşmiştir; i = 2 sonunda 5 kaydırmasız yerleşmiştir; i = 3'te anahtar 1 için 5 sağa kayar; i = 3 sonunda 1 en başa yerleşmiş ve dizi sıralanmıştır. Yeşil çerçeveli hücreler sıralı bölümdür.

Eklemeli sıralamanın maliyeti, öğelerin ne kadar "yanlış yerde" olduğuna bağlıdır. En kötü durumda dizi ters sıralıdır; her anahtar sıralı bölümün en başına kadar kaydırılır ve $n(n-1)/2$ karşılaştırma yapılır: $O(n^2)$. En iyi durumda dizi zaten sıralıdır; her anahtar için tek karşılaştırma yapılır ve hiç kay-

dırma olmaz. Toplam *n − 1* karşılaştırma: O(n). Ek bellek yalnızca anahtar ve j değişkenleridir: O(1).

Bu özellik eklemeli sıralamayı uygulamada değerli kılar. Az sayıda öğe içeren ya da neredeyse sıralı olan dizilerde çok hızlı çalışır. Sıralı bir listeye arada bir yeni kayıt eklenen durumlarda, örneğin her gün birkaç öğrencinin eklendiği alfabetik bir listede, eklemeli sıralama doğal bir seçimdir.

> **Sık yapılan hata**
>
> Anahtarı bir değişkene almadan kaydırmaya başlamak. İlk kaydırma `dizi[i] = dizi[i-1]` atamasıdır ve dizi[i]'deki değeri siler. Anahtar önceden `anahtar` değişkenine kopyalanmamışsa bu değer kaybolur. Durum takastaki geçici değişkenin aynısıdır: üzerine yazılacak bir değer önce saklanır.

4.1.7.4 Birleştirme sıralaması

Önceki üç yöntem öğeleri tek tek yerine koyar ve O(n^2) karşılaştırma yapar. **Birleştirme sıralaması** (*merge sort*) başka bir fikre dayanır: iki sıralı diziyi tek bir sıralı dizide birleştirmek çok ucuzdur. İki sıralı dizinin en küçük öğesi, ikisinin ilk öğelerinden küçük olanıdır. O öğe alınır ve aynı karşılaştırma kalan öğeler için tekrarlanır. Her karşılaştırma sonuca bir öğe ekler; toplam öğe sayısı m ise en çok *m − 1* karşılaştırma yeter.

Algoritma bu birleştirme işlemini şöyle kullanır. Dizi ortadan iki yarıya bölünür. Her yarı aynı yöntemle ayrı ayrı sıralanır. Sonra sıralı iki yarı birleştirilir. Yarılar da bölüne bölüne sonunda tek öğeli dizilere iner; tek öğeli bir dizi zaten sıralıdır. Bir problemi aynı türden küçük problemlere bölüp onların çözümlerini birleştirmeye **böl ve fethet** (*divide and conquer*) yaklaşımı denir.

```
birleştir(A, B):
    C = boş dizi
    i = 0, j = 0
    i < A.uzunluk ve j < B.uzunluk iken
        eğer A[i] ≤ B[j] ise C'nin sonuna A[i] ekle, i = i + 1
        değilse C'nin sonuna B[j] ekle, j = j + 1
    A ve B'de kalan öğeleri sırasıyla C'nin sonuna ekle
    C'yi döndür

sirala(dizi):
    eğer dizi en çok 1 öğeliyse diziyi döndür
    diziyi ortadan iki yarıya böl: sol ve sag
    birleştir(sirala(sol), sirala(sag)) sonucunu döndür
```

`sirala` işlemi kendini daha küçük diziler için yeniden çağırır. Kendi tanımında yeniden kullanılan böyle bir işleme **özyinelemeli** işlem denir. Özyinelemeyi ayrıntısıyla anlamak bu not için gerekli değildir; önemli olan, bölme işinin tek öğeli dizilere inince durması ve her düzeydeki birleştirmenin sonucu bir üst düzeye iletmesidir.

Örnek 4.1.9

{38, 27, 43, 3, 9, 82, 10, 5} dizisi birleştirme sıralamasıyla nasıl sıralanır? Kaç karşılaştırma yapılır?

Dizi önce {38, 27, 43, 3} ve {9, 82, 10, 5} yarılarına, bunlar da ikişerli parçalara, en sonunda tek öğelere bölünür. Ardından birleştirmeler aşağıdan yukarıya yapılır:

Birleştirilen iki sıralı parça	Sonuç	Karşılaştırma
[38] + [27]	[27, 38]	1
[43] + [3]	[3, 43]	1
[27, 38] + [3, 43]	[3, 27, 38, 43]	3
[9] + [82]	[9, 82]	1
[10] + [5]	[5, 10]	1
[9, 82] + [5, 10]	[5, 9, 10, 82]	3
[3, 27, 38, 43] + [5, 9, 10, 82]	[3, 5, 9, 10, 27, 38, 43, 82]	7
Toplam		**17**

Son birleştirmede {3, 27, 38, 43} ve {5, 9, 10, 82} parçaları şöyle birleşir: 3 < 5 → 3; 27 > 5 → 5; 27 > 9 → 9; 27 > 10 → 10; 27 < 82 → 27; 38 < 82 → 38; 43 < 82 → 43. Sol parça bitmiştir; kalan 82 karşılaştırmasız eklenir. 8 öğe 7 karşılaştırmayla birleşmiştir. Bütün birleştirmelerde toplam 17 karşılaştırma yapılır. Aynı dizi için seçimli sıralama $8 \times 7 / 2 = 28$ karşılaştırma yapardı.

Birleştirme sıralamasının maliyeti düzey sayısından gelir. Dizi her düzeyde ikiye bölündüğü için $log_2\ n$ düzey vardır; 8 öğede 3 düzey. Her düzeyde bütün öğeler bir kez birleştirmeye girer ve en çok n karşılaştırma yapılır. Toplam karşılaştırma yaklaşık $n \times log_2\ n$'dir; zaman karmaşıklığı en iyi, ortalama ve en kötü durumda O(n log n)'dir. Bedeli bellektir: birleştirme sonucu ayrı bir dizide oluşturulur. Bu yüzden ek bellek O(n)'dir.

4.1.7.5 Sıralama yöntemlerinin karşılaştırılması

Algoritma	Temel fikir	En kötü	En iyi	Ek alan
Seçimli	En küçüğü seç ve yerleştir	$O(n^2)$	$O(n^2)$	O(1)
Kabarcık	Komşuları değiştir	$O(n^2)$	O(n) (erken durma ile)	O(1)
Eklemeli	Öğeyi sıralı bölüme ekle	$O(n^2)$	O(n)	O(1)
Birleştirme	Böl, sırala, birleştir	O(n log n)	O(n log n)	O(n)

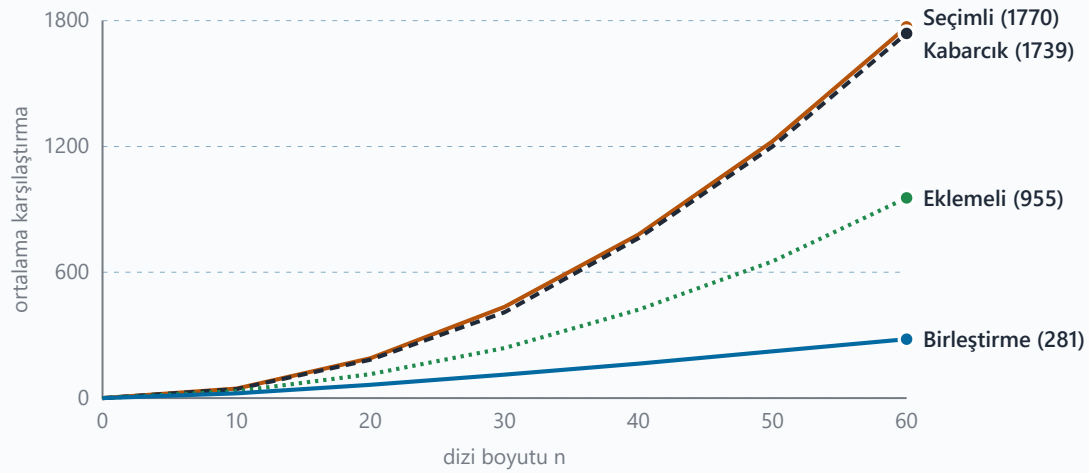
Tablo 4.1.17: Dört sıralama yönteminin karşılaştırması.

Tablo 4.1.17 dört yöntemi karmaşıklık açısından özetler. Büyük O değerleri aynı olan yöntemler bile gerçekte farklı sayıda karşılaştırma yapar. Tablo 4.1.18 üç basit yöntemi aynı beş öğeli dizilerde çalıştırarak sayar.

Girdi (n = 5)	Seçimli	Kabarcık (erken durmalı)	Eklemeli
Sıralı [1, 2, 3, 4, 5]	10 / 0	4 / 0	4 / 0
Ters sıralı [5, 4, 3, 2, 1]	10 / 2	10 / 10	10 / 10
Karışık [5, 1, 4, 2, 8]	10 / 2	9 / 4	7 / 4

Tablo 4.1.18: Beş öğeli üç dizide karşılaştırma ve takas sayıları (karşılaştırma / takas). Eklemeli sıralamada ikinci sayı kaydırma sayısıdır; seçimli sıralamada yalnızca gerçekten yer değiştiren takaslar sayılmıştır.

Tablo, karmaşıklık tablosundaki “en iyi” sütununu sayılarla doğrular. Sıralı dizide seçimli sıralama yine 10 karşılaştırma yapar; kabarcık ve eklemeli sıralama 4 karşılaştırmayla işi bitirir. Ters sıralı dizide üç yöntem de 10 karşılaştırma yapar, ama seçimli sıralama yalnızca 2 takasla yetinir. Seçimli sıralamanın az takas yapması, yazma işleminin pahalı olduğu durumlarda bir üstünlüktür.



Şekil 4.1.26: Rastgele dizilerde ortalama karşılaştırma sayısı. Her n için 20 rastgele dizi dört algoritmayla sıralanmış ve ortalamalar çizilmiştir. n = 60'ta seçimli 1.770, kabarcık 1.739, eklemeli 955, birleştirme 281 karşılaştırma yapmıştır.

Şekil 4.1.26 rastgele dizilerdeki ortalama davranışı gösterir. Seçimli ve kabarcık sıralaması yaklaşık $n^2 / 2$ karşılaştırma yapar; n = 60 için seçimli tam $60 \times 59 / 2 = 1.770$'tir. Eklemeli sıralama ortalamada bunun yaklaşık yarısını, $n^2 / 4$ civarını yapar. Bunun nedeni, rastgele bir dizide her anahtarın sıralı bölümün ortalama yarısını geçmesidir. Birleştirme sıralaması ise $n \log_2 n$ civarında kalır ve n büyüdükçe fark hızla açılır. Bu notun hesabında 2.000 rastgele diziyle yapılan ortalama, n = 60 için kabarcıkta yaklaşık 1.730, eklemelide 939, birleştirmede 281 çıkmıştır; sunumdaki 20 dizilik ortalamalar bu değerlere yakındır.

Uygulamada hazır kütüphane yöntemleri kullanılır. Java'daki `java.util.Arrays.sort` metodu O(n log n) zamanlı gelişmiş yöntemler kullanır. Nesne dizilerinde kullanılan yöntem (Timsort), birleştirme ve eklemeli sıralamanın fikirlerini birleştirir. `int` gibi temel tip dizilerinde çift pivotlu hızlı sıralama kullanılır. İki yöntem de küçük dizi parçalarında eklemeli sıralamaya geçer, çünkü eklemeli sıralama küçük ve neredeyse sıralı veride hızlıdır. Basit yöntemleri öğrenmenin değeri, bu tasarım kararlarının nedenini anlamayı sağlamasıdır.

Sıralama algoritmalarının nasıl çalıştığını gözle görmek, izleme tablolarını tamamlar. Sıralama algoritmaları görselleştiricisi, dört yöntemi rastgele, sıralı, ters sıralı ve neredeyse sıralı dizilerde adım adım çalıştırır ve karşılaştırma sayılarını gösterir.

4.1.8 Özet

- **Problem çözme** kodla başlamaz: problem anlaşılır, küçük bir örnek elle çözülür, yöntem yazılır, en son kodlanır. **Problem tanımı** amaç, girdi, çıktı, **kısıt**, örnek ve hata durumundan oluşur.
- **Algoritma**, bir problemi çözmek için tanımlanmış sonlu adımlar dizisidir; sıfır ya da daha fazla girdi alır, en az bir çıktı üretir. İyi bir algoritmanın adımları kesin, sıralı ve uygulanabilirdir; algoritma sonlu adımda durur, geçerli her girdi için doğru sonuç verir ve kaynakları verimli kullanır.
- Algoritma tasarımı bir döngüdür: örneği çöz, genel kuralı bul, sözde kod yaz, elle izle, düzelt. **Sözde kod**, algoritmayı belirli bir dile bağlı kalmadan yazar; girinti, koşul ve döngülerin kapsamını gösterir.
- İki değişkenin **takası** üç atama ve bir **geçici değişken** ister. Fibonacci'deki atama sırası da aynı nedenle önemlidir: üzerine yazılacak bir değer önce kullanılır ya da saklanır.
- Bir sonucu adım adım biriktiren **biriktiricinin** başlangıç değeri işleme göre seçilir: toplam için 0, çarpım için 1. Faktöriyel algoritmasının **döngü değişmezi** "her tur başında sonuc = (i − 1)!" ifadesidir ve algoritmanın neden doğru olduğunu açıklar.
- Palindrom denetimi iki uçtan ortaya ilerler ve en çok ⌊n / 2⌋ karşılaştırma yapar. Asal sayı denetimi bölenleri yalnızca √n'ye kadar arar (i × i ≤ n). İki algoritma da sonuç belli olunca erken durur.
- **Elle izleme** ve **izleme tablosu**, algoritmanın bir girdi için doğru çalıştığını gösterir. **Test durumları** normal, sınır, geçersiz ve büyük girdileri kapsar. Tek bir başarılı test, algoritmanın her girdi için doğru olduğunu kanıtlamaz; 21! testi taşmayı ve eksik kısıt denetimini ortaya çıkarır.
- **Zaman karmaşıklığı** işlem sayısının, **alan karmaşıklığı** ek belleğin girdi boyutuyla nasıl büyüdüğünü anlatır ve **büyük O gösterimiyle** yazılır: O(1), O(log n), O(n), O(n log n), O(n^2). 1 + 2 + … + n toplamı döngüyle O(n), Gauss formülüyle O(1) zamanda bulunur.
- **Doğrusal arama** sıralama istemez ve en kötü durumda n karşılaştırma yapar: O(n). **İkili arama** yalnızca sıralı veride çalışır, her turda aralığı yarılar ve en çok ⌊$\log_2$ n⌋ + 1 tur yapar: O(log n). n = 1.000.000'da fark 1.000.000'a karşı 20'dir.
- **Seçimli sıralama** her geçişte en küçüğü seçer; her zaman n(n − 1)/2 karşılaştırma yapar ama en çok n − 1 takas yapar. **Kabarcık sıralaması** komşuları değiştirir ve değişim olmayan geçişte durur. **Eklemeli sıralama** anahtarı sıralı bölüme sokar; neredeyse sıralı veride çok hızlıdır. Üçü de en

kötü durumda $O(n^2)$, ek alanda $O(1)$'dir.

- **Birleştirme sıralaması** diziyi ikiye böler, yarıları sıralar ve birleştirir; her durumda $O(n \log n)$ zaman, $O(n)$ ek bellek kullanır.

4.1.9 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

problem çözme *problem solving* · problem tanımı *problem definition* · kısıt *constraint* · algoritma *algorithm* · kesinlik *definiteness* · sonluluk *finiteness* · doğruluk *correctness* · sözde kod *pseudocode* · girinti *indentation* · takas *swap* · geçici değişken *temporary variable* · faktöriyel *factorial* · biriktirici *accumulator* · döngü değişmezi *loop invariant* · sınır hatası *off-by-one error* · kayan pencere *sliding window* · palindrom *palindrome* · asal sayı *prime number* · bölen *divisor* · kalan *remainder* · elle izleme *hand tracing* · izleme tablosu *trace table* · test durumu *test case* · geçersiz girdi *invalid input* · taşma *overflow* · zaman karmaşıklığı *time complexity* · alan karmaşıklığı *space complexity* · girdi boyutu *input size* · büyük O gösterimi *Big O notation* · en iyi durum *best case* · en kötü durum *worst case* · arama *search* · hedef *target* · doğrusal arama *linear search* · ikili arama *binary search* · arama aralığı *search range* · rastgele erişim *random access* · sıralama *sorting* · sıralı *sorted* · komşu *adjacent* · geçiş *pass* · seçimli sıralama *selection sort* · kabarcık sıralaması *bubble sort* · eklemeli sıralama *insertion sort* · anahtar *key* · birleştirme sıralaması *merge sort*

4.1.10 Kendini Dene

1. Üç sayıyı sıralayan algoritma (önce a–b, sonra b–c, en son yeniden a–b karşılaştırması) a = 4, b = 9, c = 2 için çalıştırılıyor. Her karşılaştırmadan sonra a, b, c ne olur? Kaç takas yapılır?

   Cevap

   1. karşılaştırma: *4 > 9* yanlış; değişiklik yok (4, 9, 2). 2. karşılaştırma: *9 > 2* doğru; b ile c yer değiştirir (4, 2, 9). 3. karşılaştırma: *4 > 2* doğru; a ile b yer değiştirir (2, 4, 9). Çıktı “2 4 9”dur; iki takas yapılmıştır.

2. Çift sayıların toplamını bulan algoritmada `i = 2` yerine yanlışlıkla `i = 1` yazılmış; artış yine `i`

`= i + 2` 'dir. n = 10 için ne yazılır? Hata hangi testle yakalanır?

Cevap

Sayaç 1, 3, 5, 7, 9 değerlerini alır ve tek sayılar toplanır: *1 + 3 + 5 + 7 + 9 = 25*. Doğru cevap 30'dur. n = 1 testi de hatayı gösterir: beklenen 0, bulunan 1'dir. n = 0 testi ise hatayı göstermez, çünkü iki sürümde de döngü hiç dönmez ve 0 yazılır.

3. Asal sayı algoritması n = 97 için çalıştırılıyor. Hangi i değerleri denenir, kaç kez % işlemi yapılır, sonuç nedir?

Cevap

i = 2'den 9'a kadar 8 değer denenir, çünkü *9 × 9 = 81 ≤ 97*, ama *10 × 10 = 100 > 97*'dir. Kalanlar sırasıyla 1, 1, 1, 2, 1, 6, 1, 7'dir; hiçbiri 0 değildir. 8 kez % işlemi yapılır ve “asal” yazılır.

4. Palindrom algoritması “Kalem” ve “Kepek” metinleri için çalıştırılıyor. Her birinde kaç karşılaştırma yapılır ve sonuç ne olur?

Cevap

“Kalem” küçültülünce “kalem” olur. İlk karşılaştırmada metin[0] = 'k' ile metin[4] = 'm' farklıdır; 1 karşılaştırmadan sonra yanlış döndürülür. “Kepek” için k ile k, sonra e ile e aynıdır; sol = sag = 2 olunca döngü biter. 2 karşılaştırmayla doğru döndürülür.

5. {4, 8, 15, 16, 23, 42} dizisinde 16 önce doğrusal aramayla, sonra ikili aramayla aranıyor. Her yöntem kaç karşılaştırma ya da tur yapar? İkili aramada her turda sol, sag ve orta nedir?

Cevap

Doğrusal arama 4, 8, 15 ve 16'ya bakar; 4 karşılaştırmadan sonra 3 döndürür. İkili arama: 1. tur sol = 0, sag = 5, orta = 2, dizi[2] = 15 < 16 → sol = 3. 2. tur sol = 3, sag = 5, orta = 4, dizi[4] = 23 > 16 → sag = 3. 3. tur sol = sag = 3, orta = 3, dizi[3] = 16 → 3 döndürülür. İkili arama 3 turda bulur.

6. 4.000 öğeli sıralı bir dizide bulunmayan bir değer aranıyor. İkili arama en çok kaç tur yapar? Dizi 8.000 öğeye çıkarsa doğrusal aramanın ve ikili aramanın en kötü durumdaki işi nasıl değişir?

Cevap

$2^{11} = 2.048 \le 4.000 < 4.096 = 2^{12}$ olduğu için $\lfloor \log_2 4.000 \rfloor = 11$'dir; ikili arama en çok *11 + 1 = 12* tur yapar. Dizi iki katına çıkınca doğrusal aramanın işi de iki katına çıkar (4.000'den 8.000 karşılaştırmaya). İkili aramanın tur sayısı yalnızca bir artar ve 13 olur.

7. [3, 1, 2, 4] dizisi seçimli, erken durmalı kabarcık ve eklemeli sıralamayla sıralanıyor. Her yöntem kaç karşılaştırma yapar? Kabarcık sıralaması kaç geçiş yapar?

   Cevap

   Seçimli: *3 + 2 + 1 = 6* karşılaştırma; dizinin içeriği sonucu değiştirmez. Kabarcık: 1. geçişte (3, 1) ve (3, 2) yer değiştirir, (3, 4) doğru sıradadır; 3 karşılaştırma, dizi [1, 2, 3, 4] olur. 2. geçişte 2 karşılaştırma yapılır, hiç takas olmaz ve algoritma durur. Toplam 2 geçiş, 5 karşılaştırma. Eklemeli: anahtar 1 için 1 karşılaştırma (3 > 1, sonra j = −1), anahtar 2 için 2 karşılaştırma (3 > 2, 1 > 2 değil), anahtar 4 için 1 karşılaştırma; toplam 4. Dizi neredeyse sıralı olduğu için eklemeli sıralama en az karşılaştırmayı yapar.

8. Palindrom algoritması için en az beş test durumu ve her biri için beklenen sonuç nedir? Her test hangi olası hatayı yakalar?

   Cevap

   Örnek bir test kümesi: boş metin → doğru (döngü hiç dönmez; algoritma çökmemelidir). Tek harf, “a” → doğru. Çift uzunlukta palindrom, “abba” → doğru (sol ile sag ortada birbirini geçer). Büyük harfli palindrom, “Kayak” → doğru (temizleme kuralı uygulanmazsa yanlış çıkar). Türkçe I içeren metin, “Iı” → doğru (I, i'ye çevrilirse yanlış çıkar). Palindrom olmayan metin, “kayık” → yanlış. Son harfi farklı metin, “kayaz” → yanlış (ilk karşılaştırmada durulmalıdır).

9. Bir algoritma n = 1.000 için 2 saniye sürüyor. n = 2.000 için yaklaşık ne kadar sürmesi beklenir? Algoritma O(n), O(n^2) ve O(log n) karmaşıklıkta olsaydı cevap nasıl değişirdi?

   Cevap

   O(n) bir algoritmada iş, n ile orantılı büyür; n iki katına çıkınca süre de yaklaşık iki katına çıkar: 4 saniye. O(n^2) bir algoritmada iş, n'nin karesiyle büyür; $2^2 = 4$ katına çıkar: yaklaşık 8 saniye. O(log n) bir algoritmada n'nin iki katına çıkması $\log_2$ n'yi yalnızca 1 artırır (yaklaşık 10'dan 11'e); süre yaklaşık %10 artar ve 2,2 saniye civarında kalır. Bunlar kaba tahminlerdir; büyük O katsayıları ve küçük terimleri yok sayar.

4.1.11 İleri okuma

- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), 4.1 Performance ve 4.2 Sorting and Searching: işlem sayısını deneyle ölçmeyi, ikili aramayı ve basit sıralama yöntemlerini Java örnekleriyle anlatır; kitabın sitesinde (introcs.cs.princeton.edu) bölümlerin özetleri ve kodları vardır.

- R. Sedgewick, K. Wayne, *Algorithms* (4. baskı, Addison-Wesley, 2011), Bölüm 2: seçimli, eklemeli ve birleştirme sıralamasını görsel izlemelerle karşılaştırır; kitap sitesindeki (algs4.cs.princeton.edu) canlandırmalar bu nottaki izleme tablolarını tamamlar.
- A. Levitin, *Introduction to the Design and Analysis of Algorithms* (3. baskı, Pearson, 2012), Bölüm 1–3: algoritma kavramını, problem tanımını ve kaba kuvvet yöntemlerini (seçimli ve kabarcık sıralaması, doğrusal arama) sade bir dille ele alır.
- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* (4. baskı, MIT Press, 2022), Bölüm 2: eklemeli sıralama üzerinden döngü değişmezi ile doğruluk kanıtını ve birleştirme sıralamasının analizini kesin biçimde verir; konuya ileri düzeyde devam etmek isteyenler içindir.

4.1.12 Kaynaklar

1. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
2. A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3. baskı, Pearson, 2012.
3. R. Sedgewick, K. Wayne, *Algorithms*, 4. baskı, Addison-Wesley, 2011. Kitap sitesi.
4. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 4.1 Performance, 4.2 Sorting and Searching.
5. Oracle, *Java SE 21 API*: java.util.Arrays (sort, binarySearch).

KISIM II · BÖLÜM 4.2

Akış Diyagramları

Bir algoritma ilk kez kâğıda dökülürken çoğu zaman yazıdan önce bir çizim yapılır. Kutular adımları, oklar adımların sırasını gösterir. Bu çizime akış diyagramı denir. Not, akış diyagramlarında kullanılan simgeleri ve iyi bir diyagramın kurallarını tanıtır. Ardından her algoritmanın yapı taşı olan üç akış türü gelir. Sıralı akışta adımlar arka arkaya çalışır. Koşullu akışta yol bir soruya göre ayrılır. Tekrarlı akışta bir grup adım tekrar tekrar çalışır. Her yapı küçük örneklerle adım adım yürütülür; değişkenlerin her adımdaki değeri izleme tablolarında verilir. Sayaç, biriktirici ve bekçi değer gibi döngü kalıpları, iç içe döngüler, erken çıkış ve bayrak da ayrı ayrı ele alınır. Son kesimler diyagramın Java koduna nasıl çevrildiğini ve bir diyagramın hangi girdilerle test edildiğini gösterir.

ÖĞRENME ÇIKTILARI

Simgeler
Temel akış diyagramı simgelerini (başla/bitir, işlem, karar, giriş/çıkış, ok, bağlayıcı) ve ek simgeleri tanımak; her adım için doğru simgeyi seçmek.

Akış
Oklarla işlem sırasını kurmak; okunabilir bir diyagramın kurallarını uygulamak; aynı algoritmayı akış diyagramı ve sözde kodla anlatmak.

Kontrol
Sıralı, koşullu ve tekrarlı akışı diyagramda göstermek; karar zinciri, bileşik koşul, sayaç, biriktirici, bekçi değer, iç içe döngü, son koşullu döngü, erken çıkış ve bayrak kalıplarını kullanmak; bir diyagramı izleme tablosuyla adım adım yürütmek.

Dönüşüm
Diyagramı sözde koda ve Java programına çevirmek; büyük bir diyagramı alt süreçlere bölmek; diyagramı normal, sınır ve döngünün sıfır, bir, çok kez döndüğü girdilerle test etmek.

4.2.1 Akış diyagramı

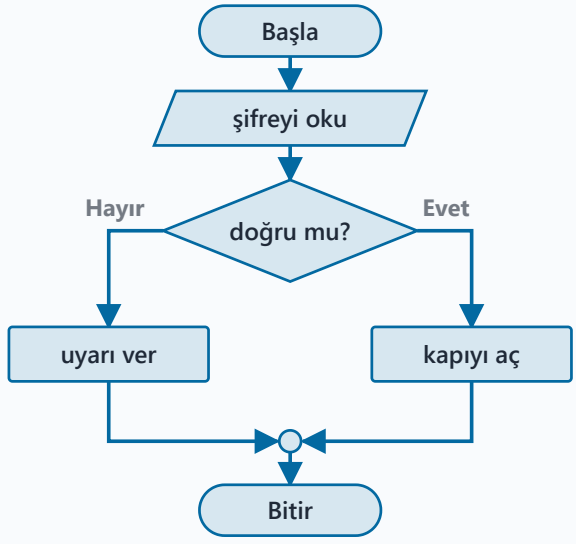
Bir algoritma, bir problemi çözmek için izlenecek açık ve sonlu adımlar dizisidir. Adımlar düz yazıyla da anlatılabilir. Ancak yazı uzadıkça hangi adımın hangisinden sonra geldiği, hangi durumda bir adımın atlandığı ya da tekrarlandığı gözden kaçar. Akış diyagramı bu sırayı göze görünür kılar.

4.2.1.1 Akış diyagramı nedir?

Tanım 4.2.1

Akış diyagramı (*flowchart*), bir algoritmanın adımlarını standart simgelerle, adımların çalışma sırasını da oklarla gösteren çizimdir. Simgenin biçimi adımın türünü (başlangıç, işlem, karar, giriş/çıkış), içine yazılan kısa ifade adımın ne yaptığını, ok da yürütmenin hangi adıma geçtiğini söyler.

Günlük bir örnek tanımı somutlaştırır. Bir apartman kapısında şifreli kilit vardır. Kilidin içindeki küçük bilgisayar her seferinde aynı işi yapar: tuşlanan şifreyi okur, doğru olup olmadığına bakar, doğruysa kapıyı açar, değilse bir uyarı sesi verir. Şekil 4.2.1 bu işi akış diyagramıyla gösterir.



Şekil 4.2.1: Kapı şifresi algoritmasının akış diyagramı. Karar kutusundan iki yol çıkar: şifre doğruysa "kapıyı aç", değilse "uyarı ver" çalışır. İki yol alttaki küçük dairede birleşir ve Bitir'e ulaşır.

Diyagram yukarıdan aşağıya okunur. Yürütme *Başla* yazan yuvarlak uçlu kutudan başlar. Ok, yürütmeyi "şifreyi oku" kutusuna götürür. Ardından eşkenar dörtgen biçimindeki karar kutusu gelir. İçinde bir soru vardır: "doğru mu?" Sorunun cevabına göre iki oktan biri izlenir. Evet okunu izleyen yol kapıyı açar, Hayır okunu izleyen yol uyarı verir. Her iki yol da sonunda aynı noktada buluşur ve *Bitir* kutusuna ulaşır.

Bu küçük çizimde akış diyagramının üç temel özelliği görülür. Birincisi, her adımın türü biçiminden anlaşılır; okuma işi ile karar işi farklı kutulardadır. İkincisi, adımların sırası oklarla bellidir; okur hangi adımın önce geldiğini tahmin etmek zorunda kalmaz. Üçüncüsü, bir karar kutusundan çıkan yollar ayrı ayrı izlenebilir; her çalıştırmada bu yollardan yalnızca biri kullanılır.

4.2.1.2 Neden diyagram çizilir?

Akış diyagramı bir programlama dilinden bağımsızdır. Aynı diyagram Java, Python ya da C ile yazılabilir. Bu yüzden bir algoritmanın mantığı, henüz hangi dille kodlanacağı belli değilken tartışılabilir. Bir ekipte programcı olmayan kişiler de, örneğin bir laboratuvar sorumlusu ya da bir iş süreci yöneticisi, diyagramı okuyup “bu durumda ne olacak?” diye sorabilir.

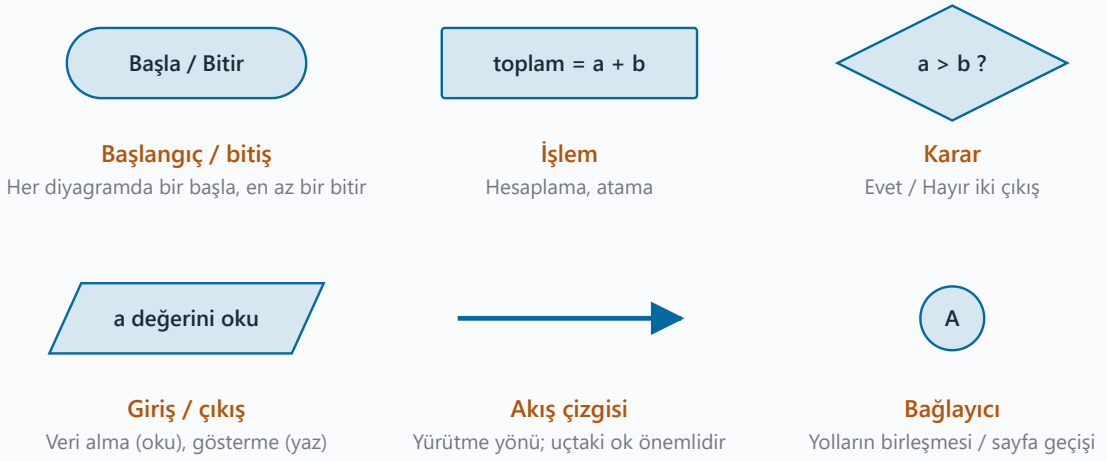
Diyagram çizmenin ikinci yararı hataların erken görülmesidir. Bir karar kutusundan yalnızca bir ok çıkıyorsa öbür durumda ne olacağı düşünülmemiştir. Hiçbir yerden ok gelmeyen bir kutu hiç çalışmaz. Geri dönen bir okun yolunda hiçbir şey değişmiyorsa döngü sonsuza kadar sürer. Bu tür eksikler kodda satırlar arasına gizlenir; diyagramda ise çizimin biçiminden fark edilir.

Üçüncü yarar öğrenmeyle ilgilidir. Programlamaya yeni başlayan biri için en zor kavramlardan biri yürütmenin sırasıdır. Bilgisayar satırları tek tek ve sırayla çalıştırır. Bir koşula göre bazı satırları atlar, bazılarını tekrar eder. Akış diyagramında bu sıra parmakla izlenebilir. Not boyunca her diyagram bu yolla, kutu kutu yürütülecektir.

4.2.2 Simgeler

Akış diyagramının simgeleri uluslararası bir standartla belirlenmiştir: ISO 5807. Standart 1985'te yayımlandı ve bugün de kullanılan çizim araçlarının çoğu ona uyar. Küçük bir algoritma için beş temel simge yeter. Diyagram büyüdüğünde ya da dosya, rapor gibi dış ortamlar gösterilmek istendiğinde ek simgelere başvurulur.

4.2.2.1 Temel simgeler



Şekil 4.2.2: Temel akış diyagramı simgeleri. Üst sırada başlangıç/bitiş (oval), işlem (dikdörtgen) ve karar (eşkenar dörtgen); alt sırada giriş/çıkış (paralelkenar), akış çizgisi (ok) ve bağlayıcı (küçük daire).

Şekil 4.2.2 temel simgeleri bir arada gösterir. Her birinin görevi aşağıda tek tek açıklanır.

Tanım 4.2.2

Başla/bitir simgesi (*terminal symbol*) oval ya da yuvarlak uçlu dikdörtgendir. Algoritmanın nereden başlayıp nerede bittiğini gösterir. Her diyagramda tek bir Başla ve en az bir Bitir bulunur.

Başla simgesine ok girmez, yalnızca ok çıkar. Bitir simgesine ok girer, ondan ok çıkmaz. Bir diyagramda birden fazla Bitir olabilir. Örneğin sayı tahmin oyununda doğru tahmin yapıldığında oyun hemen biter; bu durumda "Bildin" yazısından sonra ayrı bir Bitir çizmek diyagramı sadeleştirir. Yine de çoğu durumda yolları birleştirip tek bir Bitir'e götürmek okumayı kolaylaştırır.

Tanım 4.2.3

İşlem simgesi (*process symbol*) dikdörtgendir. Bir hesaplamayı ya da bir **atamayı** (*assignment*) gösterir. Atama, bir değişkene yeni bir değer verilmesidir: `toplam = a + b` ifadesi, sağdaki `a + b` değerini hesaplar ve sonucu `toplam` adlı değişkene yazar.

Burada **değişken** (*variable*) sözcüğü ilk kez geçiyor. Değişken, bellekte adı olan bir kutu gibi düşünülebilir. Kutunun içinde her an tek bir değer bulunur. Yeni bir değer yazıldığında eskisi silinir. Atamadaki eşittir işareti matematikteki eşitlikten farklıdır. `i = i + 1` matematikte anlamsızdır,

çünkü hiçbir sayı kendisinin bir fazlasına eşit olamaz. Akış diyagramında ve programda ise bu ifade "i'nin şimdiki değerine 1 ekle, sonucu yeniden i'ye yaz" demektir. i önce 4 ise, bu kutu çalıştıktan sonra 5 olur.

Tanım 4.2.4

Karar simgesi (*decision symbol*) eşkenar dörtgendir. İçine cevabı "evet" ya da "hayır" olan bir soru, yani bir **koşul** (*condition*) yazılır: `a > b ?`. Karar simgesine bir ok girer, ondan iki ok çıkar; çıkan oklar Evet ve Hayır diye etiketlenir.

Koşul bir karşılaştırmadır. Akış diyagramlarında ve Java'da altı karşılaştırma işleci vardır. Bunlar `<` (küçük), `<=` (küçük ya da eşit), `>` (büyük), `>=` (büyük ya da eşit), `==` (eşit) ve `!=` (eşit değil) işleçleridir. Eşitlik sorusu için iki eşittir işareti kullanılır. Bunun nedeni tek eşittir işaretinin atamaya ayrılmış olmasıdır. Bazı kitaplar ≤, ≥ ve ≠ işaretlerini de kullanır; anlam aynıdır.

Tanım 4.2.5

Giriş/çıkış simgesi (*input/output symbol*) paralelkenardır. Algoritmanın dışarıdan veri almasını ("a değerini oku") ya da dışarıya sonuç vermesini ("toplamı yaz") gösterir. Dışarıdan alınan veriye **girdi** (*input*), dışarıya verilen sonuca **çıktı** (*output*) denir.

"Oku" sözcüğü verinin klavyeden, bir dosyadan ya da bir algılayıcıdan gelmesini kapsar. Akış diyagramı verinin nereden geldiğiyle ilgilenmez; yalnızca o noktada bir değerin algoritmaya girdiğini söyler. "Yaz" da aynı biçimde ekrana, yazıcıya ya da bir dosyaya yazmayı kapsar.

Tanım 4.2.6

Akış çizgisi ya da kısaca **ok** (*flow line*), iki simgeyi birleştirir ve yürütmenin yönünü gösterir. **Bağlayıcı** (*connector*) küçük bir dairedir; birden fazla yolun birleştiği noktayı ya da diyagramın başka bir yerde sürdüğünü gösterir.

Okun ucu önemlidir. Uçsuz bir çizgi iki kutunun bağlı olduğunu söyler ama hangisinin önce çalıştığını söylemez. Bu yüzden her akış çizgisinin ucunda bir ok başı bulunur. Bağlayıcı, Şekil 4.2.1'de olduğu gibi bir karardan ayrılan iki yolun yeniden buluştuğu yeri belirginleştirir. İçine bir harf yazılan bağlayıcı ise aynı harfi taşıyan başka bir bağlayıcıyla eşleşir. Böylece sayfanın bir köşesinden öbür köşesine uzun bir ok çekmek gerekmez.

Simge	Biçim	Görev	Örnek
	Oval	Başlangıç ya da bitiş	Başla

Simge	Biçim	Görev	Örnek
	Dikdörtgen	İşlem (hesaplama, atama)	`toplam = a + b`
	Eşkenar dörtgen	Karar (koşul)	`a > b ?`
	Paralelkenar	Giriş / çıkış	a değerini oku
	Ok	Yürütme yönü	sonraki adıma geç

Tablo 4.2.1: Temel simgeler, görevleri ve içlerine yazılabilecek örnek ifadeler.

4.2.2.2 Simgenin içine ne yazılır?

Simgenin biçimi adımın türünü zaten söylediği için içine yalnızca o adımın kısa ifadesi yazılır. Karar simgesine bir *soru* yazılır: `a > b ?`. İşlem simgesine bir *emir* yazılır: `toplam = a + b`. Giriş/çıkış simgesine okunan ya da yazılan şey yazılır: "a değerini oku", "toplamı yaz". Bu ayrım bir diyagramı okuyan kişinin, kutunun içini okumadan önce bile o adımda ne olacağını bilmesini sağlar.

Soru biçiminin bir kuralı daha vardır: cevabı iki seçenekli olmalıdır. "a ile b'den hangisi büyük?" sorusunun cevabı "a" ya da "b" olur ama eşitlik durumunu açıkta bırakır. `a > b ?` sorusunun cevabı ise her zaman evet ya da hayırdır: a = b ise cevap hayırdır. Bu yüzden karar kutusuna her zaman bir karşılaştırma yazılır.

Örnek 4.2.1

Bir algoritma, öğrencinin vize ve final notundan ortalama hesaplayıp geçme durumunu yazıyor. Adımları şunlardır: vize notunu al; final notunu al; ortalamayı vize × 0,4 + final × 0,6 olarak hesapla; ortalama 50 ya da üstü mü diye bak; sonucu ekrana yaz. Her adım hangi simgeye konur, içine ne yazılır?

Çözüm. "Vize notunu al" bir girdidir: paralelkenar, içine "vize değerini oku". Final de aynı biçimde paralelkenara girer. Ortalama hesabı bir atamadır: dikdörtgen, içine `ort = vize × 0,4 + final × 0,6`. "50 ya da üstü mü" bir sorudur: eşkenar dörtgen, içine `ort >= 50 ?`. Sonucu yazmak bir çıktıdır: Evet ve Hayır dallarında birer paralelkenar, içlerine "Geçti" yaz ve "Kaldı" yaz. En başa Başla, en sona Bitir eklenir. Vize 40, final 60 ise ortalama *40 × 0,4 + 60 × 0,6 = 16 + 36 = 52* olur; 52 ≥ 50 olduğu için Evet yolu izlenir ve "Geçti" yazılır.

Sık yapılan hata

Hesaplamayı paralelkenara yazmak. `toplam = a + b` ekranla ya da klavyeyle ilgili değildir; bellekte bir değer hesaplar. Bu yüzden işlem simgesine, yani dikdörtgene aittir. Paralelkenar yalnızca dış dünyayla veri alışverişi içindir. Benzer bir karışıklık karar kutusuna emir yazıldı-

ğında görülür: “a'yı b ile karşılaştır ve büyüğü yaz” cümlesi bir soru değildir ve iki işi bir arada yapar. Doğrusu karar kutusuna `a > b ?` yazmak, yazma işlerini dallardaki paralelkenarlara taşımaktır.

4.2.2.3 Ek simgeler

Beş temel simge küçük algoritmalar için yeterlidir. Bazı durumlarda ISO 5807'deki ek simgelere başvurulur. Büyük bir diyagram birkaç sayfaya bölünebilir, ayrı çizilmiş bir alt süreç çağrılabilir ya da bir dosya veya rapor gösterilmek istenebilir. Tablo 4.2.2 en sık karşılaşılan altısını özetler.

Simge	Ad	Görev	Örnek
	Sayfa dışı bağlayıcı	Diyagram başka sayfada sürer; içine sayfa ve nokta yazılır.	2-A
	Önceden tanımlı işlem	Ayrıca çizilmiş bir alt süreci çağırır.	Ortalama hesapla
	Hazırlık	Döngü sayacının başlangıcını ve sınırını tek simgede verir.	i = 1 → n
	Belge	Yazdırılan çıktı ya da rapor.	Not çizelgesi
	Veri deposu	Diskte ya da veri tabanında saklanan veri.	Öğrenci kayıtları
	Açıklama	Kesikli çizgiyle bir simgeye not ekler; akışa katılmaz.	not 0–100 arası

Tablo 4.2.2: Ek simgeler.

Sayfa dışı bağlayıcı, sayfa içindeki bağlayıcının bir benzeridir. İçindeki “2-A” etiketi “ikinci sayfadaki A noktasında devam ediyor” anlamına gelir. Önceden tanımlı işlem simgesi, kenarlarında iki dikey çizgi bulunan bir dikdörtgendir. Bu simge büyük diyagramları parçalara bölmenin aracıdır: alt süreç bir kez çizilir, ana diyagramda gerektiği her yerde bu simgeyle çağrılır. Hazırlık simgesi altıgendir ve bir sayaçlı döngünün başlangıç değerini, sınırını ve artış miktarını tek kutuda toplar. Açıklama simgesi akışa katılmaz; bir kutuya kesikli bir çizgiyle bağlanır ve okura bilgi verir. Örneğin “not 0–100 arası” açıklaması, girdinin hangi aralıkta olması gerektiğini hatırlatır.

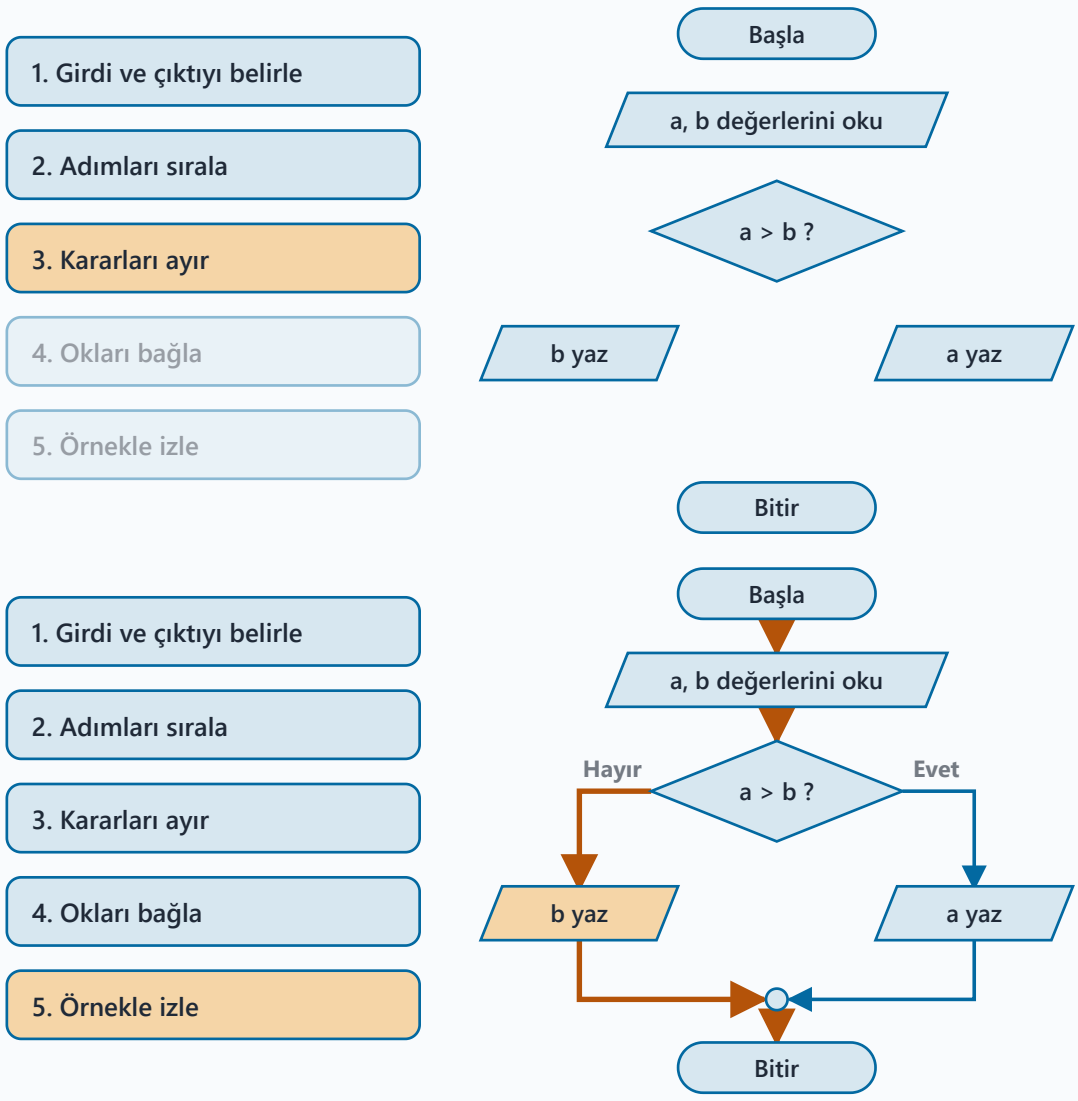
4.2.3 Diyagram tasarımı

Simgeleri tanımak bir diyagram çizmeye yetmez. Bir problemden diyagrama giden yolun belli adımları vardır. Bu adımlar izlendiğinde hem eksik bir durum kalmaz hem de ortaya okunması kolay bir çizim çıkar.

4.2.3.1 Adım adım bir diyagram: iki sayının büyüğü

Problem kısadır: iki sayı veriliyor, büyük olanı yazılacak. Diyagram beş adımda kurulur.

1. **Girdi ve çıktıyı belirle.** Algoritmanın dışarıdan ne aldığı ve dışarıya ne verdiği önce yazılır. Girdi iki sayıdır: a ve b. Çıktı büyük olan sayıdır. İkisi de paralelkenarla gösterilir: “a, b değerlerini oku” ve “büyüğü yaz”.
2. **Adımları sırala.** Başla ve Bitir eklenir. Adımlar yukarıdan aşağıya dizilir: önce okuma, en son yazma. Okunmamış bir değer karşılaştırılamaz; bu yüzden okuma kutusu her zaman kararın üstündedir.
3. **Kararları ayır.** Hangi sayının yazılacağı bir soruya bağlıdır: a, b'den büyük mü? Araya `a > b ?` karar kutusu girer. Tek “büyüğü yaz” kutusu artık iki kutuya ayrılır: “a yaz” ve “b yaz”.
4. **Okları bağla.** Her kutu bir sonrakine okla bağlanır. Karardan çıkan iki ok Evet ve Hayır diye etiketlenir. İki dal alttaki bir bağlayıcıda birleşir ve Bitir'e gider.
5. **Örnekle izle.** Diyagram bir girdiyle elle yürütülür. a = 4, b = 9 için *4 > 9* yanlıştır; Hayır yolu izlenir, “b yaz” kutusu çalışır ve 9 yazılır. Sonuç doğrudur.



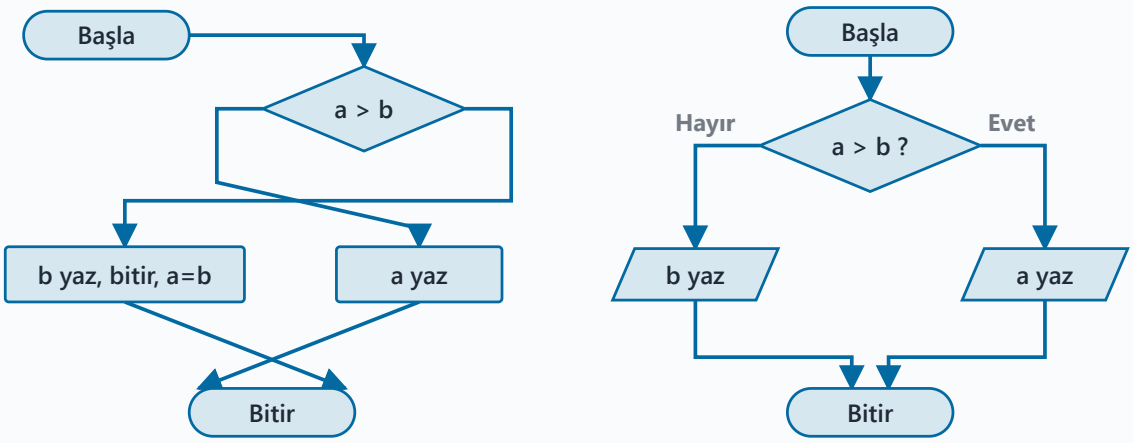
Şekil 4.2.3: İki sayının büyüğünü bulan diyagramın kuruluşu. (a) 3. adım: karar eklenince çıktı iki kutuya ayrılır; oklar henüz yoktur. (b) 5. adım: oklar bağlanmış, a = 4, b = 9 için izlenen yol renkle gösterilmiştir: Hayır yolu,

"b yaz" ve 9.

Beşinci adım çoğu zaman atlanır ama en değerli adımdır. Elle izleme sırasında yalnız bir girdi denemek yetmez. a = 9, b = 4 ile Evet yolu da denenmelidir; bu durumda 9 yazılır. Peki a = b = 5 ise? *5 > 5* yanlış olduğu için Hayır yolu izlenir ve b, yani 5 yazılır. İki sayı eşitken hangisinin yazıldığı önemli değildir; diyagram bu durumda da doğru sonucu verir. Bu küçük deneme, karar kutusuna `>` yerine `>=` yazılsaydı da sonucun değişmeyeceğini gösterir.

4.2.3.2 Okunabilirlik kuralları

Aynı algoritma iyi ya da kötü çizilebilir. Şekil 4.2.4 aynı iki sayı problemini iki biçimde gösterir. Soldaki çizimde oklar birbirini keser, karar kutusunun çıkışları etiketsizdir ve bir kutuya "b yaz, bitir, a=b" gibi üç ayrı iş sıkıştırılmıştır. Sağdaki çizim yukarıdan aşağıya akar, kararın çıkışları etiketlidir ve her kutuda tek bir iş vardır.



Şekil 4.2.4: Okunaksız ve okunaklı diyagram. Solda kesişen oklar, etiketsiz karar ve bir kutuda üç işlem; sağda yukarıdan aşağı akış, Evet/Hayır etiketli karar ve kutu başına tek işlem.

Kurallar üç başlıkta toplanır:

- **Akışın yönü.** Akış mümkün olduğunca yukarıdan aşağıya ve soldan sağa ilerler. Okur gözünü bu yönde gezdirir; ters yöne giden oklar yalnız döngülerde, geri dönüşü göstermek için kullanılır. Kesişen oklar azaltılır; kesişme kaçınılmazsa bağlayıcı kullanılır.
- **Karar etiketleri.** Karardan çıkan her ok Evet/Hayır (ya da Doğru/Yanlış) gibi anlamlı bir etiket taşır. Etiketsiz bir kararda hangi yolun hangi durumda seçildiği tahmine kalır.
- **Kutu başına bir iş.** Bir simgeye tek bir temel işlem yazılır. Bir kutuya üç işlem sıkıştırılırsa hangi işlemin önce yapıldığı belirsizleşir ve kutu bir karar içeriyorsa bu karar gözden kaçar. Diyagram büyüyüp sayfaya sığmaz hâle gelirse alt süreçlere bölünür.

Sık yapılan hata

Karardan yalnızca bir ok çıkarmak. Öğrenci aklındaki durumu, örneğin "not 60'tan büyükse Geçti yaz" durumunu çizer, öbür durumu unutur. Not 45 olduğunda yürütmenin nereye gideceği diyagramda yazmaz; akış kopar. Bir karar kutusundan her zaman iki ok çıkar. Öbür durumda yapılacak bir iş yoksa Hayır oku doğrudan birleşme noktasına gider.

4.2.3.3 Akış diyagramı ve sözde kod

Algoritma anlatmanın ikinci yaygın yolu sözde koddur.

Tanım 4.2.7

Sözde kod (*pseudocode*), bir algoritmanın adımlarını programlama diline benzeyen ama dilin kurallarına bağlı olmayan kısa cümlelerle yazma biçimidir. Satırlar sırayla okunur; bir kararın ya da döngünün içindeki satırlar girintiyle gösterilir.

Şekil 4.2.5 bir sayının mutlak değerini bulan algoritmayı iki biçimde gösterir. Sayı negatifse işareti değiştirilir, değilse olduğu gibi kalır.



Şekil 4.2.5: Mutlak değer algoritmasının akış diyagramı: x okunur, x < 0 ise x = −x yapılır, sonra x yazılır.

```
oku x
eğer x < 0 ise
    x = -x
yaz x
```

İki gösterim aynı algoritmayı anlatır ve biri öbürüne satır satır çevrilebilir. “oku x” paralelkenara, “eğer x < 0 ise” karar kutusuna, girintili `x = -x` satırı kararın Evet dalındaki dikdörtgene, “yaz x” son paralelkenara karşılık gelir. Kararın Hayır dalında hiçbir kutu yoktur; ok doğrudan yazma kutusuna gider. Sözde kodda bu durum, “eğer” satırının altında girintili tek bir satır bulunmasıyla gösterilir.

	Akış diyagramı	**Sözde kod**
Biçim	Simgeler ve oklar	Girintili kısa cümleler
Güçlü yanı	Dallanma ve döngü bir bakışta görülür	Hızlı yazılır, koda çok yakındır
Zayıf yanı	Büyüyünce sayfaya sığmaz, değiştirmek zahmetlidir	Akışın yapısı girintiye bağlıdır
Uygun olduğu yer	Öğrenme, sunum, iş süreçleri	Uzun algoritmalar, kodlamadan hemen önce

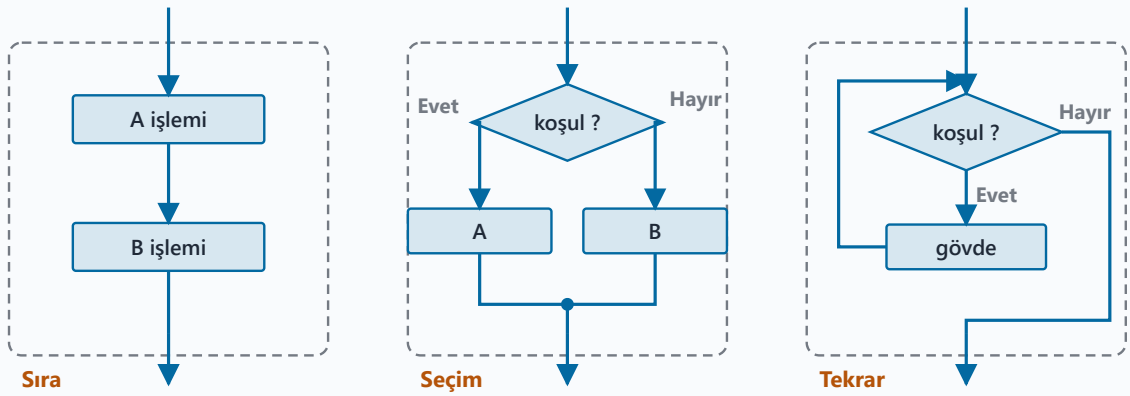
Tablo 4.2.3: Akış diyagramı ile sözde kodun karşılaştırılması.

Sözde kodun bir standardı yoktur. Kimi kitaplar “eğer … ise”, kimi “if … then” yazar; kimi atama için `=`, kimi `←` kullanır. Önemli olan her adımın açık olmasıdır. Bu notta sözde kod Türkçe sözcüklerle ve Java'ya yakın işaretlerle yazılır: atama `=`, eşitlik sorusu `==`, kalan `%`.

4.2.3.4 Yapısal programlamanın üç yapısı

Akış diyagramında oklar istenen her yere çizilebilir. Bir okun diyagramın ortasındaki rastgele bir kutuya atlaması da teknik olarak mümkündür. 1960'larda büyüyen programlarda bu serbestlik ciddi sorunlar doğurdu. Oklar birbirine dolanıyor, bir kutuya kaç farklı yoldan gelindiği izlenemiyordu. Böyle çizimlere ve programlara, birbirine dolanmış makarnaya benzetilerek **spagetti akış** (*spaghetti code*) adı verildi.

Çözüm, okları rastgele çizmek yerine yalnızca üç kalıp kullanmaktır. Şekil 4.2.6 bu üç kalıbı gösterir.



Şekil 4.2.6: Yapısal programlamanın üç yapısı. Sıra: A işlemi, ardından B işlemi. Seçim: koşula göre A ya da B. Tekrar: koşul doğru oldukça gövde tekrar çalışır. Her yapının tek girişi ve tek çıkışı vardır.

Tanım 4.2.8

Yapısal programlama (*structured programming*), algoritmaları yalnızca üç kontrol yapısıyla kurma yaklaşımıdır. **Sıralı yapı** (*sequence*): adımlar arka arkaya çalışır. **Seçim** ya da karar yapısı (*selection*): bir koşula göre iki yoldan biri çalışır. **Döngü yapısı** ya da tekrar (*repetition*): bir koşul doğru oldukça bir grup adım tekrar çalışır.

Teorem 4.2.1 (Böhm–Jacopini)

Her algoritma yalnızca sıra, seçim ve tekrar yapılarıyla yazılabilir. Corrado Böhm ve Giuseppe Jacopini bu sonucu 1966'da yayımladı.

Teoremin söylediği şey güçlüdür: rastgele atlayan oklara hiçbir zaman *gerek* yoktur. Spagetti biçimde çizilmiş her diyagramın aynı işi yapan yapısal bir karşılığı vardır. Gerekirse bir iki ek değişken (ör. bir bayrak) kullanılır.

Üç yapının ortak özelliği **tek giriş, tek çıkış** kuralıdır. Her yapıya yukarıdan tek bir ok girer, aşağıdan tek bir ok çıkar. Bu özellik sayesinde bir yapı, başka bir yapının içindeki herhangi bir kutunun yerine konabilir. Örneğin bir döngünün gövdesindeki dikdörtgen yerine bütün bir karar yapısı yerleştirilebilir. Buna **iç içe** (*nested*) yapılar denir. Büyük algoritmalar bu yolla, küçük yapılar iç içe geçirilerek kurulur. Diyagramın herhangi bir parçası kendi giriş ve çıkış okuyla ayrı ayrı anlaşılabilir ve test edilebilir.

Programlama dilleri de bu fikri benimsedi. Eski dillerde programın herhangi bir satırına atlayan `goto` deyimi yaygındı. Java'da `goto` ayrılmış bir sözcüktür, yani değişken adı olarak kullanılamaz, ama dilde hiçbir işe yaramaz. Java'da yürütmenin akışı yalnızca sıralı satırlarla, `if` / `else` ile ve `while`, `do`, `for` döngüleriyle kurulur.

4.2.3.5 Çizim araçları

Akış diyagramı çizmek için özel bir araç şart değildir. Seçim, diyagramın ne için kullanılacağına bağlıdır:

- **Kâğıt ve kalem:** İlk taslak için en hızlı yoldur; silip yeniden çizmek kolaydır. Sınavlarda da diyagram elle çizilir.
- **Word ve PowerPoint:** Ekle → Şekiller menüsündeki Akış Çizelgesi grubunda bütün temel simgeler bulunur. Bağlayıcı oklar kutulara yapıştığı için bir kutu taşındığında ok da birlikte kayar.
- **draw.io (diagrams.net):** Ücretsizdir ve tarayıcıda çalışır. Hazır akış diyagramı şekilleri sunar; çizim PNG, SVG ya da PDF olarak dışa aktarılabilir.
- **Microsoft Visio:** Kurumsal süreç ve iş akışı diyagramları için kullanılan ticari bir araçtır.
- **Flowgorithm:** Diyagramı yalnızca çizmez, çalıştırır. Değişkenlerin değerini adım adım gösterir ve diyagramı Java dahil birçok dilde koda çevirir.

- **Mermaid:** Diyagram metinle tanımlanır, araç onu çizer. Belgelerde ve kod depolarında sık kullanılır.

Araç ne olursa olsun simgelerin anlamı ve okunabilirlik kuralları aynıdır. Dersin Akış Diyagramı Yürütücü simülatörü de Flowgorithm gibi diyagramı adım adım yürütür; bu nottaki örneklerin çoğu orada da denenebilir.

4.2.4 Sıralı akış

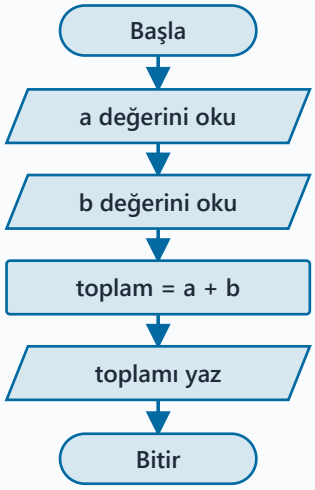
Üç yapının en basiti sıralı yapıdır. Kontrol yapılarının hepsi bu temel üzerine kurulur: bir karar ya da döngü olmayan her kesim, kendi içinde sıralı akıştır.

4.2.4.1 Sıralı yapı

Tanım 4.2.9

Sıralı akış (*sequential flow*), adımların yazıldıkları sırayla, her biri bir öncekinin bitmesinden sonra ve tam bir kez çalıştığı akıştır. Dallanma ya da tekrar içermez; tek girişi ve tek çıkışı vardır.

Şekil 4.2.7 iki sayıyı okuyup toplamlarını yazan diyagramı gösterir. Diyagramdaki bütün oklar aşağı doğrudur; hiçbir kutudan birden fazla ok çıkmaz.



Şekil 4.2.7: Sıralı akış: Başla, a oku, b oku, toplam = a + b, toplamı yaz, Bitir.

```
Başla
  a değerini oku
  b değerini oku
  toplam = a + b
  toplamı yaz
Bitir
```

Sözde kodda Başla ile Bitir arasındaki dört satır girintilidir; her satır diyagramdaki bir kutuya karşılık gelir.

4.2.4.2 Sıralı akışı adım adım yürütmek

Bir diyagramı **yürütmek**, bilgisayarın yapacağı işi elle yapmaktır. Başla kutusundan başlanır. Her adımda yalnızca bir kutu çalışır. Kutu çalıştıktan sonra değişkenlerin yeni değerleri ve varsa çıktı not edilir, sonra okun gösterdiği kutuya geçilir. Bu notlar bir tabloda toplanırsa ortaya bir **izleme tablosu** (*trace table*) çıkar. Tablo 4.2.4 a = 4, b = 7 girdisiyle yürütmenin izleme tablosudur.

Adım	Çalışan kutu	a	b	toplam	Çıktı
1	Başla	—	—	—	
2	a değerini oku	4	—	—	
3	b değerini oku	4	7	—	
4	toplam = a + b	4	7	11	
5	toplamı yaz	4	7	11	11
6	Bitir	4	7	11	11

Tablo 4.2.4: Sıralı akışın a = 4, b = 7 için izleme tablosu. Tire, değişkenin henüz bir değeri olmadığını gösterir.



Şekil 4.2.8: Sıralı akışın üç adımı; o anda çalışan kutu vurgulanmıştır. (a) 2. adım: a = 4 okunur. (b) 4. adım: toplam = 4 + 7 = 11. (c) 6. adım: Bitir'e ulaşılır.

Tablo sıralı akışın iki özelliğini açıkça gösterir. Birincisi, altı kutunun her biri tam bir kez çalışır; tabloda tekrar eden ya da atlanan bir satır yoktur. İkincisi, bir değişken değer aldıktan sonra değiştirilmedikçe o değeri korur: a, 2. adımda 4 olur ve sona kadar 4 kalır. Tablo okunurken her satır, “bu kutu ça-

lıştıktan hemen sonra" durumunu anlatır.

Diyagramın Java karşılığı da aynı sırayı izler:

```java
import java.util.Scanner;

public class Toplam {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int a = giris.nextInt();          // a değerini oku
        int b = giris.nextInt();          // b değerini oku
        int toplam = a + b;               // toplam = a + b
        System.out.println(toplam);       // toplamı yaz
    }
}
```

Programın yapısı birkaç sözcükle özetlenebilir. `public class Toplam` satırı programın adını verir. `main` metodu programın başladığı yerdir; diyagramdaki Başla kutusunun karşılığı onun açılan küme parantezi, Bitir'in karşılığı kapanan küme parantezidir. `Scanner`, Java'nın klavyeden veri okumak için sunduğu hazır sınıftır; `giris.nextInt()` klavyeden bir tam sayı okur. Bu yüzden her "oku" paralelkenarı bir `nextInt()` çağrısına dönüşür. Atama dikdörtgeni aynen `int toplam = a + b;` satırına, "yaz" paralelkenarı `System.out.println(toplam);` satırına karşılık gelir. Satır sonlarında `//` ile başlayan açıklamalar vardır. Derleyici bunları yok sayar; burada her satırın diyagramdaki karşılığını hatırlatırlar.

Program 4 ve 7 girdisiyle çalıştırıldığında 11 yazar. −3 ve 10 girdisiyle 7 yazar. Sıralı bir programda girdiler değişse bile çalışan satırlar ve sıraları hiç değişmez; değişen yalnızca değerlerdir.

4.2.4.3 Sıra neden önemlidir?

Sıralı akışta kutuların yeri keyfî değildir. Toplam kutusu okuma kutularının üstüne taşınsa a ile b'nin henüz değeri olmaz ve toplam hesaplanamaz. Bu hata kolay fark edilir. Daha sinsi hatalar, iki atamanın yer değiştirmesinden doğar.

Örnek 4.2.2

İki değişkenin değerleri yer değiştirecek; bu işleme **takas** denir. a = 4, b = 7 ise işlemden sonra a = 7, b = 4 olmalıdır. Bir öğrenci diyagrama iki kutu koyar: önce `a = b`, sonra `b = a`. Sonuç ne olur? Doğru diyagram nasıl kurulur?

Çözüm. Önerilen diyagram adım adım yürütülür. `a = b` kutusu çalışınca a, b'nin değerini alır: a = 7, b = 7. a'nın eski değeri 4 artık hiçbir yerde yoktur. `b = a` kutusu çalışınca b, a'nın şimdiki değerini alır: b = 7. Sonuç a = 7, b = 7 olur; takas başarısızdır.

Sorun, ilk atamanın a'nın eski değerini silmesidir. Çözüm, bu değeri önce üçüncü bir değişkende saklamaktır. Böyle kısa süreli kullanılan değişkene **geçici değişken** (*temporary variable*) denir. Doğru sıra üç kutudur: `gecici = a`, `a = b`, `b = gecici`. Yürütme: gecici = 4; a = 7; b = 4. Sonuç a = 7, b = 4'tür.

```
import java.util.Scanner;

public class Takas {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int a = giris.nextInt();
        int b = giris.nextInt();
        int gecici = a;                  // a'nın değeri saklanır
        a = b;
        b = gecici;
        System.out.println(a + " " + b);
    }
}
```

Program 4 7 girdisiyle çalıştırıldığında `7 4` yazar. Örnek, sıralı akışın temel kuralını gösterir: her kutu, bir öncekinin bıraktığı değerlerle çalışır. Bir atama, sağ tarafındaki değişkenin o anki değerini okur; eski ya da gelecekteki değerini değil.

Sık yapılan hata

Atamayı matematikteki eşitlik gibi okumak. `a = b` "a ile b eşittir" demek değildir, "b'nin şimdiki değerini a'ya kopyala" demektir. Kopyalamadan sonra b değişse a değişmez; a değiştirilse b değişmez. İki değişken yalnızca atama anında aynı değeri taşır.

4.2.5 Koşullu akış

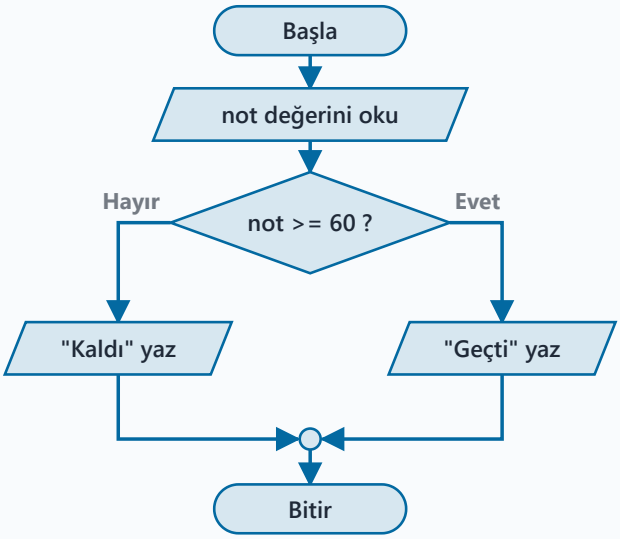
Sıralı bir algoritma her girdide aynı adımları yapar. Oysa gerçek problemlerin çoğunda yapılacak iş duruma göre değişir: not 60'ın altındaysa öğrenci kalır, şifre yanlışsa kapı açılmaz, sıcaklık sıfırın altındaysa uyarı verilir. Bu tür seçimler karar simgesiyle gösterilir.

4.2.5.1 İki yollu karar

Tanım 4.2.10

Koşullu akış (*conditional flow*), bir karar simgesinde sorulan koşulun sonucuna göre iki yoldan birinin izlendiği akıştır. Koşul doğruysa Evet yolu, yanlışsa Hayır yolu çalışır. Bu yollara **dal** (*branch*), yolun ikiye ayrılmasına **dallanma** (*branching*) denir. Dallar bir bağlayıcıda yeniden birleşir.

Şekil 4.2.9 bir sınav notuna göre “Geçti” ya da “Kaldı” yazan diyagramı gösterir. Geçme sınırı 60'tır; 60 ve üstü geçer.



Şekil 4.2.9: Koşullu akış: not 60 ya da üstüyse “Geçti”, değilse “Kaldı” yazılır. Yollar alttaki bağlayıcıda birleşir.

```
Başla
  not değerini oku
  not >= 60 ?
    Evet → "Geçti" yaz
    Hayır → "Kaldı" yaz
Bitir
```

Diyagramda üç nokta özellikle önemlidir. Karar simgesinden iki yol çıkar ama her çalıştırmada bu yollardan yalnızca biri izlenir. “Geçti” ve “Kaldı” kutuları aynı çalıştırmada asla birlikte çalışmaz. Yollar alttaki bağlayıcıda (●) yeniden birleşir; bu nokta, karar yapısının tek çıkışıdır. Son olarak koşulun sınır değeri ayrıca önem taşır. Geçme sınırı “60 ve üstü” olduğu için koşul `not >= 60` biçiminde yazılır. `not > 60` yazılsaydı tam 60 alan öğrenci kalırdı.

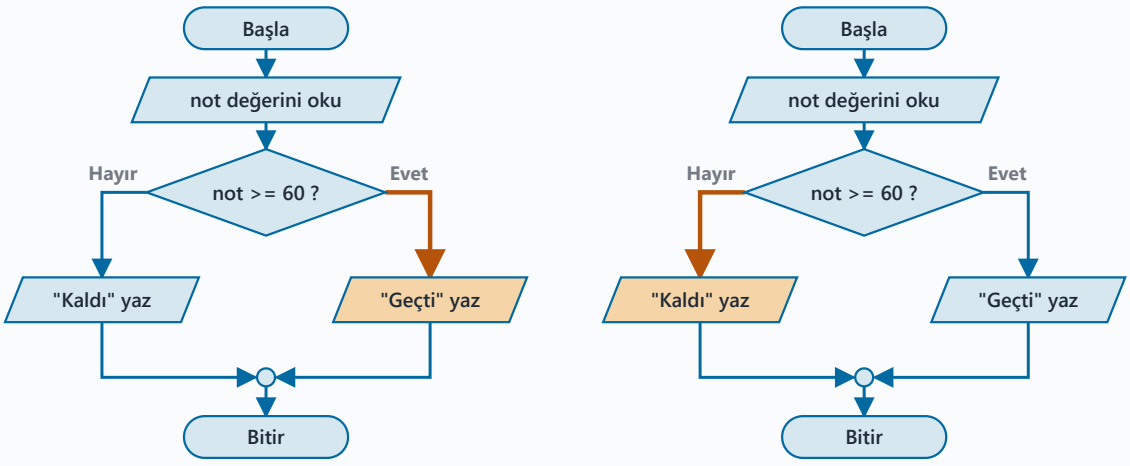
4.2.5.2 Kararı adım adım yürütmek

Koşullu bir diyagramı tam olarak anlamak için onu en az iki kez yürütmek gerekir. Girdilerden biri Evet yolunu, öbürü Hayır yolunu çalıştırmalıdır. Tablo 4.2.5 diyagramı üç girdiyle özetler: not = 72, not = 45 ve sınır değeri not = 60.

not	Karar	İzlenen ok	Çalışan kutu	Hiç çalışmayan kutu
72	72 >= 60 → doğru	Evet	"Geçti" yaz	Hayır dalı
45	45 >= 60 → yanlış	Hayır	"Kaldı" yaz	Evet dalı
60	60 >= 60 → doğru	Evet	"Geçti" yaz	Hayır dalı

Tablo 4.2.5: Geçti/Kaldı diyagramının üç çalıştırması. Her çalıştırmada kutulardan biri hiç çalışmaz.

Birinci çalıştırmada kullanıcı 72 girer ve not = 72 olur. Karar kutusunda *72 >= 60* sorulur; cevap doğrudur. Akış Evet okundan devam eder ve “Geçti” yazılır. “Kaldı” kutusu bu çalıştırmada hiç çalışmaz. Yol bağlayıcıda ana akışa döner ve Bitir'e ulaşılır. İkinci çalıştırmada not = 45 olur. *45 >= 60* yanlıştır; Hayır yolu izlenir, “Kaldı” yazılır ve bu kez “Geçti” kutusu çalışmaz. İki çalıştırma birlikte diyagramdaki her kutuyu en az bir kez çalıştırmış olur.



Şekil 4.2.10: Aynı diyagramın iki çalıştırması. (a) not = 72: Evet dalındaki “Geçti” yaz kutusu çalışır. (b) not = 45: Hayır dalındaki “Kaldı” yaz kutusu çalışır.

Üçüncü satır sınır değeridir. 60 girildiğinde *60 >= 60* doğrudur ve “Geçti” yazılır. Bu satır, koşulda `>=` yerine yanlışlıkla `>` yazılmış bir diyagramı hemen ele verir: o diyagram 60 için “Kaldı” yazardı. 72 ve 45 bu hatayı yakalayamaz, çünkü iki diyagram bu girdilerde aynı sonucu verir.

Diyagramın Java karşılığı `if` / `else` deyimidir:

```java
import java.util.Scanner;

public class GectiKaldi {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int not = giris.nextInt();        // not değerini oku
        if (not >= 60) {                  // not >= 60 ?
            System.out.println("Geçti");  // Evet dalı
        } else {
            System.out.println("Kaldı");  // Hayır dalı
        }
    }
}
```

`if` sözcüğünden sonra parantez içinde koşul yazılır. Koşul doğruysa ilk küme parantezinin içi, yanlışsa `else` sözcüğünden sonraki küme parantezinin içi çalışır. Karar simgesi parantezdeki koşula, Evet dalı ilk bloğa, Hayır dalı `else` bloğuna karşılık gelir. Kapanan son küme parantezi de bağlayıcının yerini tutar: iki blok bittikten sonra yürütme aynı noktadan sürer. Program 72, 60 ve 100 için "Geçti"; 45, 59 ve 0 için "Kaldı" yazar.

4.2.5.3 Tek yollu karar

Bazı kararlarda dallardan yalnızca birinde yapılacak iş vardır. Şekil 4.2.5'teki mutlak değer diyagramı buna örnektir: sayı negatifse işareti değiştirilir, negatif değilse hiçbir şey yapılmaz. Bu durumda karardan yine iki ok çıkar ama Hayır oku hiçbir kutuya uğramadan doğrudan birleşme noktasına gider. Java'da bu yapı `else` kısmı olmayan bir `if` ile yazılır:

```java
import java.util.Scanner;

public class MutlakDeger {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int x = giris.nextInt();
        if (x < 0) {                      // Hayır dalında kutu yok: else gerekmez
            x = -x;
        }
        System.out.println(x);
    }
}
```

Program −5 için 5, 8 için 8, 0 için 0 yazar. Üç girdi bilinçli seçilmiştir: −5 Evet dalını, 8 Hayır dalını, 0 ise sınırı dener. *0 < 0* yanlış olduğu için 0 olduğu gibi kalır; zaten mutlak değeri de 0'dır.

Karar yapısının sık görülen bir başka örneği tek/çift sınamasıdır. Bir sayının çift olup olmadığı, 2'ye bölümünden kalana bakılarak anlaşılır. **Kalan** (*remainder*) işlemi Java'da `%` ile yazılır: `7 % 2` değeri 1, `12 % 2` değeri 0'dır. Karar kutusuna `x % 2 == 0 ?` yazılır; Evet dalı "çift", Hayır dalı "tek" yazar.

```
import java.util.Scanner;

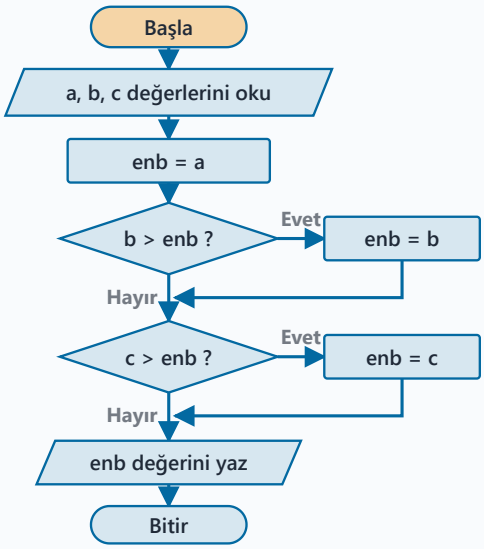
public class TekCift {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int x = giris.nextInt();
        if (x % 2 == 0) {                // x % 2 == 0 ?
            System.out.println("çift");
        } else {
            System.out.println("tek");
        }
    }
}
```

Sık yapılan hata

Tek sayıyı `x % 2 == 1 ?` koşuluyla sınamak. Bu koşul pozitif sayılarda doğru çalışır ama Java'da kalanın işareti bölünen sayının işaretini izler: `-3 % 2` değeri 1 değil, −1'dir. Bu yüzden −3 için `x % 2 == 1` yanlış çıkar ve tek bir sayı "çift" diye yazılır. `x % 2 == 0 ?` sorusu ise −3 için de doğru cevabı verir: `-3 % 2 == 0` yanlıştır ve "tek" yazılır. Kalan sıfırla karşılaştırıldığında işaret sorunu ortadan kalkar.

4.2.5.4 Üç sayının en büyüğü

İki sayının büyüğünü bulmak tek karar ister. Üç sayıda kararlar çoğalır ve diyagramın nasıl kurulacağı önem kazanır. Çok kullanılan bir yöntem "aday" fikridir. İlk sayı şimdilik en büyük kabul edilir. Sonra diğer sayılar tek tek bu adayla karşılaştırılır; daha büyüğü çıkarsa aday değişir.



Şekil 4.2.11: Üç sayının en büyüğü. enb = a ile başlanır; b daha büyükse enb = b, sonra c daha büyükse enb = c yapılır.

```
Başla
  a, b, c değerlerini oku
  enb = a
  b > enb ?  Evet → enb = b
  c > enb ?  Evet → enb = c
  enb değerini yaz
Bitir
```

Şekil 4.2.11 yöntemin diyagramıdır. İki karar arka arkaya gelir ve her birinin yalnız Evet dalında bir kutu vardır; Hayır dalları doğrudan bir sonraki adıma geçer. Yani diyagram, iki tek yollu kararın sıralı yapı içinde arka arkaya dizilmesinden oluşur. Yapıların birbirine eklenerek büyümesinin güzel bir örneğidir.

Örnek 4.2.3

Diyagram önce (a, b, c) = (4, 9, 7), sonra (3, 2, 8) için yürütülecek. Her kararda hangi yol izlenir, enb nasıl değişir?

Çözüm. Birinci çalıştırmada üç sayı okunur ve enb = 4 olur. İlk kararda *9 > 4* doğrudur; enb = 9 olur. İkinci kararda *7 > 9* yanlıştır; enb değişmez ve 9 yazılır.

Çalışan kutu	Karar	enb
enb = a	—	4
b > enb ?	9 > 4 → Evet	4
enb = b	—	9

Çalışan kutu	Karar	enb
c > enb ?	7 > 9 → Hayır	9
enb değerini yaz	çıktı: 9	9

(4, 9, 7) için izleme

İkinci çalıştırmada enb = 3 ile başlanır. *2 > 3* yanlıştır; ilk kararın Evet kutusu çalışmaz. *8 > 3* doğrudur; enb = 8 olur ve 8 yazılır.

Çalışan kutu	Karar	enb
enb = a	—	3
b > enb ?	2 > 3 → Hayır	3
c > enb ?	8 > 3 → Evet	3
enb = c	—	8
enb değerini yaz	çıktı: 8	8

(3, 2, 8) için izleme

İki çalıştırma birlikte her iki kararın her iki dalını da denemiş olur. Her karar yalnızca bir kez sorulur. Sayı sayısı artsa da yöntem değişmez: on sayı için dokuz karar arka arkaya dizilir.

```
import java.util.Scanner;

public class EnBuyuk {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int a = giris.nextInt(), b = giris.nextInt(), c = giris.nextInt();
        int enb = a;                        // ilk aday
        if (b > enb) {
            enb = b;
        }
        if (c > enb) {
            enb = c;
        }
        System.out.println(enb);
    }
}
```

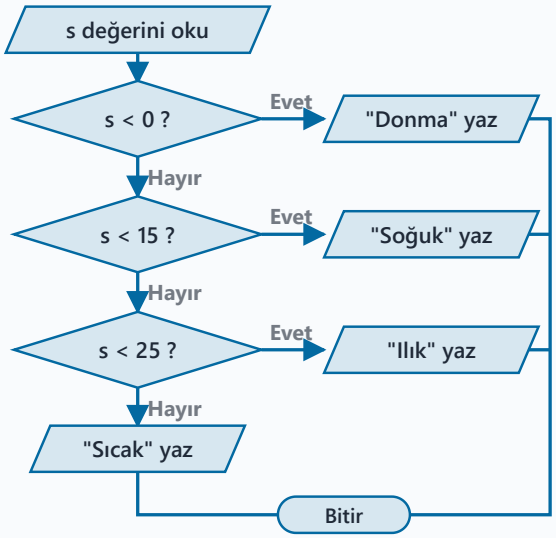
Program (4, 9, 7) için 9, (3, 2, 8) için 8 yazar. En büyük sayının başta olduğu (9, 4, 7) girdisinde iki karar da Hayır olur ve 9 yazılır. Üç sayı eşitse, örneğin (5, 5, 5), kararlar yine Hayır olur; enb = 5 kalır.

Sık yapılan hata

Adayı ilk sayı yerine sabit bir değerle, örneğin `enb = 0` ile başlatmak. Girdiler pozitif oldukça bu da doğru sonuç verir; hata ancak bütün girdiler negatif olduğunda ortaya çıkar. (−2, −7, −1) için *−2 > 0*, *−7 > 0* ve *−1 > 0* yanlıştır; aday hiç değişmez ve 0 yazılır. Oysa 0 girdilerden biri bile değildir. Doğru diyagram `enb = a` ile başlar ve bu girdide −1 yazar.

4.2.5.5 Çok yollu karar: karar zinciri

Bir karar kutusunun yalnızca iki çıkışı vardır. Seçenek sayısı ikiden fazlaysa kararlar bir zincir gibi dizilir: her karar bir öncekinin Hayır dalına yerleştirilir. Buna **çok yollu karar** (*multi-way decision*) denir.



Şekil 4.2.12: Karar zinciri: sıcaklık s < 0 ise “Donma”, s < 15 ise “Soğuk”, s < 25 ise “Ilık”, değilse “Sıcak” yazılır.

Şekil 4.2.12 bir sıcaklık değerini dört sınıftan birine ayırır. Zincirin çalışma kuralı şudur: kararlar yukarıdan aşağıya sırayla sorulur; *ilk doğru koşulun dalı* çalışır ve ondan sonraki kararlar hiç sorulmaz. Hiçbir koşul doğru değilse en alttaki Hayır dalı çalışır.

s	Sorulan kararlar	Karar sayısı	Çıktı
−5	s < 0 ? Evet	1	Donma
10	s < 0 ? Hayır · s < 15 ? Evet	2	Soğuk
15	s < 0 ? Hayır · s < 15 ? Hayır · s < 25 ? Evet	3	Ilık
18	s < 0 ? Hayır · s < 15 ? Hayır · s < 25 ? Evet	3	Ilık
30	s < 0 ? Hayır · s < 15 ? Hayır · s < 25 ? Hayır	3	Sıcak

Tablo 4.2.6: Karar zincirinin beş girdiyle yürütülmesi. Sorulan karar sayısı, ilk doğru koşulun konumuna bağlıdır.

Tablo 4.2.6'daki s = 18 satırı adım adım şöyle okunur. İlk karar *18 < 0* sorusudur; yanlıştır ve Hayır dalı ikinci karara gider. *18 < 15* de yanlıştır. Üçüncü karar *18 < 25* doğrudur; “Ilık” yazılır. s = −5 için ise ilk karar doğrudur ve öbür iki karar hiç sorulmaz.

Zincirdeki her karar yalnızca *üst* sınırı denetler. s < 15 sorusu sorulurken s'nin 0 ya da daha büyük olduğu kesindir; öyle olmasaydı yürütme bu karara hiç gelmezdi. Bu yüzden “0 ≤ s ve s < 15” gibi iki taraflı bir koşula gerek kalmaz. Sınır değerlerinin hangi sınıfa düştüğü de bu kuraldan anlaşılır: s = 15 için *15 < 15* yanlıştır, karar zincirde ilerler ve 15 “Ilık” olur. s = 0 ise “Soğuk”tur.

Kararların sırası önemlidir. Kararlar ters sırada dizilse, yani en başta `s < 25 ?` sorulsa, s = −5 için *−5 < 25* doğru çıkar ve “Ilık” yazılır; “Donma” dalına hiç ulaşılmaz. s = 10 için de aynı şey olur. Genel kural kısadır: *dar koşul önce, geniş koşul sonra* sorulur. Seçenek sayısı n ise n − 1 karar gerekir; dört sınıf için üç karar vardır.

Java'da karar zinciri `if … else if … else` biçiminde yazılır. Her `else if` bir önceki koşulun Hayır dalıdır; son `else` hiçbir koşulun tutmadığı durumdur:

```
import java.util.Scanner;

public class Sicaklik {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int s = giris.nextInt();
        if (s < 0) {
            System.out.println("Donma");
        } else if (s < 15) {
            System.out.println("Soğuk");
        } else if (s < 25) {
            System.out.println("Ilık");
        } else {
            System.out.println("Sıcak");
        }
    }
}
```

Program −5 için “Donma”, 0 ve 10 için “Soğuk”, 15 ve 18 için “Ilık”, 25 ve 30 için “Sıcak” yazar. Bu yedi girdi zincirin dört çıkışını ve her sınır değerini dener.

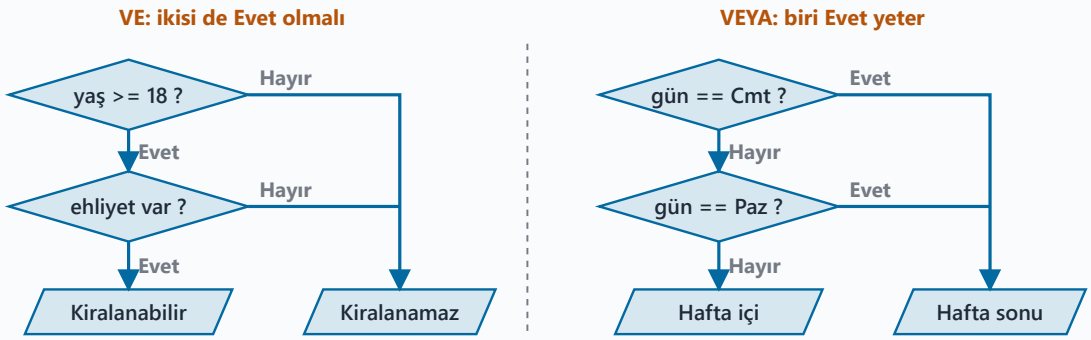
4.2.5.6 Bileşik koşul: VE ve VEYA

Bazı kararlar iki koşulun birlikte değerlendirilmesini ister. Araç kiralayan bir şirket, müşterinin hem 18 yaşını doldurmuş olmasını hem de ehliyetinin bulunmasını şart koşar. Bir takvim uygulaması, gün cumartesi *ya da* pazar ise “Hafta sonu” yazar. İki koşulu birleştiren böyle ifadelere **bileşik koşul** (*compound condition*) denir.

Tanım 4.2.11

VE ile bağlanan bileşik koşul, iki koşul da doğruysa doğrudur; biri bile yanlışsa yanlıştır. **VEYA** ile bağlanan bileşik koşul, koşullardan en az biri doğruysa doğrudur; ikisi de yanlışsa yanlıştır. Java'da VE `&&`, VEYA `||` işleciyle yazılır.

Bileşik koşul tek bir karar kutusuna yazılabilir: `yaş >= 18 VE ehliyet var ?`. Diyagramda ayrıntı gerekiyorsa iki ayrı karara açılır. Şekil 4.2.13 iki bağlacın iki kararla nasıl çizildiğini gösterir.



Şekil 4.2.13: Bileşik koşulun iki kararla çizimi. Solda VE: iki kararın da Evet olması gerekir. Sağda VEYA: kararlardan biri Evet ise yeter.

VE yapısında ikinci karar, birincinin *Evet* dalına yerleştirilir. Birinci karar Hayır ise sonuç zaten Hayır'dır ve ikinci karar hiç sorulmaz. VEYA yapısında ikinci karar birincinin *Hayır* dalındadır. Birinci karar Evet ise sonuç zaten Evet'tir ve ikinciye gerek kalmaz. Tablo 4.2.7 ile Tablo 4.2.8 bu davranışı bütün durumlar için gösterir.

yaş	ehliyet	yaş >= 18 ?	ehliyet var ?	Sonuç
20	var	Evet	Evet	Kiralanabilir
20	yok	Evet	Hayır	Kiralanamaz
16	var	Hayır	sorulmaz	Kiralanamaz
16	yok	Hayır	sorulmaz	Kiralanamaz

Tablo 4.2.7: VE kararı. Birinci karar Hayır olunca ikinci karar sorulmaz.

gün	gün == Cmt ?	gün == Paz ?	Sonuç
Cmt	Evet	sorulmaz	Hafta sonu
Paz	Hayır	Evet	Hafta sonu
Sal	Hayır	Hayır	Hafta içi

Tablo 4.2.8: VEYA kararı. Birinci karar Evet olunca ikinci karar sorulmaz.

Tanım 4.2.12

Kısa devre değerlendirme (*short-circuit evaluation*), bileşik koşulun sonucu ilk koşuldan belli olduğunda ikinci koşulun hiç hesaplanmamasıdır. `A && B` ifadesinde A yanlışsa B'ye bakılmaz; `A || B` ifadesinde A doğruysa B'ye bakılmaz.

Java'daki `&&` ve `||` işleçleri, Şekil 4.2.13'teki iki kararlı diyagramlarla aynı biçimde çalışır. Bu davranış zaman kazandırmakla kalmaz, bazı hataları da önler. `sayac > 0 && toplam / sayac > 50` koşulunda sayac 0 ise ikinci kısım hiç hesaplanmaz; böylece sıfıra bölme hatası oluşmaz.

```
import java.util.Scanner;

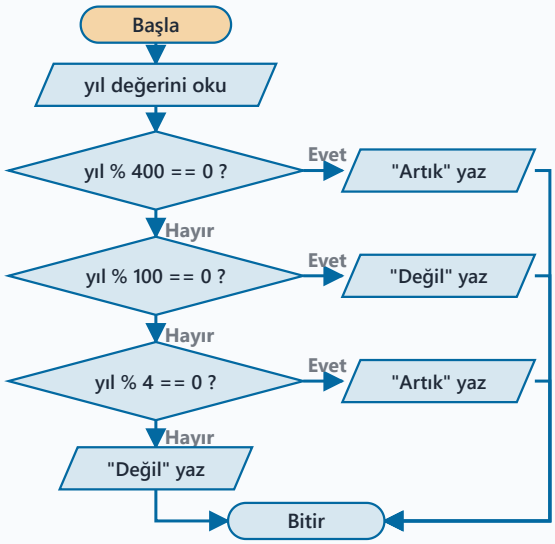
public class Kiralama {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int yas = giris.nextInt();
        boolean ehliyet = giris.nextBoolean();   // true: ehliyet var
        String gun = giris.next();               // Pzt, Sal, ..., Cmt, Paz
        if (yas >= 18 && ehliyet) {
            System.out.println("Kiralanabilir");
        } else {
            System.out.println("Kiralanamaz");
        }
        if (gun.equals("Cmt") || gun.equals("Paz")) {
            System.out.println("Hafta sonu");
        } else {
            System.out.println("Hafta içi");
        }
    }
}
```

Program “20 true Cmt” girdisiyle “Kiralanabilir” ve “Hafta sonu”; “16 true Paz” girdisiyle “Kiralanamaz” ve “Hafta sonu” yazar. Bir bileşik koşulu test etmek için her kararın iki çıkışı da denenir. VE örneğinde (20, var), (20, yok) ve (16, —) girdileri üç farklı yolu çalıştırır. 16 yaşında birinin ehliyeti olup olmaması sonucu değiştirmez, çünkü ikinci karar sorulmaz.

4.2.5.7 Artık yıl

Karar zincirinde sıranın önemini gösteren klasik örnek, artık yıl kuralıdır. Gregoryen takvimde kural üç parçalıdır: 4'e bölünen yıllar artık yıldır; ama 100'e bölünenler artık değildir; 400'e bölünenler ise yine artıktır. Artık yılda şubat 29 gün çeker.

Kuralın istisnaları vardır, istisnaların da istisnası vardır. Böyle bir kural diyagrama çevrilirken en dar koşul en başa konur. 400'e bölünme, 100'e bölünmenin; 100'e bölünme de 4'e bölünmenin özel bir durumudur. Bu yüzden kararlar 400, 100, 4 sırasıyla sorulur.



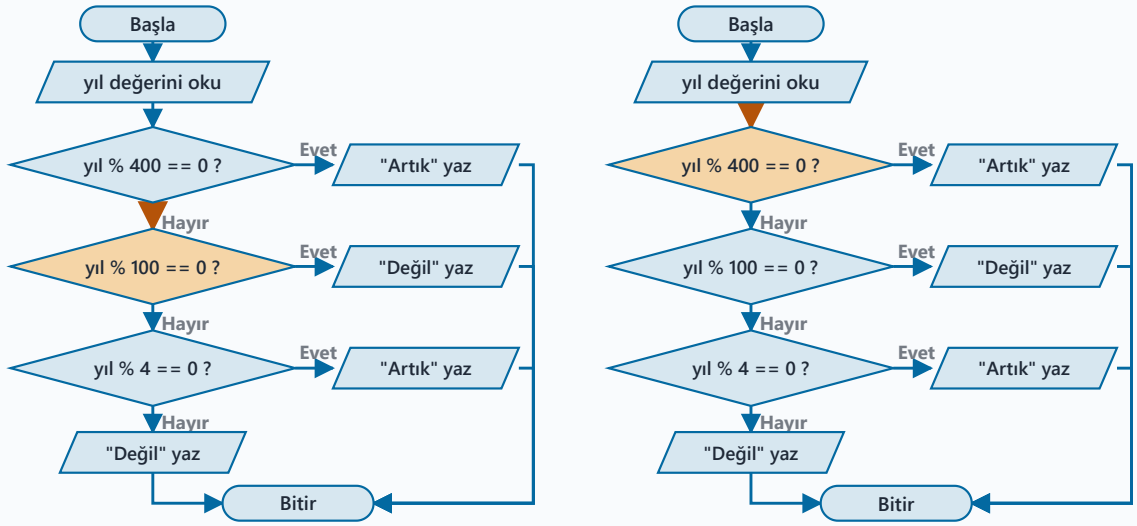
Şekil 4.2.14: Artık yıl diyagramı: yıl % 400, yıl % 100 ve yıl % 4 kararlarından oluşan üç kararlı zincir; dört çıkış.

```
Başla
  yıl değerini oku
  yıl % 400 == 0 ?  Evet → "Artık" yaz
  yıl % 100 == 0 ?  Evet → "Değil" yaz
  yıl % 4 == 0 ?    Evet → "Artık" yaz
  Hayır → "Değil" yaz
Bitir
```

yıl	yıl % 400	yıl % 100	yıl % 4	Sorulan karar	Çıktı
2024	24 → Hayır	24 → Hayır	0 → Evet	3	Artık
1900	300 → Hayır	0 → Evet	sorulmaz	2	Değil
2000	0 → Evet	sorulmaz	sorulmaz	1	Artık
2023	23 → Hayır	23 → Hayır	3 → Hayır	3	Değil

Tablo 4.2.9: Artık yıl diyagramının dört girdiyle yürütülmesi. Her hücrede kalan ve kararın sonucu yazılıdır.

Tablo 4.2.9 dört girdinin her birinin hangi yoldan geçtiğini gösterir. 2024 için *2024 % 400 = 24* ve *2024 % 100 = 24* sıfır değildir; iki karar da Hayır'dır. *2024 % 4 = 0* olduğu için üçüncü karar Evet'tir ve "Artık" yazılır. 1900'de *1900 % 400 = 300* sıfır değildir ama *1900 % 100 = 0*'dır. İkinci karar Evet olur, "Değil" yazılır ve üçüncü karar hiç sorulmaz. 1900, 4'e bölündüğü hâlde artık yıl değildir; yalnız `yıl % 4 == 0` sorusuna bakan bir diyagram burada yanılırdı. 2000'de ilk karar Evet'tir: 2000, 100'e bölündüğü hâlde 400'e de bölündüğü için artık yıldır. 2023 ise üç kararın da Hayır olduğu son dalı çalıştırır.



Şekil 4.2.15: Artık yıl zincirinde iki erken çıkış. (a) 1900: ikinci karar (yıl % 100 == 0) Evet; üçüncü karar sorulmaz. (b) 2000: ilk karar Evet; kalan iki karar sorulmaz.

Dört girdi, diyagramın dört çıkışının hepsini birer kez çalıştırır. Bir karar zincirini test ederken amaç tam olarak budur. Java karşılığı yine bir `if … else if … else` zinciridir:

```
import java.util.Scanner;

public class ArtikYil {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int yil = giris.nextInt();
        if (yil % 400 == 0) {
            System.out.println("Artık");
        } else if (yil % 100 == 0) {
            System.out.println("Değil");
        } else if (yil % 4 == 0) {
            System.out.println("Artık");
        } else {
            System.out.println("Değil");
        }
    }
}
```

Program 2024 ve 2000 için “Artık”, 1900 ve 2023 için “Değil” yazar. Yakın geleceğin ilk istisnası 2100 yılıdır: 2100, 100'e bölünür ama 400'e bölünmez; program bu yıl için “Değil” yazar. 1600 ise 400'e bölündüğü için artık yıldır.

Sık yapılan hata

Zinciri geniş koşulla başlatmak. Önce `yıl % 4 == 0 ?` sorulup Evet dalına doğrudan "Artık" yazılırsa 1900 ve 2100 de artık yıl sayılır. 100 ve 400 kuralları yalnızca 4'e bölünen yılları ilgilendirdiği için bu kurallara hiç sıra gelmez. Zincirdeki kararlar dar koşuldan geniş koşula doğru dizilmelidir.

4.2.6 Tekrarlı akış

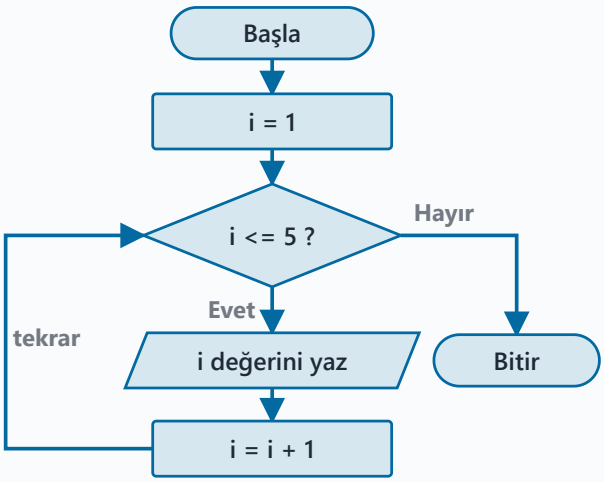
Bilgisayarı güçlü kılan şey, aynı işi yorulmadan binlerce kez yapabilmesidir. Bir sınıftaki 30 öğrencinin notunu toplamak, 30 kez aynı iki adımı yapmaktır: notu oku, toplama ekle. Bu adımları diyagrama 30 kez çizmek yerine bir kez çizip geri dönen bir okla tekrar ettirmek gerekir.

4.2.6.1 Döngü yapısı

Tanım 4.2.13

Tekrarlı akış (*repetitive flow*) ya da **döngü** (*loop*), bir grup adımın bir koşul doğru oldukça tekrar tekrar çalıştığı akıştır. Tekrarlanan adımlara **döngü gövdesi** (*loop body*), gövdenin bir kez çalışmasına **tur** (*iteration*) denir. Gövdenin sonundan koşula geri dönen ok döngüyü oluşturur.

Şekil 4.2.16 1'den 5'e kadar sayıları yazan diyagramı gösterir. Diyagramda üç parça vardır. `i = 1` kutusu sayacın ilk değerini verir. `i <= 5 ?` karar kutusu döngünün devam edip etmeyeceğine karar verir. Evet dalındaki iki kutu, yani "i değerini yaz" ve `i = i + 1`, döngü gövdesidir. Gövdenin sonundan çıkan ok sola döner ve yeniden karar kutusuna girer.



Şekil 4.2.16: Tekrarlı akış: i = 1 ile başlanır; i <= 5 oldukça i yazılır ve 1 artırılır. Soldaki "tekrar" oku döngüyü oluşturur.

```
Başla
  i = 1
  i <= 5 ?
    Evet → i değerini yaz
           i = i + 1
           karara dön
    Hayır → Bitir
```

Bu diyagramda koşul gövdeden *önce* sorulur. Koşul daha ilk denemede yanlışsa gövde hiç çalışmaz. Böyle döngülere **ön koşullu döngü** (*pre-test loop*) denir. Koşul yanlış olduğunda Hayır dalı döngünün dışına, Bitir'e çıkar.

Döngünün bir gün bitmesi için tekrar yolunda koşulu değiştiren bir işlem bulunmalıdır. Burada bu işlem `i = i + 1`'dir. i her turda büyür ve sonunda 5'i geçer. Bu kutu unutulsa i hep 1 kalır, *1 <= 5* her seferinde doğru çıkar ve diyagram durmadan 1 yazar. Böyle bitmeyen döngüye **sonsuz döngü** (*infinite loop*) denir.

4.2.6.2 Döngüyü adım adım yürütmek

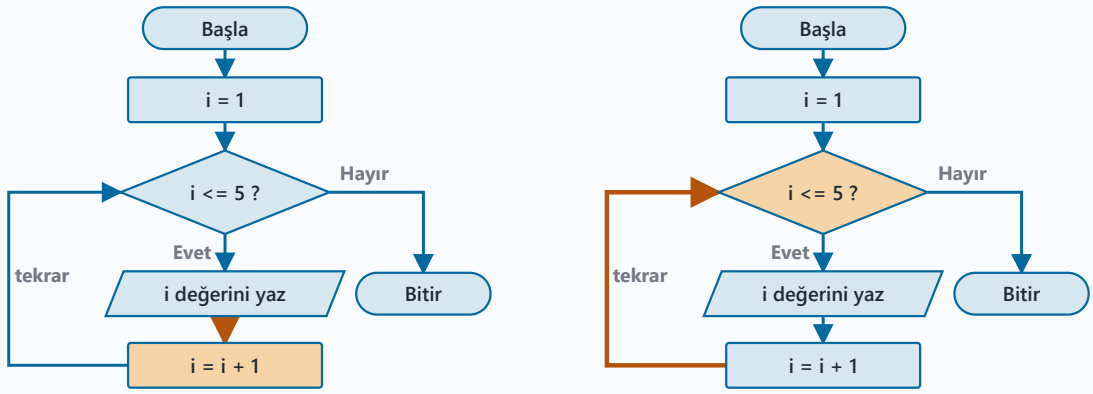
Döngüde aynı kutular tekrar tekrar çalıştığı için izleme tablosunda her tur bir satırla gösterilir. Tablo 4.2.10 diyagramın bütün turlarını verir.

Tur	i (karardan önce)	Karar	Yazılan	i (turun sonunda)
1	1	1 <= 5 → doğru	1	2
2	2	2 <= 5 → doğru	2	3
3	3	3 <= 5 → doğru	3	4
4	4	4 <= 5 → doğru	4	5

Tur	i (karardan önce)	Karar	Yazılan	i (turun sonunda)
5	5	5 <= 5 → doğru	5	6
—	6	6 <= 5 → yanlış	—	—

Tablo 4.2.10: 1'den 5'e sayıları yazan döngünün izleme tablosu. Son satır, döngüden çıkılan son karardır.

Yürütme şöyle ilerler. Sayaç ilk değerini alır: i = 1. Karar kutusunda *1 <= 5* sorulur; doğrudur ve Evet dalına, yani döngü gövdesine girilir. 1 yazılır. `i = i + 1` kutusu i'yi 2 yapar. Geri dönen ok yürütmeyi yeniden karar kutusuna götürür. *2 <= 5* de doğrudur; 2 yazılır, i = 3 olur. Aynı döngü 3, 4 ve 5 için de tekrarlanır. 5 yazıldıktan sonra i = 6 olur. Karar kutusunda *6 <= 5* sorulur; bu kez cevap yanlıştır ve Hayır dalı Bitir'e götürür.



Şekil 4.2.17: Döngünün iki anı. (a) 1. turun sonu: i = 1 + 1 = 2 olur; geri dönen ok karara götürür. (b) Son karar: 6 <= 5 yanlış; Hayır dalı döngüden çıkar.

Tablodan iki sayım çıkar. Gövde 5 kez çalıştı ama koşul 6 kez soruldu. Bu her ön koşullu döngü için geçerlidir: karar, gövdeden bir fazla çalışır, çünkü son deneme döngüyü bitiren yanlış cevaptır. Döngüden çıkıldığında i'nin değeri 5 değil 6'dır. Sayacın döngüden sonraki değeri sık sık sorulan ve sık sık yanlış cevaplanan bir sorudur.

Java'da ön koşullu döngü `while` ile yazılır. Parantezdeki koşul doğru oldukça küme parantezinin içi tekrar çalışır; kapanan küme parantezi geri dönen okun yerini tutar:

```
public class BirdenBese {
    public static void main(String[] args) {
        int i = 1;                      // sayacın ilk değeri
        while (i <= 5) {                // karar: i <= 5 ?
            System.out.print(i + " ");  // i değerini yaz
            i = i + 1;                  // sayacı güncelle
        }                               // geri dönen ok: karara dön
        System.out.println();
    }
}
```

Program `1 2 3 4 5` yazar. `System.out.print`, `println`'den farklı olarak satır sonu eklemez; sayılar bu yüzden aynı satırda, aralarında birer boşlukla yazılır. Döngüden sonraki `System.out.println()` satırı yalnızca satırı bitirir.

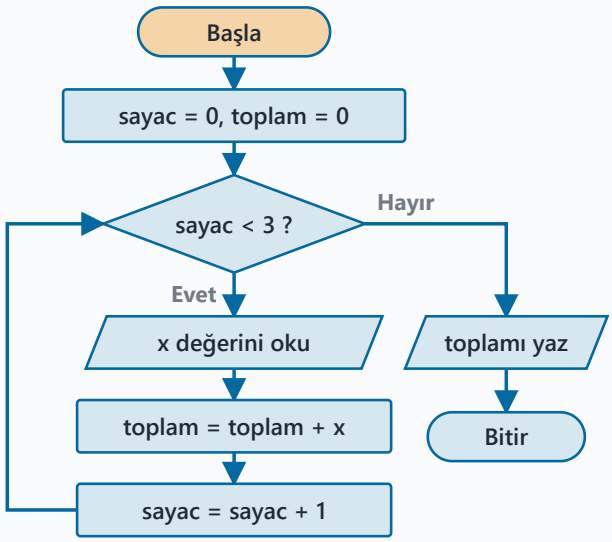
4.2.6.3 Sayaç ve biriktirici

Döngülerin çoğunda iki tür değişken bulunur. Biri döngünün kaçıncı turda olduğunu sayar, öbürü turdan tura biriken sonucu tutar.

Tanım 4.2.14

Sayaç (*counter*), kaç tekrar yapıldığını izleyen değişkendir; her turda 1 artar. **Biriktirici** (*accumulator*), kısmi sonucu tutan değişkendir; her turda yeni değer ona eklenir (ya da onunla çarpılır). Biriktiricinin başlangıç değeri işleme göre seçilir: toplam için 0, çarpım için 1.

Başlangıç değerinin seçimi rastgele değildir. Toplama için 0 seçilir, çünkü bir sayıya 0 eklemek onu değiştirmez; ilk turdan sonra toplam, ilk sayının kendisi olur. Çarpım için aynı rol 1'indir. Çarpım biriktiricisi yanlışlıkla 0 ile başlarsa her turda 0 ile çarpılır ve sonuç hep 0 çıkar. Örneğin 1'den 5'e sayıların çarpımı *1 × 1 × 2 × 3 × 4 × 5 = 120*'dir; biriktirici 0 ile başlarsa *0 × 1 × 2 × 3 × 4 × 5 = 0* bulunur.



Şekil 4.2.18: Sayaç ve biriktirici: üç sayı okunur ve toplanır. sayac < 3 oldukça x okunur, toplam = toplam + x ve sayac = sayac + 1 yapılır.

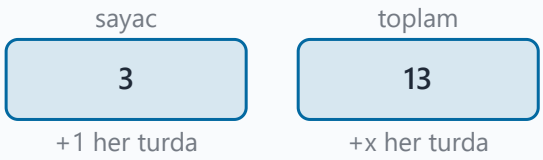
```
Başla
  sayac = 0, toplam = 0
  sayac < 3 ?
    Evet → x değerini oku
           toplam = toplam + x
           sayac = sayac + 1
           karara dön
    Hayır → toplamı yaz
Bitir
```

Şekil 4.2.18 üç sayı okuyup toplamını yazan diyagramdır. `sayac` sayaç, `toplam` biriktiricidir. İkisi de döngüden önce, bir kez 0 değerini alır. Tablo 4.2.11 girdiler 4, 7 ve 2 iken her turun sonundaki değerleri verir.

An	x	sayac	toplam	Sonraki karar
başlangıç	—	0	0	0 < 3 → doğru
1. tur	4	1	4	1 < 3 → doğru
2. tur	7	2	11	2 < 3 → doğru
3. tur	2	3	13	3 < 3 → yanlış

Tablo 4.2.11: Girdiler 4, 7, 2 için sayaç ve biriktiricinin tur tur değerleri.

Birinci turda x = 4 okunur. Biriktirici *toplam = 0 + 4 = 4* olur, sayaç *sayac = 0 + 1 = 1* olur ve ok karara döner. İkinci turda x = 7 okunur; *toplam = 4 + 7 = 11*, sayac = 2. Üçüncü turda x = 2 okunur; *toplam = 11 + 2 = 13*, sayac = 3. Karar kutusunda *3 < 3* sorulur; yanlıştır ve döngüden çıkılır. Toplam yazılır: 13. Sayaç döngünün kaç kez döndüğünü, biriktirici sonucu tuttu.



Şekil 4.2.19: Döngü bittiğinde sayaç 3, biriktirici 13'tür. Sayaç her turda 1, biriktirici her turda x kadar artmıştır.

```java
import java.util.Scanner;

public class UcSayiToplami {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int sayac = 0, toplam = 0;
        while (sayac < 3) {
            int x = giris.nextInt();      // x değerini oku
            toplam = toplam + x;          // biriktirici
            sayac = sayac + 1;            // sayaç
        }
        System.out.println(toplam);
    }
}
```

Program 4 7 2 girdisiyle 13, 10 −3 5 girdisiyle 12 yazar. Java kodunda `x` değişkeni döngünün içinde tanımlanmıştır. Her turda yeni bir değer okunur ve önceki turdaki x değeri artık gerekmez.

Sık yapılan hata

Biriktiriciyi döngünün içinde sıfırlamak. `toplam = 0` kutusu yanlışlıkla döngü gövdesinin başına çizilirse her tur, önceki turların toplamını silerek başlar. Girdiler 4, 7, 2 için tablo şöyle olur:

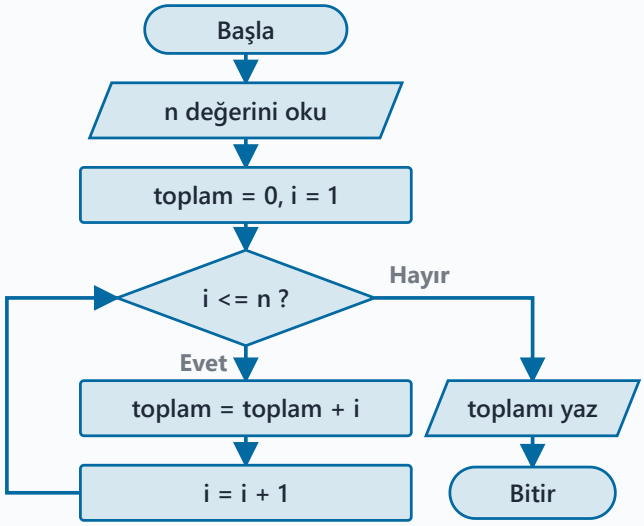
Tur	x	toplam (sıfırlandıktan sonra)	toplam (eklemeden sonra)	sayac
1. tur	4	0	4	1
2. tur	7	0	7	2
3. tur	2	0	2	3

Hatalı diyagramın izlemesi: toplam her turda sıfırlanıyor.

13 yerine 2 yazılır, yani yalnız son değer kalır. Başlangıç değeri veren kutular döngüden önce, bir kez çalışmalıdır.

4.2.6.4 1'den n'ye toplam

Sayaç ve biriktiricinin birlikte kullanıldığı klasik bir örnek, 1'den n'ye kadar sayıların toplamıdır. Bu kez sayaç yalnızca tur saymakla kalmaz, toplanacak sayının kendisi olur.



Şekil 4.2.20: 1'den n'ye toplam. n okunur; toplam = 0, i = 1 ile başlanır; i <= n oldukça toplam = toplam + i ve i = i + 1 yapılır.

```
Başla
  n değerini oku
  toplam = 0, i = 1
  i <= n ?
    Evet → toplam = toplam + i
           i = i + 1
           karara dön
    Hayır → toplamı yaz
Bitir
```

Diyagramda `i` sayaç, `toplam` biriktiricidir. Tablo 4.2.12 n = 4 için yürütmeyi gösterir. Beklenen sonuç *1 + 2 + 3 + 4 = 10*'dur.

Tur	i	Karar	toplam	i (güncellenen)
1. tur	1	1 <= 4 → doğru	0 + 1 = 1	2
2. tur	2	2 <= 4 → doğru	1 + 2 = 3	3
3. tur	3	3 <= 4 → doğru	3 + 3 = 6	4
4. tur	4	4 <= 4 → doğru	6 + 4 = 10	5
çıkış	5	5 <= 4 → yanlış	10	—

Tablo 4.2.12: 1'den n'ye toplam diyagramının n = 4 için izleme tablosu.

Tablonun her satırı bir turdur. İlk turda *1 <= 4* doğrudur; toplam 0 + 1 = 1 olur, i 2'ye çıkar. İkinci turda toplam 1 + 2 = 3, üçüncüde 3 + 3 = 6, dördüncüde 6 + 4 = 10 olur. i = 5 olduğunda *5 <= 4* yanlıştır ve döngüden çıkılır. Gövde n = 4 kez, karar n + 1 = 5 kez çalıştı. Ekrana 10 yazılır.

Sonuç bir formülle de denetlenebilir. 1'den n'ye sayıları bir kez artan, bir kez azalan sırada alt alta yazıp sütun sütun toplamak, formülü adım adım verir. n = 4 için:

artan	1	2	3	4
azalan	4	3	2	1
sütun toplamı	5	5	5	5

1'den 4'e toplamın iki kez yazılması

Her sütunun toplamı *n + 1 = 5*'tir ve n = 4 sütun vardır. İki satırın toplamı *4 × 5 = 20* eder. Bu, aranan toplamın iki katıdır; aranan toplam *20 ÷ 2 = 10* olur. Genel formül *n(n + 1) / 2*'dir. Formül, diyagramın doğru çalıştığını bağımsız bir yoldan doğrular.

n = 0 girilirse ne olur? i = 1 iken *1 <= 0* sorusu ilk denemede yanlıştır. Gövde hiç çalışmaz ve 0 yazılır. Bu da doğru cevaptır: hiç sayı toplanmamıştır. Formül de *0 × 1 / 2 = 0* verir. Ön koşullu döngünün "gövde hiç çalışmayabilir" özelliği burada işe yarar.

4.2.6.5 Sayaçlı ve bekçi değerli döngü

Şimdiye kadarki döngülerin kaç kez döneceği önceden biliniyordu: 5 kez, 3 kez, n kez. Bazı işlerde bu sayı bilinmez. Kasadaki görevli kaç ürün okutacağını önceden bilmez; ürünler bittiğinde "bitti" tuşuna basar. Bir öğretmen notları girerken not sayısını saymak yerine sona özel bir değer yazar. Böyle bir özel değere bekçi değer denir.

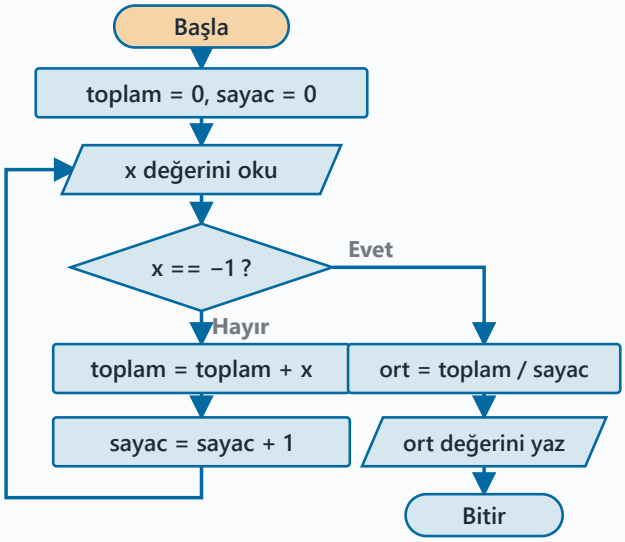
Tanım 4.2.15

Sayaçlı döngü (*counter-controlled loop*), tekrar sayısının önceden bilindiği ve bir sayaçla denetlendiği döngüdür. **Bekçi değer** (*sentinel value*), verinin bittiğini bildiren ve gerçek veride görülemeyecek biçimde seçilen özel bir değerdir. **Bekçi değerli döngü** (*sentinel-controlled loop*), bekçi değer okunana kadar dönen döngüdür.

	Sayaçla denetlenen	**Bekçi değerle denetlenen**
Tekrar sayısı	Önceden bilinir (n kez)	Bilinmez; özel bir değer gelince durur
Karar	`i <= n ?`	`x == -1 ?` (bekçi geldi mi?)
Örnek	Sınıftaki 30 öğrencinin notunu oku	Kasada okutulan ürünler; "bitti" tuşuna kadar
Java'da çoğunlukla	`for`	`while`

Tablo 4.2.13: Sayaçlı ve bekçi değerli döngünün karşılaştırılması.

Bekçi değerin seçimi veriye bağlıdır. Notlar 0 ile 100 arasındaysa −1 uygun bir bekçidir, çünkü hiçbir öğrenci −1 almaz. 0 ise uygun değildir: bir öğrencinin notu gerçekten 0 olabilir ve bu not yanlışlıkla "veri bitti" diye yorumlanır.



Şekil 4.2.21: Bekçi değerli döngü: x == −1 olana kadar oku, topla, say; sonra ortalamayı yaz. Geri dönen ok oku kutusuna döner.

```
Başla
  toplam = 0, sayac = 0
  x değerini oku
  x == -1 ?  Evet → 7. satıra git
  toplam = toplam + x
  sayac = sayac + 1;  3. satıra dön
  ort = toplam / sayac
  ort değerini yaz
Bitir
```

Şekil 4.2.21 bekçi değerli döngünün iki önemli ayrıntısını gösterir. Birincisi, karar kutusu oku kutusunun hemen ardından, toplama ve saymadan *önce* gelir. Böylece bekçi değer okunduğunda toplama da sayaca da katılmadan döngüden çıkılır. İkincisi, geri dönen ok karar kutusuna değil, oku kutusunun önüne döner. Her turda yeni bir sayı okunmalıdır; ok karara dönseydi aynı x tekrar tekrar işlenirdi.

Okunan x	x == −1 ?	toplam	sayac
80	Hayır	80	1
64	Hayır	144	2
90	Hayır	234	3
−1	Evet (bekçi)	234	3

Tablo 4.2.14: Girdiler 80, 64, 90, −1 için bekçi değerli döngünün izleme tablosu.

Tablo 4.2.14 girdiler 80, 64, 90, −1 iken yürütmeyi verir. Her okunan not önce bekçiyle karşılaştırılır. 80, 64 ve 90 gerçek nottur; toplama eklenir ve sayılır. −1 okunduğunda karar Evet olur ve döngü biter. Ortalama *234 / 3 = 78*'dir. Gövde 3 kez, karar 4 kez çalıştı. Kaç not girileceği önceden bilinmiyordu; döngüyü verinin kendisi durdurdu.

İlk girdi bekçi değerse gövde hiç çalışmaz ve sayaç 0 kalır. Bu durumda ortalama hesabı sıfıra bölmeye dönüşür. Bu yüzden bölmeden önce `sayac > 0 ?` diye sorulur. Java karşılığı bu denetimi de içerir:

```
import java.util.Scanner;

public class NotOrtalamasi {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int toplam = 0, sayac = 0;
        int x = giris.nextInt();          // ilk not döngüden önce okunur
        while (x != -1) {                 // bekçi değer geldi mi?
            toplam = toplam + x;
            sayac = sayac + 1;
            x = giris.nextInt();          // sonraki not: geri dönen ok oku kutusuna
        }
        if (sayac > 0) {
            double ort = (double) toplam / sayac;
            System.out.println(ort);
        } else {
            System.out.println("Not girilmedi");
        }
    }
}
```

Java kodunda ilk not döngüden önce bir kez okunur, sonraki notlar gövdenin sonunda okunur. Diyagramdaki "geri dönen ok oku kutusuna gider" ayrıntısının `while` ile yazılışı budur. Ortalama ondalıklı çıkabileceği için `double` türünde hesaplanır; `(double) toplam` ifadesi bölmeden önce toplamı ondalıklı sayıya çevirir. Program 80 64 90 −1 için `78.0`, 55 70 −1 için `62.5`, yalnızca −1 için "Not girilmedi" yazar. Java ondalıklı sayıları ekrana nokta ile yazar; metinde aynı değerler 78,0 ve 62,5 diye okunur.

Sık yapılan hata

Bekçi değeri işlemden sonra sorgulamak. Karar kutusu toplama ve sayma kutularının altına çizilirse −1 de veri gibi işlenir:

Okunan x	toplam	sayac	x == −1 ?
80	80	1	Hayır
64	144	2	Hayır
90	234	3	Hayır

Okunan x	toplam	sayac	x == −1 ?
−1	233	4	Evet

Hatalı sıra: önce topla ve say, sonra bekçiyi sor.

Toplam 234 yerine 233, sayaç 3 yerine 4 olur ve ortalama 78 yerine *233 / 4 = 58,25* çıkar. Bekçi değerin kararı, değeri kullanan her işlemden önce gelmelidir.

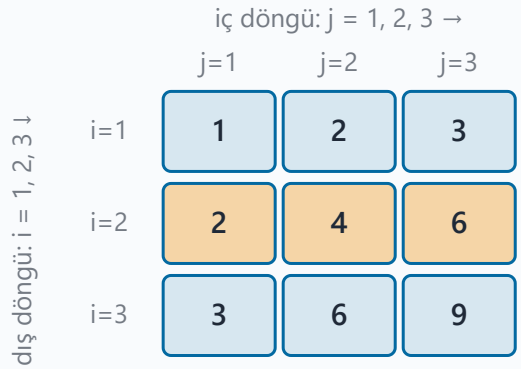
4.2.6.6 İç içe döngüler

Yapısal programlamanın tek giriş, tek çıkış kuralı bir yapıyı başka bir yapının içine koymaya izin verir. Bir kararın dalında başka bir karar bulunabilir. Bir döngünün gövdesinde ikinci bir döngü bulunabilir.

Tanım 4.2.16

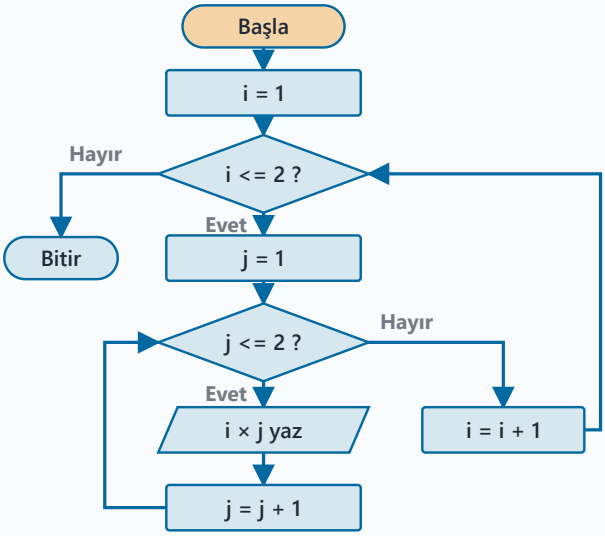
İç içe döngü (*nested loop*), gövdesinde başka bir döngü bulunan döngüdür. Dıştakine **dış döngü** (*outer loop*), içtekine **iç döngü** (*inner loop*) denir. Dış döngünün her turunda iç döngü baştan sona çalışır.

Çarpım tablosu iç içe döngünün en tanıdık örneğidir. Dış döngü satırlarda (i), iç döngü sütunlarda (j) gezinir; her hücreye i × j yazılır. Şekil 4.2.22 3 × 3'lük tabloyu gösterir. Dış döngü 3 tur döner, iç döngü de her dış turda 3 tur döner. İç gövde toplam *3 × 3 = 9* kez çalışır.



Şekil 4.2.22: İç içe döngü ızgarası. Dış döngü satırlarda (i = 1, 2, 3), iç döngü sütunlarda (j = 1, 2, 3) gezinir; hücrelerde i × j yazılıdır. Vurgulu satır i = 2 iken j'nin üç turudur.

Diyagramı küçük tutmak için 2 × 2'lik tablo çizilir. Şekil 4.2.23 iki döngünün birbirine nasıl yerleştiğini gösterir.



Şekil 4.2.23: İç içe döngü: i 1'den 2'ye, j 1'den 2'ye; iç gövde i × j yazar. İç döngü bitince i artar ve dış karara dönülür.

```
Başla
  i = 1
  i <= 2 ?  (Hayır → Bitir)
    j = 1
    j <= 2 ?  (Hayır → i = i + 1)
      i × j yaz
      j = j + 1, iç karara dön
    satır sonu (println), i = i + 1, dış karara dön
Bitir
```

Dış döngünün gövdesi üç parçadan oluşur: `j = 1`, iç döngünün tamamı ve `i = i + 1`. `j = 1` kutusunun dış gövdenin içinde olması önemlidir. Her dış turda j yeniden 1 olur; böylece iç döngü her satıra baştan başlar.

i	j	İç karar	İş
1	1	1 <= 2 → doğru	1 × 1 = 1
1	2	2 <= 2 → doğru	1 × 2 = 2
1	3	3 <= 2 → yanlış	satır sonu
2	1	1 <= 2 → doğru	2 × 1 = 2
2	2	2 <= 2 → doğru	2 × 2 = 4
2	3	3 <= 2 → yanlış	satır sonu

Tablo 4.2.15: 2 × 2 iç içe döngünün izleme tablosu. Her satır bir iç karar sorusudur.

Tablo 4.2.15 yürütmeyi verir. i = 1 iken dış karar *1 <= 2* doğrudur. j = 1 olur; iç karar doğrudur ve 1 × 1 = 1 yazılır. j = 2 olur; 1 × 2 = 2 yazılır. j = 3 olduğunda iç karar yanlıştır ve iç döngü biter. Satır sonu yazılır, i = 2 olur ve dış karara dönülür. İkinci dış turda j yeniden 1'den başlar ve aynı yol izlenir: 2 ve 4 yazılır. i = 3 olduğunda dış karar *3 <= 2* yanlıştır ve Bitir'e gidilir. Çıktı küçük bir çarpım tablosudur:

```
1 2
2 4
```

İç gövde *2 × 2 = 4* kez çalıştı. Genel olarak dış döngü n, iç döngü m tur dönüyorsa iç gövde n × m kez çalışır. Sayılar büyüdükçe bu çarpım hızla büyür: 1000 × 1000'lik bir tabloda iç gövde bir milyon kez çalışır.

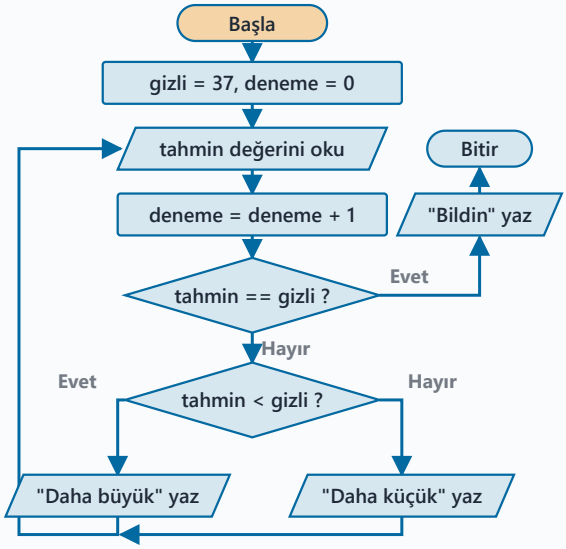
```
public class CarpimTablosu {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 2) {                  // dış karar
            int j = 1;                    // iç sayaç her dış turda yeniden 1
            while (j <= 2) {              // iç karar
                System.out.print(i * j + " ");
                j = j + 1;
            }
            System.out.println();         // satır sonu
            i = i + 1;
        }
    }
}
```

Program iki satır yazar: `1 2` ve `2 4`. Java kodunda `int j = 1;` satırı dış döngünün küme parantezinin içindedir. Bu satır dış döngüden önceye taşınırsa j ikinci dış turda 3'te kalır, iç karar ilk denemede yanlış çıkar ve ikinci satır boş yazılır.

Çok sayıda iç içe yapı, algoritmanın bir parçasının ayrı bir alt süreç olarak çizilmesi gerektiğini gösterebilir. Üç dört düzey iç içe geçmiş kararlar ve döngüler, diyagramı da kodu da okunmaz hâle getirir.

4.2.6.7 Sayı tahmin oyunu

Bilgisayar 1 ile 100 arasında gizli bir sayı tutar. Oyuncu tahmin eder; bilgisayar “Daha büyük” ya da “Daha küçük” diye ipucu verir. Oyun, doğru tahmin yapılınca biter ve kaç denemede bilindiği yazılır. Döngünün kaç tur döneceğini oyuncunun tahminleri belirler; bu açıdan oyun, bekçi değerli döngünün bir türüdür.



Şekil 4.2.24: Sayı tahmin oyunu: tahmin gizli sayıya eşit olana kadar ipucu verilerek tekrar okunur. Doğru tahminde "Bildin" yazılır ve oyun biter.

```
Başla
  gizli = 37, deneme = 0
  tahmin değerini oku
  deneme = deneme + 1
  tahmin == gizli ?
    Evet → "Bildin (" deneme ". deneme)" yaz, Bitir
  tahmin < gizli ?
    Evet → "Daha büyük" yaz
    Hayır → "Daha küçük" yaz
  3. satıra (oku) dön
Bitir
```

Diyagramda iki karar vardır. Birincisi `tahmin == gizli ?` sorusudur ve döngüyü sona erdirir. İkincisi `tahmin < gizli ?` sorusudur ve hangi ipucunun verileceğine karar verir. İkinci karar birincinin Hayır dalındadır; tahmin doğruysa ipucu sorusu hiç sorulmaz. İki ipucu kutusunun ardından ok yeniden oku kutusuna döner. Gizli sayı normalde rastgele seçilir; izlemenin tekrarlanabilmesi için burada 37 alınmıştır.

tahmin	deneme	tahmin == 37 ?	tahmin < 37 ?	Çıktı
50	1	Hayır	Hayır	Daha küçük
25	2	Hayır	Evet	Daha büyük
37	3	Evet	sorulmaz	Bildin (3. deneme)

Tablo 4.2.16: Gizli sayı 37, tahminler 50, 25, 37 için izleme tablosu.

Oyuncu önce 50 girer; deneme = 1 olur. *50 == 37* yanlıştır, ipucu sorusuna geçilir. *50 < 37* de yanlıştır; tahmin büyük olduğu için “Daha küçük” yazılır. İkinci tahmin 25'tir; *25 < 37* doğrudur ve “Daha büyük” yazılır. Üçüncü tahmin 37'dir; eşitlik doğru çıkar ve “Bildin (3. deneme)” yazılır.

```
import java.util.Scanner;

public class SayiTahmin {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int gizli = 37;                  // oyunda: new java.util.Random().nextInt(100) +
1
        int deneme = 0;
        int tahmin = giris.nextInt();
        deneme = deneme + 1;
        while (tahmin != gizli) {        // tahmin == gizli ? Hayır dalı döngüde kalır
            if (tahmin < gizli) {
                System.out.println("Daha büyük");
            } else {
                System.out.println("Daha küçük");
            }
            tahmin = giris.nextInt();
            deneme = deneme + 1;
        }
        System.out.println("Bildin (" + deneme + ". deneme)");
    }
}
```

Java programı diyagramı biraz farklı bir düzenle yazar. İlk tahmin döngüden önce okunur ve döngünün koşulu `tahmin != gizli` olur. Sonraki tahmin gövdenin sonunda okunur. Bu, bekçi değerli döngüdeki düzenin aynısıdır. Program 50 25 37 girdisiyle “Daha küçük”, “Daha büyük” ve “Bildin (3. deneme)”; 20 40 37 girdisiyle “Daha büyük”, “Daha küçük” ve “Bildin (3. deneme)” yazar. İlk tahminde 37 girilirse döngü gövdesi hiç çalışmaz ve “Bildin (1. deneme)” yazılır.

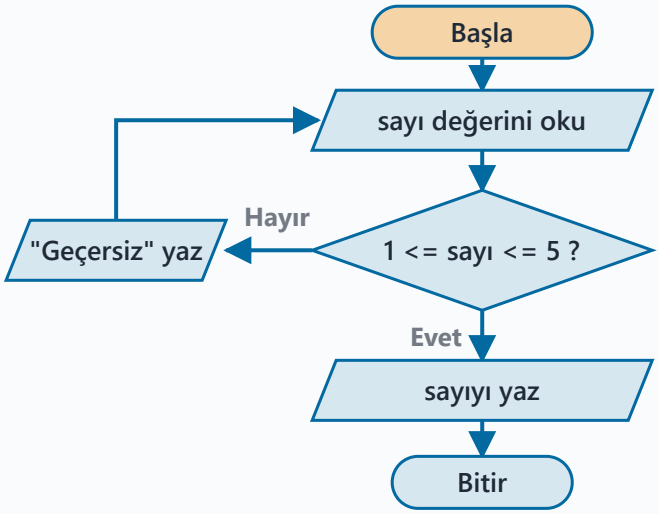
Oyunun iyi bir stratejisi vardır: her seferinde kalan aralığın ortasını tahmin etmek. 1–100 aralığında ilk tahmin 50 olur; ipucu “Daha küçük” ise aralık 1–49'a, “Daha büyük” ise 51–100'e iner. Her tahmin aralığı yaklaşık yarıya böler. Aralık en kötü durumda 100, 50, 25, 12, 6, 3, 1 diye küçülür; bu yüzden 7 tahmin her zaman yeter. Bunun nedeni $2^7 = 128$'in 100'den büyük olmasıdır.

4.2.6.8 Son koşullu döngü

Bazı işlerde döngü gövdesinin en az bir kez çalışması gerekir. Kullanıcıdan 1 ile 5 arasında bir sayı istenir; girilen sayı geçersizse yeniden sorulur. Sayı okunmadan geçerli olup olmadığı sorulamaz. Bu durumda koşulu gövdenin sonuna koymak daha doğaldır.

Tanım 4.2.17

Son koşullu döngü (*post-test loop*), koşulun gövdeden sonra sorulduğu döngüdür. Gövde önce çalışır, sonra koşula bakılır; bu yüzden gövde *en az bir kez* çalışır. Java'da `do { … } while (koşul);` biçiminde yazılır.



Şekil 4.2.25: Son koşullu döngü ile girdi doğrulama: sayı okunur; 1 ≤ sayı ≤ 5 değilse "Geçersiz" yazılır ve yeniden okunur.

```
Başla
  sayı değerini oku
  1 <= sayı <= 5 ?
    Hayır → "Geçersiz" yaz, oku'ya dön
    Evet → sayıyı yaz
Bitir
```

Şekil 4.2.25 diyagramında yürütme Başla'dan sonra doğrudan oku kutusuna girer; arada bir karar yoktur. Karar, okumanın ardından gelir. Geçersiz bir sayıda Hayır dalı "Geçersiz" yazar ve ok yeniden oku kutusuna döner. Bu tür kullanıma **girdi doğrulama** (*input validation*) denir.

Tur	Okunan sayı	1 ≤ sayı ≤ 5 ?	Çıktı
1	7	yanlış → Hayır	Geçersiz
2	0	yanlış → Hayır	Geçersiz
3	3	doğru → Evet	Geçerli: 3

Tablo 4.2.17: Girdiler 7, 0, 3 için son koşullu döngünün izleme tablosu.

İlk turda koşul henüz hiç sorulmadan 7 okunur. Sonra karar gelir: 1 ≤ 7 ≤ 5 mi? 7, 5'ten büyük olduğu için cevap yanlıştır; "Geçersiz" yazılır. İkinci turda 0 okunur; 0, 1'den küçük olduğu için yine geçersizdir. Üçüncü turda 3 okunur; 1 ≤ 3 ≤ 5 doğrudur ve "Geçerli: 3" yazılır. Gövde 3 kez çalıştı, koşul da 3 kez soruldu. Ön koşullu döngüden farklı olarak burada karar sayısı gövde sayısına eşittir.

Diyagramdaki `1 <= sayı <= 5` koşulu matematikteki gösterimdir. Java bu yazımı kabul etmez; iki karşılaştırma bir bileşik koşulla birleştirilir. Geçersizlik koşulu "sayı 1'den küçük VEYA 5'ten büyük" olur:

```
import java.util.Scanner;

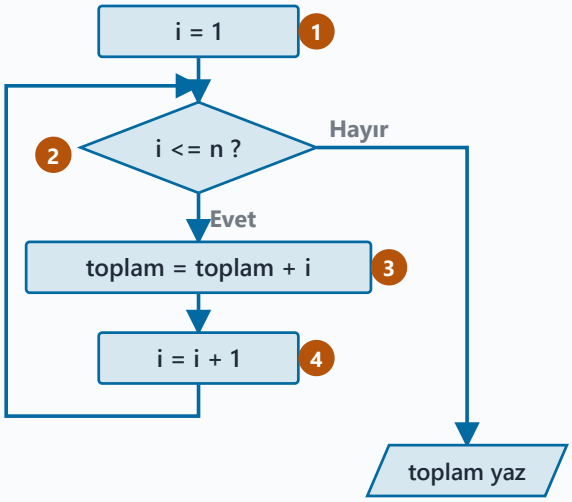
public class GirdiDogrulama {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int sayi;
        do {
            sayi = giris.nextInt();              // gövde: önce oku
            if (sayi < 1 || sayi > 5) {
                System.out.println("Geçersiz");
            }
        } while (sayi < 1 || sayi > 5);          // koşul sonda sorulur
        System.out.println("Geçerli: " + sayi);
    }
}
```

Program 7 0 3 girdisiyle iki kez "Geçersiz", sonra "Geçerli: 3" yazar. İlk girdi 4 ise gövde bir kez çalışır ve doğrudan "Geçerli: 4" yazılır. Kodda geçersizlik koşulu iki kez yazılmıştır: bir kez uyarı için `if` içinde, bir kez döngü koşulunda. Diyagramdaki tek karar kutusu kodda iki ayrı soruya dönüşür.

> **Sık yapılan hata**
>
> `while` ile `do-while` arasındaki farkı yalnız yazım farkı sanmak. Fark, koşul ilk denemede yanlış olduğunda ortaya çıkar. 1'den n'ye toplam diyagramı son koşullu çizilseydi, n = 0 için gövde bir kez çalışır, toplam 1 olurdu; oysa doğru cevap 0'dır. Gövdenin hiç çalışmaması gerekebilecek her durumda ön koşullu döngü seçilir.

4.2.6.9 for döngüsünün diyagramı

Sayaçlı döngülerde her zaman aynı dört parça bulunur: sayacın başlangıç değeri, koşul, gövde ve sayacın güncellenmesi. Java bu kalıp için daha kısa bir yazım sunar: `for` döngüsü.



Şekil 4.2.26: for döngüsünün dört parçası: (1) başlangıç i = 1, (2) koşul i <= n, (3) gövde toplam = toplam + i, (4) güncelleme i = i + 1.

```
for (i = 1; i <= n; i++) {
    toplam = toplam + i;
}
```

1. **Başlangıç** (`i = 1`): döngüye girerken yalnızca bir kez çalışır.
2. **Koşul** (`i <= n`): her turdan önce sorulur; for, ön koşullu bir döngüdür.
3. **Gövde**: koşul doğruysa çalışır.
4. **Güncelleme** (`i++`): gövdeden sonra çalışır; ardından ok koşula döner. `i++` , `i = i + 1` 'in kısa yazılışıdır.

Şekil 4.2.26 bir while diyagramından farksızdır. for, yalnızca 1, 2 ve 4 numaralı parçaları tek satırda toplar. Bazı kaynaklar bu üç parçayı ek simgelerden hazırlık simgesinde (altıgen, “i = 1 → n”) gösterir.

```
public class ForToplam {
    public static void main(String[] args) {
        int n = 4, toplam = 0;
        for (int i = 1; i <= n; i++) {    // başlangıç; koşul; güncelleme
            toplam = toplam + i;          // gövde
        }
        System.out.println(toplam);
    }
}
```

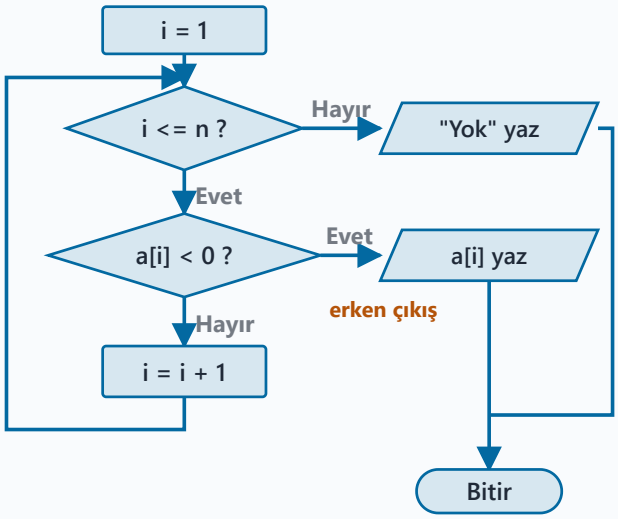
Program n = 4 için 10 yazar; while ile yazılan 1'den n'ye toplam programıyla aynı sonucu verir. Hangi döngünün seçileceği okunabilirlik sorunudur. Tekrar sayısı önceden belliyse `for`, döngüyü veri durduruyorsa `while` daha açık okunur.

4.2.6.10 Erken çıkış, bayrak ve atla

Bir dizide ilk negatif sayı aranıyor olsun: a = [4, 7, −2, 5, −8]. Aranan sayı bulunduğunda kalan turlara gerek kalmaz. Gövdenin içinden döngünün dışına giden ikinci bir ok, döngüyü hemen bitirir.

Tanım 4.2.18

Erken çıkış (*early exit*), döngünün koşulu henüz yanlış olmadan gövdenin içinden döngüden çıkılmasıdır; Java'da `break` (çık) deyimiyle yapılır. **Bayrak** (*flag*), bir olayın olup olmadığını tutan doğru/yanlış değerli değişkendir. **Atla** (*continue*) ise döngüden çıkmaz; o turun kalanını atlayıp güncellemeye ve koşula döner.



Şekil 4.2.27: İlk negatif sayıyı arayan döngü. Döngünün iki çıkışı vardır: biri koşuldan (i <= n yanlış: "Yok" yaz), biri gövdeden (a[i] < 0: a[i] yaz, erken çıkış).

Diyagramda dizinin elemanları `a[i]` ile gösterilir: a[1] dizinin ilk, a[2] ikinci elemanıdır. Dizi, aynı türden değerlerin sıralı bir listesidir; burada beş sayı vardır, yani n = 5.

i	a[i]	i <= n ?	a[i] < 0 ?	Yapılan
1	4	1 <= 5 → Evet	Hayır	i = 2
2	7	2 <= 5 → Evet	Hayır	i = 3
3	−2	3 <= 5 → Evet	Evet	−2 yaz, döngüden çık

Tablo 4.2.18: a = [4, 7, −2, 5, −8] için ilk negatif arama.

i = 1'de 4, i = 2'de 7 negatif değildir. i = 3'te −2 bulunur ve yazılır; döngü erken çıkışla biter. 5 ve −8'e hiç bakılmaz. Dizide hiç negatif olmasaydı döngü i = 6'da koşuldan çıkar ve "Yok" yazılırdı.

Erken çıkışlı döngünün iki çıkışı vardır ve bu, yapısal programlamanın tek çıkış kuralına uymaz. Tek çıkış isteniyorsa bayrak kullanılır. `bulundu = yanlış` ile başlanır, aranan bulununca `bulundu = doğru` yapılır ve döngü koşulu `i <= n VE bulundu değil ?` olur. Döngü bitince bayrağa bakılarak sonuç yazılır. Java'da iki yol da kullanılır:

```
public class IlkNegatif {
    public static void main(String[] args) {
        int[] a = {4, 7, -2, 5, -8};
        // 1. yol: break ile erken çıkış (iki çıkışlı döngü)
        int i = 0;
        while (i < a.length) {
            if (a[i] < 0) {
                break;                    // aranan bulundu: döngüden çık
            }
            i = i + 1;
        }
        if (i < a.length) {
            System.out.println(a[i]);
        } else {
            System.out.println("Yok");
        }
        // 2. yol: bayrak ile tek çıkışlı döngü
        boolean bulundu = false;
        int k = 0;
        while (k < a.length && !bulundu) {
            if (a[k] < 0) {
                bulundu = true;
            } else {
                k = k + 1;
            }
        }
        System.out.println(bulundu ? a[k] + "" : "Yok");
    }
}
```

İki yol da −2 yazar. Java'da dizi elemanları 0'dan numaralanır: `a[0]` 4, `a[2]` −2'dir. Bu yüzden kodda sayaç 0'dan başlar ve koşul `i < a.length` olur; `a.length` dizinin eleman sayısıdır (5). Bayraklı sürümde `!bulundu` ifadesi "bulundu değil" anlamına gelir. Koşul `k < a.length && !bulundu` bir bileşik koşuldur ve kısa devre değerlendirmeyle çalışır.

Atla ise döngüyü bitirmez, yalnız o turun kalanını geçer. Örneğin dizideki negatif olmayan sayıların toplamı istenirse, negatif bir sayıya gelindiğinde toplama adımı atlanır:

```
public class AtlaOrnegi {
    public static void main(String[] args) {
        int[] a = {4, 7, -2, 5, -8};
        int toplam = 0;
        for (int i = 0; i < a.length; i++) {
            if (a[i] < 0) {
                continue;                // negatifi atla, sonraki tura geç
            }
            toplam = toplam + a[i];
        }
        System.out.println(toplam);
    }
}
```

Program *4 + 7 + 5 = 16* yazar. −2 ve −8'de `continue` çalışır; toplama satırı atlanır, `i++` çalışır ve koşula dönülür. Diyagramda atla, gövdenin ortasından güncelleme kutusuna giden bir okla gösterilir.

4.2.7 Diyagramdan Java koduna

Akış diyagramının değeri, sonunda çalışan bir programa dönüşebilmesinden gelir. Önceki kesimlerde her yapının Java karşılığı ayrı ayrı verildi. Bu kesim çeviriyi bir kural listesi olarak toplar ve tam bir örnek üzerinde satır satır uygular.

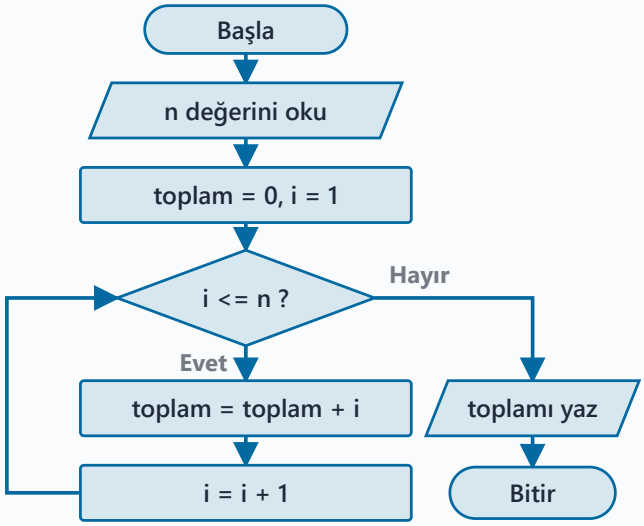
4.2.7.1 Simgelerin kod karşılıkları

Diyagram öğesi	Java karşılığı
Başla / Bitir	`main` metodunun açılan ve kapanan küme parantezi
Giriş (oku)	`int a = giris.nextInt();` ya da sabit bir değerle değişken tanımı
İşlem (atama)	Atama deyimi: `toplam = toplam + i;`
Çıkış (yaz)	`System.out.println(…);`
Karar, iki dal birleşiyor	`if (koşul) { … } else { … }`
Karar zinciri	`if … else if … else`
Karar + gövdeye dönen ok (koşul önce)	`while (koşul) { … }` ya da `for`
Gövde + karara dönen ok (koşul sonda)	`do { … } while (koşul);`
Gövdeden döngü dışına ok	`break;`
Önceden tanımlı işlem	Metot çağrısı: `ortalama(a, b, c)`

Tablo 4.2.19: Akış diyagramı öğelerinin Java karşılıkları.

Kuralların en önemlisi döngü çevirisidir. Bir karar kutusunun Evet dalından başlayıp yeniden aynı karara dönen bir ok varsa, o karar bir `while` koşuludur ve oka kadar olan kutular döngü gövdesidir. Gövdenin hangi kutularda bittiğini bulmak için geri dönen okun nereden çıktığına bakılır.

4.2.7.2 1'den n'ye toplam Java'da



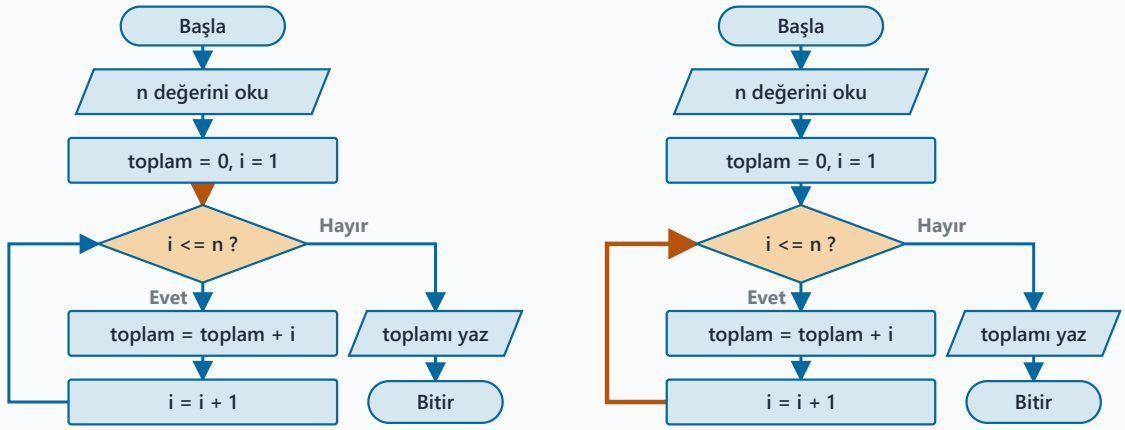
Şekil 4.2.28: Java'ya çevrilecek diyagram: 1'den n'ye toplam.

```
public static void main(String[] args) {
    int n = 3;
    int toplam = 0, i = 1;
    while (i <= n) {
        toplam = toplam + i;
        i = i + 1;
    }
    System.out.println(toplam);
}
```

Çeviri yukarıdan aşağıya kutu kutu yapılır:

1. Başla simgesi metodun başlangıcıdır: `main` açılır.
2. "n değerini oku" girdi simgesidir. Örneği kısa tutmak için girdi sabit alınmıştır: `int n = 3;`. Klavyeden okumak istenirse bu satır `int n = giris.nextInt();` olur.
3. "toplam = 0, i = 1" işlem simgesi iki değişkenin başlangıç değeridir: `int toplam = 0, i = 1;`.
4. `i <= n ?` kararından Evet dalı iki kutuya gider ve ok yeniden karara döner. Bu bir döngüdür: `while (i <= n)`.
5. Evet dalındaki iki dikdörtgen döngü gövdesidir. İkisi de küme parantezinin içine, diyagramdaki sırayla yazılır.
6. Geri dönen ok, gövde bloğunun kapanan küme parantezidir. Yürütme oradan koşula döner.
7. Hayır dalındaki "toplamı yaz" çıkış simgesi döngünün dışındadır: `System.out.println(toplam);`.
8. Bitir simgesi metodun kapanışıdır.

Program n = 3 için çalıştırıldığında yürütme diyagramla adım adım aynı yolu izler. İlk koşul denemesi *1 <= 3* doğrudur; toplam 0 + 1 = 1, i = 2 olur. Ardından toplam 3 ve 6 olur. i = 4 olduğunda *4 <= 3* yanlıştır; Hayır yolu izlenir ve yürütme döngü bloğunun dışına çıkar. Ekrana 6 yazılır. Sonuç formülle de doğrulanır: *3 × 4 / 2 = 6*.



Şekil 4.2.29: Karar simgesinin kod karşılığı. (a) İlk deneme: while (i <= n), 1 <= 3 doğru; gövdeye girilir. (b) Son deneme: 4 <= 3 yanlış; Hayır yolu döngü bloğunun dışına çıkar.

Çevirinin sonunda her simgenin bir kod karşılığı vardır, her kod satırının da bir simge karşılığı. Bu birebir eşleme, diyagramın doğru çevrildiğini denetlemenin en kolay yoludur. Programda karşılığı olmayan bir simge ya da diyagramda karşılığı olmayan bir satır varsa çeviride bir şey eksik ya da fazladır.

Sık yapılan hata

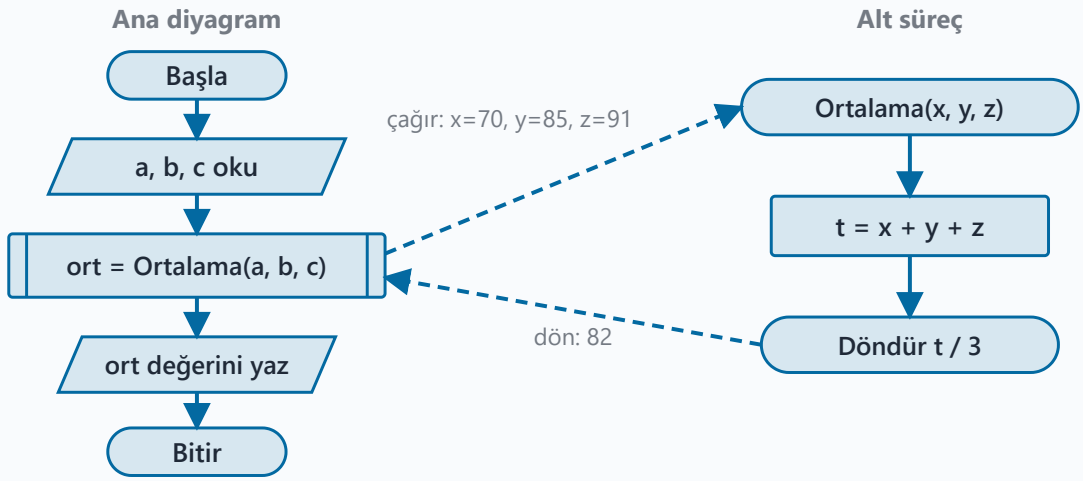
Gövde kutularından birini döngü bloğunun dışına yazmak. `while (i <= n) { toplam = toplam + i; } i = i + 1;` yazılırsa sayacın artışı döngünün dışında kalır. i hiç değişmez, koşul hep doğru çıkar ve program sonsuz döngüye girer. Diyagramda Evet dalındaki her kutu, geri dönen ok çıkmadan önceki son kutuya kadar, küme parantezlerinin içindedir.

4.2.7.3 Alt süreçle modüler diyagram

Diyagramlar büyüdükçe tek sayfaya sığmaz hâle gelir. Bir işin birkaç yerde tekrarlandığı da olur. Bu durumda işin bir kısmı ayrı bir diyagram olarak çizilir ve ana diyagramdan çağrılır.

Tanım 4.2.19

Alt süreç (*subprocess*), ayrı çizilmiş ve bir adla anılan diyagramdır. Ana diyagramda **önceden tanımlı işlem** (*predefined process*) simgesiyle çağrılır. Alt süreç bitince akış ana diyagramda, çağrıldığı noktadan sürer. Java'da her alt süreç bir **metot** (*method*) olur.



Şekil 4.2.30: Ana diyagram alt süreci çağırır, alt süreç değer döndürür. a = 70, b = 85, c = 91 için Ortalama(70, 85, 91) çağrılır ve 82 döner.

Şekil 4.2.30'daki alt diyagram Başla yerine adı ve parametreleriyle, “Ortalama(x, y, z)” ile başlar. Bitir yerine “Döndür” ile biter. Parametreler, alt sürecin çağrıldığı yerden aldığı değerlerdir. Ana diyagram `ort = Ortalama(a, b, c)` kutusunda a, b ve c'nin değerlerini x, y ve z'ye gönderir. a = 70, b = 85, c = 91 için alt süreçte *t = 70 + 85 + 91 = 246* hesaplanır ve *246 / 3 = 82* döndürülür. Ana diyagram bu değeri ort değişkenine yazar ve bir sonraki kutudan devam eder.

Alt sürecin iki yararı vardır. Birincisi, aynı alt süreç birçok yerden çağrılabilir ama yalnızca bir kez çizilir. Ortalama hesabında bir hata bulunursa tek bir yerde düzeltilir. İkincisi, ana diyagram kısalır ve “ne yapıldığını” anlatır; “nasıl yapıldığı” alt diyagramda kalır.

```
import java.util.Scanner;

public class AltSurec {
    // alt süreç: Ortalama(x, y, z)
    static int ortalama(int x, int y, int z) {
        int t = x + y + z;
        return t / 3;                    // Döndür t / 3
    }

    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int a = giris.nextInt(), b = giris.nextInt(), c = giris.nextInt();
        int ort = ortalama(a, b, c);      // önceden tanımlı işlem simgesi
        System.out.println(ort);
    }
}
```

`static int ortalama(int x, int y, int z)` satırı alt diyagramın başlığıdır: metodun adı, üç parametresi ve döndürdüğü değerin türü (`int`). `return t / 3;` satırı Döndür simgesinin karşılığıdır. Program 70 85 91 girdisiyle 82 yazar.

Sık yapılan hata

Tam sayı bölmesini unutmak. Java'da iki tam sayının bölümü de tam sayıdır; küsurat atılır. 70, 85, 91 için toplam 246 tam olarak 3'e bölündüğü için sorun görünmez. 50, 60, 71 için toplam 181'dir; gerçek ortalama yaklaşık 60,33 iken program 60 yazar. Ondalıklı sonuç gerekiyorsa metot `double` döndürmeli ve bölme `t / 3.0` biçiminde yazılmalıdır.

4.2.8 Doğrulama

Bir diyagramın doğru olduğu, birkaç girdiyle elle yürütülerek sınanır. Hangi girdilerin seçileceği rastgele değildir. İyi seçilmiş az sayıda girdi, kötü seçilmiş çok sayıda girdiden daha fazla hata yakalar.

4.2.8.1 Diyagramı test etmek

Tanım 4.2.20

Test durumu (*test case*), bir girdi ile o girdi için beklenen çıktının ikilisidir. Diyagram girdiyle yürütülür ve bulunan çıktı beklenenle karşılaştırılır. **Sınır değeri** (*boundary value*), bir koşulun cevabının değiştiği noktadaki girdidir; örneğin `not >= 60` için 60.

Test girdileri beş kurala göre seçilir:

1. Normal bir girdi seçilir ve oklar adım adım izlenir.
2. Her kararın her çıkışını çalıştıracak girdiler seçilir. Diyagramdaki her kutu en az bir kez çalışmalıdır.
3. Sınır değerleri ayrıca denenir.
4. Her döngü için gövdenin sıfır, bir ve çok kez çalıştığı durumlar denenir.
5. Başlangıca dönmeyen ya da Bitir'e ulaşmayan yollar aranır.



Şekil 4.2.31: Geçti/Kaldı diyagramının testi. (a) Normal girdi not = 72: Evet yolu. (b) not = 45: Hayır yolu; artık her kutu en az bir kez denendi. (c) Sınır değeri not = 60: 60 >= 60 doğru, "Geçti".

Şekil 4.2.31 kuralların Geçti/Kaldı diyagramına uygulanışını gösterir. 72 normal girdidir ve Evet yolunu çalıştırır. 45 öbür dalı çalıştırır; iki girdiyle her kutu denenmiş olur. 60 ise sınır değeridir ve karar kutusunda `>=` yerine `>` yazılmış olsaydı hatayı ortaya çıkaran tek girdi olurdu.

Döngüler için kural 4'ün nedeni şudur: döngü hataları çoğu zaman uçlarda saklanır. Biriktiricinin başlangıç değeri yanlışsa bu en çok gövdenin hiç çalışmadığı durumda görünür. Koşuldaki bir `<` / `<=` karışıklığı ise genellikle bir tur eksik ya da fazla dönmeye yol açar ve bir turluk durumda kolay fark edilir. Tablo 4.2.20 1'den n'ye toplam diyagramı için bu üç durumu verir.

n	Durum	Gövde kaç kez çalışır	Beklenen (n(n+1)/2)	Diyagramın çıktısı
0	sıfır tekrar	0	0	0
1	bir tekrar	1	1	1
5	çok tekrar	5	15	15

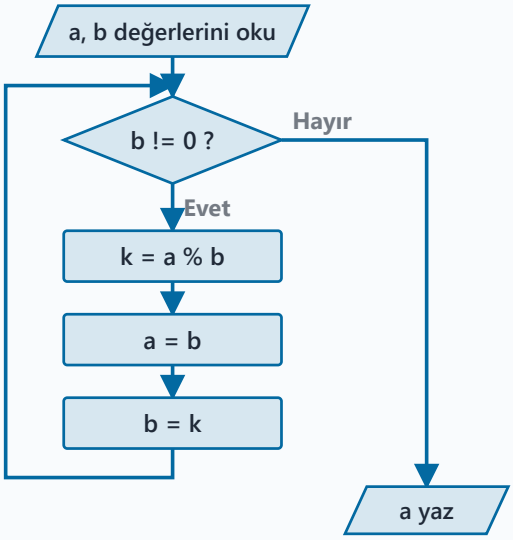
Tablo 4.2.20: 1'den n'ye toplam diyagramının döngü testleri. Beklenen değer formülden, çıktı diyagramın yürütülmesinden gelir.

Üç satırda da çıktı beklenenle aynıdır. Koşul yanlışlıkla `i < n` yazılmış olsaydı n = 1 için gövde hiç çalışmaz ve 0 yazılırdı; n = 5 için 1 + 2 + 3 + 4 = 10 yazılırdı. Hata üç girdinin ikisinde hemen görünür. Negatif bir n ise geçersiz girdidir. Diyagram bu durumda da 0 yazar; bunun istenen davranış olup olmadığı problemin tanımına bağlıdır.

4.2.8.2 İzleme tablosuyla elle izleme

İzleme tablosu bu notta sık sık kullanıldı. Kalıp kısaca şöyle özetlenebilir: her değişken bir sütun, her tur bir satırdır; karar sonucu da ayrı bir sütuna yazılır. Tablo özellikle değişkenlerin birbirine bağlı biçimde değiştiği algoritmalarda vazgeçilmezdir. Öklid algoritması bunun güzel bir örneğidir.

Öklid algoritması iki sayının **en büyük ortak bölenini** (EBOB, *greatest common divisor*) bulur. Fikri şudur: a ile b'nin ortak bölenleri, b ile a'nın b'ye bölümünden kalanın ortak bölenleriyle aynıdır. Bu yüzden (a, b) çifti (b, a % b) çiftiyle değiştirilir. Kalan her turda küçülür ve sonunda 0 olur. O anda a, aranan EBOB'dur.



Şekil 4.2.32: Öklid algoritması: b sıfır olana kadar k = a % b, a = b, b = k yapılır; sonunda a yazılır.

An	a	b	k	b != 0 ?
başlangıç	48	18	—	18 != 0 → doğru
1. tur	18	12	12	12 != 0 → doğru
2. tur	12	6	6	6 != 0 → doğru
3. tur	6	0	0	0 != 0 → yanlış

Tablo 4.2.21: Öklid algoritmasının (48, 18) için izleme tablosu. Son sütun, satırdaki değerlerle sorulan bir sonraki karardır.

Başlangıçta a = 48, b = 18'dir ve *18 != 0* doğrudur. Birinci turda *k = 48 % 18 = 12* olur (48 = 2 × 18 + 12); a = 18, b = 12 yapılır. İkinci turda *k = 18 % 12 = 6*; a = 12, b = 6. Üçüncü turda *k = 12 % 6 = 0*; a = 6, b = 0. Karar *0 != 0* yanlıştır; döngüden çıkılır ve 6 yazılır. EBOB(48, 18) = 6'dır. Denetim: *48 = 8 × 6*, *18 = 3 × 6* ve 8 ile 3'ün 1'den büyük ortak böleni yoktur.

Döngü 3 tur döndü, karar 4 kez soruldu: her ön koşullu döngüde olduğu gibi karar, gövdeden bir fazla çalıştı. Tablo diyagramdaki sıranın önemini de gösterir. Gövdedeki `a = b` ve `b = k` kutuları yer değiştirseydi, yani önce `b = k`, sonra `a = b` yapılsaydı ne olurdu?

An	a	b	k	b != 0 ?
başlangıç	48	18	—	18 != 0 → doğru
1. tur	12	12	12	12 != 0 → doğru
2. tur	0	0	0	0 != 0 → yanlış

Tablo 4.2.22: Gövdede b = k ile a = b yer değiştirince (48, 18) için izleme.

İlk turda b = 12 olur, ardından a = b ile a da 12 olur. a, b'nin eski değerini değil yeni değerini almıştır. İkinci turda *12 % 12 = 0* bulunur ve ikisi de 0 olur. Sonuç 6 yerine 0 çıkar. Bu, Örnek 4.2.2'deki takas hatasının aynısıdır: bir atama, daha sonra gerekecek bir değeri silmiştir.

```
import java.util.Scanner;

public class Oklid {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int a = giris.nextInt(), b = giris.nextInt();
        while (b != 0) {
            int k = a % b;
            a = b;
            b = k;
        }
        System.out.println(a);
    }
}
```

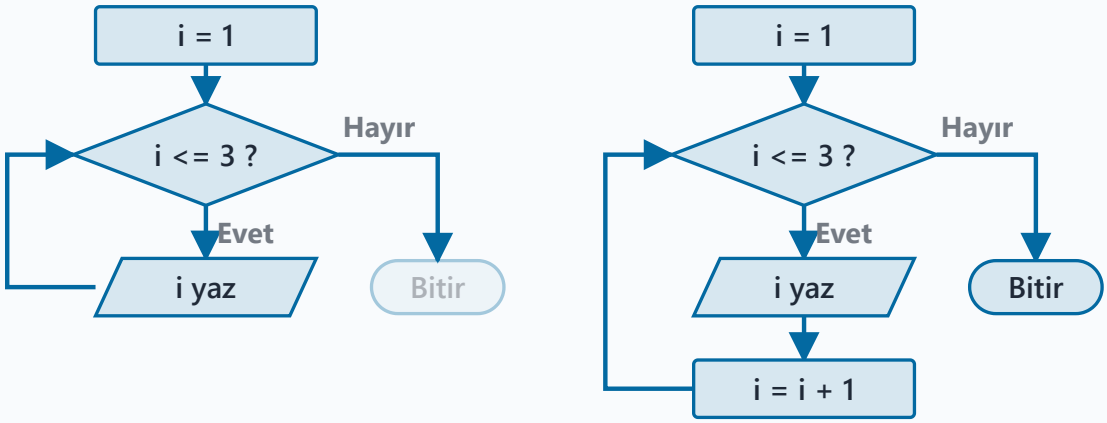
Program (48, 18) için 6, (270, 192) için 6, (36, 84) için 12 yazar. a < b olsa da algoritma çalışır. (18, 48) için ilk turda *18 % 48 = 18* bulunur ve a ile b yer değiştirir; sonuç yine 6'dır.

4.2.8.3 Sık hatalar

Öğrencilerin akış diyagramlarında en sık yaptığı hatalar birkaç kalıba indirgenebilir. Tablo 4.2.23 çizimle ilgili, Tablo 4.2.24 mantıkla ilgili hataları toplar.

Hata	Sonuç	Düzeltme
Etiketsiz karar yolu	Hangi koşulun seçildiği belirsiz	Evet/Hayır yaz
Güncellenmeyen sayaç	Sonsuz döngü	Sayaç adımı ekle
Bağlantısız simge	Akış kopar	Okla bağla
Bir simgede çok işlem	Okunabilirlik azalır	Adımlara böl

Tablo 4.2.23: Çizim hataları.



Şekil 4.2.33: Güncellenmeyen sayaç. (a) Hatalı: i hep 1 kalır, ekrana 1 1 1 1 … yazılır ve Bitir'e hiç ulaşılmaz. (b) Doğru: i = i + 1 eklenince 1 2 3 yazılır, sonra Bitir'e gidilir.

Şekil 4.2.33 en tehlikeli çizim hatasını gösterir. Soldaki diyagramda geri dönen okun yolunda i'yi değiştiren bir kutu yoktur. Koşul her seferinde aynı cevabı verir ve döngü bitmez. Bir döngü çizildiğinde ilk sorulacak soru şudur: tekrar yolundaki hangi kutu koşulu bir gün yanlış yapacak?

Hata	Belirti	Düzeltme
Biriktirici döngünün içinde sıfırlanıyor	4, 7, 2 için 13 yerine 2 yazılır: yalnız son değer kalır	`toplam = 0` kutusunu döngüden önceye al
Geri dönen ok oku kutusunu atlıyor	Aynı x tekrar tekrar işlenir; bekçi değer hiç gelmez	Ok, oku kutusunun önüne döner
Karar kutusunda atama: `x = 0 ?`	Soru mu, atama mı belirsiz	Karşılaştırma yaz: `x == 0 ?`
Bekçi değer işlemden sonra sorgulanıyor	−1 toplama katılır, sayaç bir fazla olur	Kararı toplama ve saymadan önce sor
Karar zincirinde geniş koşul önce	Alttaki dallar hiç çalışmaz	Dar koşuldan genişe sırala
Karardan tek ok çıkıyor	Diğer durumda akış kopar	İki çıkışı da çiz, Evet/Hayır yaz
Bölmeden önce sıfır denetimi yok	Hiç veri girilmezse 0'a bölme	`sayac > 0 ?` kararı ekle

Tablo 4.2.24: Mantık hataları, belirtileri ve düzeltmeleri.

Tablodaki hataların çoğu bu notun önceki kesimlerinde izleme tablolarıyla gösterildi. Ortak noktaları şudur: hepsi diyagram en az bir uygun girdiyle elle yürütüldüğünde hemen fark edilir. Biriktirici hatası 4, 7, 2 girdisiyle, bekçi hatası 80, 64, 90, −1 girdisiyle ortaya çıkar. Zincir sırası hatasını −5, sıfır denetimi eksikliğini yalnız −1'den oluşan girdi gösterir. Bir diyagram çizildikten sonra onu birkaç iyi seçilmiş girdiyle yürütmek, bu hataların hepsine karşı en etkili önlemdir.

4.2.9 Özet

- **Akış diyagramı**, bir algoritmanın adımlarını standart simgelerle, yürütme sırasını oklarla gösterir. Programlama dilinden bağımsızdır; mantığı koddan önce tartışmayı ve hataları erken görmeyi sağlar.
- Beş temel simge vardır: **başla/bitir** (oval), **işlem** (dikdörtgen: hesaplama, atama), **karar** (eşkenar dörtgen: Evet/Hayır sorusu), **giriş/çıkış** (paralelkenar: oku, yaz) ve **ok**. Yollar **bağlayıcıda** birleşir. Ek simgeler (sayfa dışı bağlayıcı, önceden tanımlı işlem, hazırlık, belge, veri deposu, açıklama) büyük diyagramlarda kullanılır.
- Karar simgesine soru, işlem simgesine emir yazılır. Atama (`=`) ile eşitlik sorusu (`==`) farklıdır.
- İyi bir diyagram yukarıdan aşağıya akar, karar çıkışlarını etiketler ve her kutuda tek iş yapar. Aynı algoritma sözde kodla da anlatılabilir; iki gösterim birbirine satır satır çevrilir.
- **Böhm–Jacopini teoremine** göre her algoritma yalnızca **sıra**, **seçim** ve **tekrar** yapılarıyla yazılabilir. Her yapının tek girişi ve tek çıkışı vardır; bu yüzden yapılar iç içe kullanılabilir.
- **Sıralı akışta** her kutu tam bir kez ve sırayla çalışır. Kutuların sırası sonucu değiştirir; takas için geçici değişken gerekir.
- **Koşullu akışta** her çalıştırmada dallardan yalnızca biri izlenir. Çok seçenekte kararlar **karar zinciri** olarak, dar koşuldan geniş koşula dizilir; n seçenek için n − 1 karar gerekir. **Bileşik koşulda** VE ve VEYA, Java'da `&&` ve `||` , **kısa devre değerlendirmeyle** hesaplanır.
- **Tekrarlı akışta** geri dönen ok döngüyü oluşturur. Ön koşullu döngüde karar, gövdeden bir fazla çalışır; tekrar yolunda koşulu değiştiren bir işlem yoksa **sonsuz döngü** oluşur.
- **Sayaç** tur sayısını, **biriktirici** kısmi sonucu tutar; biriktirici toplam için 0, çarpım için 1 ile ve döngüden önce başlar. **Bekçi değer** gerçek veride görülemeyecek bir değerdir; kararı veriyi işleyen kutulardan önce sorulur.
- **İç içe döngüde** dış döngünün her turunda iç döngü baştan çalışır; iç gövde n × m kez çalışır. **Son koşullu döngüde** (`do-while`) gövde en az bir kez çalışır. `for` , başlangıç, koşul ve güncellemeyi tek satırda toplayan ön koşullu döngüdür.
- **Erken çıkış** (`break`) döngüye ikinci bir çıkış ekler; tek çıkış isteniyorsa **bayrak** kullanılır. **Atla** (`continue`) yalnız o turun kalanını geçer.
- Diyagram Java'ya simge simge çevrilir: oku → `nextInt()` , işlem → atama, karar → `if` , karar + geri dönen ok → `while` , yaz → `println` , **alt süreç** → metot.
- Diyagram, her kararın her çıkışını çalıştıran girdilerle, **sınır değerleriyle** ve döngünün sıfır, bir, çok kez döndüğü durumlarla test edilir. **İzleme tablosunda** her değişken bir sütun, her tur bir satırdır.

4.2.10 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

akış diyagramı *flowchart* · başla/bitir simgesi *terminal symbol* · işlem simgesi *process symbol* · karar simgesi *decision symbol* · giriş/çıkış simgesi *input/output symbol* · ok *flow line* · bağlayıcı *connector* · sayfa dışı bağlayıcı *off-page connector* · önceden tanımlı işlem *predefined process* · hazırlık *preparation* · atama *assignment* · değişken *variable* · koşul *condition* · sözde kod *pseudocode* · yapısal programlama *structured programming* · spagetti akış *spaghetti code* · sıralı akış *sequential flow* · geçici değişken *temporary variable* · koşullu akış *conditional flow* · dal *branch* · çok yollu karar *multi-way decision* · bileşik koşul *compound condition* · kısa devre değerlendirme *short-circuit evaluation* · kalan *remainder* · tekrarlı akış *repetitive flow* · döngü *loop* · döngü gövdesi *loop body* · tur *iteration* · ön koşullu döngü *pre-test loop* · son koşullu döngü *post-test loop* · sonsuz döngü *infinite loop* · sayaç *counter* · biriktirici *accumulator* · sayaçlı döngü *counter-controlled loop* · bekçi değer *sentinel value* · bekçi değerli döngü *sentinel-controlled loop* · iç içe döngü *nested loop* · girdi doğrulama *input validation* · erken çıkış *early exit* · bayrak *flag* · atla *continue* · alt süreç *subprocess* · metot *method* · test durumu *test case* · sınır değeri *boundary value* · izleme tablosu *trace table* · en büyük ortak bölen (EBOB) *greatest common divisor*

4.2.11 Kendini Dene

1. Aşağıdaki adımların her biri hangi simgeyle çizilir? (a) `ort = toplam / 3` (b) Sayı negatif mi? (c) Sonucu ekrana yaz. (d) Ayrı bir sayfada çizilmiş “Not ortalaması hesapla” sürecini kullan.

   Cevap

   (a) bir atamadır: işlem simgesi, dikdörtgen. (b) cevabı evet ya da hayır olan bir sorudur: karar simgesi, eşkenar dörtgen; içine `x < 0 ?` yazılır. (c) dışarıya veri verir: giriş/çıkış simgesi, paralelkenar. (d) ayrıca çizilmiş bir alt süreci çağırır: önceden tanımlı işlem simgesi, kenarlarında dikey çizgiler olan dikdörtgen.

2. Sıralı bir diyagram a = 5 okuyor, sonra sırasıyla `b = a * 2`, `a = b + a`, `c = a + b` kutularını çalıştırıp a, b ve c'yi yazıyor. Her kutudan sonra değişkenler ne olur, çıktı nedir?

   Cevap

   `b = a * 2` : *b = 5 × 2 = 10*; a = 5. `a = b + a` : *a = 10 + 5 = 15*; a'nın eski değeri 5 artık yoktur. `c = a + b` : *c = 15 + 10 = 25*. Çıktı `15 10 25` 'tir. c hesaplanırken a'nın yeni değeri (15) kullanılır.

3. Sıcaklık zincirindeki kararlar ters sırada dizilmiş: önce `s < 25 ?` (Evet → "Ilık"), sonra `s < 15 ?` (Evet → "Soğuk"), sonra `s < 0 ?` (Evet → "Donma", Hayır → "Sıcak"). s = −5, 10, 20 ve 30 için ne yazılır? Hangi çıkışlar hiç çalışamaz?

   Cevap

   −5, 10 ve 20 için ilk karar doğrudur ve üçü de "Ilık" yazar. 30 için ilk karar yanlıştır. Yürütme ikinci karara gelir; *30 < 15* ve *30 < 0* yanlıştır ve "Sıcak" yazılır. "Soğuk" ve "Donma" dallarına hiçbir girdiyle ulaşılamaz: ikinci karara yalnızca s ≥ 25 iken gelinir ve böyle bir s ne 15'ten ne 0'dan küçüktür. Doğru sıra dar koşuldan genişe, yani s < 0, s < 15, s < 25 sırasıdır.

4. i = 10 ile başlanan bir döngüde karar `i > 0 ?`'dır; Evet dalında i yazılır ve `i = i - 3` yapılır. Çıktı nedir? Gövde ve karar kaçar kez çalışır, döngü bittiğinde i kaçtır?

   Cevap

   i sırasıyla 10, 7, 4, 1 iken koşul doğrudur; bu değerler yazılır: `10 7 4 1`. 1 yazıldıktan sonra *i = 1 − 3 = −2* olur ve *−2 > 0* yanlıştır. Gövde 4 kez, karar 5 kez çalışır; döngüden i = −2 ile çıkılır.

5. 1'den 5'e kadar sayıların çarpımını bulan bir diyagramda biriktirici `carpim = 0` ile başlatılmış; gövde `carpim = carpim * i`. Diyagram ne yazar? Düzeltilmiş diyagram ne yazar?

   Cevap

   İlk turda *0 × 1 = 0* olur ve sonraki her tur 0 ile çarptığı için sonuç 0'dır. Çarpımda etkisiz değer 1'dir; `carpim = 1` ile başlanınca turlar 1, 2, 6, 24, 120 değerlerini verir ve 120 yazılır.

6. Bekçi değerli not ortalaması diyagramına sırasıyla 70, 85, 0, −1 giriliyor. Toplam, sayaç ve ortalama ne olur? Bekçi değer olarak −1 yerine 0 seçilseydi ne olurdu?

   Cevap

   70, 85 ve 0 gerçek nottur: toplam *70 + 85 + 0 = 155*, sayaç 3 olur. −1 okununca döngü biter. Ortalama *155 / 3 ≈ 51,67*'dir. Bekçi 0 olsaydı üçüncü girdi "veri bitti" diye yorumlanırdı: toplam 155, sayaç 2 ve ortalama 77,5 çıkardı. 0 gerçek bir not olabileceği için bekçi değer olarak uygun değildir.

7. İç içe bir döngüde dış sayaç i 1'den 3'e, iç sayaç j ise i'den 3'e kadar dönüyor. İç gövde bir "*" yazıyor; her dış turun sonunda satır sonu yazılıyor. Çıktı nedir, toplam kaç yıldız yazılır?

   Cevap

   i = 1 iken j = 1, 2, 3: üç yıldız. i = 2 iken j = 2, 3: iki yıldız. i = 3 iken j = 3: bir yıldız. Çıktı alt alta `***`, `**`, `*` satırlarıdır. Toplam *3 + 2 + 1 = 6* yıldız yazılır. İç döngünün başlangıcı dış sayaca bağlı olduğu için iç gövde 3 × 3 = 9 değil, 6 kez çalışır.

8. sayac = 10 ile başlayan bir döngünün gövdesi sayacı yazıp 1 artırıyor; koşul `sayac < 5`. Döngü ön koşullu (`while`) ve son koşullu (`do-while`) çizildiğinde ne yazılır?

   Cevap

   Ön koşullu döngüde koşul gövdeden önce sorulur: *10 < 5* yanlıştır ve gövde hiç çalışmaz; hiçbir şey yazılmaz. Son koşullu döngüde gövde önce çalışır: 10 yazılır, sayac 11 olur. Sonra *11 < 5* sorulur, yanlıştır ve döngü biter. Çıktı yalnızca 10'dur. İki döngü, koşul ilk denemede yanlış olduğunda farklı davranır.

9. Öklid algoritması (84, 36) için yürütülüyor. Her turda a, b ve k ne olur? Döngü kaç tur döner, karar kaç kez sorulur, ne yazılır?

   Cevap

   Başlangıç: a = 84, b = 36; *36 != 0* doğru. 1. tur: *k = 84 % 36 = 12* (84 = 2 × 36 + 12); a = 36, b = 12. 2. tur: *k = 36 % 12 = 0*; a = 12, b = 0. *0 != 0* yanlış; döngü biter. 2 tur, 3 karar; 12 yazılır. Denetim: *84 = 7 × 12*, *36 = 3 × 12* ve 7 ile 3'ün 1'den büyük ortak böleni yoktur.

4.2.12 İleri okuma

- C. Böhm, G. Jacopini, “Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules”, *Communications of the ACM*, 9(5), 1966: her akış diyagramının sıra, seçim ve tekrardan oluşan yapısal bir karşılığı olduğunu gösteren özgün makale.
- E. W. Dijkstra, “Go To Statement Considered Harmful”, *Communications of the ACM*, 11(3), 1968: rastgele atlayan deyimlerin programları neden anlaşılmaz kıldığını tartışan, yapısal programlama tartışmasını başlatan iki sayfalık mektup.
- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), Kesim 1.3: koşullar ve döngüler Java'da, izleme tablolarıyla ve çok sayıda kısa programla anlatılır.
- A. B. Downey, C. Mayfield, *Think Java* (2. baskı, O'Reilly, 2019): koşullar, mantıksal işleçler ve döngüler programlamaya yeni başlayanlar için sade bir dille anlatılır; kitabın çevrim içi sürümü ücretsizdir.

4.2.13 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. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11. baskı, Pearson, 2018.
3. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 1.3 Conditionals and Loops.

4. Oracle, *The Java Tutorials: Control Flow Statements*.

5. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019.

KISIM II · BÖLÜM 4.3

Programlamaya Giriş

Bir algoritma kâğıt üzerinde ne kadar açık yazılmış olursa olsun, işlemci onu doğrudan çalıştıramaz. İşlemcinin anladığı tek dil, sıfır ve birlerden oluşan makine dilidir. Programlama dili bu iki dünyanın arasında durur: insan onu okuyabilir, bir çeviri aracı da onu makine diline çevirebilir. Not, önce programlama dilinin ne olduğunu, kaynak kodun nasıl makine koduna çevrildiğini ve derleyici, yorumlayıcı ve sanal makinenin bu işi nasıl paylaştığını anlatır. Ardından dilleri düzey, tür, paradigma ve tipleme biçimine göre sınıflandırır. Son kesimler bir problemin programa dönüşmesini ve yazılım geliştirme aşamalarını anlatır. Değişken, karar, döngü, fonksiyon ve dizi gibi temel yapılar ile hata bulma yolları adım adım izleme tablolarıyla gösterilir. Kod örnekleri Java, Python ve C dillerindendir; Java ve Python örneklerinin çıktıları gerçek çalıştırmalardan alınmıştır.

ÖĞRENME ÇIKTILARI

Kavramlar
Programlama dilini, söz dizimini ve anlamı tanımlamak; kaynak kod, assembly ve makine dilini birbirinden ayırt etmek.

Çalıştırma
Derleme, yorumlama ve hibrit çalışmayı açıklamak; bytecode, sanal makine ve JIT kavramlarını yerine oturtmak; derleyicinin aşamalarını ve bağlayıcı ile yükleyicinin görevini sıralamak; bir hatanın derleme, çalışma zamanı ya da mantık hatası olduğunu belirlemek.

Diller
Dilleri düzey, tür ve paradigmaya göre sınıflandırmak; statik ve dinamik tiplemenin hatayı ne zaman yakaladığını açıklamak; sözde kodu Java ve Python'a çevirmek.

Problem çözme ve yazılım süreci
Bir problemi girdi, çıktı ve kısıtlarıyla tanımlayıp algoritmadan programa ulaşmak; yazılım geliştirme aşamalarını ve geliştirme araçlarını tanımak.

Temel yapılar ve hata ayıklama
Değişken, veri tipi, işleç önceliği, karar, döngü, fonksiyon ve diziyi kullanmak; bir programı izleme tablosuyla adım adım izleyerek hatasını bulmak.

4.3.1 Temel kavramlar

Bir yemek tarifi, mutfakta ne yapılacağını adım adım anlatır. Tarifi okuyan kişi "bir tutam tuz" gibi belirsiz bir ifadeyi kendi deneyimiyle tamamlar. Bilgisayarın böyle bir deneyimi yoktur. Ona verilen her adımın tek bir anlamı olmalı ve bu anlam önceden kesin olarak tanımlanmış olmalıdır. Programlama dilleri bu ihtiyaçtan doğdu.

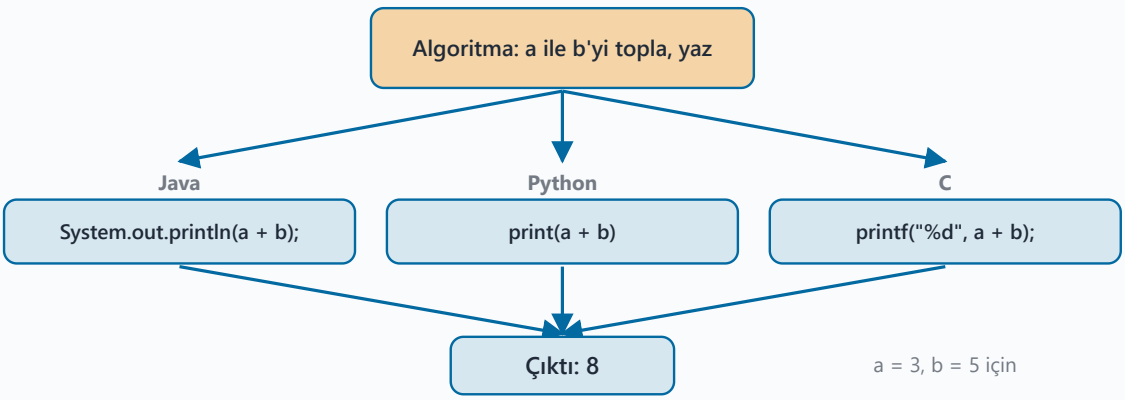
4.3.1.1 Programlama dili nedir?

Gündelik dil esnektir. "Yarın görüşürüz" cümlesinin saati belli değildir; dinleyen kişi bağlama bakarak anlamı tamamlar. Bir makinenin bağlamdan anlam çıkarması beklenemez. Makineye iş yaptırmak için her ifadenin tek bir yorumu olan, kuralları önceden yazılmış bir dil gerekir. Böyle dillere biçimsel dil denir.

Tanım 4.3.1

Programlama dili (*programming language*), insanların bilgisayara işlem adımlarını ifade etmesini sağlayan biçimsel bir dildir. Bu dille yazılmış komutlar topluluğuna **program** denir.

Programlama dili bir algoritmayı bilgisayarın yürütebileceği biçimde yazmanın aracıdır. Algoritma bir fikirdir: "a ile b'yi topla, sonucu yaz." Bu fikir Java'da, Python'da ya da C'de yazılabilir. Yazılışlar birbirinden farklıdır ama hepsi aynı adımları anlatır ve aynı girdi için aynı çıktıyı üretir. Şekil 4.3.1 bunu a = 3 ve b = 5 için gösterir: üç dilde de ekrana 8 yazılır.



Şekil 4.3.1: Aynı algoritma (a ile b'yi topla, yaz) Java, Python ve C ile yazılır. Yazım kuralları farklıdır; a = 3, b = 5 için üçü de 8 yazar.

Şekildeki satırlar tam programların yalnızca toplama kısmıdır. Bir programın çalışabilmesi için bunların çevresinde başka satırlar da gerekir. Aşağıda aynı işin Java ve C'deki tam hâli görülüyor. Java programı JDK 21 ile derlenip çalıştırılmıştır; C programı için çıktı satır satır izlenerek belirlenmiştir.

Örnek 4.3.1

a = 3 ve b = 5 değerlerini toplayıp yazan tam bir program Java ve C'de nasıl yazılır?

Java programı:

```java
public class Topla {
    public static void main(String[] args) {
        int a = 3;
        int b = 5;
        System.out.println(a + b);
    }
}
```

C programı:

```c
#include <stdio.h>

int main(void) {
    int a = 3;
    int b = 5;
    printf("%d\n", a + b);
    return 0;
}
```

İki programda da çıktı aynıdır:

```
8
```

Java'da her program bir `class` içinde yazılır ve çalışma `main` metodundan başlar. C'de çalışma `main` fonksiyonundan başlar; ekrana yazmak için `stdio.h` kütüphanesindeki `printf` kullanılır. `printf`'teki `%d` "buraya bir tam sayı gelecek" demektir, `\n` ise satır sonudur. Python'da aynı iş tek satırla yapılır: `print(a + b)`. Ayrıntılar farklıdır, ama algoritmanın üç adımı (a'yı hazırla, b'yi hazırla, toplamı yaz) üç programda da aynı sırayla bulunur.

Aynı problemin farklı dillerle çözülebilmesi bir seçim sorusu doğurur: hangi dil? Cevap problemin ihtiyaçlarına ve programın çalışacağı ortama bağlıdır. Bir telefonda çalışacak uygulama, bir sunucuda çalışacak hesap ve bir çamaşır makinesinin denetim yazılımı farklı dilleri gerektirebilir.

Sık yapılan hata

Programlamayı bir dili ezberlemek sanmak. Program yazmanın asıl işi algoritmayı kurmaktır: problemi anlamak, adımları bulmak, adımların doğru sırada ve doğru koşullarla çalıştığından emin olmak. Dil bu adımları yazmanın aracıdır. İyi kurulmuş bir algoritma bir dilden ötekine birkaç saatte aktarılabilir; kötü kurulmuş bir algoritmayı ise hiçbir dil kurtaramaz.

4.3.1.2 Söz dizimi ve anlam

Türkçede "Ali okula gitti" cümlesi doğru kurulmuştur. "Gitti okula Ali" cümlesi de anlaşılır ama kurallı değildir. "Okul Ali'ye gitti" ise kurallıdır ama anlamsızdır. Programlama dillerinde de bu iki boyut ayrı ayrı tanımlanır: bir yazılışın kurala uygun olup olmadığı ve kurala uygun bir yazılışın ne yaptığı.

Tanım 4.3.2

Bir programlama dilinin **söz dizimi** (*syntax*), hangi yapıların hangi sırayla yazılabileceğini belirleyen kurallardır. **Anlam** (*semantics*) ise kurala uygun yazılmış her yapının çalıştırıldığında ne yaptığını belirler.

Java'da her deyimin noktalı virgülle bitmesi bir söz dizimi kuralıdır. `int b = 5` satırının sonunda noktalı virgül yoksa yazılış kurala aykırıdır ve program çevrilemez. `a + b` ifadesinin a ile b'nin toplamını hesaplaması ise anlam kuralıdır. Python'da söz dizimi farklıdır: deyim sonunda noktalı virgül gerekmez, ama blokların girintisi kuralın parçasıdır.

Söz dizimi doğru olan her program istenen sonucu vermez. Üç notun ortalamasını hesaplarken toplamı 3 yerine 4'e bölen bir satır kurala tamamen uygundur. Çeviri aracı bu satırı sorunsuz kabul eder, program çalışır ve yanlış bir sayı yazar. Yazılış doğrudur ama yazılan şey, programcının istediği şey değildir. Böyle hatalara mantık hatası denir.

Örnek 4.3.2

Aşağıdaki üç Java satırının her biri hangi açıdan sorunludur: söz dizimi mi, anlam mı?

1. `int b = 5` (satır sonunda noktalı virgül yok)
2. `double ort = (n1 + n2 + n3) / 4;` (üç notun ortalaması isteniyor)
3. `System.out.println("Toplam: " + a + b);` (a = 3, b = 5 için "Toplam: 8" isteniyor)

Birinci satır söz dizimi hatasıdır; derleyici "';' expected" iletisiyle durur. İkinci satır kurala uygundur ama yanlış bölene böler: anlamı, programcının niyetiyle uyuşmaz. Üçüncü satır da kurala uygundur, ama Java'da `+` soldan sağa işlenir: önce "Toplam: " ile 3 birleşir, sonra 5 eklenir

ve ekranda “Toplam: 35” görünür. İstenen sonuç için toplamın paranteze alınması gerekir: `"Toplam: " + (a + b)`. Son iki satırdaki sorunu ancak dilin anlam kurallarını bilen biri görebilir.

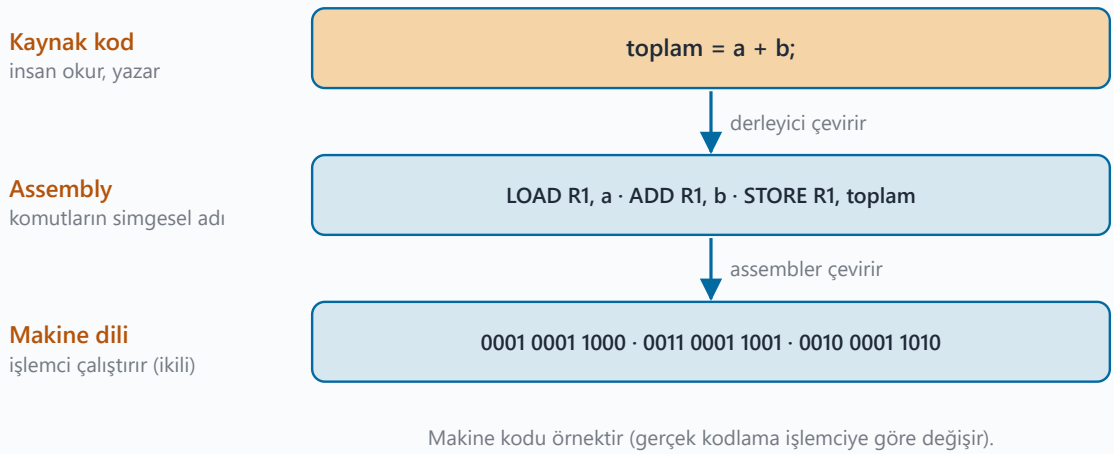
4.3.1.3 Kaynak kod, assembly ve makine dili

Programcının yazdığı metin bir metin dosyasında durur. Bu dosya Not Defteri ile açılıp okunabilir, değiştirilebilir. İşlemci ise bu metni okuyamaz. İşlemcinin anladığı komutlar ikili sayılarla kodlanmıştır ve her işlemci ailesinin kendi komut kümesi vardır.

Tanım 4.3.3

Kaynak kod (*source code*), bir programın geliştiricinin okuyup değiştirebildiği metin hâlidir. **Makine dili** (*machine language*), işlemcinin doğrudan çalıştırdığı ikili komutlardır; bu komutlardan oluşan programa **makine kodu** (*machine code*) denir. **Assembly** (*assembly language*), makine komutlarının insanın okuyabileceği simgesel adlarla yazılmış karşılığıdır.

Şekil 4.3.2 tek bir satırın bu üç biçimden nasıl geçtiğini gösterir. Derleyici, kaynak koddaki `toplam = a + b;` satırını üç işlemci komutuna çevirir. İlk komut a'nın değerini bellekten işlemcinin bir yazmacına (R1) yükler. İkincisi b'yi R1'e ekler. Üçüncüsü R1'deki sonucu bellekte toplam'ın yerine yazar. Assembly bu komutları LOAD, ADD ve STORE gibi kısa adlarla yazar. Assembler adı verilen bir araç da bu adları ikili kodlara çevirir. Şekildeki ikili kodlar yalnızca bir örnektir; gerçek kodlama işlemciden işlemciye değişir.



Şekil 4.3.2: Kaynak koddan makine diline. Derleyici tek bir satırı birkaç assembly komutuna, assembler da bu komutları ikili makine koduna çevirir.

Bu çeviriden iki sonuç çıkar. Birincisi, programlama dili ile işlemci dili aynı şey değildir. Java ya da Python bir işlemcinin dili değildir; bu dillerle yazılmış metnin işlemciye ulaşması için bir çeviri gerekir. Bu çeviriyi yapan programlara genel olarak **çeviri aracı** denir: derleyici ve yorumlayıcı. İkincisi, yüksek düzeyli bir satır çoğu zaman birden fazla makine komutuna karşılık gelir. Programcı "topla ve sakla" der; yükleme, toplama ve yazma adımlarını çeviri aracı ayrıntılandırır.

Sık yapılan hata

İşlemcinin Java ya da Python kaynak kodunu "anladığını" düşünmek. İşlemci yalnızca kendi makine dilini çalıştırır. `.java` ya da `.py` dosyasını çift tıklayıp çalıştırmaya çalışmak bu yüzden sonuç vermez. Dosyanın önce bir derleyiciden ya da yorumlayıcıdan geçmesi gerekir. Java'da bu iş iki parçaya bölünür: javac kaynak kodu bytecode'a çevirir, JVM de bytecode'u çalıştırır.

4.3.2 Programın çalıştırılması

Kaynak kod yazıldıktan sonra ekranda bir sonuç görünene kadar birkaç iş yapılır. Metin bir çeviri aracından geçer, çevrilen kod işlemcide ya da bir sanal makinede çalışır, çalışan program girdisini okuyup çıktısını üretir. Bu yolun her durağında farklı türden bir hata ortaya çıkabilir.

4.3.2.1 Kaynak koddan çıktıya

Şekil 4.3.3 bir Java programının yolunu beş durakta gösterir. Programcı kaynak kodu `Topla.java` adlı bir metin dosyasına yazar. Bilgisayar bu metni olduğu gibi çalıştıramaz. Çeviri durağında derleyici metni okur ve dilin kurallarına uyup uymadığını denetler. Hata yoksa bir çıktı dosyası üretir. C gibi dillerde bu dosya doğrudan makine kodudur. Java'da ise `Topla.class` adlı bir ara kod dosyasıdır. Çalıştırma durağında komutlar işlemcide yürütülür; Java'da bu iş Java Sanal Makinesi (JVM, *Java Virtual Machine*) üzerinden yapılır. Son durakta program çıktısını üretir: a = 3 ve b = 5 için 8.



Şekil 4.3.3: Kaynak koddan çıktıya giden hat. Kaynak kod çevrilir, çeviriden makine kodu ya da ara kod çıkar, bu kod işlemcide ya da JVM'de çalışır ve çıktı üretilir.

Java'da bu yol komut satırında iki komutla yürünür. `javac Topla.java` derleyiciyi çalıştırır ve `Topla.class` dosyasını üretir. `java Topla` ise JVM'yi başlatır ve bu dosyadaki programı çalıştırır. İki adımın ayrı olması önemlidir: derleme bir kez yapılır, üretilen dosya istendiği kadar çalıştırılabilir.

Kaynak kodda bir değişiklik yapılırsa derlemenin yinelenmesi gerekir; yoksa JVM eski `.class` dosyasını çalıştırmaya devam eder.

Sık yapılan hata

Kaynak kodu düzelttikten sonra yeniden derlemeden `java Topla` komutunu çalıştırıp "düzeltme işe yaramadı" sonucuna varmak. JVM kaynak kodu değil, derlenmiş `.class` dosyasını çalıştırır. Kaynak kod her değiştiğinde `javac` yeniden çalıştırılmalıdır. IDE'ler bu adımı kaydetme ya da çalıştırma düğmesine basma anında kendiliğinden yapar; bu yüzden adım çoğu zaman gözden kaçar.

4.3.2.2 Hata türleri

Hat üzerindeki her durak, kendine özgü bir hata türünü ortaya çıkarır. Bir hatanın hangi durakta göründüğünü bilmek, onu nerede arayacağımızı da söyler.

Tanım 4.3.4

Derleme hatası (*compile error*), kaynak kodun dilin kurallarına uymamasıdır; çeviri sırasında bulunur ve program hiç çalışmaz. Söz dizimi hatası (*syntax error*) ve tip hatası bu türdendir. **Çalışma zamanı hatası** (*runtime error*), program çalışırken ortaya çıkan ve programı durduran hatadır. **Mantık hatası** (*logic error*), programın hata iletisi vermeden çalışıp yanlış sonuç üretmesidir.

Üç türü birer küçük Java programıyla görmek kolaydır. İlk programda ikinci atama satırının sonundaki noktalı virgül unutulmuştur:

```
public class NoktaliVirgul {
    public static void main(String[] args) {
        int a = 3;
        int b = 5
        System.out.println(a + b);
    }
}
```

Derleyici programı çevirmeyi reddeder ve hatanın yerini satır numarasıyla bildirir:

```
NoktaliVirgul.java:4: error: ';' expected
        int b = 5
                ^
1 error
```

İleti dosya adını, satır numarasını (4) ve beklenen işareti (`;`) söyler. Alttaki `^` işareti, derleyicinin noktalı virgülü nerede beklediğini gösterir. Bu durumda `.class` dosyası üretilmez ve programın hiçbir satırı çalışmaz; ekranda "8" görünmez. Şekil 4.3.4 bu durumu hattın çeviri durağındaki kırmızı işaretle gösterir.

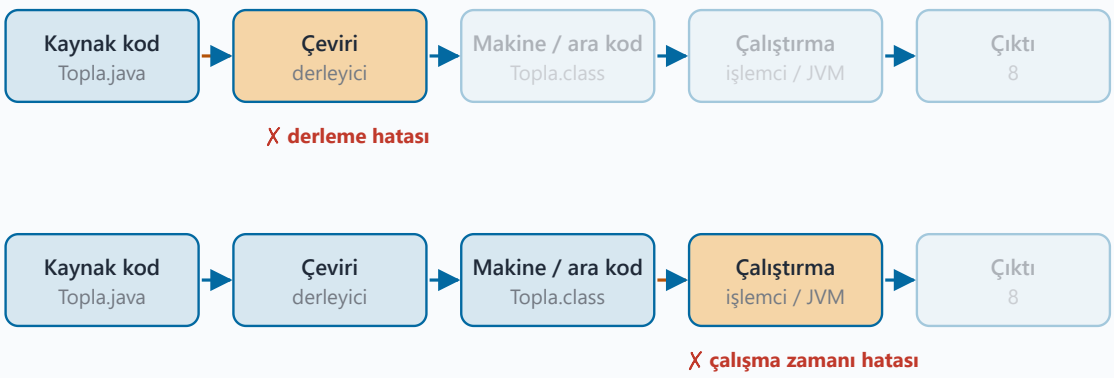
İkinci programda söz dizimi kusursuzdur ama üçüncü satırda sıfıra bölme vardır:

```
public class SifiraBolme {
    public static void main(String[] args) {
        System.out.println("Merhaba");
        int x = 10;
        System.out.println(x / 0);
        System.out.println("Son");
    }
}
```

Derleme başarıyla biter. Program çalışınca önce "Merhaba" yazılır, sonra `x / 0` hesaplanmaya çalışılır ve JVM programı durdurur:

```
Merhaba
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at SifiraBolme.main(SifiraBolme.java:5)
```

Çıktıdaki `ArithmeticException: / by zero` sıfıra bölme hatasını, `SifiraBolme.java:5` de hatanın kaynak dosyadaki satırını gösterir. "Son" satırı hiç çalışmaz. Derleyici bu hatayı yakalayamaz, çünkü x'in değeri ve bölme işlemi ancak çalışma sırasında hesaplanır. Bölen bir kullanıcı girdisi olsaydı, sıfır olup olmayacağını derleme sırasında bilmek zaten imkânsız olurdu.



Şekil 4.3.4: Hataların hat üzerindeki yeri. Üstte: söz dizimi hatası çeviri durağında yakalanır, program hiç çalışmaz. Altta: sıfıra bölme çalıştırma durağında ortaya çıkar; o ana kadarki satırlar çoktan çalışmıştır.

Üçüncü program üç notun ortalamasını hesaplar ama bir parantez eksiktir:

```java
public class OrtalamaHata {
    public static void main(String[] args) {
        int n1 = 70, n2 = 85, n3 = 90;
        System.out.println(n1 + n2 + n3 / 3);
        System.out.println((n1 + n2 + n3) / 3);
        System.out.println((n1 + n2 + n3) / 3.0);
    }
}
```

```
185
81
81.66666666666667
```

Program sorunsuz derlenir, hiçbir hata iletisi vermeden çalışır ve ilk satırda 185 yazar. Beklenen sonuç 81 ile 82 arasında bir sayıdır. Bölme toplamadan önce yapıldığı için yalnızca n3 üçe bölünmüştür: *70 + 85 + 90 / 3 = 70 + 85 + 30 = 185*. Bu bir mantık hatasıdır. İkinci satır parantezi ekler ama yine tam doğru değildir. Java'da iki tam sayının bölümü tam sayıdır; *245 / 3* işleminin kesirli kısmı atılır ve sonuç 81 olur. Üçüncü satır `3.0` ile böler; işlem ondalıklı yapılır ve doğru değer, yaklaşık 81,67 çıkar. Mantık hatalarının ortak özelliği budur: program "çalışır", sorunu ancak beklenen sonuçla gerçek sonucu karşılaştıran biri fark eder.

Hata türü	Ne zaman ortaya çıkar?	Kim bildirir?	Örnek
Derleme (söz dizimi, tip)	Çeviri sırasında; program hiç çalışmaz	Derleyici, satır numarasıyla	Unutulan `;`, metni int'e atamak
Çalışma zamanı	Yürütme sırasında, hatalı satıra gelince	JVM ya da işletim sistemi; program durur	Sıfıra bölme, dizinin dışına çıkmak
Mantık	Hiçbir zaman kendiliğinden görünmez	Kimse; ancak test ile bulunur	Eksik parantez, yanlış bölen

Tablo 4.3.1: Üç hata türünün karşılaştırması

Örnek 4.3.3

Aşağıdaki durumların her biri hangi hata türüdür? (a) Java'da `System.out.println("Merhaba")` satırının sonunda noktalı virgül yok. (b) Kullanıcıdan alınan öğrenci sayısı 0 girildiğinde ortalama hesaplanırken program duruyor. (c) Bir sınıfın ortalaması hesaplanırken toplam, öğrenci sayısı n yerine n + 1'e bölünüyor.

1. (a) Derleme hatasıdır. Noktalı virgül bir söz dizimi kuralıdır; derleyici satır numarasıyla bildirir ve program üretilmez.
2. (b) Çalışma zamanı hatasıdır. Kod kurala uygundur ve derlenir; ancak bölen 0 olduğu anda yürütme durur. Aynı program sıfırdan farklı girdilerle sorunsuz çalışır. Bu yüzden hata yalnızca belirli girdilerle ortaya çıkar.

3. (c) Mantık hatasıdır. Program her girdiyle çalışır ve bir sayı yazar; yazdığı sayı her zaman olması gerekenden küçüktür. Örneğin 60, 70 ve 80 notları için *210 / 4 = 52,5* yazılır; doğrusu *210 / 3 = 70*'tir.

Sık yapılan hata

Programın hata iletisi vermeden çalışmasını doğru çalıştığının kanıtı saymak. Derleyici yalnızca kurallara uyulup uyulmadığını denetler; programın istenen işi yapıp yapmadığını bilemez. Her programın, sonucu elle hesaplanmış birkaç girdiyle denenmesi gerekir.

4.3.2.3 Derlenen, yorumlanan ve hibrit diller

Kaynak kodu çalıştırmanın üç yaygın yolu vardır. Fark, çevirinin ne zaman ve ne kadarının yapıldığındadır.

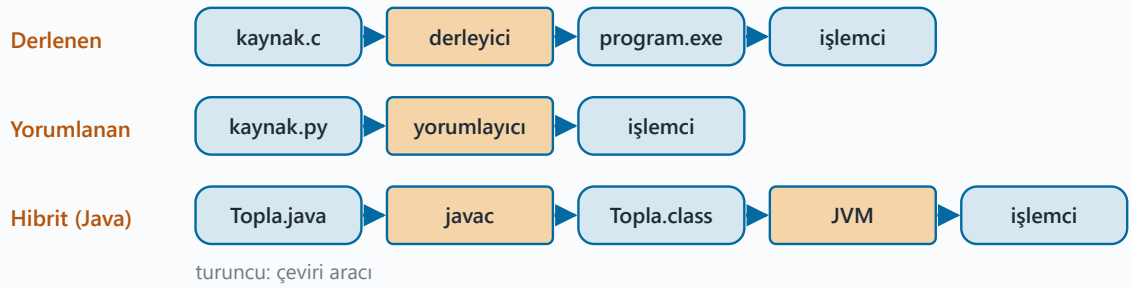
Tanım 4.3.5

Derleyici (*compiler*), programın tamamını çalıştırmadan önce hedef koda çeviren programdır; bu işe **derleme** (*compilation*) denir. **Yorumlayıcı** (*interpreter*), programı önceden bir çalıştırılabilir dosyaya çevirmeden, çalışma sırasında komut komut değerlendirerek yürüten programdır; bu işe **yorumlama** (*interpretation*) denir. **Hibrit** çalışmada kaynak kod önce bir ara koda derlenir, ara kod da bir sanal makinede yürütülür.

Yaklaşım	İşleyiş	Örnek diller
Derlenen	Program önceden hedef makine koduna çevrilir	C, C++, Rust
Yorumlanan	Komutlar çalışma sırasında değerlendirilir	Python, JavaScript
Hibrit	Ara kod sanal makinede yürütülür	Java, C#

Tablo 4.3.2: Çalıştırma yaklaşımları

Şekil 4.3.5 üç yolu yan yana gösterir. Derlenen yolda `kaynak.c` dosyası derleyiciden geçer ve `program.exe` gibi bir çalıştırılabilir dosya çıkar. Bu dosya işlemcide doğrudan çalışır; çalıştırma anında derleyiciye artık gerek yoktur. Yorumlanan yolda `kaynak.py` dosyası her çalıştırmada yorumlayıcıya verilir. Yorumlayıcı komutları sırayla işler ve işlemciye yaptırır. Ortada kalıcı bir makine kodu dosyası yoktur. Hibrit yolda `Topla.java` önce `javac` ile `Topla.class` dosyasına derlenir. Bu dosyayı JVM çalıştırır, JVM'nin kendisi de işlemcide çalışan bir programdır.



Şekil 4.3.5: Derlenen, yorumlanan ve hibrit çalıştırma hatları. Turuncu kutular çeviri araçlarıdır: derleyici, yorumlayıcı, javac ve JVM.

Hangi yolun seçildiği programın nasıl dağıtılacağını da belirler. Derlenmiş bir C programı yalnızca derlendiği işlemci ve işletim sistemi için geçerlidir. Windows için derlenmiş bir `.exe` dosyası Linux'ta ya da farklı işlemcili bir bilgisayarda çalışmaz. Yorumlanan bir Python programı kaynak kod olarak dağıtılır; Python yorumlayıcısı kurulu her bilgisayarda çalışır. Java'nın `.class` dosyası ise işlemciden bağımsızdır: aynı dosya, JVM kurulu her platformda çalışır. Java'nın bilinen "bir kez yaz, her yerde çalıştır" sloganı bu özellikten gelir.

Sık yapılan hata

Java'yı "derlenen bir dil" diye C ile aynı gruba koymak. Java kaynak kodu derlenir, ama derleyicinin ürettiği şey makine kodu değil, JVM'nin anladığı bytecode'dur. Bu kod işlemcide doğrudan çalışmaz, çalıştırmak için JVM gerekir. Java'yı hibrit sınıfa koymak bu iki aşamalı yapıyı doğru anlatır.

4.3.2.4 Derleyici ve yorumlayıcı adım adım

Derleyici ile yorumlayıcının farkı en iyi aynı küçük program üzerinde görülür. Aşağıdaki üç satırlık sözde kodun üçüncü satırında sıfıra bölme vardır:

Satır	Sözde kod
1	`yaz "Merhaba"`
2	`x ← 10`
3	`yaz x / 0`

Sözde kodda atama `←` ile gösterilir: `x ← 10`, "x'e 10 değerini ver" demektir. Böylece atama, eşitlik karşılaştırmasıyla karışmaz. Program önce bir derleyiciyle, sonra satır satır çalışan basit bir yorumlayıcıyla işlensin. Her adımda olanlar Tablo 4.3.3 içinde sıralanmıştır.

Adım	Derleyici	Basit yorumlayıcı
1	Satır 1, 2 ve 3 baştan sona okunur ve çevrilir; hiçbir satır çalışmadı	Satır 1 çevrilir ve hemen çalışır: `Merhaba`

Adım	Derleyici	Basit yorumlayıcı
2	Bölme kurala uygundur; çeviri başarılı, çalıştırılabilir dosya üretilir	Satır 2 çevrilir ve çalışır: x = 10; satır 3 henüz okunmadı
3	Çalıştırma başlar; satır 1: `Merhaba`	Satır 3 çevrilir, çalışırken hata: sıfıra bölme
4	Satır 2: x = 10; satır 3: çalışma zamanı hatası	—
Ekran	`Merhaba`, ardından hata	`Merhaba`, ardından hata

Tablo 4.3.3: Aynı programın derleyici ve yorumlayıcıyla işlenmesi

Bu programda iki yol aynı ekranla biter: “Merhaba” ve ardından hata. Fark, hatanın türünde değil, işin sırasındadır. Derleyici önce bütün metni çevirir, çalıştırma ayrı bir aşama olarak sonra gelir. Yorumlayıcı çevirme ile çalıştırmayı satır satır iç içe yürütür. Sıfıra bölme çalışma zamanı hatası olduğu için iki yolda da ancak üçüncü satır çalışırken görünür. Şekil 4.3.6 iki yolun son durumunu gösterir.



Şekil 4.3.6: Üç satırlık programın son durumu. Üstte derleyici: üç satırın tamamı önce çevrildi (✓), çalıştırmada satır 3 hata verdi. Altta yorumlayıcı: her satır çevrilip hemen çalıştı; satır 3'e gelince hata.

Hata bir söz dizimi hatası olduğunda iki yol birbirinden ayrılır. Üçüncü satırda bir parantez kapanmamış olsun: `yaz (x + 2`. Derleyici çeviri sırasında bu satıra takılır ve hiçbir çalıştırılabilir dosya üretmez. Ekranda yalnızca “satır 3: söz dizimi hatası” iletisi görünür; “Merhaba” bile yazılmaz. Satır satır çalışan basit bir yorumlayıcı ise satır 1'i çalıştırıp “Merhaba” yazar, satır 2'de x'e 10 atar ve ancak satır 3'e gelince hatayı fark eder.

Python ve JavaScript gibi bugünkü yorumlayıcılar bu basit modelden biraz ayrılır: çalıştırmaya başlamadan önce dosyanın tamamını ayrıştırırlar. Bu yüzden söz dizimi hatasını en başta bildirirler. Aşağıdaki Python dosyası bunu gösterir:

```
print("Merhaba")
x = 10
print((x + 2)
print("Son")
```

```
  File "sozdizimi.py", line 3
    print((x + 2)
          ^
SyntaxError: '(' was never closed
```

Ekranda “Merhaba” yoktur. Python, birinci satırı çalıştırmadan önce dosyanın tamamını okumuş ve üçüncü satırdaki kapanmamış parantezi bulmuştur. Aynı dosyada söz dizimi hatası yerine sıfıra bölme olduğunda durum değişir:

```
print("Merhaba")
x = 10
print(x / 0)
print("Son")
```

```
Merhaba
Traceback (most recent call last):
  File "sifira.py", line 3, in <module>
    print(x / 0)
          ~~^~~
ZeroDivisionError: division by zero
```

Bu kez “Merhaba” yazılmıştır, çünkü sıfıra bölme ancak üçüncü satır çalışırken anlaşılır. Python'un iletisi Java'nınkine benzer: dosya adı, satır numarası, hata türü (`ZeroDivisionError`) ve kısa açıklama.

Örnek 4.3.4

Dört satırlık bir programın 2. satırında söz dizimi hatası, 4. satırında sıfıra bölme var. Program (a) derleyiciyle, (b) satır satır çalışan basit bir yorumlayıcıyla, (c) Python ile çalıştırılırsa ekranda ne görünür? Satır 1 “Başla”, satır 3 “Orta” yazıyor.

1. (a) Derleyici önce dört satırın hepsini çevirir. Satır 2'deki söz dizimi hatasında durur ve çalıştırılabilir dosya üretmez. Ekranda yalnızca satır 2 için bir derleme hatası iletisi görünür; “Başla” yazılmaz.
2. (b) Basit yorumlayıcı satır 1'i çevirip çalıştırır: “Başla”. Satır 2'yi çevirmeye çalışınca söz dizimi hatasıyla durur. Satır 3 ve 4'e hiç ulaşılmaz. Ekranda “Başla” ve hata iletisi görünür.

3. (c) Python dosyanın tamamını önce ayrıştırır ve satır 2'deki hatayı bulur. Ekranda yalnızca `SyntaxError` iletisi görünür; davranış (a) ile aynıdır. Satır 4'teki sıfıra bölme, üç durumda da hiç ortaya çıkmaz; çünkü programın oraya kadar çalışması gerekirdi.

4.3.2.5 Bytecode, sanal makine ve JIT

Python'a "yorumlanan dil" denir, ama Python yorumlayıcısının içinde de bir derleme adımı vardır. Python'un en yaygın uygulaması olan CPython, kaynak kodu önce **bytecode** adı verilen bir ara koda derler. Sonra bu bytecode'u kendi içindeki bir sanal makinede komut komut yürütür. Ortaya bir makine kodu dosyası çıkmaz. Derleme adımı programcıdan gizlidir: `python dosya.py` komutu iki aşamayı art arda yapar.

Tanım 4.3.6

Bytecode (*bytecode*, ara kod), belirli bir işlemciye değil, bir sanal makineye göre tasarlanmış basit komutlardan oluşan ara koddur. **Sanal makine** (*virtual machine*), bu komutları okuyup yürüten yazılımdır; gerçek bir işlemcinin işini yazılımla taklit eder.

Python'un ürettiği bytecode, `dis` modülüyle görülebilir. Aşağıdaki program `alan = en * boy + 2` satırını derler ve bytecode'unu yazdırır:

```
import dis
dis.dis(compile("alan = en * boy + 2", "<kod>", "exec"))
```

```
  0           RESUME                   0

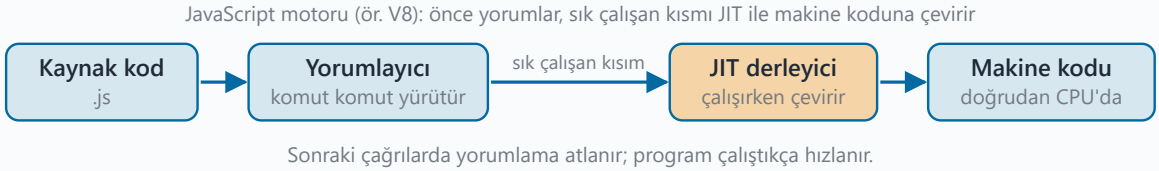
  1           LOAD_NAME                0 (en)
              LOAD_NAME                1 (boy)
              BINARY_OP                5 (*)
              LOAD_CONST               0 (2)
              BINARY_OP                0 (+)
              STORE_NAME               2 (alan)
              RETURN_CONST             1 (None)
```

Çıktıdaki her satır bir sanal makine komutudur. `LOAD_NAME` bir değişkenin değerini sanal makinenin yığınına koyar. `BINARY_OP 5 (*)` yığının üstündeki iki değeri çarpar, `LOAD_CONST` 2 sabitini yığına ekler, `BINARY_OP 0 (+)` toplar. `STORE_NAME` de sonucu alan adına bağlar. Komut adları Python sürümüne göre biraz değişebilir; burada Python 3.13 kullanılmıştır. Sıralama ise her sürümde aynıdır: önce en ile boy çarpılır, sonra 2 eklenir. Bu sıra işleç önceliğinden gelir.

Bytecode'u komut komut yürütmek, makine kodunu doğrudan çalıştırmaktan yavaştır. Her komut için sanal makinenin “bu komut ne demek?” sorusunu yanıtlaması gerekir. Bu yavaşlığı azaltmanın yolu **JIT derleme**dir.

Tanım 4.3.7

JIT derleyici (*just-in-time compiler*, tam zamanında derleyici), program çalışırken sık çalışan kod parçalarını makine koduna çeviren derleyicidir. Çevrilen parça sonraki çalışmalarda yorumlanmaz, doğrudan işlemcide çalışır.



Şekil 4.3.7: JIT derleme. JavaScript motoru (ör. V8) önce kodu yorumlar; sık çalışan kısmı çalışma sırasında makine koduna çevirir. Sonraki çağrılarda yorumlama atlanır ve program çalıştıkça hızlanır.

Şekil 4.3.7 bu fikri JavaScript üzerinden gösterir. Chrome tarayıcısındaki V8 motoru kodu önce yorumlar. Bir fonksiyonun binlerce kez çağrıldığını görünce onu makine koduna derler. JVM de aynı yolu izler: bytecode önce yorumlanır, sık çalışan metotlar JIT ile makine koduna çevrilir. Bu yüzden uzun süre çalışan Java programları, derlenmiş C programlarına yakın hızlara ulaşabilir.

Bu örnekler önemli bir noktayı ortaya koyar: “derlenen” ya da “yorumlanan” bir dilin değil, o dilin uygulamasının özelliğidir. Python için makine koduna derleyen uygulamalar da yazılmıştır, C için yorumlayıcılar da. Gerçek sistemlerin çoğu iki yöntemi birleştirir. Bir dile “yorumlanan” demek, o dilin en yaygın uygulamasının nasıl çalıştığını söylemenin kısa yoludur.

Sık yapılan hata

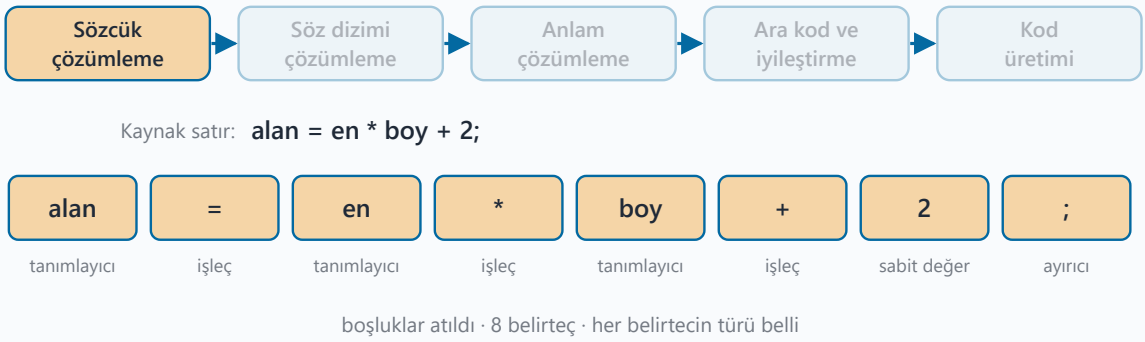
“Python yorumlanan bir dil olduğu için hiçbir derleme yapılmaz” demek. CPython her çalıştırmada kaynak kodu bytecode'a derler. İçe aktarılan modüllerin bytecode'unu `__pycache__` klasöründe `.pyc` dosyası olarak saklar; kaynak değişmediyse sonraki çalıştırmalarda bu dosyayı kullanır. Python'u yorumlanan yapan, bytecode'un makine koduna değil, bir sanal makinede komut komut yürütülmesidir.

4.3.2.6 Derleyicinin aşamaları

Derleyici tek bir büyük adımda çalışmaz. İşini birbirini izleyen beş aşamaya böler; her aşamanın çıktısı bir sonrakinin girdisidir. Aşamaları tek bir satır üzerinde izlemek, derleyicinin hangi hatayı nerede yakaladığını da açıklar. İzlenecek satır şudur:

```
alan = en * boy + 2;
```

1. Sözcük çözümleme (*lexical analysis*). Derleyici satırı bir karakter dizisi olarak görür: a, l, a, n, boşluk, = … İlk aşama bu karakterleri anlamlı parçalara ayırır. Bu parçalara **belirteç** (*token*) denir. Boşluklar ve açıklamalar bu aşamada atılır. Örnek satırdan sekiz belirteç çıkar ve her birinin türü bellidir (Şekil 4.3.8).



Şekil 4.3.8: Sözcük çözümleme. `alan = en * boy + 2;` satırı sekiz belirtece ayrılır: üç tanımlayıcı, üç işleç, bir sabit değer ve bir ayırıcı.

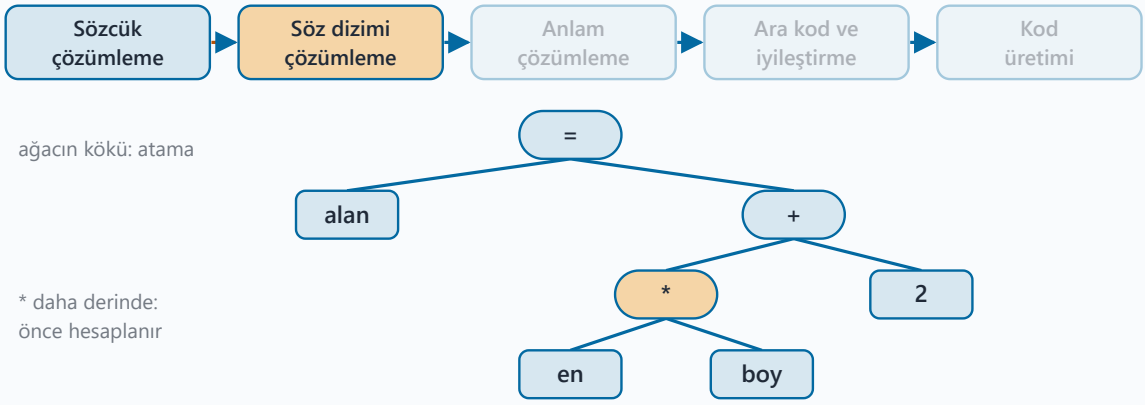
Bu aşamada yakalanan hatalar, dilde hiçbir belirtece uymayan karakter dizileridir. Java'da `#` karakteri bir işleç değildir. Kapanmamış bir dizgi de belirtece dönüştürülemez. JDK 21'in derleyicisi bu iki durumu şöyle bildirir:

```
GecersizKarakter.java:4: error: illegal character: '#'
        int alan = en # boy;
                      ^
GecersizKarakter.java:4: error: not a statement
        int alan = en # boy;
                       ^
2 errors
```

```
KapanmamisDizgi.java:3: error: unclosed string literal
        System.out.println("Merhaba);
                           ^
1 error
```

2. Söz dizimi çözümleme (*parsing, syntax analysis*). Belirteçler dilin kurallarına göre bir ağaca dizilir; bu ağaca **söz dizimi ağacı** (*syntax tree*) denir. Şekil 4.3.9 örnek satırın ağacını gösterir. Kökte atama (`=`) vardır; sol kolu alan, sağ kolu `en * boy + 2` ifadesidir. Çarpma ağacın daha derinindedir,

bu yüzden önce hesaplanır. İşleç önceliği bu ağacın biçimine böylece yerleşir.

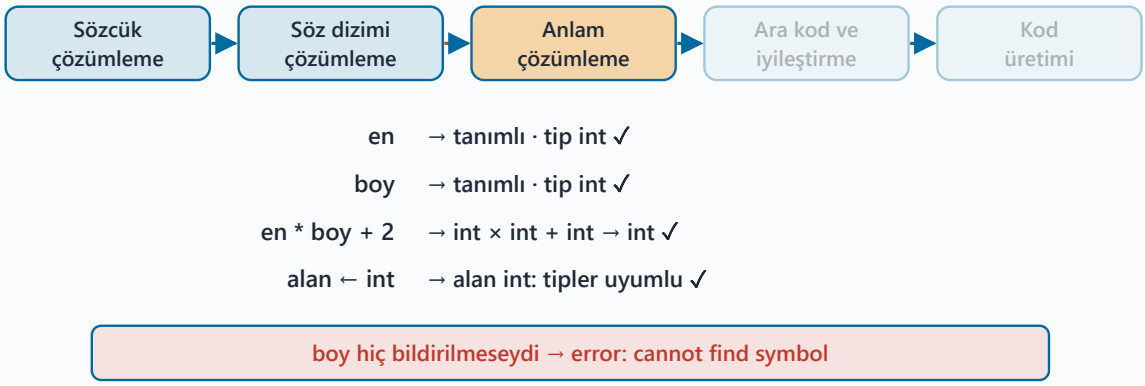


Şekil 4.3.9: Söz dizimi çözümleme. Belirteçler bir ağaca dizilir; kökte atama vardır, daha derindeki çarpma önce hesaplanır.

Eksik noktalı virgül, kapanmamış parantez ve yanlış sırada yazılmış belirteçler bu aşamada yakalanır. Örneğin `int 2x = 5;` satırında sözcük çözümleme bir sorun görmez: 2 bir sayı, x bir tanımlayıcıdır. Ama `int` belirtecinden sonra bir sayının gelmesi kurala aykırıdır. Bu yüzden derleyici hatayı söz dizimi aşamasında bildirir:

```
GecersizSozcuk.java:3: error: not a statement
        int 2x = 5;
        ^
GecersizSozcuk.java:3: error: ';' expected
        int 2x = 5;
           ^
2 errors
```

3. Anlam çözümleme (*semantic analysis*). Söz dizimi doğru olan bir ağacın anlamlı olup olmadığı denetlenir. Kullanılan her adın daha önce bildirilmiş olması gerekir. İşlenenlerin tipleri de birbirine uymalıdır. Örnek satırda en ve boy int olarak bildirilmiştir; *int × int + int* ifadesi int verir ve int tipindeki alan'a atanabilir (Şekil 4.3.10).



Şekil 4.3.10: Anlam çözümleme. Her adın tanımlı olduğu ve tiplerin uyumlu olduğu denetlenir; boy hiç bildirilmeseydi "cannot find symbol" hatası verilirdi.

boy değişkeni bildirilmeden kullanılırsa javac şu iletiyi verir:

```
public class Tanimsiz {
    public static void main(String[] args) {
        int en = 3;
        int alan = en * boy + 2;
    }
}
```

```
Tanimsiz.java:4: error: cannot find symbol
        int alan = en * boy + 2;
                        ^
  symbol:   variable boy
  location: class Tanimsiz
1 error
```

İleti, bulunamayan simgenin bir değişken olduğunu ve adının boy olduğunu söyler. Tip uyuşmazlığı da bu aşamanın işidir. int tipindeki bir değişkene metin atamak söz dizimi açısından kusursuzdur, ama anlam açısından geçersizdir:

```
public class TipHatasi {
    public static void main(String[] args) {
        int x = 5;
        x = "beş";
    }
}
```

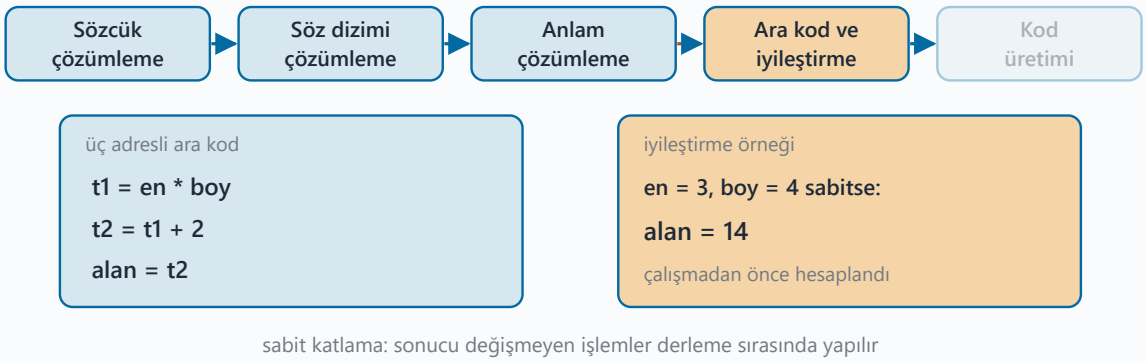
```
TipHatasi.java:4: error: incompatible types: String cannot be converted to int
        x = "beş";
            ^
1 error
```

İleti “String tipi int'e dönüştürülemez” demektir. Derleyici hatayı, program hiç çalışmadan, 4. satırdaki `"beş"` değerinin altında gösterir.

4. Ara kod üretimi ve iyileştirme (*intermediate code, code optimization*). Denetimden geçen ağaç, her biri tek bir işlem yapan basit komutlara dönüştürülür. Bu biçime üç adresli kod denir, çünkü her komutta en fazla üç ad bulunur:

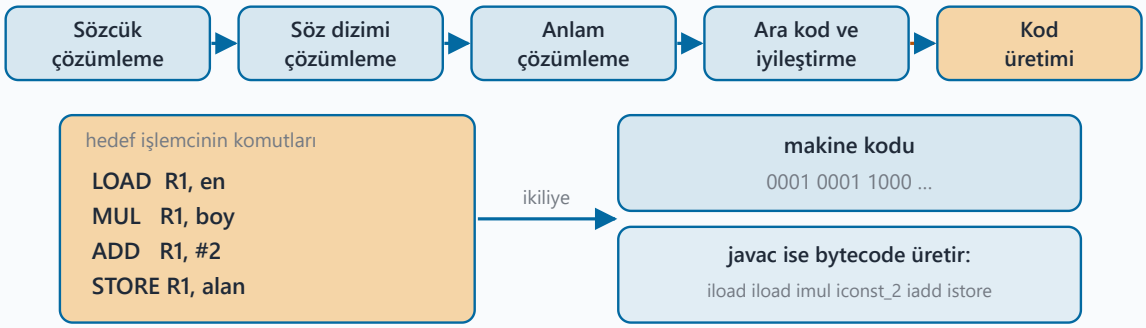
```
t1 = en * boy
t2 = t1 + 2
alan = t2
```

Burada t1 ve t2 derleyicinin uydurduğu geçici adlardır. Bu biçim üzerinde **kod iyileştirme** yapılır: gereksiz işlemler atılır, sonucu derleme sırasında bilinen işlemler önceden hesaplanır. en = 3 ve boy = 4 değişmeyen sabitlerse, *3 × 4 + 2 = 14* hesabı program çalışmadan yapılır ve kodda yalnızca `alan = 14` kalır. Bu iyileştirmeye **sabit katlama** (*constant folding*) denir (Şekil 4.3.11).



Şekil 4.3.11: Ara kod ve iyileştirme. İfade, her biri tek işlem yapan komutlara bölünür; değerleri sabit olan işlemler derleme sırasında hesaplanır.

5. Kod üretimi (*code generation*). Son aşamada hedef işlemcinin komutları seçilir ve ikili koda çevrilir. Örnek satır için bir işlemcide LOAD, MUL, ADD ve STORE komutları üretilir (Şekil 4.3.12). Java derleyicisinin hedefi gerçek bir işlemci değil, JVM'dir. Bu yüzden javac makine komutları yerine bytecode üretir.



Şekil 4.3.12: Kod üretimi. Hedef işlemcinin komutları seçilir ve ikiliye çevrilir; javac ise JVM için bytecode üretir.

Örnek 4.3.5

javac'ın ürettiği bytecode, JDK ile gelen `javap -c` komutuyla okunabilir. Aşağıdaki iki program arasındaki tek fark, ikincisinde en ve boy'un `final` (değişmez) olarak bildirilmesidir. İki programın bytecode'u nasıl ayrılır?

```
public class AlanIfade {
    public static void main(String[] args) {
        int en = 3;
        int boy = 4;
        int alan = en * boy + 2;
        System.out.println(alan);
    }
}
```

```
public static void main(java.lang.String[]);
    Code:
       0: iconst_3
       1: istore_1
       2: iconst_4
       3: istore_2
       4: iload_1
       5: iload_2
       6: imul
       7: iconst_2
       8: iadd
       9: istore_3
      10: getstatic     #7                  // Field
java/lang/System.out:Ljava/io/PrintStream;
      13: iload_3
      14: invokevirtual #13                 // Method java/io/PrintStream.println:
(I)V
      17: return
}
```

İlk programın bytecode'unda `iconst_3` ve `istore_1` 3 değerini 1 numaralı yerel değişkene (en) yazar. `iconst_4` ve `istore_2` 4'ü boy'a yazar. Sonraki beş komut (`iload_1` , `iload_2` , `imul` , `iconst_2` , `iadd`) ifadeyi hesaplar, `istore_3` de sonucu alan'a yazar. Bu sıra, sunumdaki “iload iload imul iconst_2 iadd istore” dizisinin aynısıdır.

```
public class SabitKatlama {
    public static void main(String[] args) {
        final int en = 3;
        final int boy = 4;
        int alan = en * boy + 2;
        System.out.println(alan);
    }
}
```

```
public static void main(java.lang.String[]);
    Code:
       0: bipush        14
       2: istore_1
       3: getstatic     #7                  // Field
java/lang/System.out:Ljava/io/PrintStream;
       6: iload_1
       7: invokevirtual #13                 // Method java/io/PrintStream.println:
(I)V
      10: return
}
```

İkinci programda çarpma ve toplama komutları hiç yoktur. `bipush 14` komutu 14 değerini doğrudan yığına koyar. en ve boy `final` olduğu için derleyici değerlerinin değişmeyeceğini bilir ve *3 × 4 + 2* işlemini derleme sırasında yapar. Bu, sabit katlamanın gerçek bir örneğidir. İki program da çalışınca 14 yazar.

İlk üç aşama hata arar ve bulduğu hatayı satır numarasıyla bildirir. Son iki aşama çalışacak kodu üretir. Sıfıra bölme gibi değere bağlı hatalar hiçbir aşamada görünmez, çünkü derleyici programı çalıştırmaz; yalnızca metnine bakar.

Aşama	Girdi → çıktı	Yakalanan hata (javac iletisi)
Sözcük çözümleme	karakterler → belirteçler	Geçersiz karakter (`illegal character`), kapanmamış dizgi (`unclosed string literal`)
Söz dizimi çözümleme	belirteçler → söz dizimi ağacı	Eksik `;` (`';' expected`), kapanmamış parantez
Anlam çözümleme	ağaç → denetlenmiş ağaç	Tanımsız ad (`cannot find symbol`), tip uyuşmazlığı (`incompatible types`)
Ara kod ve iyileştirme	ağaç → basit komutlar	—
Kod üretimi	basit komutlar → makine kodu ya da bytecode	—

Tablo 4.3.4: Derleyicinin aşamaları ve yakaladığı hatalar

Sık yapılan hata

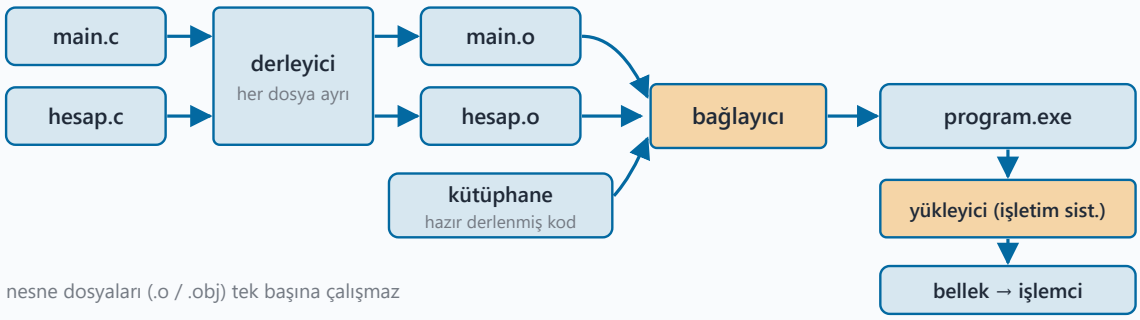
Derleyicinin bildirdiği satır numarasını her zaman hatanın kendisi sanmak. Derleyici hatayı fark ettiği yeri bildirir; bu yer bazen asıl hatanın bir satır sonrasıdır. Bir satırın sonundaki noktalı virgül unutulduğunda bazı derleyiciler hatayı bir sonraki satırın başında görür. İleti bir satırı gösteriyorsa, bir önceki satıra da bakmak gerekir.

4.3.2.7 Bağlayıcı ve yükleyici

Gerçek programlar tek bir dosyadan oluşmaz. Bir hesap makinesi programında kullanıcı arayüzü bir dosyada, hesap fonksiyonları başka bir dosyada durabilir. Programın ekrana yazması, dosya açması ya da karekök alması için ayrıca hazır kütüphane kodu kullanılır. Derleyici her kaynak dosyayı ayrı ayrı çevirir. Bu parçaların tek bir programda birleşmesi ve çalışmak üzere belleğe alınması iki ayrı aracın işidir.

Tanım 4.3.8

Nesne dosyası (*object file*), tek bir kaynak dosyanın derlenmesiyle oluşan, tek başına çalışmayan makine kodu dosyasıdır (`.o` ya da `.obj`). **Bağlayıcı** (*linker*), nesne dosyalarını ve kullanılan kütüphane kodunu tek bir **çalıştırılabilir dosyada** (*executable file*) birleştiren programdır. **Yükleyici** (*loader*), işletim sisteminin çalıştırılabilir dosyayı belleğe alıp çalıştırmayı başlatan parçasıdır.



Şekil 4.3.13: İki kaynak dosya ayrı derlenir; bağlayıcı nesne dosyalarını ve kütüphaneyi birleştirir, yükleyici programı belleğe alır.

Şekil 4.3.13 bir C programının yolunu gösterir. `main.c` ana programı, `hesap.c` hesap fonksiyonlarını içerir. Derleyici bunları `main.o` ve `hesap.o` nesne dosyalarına çevirir. `main.o` içinde `hesap.c`'deki bir fonksiyona yapılan çağrı vardır, ama bu fonksiyonun adresi henüz bilinmez. Derleyici yalnızca "burada adı şu olan bir fonksiyon çağrılıyor" notunu bırakır. Bağlayıcı bu notları çözer. Çağrılan fonksiyonu `hesap.o`'da, `printf` gibi hazır fonksiyonları kütüphanede bulur. Her çağrıyı doğru adrese bağlar ve sonunda `program.exe` dosyasını yazar.

Çağrılan bir fonksiyon hiçbir parçada bulunamazsa bağlayıcı **bağlama hatası** (*linking error*) verir. Bu hata derleme hatasından farklıdır. Her dosya tek başına kusursuz derlenmiş olabilir; eksik olan, parçalar arasındaki bağdır. Örneğin `main.c`'de çağrılan `ortalama` fonksiyonu `hesap.c`'de `ortalma` diye yanlış yazılmışsa iki dosya ayrı ayrı derlenir, ama bağlayıcı `ortalama` adını hiçbir yerde bulamaz.

Program çift tıklandığında ya da komut satırından çağrıldığında yükleyici devreye girer. Dosyayı diskten belleğe kopyalar. Programın ihtiyaç duyduğu paylaşılan kütüphaneleri de yükleyip bağlar: Windows'ta `.dll`, Linux'ta `.so` dosyaları. Son olarak işlemciyi programın ilk komutuna yönlendirir. Paylaşılan bir kütüphaneyi birçok program ortak kullanır; aynı kodun her programa ayrı ayrı kopyalanmasına gerek kalmaz.

Java'da ayrı bir bağlama adımı yoktur. Her sınıf kendi `.class` dosyasına derlenir. JVM'nin **sınıf yükleyicisi** (*class loader*), program çalışırken bir sınıfa ilk kez ihtiyaç duyulduğu anda o sınıfın dosyasını bulup yükler. Bir sınıf dosyası eksikse hata derleme ya da bağlama sırasında değil, çalışma sırasında ortaya çıkar.

Sık yapılan hata

Bir nesne dosyasını (`.o` , `.obj`) çalıştırılabilir program sanmak. Nesne dosyasında makine kodu vardır ama başka dosyalara ve kütüphanelere yapılan çağrılar henüz bağlanmamıştır. Çalışabilmesi için bağlayıcıdan geçmesi gerekir.

4.3.2.8 Derleme mi, yorumlama mı?

Derleme ile yorumlama arasındaki seçim bir değiş tokuştur. Bir yanda hız ve erken hata denetimi, öbür yanda hızlı deneme ve esneklik vardır. Tablo 4.3.5 üç yaklaşımı beş ölçüte göre karşılaştırır.

Ölçüt	Derlenen (C)	Yorumlanan (Python)	Hibrit (Java)
Çalışma hızı	Genellikle en hızlı: doğrudan makine kodu	Daha yavaş: komutlar çalışırken çözülür	JIT sayesinde derlenene yakın
Hata bildirimi	Söz dizimi ve tip hataları çalıştırmadan önce	Tip hataları çoğunlukla ancak o satır çalışınca	Çoğu derlemede, kalanı çalışırken
Dağıtım	Her işletim sistemi ve işlemci için ayrı dosya	Kaynak kod + hedefte kurulu yorumlayıcı	Tek `.class` / `.jar` + her platformda JVM
Geliştirme döngüsü	Her değişiklikten sonra yeniden derleme	Kaydet ve hemen çalıştır	Derle, sonra çalıştır
Başlangıç	Hemen başlar, ek yazılım gerekmez	Önce yorumlayıcı yüklenir	Önce JVM başlatılır

Tablo 4.3.5: Derlenen, yorumlanan ve hibrit çalışmanın karşılaştırması

Bir oyun motoru ya da bir video kodlayıcı her milisaniyeyi önemser; derlenmiş C ya da C++ bu yüzden tercih edilir. Bir veri analisti bir veri kümesine farklı sorular sorarken her denemeden sonra derleme beklemek istemez; Python'un “kaydet ve çalıştır” döngüsü ona uygundur. Bir bankanın sunucu yazılımı yıllarca çalışır ve farklı donanımlara taşınır. Java'nın taşınabilir bytecode'u ve JIT ile kazandığı hız bu ihtiyaca uyar. Aynı dil için hem derleyici hem yorumlayıcı yazılabilmesi de bu değiş tokuşun sonucudur.

Örnek 4.3.6

Bir öğrenci aynı programı üç farklı bilgisayarda çalıştırmak istiyor: Windows dizüstü, Linux sunucu ve macOS masaüstü. Program C, Python ve Java ile yazılmış olsaydı her durumda ne dağıtması gerekirdi?

1. C: Her platform için ayrı ayrı derlenmiş üç çalıştırılabilir dosya. Windows'ta üretilen `.exe` Linux'ta çalışmaz. Kaynak kod da dağıtılabilir; o zaman her bilgisayarda bir C derleyicisi bulunmalıdır.
2. Python: Kaynak kod dosyası (`.py`). Üç bilgisayarda da Python yorumlayıcısının kurulu olması gerekir.

3. Java: Tek bir `.class` ya da `.jar` dosyası. Üç bilgisayarda da JVM bulunmalıdır; aynı dosya üçünde de değişmeden çalışır.

4.3.3 Programlama dilleri

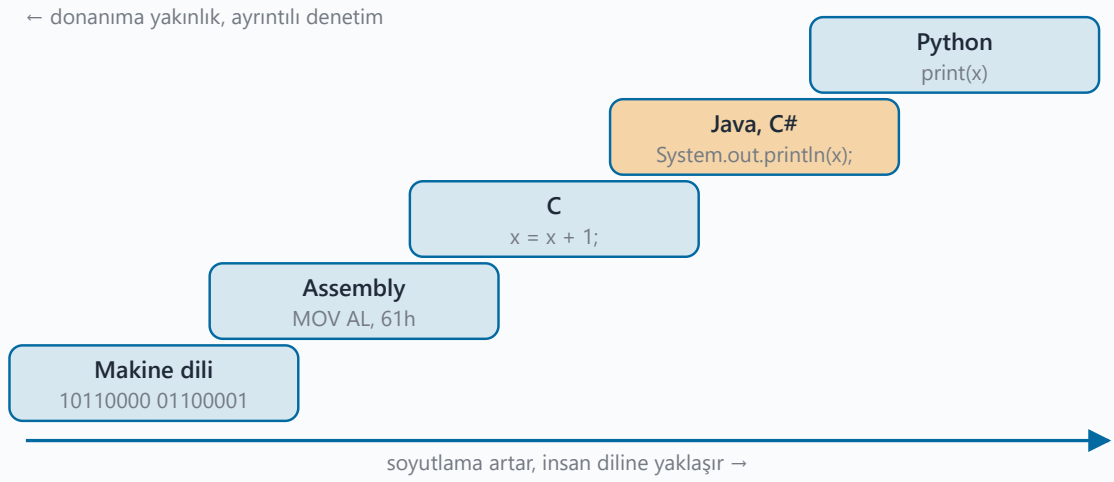
Bugün kullanılan programlama dillerinin sayısı yüzlerle ölçülür. Hepsini tek tek tanımak ne mümkündür ne de gereklidir. Dilleri birkaç ölçüte göre gruplamak, yeni bir dille karşılaşıldığında onu yerine oturtmayı kolaylaştırır. Donanıma yakınlık, ortaya çıkış dönemi, kullanım amacı, düşünme biçimi ve tip denetiminin zamanı bu ölçütlerin başlıcalarıdır.

4.3.3.1 Dillerin düzeyi

Tanım 4.3.9

Düşük düzeyli dil (*low-level language*), komutları işlemcinin komutlarına bire bir ya da bire yakın karşılık gelen dildir: makine dili ve assembly. **Yüksek düzeyli dil** (*high-level language*), problemi insanın ifade biçimine yakın yazdıran, donanım ayrıntılarını gizleyen dildir: C, Java, Python.

Şekil 4.3.14 dilleri bir merdiven üzerinde dizer. En altta makine dili vardır: `10110000 01100001`. Bu iki bayt, Intel uyumlu işlemcilerde "AL yazmacına 61h (onaltılı 61, onluk 97) değerini yükle" komutudur. Assembly aynı komutu `MOV AL, 61h` diye yazar. Bir basamak yukarıda C vardır: `x = x + 1;` satırı yazmaçlardan söz etmez, bir değişkenden söz eder. Java ve C# daha da yukarıdadır; bellek yönetimini programcıdan alır. Python en üsttedir: `print(x)` yazmak için bir sınıf ya da tip bildirimi gerekmez.



Şekil 4.3.14: Dil düzeyleri merdiveni. Yukarı çıkıldıkça soyutlama artar ve dil insan diline yaklaşır; aşağı inildik-

çe donanıma yakınlık ve ayrıntılı denetim artar.

Merdivende yukarı çıkmak bir kazanç ve bir bedel getirir. Kazanç, aynı işi daha az ve daha anlaşılır satırla yazmaktır. Bir dizgiyi ekrana yazmak Python'da tek satırdır; assembly'de onlarca satır sürebilir. Bedel, donanım üzerindeki denetimin azalmasıdır. Bir sürücü yazılımı belirli bir bellek adresine belirli bir anda yazmak zorundaysa C ya da assembly gibi alt basamaklardaki diller gerekir. Bu yüzden düzey bir kalite ölçüsü değildir; bir dilin soyutlama düzeyini anlatır. Yüksek düzeyli bir dil düşük düzeyli bir dilden "daha iyi" değildir, başka işlere daha uygundur.

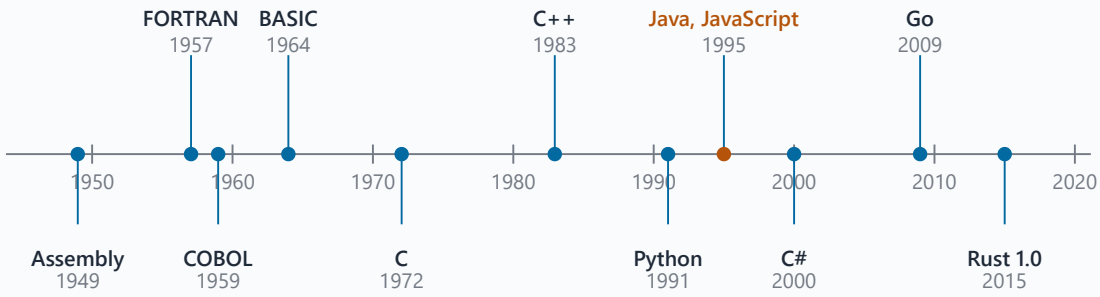
Örnek 4.3.7

`10110000 01100001` makine komutundaki ikinci bayt hangi sayıdır? Bu sayı assembly'de neden `61h` diye yazılır?

1. İkinci bayt `01100001`'dir. Basamak değerleriyle: *0·128 + 1·64 + 1·32 + 0·16 + 0·8 + 0·4 + 0·2 + 1·1 = 64 + 32 + 1 = 97*.
2. Bayt dört bitlik iki gruba ayrılır: `0110` = 6 ve `0001` = 1. Her dört bit tek bir onaltılı basamaktır; sayı onaltılı tabanda 61'dir. Denetim: *6 · 16 + 1 = 97*.
3. Sondaki h harfi sayının onaltılı (*hexadecimal*) olduğunu gösterir. Assembly programcıları onaltılı yazımı sever, çünkü her basamak tam dört bite karşılık gelir ve ikili koda dönüşüm kolaydır. 97, ASCII tablosunda küçük "a" harfinin kodudur.

4.3.3.2 Programlama dillerinin gelişimi

İlk bilgisayarlar doğrudan makine diliyle programlandı. Programcılar komutları ikili sayılar olarak yazıyor, bazen anahtarları elle tek tek çeviriyordu. 1940'ların sonunda assembly ortaya çıktı: komutlara akılda kalır adlar verildi ve bu adları ikiliye çeviren assembler programları yazıldı. Şekil 4.3.15 sonraki yetmiş yılı bir zaman çizelgesinde özetler.



Şekil 4.3.15: Programlama dillerinin zaman çizelgesi, 1949–2015. Her dilin altındaki yıl, ilk sürümünün ya da adının duyurulduğu yıldır.

Gelişim üç döneme ayrılabilir.

- **1940–1970.** Makine dili ve assembly dönemi. 1957'de IBM'de John Backus'un ekibi FORTRAN'ı geliştirdi. FORTRAN, bilimsel formülleri neredeyse matematikteki gibi yazmayı sağlayan ilk yaygın yüksek düzeyli dildir. 1959'da iş ve muhasebe uygulamaları için COBOL tasarlandı. 1964'te Dartmouth Koleji'nde öğrencilerin programlama öğrenmesi için BASIC geliştirildi.
- **1970–1990.** 1972'de Bell Laboratuvarları'nda Dennis Ritchie C dilini geliştirdi. UNIX işletim sistemi büyük ölçüde C ile yeniden yazıldı. C bir yandan donanıma yakın, öbür yandan farklı işlemcilere taşınabilir olduğu için sistem programlamanın temel dili oldu. 1983'te Bjarne Stroustrup, C'ye sınıf ve nesne kavramlarını ekleyen C++'ı duyurdu.
- **1990 sonrası.** 1991'de Python, 1995'te Java ve JavaScript, 2000'de C# ortaya çıktı. Bu dillerin ortak noktası, bellek yönetimini programcıdan alan yönetilen ortamlar ve web'e uygunluktur. Zengin kütüphaneleri ve açık kaynak ekosistemleri de yaygınlaşmalarında etkili oldu. Daha yakın dönemde Go (2009) eşzamanlı sunucu programlarına, Rust (1.0 sürümü 2015) ise güvenli sistem programlamaya odaklandı.

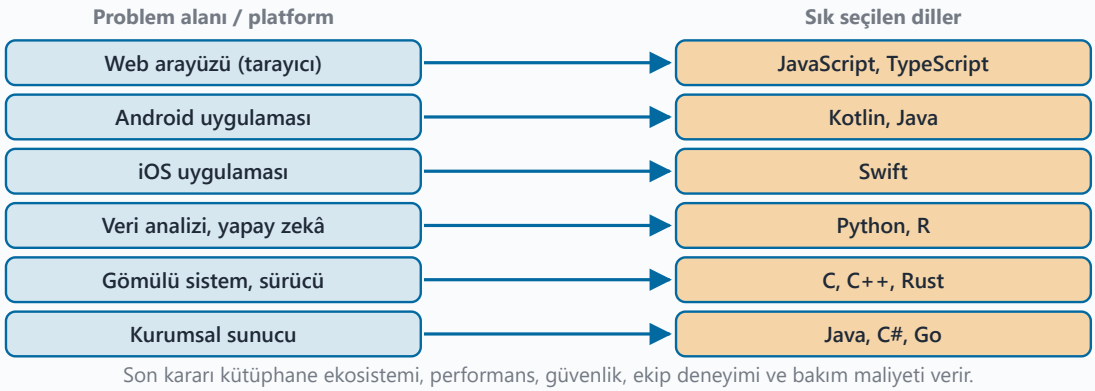
Bu tarihte bir eğilim göze çarpar: her yeni kuşak, öncekinin programcıya bıraktığı bir işi üstlendi. Assembly ikili kodlamayı, FORTRAN yazmaç yönetimini, Java bellek yönetimini programcının elinden aldı. Programcıya kalan iş, problemin kendisine biraz daha yaklaştı.

4.3.3.3 Kullanım alanları ve dil seçimi

Programlar çok farklı ortamlarda çalışır ve her ortamın ihtiyaçları başkadır. Başlıca kullanım alanları şunlardır:

- **Web:** tarayıcıda çalışan arayüzler (istemci), isteklere yanıt veren sunucu programları ve web servisleri.
- **Mobil:** telefon ve tablet uygulamaları.
- **Veri ve yapay zekâ:** veri analizi, istatistiksel modelleme, makine öğrenmesi ve otomasyon.
- **Sistem:** işletim sistemleri, aygıt sürücüleri ve bir ev aletinin içindeki gömülü yazılım.
- **Bilimsel hesaplama:** simülasyon ve mühendislik çözümleri; bir köprünün yük altındaki davranışı ya da bir devrenin frekans yanıtı.
- **Oyun:** oyun motorları, grafik ve kullanıcıyla etkileşim.

Her alanda bazı diller öne çıkar (Şekil 4.3.16). Tarayıcıda çalışan arayüzler için JavaScript (ve onun tipli bir uzantısı olan TypeScript) neredeyse tek seçenektir, çünkü tarayıcıların doğrudan çalıştırdığı dil budur. Android uygulamaları Kotlin ya da Java ile, iOS uygulamaları Swift ile yazılır; bu platformların resmî araçları bu dilleri destekler. Veri analizinde Python ve R, zengin kütüphaneleri sayesinde yaygındır. Gömülü sistemlerde bellek ve işlemci kısıtlı olduğundan C, C++ ve Rust tercih edilir. Kurumsal sunucularda Java, C# ve Go sık görülür.



Şekil 4.3.16: Problem alanına ve platforma göre sık seçilen diller. Son kararı kütüphane ekosistemi, performans, güvenlik, ekip deneyimi ve bakım maliyeti verir.

Dil seçerken önce problem alanı ve programın çalışacağı platform değerlendirilir. Sonra şu sorular sorulur. Gereken işi yapan hazır kütüphaneler var mı? Dilin topluluğu soruların yanıtını bulmaya yetecek kadar büyük mü? Program yıllarca bakım görecekse o dili bilen geliştirici bulmak kolay mı? Hız ve bellek kullanımı ne kadar önemli? Güvenlik açısından dilin bir zayıflığı var mı? Ekip hangi dili zaten biliyor? Program nasıl dağıtılacak?

Örnek 4.3.8

Bir elektrik-elektronik mühendisliği öğrenci grubu iki proje yapacak: (a) bir mikrodenetleyici üzerinde çalışıp sıcaklık sensörünü okuyan ve fanı açıp kapatan bir program, (b) sensörden bir hafta boyunca toplanan verinin grafiğini çizip ortalamasını ve en yüksek değerini bulan bir çözümleme. Hangi diller uygundur?

1. (a) Mikrodenetleyicinin belleği birkaç kilobayt olabilir, işletim sistemi bulunmayabilir. Programın donanım yazmaçlarına doğrudan erişmesi gerekir. Bu koşullar düşük düzeye yakın, derlenen bir dili gerektirir: C ya da C++. Rust da giderek yaygınlaşan bir seçenektir.
2. (b) Program bir dizüstü bilgisayarda bir kez çalışacak; hız ikinci plandadır. Grafik ve istatistik için hazır kütüphaneler ve hızlı deneme önemlidir. Python (ya da R) uygundur.
3. İki projede farklı dillerin seçilmesi bir çelişki değildir. Seçim, problemin ve platformun ihtiyaçlarından çıkar.

Sık yapılan hata

Bir dili yalnızca popüler olduğu için seçmek. En popüler dil bir mikrodenetleyiciye sığmayabilir ya da gereken kütüphane o dilde bulunmayabilir. Popülerlik, topluluk ve kaynak bolluğu açısından bir artıdır, ama tek başına doğru seçim için yeterli değildir.

4.3.3.4 Dil türleri

Diller kullanım amacına göre de gruplanır. Bir dil birden fazla gruba girebilir; Python hem genel amaçlı hem de bir betik dilidir.

Tür	Özellik	Örnek diller	Örnek satır
Genel amaçlı	Her tür problem için değişken, karar, döngü ve fonksiyon sunar	Java, C, Python	`for (int i = 0; i < 3; i++)`
Alana özgü	Tek bir işe odaklanır	SQL, Excel formülleri	`SELECT ad FROM ogrenci WHERE ort >= 60;`
Betik	Tekrarlanan işleri otomatikleştirir, genellikle yorumlanır	Python, JavaScript, Bash, PowerShell	`Copy-Item *.txt yedek\`
İşaretleme	İçeriğin yapısını tanımlar; karar ve döngü yoktur	HTML, XML, Markdown	`<h1>Başlık</h1>`

Tablo 4.3.6: Dil türleri ve örnekleri

Genel amaçlı diller (*general-purpose languages*) değişken, karar, döngü ve fonksiyon gibi yapıların tamamını sunar. Bu yapılarla, ilke olarak, bilgisayarın yapabileceği her hesap yazılabilir. Tablodaki `for` satırı Java'da üç tur dönen bir döngünün başıdır.

Alana özgü diller (*domain-specific languages*) tek bir işi çok iyi yapar. SQL veri tabanını sorgulamak için tasarlanmıştır. Tablodaki satır, öğrenci tablosundan ortalaması 60 ve üzeri olanların adlarını getirir. Satır, sonucun nasıl bulunacağını değil, ne istendiğini söyler; tabloyu nasıl tarayacağına veri tabanı yönetim sistemi karar verir. Excel'deki `=ORTALAMA(B2:B30)` gibi formüller de alana özgü bir dildir: hücrelerde hesap yapar.

Betik dilleri (*scripting languages*) tekrarlanan işleri otomatikleştirmek için kullanılır. Tablodaki PowerShell komutu, bulunduğu klasördeki bütün `.txt` dosyalarını `yedek` klasörüne kopyalar. Yüzlerce dosyayı elle kopyalamak yerine bu tek satır çalıştırılır. Betik dilleri genellikle yorumlanır; yazılan satır hemen denenebilir.

İşaretleme dilleri (*markup languages*) bir belgenin yapısını tanımlar. HTML'deki `<h1>Başlık</h1>` "bu metin birinci düzey başlıktır" der. İşaretleme dillerinde karar ve döngü yoktur; bir koşula göre farklı iş yapamaz ve bir işi tekrarlayamaz. Bu yüzden programlama dili sayılmazlar. Bir web sayfasında HTML yapıyı, CSS görünümü, JavaScript ise davranışı tanımlar; bu üçlüden yalnızca JavaScript bir programlama dilidir.

Bu gruplara blok tabanlı görsel diller de eklenebilir. Scratch gibi dillerde komutlar yazılmaz; renkli bloklar sürüklenip birbirine takılır. Bloklar yalnızca kurala uygun biçimde birleşebildiği için söz dizimi hatası yapılamaz. Bu özellik, programlamayı öğretmeye yeni başlanırken dikkati söz diziminden algoritmaya çevirir.

Sık yapılan hata

HTML'yi bir programlama dili saymak. HTML bir sayfadaki başlığı, paragrafı ve bağlantıyı tanımlar ama hiçbir hesap yapmaz: iki sayıyı toplayamaz, bir koşulu sınayamaz, bir işi tekrarlayamaz. Bir web sayfasının "programlanan" kısmı, onunla birlikte çalışan JavaScript kodudur.

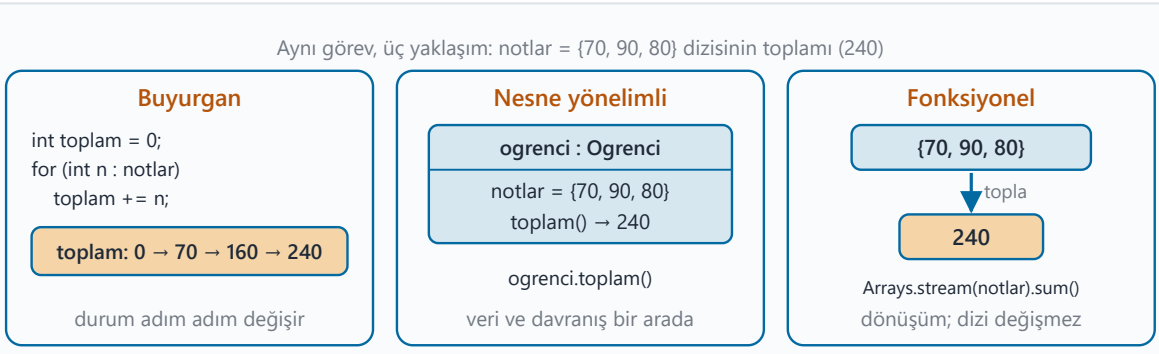
4.3.3.5 Programlama paradigmaları

Aynı problemi çözmenin birden çok düşünme biçimi vardır. Biri adımları sırayla sayar, biri problemi birbiriyle konuşan nesneler olarak görür, biri de veriyi dönüştüren fonksiyonlar zinciri kurar. Bu düşünme biçimlerine programlama paradigması denir.

Tanım 4.3.10

Programlama paradigması (*programming paradigm*), bir programın nasıl düzenleneceğine ilişkin temel yaklaşımdır. **Buyurgan yaklaşım** (*imperative programming*) işlemlerin nasıl yapılacağını adım adım belirtir ve programın durumunu bu adımlarla değiştirir. **Nesne yönelimli yaklaşım** (*object-oriented programming*) veri ile o veri üzerinde çalışan davranışı nesnelerde birleştirir. **Fonksiyonel yaklaşım** (*functional programming*) programı, değiştirilemez veri üzerinde çalışan fonksiyonların birleşimi olarak kurar.

Şekil 4.3.17 üç yaklaşımı aynı görev üzerinde karşılaştırır: notlar = {70, 90, 80} dizisinin toplamını bulmak. Aşağıdaki Java programı üç yaklaşımı da içerir.



Şekil 4.3.17: Aynı görev, üç yaklaşım. Buyurgan döngü durumu adım adım değiştirir; nesne yönelimli çözümde toplamı nesnenin kendi metodu hesaplar; fonksiyonel çözüm diziyi değiştirmeden bir dönüşümle toplar.

```java
import java.util.Arrays;

public class Paradigmalar {
    static class Ogrenci {
        private final int[] notlar;
        Ogrenci(int[] notlar) { this.notlar = notlar; }
        int toplam() {
            int t = 0;
            for (int n : notlar) t += n;
            return t;
        }
    }

    public static void main(String[] args) {
        int[] notlar = {70, 90, 80};
        // buyurgan
        int toplam = 0;
        for (int n : notlar)
            toplam += n;
        System.out.println(toplam);
        // nesne yönelimli
        Ogrenci ogrenci = new Ogrenci(notlar);
        System.out.println(ogrenci.toplam());
        // fonksiyonel
        System.out.println(Arrays.stream(notlar).sum());
        System.out.println(Arrays.toString(notlar));
    }
}
```

```
240
240
240
[70, 90, 80]
```

Buyurgan çözüm, toplam adlı bir değişkeni 0 ile başlatır ve döngünün her turunda bu değişkeni değiştirir. Toplamın değerleri sırasıyla 0, 70, 160 ve 240'tır. Program bir tarif gibi okunur: “önce şunu yap, sonra bunu yap.” C ve assembly temelde buyurgan dillerdir.

Nesne yönelimli çözümde notlar bir `Ogrenci` nesnesinin içindedir. Toplamı hesaplama işi de aynı nesneye aittir: `ogrenci.toplam()`. Nesneyi kullanan kod, toplamın nasıl hesaplandığını bilmek zorunda değildir; yalnızca nesneye sorar. Program büyüdükçe bu düzen işe yarar. Öğrenciyle ilgili her şey tek bir yerde durur, değişiklik gerektiğinde nereye bakılacağı bellidir. Java, C++ ve C# nesne yönelimli dillerdir.

Fonksiyonel çözümde `Arrays.stream(notlar).sum()` ifadesi diziyi bir değer akışına çevirir ve akışı toplar. Ara durum tutan bir değişken yoktur ve dizinin kendisi değişmez. Programın son satırı bunu doğrular: dizi hâlâ `[70, 90, 80]`'dir. Fonksiyonel yaklaşım, aynı girdiye her zaman aynı çıktıyı veren

ve dışarıdaki hiçbir şeyi değiştirmeyen fonksiyonları öne çıkarır. Böyle fonksiyonları sınamak ve birleştirmek kolaydır. Haskell tamamen fonksiyonel bir dildir; Python, JavaScript ve Java ise fonksiyonel öğeler de sunar.

Pek çok bugünkü dil birden fazla paradigmayı destekler. Java'da üç çözüm de yazılabilir; Python'da da öyle:

```
notlar = [70, 90, 80]
toplam = 0
for n in notlar:
    toplam += n
print(toplam)
print(sum(notlar))
from functools import reduce
print(reduce(lambda a, b: a + b, notlar, 0))
```

```
240
240
240
```

Python'daki `reduce` fonksiyonu, iki sayıyı toplayan küçük bir fonksiyonu dizinin öğelerine 0'dan başlayarak sırayla uygular: *((0 + 70) + 90) + 80 = 240*. Hangi yaklaşımın seçileceği problemin yapısına ve ekibin alışkanlığına bağlıdır. Büyük programlar çoğu zaman üç yaklaşımı birlikte kullanır.

Örnek 4.3.9

Bir dizideki sayıların en büyüğü bulunacak. Buyurgan ve fonksiyonel yaklaşım bu işi nasıl ifade eder? Dizi {12, 45, 7, 30} olsun.

1. Buyurgan: `eb` adlı bir değişken dizinin ilk öğesiyle (12) başlatılır. Her öğe `eb` ile karşılaştırılır; daha büyükse `eb` güncellenir. Durum adım adım değişir: 12 → 45 → 45 → 45. Sonuç 45'tir.
2. Fonksiyonel: Java'da `Arrays.stream(dizi).max().getAsInt()`, Python'da `max(dizi)`. İfade “dizinin en büyüğü” der; karşılaştırmaların sırası ve ara değişken programcıya görünmez. Dizi değişmez, sonuç yine 45'tir.
3. İki çözümde de bilgisayar aynı karşılaştırmaları yapar. Fark programın okunuşundadır: biri “nasıl”ı, öbürü “ne”yi öne çıkarır.

4.3.3.6 Statik ve dinamik tipleme

Her değerin bir tipi vardır: 5 bir tam sayı, `"beş"` bir dizgi, `true` bir mantıksal değerdir. Diller, tiplerin ne zaman denetlendiğine göre ikiye ayrılır.

Tanım 4.3.11

Statik tipleme (*static typing*), değişkenin tipinin kaynak kodda belli olduğu ve derleme sırasında denetlendiği tipleme biçimidir: Java, C, C#. **Dinamik tipleme** (*dynamic typing*), tipin değişkende değil değerde taşındığı ve çalışma sırasında denetlendiği biçimdir: Python, JavaScript.

Java'da bir değişken bildirilirken tipi de yazılır: `int x = 5;`. Bu andan sonra x yalnızca tam sayı tutabilir. `x = "beş";` satırı derleyicinin anlam çözümleme aşamasında yakalanır: "incompatible types: String cannot be converted to int". Program hiç çalışmadan hata bulunur.

Python'da değişkenin tipi bildirilmez. Aynı ad önce bir sayıyı, sonra bir metni gösterebilir. Tip, değerin kendisinde taşınır:

```
x = 5
print(type(x).__name__)
x = "beş"
print(type(x).__name__)
print("a" + 1)
```

```
int
str
Traceback (most recent call last):
  File "dinamik.py", line 5, in <module>
    print("a" + 1)
          ~~~~^~~
TypeError: can only concatenate str (not "int") to str
```

İlk iki satırın çıktısı x'in önce `int`, sonra `str` (dizgi) tipinde bir değeri gösterdiğini söyler. İkisi de geçerlidir. Beşinci satırdaki `"a" + 1` ise bir dizgiye sayı eklemeye çalışır ve `TypeError` verir. Önemli nokta hatanın ne zaman çıktığıdır. Python dosyayı ayrıştırırken bu satırda bir sorun görmez, çünkü satırın söz dizimi doğrudur. Hata ancak satır çalıştığında, önceki iki `print` satırının çıktısı ekrana yazıldıktan sonra ortaya çıkar. Bu satır programın nadiren çalışan bir dalında olsaydı, hata aylarca fark edilmeyebilirdi.

Bazı diller tip uyuşmazlığında hata vermek yerine tipleri sessizce dönüştürür. JavaScript bunun bilinen örneğidir. Aşağıdaki satırlar Node.js ile çalıştırılmıştır:

```
console.log("5" * 2);
console.log("5" + 2);
console.log(typeof ("5" * 2), typeof ("5" + 2));
console.log("5" - 2);
```

```
10
52
number string
3
```

`"5" * 2` ifadesinde çarpma yalnızca sayılarla yapılabildiği için JavaScript `"5"` dizgisini 5 sayısına çevirir ve 10 bulur. `"5" + 2` ifadesinde ise `+` dizgi birleştirme anlamına da geldiği için 2 dizgiye çevrilir ve `"52"` elde edilir. Üçüncü satır iki sonucun tiplerini gösterir: biri sayı, öbürü dizgidir. Aynı iki değer, işleme göre farklı tiplere dönüşmüştür. Bu tür sessiz dönüşümler kısa betiklerde kolaylık sağlar, büyük programlarda ise bulunması zor mantık hatalarına yol açar.

Özellik	Statik (Java)	Dinamik (Python)
Tip nerede durur?	Değişkende, bildirimde yazılır	Değerde, bildirim yok
Tip hatası ne zaman bulunur?	Derleme sırasında, program çalışmadan	O satır çalıştığında
Aynı ad farklı tipte değer tutabilir mi?	Hayır	Evet
Güçlü yön	Hatalar erken bulunur, IDE daha çok yardım eder	Kısa kod, hızlı deneme

Tablo 4.3.7: Statik ve dinamik tiplemenin karşılaştırması

> **Sık yapılan hata**
>
> Dinamik tiplemeyi “tip yok” sanmak. Python'da her değerin kesin bir tipi vardır ve Python tip kurallarını uygular: `"a" + 1` hata verir. Dinamik tiplemede eksik olan tip kuralları değil, bu kuralların program çalışmadan denetlenmesidir.

4.3.3.7 Sözde koddan dillere

Bir algoritma önce sözde kodla yazılıp sonra bir dile çevrildiğinde, sözde koddaki her yapının dilde bir karşılığı bulunur. Tablo 4.3.8 en sık kullanılan yapıların Java ve Python karşılıklarını verir.

Sözde kod	Java	Python
`oku x`	`x = giris.nextInt();`	`x = int(input())`
`yaz "Toplam: ", t`	`System.out.println("Toplam: " + t);`	`print("Toplam:", t)`
`x ← x + 1`	`x = x + 1;`	`x = x + 1`
`eğer x > 0 ise … değilse …`	`if (x > 0) { … } else { … }`	`if x > 0: … else: …`
`her i ← 1 … n için`	`for (int i = 1; i <= n; i++)`	`for i in range(1, n + 1):`
`koşul doğru iken tekrarla`	`while (kosul) { … }`	`while kosul:`

Tablo 4.3.8: Sözde kod yapılarının Java ve Python karşılıkları

Sözde kodda atama için `←` kullanılır, çünkü `=` işareti matematikte eşitlik anlamına gelir. Java ve Python atamayı `=` ile, eşitlik karşılaştırmasını `==` ile yazar. Java'daki `giris`, klavyeden okuyan bir `Scanner` nesnesidir. Python'da `input()` her zaman dizgi döndürür; sayı isteniyorsa `int(...)` ile çevrilmesi gerekir.

İki dil arasındaki en görünür fark blokların sınırlarıdır. Java bir bloğu süslü parantezlerle (`{ }`) belirler, her deyimi de `;` ile bitirir. Python bunların ikisini de istemez; bloğu girinti belirler. Bu yüzden Python'da girinti yalnızca görünüş değildir, programın anlamının parçasıdır:

```
x = -3
if x > 0:
    print("pozitif")
print("bitti")
if x > 0:
    print("pozitif")
    print("bitti")
```

```
bitti
```

x = −3 olduğu için iki koşul da yanlıştır. İlk `if` bloğunda `print("bitti")` girintisizdir, yani bloğun dışındadır ve koşuldan bağımsız çalışır. İkinci blokta aynı satır girintilidir, bloğun içindedir ve koşul yanlış olduğu için çalışmaz. İki parça arasındaki tek fark dört boşluktur; çıktıda yalnızca bir “bitti” görünür.

Örnek 4.3.10

1'den n'ye kadar olan sayıların toplamını bulan sözde kod, n = 5 için Java ve Python'a nasıl çevrilir?

```
toplam ← 0
her i ← 1 … n için
    toplam ← toplam + i
yaz "Toplam: ", toplam
```

Java karşılığı (n = 5 sabit alınmıştır):

```java
public class Bilesik {
    public static void main(String[] args) {
        int toplam = 0;
        for (int i = 1; i <= 5; i++) {
            toplam = toplam + i;
        }
        System.out.println("Toplam: " + toplam);
    }
}
```

Python karşılığı:

```python
n = 5
toplam = 0
for i in range(1, n + 1):
    toplam = toplam + i
print("Toplam:", toplam)
i = 1
while i <= 3:
    print(i)
    i = i + 1
print("bitti")
```

Python dosyası ayrıca bir `while` döngüsüyle 1, 2, 3 yazar. Çıktılar:

```
Toplam: 15
```

```
Toplam: 15
1
2
3
bitti
```

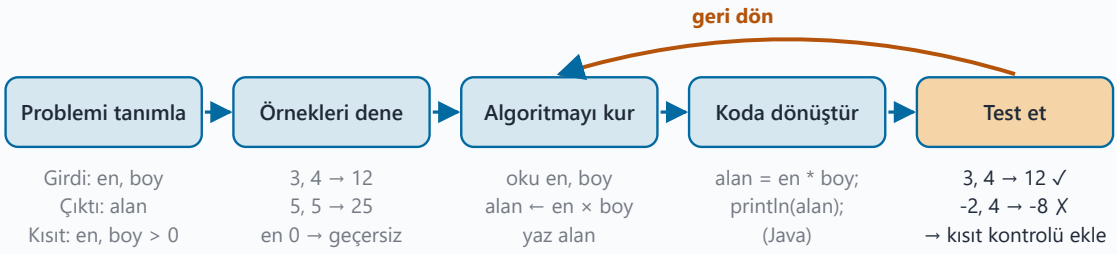
Toplam her turda şöyle değişir: *0 + 1 = 1*, *1 + 2 = 3*, *3 + 3 = 6*, *6 + 4 = 10*, *10 + 5 = 15*. Python'daki `range(1, n + 1)` ifadesi 1'den başlar ve n + 1'e ulaşmadan durur; yani 1, 2, 3, 4, 5 değerlerini üretir. `range(1, n)` yazılsaydı 5 atlanır ve toplam 10 çıkardı.

4.3.4 Problem çözme

Program yazmak klavyenin başında başlamaz. Önce problemin ne olduğu anlaşılır, sonra elle birkaç örnek çözülür, en son kod yazılır. Bu sıra atlandığında ortaya çoğu zaman "çalışan ama yanlış işi yapan" programlar çıkar.

4.3.4.1 Problemden programa

Örnek problem basittir: bir dikdörtgenin alanını hesaplayan bir program. Şekil 4.3.18 bu problemin programa dönüşmesini beş aşamada gösterir. Aşamaların her birinin çıktısı bir sonrakinin girdisidir.



Şekil 4.3.18: Problemden programa beş aşama. Testte negatif girdi yanlış sonuç verince (turuncu ok) algoritmaya dönülür ve bir kısıt denetimi eklenir.

1. **Problemi tanımla.** Ne istendiği netleşir: dikdörtgenin alanı. Programın **girdisi** en ve boy, **çıktısı** alandır. Bir de **kısıt** vardır: bir uzunluk sıfır ya da negatif olamaz, yani en ve boy 0'dan büyük olmalıdır. Kısıtlar çoğu zaman problemde açıkça yazılmaz; problemi tanımlayan kişinin onları bulması gerekir.
2. **Örnekleri dene.** Elle birkaç örnek çözülür: *3 × 4 = 12*, *5 × 5 = 25*. Bir de sınır durumu düşünülür: en 0 olursa ne olmalı? Kısıta göre bu girdi geçersizdir. Elle çözülen örnekler, sonradan programın doğru çalışıp çalışmadığını sınamak için beklenen sonuçları da verir.
3. **Algoritmayı kur.** Dilden bağımsız adımlar sözde kodla yazılır: en'i oku, boy'u oku, alanı hesapla, alanı yaz.
4. **Koda dönüştür.** Algoritma seçilen dile çevrilir. Java'da hesap satırı `alan = en * boy;`, yazdırma satırı `System.out.println(alan);` olur.
5. **Test et.** Program elle çözülen örneklerle çalıştırılır. 3 ve 4 için 12 çıkar: doğru. −2 ve 4 için −8 çıkar: bir alan negatif olamayacağına göre bu yanlıştır. Hata kodda değil, algoritmadadır; algoritma kısıtı hiç denetlemiyordur. Algoritmaya dönülür ve "en ve boy 0'dan büyük değilse hata ver" adımı eklenir.

Son aşamadaki geri dönüş sürecin doğal bir parçasıdır. Aşamalar düz bir çizgi değil, bir döngü oluşturur: test, kodlamada ya da algoritmada bir eksik bulursa ilgili aşamaya dönülür. Düzeltilmiş algoritma şudur:

```
oku en
oku boy
eğer en > 0 ve boy > 0 ise
    alan ← en × boy
    yaz "Alan: ", alan
değilse
    yaz "Geçersiz ölçü"
```

Bu algoritmanın Java karşılığı aşağıdadır. `&&` işareti Java'da "ve" bağlacıdır: iki koşul da doğruysa sonuç doğrudur.

```
import java.util.Scanner;

public class Alan {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int en = giris.nextInt();
        int boy = giris.nextInt();
        if (en > 0 && boy > 0) {
            int alan = en * boy;
            System.out.println("Alan: " + alan);
        } else {
            System.out.println("Geçersiz ölçü");
        }
    }
}
```

Program dört girdiyle çalıştırılmıştır: (3, 4), (−2, 4), (0, 5) ve (5, 2). Çıktılar sırasıyla şunlardır:

```
Alan: 12
```

```
Geçersiz ölçü
```

```
Geçersiz ölçü
```

```
Alan: 10
```

Sık yapılan hata

Problemi tanımlamadan kod yazmaya başlamak. Yanlış tanımlanan bir problem, doğru çalışan ama yanlış işi yapan bir programa dönüşür. “Not ortalaması hesaplayan program” isteği birkaç soruyu yanıtsız bırakır. Notlar ağırlıklı mı? Kaç not girilecek? Geçersiz not (−5, 120) girilirse ne olacak? Bu sorular sorulmadan yazılan program, ilk gerçek kullanımda değiştirilmek zorunda kalır.

4.3.4.2 Alan algoritmasını izleme

Bir algoritmanın doğru çalıştığından emin olmanın en güvenilir yolu, onu kâğıt üzerinde satır satır izlemektir. Her adımda değişkenlerin değeri bir tabloya yazılır. Bu tabloya **izleme tablosu** (*trace table*) denir. Değişkenler bellekte birer kutu gibi düşünülebilir: `oku en` satırı klavyeden gelen sayıyı “en” adlı kutuya yazar. Henüz değer verilmemiş kutular tabloda “—” ile gösterilir.

Tablo 4.3.9 algoritmanın en = 3, boy = 4 girdisiyle çalışmasını izler.

Adım	Satır	en	boy	alan	Açıklama / çıktı
1	`oku en`	3	—	—	3 okunur ve en kutusuna yazılır
2	`oku boy`	3	4	—	4 okunur
3	`eğer en > 0 ve boy > 0`	3	4	—	3 > 0 ve 4 > 0 → doğru; “ise” dalına girilir
4	`alan ← en × boy`	3	4	12	*3 × 4 = 12*
5	`yaz "Alan: ", alan`	3	4	12	Çıktı: `Alan: 12` ; “değilse” dalı atlanır

Tablo 4.3.9: Alan algoritmasının izlenmesi, 1. çalıştırma (en = 3, boy = 4)

Tablo 4.3.10 aynı algoritmayı en = −2, boy = 4 girdisiyle izler. Bu, test aşamasında yanlış sonuç veren girdidir.

Adım	Satır	en	boy	alan	Açıklama / çıktı
1	`oku en`	−2	—	—	−2 okunur
2	`oku boy`	−2	4	—	4 okunur
3	`eğer en > 0 ve boy > 0`	−2	4	—	−2 > 0 yanlış; “ve” ile bağlı koşulun tamamı yanlış → “değilse” dalı
4	`yaz "Geçersiz ölçü"`	−2	4	—	Çıktı: `Geçersiz ölçü` ; alan hiç hesaplanmadı

Tablo 4.3.10: Alan algoritmasının izlenmesi, 2. çalıştırma (en = −2, boy = 4)

İkinci izlemede alan kutusu baştan sona boş kalır. Program “ise” dalına hiç girmemiştir. Testte bulunan hata böylece düzeltilmiştir: −2 için artık −8 değil, bir uyarı yazılır.

Örnek 4.3.11

Alan algoritmasına çevre hesabı eklenmek isteniyor: *çevre = 2 × (en + boy)*. Algoritma nasıl değişir? en = 6, boy = 1,5 için çıktılar nedir?

1. "ise" dalına iki satır eklenir: `cevre ← 2 × (en + boy)` ve `yaz "Çevre: ", cevre`. Kısıt denetimi aynı kalır; geçersiz ölçüde çevre de hesaplanmaz.
2. Koşul: 6 > 0 ve 1,5 > 0 → doğru.
3. alan = *6 × 1,5 = 9*.
4. cevre: önce parantez, *6 + 1,5 = 7,5*; sonra *2 × 7,5 = 15*.
5. Çıktı: "Alan: 9" ve "Çevre: 15". Parantez yazılmasaydı çarpma önce yapılır ve *2 × 6 + 1,5 = 13,5* bulunurdu.

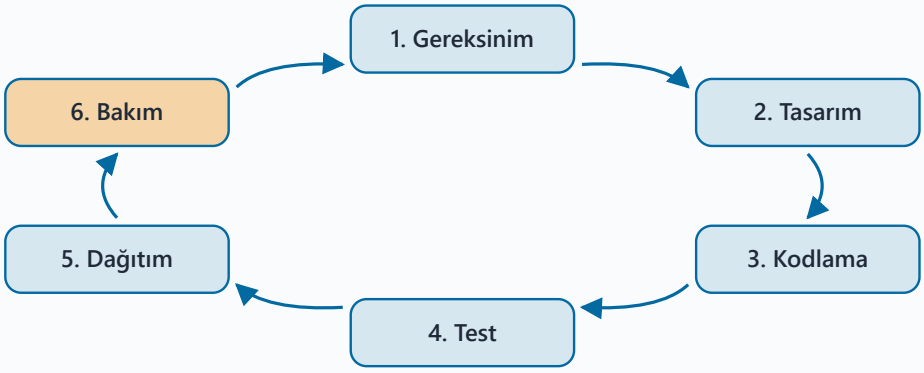
Bu girdide en ve boy kesirli olabildiği için Java karşılığında int yerine double kullanılması gerekir.

4.3.5 Yazılım geliştirme

Dikdörtgen alanı gibi küçük bir program tek kişinin bir saatlik işidir. Bir bankanın mobil uygulaması ya da bir hastanenin randevu sistemi ise onlarca kişinin aylarca, hatta yıllarca süren işidir. Bu ölçekte işin düzenlenmesi en az kodun kendisi kadar önemlidir.

4.3.5.1 Yazılım geliştirme süreci

Problemden programa giden yol büyük ölçekte altı aşamaya ayrılır (Şekil 4.3.19). Aşamalar bir "öğrenci not ortalamasını hesaplayan uygulama" üzerinden şöyle açıklanabilir.



Şekil 4.3.19: Yazılım geliştirme süreci. Gereksinimden bakıma altı aşama; testte bulunan hata kodlamaya, bakımda gelen yeni istekler gereksinime geri götürür.

1. **Gereksinim.** Kullanıcı ne istiyor? Öğrenci her dersin notunu ve kredisini girecek, uygulama ağırlıklı ortalamayı gösterecek. Not 0–100 arasında olmalı. Uygulama telefonda çalışacak. Gereksinimler kullanıcıyla konuşularak yazılır ve sonraki aşamaların hepsi bu yazıya dayanır.
2. **Tasarım.** Ekranlar, veri ve algoritmalar planlanır. Hangi ekranda ne girilecek? Notlar nerede saklanacak? Ortalama hangi formülle hesaplanacak? Akış diyagramı ve sözde kod bu aşamanın araçlarıdır.
3. **Kodlama.** Tasarım bir programlama diliyle yazılır.
4. **Test.** Beklenen sonuçlar gerçek sonuçlarla karşılaştırılır. 60 puanlı 3 kredilik ve 90 puanlı 2 kredilik iki ders için beklenen ortalama *(60 · 3 + 90 · 2) / 5 = 360 / 5 = 72*'dir. Uygulama 75 gösteriyorsa (iki notun ağırlıksız ortalaması) bir hata vardır ve kodlamaya geri dönülür.
5. **Dağıtım.** Program kullanıcıya ulaştırılır: uygulama mağazasında yayımlanır ya da kurulum dosyası paylaşılır.
6. **Bakım.** Kullanım başladıktan sonra yeni hatalar bulunur, yeni istekler gelir: "harf notunu da göstersin". Bu istekler yeni gereksinimlerdir ve döngü yeniden başlar.

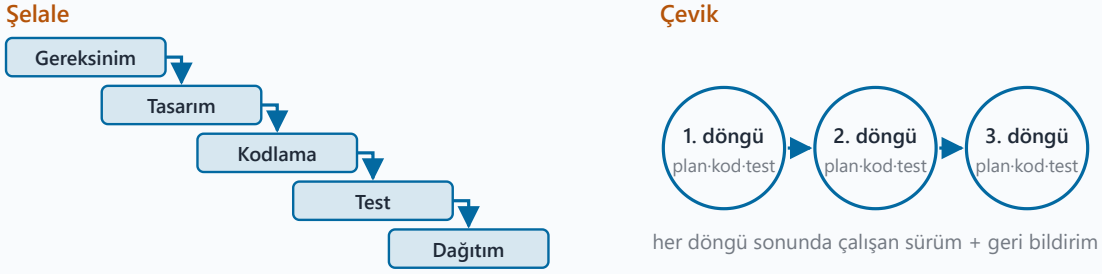
Aşamalar projeye göre ardışık ya da yinelemeli yürütülür. Kullanıcıdan gelen geri bildirim gereksinimi ve tasarımı değiştirebilir. Bir yazılımın ömrü boyunca harcanan emeğin büyük kısmı ilk sürüme değil, bakıma gider. Bu yüzden kodun ilk günden okunabilir ve değiştirilebilir yazılması önemlidir.

Sık yapılan hata

Test aşamasını "programın çalıştığını görmek" sanmak. Testin amacı programın doğru çalıştığını kanıtlamak değil, yanlış çalıştığı durumları bulmaktır. İyi bir test, sonucu önceden elle hesaplanmış normal girdileri, sınır durumlarını (0, boş liste, en küçük ve en büyük değer) ve geçersiz girdileri içerir.

4.3.5.2 Geliştirme yaklaşımları

Altı aşamanın nasıl düzenleneceğine dair farklı yaklaşımlar vardır (Şekil 4.3.20).



Şekil 4.3.20: Şelale modeli aşamaları basamak basamak, bir kez geçer. Çevik yaklaşım kısa döngülerle ilerler; her döngünün sonunda çalışan bir sürüm ve kullanıcı geri bildirimi vardır.

Şelale modelinde (*waterfall model*) aşamalar sırayla ve birer kez yapılır. Gereksinimler tamamen yazılmadan tasarıma, tasarım bitmeden kodlamaya geçilmez. Her aşamanın sonunda bir belge üretilir. Bu düzenin güçlü yanı açıklığıdır: kimin ne zaman ne yapacağı bellidir. Zayıf yanı değişikliğe kapalı olmasıdır. Kodlama sırasında gereksinimde bir eksik fark edilirse geri dönmenin maliyeti yüksektir.

Çevik yaklaşımda (*agile*) iş kısa döngülere bölünür; her döngü birkaç haftalıktır. Her döngüde biraz planlama, biraz kodlama ve test yapılır. Döngü sonunda kullanıcıya çalışan bir sürüm gösterilir ve geri bildirim alınır. Böylece yanlış anlaşılan bir gereksinim aylar sonra değil, birkaç hafta içinde ortaya çıkar. Bu yaklaşımın bedeli, kullanıcıyla sürekli iletişim kurma gereğidir.

Prototiplemede (*prototyping*) gerçek ürüne başlamadan önce hızlıca bir taslak hazırlanır. Taslakta ekranlar vardır ama arkalarında gerçek hesap ya da veri tabanı bulunmayabilir. Kullanıcı ne istediğini tam anlatamıyorsa, taslağı gördüğünde fikri netleşir. Taslağın amacı gereksinimi görünür kılmaktır. Ürünün kendisi sanılıp doğrudan kullanıma sunulmamalıdır.

Yaklaşım	Güçlü yön	Dikkat
Şelale modeli	Aşamalar ve belgeler nettir	Değişikliğe dönüş maliyetlidir
Çevik	Kısa döngülerle geri bildirim alır	Sürekli iletişim gerektirir
Prototipleme	Belirsiz gereksinimi görünür kılar	Prototip ürün sanılmamalıdır

Tablo 4.3.11: Geliştirme yaklaşımlarının karşılaştırması

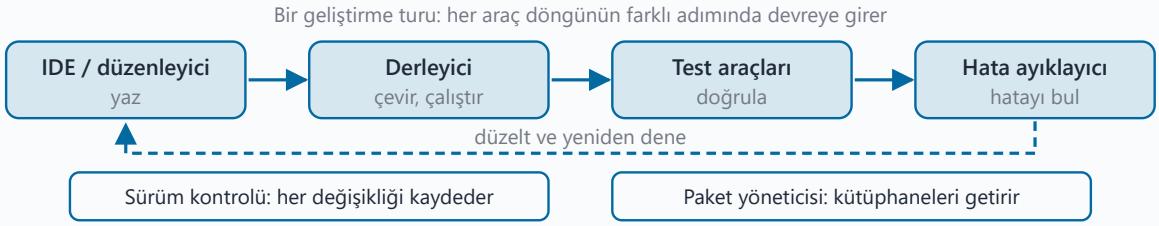
Örnek 4.3.12

Üç proje için hangi yaklaşım uygundur? (a) Bir uçağın iniş takımını denetleyen yazılım; gereksinimler uluslararası standartlarla önceden kesin olarak belirlenmiş. (b) Bir kafenin sipariş uygulaması; sahibi "ekranları görünce karar veririm" diyor. (c) Bir öğrenci kulübünün etkinlik uygulaması; özellikler her dönem üyelerin isteğiyle değişiyor.

1. (a) Şelale modeli. Gereksinimler sabittir ve her aşamanın belgelenmesi denetim için zorunludur. Değişiklik beklenmediği için modelin zayıf yanı burada sorun olmaz.
2. (b) Prototipleme. Kullanıcı ne istediğini ancak somut ekranları görünce söyleyebilir. Hızlı bir taslak bu belirsizliği giderir.
3. (c) Çevik yaklaşım. İstekler sürekli değişir; kısa döngüler ve her döngü sonunda alınan geri bildirim buna uyar.

4.3.5.3 Geliştirme araçları

Bir geliştiricinin günlük işi bir döngüdür: kodu yaz, çevir ve çalıştır, sonucu sına, hata varsa bul, düzelt ve yeniden dene. Bu döngünün her adımında farklı bir araç devreye girer (Şekil 4.3.21).



Şekil 4.3.21: Bir geliştirme turu. Her araç döngünün farklı bir adımında devreye girer; sürüm kontrolü ve paket yöneticisi bütün döngüyü destekler.

- **Kod düzenleyici ve IDE.** Kod bir metin düzenleyicide yazılabilir, ama programcılar çoğunlukla bir **tümleşik geliştirme ortamı** (IDE, *integrated development environment*) kullanır. IDE yazılan kodu renklendirir, adları tamamlar, söz dizimi hatalarını yazarken altını çizerek gösterir ve projenin dosyalarını düzenler. IntelliJ IDEA, Eclipse ve Visual Studio Code yaygın örneklerdir.
- **Derleyici ve çalışma ortamı.** Programı çevirir ve çalıştırır: javac ve JVM, Python yorumlayıcısı, C derleyicisi.
- **Hata ayıklayıcı** (*debugger*). Programı istenen satırda durdurur, değişkenlerin o anki değerlerini gösterir ve satır satır ilerletir.
- **Sürüm kontrolü.** Dosyalardaki her değişikliği kaydeder; ekip çalışmasını mümkün kılar.
- **Test araçları.** Beklenen davranışı otomatik olarak doğrular. Programcı bir kez "bu girdiye bu çıktı beklenir" diye yazar, araç her değişiklikten sonra bütün testleri saniyeler içinde yeniden çalıştırır. Java'da JUnit bu işin bilinen aracıdır.
- **Paket yöneticisi** (*package manager*). Programın kullandığı hazır kütüphaneleri (bağımlılıkları) indirir, sürümlerini izler ve günceller: Python'da pip, JavaScript'te npm, Java'da Maven ve Gradle.

Sık yapılan hata

IDE'nin altını çizmediği kodu hatasız sanmak. IDE söz dizimi ve tip hatalarını yazarken gösterebilir, çünkü derleyicinin ilk aşamalarını arka planda çalıştırır. Çalışma zamanı ve mantık hatalarını ise göremez. Kırmızı çizgi olmaması yalnızca derlemenin başarılı olacağını söyler.

4.3.5.4 Sürüm kontrolü

Bir öğrenci ödevini yazarken dosyanın kopyalarını `odev_son.java`, `odev_son2.java`, `odev_GERCEKSON.java` diye saklıyorsa, sürüm kontrolünü elle yapmaya çalışıyordur. Bu yöntem birkaç gün içinde karışır. Hangi kopyada hangi değişikliğin yapıldığı unutulur; iki kişi aynı dosyada çalışıyorsa iş içinden çıkılmaz hâle gelir.

Tanım 4.3.12

Sürüm kontrolü (*version control*), dosyalardaki değişikliklerin geçmişini saklayan ve birden çok kişinin aynı dosyalar üzerinde çalışmasını düzenleyen sistemdir. **Kayıt** (*commit*), dosyaların belirli bir andaki hâlinin kimin, ne zaman ve neyi değiştirdiği bilgisiyle birlikte saklanmasıdır. **Dal** (*branch*), ana koddan ayrılıp bağımsız olarak ilerleyen bir değişiklik çizgisidir. **Birleştirme** (*merge*), bir dalın değişikliklerinin başka bir dala aktarılmasıdır.



Şekil 4.3.22: Ana dalda dört kayıt. İkinci kayıttan ayrılan deneme dalı (D1) yeni bir hesabı dener ve dördüncü kayıtta ana dala birleştirilir.

Şekil 4.3.22 dikdörtgen programının geçmişini gösterir. K1 ilk sürümdür. K2'de alan hesabı, K3'te çevre hesabı eklenmiştir. K2'den sonra bir geliştirici yeni bir hesap denemek için bir dal açmış (D1) ve ana kodu bozmadan orada çalışmıştır. Deneme başarılı olunca dal ana dala birleştirilmiştir (K4). Deneme başarısız olsaydı dal silinir, ana dal hiç etkilenmezdi. K3'teki bir değişiklik hata getirdiyse proje tek komutla K2'deki hâline döndürülebilir.

En yaygın sürüm kontrol aracı Git'tir. Git'te yukarıdaki geçmiş birkaç komutla oluşur. `git init` bir klasörü depo hâline getirir. `git add` değişen dosyaları bir sonraki kayda hazırlar. `git commit -m "alan eklendi"` kaydı açıklamasıyla oluşturur. `git branch deneme` yeni bir dal açar; `git merge deneme` o dalı bulunulan dala birleştirir. `git log` ise bütün kayıtları yazarı, tarihi ve açıklamasıyla birlikte listeler.

Ekip çalışmasında her geliştiricinin bilgisayarında deponun bir kopyası bulunur. GitHub ve GitLab gibi hizmetler depoyu ağ üzerinde saklar ve kopyalar arasında ortak bir buluşma noktası olur. İki kişi aynı dosyanın farklı satırlarını değiştirdiyse Git değişiklikleri kendiliğinden birleştirir. Aynı satırı değiştirdilerse bir **çakışma** (*merge conflict*) oluşur. Git iki sürümü yan yana gösterir ve hangisinin kalacağını bir kişinin seçmesini ister.

Örnek 4.3.13

Ayşe ve Mehmet aynı depodan çalışıyor. Programın 12. satırı `double kdv = fiyat * 0.18;`. Ayşe bu satırı `fiyat * 0.20` yapıp kaydediyor. Mehmet de aynı satırı `fiyat * KDV_ORANI` yapıyor ve dosyanın başına sabiti ekliyor. İkisi değişikliklerini birleştirmek istediğinde ne olur?

1. Dosyanın başına eklenen sabit satırı yalnızca Mehmet'in değişikliğidir; Git bunu sorunsuz alır.
2. 12. satırı ikisi de değiştirmiştir. Git hangisinin doğru olduğunu bilemez ve çakışma bildirir. Satırın iki sürümü de dosyada işaretlenerek gösterilir.
3. İki geliştirici konuşur ve satırın son hâline karar verir: `fiyat * KDV_ORANI`, sabitin değeri de 0.20. Seçilen hâl kaydedilir ve birleştirme tamamlanır.

Sık yapılan hata

Sürüm kontrolünü yalnızca büyük ekiplerin aracı sanmak. Tek başına çalışan bir öğrenci de geçmişe dönme imkânından yararlanır. Bir değişiklik programı bozduğunda, en son çalışan sürüme dönmek bir komut sürer. Kayıtların açıklamaları da kodun neden değiştiğini sonradan hatırlamayı sağlar.

4.3.6 Temel programlama yapıları

Diller birbirinden ne kadar farklı görünürse görünsün, hepsi aynı birkaç yapı taşı üzerine kurulur. Değerler adlandırılmış yerlerde saklanır ve ifadelerle hesaplanır. Koşullara göre farklı yollar seçilir, işler tekrarlanır. Sık yapılan işlere ad verilir, çok sayıda değer tek bir ad altında toplanır. Yapılar önce sözde kodla, sonra Java ve Python karşılıklarıyla verilir.

4.3.6.1 Değişkenler ve veri tipleri

Tanım 4.3.13

Değişken (*variable*), bellekte bir değeri tutan adlandırılmış bir yerdir; değeri program boyunca değişebilir. **Veri tipi** (*data type*), bir değerin bellekte nasıl temsil edileceğini ve üzerinde hangi işlemlerin yapılabileceğini belirler.

Değişken, üzerinde adı yazılı bir kutu gibi düşünülebilir (Şekil 4.3.23). `yas` kutusunda 19, `boy` kutusunda 1.75, `ad` kutusunda "Ali", `gecti` kutusunda doğru değeri vardır. Kutuya yeni bir değer yazıldığında eskisi silinir. Kutunun tipi, içine ne konabileceğini sınırlar: tam sayı kutusuna metin konamaz.



Şekil 4.3.23: Bellekte adlandırılmış kutular. Her kutunun bir adı, bir değeri ve bir tipi vardır.

Tip	Ne tutar?	Örnek değer	Java	Yapılabilen işlemler
Tam sayı	Kesirsiz sayı	19, −3, 0	`int`	Dört işlem, kalan (mod)
Kayan noktalı sayı	Kesirli sayı	1.75, 81.67	`double`	Dört işlem
Dizgi	Karakter dizisi, metin	"Ali"	`String`	Birleştirme, uzunluk, karakter alma
Mantıksal	Doğru ya da yanlış	true, false	`boolean`	ve, veya, değil

Tablo 4.3.12: Temel veri tipleri ve Java karşılıkları

Kodda ondalık ayırıcı noktadır: Java ve Python `1,75` yazımını sayı olarak kabul etmez. Bu notta düz metinde Türkçe yazım kuralına uyularak virgül (1,75), kod ve program çıktılarında nokta (1.75) kullanılır.

Veri tipi seçimi değerin anlamından çıkar. Bir öğrencinin adı harflerden oluştuğu için dizgidir. Not ortalaması 81,67 gibi kesirli olabileceği için kayan noktalı sayıdır. “Dersi geçti mi?” sorusunun yanıtı yalnızca evet ya da hayır olabileceği için mantıksaldır. Öğrenci sayısı kesirli olamayacağı için tam sayıdır.

Değişkenlere anlamlı adlar vermek programın amacını görünür kılar. `a = b * c` satırı hiçbir şey anlatmaz; `alan = en * boy` satırı kendini açıklar. Adlar programın okunabilirliğinin en ucuz ve en etkili aracıdır.

Sık yapılan hata

Kesirli olabilecek bir değeri tam sayı tipinde tutmak. Java'da `int ort = 245 / 3;` satırı hata vermez; 81 değerini saklar ve kesirli kısım sessizce kaybolur. Ortalama, oran, ölçü gibi değerler için `double` kullanılır. Bölme işleminin işlenenlerinden en az biri de kesirli olmalıdır: `245 / 3.0`.

4.3.6.2 İşleç önceliği

Bir ifadede birden çok işlem varsa hangisinin önce yapılacağını **işleç önceliği** (*operator precedence*) belirler. Programlama dillerindeki kurallar okulda öğrenilen matematik kurallarıyla aynıdır:

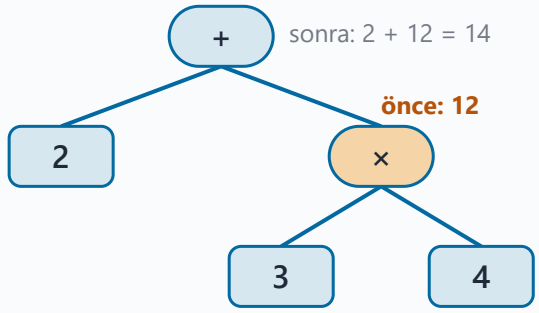
1. Parantez içi önce hesaplanır.
2. Çarpma, bölme ve mod (kalan) toplamadan ve çıkarmadan önce yapılır.
3. Toplama ve çıkarma en son yapılır.
4. Aynı düzeydeki işlemler soldan sağa yapılır.

Mod işlemi (Java ve Python'da `%`) bölmeden kalanı verir: 7 mod 3, 7'nin 3'e bölümünden kalan olan 1'dir. Tablo 4.3.13 bu kuralların beş ifadeye uygulanmasını gösterir.

İfade	Hesaplama sırası	Sonuç
2 + 3 × 4	Önce *3 × 4 = 12*, sonra *2 + 12*	14
(2 + 3) × 4	Önce parantez *2 + 3 = 5*, sonra *5 × 4*	20
20 / 4 × 2	Aynı düzey, soldan sağa: *20 / 4 = 5*, sonra *5 × 2*	10
10 − 4 − 3	Soldan sağa: *10 − 4 = 6*, sonra *6 − 3*	3
7 mod 3 + 1	Önce *7 mod 3 = 1*, sonra *1 + 1*	2

Tablo 4.3.13: İşleç önceliği örnekleri

Derleyici işleç önceliğini ifadenin ağacına yerleştirir. Şekil 4.3.24 *2 + 3 × 4* ifadesinin ağacını gösterir: çarpma daha derinde olduğu için önce yapılır, toplama kökte olduğu için en son yapılır.



Şekil 4.3.24: *2 + 3 × 4* ifadesinin ağacı. Çarpma daha derinde olduğu için önce yapılır (12), sonra kökteki toplama (14).

Tablodaki ifadeler Java'da da aynı sonuçları verir. Aşağıdaki program tablonun beş ifadesini ve soldan sağa kuralını sınayan iki ek ifadeyi yazdırır:

```
public class Oncelik {
    public static void main(String[] args) {
        System.out.println(2 + 3 * 4);
        System.out.println((2 + 3) * 4);
        System.out.println(20 / 4 * 2);
        System.out.println(10 - 4 - 3);
        System.out.println(7 % 3 + 1);
        System.out.println(20 / (4 * 2));
        System.out.println(10 - (4 - 3));
    }
}
```

```
14
20
10
3
2
2
9
```

Son iki satır soldan sağa kuralının neden önemli olduğunu gösterir. *20 / 4 × 2* ifadesi 10 verir; parantezle *20 / (4 × 2)* yazılınca sonuç 2'dir. Java'da bu ikinci sonuç aslında 2,5 olmalıydı: iki tam sayının bölümü tam sayı olduğu için *20 / 8* işleminin kesirli kısmı atılır. Benzer biçimde *10 − 4 − 3 = 3* iken *10 − (4 − 3) = 9*'dur.

Python'da aynı ifadeler bir fark dışında aynı sonuçları verir:

```
185.0
81.67
52.33
14 20 10.0 3 2
10
```

Çıktının dördüncü satırında `20 / 4 * 2` ifadesi `10.0` verir, çünkü Python'da `/` işleci her zaman kayan noktalı sonuç üretir. Tam sayı bölmesi için `//` işleci vardır ve son satırdaki gibi 10 verir. İlk satırdaki 185.0 ise eksik parantezin sonucudur. Bu hata Python'da da aynen ortaya çıkar.

Örnek 4.3.14

Bir öğrencinin vize notu 60, final notu 80; vizenin ağırlığı %40, finalin %60. Programcı `ort ← vize × 0,4 + final × 0,6` yazıyor. Sonuç doğru mudur? Aynı hesap `ort ← vize + final × 0,6 / 2` diye yazılsaydı ne çıkardı?

1. İlk ifadede iki çarpma önce yapılır: *60 × 0,4 = 24* ve *80 × 0,6 = 48*. Sonra toplama: *24 + 48 = 72*. Parantez gerekmez, çünkü çarpma zaten önce yapılır. Sonuç doğrudur.
2. İkinci ifadede önce çarpma ve bölme, soldan sağa: *80 × 0,6 = 48*, *48 / 2 = 24*. Sonra toplama: *60 + 24 = 84*. Program hata vermez ama 84 yazar; bu bir mantık hatasıdır.

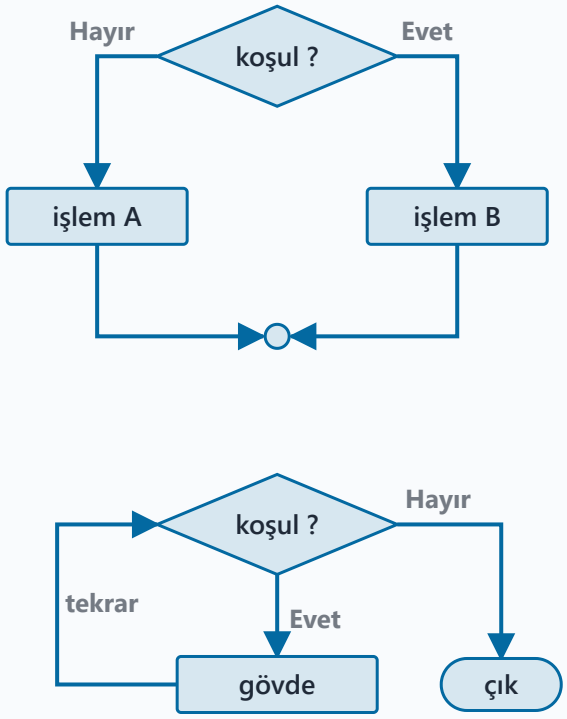
Sık yapılan hata

Ortalama formülünde parantezi unutmak: `ort ← a + b / 2`. Bu ifade önce b'yi ikiye böler, sonra a'yı ekler. a = 70, b = 90 için sonuç *70 + 45 = 115* çıkar; doğrusu *(70 + 90) / 2 = 80*'dir. Program hata vermez. Bu yüzden böyle hatalar yalnızca test edilerek bulunur. Önceliğinden emin olunmayan her yerde parantez kullanmak hem hatayı önler hem de ifadeyi okuyana ni-

yeti açıkça gösterir.

4.3.6.3 Karar ve döngü

Bir program yalnızca yukarıdan aşağı sırayla çalışsaydı, her girdi için aynı adımları yapardı. Gerçek problemler ise koşullara göre farklı davranmayı ve aynı işi tekrar tekrar yapmayı gerektirir. Bunu sağlayan iki yapı karar ve döngüdür (Şekil 4.3.25).



Şekil 4.3.25: Solda karar yapısı: koşula göre iki yoldan yalnızca biri izlenir. Sağda döngü: koşul doğru oldukça gövde tekrarlanır; geri dönen ok döngünün işaretidir.

Tanım 4.3.14

Seçim ya da karar yapısı (*selection*), bir koşulun doğru ya da yanlış olmasına göre farklı işlem yollarından birini seçer. **Döngü** (*loop*), bir işlem grubunu bir koşul doğru olduğu sürece tekrarlar. Tekrarlanan işlem grubuna döngünün **gövdesi** (*body*), gövdenin her çalışmasına bir **tur** (*iteration*) denir.

Karar yapısında her çalıştırmada iki yoldan yalnızca biri izlenir. Alan algoritmasında koşul doğruysa alan hesaplanır, yanlışsa uyarı yazılır; ikisi aynı çalıştırmada birlikte olmaz. Döngüde ise akış, gövdenin sonunda koşula geri döner. Koşul her turun başında yeniden denetlenir. Koşul yanlış olunca döngüden çıkılır ve döngüden sonraki satıra geçilir.

Bir döngünün bitebilmesi için gövdenin içinde koşulu etkileyen bir değişiklik olmalıdır. Koşul `i ≤ 3` ise gövde i'yi bir gün 3'ün üstüne çıkarmalıdır. Böyle bir değişiklik yoksa koşul hep doğru kalır ve döngü hiç bitmez.

4.3.6.4 Döngü izleme: sonsuz döngü

Aşağıdaki sözde kod 1'den 3'e kadar sayıları yazmak için yazılmıştır. Programcı ilk sürümde 4. satırı unutmuştur:

```
i ← 1
i ≤ 3 iken tekrarla
    yaz i
    i ← i + 1      // ilk sürümde yoktu
yaz "bitti"
```

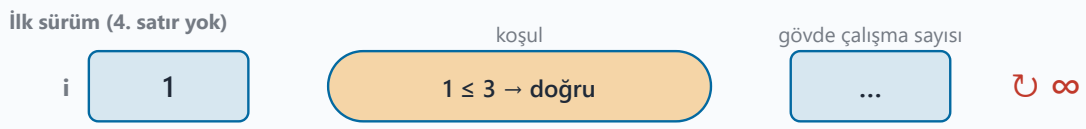
Tanım 4.3.15

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

Tablo 4.3.14 ilk sürümün ilk birkaç adımını izler. i ← 1 ile döngü değişkeni hazırlanır. Koşul *1 ≤ 3* doğrudur, gövdeye girilir ve 1 yazılır. Gövdede başka satır olmadığı için koşula dönülür. i hâlâ 1'dir, koşul yine doğrudur. Durum bir önceki turla birebir aynıdır.

Adım	Satır	i	Koşul	Gövde çalışma sayısı	Çıktı
1	`i ← 1`	1	—	0	
2	`i ≤ 3 iken`	1	1 ≤ 3 → doğru	0	
3	`yaz i`	1	—	1	1
4	`i ≤ 3 iken`	1	1 ≤ 3 → doğru	1	
5	`yaz i`	1	—	2	1
…	…	1	hep doğru	sınırsız artar	1, 1, 1, …

Tablo 4.3.14: İlk sürümün izlenmesi (4. satır yok)



Şekil 4.3.26: Sonsuz döngü. i hep 1'dir, koşul hep doğrudur ve gövde sınırsız sayıda çalışır.

Koşulu yanlış yapacak bir değişiklik olmadığı için döngü hiç bitmez (Şekil 4.3.26). Ekran 1'lerle dolar. Konsolda çalışan bir program ancak Ctrl+C tuşlarıyla durdurulabilir.

Düzeltilmiş sürümde gövdeye `i ← i + 1` satırı eklenir. Tablo 4.3.15 bu sürümü baştan sona izler.

Adım	Koşul denetimi	Gövde	i (turdan sonra)	Gövde çalışma sayısı
Başlangıç	—	`i ← 1`	1	0
1. tur	1 ≤ 3 → doğru	1 yazılır, i ← 2	2	1
2. tur	2 ≤ 3 → doğru	2 yazılır, i ← 3	3	2
3. tur	3 ≤ 3 → doğru (eşitlik)	3 yazılır, i ← 4	4	3
Çıkış	4 ≤ 3 → yanlış	Döngüden çıkılır; “bitti” yazılır	4	3

Tablo 4.3.15: Düzeltilmiş sürümün izlenmesi

Döngü değişkeni artık her turda ilerler. Üçüncü turda koşul *3 ≤ 3* yine doğrudur, çünkü ≤ eşitliği de kapsar. Bu ayrıntı atlanırsa sayıların yalnızca 1 ve 2'ye kadar yazılacağı sanılabilir. Döngü bittiğinde gövde 3 kez çalışmış, koşul ise 4 kez denetlenmiştir: üç doğru ve bir yanlış. Bir döngüde koşul her zaman gövdeden bir fazla denetlenir. Son denetim döngüyü bitiren denetimdir. Java karşılığı ve çıktısı:

```java
public class Dongu {
    public static void main(String[] args) {
        int i = 1;
        while (i <= 3) {
            System.out.println(i);
            i = i + 1;
        }
        System.out.println("bitti");
    }
}
```

```
1
2
3
bitti
```

Örnek 4.3.15

Aşağıdaki döngü kaç tur döner, ne yazar? `k ← 1 ; k < 20 iken tekrarla: yaz k; k ← k × 3` . Gövdedeki `k ← k × 3` satırı unutulsaydı ne olurdu?

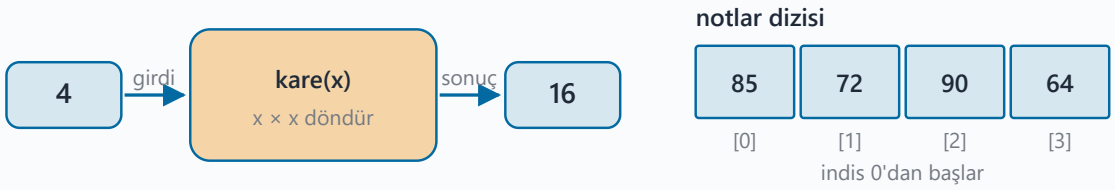
1. 1. tur: *1 < 20* doğru; 1 yazılır, k = 3.
2. 2. tur: *3 < 20* doğru; 3 yazılır, k = 9.
3. 3. tur: *9 < 20* doğru; 9 yazılır, k = 27.
4. *27 < 20* yanlış; döngü biter. Üç tur döner ve ekrana 1, 3, 9 yazılır; koşul 4 kez denetlenir.
5. Gövdedeki çarpma satırı unutulsaydı k hep 1 kalırdı. Koşul hep doğru olacağı için döngü sonsuza kadar 1 yazardı.

Sık yapılan hata

Döngü değişkenini yanlış yönde değiştirmek. `i ← 1; i ≤ 3 iken tekrarla: yaz i; i ← i − 1` döngüsünde gövde i'yi değiştirir, ama koşulu yanlış yapacak yönde değil. i 1, 0, −1, −2 diye azalır ve hep 3'ten küçük kalır. Gövdede bir değişiklik olması yetmez; değişiklik koşulu bir gün yanlış yapmalıdır.

4.3.6.5 Fonksiyonlar ve diziler

Programlar büyüdükçe iki ihtiyaç ortaya çıkar. Birincisi, aynı işin programın birçok yerinde yapılmasıdır; dikdörtgen alanı belki on yerde hesaplanır. İkincisi, çok sayıda benzer değerin saklanmasıdır; bir sınıfın 40 öğrencisinin notları için 40 ayrı değişken tanımlamak anlamsızdır. Fonksiyon birinci ihtiyacı, dizi ikincisini karşılar (Şekil 4.3.27).



Şekil 4.3.27: Solda kare fonksiyonu: 4 girdisini alır, *4 × 4 = 16* döndürür. Sağda notlar dizisi: dört hücre, indisler 0'dan başlar.

Tanım 4.3.16

Fonksiyon (*function*), belirli bir görevi yapan ve bir adla çağrılan kod bloğudur. Girdi olarak **parametre** (*parameter*) alabilir ve bir sonuç **döndürebilir** (*return*). Fonksiyonu çağırırken verilen değerlere **argüman** (*argument*) denir. **Dizi** (*array*), aynı tipteki çok sayıda değeri tek bir ad altında, sıra numarasıyla saklayan yapıdır. Bir öğenin sıra numarasına **indis** (*index*) denir.

Fonksiyon bir görevi adlandırır. `kare(x)` fonksiyonu "bir sayının karesini al" görevinin adıdır. Fonksiyonu kullanan kod, karenin nasıl hesaplandığıyla ilgilenmez; yalnızca `kare(4)` yazar ve 16 alır. Formül tek bir yerde durur. Formülde bir hata varsa bir kez düzeltilir ve düzeltme fonksiyonun çağrıldığı her yere yansır.

Dizide değerler sıra numaralarıyla seçilir. Java, C ve Python'da sıra numarası 0'dan başlar. Dört öğeli `notlar` dizisinde ilk öğe `notlar[0]` = 85, son öğe `notlar[3]` = 64'tür. Son indis her zaman uzunluk − 1'dir. Bu kural ilk bakışta tuhaf gelse de bir mantığı vardır: indis, öğenin dizinin başından kaç adım uzakta olduğunu söyler; ilk öğe başlangıçtan sıfır adım uzaktadır.

Bu iki yapı programı küçük, ayrı ayrı sınanabilir parçalara böler. Not ortalaması hesaplayan bir program, notları okuyan, ortalamayı hesaplayan ve harf notunu bulan üç fonksiyona ayrılabilir; notlar da tek bir dizide tutulur.

4.3.6.6 Fonksiyon çağrısını izleme

Aşağıdaki sözde kod, alan fonksiyonunu tanımlar ve iki kez çağırır:

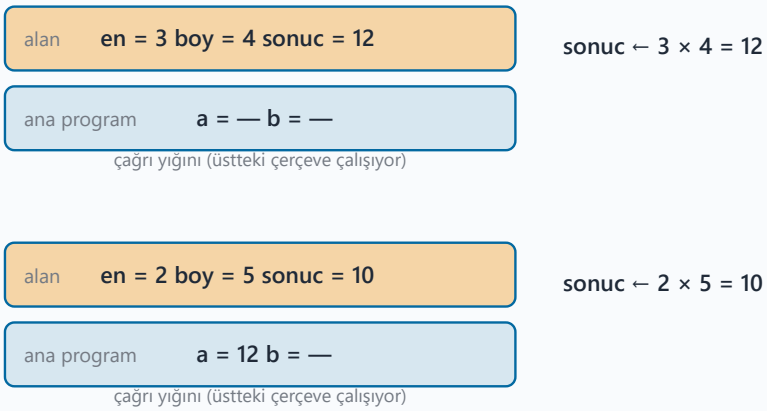
```
fonksiyon alan(en, boy)
    sonuc ← en × boy
    döndür sonuc
son
a ← alan(3, 4)
b ← alan(2, 5)
yaz a + b
```

Fonksiyon çağrıları bellekte **çağrı yığını** (*call stack*) adı verilen bir yapıyla izlenir. Her çağrı için yığının üstüne yeni bir **çerçeve** (*stack frame*) eklenir. Çerçevede fonksiyonun parametreleri ve kendi değişkenleri durur. Fonksiyon bitince çerçevesi yığından kaldırılır. Yığının en üstündeki çerçeve, o anda çalışan fonksiyonundur. Tablo 4.3.16 programı adım adım izler.

Adım	Olay	Ana program	alan çerçevesi
1	Tanım (satır 1–4) kaydedilir, çalışmaz; program 5. satırdan başlar	a = —, b = —	yok
2	`alan(3, 4)` çağrılır; yeni çerçeve, 3 → en, 4 → boy	a = —, b = —	en = 3, boy = 4
3	Gövde: *sonuc ← 3 × 4 = 12*	a = —, b = —	en = 3, boy = 4, sonuc = 12

Adım	Olay	Ana program	alan çerçevesi
4	`döndür sonuc` : 12 çağıran yere gönderilir	a = —, b = —	kaldırılacak
5	Çerçeve kalkar; dönen değer a'ya atanır	a = 12, b = —	yok
6	`alan(2, 5)` çağrılır; yepyeni bir çerçeve	a = 12, b = —	en = 2, boy = 5
7	Gövde: *sonuc ← 2 × 5 = 10*	a = 12, b = —	en = 2, boy = 5, sonuc = 10
8	10 döndürülür, çerçeve kapanır	a = 12, b = —	kaldırılacak
9	Dönen değer b'ye atanır	a = 12, b = 10	yok
10	`yaz a + b` → 22	a = 12, b = 10	yok

Tablo 4.3.16: Fonksiyon çağrısının izlenmesi



Şekil 4.3.28: Çağrı yığınının iki anı. Üstte birinci çağrı çalışıyor: çerçevede en = 3, boy = 4, sonuc = 12. Altta ikinci çağrı: aynı fonksiyon için yeni bir çerçeve, en = 2, boy = 5, sonuc = 10.

İzlemeden üç sonuç çıkar. Birincisi, fonksiyon tanımı tek başına hiçbir şey yapmaz; ancak çağrıldığında çalışır. İkincisi, her çağrı kendi çerçevesini açar. Birinci çağrının sonuc = 12 değeri ikinci çağrının çerçevesinde görünmez, çünkü birinci çerçeve çoktan kaldırılmıştır. Fonksiyonun içindeki değişkenler yalnızca o çağrı sürdükçe yaşar. Üçüncüsü, ana program fonksiyonun içini görmez; yalnızca döndürülen değeri alır. Bir kez yazılan fonksiyon iki kez kullanılmış, alan formülü tek bir yerde kalmıştır (Şekil 4.3.28).

Java karşılığında fonksiyonlara metot denir. `static int alan(int en, int boy)` başlığı, metodun iki tam sayı aldığını ve bir tam sayı döndürdüğünü söyler:

```java
public class AlanFonksiyon {
    static int alan(int en, int boy) {
        int sonuc = en * boy;
        return sonuc;
    }

    public static void main(String[] args) {
        int a = alan(3, 4);
        int b = alan(2, 5);
        System.out.println(a + b);
    }
}
```

```
22
```

Örnek 4.3.16

`fonksiyon cevre(en, boy): döndür 2 × (en + boy)` tanımlı. Ana program `c ← cevre(3, 4) + cevre(1, 1)` satırını çalıştırıyor. Kaç çerçeve açılır, c kaç olur?

1. Önce `cevre(3, 4)` çağrılır: çerçeve açılır, en = 3, boy = 4. *2 × (3 + 4) = 14* döndürülür, çerçeve kapanır.
2. Sonra `cevre(1, 1)` çağrılır: yeni bir çerçeve, en = 1, boy = 1. *2 × (1 + 1) = 4* döndürülür, çerçeve kapanır.
3. İki dönen değer toplanır: *c = 14 + 4 = 18*. Toplam iki çerçeve açılmıştır; ikisi aynı anda yığında bulunmaz.

Sık yapılan hata

Fonksiyonun içinde tanımlanan bir değişkeni fonksiyonun dışında kullanmaya çalışmak. alan fonksiyonundaki `sonuc` değişkeni yalnızca fonksiyonun çerçevesinde vardır; ana programda `yaz sonuc` yazmak Java'da “cannot find symbol” derleme hatası verir. Fonksiyonun ürettiği değer dışarıya yalnızca `döndür` ile çıkar.

4.3.6.7 Dizi ile ortalama

Aşağıdaki sözde kod dört notun ortalamasını bir dizi ve bir döngüyle hesaplar:

```
notlar ← [85, 72, 90, 64]
toplam ← 0
her i ← 0, 1, 2, 3 için
    toplam ← toplam + notlar[i]
ort ← toplam / 4
yaz "Ortalama: ", ort
```

Toplamı tutan değişkene **biriktirici** denir. Biriktirici döngüden önce 0 ile başlatılır, her turda bir değer daha eklenir. Başlangıç değeri verilmeseydi toplamın ilk değeri belirsiz olurdu. Tablo 4.3.17 döngünün her turunu izler.

Adım	i	notlar[i]	İşlem	toplam
Başlangıç	—	—	`toplam ← 0`	0
1. tur	0	85	*0 + 85*	85
2. tur	1	72	*85 + 72*	157
3. tur	2	90	*157 + 90*	247
4. tur	3	64	*247 + 64*	311
Döngüden sonra	—	—	*ort ← 311 / 4*	311 (ort = 77,75)

Tablo 4.3.17: Notların ortalamasının izlenmesi (notlar = [85, 72, 90, 64])



Şekil 4.3.29: Döngü bittikten sonra dizi ve biriktirici: toplam 311, ortalama 77,75.

Döngü dört tur sonra biter. Son indis 3'tür, yani uzunluk − 1. Sonuç “Ortalama: 77.75” olarak yazılır (Şekil 4.3.29). Java programında döngü sınırı sabit 4 yerine dizinin uzunluğu olan `notlar.length` ile yazılmıştır. Böylece diziye not eklenince döngü kendiliğinden uyum sağlar. Bölme de `4.0` ile yapılır ki sonuç tam sayıya yuvarlanmasın:

```java
public class Ortalama {
    public static void main(String[] args) {
        int[] notlar = {85, 72, 90, 64};
        int toplam = 0;
        for (int i = 0; i < notlar.length; i++) {
            toplam = toplam + notlar[i];
        }
        double ort = toplam / 4.0;
        System.out.println("Ortalama: " + ort);
    }
}
```

```
Ortalama: 77.75
```

Python'da aynı hesap daha kısadır. `sum` ve `len` hazır fonksiyonları döngüyü programcıdan gizler:

```python
notlar = [85, 72, 90, 64]
toplam = 0
for i in range(4):
    toplam = toplam + notlar[i]
ort = toplam / 4
print("Ortalama:", ort)
print(sum(notlar) / len(notlar))
```

```
Ortalama: 77.75
77.75
```

Döngü bir tur fazla dönseydi ne olurdu? Dört öğeli dizide 4 numaralı hücre yoktur. Java bu durumu bir çalışma zamanı hatasıyla bildirir:

```java
public class DiziDisi {
    public static void main(String[] args) {
        int[] notlar = {85, 72, 90, 64};
        System.out.println(notlar[3]);
        System.out.println(notlar[4]);
    }
}
```

```
64
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 4 out of bounds
for length 4
        at DiziDisi.main(DiziDisi.java:5)
```

`notlar[3]` sorunsuz okunur ve 64 yazılır. `notlar[4]` istendiğinde JVM programı durdurur. İletideki "Index 4 out of bounds for length 4" ifadesi "uzunluğu 4 olan dizide 4 numaralı indis sınırların dışında" demektir. Döngü sınırının `i <= notlar.length` diye yazılması bu hatanın en sık nedenidir; doğrusu `i < notlar.length`'tir.

Örnek 4.3.17

sıcaklık = [18, 21, 25, 19, 22] dizisinde 20'nin üzerindeki günler sayılacak. Bir sayaç ve bir döngüyle izleme nasıl yapılır? Kaç gün bulunur?

1. `sayac ← 0` . i = 0: sıcaklık[0] = 18; *18 > 20* yanlış, sayaç 0.
2. i = 1: 21 > 20 doğru, sayaç 1.
3. i = 2: 25 > 20 doğru, sayaç 2.
4. i = 3: 19 > 20 yanlış, sayaç 2.
5. i = 4: 22 > 20 doğru, sayaç 3. Son indis *5 − 1 = 4*'tür; döngü biter. Sonuç 3 gündür.

Bu örnekte döngünün içinde bir karar yapısı vardır. Döngü her öğeyi ziyaret eder, karar ise hangi öğenin sayılacağını seçer.

4.3.6.8 Kütüphaneler ve API

Karekök almak, bir listeyi sıralamak ya da bir dosyayı okumak her programcının sık ihtiyaç duyduğu işlerdir. Her programcının bunları sıfırdan yazması hem zaman kaybı hem de hata kaynağı olurdu. Bu yüzden diller hazır, test edilmiş fonksiyonlarla birlikte gelir.

Tanım 4.3.17

Kütüphane (*library*), başkalarının yazıp test ettiği hazır fonksiyonların topluluğudur. **API** (*application programming interface*, uygulama programlama arayüzü), bir kütüphanenin ya da bir hizmetin dışarıya sunduğu fonksiyonları, bu fonksiyonların girdilerini ve dönüş değerlerini tanımlar.



Şekil 4.3.30: Program kütüphane fonksiyonunu API üzerinden çağırır: karekök(16) çağrısı 4.0 döndürür. Kütüphanenin içi görünmez.

API bir sözleşme gibidir (Şekil 4.3.30). Karekök fonksiyonunun API'si üç şey söyler: fonksiyonun adı, bir sayı aldığı ve o sayının karekökünü döndürdüğü. Fonksiyonun içinde hangi yöntemin kullanıldığı, kayan nokta ayrıntılarının nasıl ele alındığı kullanıcıyı ilgilendirmez. Kütüphaneyi yazanlar içini daha hızlı bir yöntemle değiştirse bile, API aynı kaldıkça onu kullanan programlar değişmeden çalışmaya devam eder.

Çağrı	Dil	Sonuç
`Math.sqrt(16)`	Java	4.0
`Math.max(3, 8)`	Java	8
`len("Giresun")`	Python	7

Tablo 4.3.18: Kütüphane çağrısı örnekleri

Tablodaki sonuçlar gerçek çalıştırmalardan alınmıştır. Java'da `Math.sqrt` sonucu her zaman `double` döndürür; bu yüzden 4 değil, 4.0 yazılır. Java'da bir dizginin uzunluğu bir fonksiyonla değil, dizginin kendi metoduyla bulunur: `"Giresun".length()`.

```
public class Kutuphane {
    public static void main(String[] args) {
        System.out.println(Math.sqrt(16));
        System.out.println(Math.max(3, 8));
        System.out.println("Giresun".length());
    }
}
```

```
4.0
8
7
```

```
import math
print(math.sqrt(16))
print(max(3, 8))
print(len("Giresun"))
```

```
4.0
8
7
```

API kavramı kütüphanelerle sınırlı değildir. Bir hava durumu uygulaması, bir meteoroloji hizmetinin ağ üzerinden sunduğu API'yi çağırır: şehir adını gönderir, sıcaklığı alır. Uygulamayı yazanlar o hizmetin sunucularında ne olduğunu bilmez, yalnızca API'nin belgesini okur.

Sık yapılan hata

Bir kütüphane fonksiyonunun dönüş tipini bilmeden kullanmak. Java'da `Math.sqrt(16)` 4.0 döndürür; sonucu `int k = Math.sqrt(16);` diye bir tam sayıya atamak derleme hatası verir, çünkü double değer int'e kendiliğinden dönüştürülmez. API belgesi her fonksiyonun ne aldığını ve ne döndürdüğünü yazar; kullanmadan önce bakmak gerekir.

4.3.7 Hata ayıklama

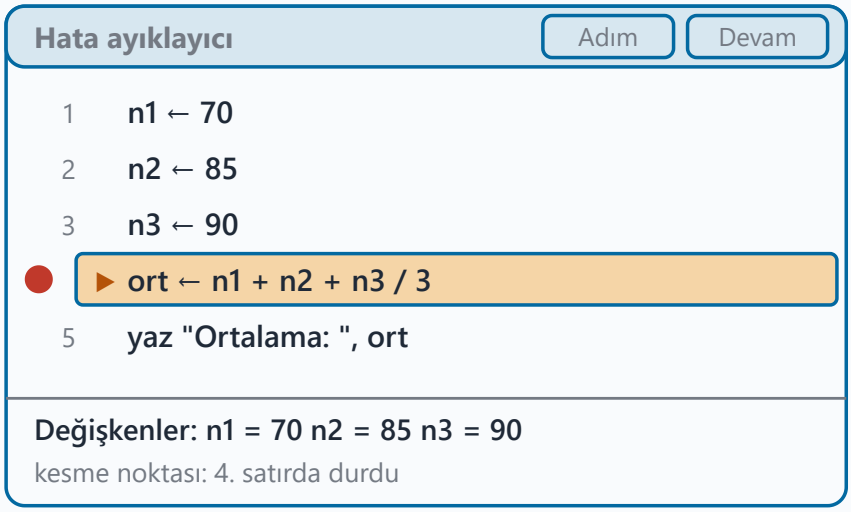
Her programcı hata yapar; deneyimli programcıyı ayıran, hatayı bulmanın düzenli bir yolunu bilmesidir. Bir programdaki hatayı bulup giderme işine **hata ayıklama** (*debugging*) denir.

4.3.7.1 Hata ayıklama teknikleri

Hata ayıklama bir tahmin oyunu değil, adım adım yürüyen bir araştırmadır. Aşağıdaki teknikler bu araştırmanın araçlarıdır.

1. **Hatayı yeniden üret.** Aynı girdiyle hata her seferinde çıkıyor mu? Hatayı her istendiğinde ortaya çıkaran girdi bulunmadan aramaya başlamak boşa çabadır. Bu girdi, düzeltmeden sonra hatanın gerçekten gittiğini sınamak için de gerekir.
2. **Ara değerleri yazdır.** Şüpheli satırdan önce ve sonra değişkenlerin değerleri ekrana yazdırılır. Beklenen ile gerçek değerin ilk ayrıldığı yer, hatanın yeridir.
3. **Kesme noktası koy.** Hata ayıklayıcıda bir satıra **kesme noktası** (*breakpoint*) konur. Program o satıra gelince durur ve o andaki bütün değişkenler incelenir.
4. **Adım adım ilerle.** Program durduktan sonra satır satır ilerletilir. Her satırdan sonra hangi değerin değiştiğine bakılır.

5. **Alanı daralt.** Kodun yarısı geçici olarak devre dışı bırakılır ve hatanın hangi yarıda olduğu bulunur. Sonra o yarı yeniden ikiye bölünür. Yüz satırlık bir kodda hata yedi bölmede tek satıra iner, çünkü $2^7 = 128 \geq 100$.
6. **Anlatarak düşün.** Kodu satır satır birine, hatta masadaki bir oyuncağa anlatmak çoğu zaman hatayı gösterir. Anlatırken "bu satır şunu yapıyor" denir ve bir anda satırın aslında başka bir şey yaptığı fark edilir.



Şekil 4.3.31: Hata ayıklayıcı penceresi. Program 4. satırdaki kesme noktasında durmuştur; değişkenler panelinde n1 = 70, n2 = 85, n3 = 90 görünür.

Şekil 4.3.31 bir hata ayıklayıcının görünümünü taklit eder. Program 4. satırdaki kesme noktasında durmuştur ve henüz o satırı çalıştırmamıştır. Değişkenler panelinde üç not görünür. Satır `ort ← n1 + n2 + n3 / 3`'tür. "Adım" düğmesine basıldığında önce *n3 / 3 = 30* hesaplanır, sonra *70 + 85 + 30 = 185*. Ortalama 185 çıkar; beklenen değer ise *245 / 3 ≈ 81,67*'dir. Hata ayıklayıcı, sorunun toplama değil bölme sırasında olduğunu bir bakışta gösterir.

> **Sık yapılan hata**
>
> Hatanın nedenini bulmadan belirtisini gizlemek. Ortalama 185 çıkıyor diye sonucu 2,27'ye bölmek bu girdide 81,5 civarı bir değer verir ama başka her girdide yanlış sonuç üretir. Düzeltme, hatanın temel nedenine yapılır: eksik parantez eklenir.

4.3.7.2 İzleyerek hata bulma

Aşağıdaki sözde kod üç sayının en büyüğünü bulmak için yazılmıştır. eb değişkeni "en büyük" değerini tutar:

```
oku a, b, c
eb ← 0
eğer a > eb ise eb ← a
eğer b > eb ise eb ← b
eğer c > eb ise eb ← c
yaz "En büyük: ", eb
```

İlk test için 7, 12 ve 5 seçilir; beklenen çıktı 12'dir. Tablo 4.3.19 programı izler.

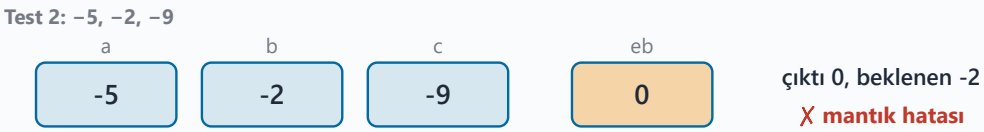
Satır	Koşul	eb
`eb ← 0`	—	0
`eğer a > eb`	7 > 0 → doğru	7
`eğer b > eb`	12 > 7 → doğru	12
`eğer c > eb`	5 > 12 → yanlış	12
`yaz`	—	Çıktı: `En büyük: 12` ✓

Tablo 4.3.19: Test 1 (a = 7, b = 12, c = 5; beklenen 12)

Çıktı beklenenle aynıdır. Ama tek bir başarılı test programın doğru olduğunu kanıtlamaz. İkinci test bir sınır durumunu dener: sayıların hepsi negatif olsun, −5, −2 ve −9. Beklenen çıktı −2'dir (Tablo 4.3.20).

Satır	Koşul	eb
`eb ← 0`	—	0
`eğer a > eb`	−5 > 0 → yanlış	0
`eğer b > eb`	−2 > 0 → yanlış	0
`eğer c > eb`	−9 > 0 → yanlış	0
`yaz`	—	Çıktı: `En büyük: 0` ✗

Tablo 4.3.20: Test 2 (a = −5, b = −2, c = −9; beklenen −2)



Şekil 4.3.32: Test 2'nin sonu. eb hiç değişmemiştir; çıktı 0, beklenen −2: mantık hatası.

Üç karşılaştırmanın üçü de yanlıştır, çünkü hiçbir girdi 0'dan büyük değildir; eb hiç değişmez ve çıktı 0 olur (Şekil 4.3.32). Girdilerin hiçbiri 0 değildir. İzleme suçluyu gösterir: 2. satırdaki başlangıç değeri girdilerin dışından gelir. 0, "hiçbir sayıdan küçük olmayan" bir başlangıç değeri değildir; yalnızca pozitif sayılar için işe yarar.

Düzeltme, başlangıç değerini girdilerin birinden almaktır: `eb ← a`. eb artık bir girdiyle başladığı için sonuç her zaman girdilerden biri olur. a'nın kendisiyle karşılaştırılmasına da gerek kalmaz. Düzeltilmiş program Test 2 ile yeniden izlenir: eb = −5 ile başlar; *−2 > −5* doğru, eb = −2; *−9 > −2* yanlış. Çıktı −2'dir ✓. Bir düzeltmeden sonra önceki testlerin de yinelenmesi gerekir, çünkü düzeltme daha önce çalışan bir durumu bozmuş olabilir. Test 1 yeniden denenir ve yine 12 çıkar.

Aynı programın hatalı ve düzeltilmiş Java sürümleri aynı girdilerle çalıştırılmıştır:

```
import java.util.Scanner;

public class EnBuyukHatali {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int a = giris.nextInt(), b = giris.nextInt(), c = giris.nextInt();
        int eb = 0;
        if (a > eb) eb = a;
        if (b > eb) eb = b;
        if (c > eb) eb = c;
        System.out.println("En büyük: " + eb);
    }
}
```

Girdi 7 12 5 ve −5 −2 −9 için çıktılar:

```
En büyük: 12
```

```
En büyük: 0
```

```
import java.util.Scanner;

public class EnBuyuk {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        int a = giris.nextInt(), b = giris.nextInt(), c = giris.nextInt();
        int eb = a;
        if (b > eb) eb = b;
        if (c > eb) eb = c;
        System.out.println("En büyük: " + eb);
    }
}
```

Düzeltilmiş sürüm aynı iki girdi ve eşit değerli bir girdi (3, 3, 1) için:

```
En büyük: 12
```

```
En büyük: -2
```

```
En büyük: 3
```

Örnek 4.3.18

Düzeltilmiş en büyük programı için hangi test girdileri seçilmelidir? Her biri neyi sınar?

1. En büyük sayının üç konumda da bulunduğu girdiler: (12, 7, 5), (7, 12, 5), (7, 5, 12). Bir karşılaştırma satırındaki yazım hatası ancak en büyük o konumdayken ortaya çıkar.
2. Hepsi negatif: (−5, −2, −9). Başlangıç değeri hatasını yakalar.
3. Eşit değerler: (3, 3, 1) ve (4, 4, 4). `>` yerine `>=` yazılması sonucu değiştirmez ama eşitlik durumunun da denendiğini gösterir.
4. Sıfır içeren girdi: (0, −1, −3). Beklenen 0'dır; hatalı sürüm de 0 verir. Bu test tek başına hatalı sürümü yakalayamaz, yanlış başlangıç değerinin şans eseri doğru sonuç verdiği bir durumdur.

4.3.7.3 Sık yapılan hatalar

Bazı hatalar o kadar sık yapılır ki adları vardır. Tablo 4.3.21 bunları örnekleri ve sonuçlarıyla listeler.

Hata	Örnek	Ne olur?
Başlangıç değeri yok ya da yanlış	toplam başlatılmadan `toplam ← toplam + x`	Java derlemez; bazı dillerde rastgele bir değerle hesaplar
Sınır hatası (bir eksik / bir fazla)	`her i ← 1 … n − 1 için`	Son öğe atlanır ya da bir fazla tur döner
Sonsuz döngü	Döngü değişkeni gövdede güncellenmez	Program bitmez
Sıfıra bölme	`ort ← toplam / n`, n = 0	Çalışma zamanı hatası
Dizinin dışına çıkmak	4 öğeli dizide `notlar[4]`	Çalışma zamanı hatası
Parantez unutmak	`ort ← a + b / 2`	Yanlış sonuç (mantık hatası)
Girdiyi doğrulamamak	Yaş olarak −3 kabul etmek	Anlamsız sonuç, sonraki adımlarda hata

Tablo 4.3.21: Sık yapılan hatalar

Başlangıç değeri hatasında diller farklı davranır. Java, değer verilmemiş bir yerel değişkenin okunmasına izin vermez:

```
public class Baslatilmamis {
    public static void main(String[] args) {
        int toplam;
        toplam = toplam + 5;
        System.out.println(toplam);
    }
}
```

```
Baslatilmamis.java:4: error: variable toplam might not have been initialized
        toplam = toplam + 5;
                 ^
1 error
```

C'de ise aynı kod derlenir. Değer verilmemiş yerel değişken, bellekte o anda ne varsa onu tutar ve program her çalıştırmada farklı bir sonuç verebilir. Java'nın katılığı burada bir korumadır: hatayı çalışma zamanından derleme zamanına taşır.

Sınır hatası (*off-by-one error*) en sinsi hatalardandır. 1'den n'ye kadar toplam isterken `her i ← 1 … n − 1 için` yazılırsa n atlanır. n = 5 için toplam 15 yerine *1 + 2 + 3 + 4 = 10* çıkar. Program çalışır, sonuç "makul" görünür ve hata kolayca gözden kaçar.

Bu hataların çoğu **sınır durumlarıyla** (*edge cases*) ortaya çıkar: 0, boş liste, en küçük ve en büyük değer, negatif sayı. Normal girdilerle çalışan bir program sınırlarda çoğu zaman bozulur. Bu yüzden her test kümesinde birkaç sınır durumu bulunmalıdır.

Örnek 4.3.19

Bir program, girilen notların ortalamasını `ort ← toplam / n` ile hesaplıyor. Hangi sınır durumları denenmelidir, her biri hangi hatayı ortaya çıkarabilir?

1. Hiç not girilmemesi (n = 0): sıfıra bölme. Program, n = 0 iken "not girilmedi" yazacak biçimde düzeltilir.
2. Tek not (n = 1): ortalama notun kendisi olmalıdır. Döngü sınırında bir eksik varsa sonuç 0 çıkar.
3. En küçük ve en büyük notlar (0 ve 100): girdi denetimi bu değerleri yanlışlıkla reddetmemelidir; örneğin `not > 0` yerine `not ≥ 0` yazılmalıdır.
4. Geçersiz not (−5 ya da 120): program bu değeri kabul etmemelidir.
5. Kesirli sonuç (70, 85, 90): ortalama 81,67 civarında olmalıdır. Tam sayı bölmesi varsa 81 çıkar.

4.3.7.4 Yazılım hatalarının bedeli

Bir ödevdeki hata düşük bir notla sonuçlanır. Gerçek sistemlerdeki hataların bedeli ise milyonlarca dolar, hatta insan hayatı olabilir. Üç ünlü örnek, bu notta anlatılan hata türlerinin gerçek dünyadaki karşılığını gösterir.

Ariane 5 (1996). Avrupa Uzay Ajansı'nın yeni roketi Ariane 5, ilk uçuşunda kalkıştan yaklaşık 37 saniye sonra rotasından saptı ve birkaç saniye içinde parçalandı. Soruşturma kurulunun raporuna göre neden bir sayı dönüşümüydü. Seyrüsefer yazılımı, 64 bitlik kayan noktalı bir yatay hız değerini 16 bitlik işaretli bir tam sayıya dönüştürüyordu. 16 bitlik işaretli bir tam sayı en fazla 32.767 değerini tutabilir. Yazılım daha yavaş tırmanan Ariane 4'ten alınmıştı ve orada bu sınır hiç aşılmamıştı. Ariane 5'in yatay hızı çok daha büyüktü, dönüşüm taşma hatası verdi ve seyrüsefer sistemi durdu.

Örnek 4.3.20

Java'da 40000.0 değerindeki bir double, 16 bitlik `short` tipine zorla dönüştürülürse ne olur?

```
public class Tasma {
    public static void main(String[] args) {
        double hiz = 40000.0;
        short kisa = (short) hiz;
        System.out.println(kisa);
        System.out.println(Short.MAX_VALUE);
    }
}
```

```
-25536
32767
```

short tipinin en büyük değeri 32.767'dir. 40.000 bu sınırı aşar. Java hata vermez; sayının ikili gösteriminin yalnızca son 16 bitini tutar. 16 bit $2^{16} = 65.536$ farklı değer tutabildiği için sonuç $40.000 - 65.536 = -25.536$ olur. Pozitif bir hız sessizce negatif bir sayıya dönüşmüştür. Ariane 5'in yazılımı Ada diliyle yazılmıştı ve taşmada sessiz kalmak yerine bir hata verdi. Ancak o hatayı yakalayıp işleyen kod yoktu ve sistem kapandı. İki durumda da kök neden aynıdır: değerin, tipin alabileceği aralığın dışına çıkabileceği hiç denenmemiştir.

Mars Climate Orbiter (1999). NASA'nın Mars'a gönderdiği sonda, yörüngeye girerken gezegene fazla yaklaştı ve kayboldu. Yer istasyonundaki bir yazılım iticilerin etkisini İngiliz birimiyle (pound-kuvvet·saniye) üretiyordu. Bu veriyi kullanan öbür yazılım ise metrik birim (newton·saniye) bekliyordu. 1 pound-kuvvet·saniye yaklaşık 4,45 newton·saniyedir. Bu yüzden itmenin etkisi yaklaşık 4,45 kat küçük hesaplandı ve yörünge hesabı dokuz aydan uzun süren yolculuk boyunca giderek kaydı. İki programın her biri tek başına doğru çalışıyordu; hata aralarındaki arayüzdeydi.

Therac-25 (1985–1987). Kanser tedavisinde kullanılan bir radyoterapi cihazının yazılımında bir zamanlama hatası vardı. Operatör ayarları belirli bir sırayla ve çok hızlı girdiğinde güvenlik denetimi atlanıyordu. Önceki modellerde bu denetimi donanım kilitleri de yapıyordu; Therac-25'te bu iş yalnızca yazılıma bırakılmıştı. 1985–1987 arasındaki altı kazada hastalar normalin çok üstünde radyasyon dozu aldı; en az üç hasta hayatını kaybetti.

Üç hata da derleyicinin yakalayamayacağı türdendir. Biri bir sınır durumu, biri birim tutarsızlığı, biri zamanlamaya bağlı bir mantık hatasıdır. Ortak ders birkaç maddede toplanır. Sınır durumları denenmeli, birimler değişken adlarında ve belgelerde açıkça yazılmalıdır. Bir sistemden alınan yazılım, yeni sistemin koşullarıyla yeniden sınanmalı ve kod bir başkasına incelettirilmelidir.

> **Sık yapılan hata**
>
> Birimi değişken adında belirtmemek. `hiz` adlı bir değişkenin km/sa mı m/s mi olduğu koddan anlaşılmaz. `hizKmSa` ya da `hizMs` gibi adlar, iki programcının aynı sayıyı farklı birimlerde yorumlamasını önler. Mars Climate Orbiter'ın kaybı, bu küçük alışkanlığın değerini gösterir.

4.3.7.5 Okunabilir kod

Kod bir kez yazılır, defalarca okunur. Onu haftalar sonra yazanın kendisi, ekip arkadaşları ve bakımı devralan başkaları okur. Okunabilir kod hataları hem önler hem de bulunmalarını kolaylaştırır. Aşağıdaki iki sözde kod aynı işi yapar.

Çalışır ama okunmaz:

```
a ← oku()
b ← oku()
c ← a*b/2
yaz c
```

Aynı program, okunur:

```
// üçgenin alanı = taban × yükseklik / 2
taban ← oku()
yukseklik ← oku()
alan ← taban × yukseklik / 2
yaz "Alan: ", alan
```

İlk programı okuyan, c'nin ne olduğunu anlamak için formülü çözmek zorundadır; `a*b/2` bir üçgen alanı olabilir, bir ortalama hesabının yanlış yazılmış hâli de olabilir. İkinci program kendini anlatır. Aradaki farklar şunlardır:

- **Anlamlı adlar.** c yerine alan, a yerine taban yazılır. Okuyan kişi değişkenin rolünü adından anlar.

- **Tutarlı girinti ve boşluk.** İşlemlerin çevresindeki boşluklar ve iç içe blokların düzenli girintisi, yapının sınırını bir bakışta gösterir.
- **Sihirli sayı yerine adlandırılmış sabit.** Kodun içine gömülü, anlamı açıklanmamış sayıya **sihirli sayı** (*magic number*) denir. `fiyat * 0.20` yerine `KDV_ORANI ← 0.20` tanımlanır ve `fiyat * KDV_ORANI` yazılır. Oran değiştiğinde tek bir satır düzeltilir.
- **Tekrarı fonksiyona taşımak.** Aynı formül üç yerde yazılıysa bir düzeltmenin üç kez yapılması gerekir ve biri mutlaka unutulur.
- **Açıklama satırı.** Açıklama kodun ne yaptığını değil, neden yaptığını anlatır. `// x'e 1 ekle` açıklaması kodu tekrar eder ve bir şey katmaz. `// dizi 0'dan başlıyor, sıra numarası için 1 ekle` açıklaması ise bir niyeti açıklar.

Örnek 4.3.21

Okunabilir üçgen alanı programının Java karşılığı nedir? Taban 6, yükseklik 5 ve taban 7, yükseklik 3 için çıktılar nelerdir?

```java
import java.util.Scanner;

public class UcgenAlani {
    public static void main(String[] args) {
        Scanner giris = new Scanner(System.in);
        // üçgenin alanı = taban × yükseklik / 2
        double taban = giris.nextDouble();
        double yukseklik = giris.nextDouble();
        double alan = taban * yukseklik / 2;
        System.out.println("Alan: " + alan);
    }
}
```

```
Alan: 15.0
```

```
Alan: 10.5
```

6 × 5 / 2 = 15 ve *7 × 3 / 2 = 10,5*. Değişkenler double olduğu için ikinci sonuç kesirli kısmıyla doğru çıkar. int kullanılsaydı *21 / 2* tam sayı bölmesi 10 verirdi. Adlandırılmış sabit kullanımı da benzer biçimde yazılır:

```java
public class KDV {
    static final double KDV_ORANI = 0.20;

    public static void main(String[] args) {
        double fiyat = 250.0;
        double kdv = fiyat * KDV_ORANI;
        System.out.println("KDV: " + kdv);
        System.out.println("Toplam: " + (fiyat + kdv));
    }
}
```

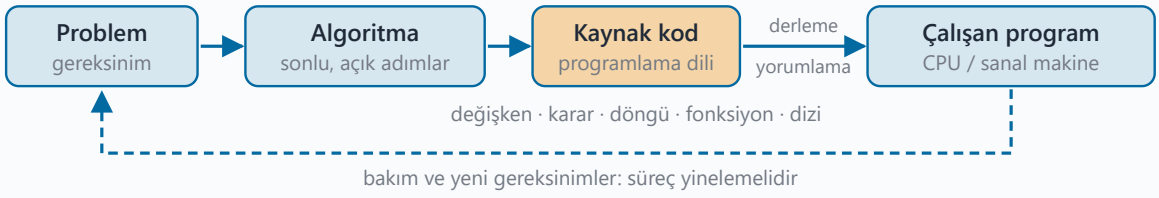
```
KDV: 50.0
Toplam: 300.0
```

250 TL'lik bir ürün için KDV *250 × 0,20 = 50* TL, toplam 300 TL'dir. Oran değişirse yalnızca `KDV_ORANI` satırı değiştirilir.

Sık yapılan hata

Kısa adların kodu hızlandırdığını ya da küçülttüğünü düşünmek. Derleyici adları makine koduna taşımaz; `c` ile `ucgeninAlani` aynı hızda çalışır. Kısa adların tek etkisi, kodu okuyanın işini zorlaştırmaktır. Döngü sayacı i gibi kısa ve yerleşik kullanımlar bunun istisnasıdır.

4.3.8 Özet



Şekil 4.3.33: Problemden çalışan programa: problem algoritmaya, algoritma kaynak koda dönüşür; kaynak kod derleme ya da yorumlama ile çalışır. Bakım ve yeni gereksinimler süreci başa döndürür.

- **Programlama dili**, algoritmayı bilgisayarın yürütebileceği biçimde ifade eden biçimsel bir dildir. **Söz dizimi** neyin yazılabileceğini, **anlam** yazılanın ne yaptığını belirler. Söz dizimi doğru bir program yanlış iş yapabilir.

- **Kaynak kod** insanın okuduğu metindir; **makine dili** işlemcinin çalıştırdığı ikili komutlardır; **assembly** makine komutlarının simgesel yazılışıdır. Yüksek düzeyli bir satır çoğu zaman birkaç makine komutuna karşılık gelir.
- Kaynak kod üç yolla çalıştırılır. **Derlenen** dillerde (C) program önceden makine koduna çevrilir. **Yorumlanan** dillerde (Python) komutlar çalışma sırasında değerlendirilir. **Hibrit** dillerde (Java) kaynak kod **bytecode**'a derlenir ve bytecode bir **sanal makinede** (JVM) yürütülür. **JIT** derleyici sık çalışan kodu çalışma sırasında makine koduna çevirir. Derlenen/yorumlanan ayrımı dilin değil, uygulamanın özelliğidir.
- Hatalar üç türdür. **Derleme hatası** çeviri sırasında bulunur, program hiç çalışmaz. **Çalışma zamanı hatası** yürütme sırasında programı durdurur. **Mantık hatası** hata iletisi vermez, yanlış sonuç üretir ve ancak test ile bulunur.
- Derleyici beş aşamada çalışır: sözcük çözümleme (belirteçler), söz dizimi çözümleme (söz dizimi ağacı), anlam çözümleme (adlar ve tipler), ara kod ve iyileştirme (ör. sabit katlama), kod üretimi. İlk üç aşama hata arar. **Bağlayıcı** nesne dosyalarını ve kütüphaneleri birleştirir, **yükleyici** programı belleğe alıp başlatır.
- Diller **düzeye** (düşük: makine dili, assembly; yüksek: C, Java, Python), **türe** (genel amaçlı, alana özgü, betik, işaretleme) ve **paradigmaya** (buyurgan, nesne yönelimli, fonksiyonel) göre sınıflandırılır. Düzey bir kalite ölçüsü değil, soyutlama düzeyidir. Dil seçimini problem alanı, platform, kütüphaneler, performans, güvenlik ve ekip deneyimi belirler.
- **Statik tiplemede** (Java) tip hataları derleme sırasında, **dinamik tiplemede** (Python) ilgili satır çalıştığında bulunur.
- Problemden programa giden yol: problemi tanımla (girdi, çıktı, kısıt), elle örnek çöz, algoritmayı kur, koda dönüştür, test et. Test bir eksik bulursa önceki aşamaya dönülür.
- **Yazılım geliştirme süreci** gereksinim, tasarım, kodlama, test, dağıtım ve bakımdan oluşan yinelemeli bir süreçtir. **Şelale modeli** aşamaları bir kez ve sırayla geçer. **Çevik** yaklaşım kısa döngülerle ilerler. **Prototipleme** belirsiz gereksinimi görünür kılar. IDE, derleyici, hata ayıklayıcı, sürüm kontrolü, test araçları ve paket yöneticisi bu sürecin araçlarıdır.
- **Değişken**, **karar**, **döngü**, **fonksiyon** ve **dizi** temel programlama yapılarıdır. İşleç önceliği matematikteki gibidir. Döngünün bitmesi için gövdenin koşulu etkilemesi gerekir. Her fonksiyon çağrısı çağrı yığınında kendi çerçevesini açar. Dizi indisleri 0'dan başlar, son indis uzunluk − 1'dir.
- Hatalar sınır durumlarıyla (0, boş liste, negatif sayı, en küçük ve en büyük değer) test edilerek ve program izleme tablosuyla adım adım izlenerek bulunur. Okunabilir kod (anlamlı adlar, adlandırılmış sabitler, fonksiyonlar, "neden" anlatan açıklamalar) hataları hem önler hem de bulunmalarını kolaylaştırır.

4.3.9 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

programlama dili *programming language* | söz dizimi *syntax* | anlam *semantics* | kaynak kod *source code*

makine dili *machine language* · makine kodu *machine code* · assembly *assembly language* · çeviri aracı *translator* · derleyici *compiler* · derleme *compilation* · yorumlayıcı *interpreter* · yorumlama *interpretation* · derlenen / yorumlanan *compiled / interpreted* · bytecode (ara kod) *bytecode* · ara kod *intermediate code* · sanal makine *virtual machine* · Java Sanal Makinesi (JVM) *Java Virtual Machine* · JIT (tam zamanında) derleyici *just-in-time compiler* · derleme hatası *compile error* · söz dizimi hatası *syntax error* · çalışma zamanı hatası *runtime error* · mantık hatası *logic error* · sözcük çözümleme *lexical analysis* · belirteç *token* · söz dizimi çözümleme *parsing* · söz dizimi ağacı *syntax tree* · anlam çözümleme *semantic analysis* · kod iyileştirme *code optimization* · sabit katlama *constant folding* · nesne dosyası *object file* · bağlayıcı *linker* · bağlama hatası *linking error* · yükleyici *loader* · çalıştırılabilir dosya *executable file* · sınıf yükleyici *class loader* · yüksek / düşük düzeyli dil *high-level / low-level language* · alana özgü dil *domain-specific language* · betik dili *scripting language* · işaretleme dili *markup language* · programlama paradigması *programming paradigm* · buyurgan yaklaşım *imperative programming* · nesne yönelimli *object-oriented* · fonksiyonel *functional programming* · statik tipleme *static typing* · dinamik tipleme *dynamic typing* · şelale modeli *waterfall model* · çevik *agile* · prototipleme *prototyping* · IDE *integrated development environment* · hata ayıklayıcı *debugger* · kesme noktası *breakpoint* · paket yöneticisi *package manager* · sürüm kontrolü *version control* · kayıt *commit* · dal *branch* · birleştirme *merge* · çakışma *merge conflict* · veri tipi *data type* · işleç önceliği *operator precedence* · sonsuz döngü *infinite loop* · parametre *parameter* · argüman *argument* · çağrı yığını *call stack* · dizi *array* · indis *index* · kütüphane *library* · API (uygulama programlama arayüzü) *application programming interface* · sınır hatası *off-by-one error* · sihirli sayı *magic number*

4.3.10 Kendini Dene

1. Aşağıdaki Java hatalarının her biri hangi türdendir? Derleme hatasıysa derleyicinin hangi aşamasında yakalanır? (a) `System.out.println("Merhaba);` (b) Bildirilmemiş `toplam` değişkenini kullanmak. (c) n öğeli bir dizide `i <= n` koşuluyla dönen döngü. (d) Ortalamayı hesaplarken toplamı

n − 1'e bölmek.

Cevap

(a) Derleme hatasıdır; kapanmamış dizgi bir belirtece dönüştürülemediği için sözcük çözümleme aşamasında yakalanır (javac: “unclosed string literal”). (b) Derleme hatasıdır; adın bildirilip bildirilmediğine anlam çözümleme aşamasında bakılır (“cannot find symbol”). (c) Çalışma zamanı hatasıdır; son turda i = n olur ve `dizi[n]` dizinin dışındadır, Java `ArrayIndexOutOfBoundsException` verir. (d) Mantık hatasıdır; program her girdiyle çalışır ama ortalamayı olduğundan büyük bulur.

2. Üç satırlık bir Python dosyası şöyle: `print("A")`, `print(1 / 0)`, `print("B"` (son satırda parantez kapanmamış). Dosya çalıştırılınca ekranda ne görünür? Satır satır çalışan basit bir yorumlayıcı ne gösterirdi?

Cevap

Python çalıştırmaya başlamadan önce dosyanın tamamını ayrıştırır. Üçüncü satırdaki kapanmamış parantezi bulur ve yalnızca `SyntaxError: '(' was never closed` iletisini verir. “A” yazılmaz, sıfıra bölmeye hiç ulaşılmaz. Satır satır çalışan basit bir yorumlayıcı ise önce “A” yazar, ikinci satırda sıfıra bölme hatasıyla durur; üçüncü satırdaki söz dizimi hatasını hiç görmez.

3. `toplam = toplam + 1;` satırı sözcük çözümlemede kaç belirtece ayrılır? Belirteçlerin türleri nelerdir?

Cevap

Altı belirteç: `toplam` (tanımlayıcı), `=` (işleç), `toplam` (tanımlayıcı), `+` (işleç), `1` (sabit değer), `;` (ayırıcı). Boşluklar atılır. Aynı ad iki kez geçtiği için iki ayrı belirteç olarak sayılır.

4. Aşağıdaki sözde kod ne yazar? Gövde kaç kez çalışır, koşul kaç kez denetlenir? `s ← 0`; `i ← 1`; `i ≤ 4 iken tekrarla: s ← s + i × i; i ← i + 2`; `yaz s`.

Cevap

1. tur: *1 ≤ 4* doğru; *s = 0 + 1 × 1 = 1*, i = 3. 2. tur: *3 ≤ 4* doğru; *s = 1 + 3 × 3 = 10*, i = 5. Denetim: *5 ≤ 4* yanlış, döngü biter. Ekrana 10 yazılır. Gövde 2 kez çalışır, koşul 3 kez denetlenir. Çarpma toplamadan önce yapıldığı için `s + i × i` ifadesinde parantez gerekmez.

5. `fonksiyon f(x): döndür 2 × x + 1` tanımlı. `yaz f(f(2))` ne yazar? Kaç çerçeve açılır, ikisi aynı

anda yığında bulunur mu?

Cevap

Önce içteki çağrı çalışır: f(2) için bir çerçeve açılır, x = 2, *2 × 2 + 1 = 5* döndürülür ve çerçeve kapanır. Sonra dıştaki çağrı f(5) için yeni bir çerçeve açılır: *2 × 5 + 1 = 11*. Ekrana 11 yazılır. İki çerçeve açılır ama aynı anda yığında bulunmaz: dıştaki çağrı, argümanı hesaplanıp içteki çerçeve kapandıktan sonra başlar.

6. Java'da `int s = 17 % 5 * 2 + 9 / 2;` satırından sonra s kaçtır? Hesaplama sırasını yazın.

Cevap

`%`, `*` ve `/` aynı öncelik düzeyindedir ve toplamadan önce, soldan sağa yapılır. *17 % 5 = 2*, *2 × 2 = 4*. Ardından *9 / 2* iki tam sayının bölümü olduğu için 4'tür (4,5 değil). Son olarak *4 + 4 = 8*. s = 8 olur.

7. Bir Python fonksiyonu x 10'dan büyükse `"a" + 5` döndürüyor, değilse `"küçük"`. Program `mesaj(3)` ve ardından `mesaj(20)` çağrılarını yapıyor. Hata ne zaman ortaya çıkar? Aynı mantık Java'da yazılsaydı ne olurdu?

Cevap

Python dosyayı sorunsuz ayrıştırır, çünkü söz dizimi doğrudur. `mesaj(3)` "küçük" döndürür ve yazılır. `mesaj(20)` çağrısında hatalı satır ilk kez çalışır ve `TypeError` verir. Dinamik tiplemede tip hatası ancak satır çalışınca görülür; x hiç 10'u geçmeseydi hata hiç görülmezdi. Java'da dönüş tipi `String` olan bir metotta `"a" + 5` dizgi birleştirmedir ve geçerlidir: "a5" döndürür. Dönüş tipi `int` olsaydı derleyici tip uyuşmazlığını program hiç çalışmadan bildirirdi.

8. Bir dizinin en küçük öğesini bulmak için `ek ← 0` ile başlayan ve her öğeyi `eğer d[i] < ek ise ek ← d[i]` ile karşılaştıran bir program yazılmış. d = [3, 8, 1, 6] için çıktı nedir? Hata nedir, nasıl düzeltilir?

Cevap

Dört karşılaştırmanın hiçbiri doğru değildir: 3, 8, 1 ve 6'nın hiçbiri 0'dan küçük değildir. ek hiç değişmez ve çıktı 0 olur; doğru sonuç 1'dir. Hata başlangıç değerindedir: 0, girdilerin dışından gelen bir değerdir ve pozitif sayılar için en küçük değer gibi davranır. Düzeltme `ek ← d[0]` 'dır. İzleme: ek = 3; 8 < 3 yanlış; 1 < 3 doğru, ek = 1; 6 < 1 yanlış. Çıktı 1 olur.

4.3.11 İleri okuma

- A. B. Downey, C. Mayfield, *Think Java* (2. baskı, O'Reilly, 2019): Java ile programlamaya giriş; ilk bölümleri derleme, yorumlama, hata türleri ve hata ayıklamayı bu nottaki düzeyde anlatır. Green Tea Press sitesinde ücretsiz okunabilir.
- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016): Java ile programlamayı bilim ve mühendislik örnekleri üzerinden öğretir; değişken, döngü, fonksiyon ve diziler için bol örnek içerir.
- R. W. Sebesta, *Concepts of Programming Languages* (12. baskı, Pearson, 2019): dillerin tarihçesini, söz dizimi ve anlamın tanımlanmasını, tip sistemlerini ve paradigmaları ayrıntılı karşılaştırır.
- A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (2. baskı, Addison-Wesley, 2006): derleyicinin aşamalarını anlatan temel kaynak; sözcük çözümleme, söz dizimi çözümleme ve kod iyileştirme konularını derinleştirmek isteyenler için.

4.3.12 Kaynaklar

1. R. W. Sebesta, *Concepts of Programming Languages*, 12. baskı, Pearson, 2019.
2. A. V. Aho, M. S. Lam, R. Sethi, J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2. baskı, Addison-Wesley.
3. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016.
4. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019 (ücretsiz çevrim içi sürüm).
5. Oracle, *The Java Tutorials: Getting Started*.

KISIM II · BÖLÜM 5

Java'ya Giriş ve İlk Program

Bir programı yazmak, onu çalıştırmanın yalnızca ilk adımıdır. Yazılan metin bir dosyaya kaydedilir, bir derleyiciden geçer, ara bir koda dönüşür ve sanal bir makinede çalışır. Bu yolda her adımın kendi kuralları ve kendi hata iletileri vardır. Bu not, program ve programlama dili kavramlarıyla başlar. Ardından Java'nın doğuşunu, JDK, JRE ve JVM ilişkisini ve ilk Java programını satır satır anlatır. Konsola yazdırma, kaçış dizileri ve dizgi birleştirme örneklerle işlenir. Derleme ve çalıştırma adımları, bytecode'un içi, dilin temel yapı taşları ve açıklamalar bunları izler. Son kesimler hata türlerini, hata iletilerini okumayı ve kodlamadan önce yapılacak hazırlığı ele alır. Notta gösterilen bütün programlar JDK 21 ile derlenip çalıştırılmıştır; derleyici iletileri ve konsol çıktıları bu çalıştırmalardan birebir alınmıştır.

Bu bölümün kodu: `github.com/sercankulcu/computer-programming-java/Bolum05/src`

ÖĞRENME ÇIKTILARI

Program
Programı girdi, işlem, durum ve çıktıdan oluşan bir bütün olarak açıklamak; programlama dilinin söz dizimi, anlam, tip sistemi ve çalışma modeli yönlerini ayırt etmek.

Java
Kaynak kod, bytecode ve JVM ilişkisini kurmak; JDK, JRE ve JVM'nin içeriğini ve görevini açıklamak.

İlk program
Bir Java dosyasını yazmak, `javac` ile derlemek ve `java` ile çalıştırmak; `main` metodunu, komut satırı argümanlarını, `print` / `println` farkını, kaçış dizilerini ve dizgi birleştirmeyi doğru kullanmak.

Hatalar
Söz dizimi, tip, çalışma zamanı ve mantık hatalarını ayırt etmek; derleyici ve JVM iletilerini okuyup hatanın yerini ve nedenini bulmak.

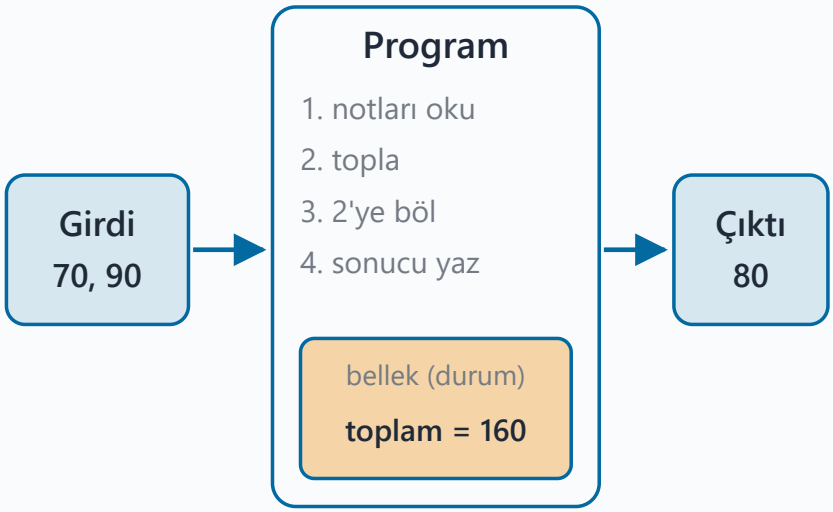
5.1 Program ve programlama dili

Bilgisayar kendi başına hiçbir iş yapmaz. Hesap makinesi uygulaması da, bir oyun da, bir banka sistemi de bilgisayara ne yapacağını adım adım söyleyen komutlardan oluşur. Bu komutları yazmaya **programlama**, yazılan komutların bütününe **program** denir. Komutlar, bilgisayarın ve insanın anlaşabileceği yapay bir dille yazılır.

5.1.1 Program nedir?

> **Tanım 5.1**
>
> **Program** (*program*), bilgisayarın belirli bir görevi yerine getirmek için yürüttüğü komutlar bütünüdür. Bir program girdi alabilir, veriyi işler, ara değerleri bellekte saklar ve çıktı üretir.

Tanımdaki dört öğe her programda görülür. **Girdi** (*input*), programa dışarıdan verilen bilgidir: klavyeden yazılan bir sayı, bir dosyanın içeriği ya da bir algılayıcıdan gelen ölçüm. **İşlem**, girdiden yeni değerler üreten adımlardır: toplama, karşılaştırma, sıralama. **Durum** (*state*), programın çalışırken bellekte tuttuğu değerlerin o anki hâlidir. **Çıktı** (*output*) ise programın dışarıya verdiği sonuçtur: ekrana yazılan bir satır, kaydedilen bir dosya ya da yakılan bir lamba.



Şekil 5.1: İki notun ortalamasını bulan program. Girdi 70 ve 90'dır. Program notları okur, toplar, 2'ye böler ve sonucu yazar. Ara değer (toplam = 160) bellekte durum olarak saklanır; çıktı 80'dir.

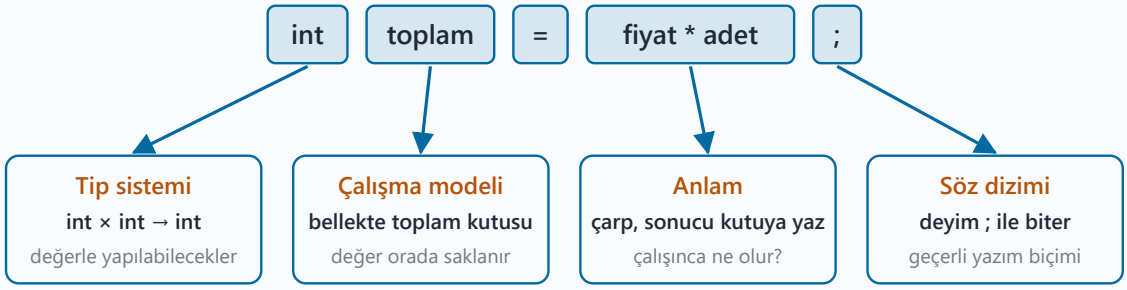
Şekil 5.1 bu öğeleri küçük bir örnek üzerinde gösterir. Program iki not alır: 70 ve 90. İlk adımda notlar okunur. İkinci adımda toplanır ve toplam, yani 160, bellekte saklanır. Üçüncü adımda toplam 2'ye bölünür ve 80 bulunur. Son adımda 80 ekrana yazılır. Aynı program 50 ve 60 notları için 55 yazar.

Program değişmez, yalnızca girdi değişir. Programı değerli yapan da budur: bir kez yazılır, farklı girdilerle tekrar tekrar çalıştırılır.

Bir programın davranışını iki şey belirler. Birincisi, programın yazıldığı dilin kurallarıdır: hangi işaretin ne anlama geldiği, bir sayının nasıl saklandığı. İkincisi, programcının seçtiği yöntem, yani **algoritmadır**. Ortalama örneğinde "önce topla, sonra böl" bir algoritmadır. Aynı algoritma farklı dillerde yazılabilir. Bu notta dil olarak Java kullanılır.

5.1.2 Programlama dilinin yönleri

Türkçe gibi doğal dillerde de bir cümlenin iki yönü vardır: doğru yazılıp yazılmadığı ve ne anlattığı. "Ben kitabı masada okudu." cümlesi dil bilgisi bakımından kusurludur. "Masa kitabı okudu." ise doğru kurulmuştur ama anlamsızdır. Programlama dillerinde bu iki yöne iki yön daha eklenir. Bir programlama dili dört yönden incelenir.



Şekil 5.2: `int toplam = fiyat * adet;` deyiminin dört yönü. Söz dizimi deyimin `;` ile bitmesini ister. Tip sistemi iki `int`'in çarpımının da `int` olduğunu söyler. Anlam "çarp, sonucu kutuya yaz" demektir. Çalışma modeli ise sonucun bellekteki `toplam` kutusunda saklandığını anlatır.

Tanım 5.2

Söz dizimi (*syntax*), bir programın geçerli yazım biçimini belirleyen kurallardır. **Anlam** (*semantics*), geçerli bir program parçası çalıştırıldığında ne olacağını tanımlar. **Tip sistemi** (*type system*), her değerin bir tipi olmasını ve bir değerle yalnızca tipine uygun işlemlerin yapılmasını sağlar. **Çalışma modeli** (*execution model*), komutların hangi sırayla yürütüldüğünü ve değerlerin bellekte nasıl tutulduğunu anlatır.

Şekil 5.2 bu dört yönü tek bir deyim üzerinde gösterir: `int toplam = fiyat * adet;`. Söz dizimi bakımından deyim bir tip adıyla başlar, ardından bir ad, eşittir işareti, bir ifade ve noktalı virgül gelir. Noktalı virgül unutulursa deyim geçersizdir. Tip sistemi bakımından `fiyat` ve `adet` tam sayıdır (`int`), iki tam sayının çarpımı da tam sayıdır. Bu yüzden sonuç `int` tipindeki `toplam`'a konabilir. Anlam bakımından deyim "fiyat ile adedi çarp, sonucu toplam'a yaz" demektir. Çalışma modeli bakımından bellekte `toplam` adlı bir kutu ayrılır ve sonuç bu kutuda saklanır.

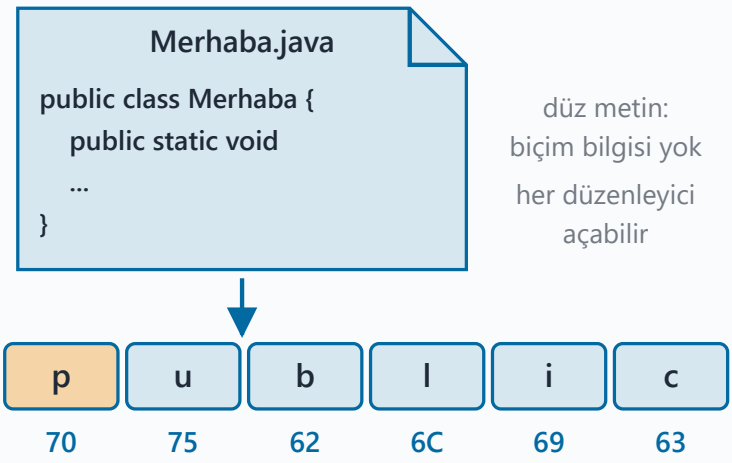
Java'da söz dizimi ve tip kuralları program çalıştırılmadan önce denetlenir. Bu denetimi **derleyici** (*compiler*) yapar. Noktalı virgülü unutulmuş ya da bir metni sayı kutusuna koymaya çalışan bir program hiç çalıştırılmaz; derleyici hatayı satır numarasıyla bildirir. Anlamla ilgili yanlışlar ise çoğu zaman ancak program çalışırken ortaya çıkar. Ortalamayı bulurken bölmeyi unutan bir program kusursuz derlenir ama yanlış sonuç üretir.

5.2 Kaynak kod ve araçlar

Programcının yazdığı metne **kaynak kod** (*source code*) denir. Kaynak kod insan için yazılır: okunabilir sözcüklerden, sayılardan ve işaretlerden oluşur. İşlemci ise yalnızca kendi makine komutlarını anlar. Kaynak kodu çalışan bir programa dönüştürmek için birkaç araç birlikte kullanılır.

5.2.1 Düz metin kaynak dosyası

Bir Java programı, uzantısı `.java` olan bir dosyaya yazılır. Bu dosyaya **kaynak dosya** (*source file*) denir. Kaynak dosya **düz metindir** (*plain text*). Düz metin dosyasında yalnızca karakterler bulunur; yazı tipi, renk, kalın yazı ya da sayfa düzeni gibi biçim bilgisi yoktur. Bu yüzden bir Java dosyası Not Defteri'yle, VS Code gibi bir kod düzenleyiciyle ya da bir tümleşik geliştirme ortamıyla açılabilir. Word belgesi ise düz metin değildir; içinde biçim bilgisi de saklanır. Kaynak kod Word'de yazılıp `.docx` olarak kaydedilirse derleyici dosyayı okuyamaz.



Şekil 5.3: `Merhaba.java` düz metin dosyasıdır. Diskte her karakter bir ya da birkaç bayt olarak saklanır: "public" sözcüğünün harfleri onaltılık 70, 75, 62, 6C, 69, 63 baytlarıdır. ASCII karakterleri UTF-8'de 1 bayt, ç, ğ, ş gibi harfler 2 bayt tutar.

Diskte her şey bayttır. Bir düz metin dosyasında her karakter, bir **karakter kodlamasına** (*character encoding*) göre bir ya da birkaç bayta çevrilir. Bugün en yaygın kodlama UTF-8'dir. UTF-8'de İngilizce alfabedeki harfler, rakamlar ve noktalama işaretleri ASCII'deki gibi tek bayt tutar. Şekil 5.3 "public"

sözcüğünün baytlarını gösterir: p harfi onaltılık 70, yani onluk 112'dir. Türkçeye özgü ç, ğ, ı, ö, ş, ü harfleri UTF-8'de ikişer bayt tutar.

Kodlama, dosyayı yazan ve okuyan programların anlaşmasıdır. Dosya UTF-8 ile kaydedilip başka bir kodlamayla okunursa Türkçe harfler bozulur. “Dünya” sözcüğü “DÃ¼nya” gibi görünebilir. Java derleyicisi de kaynak dosyayı okurken bir kodlama varsayar. JDK 18'den beri varsayılan kodlama UTF-8'dir. Bu notun örnekleri JDK 21 ile hiçbir ek seçenek verilmeden derlenmiştir; “Dünya”, “Satır” ve “İndirim” gibi sözcükler doğru çıkmıştır. Daha eski bir JDK kullanılıyorsa derleyiciye kodlamayı `-encoding UTF-8` seçeneğiyle açıkça söylemek gerekir.

Kodlama dışında iki ayrıntı daha tutarlı olmalıdır. Birincisi satır sonlarıdır: Windows satır sonunu iki karakterle (CR LF), Linux ve macOS tek karakterle (LF) yazar. Java derleyicisi ikisini de kabul eder. Ama aynı projede karışık satır sonları, değişiklikleri izleyen araçlarda gereksiz farklar doğurur. İkincisi dosya adıdır. Java'da dosya adı, içindeki public sınıfın adıyla aynı olmak zorundadır: `Merhaba` sınıfı `Merhaba.java` dosyasına yazılır.

Düz metnin bir yararı daha vardır. **Sürüm kontrol sistemleri** (*version control system*), bir dosyanın hangi satırını kimin, ne zaman değiştirdiğini kaydeder. Git en yaygın örnektir. Bu araçlar satır satır karşılaştırma yaptığı için düz metin dosyalarla en iyi sonucu verir.

5.2.2 Programlama araçları

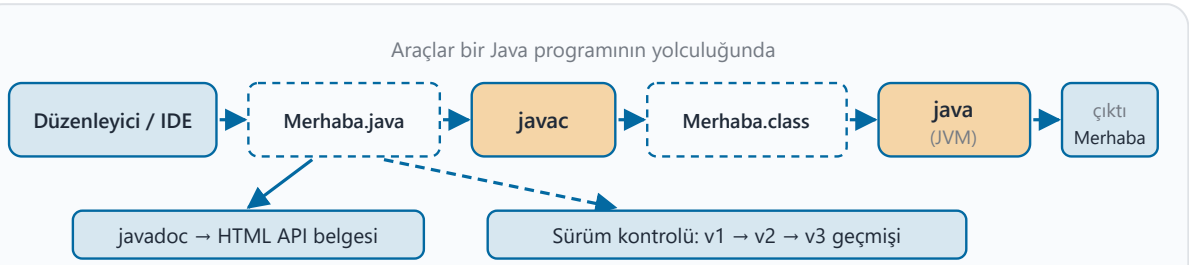
Bir Java programı yazılıp çalıştırılana kadar birkaç aracın elinden geçer. Her aracın tek ve belirli bir görevi vardır.

Araç	Görevi	Örnek
Düzenleyici	Kaynak kodu yazmak ve kaydetmek	Not Defteri, VS Code
IDE	Proje yönetimi, kod tamamlama, derleme, çalıştırma ve hata ayıklama	IntelliJ IDEA, Eclipse, NetBeans
`javac`	Kaynak kodu bytecode'a derlemek	`javac Merhaba.java`
`java`	Bir sınıfı JVM üzerinde çalıştırmak	`java Merhaba`
`javadoc`	Kaynak koddaki belge açıklamalarından HTML biçiminde API belgesi üretmek	`javadoc Merhaba.java`
Sürüm kontrolü	Değişiklik geçmişini tutmak	Git

Tablo 5.1: Java programlama araçları ve görevleri.

Kaynak kodu yazmak için bir **düzenleyici** (*editor*) yeterlidir. Kod düzenleyicileri sıradan metin düzenleyicilerden farklı olarak anahtar sözcükleri renklendirir, girintiyi kendiliğinden ayarlar ve satır numaralarını gösterir. **Tümleşik geliştirme ortamı** (*integrated development environment*, IDE) düzenleyiciyi derleyici, çalıştırıcı ve hata ayıklayıcıyla tek bir pencerede birleştirir.

`javac` Java derleyicisidir; kaynak kodu okur, denetler ve **bytecode** adı verilen ara koda çevirir. `java` komutu bir Java Sanal Makinesi başlatır ve bytecode'u çalıştırır. `javadoc`, kodun içine özel biçimde yazılmış açıklamalardan web sayfası biçiminde belge üretir. Java'nın kendi sınıf kütüphanesinin çevrim içi belgeleri de bu araçla üretilmiştir.



Şekil 5.4: Araçlar bir Java programının yolculuğunda. Düzenleyicide yazılan `Merhaba.java`, `javac` ile `Merhaba.class` dosyasına derlenir. `java` komutu bu dosyayı JVM'de çalıştırır ve konsola `Merhaba` yazılır. `javadoc` kaynak koddan belge üretir; sürüm kontrolü kaynak kodun geçmişini (v1 → v2 → v3) tutar.

Şekil 5.4 araçların sırasını gösterir. Dikkat edilecek nokta şudur: `javadoc` ve sürüm kontrolü kaynak dosyayla çalışır, `.class` dosyasıyla değil. `.class` dosyası her derlemede yeniden üretilebildiği için sürüm kontrolüne genellikle eklenmez. Saklanması gereken asıl ürün kaynak koddur.

5.3 Java

Java, bugün en çok kullanılan programlama dillerinden biridir. Bankacılık sistemlerinde, Android uygulamalarında, büyük veri araçlarında ve üniversitelerin programlamaya giriş derslerinde karşımıza çıkar. Bu yaygınlığın en önemli nedeni, Java programlarının işletim sisteminden ve işlemciden bağımsız çalışabilmesidir. Bu kesim Java'nın nasıl doğduğunu, nerelerde kullanıldığını ve bir Java programını çalıştırmak için hangi yazılımların gerektiğini anlatır.

5.3.1 Java'nın doğuşu ve sürümleri

Java'nın hikâyesi 1991'de Sun Microsystems'te başlar. James Gosling ve ekibi “Green projesi” adlı bir çalışmada televizyon kutuları gibi akıllı ev cihazları için bir dil tasarlamaya girişti. Dilin ilk adı Oak'tı (meşe). Ev cihazları pazarı beklenen ilgiyi görmedi, ama aynı yıllarda Web hızla yayılıyordu. Farklı işletim sistemli bilgisayarlarda aynı biçimde çalışabilen bir dil tam da Web'in ihtiyacıydı. Dil 1995'te Java adıyla yayımlandı.



Şekil 5.5: Java'nın zaman çizelgesi. 1991'de Green projesi (ilk adı Oak), 1995'te Java 1.0, 2006'da açık kaynak OpenJDK, 2010'da Oracle'ın Sun'ı satın alması. Vurgu rengindeki noktalar uzun süreli destek (LTS) sürümleridir: Java 8 (2014), 11 (2018), 17 (2021), 21 (2023) ve 25 (2025). 2018'den beri yeni sürümler altı ayda bir çıkar.

Java'nın tasarım hedefi tek bir cümleyle özetlenir: **bir kez yaz, her yerde çalıştır** (*write once, run anywhere*). Bir Java programı bir kez derlenir. Derleyicinin ürettiği bytecode, Windows'ta, Linux'ta ya da macOS'ta, o sistem için yazılmış Java Sanal Makinesi üzerinde değiştirilmeden çalışır. Programcının her işletim sistemi için ayrı derleme yapması gerekmez.

Java'nın öteki temel özellikleri şunlardır:

- **Nesne yönelimli** (*object-oriented*): Programlar sınıflar ve nesneler etrafında kurulur. En küçük Java programı bile bir sınıfın içine yazılır.
- **Güçlü tip denetimi**: Her değişkenin bir tipi vardır ve derleyici tip uyuşmazlıklarını program çalışmadan önce yakalar.
- **Otomatik bellek yönetimi**: Artık kullanılmayan nesnelerin belleği **çöp toplama** (*garbage collection*) ile kendiliğinden geri alınır. Programcı belleği elle serbest bırakmaz.
- **Geniş sınıf kütüphanesi**: Ekrana yazma, dosya okuma, ağ bağlantısı, tarih hesabı gibi işler için hazır sınıflar vardır.

Java'nın sahibi 2010'da değişti: Oracle, Sun Microsystems'i satın aldı ve dili o günden beri geliştiriyor. Java'nın kaynak kodu 2006'dan beri açıktır ve OpenJDK projesinde yayımlanır. Bugün Oracle'ın yanı sıra Eclipse Adoptium (Temurin), Amazon (Corretto) ve Microsoft gibi kuruluşlar da OpenJDK tabanlı JDK dağıtımları sunar. Bu notun örnekleri Eclipse Adoptium'un JDK 21 dağıtımıyla çalıştırılmıştır.

2018'den beri her altı ayda bir yeni Java sürümü çıkar: Mart ve Eylül aylarında. Her sürüm yalnızca birkaç ay güncelleme alır. Bazı sürümler ise **uzun süreli destek** (*long-term support*, LTS) sürümü olarak ilan edilir ve yıllarca güvenlik güncellemesi alır. Kurumlar ve dersler bu yüzden LTS sürümlerini tercih eder. Java 8, 11, 17, 21 ve 25 LTS sürümleridir. Bu derste Java 21 kullanılır.

5.3.2 Java nerelerde kullanılır?

Java'nın kullanım alanları çok geniştir. Aşağıdaki tablo en bilinen alanları özetler.

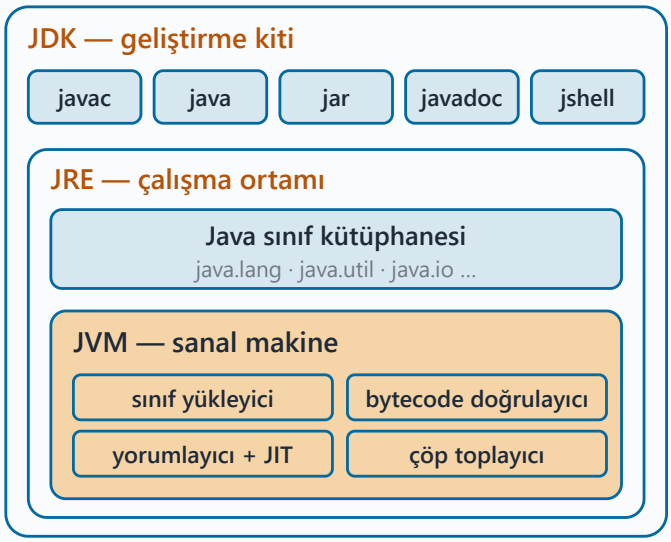
Alan	Açıklama
Kurumsal sunucular	Bankacılık, e-ticaret ve kamu sistemlerinin arka ucu; Spring ve Jakarta EE çatıları
Android	Uygulamalar Java ya da Kotlin ile yazılır; bytecode, telefondaki Android çalışma ortamına (ART) uygun biçime çevrilir
Büyük veri	Hadoop ve Kafka gibi veri işleme araçları JVM üzerinde çalışır
Geliştirme araçları	IntelliJ IDEA, Eclipse ve NetBeans gibi IDE'lerin kendisi de Java ile yazılmıştır
Akıllı kartlar	SIM kartlar ve banka kartlarındaki küçük programlar Java Card ile çalışır
Oyun	Minecraft'ın Java Sürümü; masaüstü oyunlar ve simülasyonlar

Tablo 5.2: Java'nın başlıca kullanım alanları.

Tablodaki örneklerin bir kısmı Java dilinin değil, Java Sanal Makinesinin başarısıdır. JVM yalnızca Java'yı çalıştırmaz. Kotlin, Scala ve Groovy gibi diller de aynı bytecode'a derlenir ve aynı sanal makinede çalışır. Android uygulamalarının bugün çoğu Kotlin ile yazılır, ama arkadaki çalışma modeli aynıdır.

5.3.3 JDK, JRE ve JVM

Java ile ilgili belgelerde üç kısaltma sürekli geçer: JDK, JRE ve JVM. Üçü iç içe katmanlardır; dıştaki katman içtekini kapsar.



Şekil 5.6: JDK, JRE ve JVM iç içe katmanlardır. JDK geliştirme araçlarını (javac, java, jar, javadoc, jshell) ve JRE'yi içerir. JRE, Java sınıf kütüphanesini (java.lang, java.util, java.io …) ve JVM'yi içerir. JVM'nin parçaları sınıf yükleyici, bytecode doğrulayıcı, yorumlayıcı ile JIT derleyicisi ve çöp toplayıcıdır.

Tanım 5.3

Java Sanal Makinesi (*Java Virtual Machine*, JVM), bytecode'u yürüten yazılımdır. **JRE** (*Java Runtime Environment*, Java çalışma ortamı), JVM ile Java sınıf kütüphanesinin birleşimidir: bir Java programını çalıştırmak için gereken her şeyi içerir. **JDK** (*Java Development Kit*, Java geliştirme kiti), JRE'ye geliştirme araçlarını (`javac` , `jar` , `javadoc` ...) ekler.

Kısaca: JDK = JRE + geliştirme araçları, JRE = JVM + sınıf kütüphanesi. Şekil 5.6 bu ilişkiyi gösterir. Yalnızca hazır bir Java programını çalıştıracak kullanıcıya JRE yeterlidir. Program yazacak ve derleyecek geliştiricinin ise JDK kurması gerekir, çünkü `javac` yalnızca JDK'de bulunur.

Bu ayrım günümüzde biraz değişti. Oracle, Java 11'den beri ayrı bir JRE paketi sunmuyor; geliştirici de kullanıcı da JDK kurar. Bazı dağıtıcılar JRE'yi hâlâ ayrıca sunar. Kavram olarak üç katmanı bilmek yine de önemlidir, çünkü hata iletileri ve belgeler bu adları kullanır.

JVM'nin beş parçası programın yaşamında sırayla iş görür. **Sınıf yükleyici** (*class loader*), `.class` dosyasını diskten okuyup belleğe alır. **Bytecode doğrulayıcı** (*bytecode verifier*), yüklenen kodun kurallara uyduğunu denetler; örneğin bir komutun yığında olmayan bir değeri kullanmaya çalışmadığını. **Yorumlayıcı** (*interpreter*) bytecode komutlarını tek tek yürütür. **JIT derleyicisi** (*just-in-time compiler*, tam zamanında derleyici) sık çalışan kısımları işlemcinin makine koduna çevirerek hızlandırır. **Çöp toplayıcı** da program çalıştıkça kullanılmayan belleği geri alır.

Sık yapılan hata

Bilgisayarda birden çok Java kurulu olabilir ve komut satırı ilk bulduğunu kullanır. Bu notun hazırlandığı bilgisayarda `javac` JDK 21'den, `PATH` 'teki `java` ise eski bir Java 8 kurulumundan geliyordu. JDK 21 ile derlenen `Merhaba.class` , Java 8 ile çalıştırılınca şu ileti alındı:

```
Exception in thread "main" java.lang.UnsupportedClassVersionError: Merhaba has been
compiled by a more recent version of the Java Runtime (class file version 65.0),
this version of the Java Runtime only recognizes class file versions up to 52.0
```

İleti açıktır: sınıf dosyası sürüm 65 (Java 21) biçimindedir, çalıştıran JVM ise en çok sürüm 52'yi (Java 8) tanır. Eski bir JVM yeni bytecode'u çalıştıramaz. Çözüm, `java -version` ile `javac -version` çıktılarının aynı sürümü gösterdiğinden emin olmaktır.

5.4 İlk Java programı

Programlama öğrenmenin geleneksel ilk adımı ekrana kısa bir selam yazdıran bir programdır. Bu program küçüktür, ama bir Java programının bütün zorunlu parçalarını içerir. Bu kesim programı önce parçalarına ayırır, sonra JVM'nin onu nasıl çalıştırdığını adım adım izler.

5.4.1 Programın parçaları

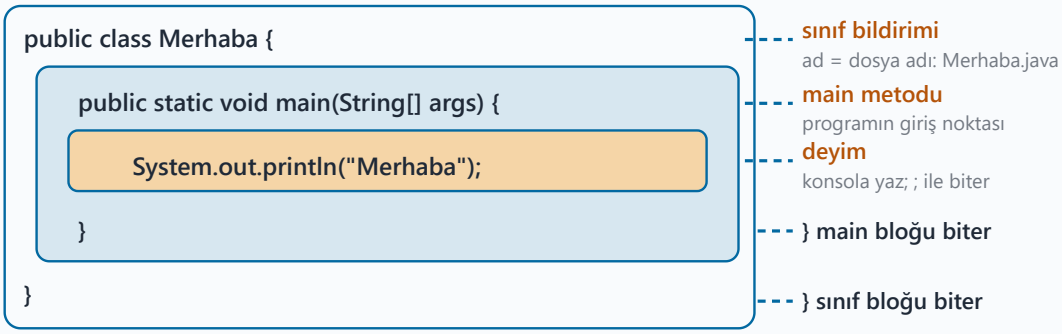
Aşağıdaki program `Merhaba.java` adlı bir dosyaya kaydedilir:

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

Program derlenip çalıştırıldığında konsola tek bir satır yazar:

```
> javac Merhaba.java
> java Merhaba
Merhaba
```

Konsol oturumlarında `>` ile başlayan satırlar kullanıcının yazdığı komutlardır; altındaki satırlar komutun çıktısıdır. `javac` komutunun altında hiçbir satır yoktur: derleme başarılı olduğunda derleyici sessiz kalır.



Şekil 5.7: Merhaba programının parçaları. Dıştaki blok sınıf bildirimidir; sınıfın adı dosya adıyla aynıdır. İçteki blok programın giriş noktası olan `main` metodudur. En içteki satır konsola yazan deyimdir ve `;` ile biter. Her `}` açtığı bloğu kapatır.

Şekil 5.7 programı iç içe üç katman olarak gösterir. Satırları tek tek inceleyelim.

1. `public class Merhaba {` bir **sınıf bildirimidir** (*class declaration*). Java'da her kod bir sınıfın içinde bulunur. `class` anahtar sözcüğünden sonra sınıfın adı gelir. `public`, sınıfın her yerden erişilebilir olduğunu söyler. Public bir sınıfın adı dosya adıyla birebir aynı olmalıdır: `Merhaba` sınıfı `Merhaba.java` dosyasına yazılır. Satır sonundaki `{` sınıfın gövdesini açar.
2. `public static void main(String[] args) {` **main metodunu** (*main method*) bildirir. Metot, bir ada sahip bir iş birimidir. JVM bir programı çalıştırırken işe her zaman `main` metodundan başlar; bu yüzden `main` programın **giriş noktasıdır** (*entry point*).

3. `System.out.println("Merhaba");` programın asıl işini yapan **deyimdir** (*statement*). `System.out` standart çıktı akışını, yani konsolu temsil eder. `println` metodu kendisine verilen metni konsola yazar ve satırı bitirir. Çift tırnak içindeki `"Merhaba"` bir **dizgidir** (*string*): karakterlerden oluşan bir metin değeri. Deyim noktalı virgülle biter.
4. İlk `}` `main` metodunun gövdesini, ikinci `}` sınıfın gövdesini kapatır.

`main` metodunun başındaki sözcüklerin her birinin bir görevi vardır:

Sözcük	Anlamı
`public`	Metot her yerden erişilebilir; JVM de onu dışarıdan çağırabilir
`static`	Metot sınıfa aittir; çağırmak için önce bir nesne oluşturmak gerekmez
`void`	Metot bir değer döndürmez
`main`	Metodun adı; JVM giriş noktasını bu adla arar
`String[] args`	Metodun parametresi: komut satırı argümanlarını taşıyan dizgi dizisi

Tablo 5.3: `main` metodu bildirimindeki sözcükler.

Bu sözcüklerin her biri (erişim belirleyiciler, `static` metotlar, dönüş tipleri, diziler) başlı başına bir konudur. İlk programlar için bu satırı olduğu gibi yazmak yeterlidir. Yine de her sözcüğün neden orada olduğunu bilmek, yanlış yazıldığında alınan iletiyi anlamayı kolaylaştırır.

5.4.2 main metodunun imzası

Bir metodun adı, parametreleri ve dönüş tipi onun **imzasını** oluşturur. JVM bir sınıfı çalıştırırken tam olarak `public static void main(String[] args)` imzalı bir metot arar. İlginç olan şudur: imzası yanlış yazılmış bir `main` derleme hatası vermez. Derleyici için `Main` adlı ya da `int` döndüren bir metot sıradan, geçerli bir metottur. Hata ancak `java` komutu programı başlatmaya çalışırken ortaya çıkar.

Aşağıdaki tablo, `A.java` dosyasındaki `A` sınıfında `main` farklı biçimlerde yazıldığında ne olduğunu gösterir. Her sürüm `javac A.java` ile hatasız derlendi, sonra `java A` ile çalıştırıldı.

Yazılan	`java A` ile çalıştırınca
`public static void main(String[] args)`	çalışır
`public static void Main(String[] args)`	`Error: Main method not found in class A, …`
`public void main(String[] args)`	`Error: Main method is not static in class A, …`
`public static int main(String[] args)`	`Error: Main method must return a value of type void in class A, …`
`static void main(String[] args)`	`Error: Main method not found in class A, …`

Yazılan	`java A` ile çalıştırınca
`public static void main(String a[])`	çalışır

Tablo 5.4: `main` imzasının çeşitlemeleri ve `java A` komutunun JDK 21'deki sonucu. İletilerin yalnızca ilk satırı verilmiştir.

JVM'nin iletisi aslında daha uzundur ve doğru imzayı da önerir. Örneğin `Main` büyük harfle yazıldığında tam ileti şöyledir:

```
Error: Main method not found in class A, please define the main method as:
   public static void main(String[] args)
or a JavaFX application class must extend javafx.application.Application
```

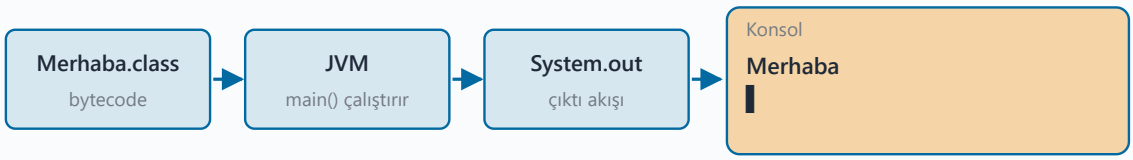
Son satırdaki JavaFX cümlesi, grafik arayüzlü JavaFX uygulamalarının başka bir başlatma yolu olduğunu hatırlatır; bu derste kullanılmaz. Tablodan üç sonuç çıkar:

- Java **büyük-küçük harfe duyarlıdır** (*case-sensitive*): `Main` ile `main` farklı adlardır.
- `public`, `static` ve `void` zorunludur. Biri eksik olduğunda JVM metodu ya hiç bulamaz ya da bulup reddeder.
- Parametrenin adı serbesttir: `args` yerine `a` yazılabilir. `String a[]` biçimindeki eski yazım da geçerlidir. Yine de Java geleneği `String[] args` yazımıdır.

Java'nın yeni sürümlerinde bu kurallar gevşetildi. Java 25'ten itibaren `public` ya da `static` olmayan, hatta parametresi olmayan bir `main` metodu da başlatılabilir. Bu ders Java 21 ve her sürümde çalışan klasik imzayı kullanır.

5.4.3 Program adım adım

`java Merhaba` komutu yazıldıktan sonra ekranda `Merhaba` görünene kadar birkaç adım geçer. Bu adımları bilmek, çalışma zamanı hatalarını ve yığın izlerini anlamanın temelidir.



Şekil 5.8: Merhaba programı çalışırken. JVM `Merhaba.class` dosyasındaki bytecode'u yükler ve `main()` metodunu çalıştırır; `System.out` çıktı akışı metni konsola taşır.

Adım	Satır	Olan	Çağrı yığını	Konsol
1	1	JVM başlar, `Merhaba.class` dosyasını belleğe yükler	boş	

Adım	Satır	Olan	Çağrı yığını	Konsol
2	2	JVM `main` metodunu çağırır; argüman verilmediği için `args` boş bir dizidir	`main(args)`	
3	3	`"Merhaba"` metni `println` metoduna gönderilir	`main(args)` → `println`	
4	3	`println` metni yazar ve satır sonu ekler; `main`'e dönülür	`main(args)`	`Merhaba` ⏎
5	4	`main` bloğunun `}` karakterine gelinir; `main` biter	boş	`Merhaba` ⏎
6	5	Çalışacak başka kod yoktur; JVM kapanır	—	`Merhaba` ⏎

Tablo 5.5: `java Merhaba` komutunun adım adım izlemesi. Çağrı yığını o anda çalışan metotları, konsol ise o ana kadarki çıktıyı gösterir (⏎ satır sonudur).

Tablodaki "çağrı yığını" sütunu, o anda hangi metotların çalıştığını gösterir. Bir metot başka bir metodu çağırdığında çağrılan metot yığının tepesine eklenir; bitince kaldırılır. 3. adımda `main`, `println`'i çağırmıştır ve ikisi birden yığındadır. `println` bitince yalnız `main` kalır. `main` de bitince yığın boşalır ve program sonlanır. Toplam 1 deyim çalışmış ve 1 satır çıktı üretilmiştir.

5.4.4 Komut satırı argümanları

`main` metodunun parametresi olan `args`, programa komut satırından bilgi vermenin yoludur. `java` komutunda sınıf adından sonra yazılan sözcüklere **komut satırı argümanları** (*command-line arguments*) denir. JVM bu sözcükleri bir dizgi dizisine koyar ve `main`'e verir.

Dizi, aynı tipte birden çok değeri sıra numarasıyla tutan yapıdır. Dizideki her hücrenin bir **indisi** vardır ve indisler 0'dan başlar. `args[0]` ilk argüman, `args[1]` ikinci argümandır. `args.length` dizideki argüman sayısını verir. Aşağıdaki program argümanları kullanır:

```
public class Selamla {
    public static void main(String[] args) {
        System.out.println(args.length);
        System.out.println("Selam " + args[0]);
        System.out.println("Selam " + args[1]);
    }
}
```

Program önce iki argümanla, sonra tek argümanla çalıştırıldı:

```
> java Selamla Ali Veli
2
Selam Ali
Selam Veli
> java Selamla Ali
1
Selam Ali
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 1 out of bounds
for length 1
        at Selamla.main(Selamla.java:5)
```



Şekil 5.9: `args` dizisinin iki çalıştırmadaki durumu. Üstte `java Selamla Ali Veli` : iki hücre dolu, `args.length = 2` . Altta `java Selamla Ali` : yalnızca `args[0]` var, `args[1]` yok; uzunluk 1'dir.

İlk çalıştırmada `args` dizisi `["Ali", "Veli"]` olur (Şekil 5.9, üstte). 3. satır argüman sayısını, yani 2'yi yazar. 4. satırda `"Selam " + args[0]` ifadesi iki dizgiyi yan yana ekler ve `Selam Ali` yazılır. 5. satır aynı biçimde `Selam Veli` yazar.

İkinci çalıştırmada kod da `.class` dosyası da aynıdır; yeniden derlemeye gerek yoktur. Değişen yalnızca girdidir. Dizide tek hücre vardır. 3. ve 4. satırlar `1` ve `Selam Ali` yazar. 5. satır var olmayan `args[1]`'e erişmeye çalışır. JVM bu durumda bir **istisna** (*exception*) **fırlatır** (*throw*): `ArrayIndexOutOfBoundsException` , yani "dizi indisi sınırların dışında". İleti, 1 numaralı indisin uzunluğu 1 olan bir dizinin dışında kaldığını söyler. Alt satır hatanın `Selamla.java` dosyasının 5. satırında, `main` metodunda oluştuğunu gösterir. Program o noktada durur.

Bu hatayı derleyici bilemezdi. Derleme anında programa kaç argüman verileceği belli değildir. Hata, çalışma sırasında ve girdiye bağlı olarak ortaya çıkar. Argüman hiç verilmezse hata bir satır önce, 4. satırda oluşur:

```
> java Selamla
0
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of bounds
for length 0
        at Selamla.main(Selamla.java:4)
```

Örnek 5.1

Aynı `Selamla` programı `java Selamla Ayşe "Ali Veli"` komutuyla çalıştırılıyor. Çıktı ne olur?

Komut satırı argümanları boşluklarla ayrılır. Çift tırnak ise içindeki boşluğu argümanın parçası yapar. Bu yüzden `"Ali Veli"` iki değil, tek bir argümandır ve tırnaklar argümana dahil değildir. `args` dizisi `["Ayşe", "Ali Veli"]` olur, uzunluğu 2'dir:

```
> java Selamla Ayşe "Ali Veli"
2
Selam Ayşe
Selam Ali Veli
```

Fazla argüman ise hata doğurmaz. `java Selamla 1 2 3` komutunda dizinin uzunluğu 3'tür. Program yalnızca ilk iki hücreyi kullandığı için üçüncüsü görmezden gelinir:

```
> java Selamla 1 2 3
3
Selam 1
Selam 2
```

5.4.5 print ve println

Konsola yazmanın iki temel yolu vardır: `System.out.print` ve `System.out.println`. İkisi de kendisine verilen değeri yazar. İkisinin farkı, yazma işi bittikten sonra ortaya çıkar. `println` (*print line*) yazdıktan sonra bir **satır sonu** (*newline*) ekler; sonraki çıktı alt satırdan başlar. `print` satır sonu eklemez; sonraki çıktı aynı satırda, kaldığı yerden sürer.

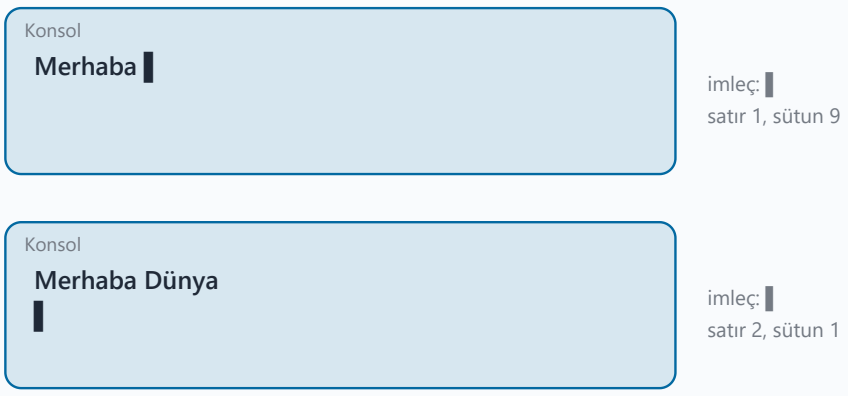
Konsolda yazının nereye geleceğini **imleç** (*cursor*) belirler. İmleci, bir sonraki karakterin yazılacağı yer olarak düşünmek yeterlidir. Aşağıdaki programda imlecin yolculuğunu izleyelim:

```
public class Selam {
    public static void main(String[] args) {
        System.out.print("Merhaba ");
        System.out.println("Dünya");
        System.out.println("Java");
    }
}
```

```
Merhaba Dünya
Java
```

Satır	Deyim	Konsol	İmleç
2	(`main` başlar)	▌	satır 1, sütun 1
3	`print("Merhaba ")`	`Merhaba` ▌	satır 1, sütun 9
4	`println("Dünya")`	`Merhaba Dünya` ▌	satır 2, sütun 1
5	`println("Java")`	`Merhaba Dünya` `Java` ▌	satır 3, sütun 1

Tablo 5.6: `Selam` programında her deyimden sonra konsol ve imlecin konumu. ▌ imleci gösterir; satır ve sütunlar 1'den başlar.



Şekil 5.10: Üstte `print("Merhaba ")` sonrası: imleç aynı satırda, boşluğun hemen arkasında (satır 1, sütun 9). Altta `println("Dünya")` sonrası: satır bitti, imleç alt satırın başında (satır 2, sütun 1).

"Merhaba " dizgisi sekiz karakterdir: yedi harf ve sondaki boşluk. `print` bu sekiz karakteri yazar ve imleç 9. sütunda kalır (Şekil 5.10). Sondaki boşluk önemlidir. O olmasaydı "Dünya" bitişik yazılır ve çıktı `MerhabaDünya` olurdu. `println("Dünya")` aynı satıra devam eder, sonra satırı bitirir. `println("Java")` yeni satıra yazılır ve o satırı da bitirir. Üç çağrı yapılmıştır, ama yalnızca iki satır oluşmuştur, çünkü satırı yalnızca `println` bitirir.

Parametresiz `System.out.println()` yalnızca bir satır sonu yazar. İmleç bir satırın ortasındaysa o satırı bitirir; satırın başındaysa boş bir satır oluşturur. `print` ise her zaman bir değer ister; parametresiz `System.out.print()` derleme hatasıdır.

Sık yapılan hata

Programın son çıktısını `print` ile yazmak. Çıktı ekranda görünür, ama satır bitmediği için konsolun komut istemi aynı satıra yapışır: `Sonuç: 80C:\ders>` gibi. Programın son satırında `println` kullanmak bu karışıklığı önler.

5.5 Kaçış dizileri ve dizgi birleştirme

Bir dizgi çift tırnakla başlar ve çift tırnakla biter. Bu basit kural iki soru doğurur. Birincisi, metnin içinde çift tırnak geçecekse ne yapılır? İkincisi, klavyeden doğrudan yazılamayan karakterler, örneğin satır sonu ya da sekme, dizgiye nasıl konur? Java iki soruyu da aynı yolla çözer: kaçış dizileri.

5.5.1 Kaçış dizileri

Tanım 5.4

Kaçış dizisi (*escape sequence*), dizgi ya da karakter sabiti içinde ters eğik çizgiyle (`\`) başlayan ve tek bir özel karakteri temsil eden yazımdır. Ters eğik çizgi, ardından gelen karakterin olağan anlamından "kaçmasını" sağlar.

Dizi	Anlamı	Örnek	Çıktı
`\n`	yeni satır (satır sonu)	`"A\nB"`	A, alt satırda B
`\t`	sekme (tab)	`"Ad\tNot"`	`Ad Not`
`\"`	çift tırnak	`"\"Java\""`	`"Java"`
`\\`	ters eğik çizgi	`"C:\\ders"`	`C:\ders`
`\'`	tek tırnak (karakter sabiti içinde)	`'\''`	`'`

Tablo 5.7: Sık kullanılan kaçış dizileri.

Kaynak koddaki `\n` iki karakter gibi görünür: ters eğik çizgi ve n harfi. Derleyici ise onu tek bir karaktere, satır sonu karakterine çevirir. Ekranda ne ters eğik çizgi ne de n görünür; yalnızca satır biter. Aynı biçimde `\"` tek bir çift tırnak karakteridir ve dizgiyi bitirmez. `\\` ise tek bir ters eğik çizgi verir. Ters eğik çizginin kendisi kaçış dizilerini başlattığı için onu yazmanın tek yolu iki kez yazmaktır.

Tek tırnak dizgi içinde sorun çıkarmaz: `"Ali'nin notu"` kaçışsız yazılır. Tek tırnak yalnızca tek bir karakteri tutan `char` sabitlerinde özel anlamlıdır. `'A'` bir karakter sabitidir; tek tırnak karakterinin kendisi `'\''` diye yazılır.

5.5.2 Sekme ve satır sonu adım adım

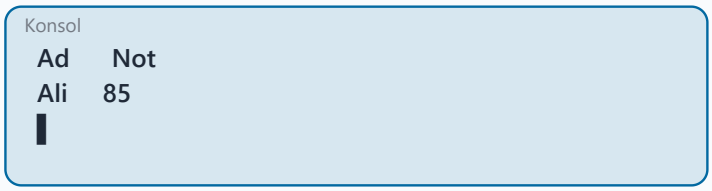
Sekme karakteri `\t` imleci bir sonraki **sekme durağına** (*tab stop*) taşır. Konsollarda sekme durakları genellikle 8 sütunda bir bulunur: 1, 9, 17, 25 … sütunlarında. Sekme, imleç hangi sütunda olursa olsun onu bir sonraki durağa götürür. Bu yüzden boyları farklı sözcüklerden sonra yazılan değerler aynı sütunda hizalanır. Aşağıdaki program bunu kullanır:

```
public class Kacis {
    public static void main(String[] args) {
        System.out.println("Ad\tNot");
        System.out.println("Ali\t85");
        System.out.println("\"Java\" dili");
        System.out.print("Satır 1\nSatır 2\n");
    }
}
```

```
Ad      Not
Ali     85
"Java" dili
Satır 1
Satır 2
```

Satır	Yazılan	Ne oldu?
3	`Ad`, sekme, `Not`, satır sonu	“Ad” 1. ve 2. sütunları kaplar; sekme imleci 9. sütuna taşır. Araya 6 boşluk girer; “Not” 9. sütundan başlar.
4	`Ali`, sekme, `85`, satır sonu	“Ali” 3 sütun kaplar; sekme imleci yine 9. sütuna taşır. Araya 5 boşluk girer; 85, “Not” başlığının tam altına gelir.
5	`"Java" dili`, satır sonu	`\"` dizgiyi bitirmeden çift tırnak yazar.
6	`Satır 1`, satır sonu, `Satır 2`, satır sonu	Tek bir `print` iki satır üretir: dizgideki iki `\n` iki satırı bitirir.

Tablo 5.8: `Kacis` programının satır satır izlemesi. Sütun numaraları 1'den başlar.



Şekil 5.11: İlk iki satırdan sonra konsol. Sekmeler 8 sütunluk duraklara hizalandığı için "Ad" ile "Ali"nin boyları farklı olsa da "Not" ve 85 aynı sütundan başlar.

Son satırdaki `print` çağrısı dikkat çekicidir. `print` kendisi satır sonu eklemez, ama dizginin içindeki `\n` karakterleri satırı bitirir. Yani `System.out.print("Satır 1\nSatır 2\n")` ile iki ayrı `println` aynı çıktıyı verir. Bir ayrıntı: `println`'in eklediği satır sonu işletim sistemine göre değişir. Windows'ta `println` iki karakter (`\r\n`) yazar, `\n` ise her yerde tek karakterdir. Konsolda ikisi aynı görünür.

Sekmeyle hizalamanın bir sınırı vardır. Bir sözcük 8 karakter ya da daha uzunsa sekme imleci bir sonraki durağa, yani 17. sütuna taşır ve hizalama bozulur. Bu yüzden sekme kısa başlıklı basit tablolar için uygundur.

Örnek 5.2

Bir öğrencinin dönem derslerini kredi ve AKTS değerleriyle hizalı bir tablo olarak yazdıran, altına da toplamları ekleyen program nasıl yazılır?

Her satır bir `println` ile yazılır ve sütunlar `\t` ile ayrılır. Toplamlar tek bir `print` ve iki `\n` ile iki satıra yazılır:

```
public class Kart {
    public static void main(String[] args) {
        System.out.println("Ders\tKredi\tAKTS");
        System.out.println("Fizik I\t4\t6");
        System.out.println("Java\t3\t5");
        System.out.print("Toplam kredi: 7\nToplam AKTS: 11\n");
    }
}
```

```
Ders    Kredi   AKTS
Fizik I 4       6
Java    3       5
Toplam kredi: 7
Toplam AKTS: 11
```

"Ders" 4, "Fizik I" 7, "Java" 4 karakterdir. Üçü de 8'den kısa olduğu için ilk sekme her satırda imleci 9. sütuna taşır ve kredi değerleri alt alta gelir. İkinci sekme 17. sütuna taşır; AKTS değerleri de hizalanır. "Fizik I" 7 karakter olduğu için ardından yalnızca 1 boşluk görünür.

5.5.3 Kaçış dizisi tuzakları

Ters eğik çizgi en çok Windows dosya yolları yazılırken sorun çıkarır. Windows klasörleri ters eğik çizgiyle ayırır: `C:\yeni`. Bu yol bir dizgiye olduğu gibi yazılırsa derleyici `\y`'yi bir kaçış dizisinin başlangıcı sayar:

```
public class Y {
    public static void main(String[] args) {
        System.out.println("C:\yeni");
    }
}
```

```
Y.java:3: error: illegal escape character
        System.out.println("C:\yeni");
                           ^
1 error
```

`\y` diye bir kaçış dizisi yoktur. Derleyici "geçersiz kaçış karakteri" (`illegal escape character`) der ve `^` ile sorunlu yeri gösterir. Program derlenmez. Daha sinsi durum, ters eğik çizgiden sonra gelen harfin geçerli bir kaçış dizisi oluşturmasıdır:



Şekil 5.12: `"C:\new"` dizgisi kaynakta 6 karakter gibi görünür, ama bellekte 5 karakterdir: C, :, yeni satır, e, w. Ekranda "C:" ve alt satırda "ew" görünür.

Deyim	Sonuç
`System.out.println("C:\yeni");`	derleme hatası: `illegal escape character`; `\y` diye bir kaçış dizisi yok
`System.out.println("C:\new");`	derlenir ama iki satır yazar: `C:` ve `ew`
`System.out.println("C:\\yeni");`	`C:\yeni`: ters eğik çizgi iki kez yazılır
`System.out.println("Ali'nin notu");`	`Ali'nin notu`: dizgi içinde tek tırnak kaçışsız yazılır
`System.out.println("\u00C7ay");`	`Çay`: `\u` ve dört onaltılık basamak bir Unicode karakteri verir
`System.out.println("C:/yeni");`	`C:/yeni`: Java'nın dosya işlemleri Windows'ta da eğik çizgiyi kabul eder

Deyim	Sonuç
`System.out.println("A\\nB");`	`A\nB` (tek satır): `\\` bir ters eğik çizgi verir, ardından gelen n sıradan bir harftir

Tablo 5.9: Kaçış dizisi tuzakları ve JDK 21'deki sonuçları.

Derlenebilen bütün örnekler, iki tablodaki sırayla, tek bir programda çalıştırıldığında çıktı şöyledir:

```
public class K {
    public static void main(String[] args) {
        System.out.println("A\nB");
        System.out.println("Ad\tNot");
        System.out.println("\"Java\"");
        System.out.println("C:\\ders");
        System.out.println('\'');
        System.out.println("C:\new");
        System.out.println("C:\\yeni");
        System.out.println("Ali'nin notu");
        System.out.println("\u00C7ay");
        System.out.println("C:/yeni");
        System.out.println("A\\nB");
    }
}
```

```
A
B
Ad      Not
"Java"
C:\ders
'
C:
ew
C:\yeni
Ali'nin notu
Çay
C:/yeni
A\nB
```

Çıktının 7. ve 8. satırları `"C:\new"` deyiminden gelir (Şekil 5.12). Program hatasız çalıştığı için bu yanlış kolay fark edilmez: derleyici uyarmaz, yalnızca çıktı tuhaftır. Son satır ise tersini gösterir: `"A\\nB"` yeni satır üretmez, ekrana ters eğik çizgi ve n harfini yazar.

`\u` ile başlayan yazıma **Unicode kaçışı** (*Unicode escape*) denir. Ardından gelen dört onaltılık basamak, karakterin Unicode numarasıdır. Ç harfinin numarası onaltılık 00C7'dir. Unicode kaçışı, klavyede bulunmayan ya da dosya kodlaması yüzünden bozulma riski taşıyan karakterleri yazmak için kullanılır:

```java
public class U {
    public static void main(String[] args) {
        System.out.println("Dağ");
        System.out.println("İstanbul");
        System.out.println("Çay");
    }
}
```

```
Dağ
İstanbul
Çay
```

Sık yapılan hata

Windows yolunu tek ters eğik çizgiyle yazmak. `"C:\ders\notlar"` derlenir mi derlenmez mi, harflere bağlıdır: `\d` geçersiz olduğu için derleme hatası alınır. `"C:\temp\new"` ise derlenir ama `\t` sekmeye, `\n` satır sonuna dönüşür. Kural basittir: dizgideki her ters eğik çizgi `\\` diye yazılır ya da yol `"C:/temp/new"` biçiminde eğik çizgiyle yazılır.

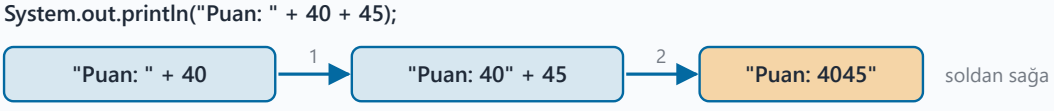
5.5.4 Dizgi birleştirme

`+` işareti Java'da iki ayrı iş yapar. İki sayı arasında **toplar**: `40 + 45` sonucu 85'tir. İşlenenlerden en az biri dizgiyse **birleştirir**: öteki işlenen metne çevrilir ve sona eklenir. `"Puan: " + 85` sonucu `"Puan: 85"` dizgisidir. Bu ikinci işe **dizgi birleştirme** (*string concatenation*) denir.

Bir ifadede birden çok `+` varsa iki kural sonucu belirler:

1. Aynı öncelikteki işlemler **soldan sağa** yapılır. `a + b + c` ifadesi `(a + b) + c` gibi değerlendirilir.
2. `*` ve `/` her zaman `+`'dan önce yapılır. Parantez ise her şeyden önce gelir.

Her `+` adımında "işlenenlerden biri dizgi mi?" sorusu yeniden sorulur. Cevap evetse birleştirme, hayırsa toplama yapılır. Dizgi bir kez ortaya çıktıktan sonra sağındaki bütün `+`'lar birleştirme olur, çünkü soldaki ara sonuç artık bir dizgidir.



Şekil 5.13: `"Puan: " + 40 + 45` soldan sağa iki birleştirmeyle değerlendirilir: önce `"Puan: " + 40` → `"Puan: 40"`, sonra `"Puan: 40" + 45` → `"Puan: 4045"`.

```java
public class B {
    public static void main(String[] args) {
        System.out.println(40 + 45 + " puan");
        System.out.println("Puan: " + 40 + 45);
        System.out.println("Puan: " + (40 + 45));
        System.out.println("Ort: " + 85 / 2);
        System.out.println(5 + 3);
        System.out.println("5 + 3");
    }
}
```

```
85 puan
Puan: 4045
Puan: 85
Ort: 42
8
5 + 3
```

İfade	Adımlar	Çıktı
`40 + 45 + " puan"`	`40 + 45` → 85 (iki `int` : toplama); `85 + " puan"` → `"85 puan"`	`85 puan`
`"Puan: " + 40 + 45`	`"Puan: " + 40` → `"Puan: 40"` ; `"Puan: 40" + 45` → `"Puan: 4045"`	`Puan: 4045`
`"Puan: " + (40 + 45)`	parantez önce: `40 + 45` → 85; `"Puan: " + 85` → `"Puan: 85"`	`Puan: 85`
`"Ort: " + 85 / 2`	`/` önce: `85 / 2` → 42 (iki tam sayı bölününce kesir atılır); `"Ort: " + 42`	`Ort: 42`
`5 + 3`	iki `int` : toplama	`8`
`"5 + 3"`	tırnak içindeki her şey metindir; hesap yapılmaz	`5 + 3`

Tablo 5.10: `+` içeren ifadelerin adım adım değerlendirilmesi.

Tablonun dördüncü satırı ayrı bir dikkat ister. `85 / 2` işleminde iki işlenen de tam sayı olduğu için sonuç da tam sayıdır. Kesirli kısım yuvarlanmaz, atılır: 42,5 değil 42 çıkar. Burada `/`'nin `+`'dan önce yapıldığına ve sonucun dizgiye 42 olarak eklendiğine dikkat edelim.

Sık yapılan hata

Bir sonucu açıklamayla birlikte yazdırırken toplamayı parantezsiz yazmak. `System.out.println("Toplam: " + a + b);` deyimi a = 40 ve b = 45 için `Toplam: 4045` yazar, çünkü en soldaki işlenen dizgidir ve her iki sayı da metne eklenir. Toplamın kendisi isteniyorsa `"Toplam: " + (a + b)` yazılmalıdır.

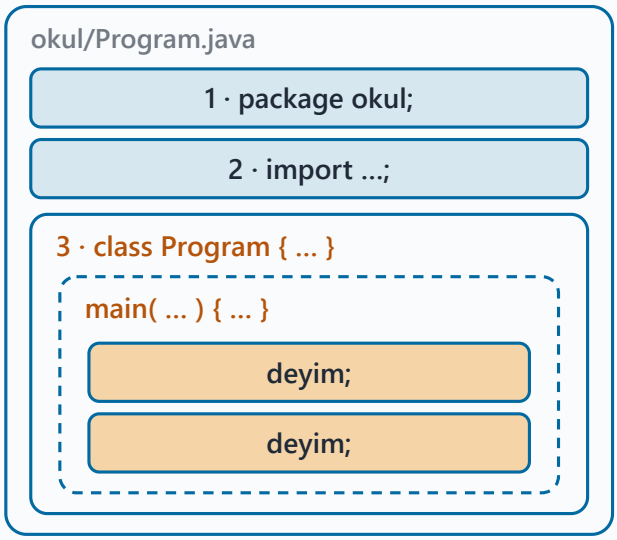
5.6 Java dosyasının yapısı

Merhaba programı tek bir sınıftan oluşuyordu. Gerçek programlarda bir dosyanın başında sınıftan önce iki tür bildirim daha bulunabilir: `package` ve `import`. Bir Java dosyasının parçaları belirli bir sırayla yazılır.

```
package okul;

import java.util.Scanner;

public class Program {
    public static void main(String[] args) {
        System.out.println("Merhaba");
    }
}
```



Şekil 5.14: `okul/Program.java` dosyasının iç içe yapısı: 1. `package` bildirimi, 2. `import` bildirimleri, 3. sınıf. Sınıfın içinde `main` metodu, metodun içinde deyimler bulunur.

Şekil 5.14 yapıyı gösterir. Sıra şöyledir:

1. `package okul;` sınıfın hangi **pakete** (*package*) ait olduğunu söyler. Paket, birbiriyle ilgili sınıfları bir arada tutan ad alanıdır; bilgisayardaki klasörlere benzer. Paket bildirimi varsa dosyanın ilk deyimi olmak zorundadır ve dosya paket adıyla aynı adlı bir klasörde durur: `okul/Program.java`.
2. `import java.util.Scanner;` başka bir paketteki bir sınıfı kısa adıyla kullanmayı sağlar. Bu satır olmasaydı sınıf her yerde tam adıyla, `java.util.Scanner` diye yazılmalıydı. `import` bildirimleri paket bildiriminden sonra, sınıftan önce gelir. Bu örnekte `Scanner` kullanılmamıştır; yalnızca yeri göstermek için yazılmıştır. Kullanılmayan bir `import` hata vermez.

3. `public class Program { … }` sınıfın kendisidir. Public sınıfın adı dosya adıyla aynıdır.

`System` sınıfı için `import` gerekmediğine dikkat edelim. `System`, `String` gibi en temel sınıflar `java.lang` paketindedir ve bu paket her dosyaya kendiliğinden eklenir.

Paketli bir sınıf derlenirken ve çalıştırılırken paket klasörünün bir üstündeki klasörde durulur. Çalıştırırken sınıfın paket adıyla birlikte tam adı yazılır:

```
> javac okul/Program.java
> java okul.Program
Merhaba
> java Program
Error: Could not find or load main class Program
Caused by: java.lang.ClassNotFoundException: Program
```

Son komutta yalnızca `Program` yazıldığında JVM, paketsiz bir `Program` sınıfı arar ve bulamaz. Sınıfın gerçek adı `okul.Program`'dır. İlk programlarda paket kullanılmaz; sınıflar varsayılan, adsız pakette durur. IDE'ler ise yeni bir proje oluştururken çoğu zaman bir paket bildirimi ekler. Bir IDE'de yazılmış dosya komut satırında derlenirken bu ayrıntı unutulmamalıdır.

5.7 Derleme ve çalıştırma

Bir Java programı yazıldığı anda çalışmaz. Önce derlenir, sonra çalıştırılır. Bu iki adım iki ayrı komutla yapılır ve iki ayrı dosya türüyle ilgilidir: derleyici `.java` dosyasını okur, JVM ise `.class` dosyasını çalıştırır. Bu ayrımı iyi kavramak, yeni başlayanların en sık takıldığı sorunların çoğunu önler.

5.7.1 Derleme hattı

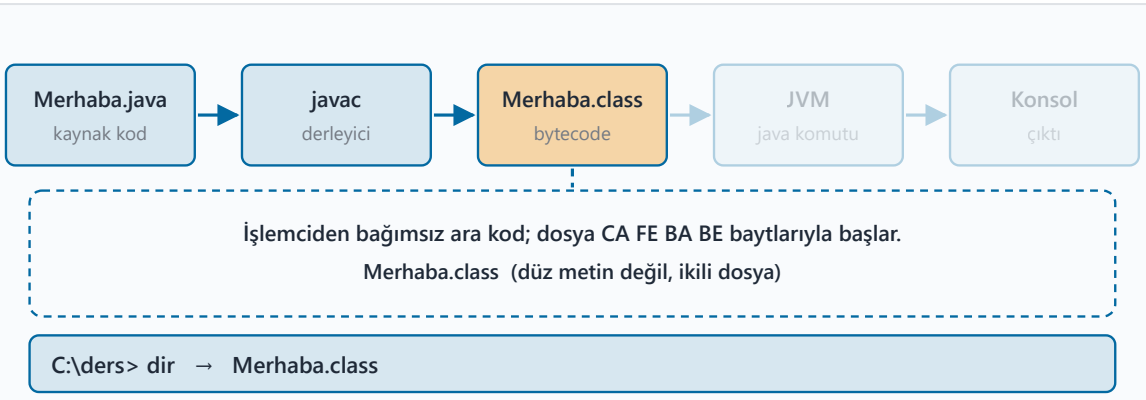
Tanım 5.5

Derleme (*compilation*), kaynak kodun bütünüyle denetlenip başka bir dile çevrilmesidir. Java'da derleyici `javac`, kaynak kodu **bytecode** adı verilen ve işlemciden bağımsız olan bir ara koda çevirir. Bytecode `.class` uzantılı dosyalarda saklanır ve Java Sanal Makinesinde çalışır.

Bir Java programının yazılmasından çıktısının görünmesine kadar beş adım vardır. Bu adımların hepsine birlikte **derleme hattı** diyelim:

1. **Yaz.** Kaynak kod bir düzenleyicide yazılır ve `Merhaba.java` adıyla kaydedilir. Dosya adı public sınıfın adıyla aynı olmalıdır.
2. **Derle.** `javac Merhaba.java` komutu derleyiciyi çalıştırır. Derleyici dosyanın tamamını, program daha çalışmadan önce okur; söz dizimini ve tipleri denetler.

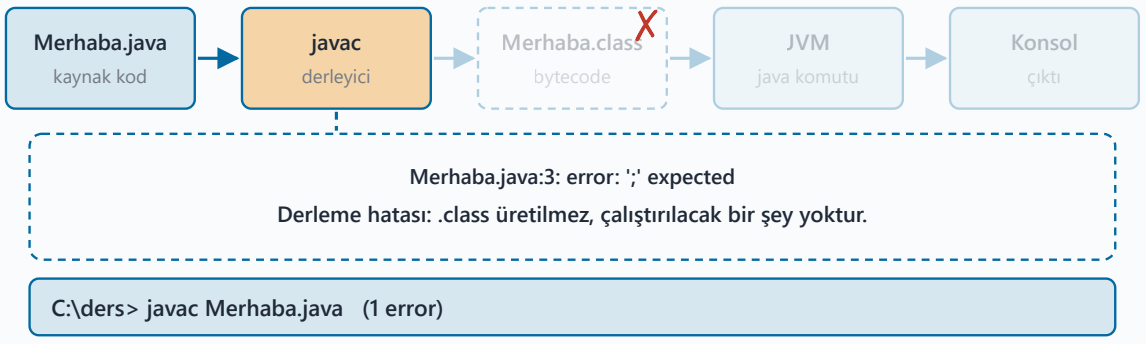
3. **Bytecode.** Derleme başarılıysa aynı klasörde `Merhaba.class` dosyası oluşur. Bu dosya düz metin değil, ikili bir dosyadır. Windows'ta, Linux'ta ve macOS'ta aynıdır.
4. **Çalıştır.** `java Merhaba` komutu JVM'yi başlatır. Bu komuta dosya adı değil, sınıf adı verilir; `.class` uzantısı yazılmaz. JVM'nin sınıf yükleyicisi `.class` dosyasını belleğe alır, doğrulayıcı bytecode'u denetler, sonra `main` çağrılır.
5. **Çıktı.** JVM bytecode'u yorumlar; sık çalışan kısımları JIT derleyicisiyle makine koduna çevirir. `println` konsola `Merhaba` yazar.



Şekil 5.15: Derleme hattının üçüncü adımı. `javac` başarılı olunca klasörde `Merhaba.class` oluşur. Bu dosya işlemciden bağımsız bir ara koddur ve ilk dört baytı onaltılık CA FE BA BE'dir.

Her `.class` dosyası aynı dört baytla başlar: onaltılık `CA FE BA BE`. Bu "sihirli sayı", JVM'nin dosyanın gerçekten bir Java sınıf dosyası olduğunu anlamasını sağlar. Bu notta derlenen `Merhaba.class` dosyasının ilk sekiz baytı `ca fe ba be 00 00 00 41`'dir. Son iki bayt sınıf dosyasının sürüm numarasıdır: onaltılık 41, onluk 65'tir ve Java 21'i gösterir. Eski bir JVM'nin yeni bir sınıf dosyasını reddederken andığı "class file version 65.0" bu sayıdır.

Derleyici bir hata bulursa hat ikinci adımda durur:



Şekil 5.16: Derleme hatası olduğunda: `javac` hatayı satır numarasıyla bildirir (`Merhaba.java:3: error: ';' expected`) ve `.class` dosyası üretmez. Çalıştırılacak bir şey yoktur.

Derleme hatası olan bir program hiç çalışmaz; bir kısmı da çalışmaz. Derleyici bütün dosyayı denetler ve tek bir hata bile bulsa `.class` üretmez. Bu, satır satır çalışan yorumlanan dillerden önemli bir farktır. Python gibi bir dilde hatalı satırdan önceki satırlar çalışabilir. Java'da hatanın 50. satırda olması, 1. satırın da çalışmayacağı anlamına gelir.

5.7.2 JDK kurulumu ve komutlar

Java programı yazmak için önce bir JDK kurulur. Kurulumdan sonra komut satırında birkaç temel komut kullanılır:

Komut	Ne yapar?
`java -version`	kurulu Java sürümünü gösterir
`javac -version`	derleyici sürümünü gösterir (`javac` yalnız JDK'de bulunur)
`javac Merhaba.java`	derler: `Merhaba.class` üretir
`java Merhaba`	çalıştırır (uzantı yazılmaz)
`java Merhaba.java`	Java 11 ve sonrası: tek dosyalık programı derleyip çalıştırır, `.class` dosyası bırakmaz

Tablo 5.11: Temel JDK komutları.

Kurulumdan sonra ilk iş, iki sürüm komutunu çalıştırıp aynı sürümü gösterdiklerini görmektir. Bu notun bilgisayarında JDK 21 komutlarının çıktısı şöyledir:

```
> 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
```

Tablonun son satırındaki kullanım kısa denemeler için pratiktir. `java` komutuna bir `.java` dosyası verilirse JVM dosyayı bellekte derler ve hemen çalıştırır:

```
> java Merhaba.java
Merhaba
```

Bu yol tek dosyalık programlar için çalışır ve diskte `.class` dosyası bırakmaz. Derleme ile çalıştırmanın iki ayrı adım olduğunu görmek için bu notta iki adımlı yol kullanılır.

PATH ve JAVA_HOME. Komut satırına `javac` yazıldığında işletim sistemi bu adlı programı `PATH` **ortam değişkenindeki** (*environment variable*) klasörlerde arar. JDK'nin `bin` klasörü `PATH`'te değilse "komut bulunamadı" hatası alınır. Çözüm, JDK'nin `bin` klasörünü, örneğin `C:\Program`

`Files\Eclipse Adoptium\jdk-21.0.3.9-hotspot\bin` klasörünü, `PATH`'e eklemektir. `JAVA_HOME` ortam değişkeni ise JDK'nin ana klasörünü gösterir; derleme araçları ve IDE'ler Java'yı bulmak için bu değişkene bakar.

Windows konsolunda Türkçe. Türkçe harfler iki yerde bozulabilir. Birincisi derleme sırasıdır: derleyici dosyayı yanlış kodlamayla okursa harfler daha bytecode'a girerken bozulur. JDK 18 ve sonrasında derleyicinin varsayılan kodlaması UTF-8'dir; daha eski sürümlerde `javac -encoding UTF-8 Merhaba.java` yazmak gerekir. İkincisi konsolun kendisidir. Program doğru çalıştığı hâlde ekranda harfler bozuk görünüyorsa konsolun kod sayfası UTF-8'e çevrilebilir: önce `chcp 65001` komutu çalıştırılır.

5.7.3 Derle, değiştir, yeniden derle

Yeni başlayanları en çok şaşırtan durumlardan biri şudur: kaynak kod değiştirilir, program çalıştırılır, ama eski çıktı görünür. Nedeni, `java` komutunun kaynak koda hiç bakmamasıdır. JVM yalnızca `.class` dosyasındaki bytecode'u çalıştırır. Aşağıdaki oturumda klasörde başta yalnızca `Merhaba.java` vardır ve program `Merhaba` yazar.

```
> java Merhaba
Error: Could not find or load main class Merhaba
Caused by: java.lang.ClassNotFoundException: Merhaba
> javac Merhaba.java
> java Merhaba.class
Error: Could not find or load main class Merhaba.class
Caused by: java.lang.ClassNotFoundException: Merhaba.class
> java Merhaba
Merhaba
```

Bu kısımda üç ayrı durum görülür. İlk komutta henüz `Merhaba.class` yoktur; JVM `Merhaba` adlı sınıfı bulamaz. İkinci satırdaki `Caused by`, hatanın altta yatan nedenini verir: `ClassNotFoundException`, yani sınıf bulunamadı. Derleme sessizce başarılı olur. Üçüncü komutta `java`'ya dosya adı verilmiştir. JVM "Merhaba.class" adlı bir sınıf arar. Java'da sınıf adları nokta içeremediği için bunu `class` adlı bir sınıf gibi yorumlar ve bulamaz. Uzantısız doğru komut programı çalıştırır.

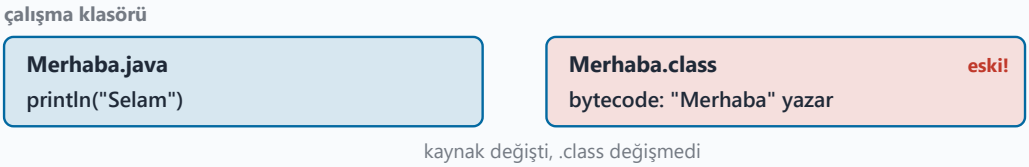
Şimdi düzenleyicide `"Merhaba"` metni `"Selam"` olarak değiştirilip dosya kaydedilir ve oturum sürer:

```
> java Merhaba
Merhaba
> javac Merhaba.java
> java Merhaba
Selam
```

Adım	Komut ya da işlem	`Merhaba.java`	`Merhaba.class`	Çıktı
1	`javac Merhaba.java`	`println("Merhaba")`	"Merhaba" yazar (yeni)	—

Adım	Komut ya da işlem	`Merhaba.java`	`Merhaba.class`	Çıktı
2	`java Merhaba`	`println("Merhaba")`	"Merhaba" yazar	`Merhaba`
3	düzenle ve kaydet	`println("Selam")`	"Merhaba" yazar (eski!)	—
4	`java Merhaba`	`println("Selam")`	"Merhaba" yazar (eski!)	`Merhaba`
5	`javac Merhaba.java`	`println("Selam")`	"Selam" yazar (yeni)	—
6	`java Merhaba`	`println("Selam")`	"Selam" yazar	`Selam`

Tablo 5.12: Oturum boyunca iki dosyanın içeriği. Kaynak değiştiği hâlde `.class` ancak yeniden derlenince güncellenir.



Şekil 5.17: Kaynak düzenlendikten sonra klasör: `Merhaba.java` artık `println("Selam")` içerir, ama `Merhaba.class` hâlâ "Merhaba" yazan eski bytecode'dur. Kaynak değişti, `.class` değişmedi.

Tablo 5.12 ve Şekil 5.17 aynı sonucu gösterir: `.class` dosyası kendiliğinden güncellenmez. Dördüncü adımda çıktı hâlâ `Merhaba`'dır. Kaynak kodda yapılan her değişiklikten sonra yeniden derlemek gerekir.

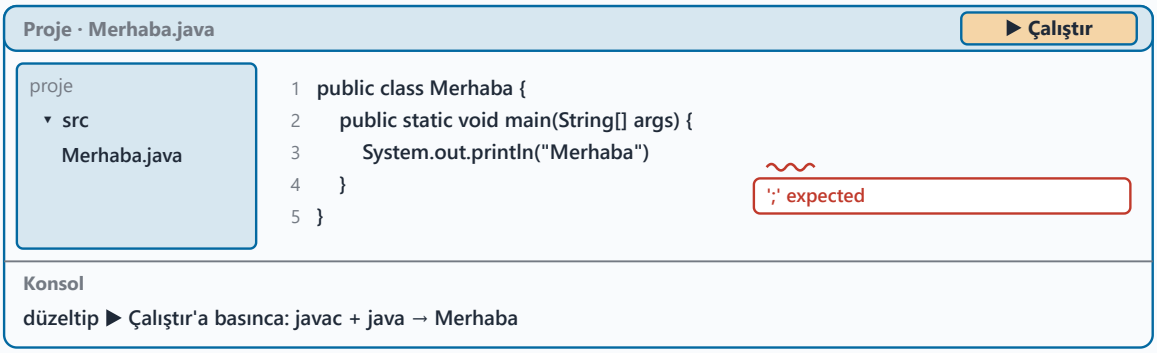
Sınıf adındaki harf büyüklüğü de önemlidir. Windows dosya adlarında büyük-küçük harf ayırmaz, Java ise ayırır. `java merhaba` komutunda Windows `Merhaba.class` dosyasını bulur, ama JVM dosyanın içindeki sınıfın adının `merhaba` değil `Merhaba` olduğunu görür ve reddeder:

```
> java merhaba
Error: Could not find or load main class merhaba
Caused by: java.lang.NoClassDefFoundError: merhaba (wrong name: Merhaba)
```

İletideki `wrong name: Merhaba` ifadesi "yanlış ad, doğrusu Merhaba" demektir.

5.7.4 IDE ile çalışmak

Komut satırında derleme ve çalıştırma, arka planda olanı açıkça gösterir. Günlük çalışmada ise çoğu programcı bir IDE kullanır. IDE aynı adımları bir düğmenin arkasına gizler.



Şekil 5.18: Bir IDE penceresi. Solda proje ağacı, ortada düzenleyici, altta konsol. 3. satırdaki eksik noktalı virgül daha derlemeden kırmızı dalgalı çizgiyle işaretlenmiş ve derleyicinin iletisi (`';' expected`) gösterilmiştir. Hata düzeltilip ▶ Çalıştır düğmesine basılınca IDE önce derler, sonra çalıştırır.

IDE'nin sağladığı başlıca kolaylıklar şunlardır:

- ▶ **Çalıştır düğmesi** derleme ve çalıştırma adımlarını arka arkaya yapar. Değiştirilen dosyalar önce yeniden derlenir; bu yüzden eski `.class` sorunu yaşanmaz.
- **Anında hata gösterimi**: IDE kodu yazılırken arka planda derler. Derleme hataları kırmızı alt çizgiyle işaretlenir; imleç üzerine gelince derleyicinin iletisi görünür.
- **Kod tamamlama**: `System.out.` yazıldığında kullanılabilecek metotlar listelenir. Yanlış yazılmış metot adları böylece azalır.
- **Otomatik girinti** ve **yeniden adlandırma**: bir değişkenin adı değiştirildiğinde geçtiği her yer birlikte değişir.
- **Hata ayıklayıcı** (*debugger*): programı bir **kesme noktasında** (*breakpoint*) durdurur, değişkenlerin o anki değerlerini gösterir ve programı satır satır ilerletir.

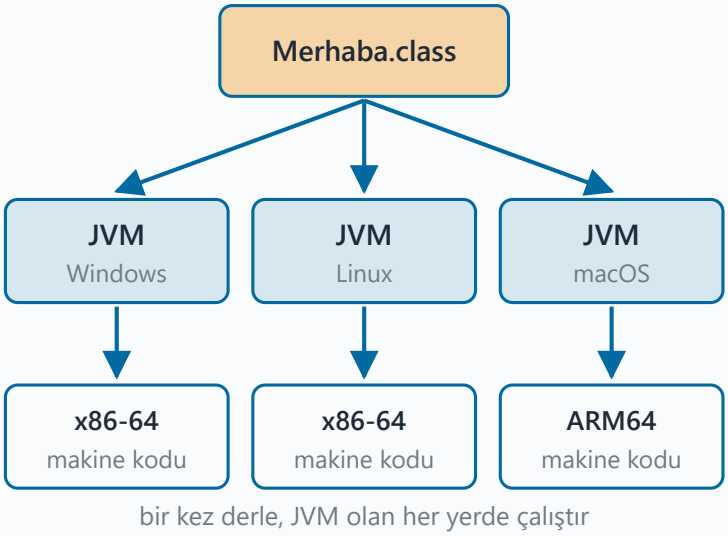
Yaygın Java IDE'leri IntelliJ IDEA, Eclipse ve NetBeans'tir. VS Code da Java eklentileriyle aynı işleri yapar. IDE büyük kolaylık sağlar, ama arka planda olanı bilmek yine gereklidir. Sunucularda ya da uzak bağlantılarda çoğu zaman IDE yoktur; yalnızca komut satırı vardır. IDE'nin gösterdiği iletiler de `javac`'ın iletileridir. Komut satırındaki derleme adımlarını bilen biri bu iletileri her ortamda okuyabilir.

5.8 Bytecode ve JVM

Java'nın "bir kez derle, her yerde çalıştır" ilkesi bytecode sayesinde mümkündür. Bu kesim bytecode'un neden işlemciden bağımsız olduğunu, bir `.class` dosyasının içinde ne bulunduğunu ve JVM'nin bytecode'u nasıl yürüttüğünü anlatır.

5.8.1 Bir kez derle, her yerde çalıştır

C gibi dillerde derleyici, kaynak kodu doğrudan belirli bir işlemcinin makine koduna çevirir. Intel işlemcili bir Windows bilgisayar için derlenen program, ARM işlemcili bir Mac'te çalışmaz; orası için ayrıca derlenmesi gerekir. Java derleyicisi ise hiçbir gerçek işlemcinin makine kodunu üretmez. Var olmayan, hayali bir işlemcinin, yani Java Sanal Makinesinin komutlarını üretir. Bu komutlara **bytecode** denir.



Şekil 5.19: Aynı `Merhaba.class` dosyası Windows (x86-64), Linux (x86-64) ve macOS (ARM64) üzerindeki JVM'lerde çalışır. Her JVM bytecode'u kendi işlemcisinin makine koduna çevirir.

Hayali işlemciyi gerçek kılan, her platform için ayrı yazılmış JVM'dir (Şekil 5.19). Windows'un JVM'si de, macOS'un JVM'si de aynı bytecode'u okur. Her biri onu kendi işlemcisinde çalıştırır. Bunun için iki yol kullanılır. Yorumlayıcı bytecode komutlarını tek tek okuyup yürütür. JIT derleyicisi ise sık çalışan kısımları, örneğin binlerce kez dönen bir döngüyü, o işlemcinin makine koduna çevirir ve sonraki turlarda doğrudan makine kodunu çalıştırır. Böylece Java programları hem taşınabilir hem de hızlı olur.

Sınıf yükleme, bytecode doğrulama ve çöp toplama da çalışma ortamının parçalarıdır. Doğrulayıcı özellikle önemlidir. Bir `.class` dosyası ağdan indirilmiş ya da bozulmuş olabilir. Doğrulayıcı, kodun JVM'nin kurallarına uyduğunu çalıştırmadan önce denetler.

5.8.2 Bytecode'un içi: javap

`.class` dosyası ikili bir dosyadır; Not Defteri'yle açılırsa anlamsız karakterler görünür. İçindeki komutları okumak için JDK ile gelen `javap` aracı kullanılır. `-c` seçeneği her metodun bytecode komutlarını okunur biçimde listeler:

```
> javap -c Merhaba
Compiled from "Merhaba.java"
public class Merhaba {
  public Merhaba();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #7                  // Field
java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #13                 // String Merhaba
       5: invokevirtual #15                 // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
       8: return
}
```

Çıktıda iki metot vardır. İlki `public Merhaba();`, yani sınıfın **kurucusudur** (*constructor*). Kaynak kodda böyle bir metot yazılmamıştı. Bir sınıfa hiç kurucu yazılmazsa derleyici boş bir kurucuyu kendisi ekler. İkinci metot `main`'dir ve dört komuttan oluşur:

Konum	Komut	Ne yapar?
0	`getstatic`	`System.out` nesnesinin başvurusunu işlenen yığınına koyar
3	`ldc`	`"Merhaba"` sabitini işlenen yığınına koyar
5	`invokevirtual`	yığından iki değeri alır ve `println`'i çağırır
8	`return`	`main`'i bitirir

Tablo 5.13: `Merhaba.main` metodunun bytecode komutları.

Komutların solundaki sayılar, komutun metot içindeki **bayt konumudur**. Her komutun işlem kodu (*opcode*) tek bir bayttır; "bytecode" adı da buradan gelir. Bazı komutların ardından işlenen baytları gelir. `getstatic` 3 bayt tutar (1 işlem kodu, 2 işlenen); bu yüzden sonraki komut 3. konumdadır. `ldc` 2 bayt tutar ve `invokevirtual` 5. konumdan başlar. O da 3 bayt tutar; `return` 8. konumdadır. `#7`, `#13` gibi işaretler, sınıf dosyasındaki **sabit havuzunun** (*constant pool*) giriş numaralarıdır. `javap` her girişin ne olduğunu sağdaki açıklamada yazar: `#13`, "Merhaba" dizgisidir.

JVM bir **yığın makinesidir** (*stack machine*). Gerçek işlemciler hesap yaparken değerleri yazmaç adı verilen küçük bellek hücrelerinde tutar. JVM ise değerleri bir **işlenen yığınına** (*operand stack*) koyar ve işlemleri yığının tepesindeki değerlerle yapar. Yığın, üst üste konmuş tabaklara benzer: en son konan değer en önce alınır. Yazmaç kullanmamak bytecode'u işlemciden bağımsız tutar, çünkü her işlemcinin yazmaç sayısı farklıdır.

Kaynak koddaki açıklamalar ve boşluklar bytecode'a girmez. Bol açıklamalı bir Merhaba programı da (5.10) aynı dört komuta derlenir.

5.8.3 JVM bytecode'u yürütüyor

Yığın makinesinin nasıl çalıştığını görmek için biraz daha uzun bir programı izleyelim. Program iki sayıyı toplar ve sonucu yazar:

```
public class Topla {
    public static void main(String[] args) {
        int a = 3;
        int b = 4;
        int c = a + b;
        System.out.println(c);
    }
}
```

```
7
```

`javap -c Topla` komutu `main` metodu için şu komutları verir:

```
 0: iconst_3
 1: istore_1
 2: iconst_4
 3: istore_2
 4: iload_1
 5: iload_2
 6: iadd
 7: istore_3
 8: getstatic     // System.out
11: iload_3
12: invokevirtual // println(int)
15: return
```

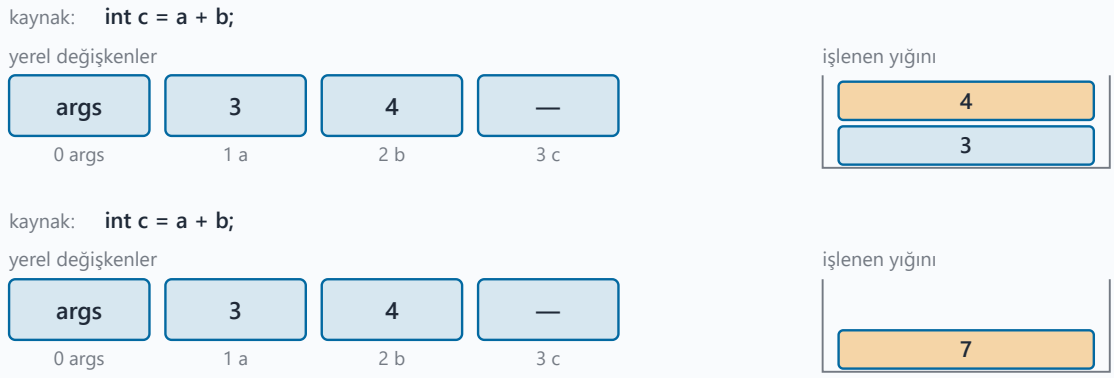
Kaynaktaki dört satır 12 bytecode komutuna dönüşmüştür. Komutların adları bir düzene uyar. Baştaki `i` harfi `int` anlamına gelir. `iconst_3` yığına 3 sabitini koyar. `istore_1` yığının tepesindeki değeri alıp 1 numaralı yerel değişkene yazar. `iload_1` 1 numaralı yerel değişkenin değerini yığına kopyalar. `iadd` yığından iki değer alır, toplar ve sonucu yığına koyar.

Metodun **yerel değişkenleri** (*local variables*) numaralı yuvalarda tutulur. 0 numaralı yuvada `main`'in parametresi `args` durur. Sonraki yuvalar kaynak koddaki sırayla verilir: 1 numara `a`, 2 numara `b`, 3 numara `c`'dir. Aşağıdaki tablo her komuttan sonra yuvaları ve işlenen yığınını gösterir.

Konum	Komut	1 (a)	2 (b)	3 (c)	İşlenen yığını	Konsol
0	`iconst_3`	—	—	—	[3]	
1	`istore_1`	3	—	—	[ ]	
2	`iconst_4`	3	—	—	[4]	

Konum	Komut	1 (a)	2 (b)	3 (c)	İşlenen yığını	Konsol
3	`istore_2`	3	4	—	[ ]	
4	`iload_1`	3	4	—	[3]	
5	`iload_2`	3	4	—	[3, 4]	
6	`iadd`	3	4	—	[7]	
7	`istore_3`	3	4	7	[ ]	
8	`getstatic`	3	4	7	[System.out]	
11	`iload_3`	3	4	7	[System.out, 7]	
12	`invokevirtual`	3	4	7	[ ]	`7`
15	`return`	3	4	7	[ ]	`7`

Tablo 5.14: `Topla.main` bytecode'unun izleme tablosu. Yığın soldan sağa yazılmıştır; en sağdaki değer tepededir. — boş yuvayı gösterir.



Şekil 5.20: `iadd` komutundan önce ve sonra. Üstte `iload_1` ve `iload_2` sonrası: yığında 3 ve 4 var. Altta `iadd` sonrası: iki değer alındı, toplamları 7 yığına kondu. Yerel değişkenler bu sırada değişmez.

İzlemede bir düzen göze çarpar. `int c = a + b;` satırı dört komuta dönüşür: iki yükleme, bir toplama, bir saklama. Toplama komutu `iadd` değişken adlarını hiç bilmez; yalnızca yığının tepesindeki iki değerle çalışır. Değerleri yığına getirmek `iload`, yığından değişkene götürmek `istore` komutlarının işidir (Şekil 5.20).

`System.out.println(c);` satırı da aynı düzeni izler. Önce `println`'in çağrılacağı nesne (`System.out`), sonra yazdırılacak değer (7) yığına konur. `invokevirtual` ikisini birden yığından alır ve `println(int)` metodunu çağırır. Konsola 7 yazılır. `return` ile `main` biter. Metodun çerçevesi kaldırılır ve yerel değişkenler ile işlenen yığını da çerçeveyle birlikte yok olur.

Komutların konumlarındaki boşluklar da açıklanabilir. 0'dan 7'ye kadar her komut tek bayttır, çünkü `iconst_3`, `istore_1` gibi komutlar işlenenlerini adlarında taşır. `getstatic` 3 bayt tuttuğu için sonraki komut 11. konumdadır. `iload_3` 1 bayt, `invokevirtual` 3 bayttır ve `return` 15. konuma düşer.

Örnek 5.3

`Topla` programında 3 yerine 5, 4 yerine 2 yazılır ve toplama yerine çarpma yapılırsa bytecode nasıl değişir?

```
public class Carp {
    public static void main(String[] args) {
        int a = 5;
        int b = 2;
        int c = a * b;
        System.out.println(c);
    }
}
```

Komutların yapısı aynı kalır; yalnızca sabitler ve işlem değişir. `iconst_3` yerine `iconst_5`, `iconst_4` yerine `iconst_2`, `iadd` yerine çarpma komutu `imul` gelir. `javap -c Carp` bu tahmini doğrular:

```
public static void main(java.lang.String[]);
  Code:
     0: iconst_5
     1: istore_1
     2: iconst_2
     3: istore_2
     4: iload_1
     5: iload_2
     6: imul
     7: istore_3
     8: getstatic     #7                  // Field
java/lang/System.out:Ljava/io/PrintStream;
    11: iload_3
    12: invokevirtual #13                 // Method java/io/PrintStream.println:
(I)V
    15: return
```

Program 10 yazar. Yığın, `imul` 'dan hemen önce [5, 2], hemen sonra [10]'dur.

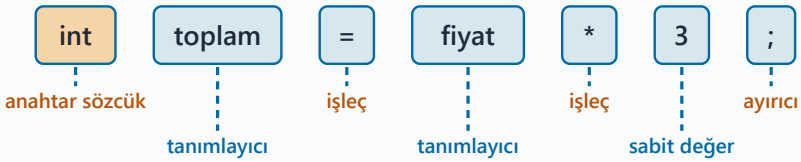
5.9 Dilin yapı taşları

Derleyici bir Java dosyasını okurken onu önce küçük parçalara ayırır, sonra bu parçalardan daha büyük yapılar kurar. Harflerden sözcükler, sözcüklerden cümleler, cümlelerden paragraflar oluşması gibi. Bu kesim en küçük parçalardan, belirteçlerden başlar ve ifade, deyim ve blok kavramlarına ulaşır.

5.9.1 Belirteçler

Tanım 5.6

Belirteç (*token*), kaynak kodun derleyici için anlamlı olan en küçük parçasıdır. Java'da beş tür belirteç vardır: **anahtar sözcük** (*keyword*), **tanımlayıcı** (*identifier*), **sabit değer** (*literal*), **işleç** (*operator*) ve **ayırıcı** (*separator*).



Şekil 5.21: `int toplam = fiyat * 3;` deyiminin yedi belirteci: `int` anahtar sözcük, `toplam` ve `fiyat` tanımlayıcı, `=` ve `*` işleç, `3` sabit değer, `;` ayırıcı.

Beş türü Şekil 5.21 üzerinden tanıyalım:

- **Anahtar sözcükler** dilin kendisine ayrılmış sözcüklerdir: `int`, `class`, `public`, `static`, `void`, `if` gibi. Anlamları sabittir ve başka bir amaçla kullanılamazlar. `int class = 5;` yazılamaz.
- **Tanımlayıcılar** programcının verdiği adlardır: değişken, metot ve sınıf adları. `toplam`, `fiyat`, `Merhaba`, `main` birer tanımlayıcıdır. Bir tanımlayıcı harfle, `_` ya da `$` ile başlar; sonra harf ve rakamlarla sürer. Boşluk içeremez ve rakamla başlayamaz.
- **Sabit değerler** kodun içine doğrudan yazılmış değerlerdir: `3`, `120`, `2.5`, `"Merhaba"`, `'A'`, `true`.
- **İşleçler** işlem yapan işaretlerdir: `+`, `-`, `*`, `/`, `=`, `>` gibi.
- **Ayırıcılar** yapıyı belirleyen işaretlerdir: `;` `,` `.` `(` `)` `{` `}` `[` `]`.

Java **büyük-küçük harfe duyarlı** bir dildir. `sayi` ile `Sayi` iki ayrı tanımlayıcıdır. `System` doğru, `system` yanlıştır. `Int` diye bir anahtar sözcük yoktur; yalnızca `int` vardır. Deyimler çoğunlukla noktalı virgülle biter; bloklar süslü parantezle çevrilir.

5.9.2 Boşluk, girinti ve süslü parantez

Aşağıdaki iki program aynıdır. İlki tek satıra yazılmıştır, ikincisi Java geleneğine göre girintilenmiştir:

```
public class Tek { public static void main(String[] args) { System.out.println("Merhaba");
} }
```

```
public class Tek {
    public static void main(String[] args) {
        System.out.println("Merhaba");
    }
}
```

İki program da derlenir ve aynı bytecode'a dönüşür. Tek satırlık sürümün `javap -c` çıktısındaki `main` komutları, Merhaba programındaki dört komutun aynısıdır: `getstatic`, `ldc`, `invokevirtual`, `return`. Derleyici için boşluk, sekme ve satır sonu yalnızca belirteçleri birbirinden ayırır. `x=1;` ile `x = 1;` aynıdır. Ama `publicclass` tek bir sözcük sayılır ve iki belirteç olarak okunmaz. (İki sınıf dosyası bayt bayt aynı değildir, çünkü `.class` dosyası hata iletileri için satır numaralarını da saklar. Komutlar ise aynıdır.)

Boşluk derleyiciyi ilgilendirmiyorsa girinti neden önemlidir? Çünkü kod derleyici için olduğu kadar insan için de yazılır. **Girinti** (*indentation*), blokların iç içeliğini okuyana gösterir. Java geleneğinde kurallar şunlardır:

- Her yeni blok bir öncekinden 4 boşluk içeriden başlar.
- Açılış süslü parantezi `{` satırın sonunda durur.
- Kapanış süslü parantezi `}` kendi satırındadır ve açtığı satırın hizasına döner.

Bu kurallara uyulduğunda bir `}`'nin hangi bloğu kapattığı bir bakışta görülür. Girintisi bozuk kodda bir süslü parantezin eksik olduğunu fark etmek çok zorlaşır.

Bir istisna vardır: dizgi içindeki boşluklar metnin parçasıdır. `"Merhaba "` ile `"Merhaba"` farklı dizgilerdir; ilki sekiz, ikincisi yedi karakterdir. `print("Merhaba ")` ile `print("Merhaba")` farklı çıktı verir.

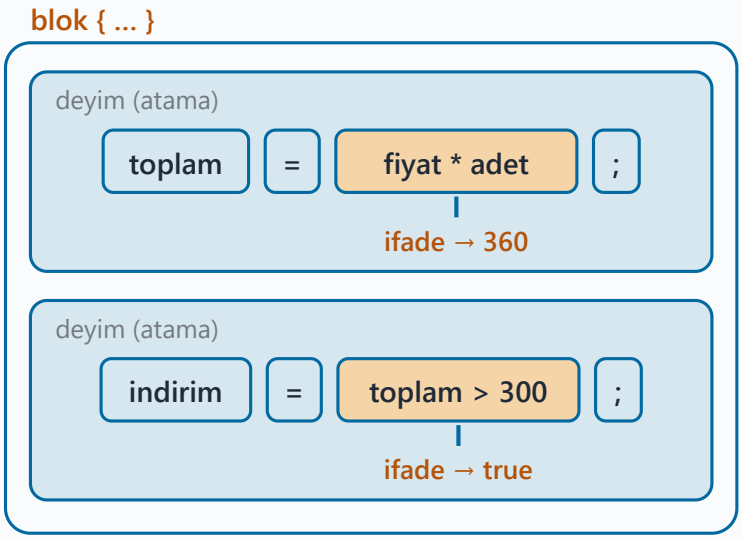
5.9.3 İfade, deyim ve blok

Belirteçler bir araya gelerek ifadeleri, ifadeler de deyimleri oluşturur.

Tanım 5.7

İfade (*expression*), değerlendirildiğinde bir değer üreten kod parçasıdır: `2 + 3` → 5, `sayi > 0` → `true` ya da `false`. **Deyim** (*statement*), bir işlemi tamamlayan kod birimidir ve çoğunlukla `;` ile biter: `toplam = 5;`, `System.out.println(toplam);`. **Blok** (*block*), süslü parantez içindeki deyimleri tek bir yapı olarak gruplar.

Fark şu soruyla anlaşılır: "Bu parça bir değer mi, yoksa bir iş mi?" `fiyat * adet` bir değerdir (360 gibi); tek başına bir şey yapmaz. `toplam = fiyat * adet;` ise bir iştir: değeri hesaplar ve `toplam`'a yazar. Bir deyimin içinde bir ya da daha çok ifade bulunabilir.



Şekil 5.22: Blok deyimleri, deyimler ifadeleri içerir. Bloktaki ilk atama deyiminin ifadesi `fiyat * adet` 360 değerini, ikincisinin ifadesi `toplam > 300` `true` değerini üretir.

Aşağıdaki kod parçası dört kavramı birlikte kullanır. Parça bir sınıfın `main` metoduna konup çalıştırıldığında konsola tek satır yazılır:

```
int fiyat = 120;           // tanımlama deyimi
int adet = 3;              // tanımlama deyimi
int toplam = fiyat * adet; // ifade + atama
if (toplam > 300) {        // koşul + blok
    System.out.println("İndirim");
}
```

```
İndirim
```

Adım	Satır	Olan	fiyat	adet	toplam	Konsol
1	1	`fiyat` için bellekte `int` kutusu ayrılır, 120 yazılır	120	?	?	
2	2	`adet` kutusu oluşur	120	3	?	
3	3	sağdaki ifade değerlendirilir: `fiyat * adet` → 120 * 3 → 360	120	3	?	
4	3	üretilen değer `toplam`'a atanır	120	3	360	
5	4	koşul ifadesi: `toplam > 300` → 360 > 300 → `true`; bloğa girilir	120	3	360	
6	5	bloktaki deyim çalışır	120	3	360	`İndirim`
7	6	blok biter	120	3	360	`İndirim`

Tablo 5.15: Kod parçasının adım adım izlemesi. "?" henüz değer verilmemiş değişkeni gösterir.

Birinci ve ikinci satırlar **bildirim deyimleridir** (*declaration statement*): bir değişken tanımlar ve ona ilk değerini verir. Üçüncü satırda önce ifade değerlendirilir, sonra atama yapılır (Tablo 5.15, 3. ve 4. adımlar). Eşittir işaretinin sağı her zaman önce hesaplanır. Dördüncü satırdaki `if` deyimi bir koşul ifadesi ve bir bloktan oluşur. Koşul `true` çıktığı için blok çalışır. `adet` 2 olsaydı `toplam` 240 olur, koşul `false` çıkar ve blok hiç çalışmazdı. Blok, birden çok deyimi tek bir birim hâline getirir: koşul doğruysa bloktaki deyimlerin hepsi, yanlışsa hiçbiri çalışmaz.

5.10 Açıklamalar

Kod yalnızca derleyiciye değil, onu okuyacak insanlara da yazılır: aylar sonra kendi koduna dönen programcıya, takım arkadaşlarına, dersi değerlendiren öğretim elemanına. **Açıklama** (*comment*), kaynak koda okuyan kişi için eklenen ve derleyicinin görmezden geldiği metindir.

5.10.1 Üç açıklama biçimi

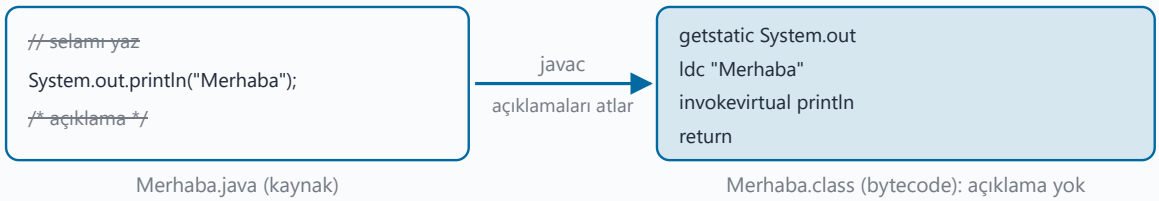
Java'da üç açıklama biçimi vardır:

- `//` ile başlayan açıklama satırın sonuna kadar sürer. Kısa notlar için kullanılır.
- `/*` ile başlayıp `*/` ile biten açıklama birden çok satırı kapsayabilir.
- `/**` ile başlayıp `*/` ile biten açıklama bir **Javadoc** açıklamasıdır. `javadoc` aracı bu açıklamalardan API belgesi üretir. Genellikle sınıf ve metot bildirimlerinin hemen üstüne yazılır.

```
/** Merhaba programı (Javadoc). */
public class Merhaba {
    /* Birden fazla satıra
       yayılan açıklama */
    public static void main(String[] args) {
        // selamı yaz
        System.out.println("Merhaba");
    }
}
```

```
> java Merhaba
Merhaba
> javap -c Merhaba
Compiled from "Merhaba.java"
public class Merhaba {
  public Merhaba();
    Code:
       0: aload_0
       1: invokespecial #1                  // Method java/lang/Object."<init>":()V
       4: return

  public static void main(java.lang.String[]);
    Code:
       0: getstatic     #7                  // Field
java/lang/System.out:Ljava/io/PrintStream;
       3: ldc           #13                 // String Merhaba
       5: invokevirtual #15                 // Method java/io/PrintStream.println:
(Ljava/lang/String;)V
       8: return
}
```



Şekil 5.23: `javac` açıklamaları atlar. Kaynaktaki `// selamı yaz` ve `/* açıklama */` satırları bytecode'a girmez; `.class` dosyasında yalnızca komutlar bulunur.

Açıklamalı programın `javap` çıktısı, açıklamasız Merhaba programınınkiyle aynıdır (Şekil 5.23). Derleyici açıklamaları atlar; açıklamalar bytecode'a girmez ve programı yavaşlatmaz.

İyi bir açıklama, kodun *ne* yaptığını tekrarlamak yerine *neden* yaptığını anlatır. `i = i + 1; // i'yi bir artır` açıklaması işe yaramaz; koda bakan zaten bunu görür. `toplam = toplam / 2; // iki sınavın ağırlığı eşit` ise koddan çıkarılamayan bir bilgiyi verir.

5.10.2 Açıklama tuzakları

Açıklamalar basit görünür, ama birkaç tuzak barındırır.

İç içe çok satırlı açıklama. `/* … */` açıklamaları iç içe yazılamaz. Açıklama, ilk karşılaşılan `*/`'da biter:

```
public class Ic {
  public static void main(String[] args) {
    /* dış /* iç */ kalan */
    System.out.println("Merhaba");
  }
}
```

```
Ic.java:3: error: illegal start of expression
    /* dış /* iç */ kalan */
                          ^
1 error
```

Derleyici `/* dış /* iç */` kısmını açıklama sayar; ortadaki ikinci `/*` açıklamanın içinde kaldığı için bir anlam taşımaz. Açıklama `iç`'ten sonraki `*/`'da biter. Geriye `kalan */` kalır ve derleyici bunu kod sanır. `kalan` bir tanımlayıcı, `*` çarpma işleci sayılır. Çarpmanın sağında bir değer beklenirken `/` gelir ve hiçbir ifade `/` ile başlayamaz. Derleyici "ifadenin geçersiz başlangıcı" (`illegal start of expression`) der; `^` işareti bu son `/` karakterini gösterir. İçinde `/* */` bulunan bir bölümü devre dışı bırakmak için satırları tek tek `//` ile açıklama yapmak daha güvenlidir.

Tırnak içindeki açıklama işaretleri. Dizgi içindeki `//` ya da `/* */` açıklama değildir; metnin parçasıdır. Aşağıdaki programda hangi satırların çalıştığını izleyelim:

```
public class Y {
    public static void main(String[] args) {
        // System.out.println("1");
        System.out.println("2"); // System.out.println("3");
        /* System.out.println("4");
        System.out.println("5"); */
        System.out.println("6 // değil");
        System.out.println("7 /* yorum değil */");
        System.out.println("a // b");
    }
}
```

```
2
6 // değil
7 /* yorum değil */
a // b
```

İlk satır baştan sona açıklamadır. İkinci satırda `"2"` yazılır; `//`'dan sonrası, yani 3'ü yazacak deyim, açıklamadır. `/* … */` iki satırı (4 ve 5) birlikte açıklama yapar. Son üç satırdaki `//` ve `/* */` işaretleri tırnak içinde olduğu için aynen yazılır.

Kodu geçici olarak devre dışı bırakmak. Bir satırın başına `//` koymak, o satırı silmeden çalışmaz hâle getirir. Hata ararken sık kullanılan bir yöntemdir. IDE'lerde Ctrl+/ kısayolu seçili satırları bu biçimde açıklamaya çevirir ve geri alır.

Eskimiş açıklama. En sinsi tuzak, kodla çelişen açıklamadır:

```
double fiyat = net * 1.20; // KDV %18
```

Kod derlenir ve doğru çalışır: KDV oranı güncellenmiş, kod %20'ye göre yazılmıştır. Ama açıklama eski oranı söyler ve okuyanı yanıltır. Açıklama ile kod çeliştiğinde derleyici kodu çalıştırır; doğru olan her zaman koddur. Kod değiştirildiğinde ilgili açıklama da güncellenmelidir.

Örnek 5.4

Aşağıdaki program ne yazar?

```
public class Yorum {
    public static void main(String[] args) {
        int x = 5; // x = 10;
        /* x = 20; */ x = x + 1;
        System.out.println("x = " + x); // x = 100
        System.out.println("// x = " + (x * 2));
    }
}
```

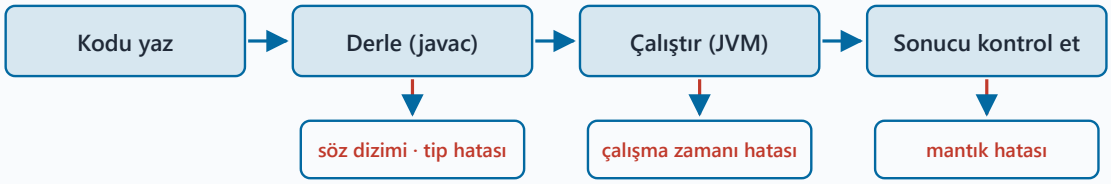
3. satırda `x` 5 olur; `// x = 10;` açıklamadır. 4. satırda `/* x = 20; */` açıklamadır, ama aynı satırdaki `x = x + 1;` açıklamanın dışında kalır ve çalışır: `x` 6 olur. 5. satır `x = 6` yazar; satır sonundaki `// x = 100` yalnızca yanlış bir nottur. 6. satırdaki `//` tırnak içinde olduğu için metnin parçasıdır; `x * 2` parantez içinde önce hesaplanır ve 12 bulunur:

```
x = 6
// x = 12
```

5.11 Hatalar

Her programcı hata yapar. İyi programcıyı ayıran, hatayı çabuk bulup düzeltebilmesidir. Bunun ilk adımı, hatanın türünü tanımaktır. Hatanın ne zaman ortaya çıktığı, onu kimin yakaladığını ve nasıl aranacağını belirler.

5.11.1 Hata türleri



Şekil 5.24: Hataların yakalandığı aşamalar. Söz dizimi ve tip hataları derlemede (`javac`), çalışma zamanı hataları programı çalıştırırken (JVM), mantık hataları ancak sonuç kontrol edilirken bulunur.

Hata	Ne zaman?	Kim yakalar?	Örnek
Söz dizimi	derleme sırasında	`javac`	eksik noktalı virgül
Tip	derleme sırasında	`javac`	metni sayı değişkenine atama
Çalışma zamanı	program çalışırken	JVM	tam sayıyı sıfıra bölme
Mantık	program çalışır ama sonuç yanlıştır	yalnızca test	yanlış işlem sırası

Tablo 5.16: Hata türleri.

Söz dizimi hatası (*syntax error*), dilin yazım kurallarının bozulmasıdır: eksik noktalı virgül, kapanmamış parantez ya da tırnak. **Tip hatası** (*type error*), söz dizimi doğru olduğu hâlde bir değerin tipine uymayan bir yerde kullanılmasıdır. Bu ikisine birlikte **derleme hatası** (*compile error*) denir. Derleme hatası olan program `.class` dosyasına dönüşmez ve hiç çalışmaz.

Çalışma zamanı hatası (*runtime error*), derlenmiş bir programın çalışırken yapamayacağı bir işle karşılaşmasıdır. Java'da bu durumda JVM bir istisna fırlatır ve program, özel bir önlem alınmamışsa durur. **Mantık hatası** (*logic error*) ise en sinsi türdür: program derlenir, hatasız çalışır ve bir sonuç üretir, ama sonuç yanlıştır. Hiçbir araç hata iletisi vermez. Mantık hatasını yalnızca sonucu beklenen değerle karşılaştıran biri yakalayabilir.

Hatalar ne kadar erken yakalanırsa o kadar ucuza düzeltilir. Derleme hatası programcının bilgisayarında, saniyeler içinde bulunur. Mantık hatası ise program kullanıcılara ulaştıktan sonra fark edilebilir. Java'nın derleme sırasında tip denetimi yapması bu yüzden bir avantajdır: birçok yanlışı programcı daha programı çalıştırmadan görür.

5.11.2 Dört hata, dört örnek

Aşağıdaki dört program her hata türünden birer örnek verir. Her birinin JDK 21'deki gerçek iletisi verilmiştir.

Söz dizimi hatası. 3. satırın sonunda noktalı virgül yoktur:

```java
public class Hata1 {
    public static void main(String[] args) {
        System.out.println("Merhaba")
    }
}
```

```
Hata1.java:3: error: ';' expected
        System.out.println("Merhaba")
                                    ^
1 error
```

Derleyici kuralın bozulduğu yeri satır numarasıyla gösterir: `Hata1.java:3`. `';' expected` "noktalı virgül bekleniyordu" demektir. Son satırdaki `1 error` toplam hata sayısıdır. `.class` üretilmez. Düzeltme: satır sonuna `;` eklemek.

Tip hatası. Bir dizgi, `int` tipindeki bir değişkene konmaya çalışılır:

```java
public class Hata2 {
    public static void main(String[] args) {
        int sayi = "42";
    }
}
```

```
Hata2.java:3: error: incompatible types: String cannot be converted to int
        int sayi = "42";
                   ^
1 error
```

Söz dizimi kusursuzdur: tip, ad, eşittir, değer, noktalı virgül. Ama `"42"` çift tırnak içinde olduğu için bir sayı değil, iki karakterlik bir metindir. İleti "uyumsuz tipler: String, int'e dönüştürülemez" der. `^` uyumsuz değeri gösterir. Düzeltme: tırnakları kaldırıp `int sayi = 42;` yazmak.

Çalışma zamanı hatası. Program derlenir, ama çalışırken bir tam sayı sıfıra bölünür:

```java
public class Hata3 {
    public static void main(String[] args) {
        int a = 10, b = 0;
        System.out.println(a / b);
    }
}
```

```
> javac Hata3.java
> java Hata3
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Hata3.main(Hata3.java:4)
```

`javac` sessiz kalır: söz dizimi ve tipler doğrudur. `b`'nin değerinin 0 olacağını derleyici denetlemez. Program çalışırken 4. satırda tam sayı bölmesi sıfıra yapılır. Matematikte de tanımsız olan bu işlem karşısında JVM bir `ArithmeticException` (aritmetik istisna) fırlatır. İleti `/ by zero`, yani "sıfıra bölme" der ve hatanın yerini verir: `Hata3.java`, 4. satır, `main` metodu. Program orada durur.

Mantık hatası. İki notun ortalaması bulunmak istenir:

```
public class Hata4 {
    public static void main(String[] args) {
        int a = 70, b = 90;
        System.out.println(a + b / 2);
    }
}
```

```
> javac Hata4.java
> java Hata4
115
```

Hata iletisi yoktur, ama sonuç yanlıştır. 70 ve 90'ın ortalaması 80'dir, program ise 115 yazar. Neden, işleç önceliğidir: `/`, `+`'dan önce yapılır. `a + b / 2` ifadesi `a + (b / 2)` gibi değerlendirilir: 90 / 2 = 45, 70 + 45 = 115. Düzeltme: `(a + b) / 2` yazmak; bu ifade (70 + 90) / 2 = 160 / 2 = 80 verir.

> **Sık yapılan hata**
>
> "Program derlendi ve çalıştı, öyleyse doğrudur." sonucuna varmak. Derleme yalnızca söz dizimi ve tip hatalarının olmadığını gösterir. Mantık hatası olan bir program da sorunsuz derlenip çalışır. Programın doğru olduğuna ancak bilinen girdiler için çıktısını beklenen sonuçla karşılaştırarak güvenilebilir.

5.11.3 Sık derleme ve başlatma hataları

Yeni başlayanların karşılaştığı derleme hatalarının büyük bölümü birkaç kalıba uyar. Aşağıdaki tablo bunları JDK 21'in gerçek iletileriyle verir.

Hata	Gerçek ileti	Çözüm
Noktalı virgül yok	`Merhaba.java:3: error: ';' expected`	önceki deyimin sonuna `;` eklemek

Hata	Gerçek ileti	Çözüm
Metot adı yanlış (`System.out.printn`)	`error: cannot find symbol` `symbol: method printn(String)` `location: variable out of type PrintStream`	adı `println` yapmak
Küçük harf (`system.out`)	`error: package system does not exist`	`System`'i büyük S ile yazmak
Sınıf adı ile dosya adı uyuşmuyor	`error: class Selam is public, should be declared in a file named Selam.java`	dosya adını public sınıf adıyla aynı yapmak
Yanlış çalıştırma (`java Selam`)	`Error: Could not find or load main class Selam`	`.class` dosyası yok ya da ad/konum yanlış; önce derlemek, uzantı yazmamak

Tablo 5.17: Sık karşılaşılan hatalar ve iletileri. İlk dördü derleme, sonuncusu başlatma hatasıdır.

İletilerin tam hâli, sorunun nerede olduğunu daha açık gösterir. Metot adı yanlış yazıldığında:

```
Merhaba.java:3: error: cannot find symbol
        System.out.printn("Merhaba");
                   ^
  symbol:   method printn(String)
  location: variable out of type PrintStream
1 error
```

`cannot find symbol`, derleyicinin bir adı (değişken, metot ya da sınıf adı) tanımadığını söyler. Altındaki `symbol:` satırı hangi adın bulunamadığını, `location:` satırı ise nerede arandığını verir. Bu iletiye göre bir `String` alan `printn` metodu, `PrintStream` tipindeki `out` değişkeninde aranmış ve bulunamamıştır. `^` işareti `out`'tan sonraki noktayı gösterir; metot oraya bağlanmaya çalışılmıştır.

`System` küçük harfle yazıldığında ileti ilk bakışta şaşırtıcıdır:

```
Merhaba.java:3: error: package system does not exist
        system.out.println("Merhaba");
              ^
1 error
```

Derleyici `system` adında bir sınıf tanımaz. `system.out.println` yazımındaki noktaları görünce `system`'in bir paket adı olabileceğini düşünür ve öyle bir paket de bulamaz. İletinin “paket” demesi bu yüzdendir; asıl sorun büyük-küçük harftir. Aynı durum adı yanlış yazılmış bir `System` için de görülür: `Sytem.out.println` için ileti `package Sytem does not exist` olur.

Public sınıfın adı dosya adından farklıysa derleyici hatayı sınıf bildirimi satırında verir:

```
public class Selam {
    public static void main(String[] args) {
        System.out.println("x");
    }
}
```

```
> javac Merhaba.java
Merhaba.java:1: error: class Selam is public, should be declared in a file named
Selam.java
public class Selam {
       ^
1 error
> java Selam
Error: Could not find or load main class Selam
Caused by: java.lang.ClassNotFoundException: Selam
```

Derleme başarısız olduğu için `Selam.class` da üretilmemiştir; ardından gelen `java Selam` komutu sınıfı bulamaz. Harf büyüklüğü de sayılır. `Merhaba` sınıfı `merhaba.java` dosyasına kaydedilirse Windows dosya adlarında harf büyüklüğünü ayırmasa da derleyici ayırır:

```
merhaba.java:1: error: class Merhaba is public, should be declared in a file named
Merhaba.java
public class Merhaba {
       ^
1 error
```

Sık görülen iki derleme hatası daha vardır. Kapanmamış bir tırnak, satırın sonuna kadar her şeyi dizginin parçası yapar:

```
Tirnak.java:3: error: unclosed string literal
        System.out.println("Merhaba);
                           ^
1 error
```

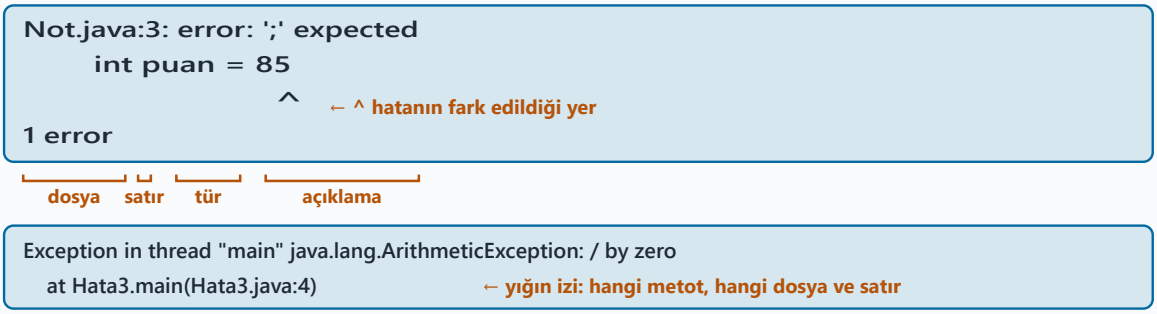
"Kapanmamış dizgi sabiti" iletisindeki `^`, dizginin başladığı tırnağı gösterir. Eksik bir kapanış süslü parantezi ise dosyanın sonunda fark edilir. Aşağıdaki iletide sınıfı kapatan son `}` silinmiştir:

```
Merhaba.java:4: error: reached end of file while parsing
    }
     ^
1 error
```

"Ayrıştırma sırasında dosyanın sonuna ulaşıldı" iletisi, derleyicinin hâlâ bir bloğun kapanmasını beklerken dosyanın bittiğini söyler. Hatanın gerçek yeri dosyanın sonu değil, `}`'nin eksik olduğu yerdir. Doğru girintilenmiş kodda bu yer kolayca bulunur.

5.11.4 Hata iletisini okumak

Hata iletileri ilk bakışta korkutucu görünür, ama hep aynı düzende yazılır. Düzeni bir kez öğrenmek, her iletiyi okumayı kolaylaştırır.



Şekil 5.25: Hata iletisinin parçaları. Üstte `javac` iletisi: dosya, satır, tür, açıklama; altında sorunlu kaynak satırı ve hatanın fark edildiği yeri gösteren `^`. Altta bir çalışma zamanı iletisi: istisnanın türü ve ayrıntısı, ardından hangi metotta, hangi dosyanın hangi satırında oluştuğunu söyleyen yığın izi.

Bir `javac` iletisinin ilk satırı `dosya:satır: error: açıklama` biçimindedir (Şekil 5.25). `Not.java:3: error: ';' expected` iletisinde dosya `Not.java`, satır 3, tür `error` (hata), açıklama `';' expected`'dır. İkinci satır sorunlu kaynak satırını aynen gösterir. Üçüncü satırdaki `^` işareti, derleyicinin sorunu fark ettiği sütunu gösterir. En sonda toplam hata sayısı yazılır.

`^` her zaman hatanın kendisini değil, derleyicinin onu *fark ettiği* yeri gösterir. Eksik noktalı virgül, deyimin bittiği yerde, yani satır sonunda fark edilir. Eksik bir `}` ise dosyanın sonuna kadar fark edilmez ve `reached end of file while parsing` olarak bildirilir. Bu yüzden hata gösterilen yerde bulunamazsa bir önceki satıra ya da yukarıdaki bloklara bakmak gerekir.

Çalışma zamanı iletileri farklı bir düzendedir. İlk satır `Exception in thread "main"` ile başlar ve istisnanın türünü ve ayrıntısını verir: `java.lang.ArithmeticException: / by zero`. Altındaki `at` ile başlayan satırlar **yığın izidir** (*stack trace*): hata anında çağrı yığınında hangi metotların bulunduğunu, en içteki metottan başlayarak listeler. `at Hata3.main(Hata3.java:4)`, hatanın `Hata3` sınıfının `main` metodunda, `Hata3.java` dosyasının 4. satırında oluştuğunu söyler. İlk programlarda yığın izi tek satırdır, çünkü yalnızca `main` çalışır. Metotlar birbirini çağırdıkça iz uzar.

5.11.5 Hataları sırayla düzeltmek

Bir programda birden çok hata olabilir. Derleyici bunları her zaman bir anda bildirmez. Aşağıdaki programda iki hata vardır: 3. satırda noktalı virgül eksiktir, 4. satırda metot adı yanlıştır.

```
public class Not {
    public static void main(String[] args) {
        int puan = 85
        System.out.printn("Puan: " + puan);
    }
}
```

İlk derleme yalnızca bir hata bildirir:

```
Not.java:3: error: ';' expected
        int puan = 85
                    ^
1 error
```

4. satırdaki yanlış henüz görünmez. Bunun nedeni derleyicinin aşamalı çalışmasıdır. Derleyici önce söz dizimini çözümler. Söz dizimi bozuk olduğu sürece anlam denetimine, yani adların ve tiplerin denetimine geçmez. Hata düzeltilir, 3. satır `int puan = 85;` olur ve dosya kaydedilir. İkinci derleme bu kez başka bir hata bildirir:

```
Not.java:4: error: cannot find symbol
        System.out.printn("Puan: " + puan);
                   ^
  symbol:   method printn(String)
  location: variable out of type PrintStream
1 error
```

Söz dizimi artık doğrudur; anlam denetimi çalışır ve `printn` adlı metodu bulamaz. `symbol:` satırı bulunamayan adı söyler. Metot adı `println` olarak düzeltilir. Üçüncü derleme sessizce başarılı olur, `Not.class` oluşur ve program çalışır:

```
> javac Not.java
> java Not
Puan: 85
```

Derleme	3. satır	4. satır	Bildirilen hata	`Not.class`
1	`int puan = 85` ✗	`printn` ✗	`Not.java:3: error: ';' expected`	yok
2	`int puan = 85;` ✓	`printn` ✗	`Not.java:4: error: cannot find symbol`	yok
3	`int puan = 85;` ✓	`println` ✓	yok	oluştu

Tablo 5.18: Hataların sırayla düzeltilmesi.

Bu örnekten çıkan ders şudur: bir hatayı düzelttikten sonra hata sayısı azalmayabilir, hatta artabilir. Bu, düzeltmenin yanlış olduğu anlamına gelmez; derleyici yalnızca bir sonraki aşamaya geçmiştir. Tersine, aynı aşamadaki hatalar birlikte bildirilir. Aşağıdaki programdaki iki tip hatası tek derlemede görünür:

```
Iki.java:3: error: incompatible types: String cannot be converted to int
        int a = "5";
                ^
Iki.java:4: error: incompatible types: String cannot be converted to int
        int b = "6";
                ^
2 errors
```

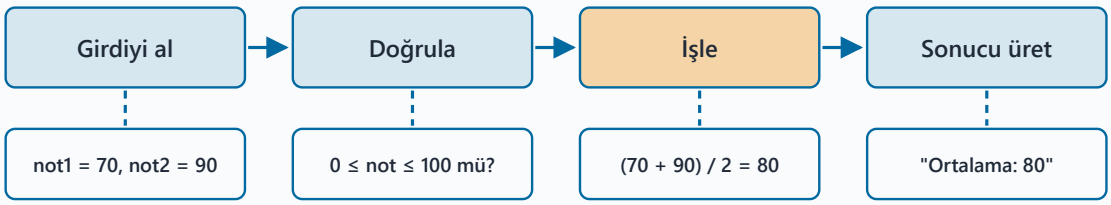
Bir söz dizimi hatası ise ardından gelen satırları da karıştırabilir ve gerçekte var olmayan hatalar bildirilebilir. Bu yüzden her zaman **ilk hatadan** başlanır. İlk hata düzeltilir, dosya yeniden derlenir ve yeni iletiler okunur.

5.12 Kodlamadan önce

Hata iletilerini okumayı bilmek önemlidir, ama en ucuz hata hiç yapılmayandır. Mantık hatalarının çoğu kod yazılmadan önce, kâğıt üzerinde yapılan küçük bir hazırlıkla önlenebilir.

5.12.1 Program bir tariftir

Bir yemek tarifi malzemeleri sayar, adımları sırayla verir ve sonunda ne elde edileceğini söyler. Bir program da aynı yapıdadır. Girdiler malzemelere, deyimler tarifin adımlarına, çıktı da ortaya çıkan yemeğe karşılık gelir. İyi bir tarifte her adımın ne gerektirdiği ve ne ürettiği bellidir: “Hamur mayalandıktan sonra (ön koşul) fırına verilir ve 30 dakika pişirilir (adım); ekmek kızarır (sonuç).”



Şekil 5.26: Ortalama programının tarifi: girdiyi al (not1 = 70, not2 = 90), doğrula (0 ≤ not ≤ 100 mü?), işle ((70 + 90) / 2 = 80), sonucu üret (“Ortalama: 80”).

Şekil 5.26 iki notun ortalamasını bulan programın tarifini dört adımda verir. Her adımın ön koşulu ve ürettiği değer bellidir:

1. **Girdiyi al.** Ön koşul yoktur. Ürettiği değerler: `not1` ve `not2`.
2. **Doğrula.** Ön koşul: iki not okunmuştur. Her notun 0 ile 100 arasında olup olmadığı denetlenir. Geçersiz bir not varsa tarif burada durur ve bir hata iletisi verir.
3. **İşle.** Ön koşul: iki not da geçerlidir. Ortalama `(not1 + not2) / 2` ile hesaplanır.
4. **Sonucu üret.** Ön koşul: ortalama hesaplanmıştır. "Ortalama: 80" gibi bir satır yazılır.

Tarifi kodlamadan önce elle sınamak gerekir. 70 ve 90 için adımları izleyince 80 çıkıyor mu? 160 / 2 = 80; evet. 101 girilirse hangi adım durdurur? İkinci adım: 101 > 100 olduğu için doğrulama başarısız olur. Elle sınama, mantık hatalarını kod yazılmadan yakalar. İşlem sırası hatası örneğinde (a + b / 2) tarif açıkça "önce topla, sonra böl" dediği için hata daha tarif aşamasında görülebilirdi.

Büyük tarifler alt tariflere ayrılır. Bir pasta tarifinde "krema hazırla" ayrı bir tarif olabilir. Programlarda da büyük işler alt görevlere bölünür; Java'da her alt görev bir metot olarak yazılır. İlk programlarda tek bir `main` metodu yeterlidir, ama tarif düşüncesi baştan alışkanlık hâline getirilmelidir.

5.12.2 Örnek girdi ve beklenen çıktı

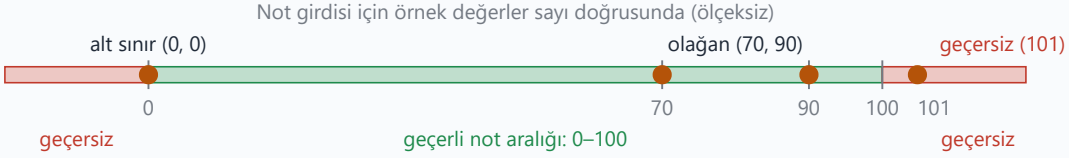
Kodlamaya başlamadan önce yapılacak dört iş vardır:

1. Örnek girdileri ve her biri için **beklenen çıktıyı** yazmak.
2. Çözümü doğal dille ya da sözde kodla adımlara ayırmak.
3. **Sınır durumlarını** (*boundary case*) ve hatalı girdide ne yapılacağını belirlemek.
4. Ancak bundan sonra adımları Java yapılarına dönüştürmek.

Birinci adımın ürünü küçük bir tablodur. Ortalama programı için şöyle bir tablo hazırlanabilir:

Örnek girdi	Beklenen çıktı	Neden?
70, 90	80	olağan durum
0, 0	0	alt sınır
101, 50	hata iletisi	geçersiz girdi

Tablo 5.19: Ortalama programı için örnek girdiler ve beklenen çıktılar.



Şekil 5.27: Not girdisi için örnek değerler sayı doğrusunda (ölçeksiz). Geçerli aralık 0–100'dür. 0 alt sınır, 70 ve 90 olağan değerler, 101 ise geçersiz bir değerdir.

Tablodaki her satır farklı bir şeyi sınar (Şekil 5.27). **Olağan durum**, programın gündelik kullanımda doğru çalışıp çalışmadığını gösterir. **Sınır durumu**, geçerli aralığın uçlarındaki değerlerdir; programlar en sık burada yanılır. `<` yerine `<=` yazmak gibi küçük bir yanlış yalnızca sınırda ortaya çıkar. **Geçersiz girdi** ise programın beklenmedik bir değer karşısında ne yaptığını gösterir: çökmemeli, anlaşılır bir ileti vermelidir.

Tablonun bir yararı daha vardır: program yazıldıktan sonra her satır bir sınamaya dönüşür. Program bu girdilerle çalıştırılır ve çıktısı beklenen sütunuyla karşılaştırılır. Farklı türde örnekler seçmek bu yüzden önemlidir. Yalnızca olağan girdilerle sınanan bir program, iki tam sayının bölümünde kesirli kısmın atılması gibi bir yanlışı gizleyebilir. Örneğin 75 ve 80 için doğru ortalama 77,5'tir, ama iki `int`'in bölümü 77 verir. Bu yanlış, toplamı tek sayı olan bir girdi denenmeden görünmez.

5.13 Özet

- **Program**, bilgisayarın bir görevi yerine getirmek için yürüttüğü komutlar bütünüdür; girdi alır, işler, durum saklar ve çıktı üretir. Bir programlama dili **söz dizimi**, **anlam**, **tip sistemi** ve **çalışma modeli** yönlerinden incelenir.
- **Kaynak kod** düz metin bir `.java` dosyasına yazılır. Karakter kodlaması (UTF-8), satır sonu ve dosya adı tutarlı olmalıdır. Düzenleyici, IDE, `javac`, `java`, `javadoc` ve sürüm kontrolü programın yolculuğundaki araçlardır.
- Java 1995'te Sun Microsystems'te doğdu; 2010'dan beri Oracle geliştiriyor, kaynak kodu OpenJDK'dadır. Tasarım hedefi **bir kez yaz, her yerde çalıştır** ilkesidir. Altı ayda bir yeni sürüm çıkar; 8, 11, 17, 21 ve 25 **LTS** sürümleridir.
- **JDK** = JRE + geliştirme araçları; **JRE** = JVM + sınıf kütüphanesi. **JVM** bytecode'u yürütür; sınıf yükleyici, doğrulayıcı, yorumlayıcı, JIT derleyicisi ve çöp toplayıcı içerir.
- Her Java kodu bir sınıfın içindedir. Public sınıfın adı dosya adıyla aynıdır. JVM çalıştırmaya `public static void main(String[] args)` imzalı metottan başlar. İmzası yanlış bir `main` derlenir ama başlatılamaz. Java büyük-küçük harfe duyarlıdır.
- **Komut satırı argümanları** `args` dizisiyle `main`'e ulaşır; indisler 0'dan başlar, `args.length` argüman sayısıdır. Olmayan bir argümana erişmek çalışma sırasında `ArrayIndexOutOfBoundsException` fırlatır.
- `println` yazdıktan sonra satır sonu ekler, `print` eklemez. **Kaçış dizileri** (`\n`, `\t`, `\"`, `\\`, `\'`, `\u`) dizgiye özel karakterler koyar. Windows yollarında tek ters eğik çizgi ya derleme hatası verir ya da sessizce yanlış çıktı üretir.
- `+` iki sayı arasında toplar, işlenenlerden biri dizgiyse birleştirir. İşlemler soldan sağa yapılır; `*` ve `/` önce gelir: `"Puan: " + 40 + 45` → `Puan: 4045`, `"Puan: " + (40 + 45)` → `Puan: 85`.
- `javac Merhaba.java` derler ve `Merhaba.class` üretir; `java Merhaba` çalıştırır. Kaynak değişince yeniden derlemek gerekir, çünkü JVM yalnızca `.class` dosyasını çalıştırır. Derleme hatası varsa `.class` üretilmez.

- **Bytecode** işlemciden bağımsızdır ve `CA FE BA BE` ile başlayan dosyalarda saklanır. JVM bir **yığın makinesidir**: `iload` değerleri **işlenen yığınına** koyar, `iadd` yığındaki değerleri toplar, `istore` sonucu yerel değişkene yazar. `javap -c` bytecode'u okunur biçimde gösterir.
- Derleyici kodu **belirteçlere** ayırır: anahtar sözcük, tanımlayıcı, sabit değer, işleç, ayırıcı. Boşluk yalnızca belirteçleri ayırır; girinti okuyan içindir. **İfade** değer üretir, **deyim** işlemi tamamlar, **blok** deyimleri gruplar.
- Açıklamalar `//`, `/* */` ve `/** */` ile yazılır ve bytecode'a girmez. `/* */` iç içe yazılamaz; tırnak içindeki açıklama işaretleri metnin parçasıdır.
- **Söz dizimi** ve **tip hataları** derlemede, **çalışma zamanı hataları** çalışırken, **mantık hataları** ancak sonuç denetlenince bulunur. `javac` iletisi `dosya:satır: error: açıklama` biçimindedir; `^` hatanın fark edildiği yeri gösterir. Çalışma zamanı iletisindeki **yığın izi** hatanın metodunu ve satırını verir. Hatalar her zaman ilk hatadan başlanarak düzeltilir.
- Kodlamadan önce örnek girdiler, beklenen çıktılar, sınır durumları ve hata davranışı belirlenir; tarif elle sınanır.

5.14 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

program *program* · söz dizimi *syntax* · anlam *semantics* · tip sistemi *type system* · çalışma modeli *execution model* · kaynak kod *source code* · kaynak dosya *source file* · karakter kodlaması *character encoding* · derleyici *compiler* · derleme *compilation* · bytecode (ara kod) *bytecode* · Java Sanal Makinesi (JVM) *Java Virtual Machine* · JRE (çalışma ortamı) *Java Runtime Environment* · JDK (geliştirme kiti) *Java Development Kit* · sınıf kütüphanesi *class library* · sınıf yükleyici *class loader* · bytecode doğrulayıcı *bytecode verifier* · JIT *just-in-time compilation* · çöp toplama *garbage collection* · uzun süreli destek (LTS) *long-term support* · IDE *integrated development environment* · Javadoc (API belgesi) *Javadoc* · sınıf bildirimi *class declaration* · main metodu *main method* · büyük-küçük harfe duyarlı *case-sensitive* · komut satırı argümanı *command-line argument* · konsol *console* · satır sonu *newline* · kaçış dizisi *escape sequence* · sekme durağı *tab stop* · Unicode kaçışı *Unicode escape* · dizgi birleştirme *string concatenation* · paket *package* · ortam değişkeni *environment variable* · yığın makinesi *stack machine* · işlenen yığını *operand stack* · belirteç *token* · anahtar sözcük *keyword* · tanımlayıcı *identifier* · sabit değer *literal* · ayırıcı *separator* · ifade *expression* · deyim *statement* · bildirim deyimi *declaration statement* · açıklama *comment* · söz dizimi hatası *syntax error* · tip hatası *type error* · derleme hatası *compile error* · çalışma zamanı hatası *runtime error*

mantık hatası *logic error* · istisna *exception* · yığın izi *stack trace* · hata iletisi *error message* · sınır durumu *boundary case*

5.15 Kendini Dene

1. Bir bilgisayarda yalnızca JRE kurulu ve JDK yok. Bu bilgisayarda `Merhaba.java` derlenebilir mi? Başka bir bilgisayarda derlenmiş `Merhaba.class` çalıştırılabilir mi?

Cevap

Derlenemez: `javac` bir geliştirme aracıdır ve yalnızca JDK'de bulunur. Çalıştırılabilir: JRE, JVM'yi ve sınıf kütüphanesini içerir, bu da `java Merhaba` için yeterlidir. Bir koşulla: `.class` dosyası, JRE'nin sürümünden daha yeni bir JDK ile derlenmemiş olmalıdır. Aksi hâlde `UnsupportedClassVersionError` alınır.

2. Aşağıdaki programın çıktısı nedir, kaç satır oluşur?

```
public class KD2 {
    public static void main(String[] args) {
        System.out.print("1");
        System.out.print("2\n3");
        System.out.println("4");
        System.out.print("\"5\"");
        System.out.println();
    }
}
```

Cevap

```
12
34
"5"
```

Üç satır oluşur. “1” ve “2” aynı satıra yazılır; `"2\n3"` içindeki `\n` satırı bitirir ve “3” alt satırdan başlar. `println("4")` “4”ü aynı satıra yazıp satırı bitirir. `\"` çift tırnak yazdığı için üçüncü satır `"5"` 'tir; son `println()` bu satırı bitirir.

3. Aşağıdaki programın çıktısı nedir? Her satırdaki `+`'ların hangisi toplama, hangisi birleştirmedir?

```
public class KD3 {
    public static void main(String[] args) {
        System.out.println("Sonuç: " + 10 / 4 + 1);
        System.out.println(10 / 4 + 1 + " sonuç");
        System.out.println("Sonuç: " + (10 / 4 + 1));
    }
}
```

Cevap

```
Sonuç: 21
3 sonuç
Sonuç: 3
```

1. satır: `/` önce yapılır, 10 / 4 = 2 (tam sayı bölmesi). Ardından `"Sonuç: " + 2` birleştirmedir ve `"Sonuç: 2"` verir; `+ 1` de birleştirmedir: `Sonuç: 21`. 2. satır: 10 / 4 = 2, `2 + 1` iki sayı arasında olduğu için toplamadır ve 3 verir; `3 + " sonuç"` birleştirmedir. 3. satır: parantez önce hesaplanır, 2 + 1 = 3; sonra birleştirme yapılır.

4. `B` programı aşağıdadır. `java B` ve `java B Ece` komutlarıyla çalıştırılınca ne olur? Hata hangi türdendir?

```
public class B {
    public static void main(String[] args) {
        System.out.println("İlk argüman: " + args[0]);
    }
}
```

Cevap

```
> java B
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out
of bounds for length 0
        at B.main(B.java:3)
> java B Ece
İlk argüman: Ece
```

Argüman verilmeyince `args` boş bir dizidir (uzunluğu 0) ve `args[0]` yoktur. JVM 3. satırda `ArrayIndexOutOfBoundsException` fırlatır. Program derlendiği ve hata girdiye bağlı olarak çalışırken çıktığı için bu bir çalışma zamanı hatasıdır. Tek argümanla program sorunsuz çalışır.

5. Aşağıdaki hataların her biri hangi türdendir: söz dizimi, tip, çalışma zamanı ya da mantık? (a) `Public class D {` (b) `int x = 3.5;` (c) Bir dikdörtgenin çevresi için `2 * a + b` yazmak (d) `int a = 10, b = 0;` sonra `a / b`.

Cevap

(a) Söz dizimi hatası: `Public` büyük harfle bir anahtar sözcük değildir ve derleyici satırın başında bir sınıf bildirimi bekler. İleti: `class, interface, enum, or record expected`. (b) Tip hatası: 3,5 kesirli bir sayıdır ve `int`'e sığmaz. İleti: `incompatible types: possible lossy conversion from double to int`. (c) Mantık hatası: kod derlenir ve çalışır, ama doğrusu `2 * (a + b)`'dir. (d) Çalışma zamanı hatası: `ArithmeticException: / by zero`.

6. `System.out.println("C:\temp\new");` deyimi derlenir mi? Derlenirse ne yazar? Doğru yol nasıl yazılır?

Cevap

Derlenir, çünkü `\t` ve `\n` geçerli kaçış dizileridir. Ama çıktı yanlıştır: `\t` sekmeye, `\n` satır sonuna dönüşür. Ekranda "C:", bir sekme boşluğu ve "emp" görünür, alt satırda "ew" yazar:

```
C:      emp
ew
```

Doğru yazım `"C:\\temp\\new"` ya da `"C:/temp/new"`'dir.

7. Aşağıdaki programın `main` metodu için `javap -c` çıktısında ilk sekiz komut ne olur? `iload` ko-

mutlarının sırası neden önemlidir?

```
public class Cikar {
    public static void main(String[] args) {
        int a = 2;
        int b = 5;
        int c = b - a;
        System.out.println(c);
    }
}
```

Cevap

```
public static void main(java.lang.String[]);
  Code:
     0: iconst_2
     1: istore_1
     2: iconst_5
     3: istore_2
     4: iload_2
     5: iload_1
     6: isub
     7: istore_3
          8:  getstatic          #7                                  //  Field
java/lang/System.out:Ljava/io/PrintStream;
    11: iload_3
    12: invokevirtual #13                 // Method java/io/PrintStream.println:
(I)V
    15: return
```

`a` 1, `b` 2 numaralı yuvadadır. `b - a` için önce `b` (`iload_2`), sonra `a` (`iload_1`) yığına konur. `isub` alttaki değerden üstteki değeri çıkarır: 5 − 2 = 3. Toplamada sıra sonucu değiştirmez, çıkarmada değiştirir; yükleme sırası kaynak koddaki işlenen sırasını izler. Program 3 yazar.

8. `Merhaba.java` derlenmiş ve `Merhaba` yazan bir `Merhaba.class` oluşmuştur. Sonra kaynakta `"Merhaba"`, `"Selam"` yapılıp kaydediliyor ve yeniden derlenmeden sırayla `java Merhaba`, `java`

`Merhaba.java`, `java Merhaba` komutları çalıştırılıyor. Her komut ne yazar?

Cevap

```
> java Merhaba
Merhaba
> java Merhaba.java
Selam
> java Merhaba
Merhaba
```

`java Merhaba`, `.class` dosyasındaki eski bytecode'u çalıştırır ve `Merhaba` yazar. `java Merhaba.java` ise kaynak dosyayı o anda bellekte derleyip çalıştırır ve `Selam` yazar; diskteki `.class` dosyasına dokunmaz. Bu yüzden son komut yine `Merhaba` yazar. `.class` ancak `javac` ile yenilenir.

9. Aşağıdaki program derleniyor. Programın hatasız derlenmesi için kaç derleme gerekir? Her derlemede hangi ileti alınır?

```java
public class Iki {
    public static void main(String[] args) {
        System.out.println("A")
        system.out.println("B");
    }
}
```

Cevap

Üç derleme gerekir. İlk derleme yalnızca söz dizimi hatasını bildirir; 4. satırdaki küçük harf hatası henüz görünmez:

```
Iki.java:3: error: ';' expected
        System.out.println("A")
                                ^
1 error
```

3. satıra `;` eklenince ikinci derleme anlam denetimine geçer ve 4. satırı bildirir:

```
Iki.java:4: error: package system does not exist
        system.out.println("B");
              ^
1 error
```

`system`, `System` yapılınca üçüncü derleme başarılı olur. İki hata vardı, ama derleyici her seferinde yalnızca birini gösterdi.

10. Bir dikdörtgenin kenar uzunluklarını alıp alanını yazan bir program yazılacak. Kodlamadan önce hangi örnek girdiler ve beklenen çıktılar tabloya yazılmalıdır?

Cevap

Örnek bir tablo: olağan durum, 3 ve 4 → 12. Kare, 5 ve 5 → 25 (iki kenar eşit olduğunda da çalışmalı). Sınır durumu, 0 ve 7 → 0 ya da "kenar 0 olamaz" iletisi; hangisinin isteneceği önceden kararlaştırılmalıdır. Geçersiz girdi, −2 ve 4 → hata iletisi (uzunluk negatif olamaz). Bu son iki satır, problemin tanımında eksik kalan kararları kodlamadan önce ortaya çıkarır.

5.16 İleri okuma

- Oracle, *The Java Tutorials*, "Getting Started" ve "A Closer Look at the 'Hello World!' Application": ilk programın her parçasını, `main` imzasını ve derleme–çalıştırma adımlarını resmî kaynaktan, kısa örneklerle anlatır.
- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), 1.1 Your First Program: düzenleme, derleme ve çalıştırma döngüsünü, komut satırı argümanlarını ve sık hata iletilerini ele alır; kitabın sitesinde (introcs.cs.princeton.edu) bölüm özeti ve alıştırmalar vardır.

- A. B. Downey, C. Mayfield, *Think Java* (2. baskı, O'Reilly, 2019), Bölüm 1: program, derleme, hata türleri ve kaçış dizilerini programlamaya yeni başlayanlar için sade bir dille anlatır; kitabın ücretsiz çevrim içi sürümü vardır.
- Oracle, *The Java Language Specification, Java SE 21 Edition*, Bölüm 3 “Lexical Structure”: belirteçlerin, tanımlayıcıların, açıklamaların ve kaçış dizilerinin kesin tanımını verir. *The Java Virtual Machine Specification*'ın 2. ve 6. bölümleri ise işlenen yığınını ve `iload`, `iadd` gibi komutları ayrıntılı açıklar.

5.17 Kaynaklar

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11. baskı, Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13. baskı, Pearson, 2024.
3. Oracle, *The Java Tutorials*: A Closer Look at the "Hello World!" Application.
4. Oracle, *The Java Language Specification, Java SE 21 Edition*: Chapter 3. Lexical Structure.
5. Oracle, *The Java Virtual Machine Specification, Java SE 21 Edition*: çevrim içi sürüm.
6. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 1.1 Your First Program.
7. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019 (ücretsiz çevrim içi sürüm).
8. OpenJDK, JEP 330: Launch Single-File Source-Code Programs (Java 11); JEP 400: UTF-8 by Default (Java 18); JEP 512: Compact Source Files and Instance Main Methods (Java 25).

KISIM III

Java ile Programlama

Bölüm 6–12

KISIM III · BÖLÜM 6

Değişkenler, Veri Tipleri ve İşleçler

Her program veriyle çalışır: bir öğrencinin yaşı, bir ürünün fiyatı, bir sınavın ortalaması. Program bu verileri bellekte adlandırılmış kutularda, yani değişkenlerde tutar. Her kutunun bir tipi vardır ve tip, kutuya hangi değerlerin sığacağını belirler. Bu not, Java'da değişken bildirmeyi ve değer vermeyi, sekiz ilkel tipi ve aralıklarını, sınır aşıldığında ortaya çıkan taşmayı ve ondalık sayıların neden tam saklanamadığını anlatır. Ardından tipler arasındaki dönüşümler, tam sayı bölmesi, aritmetik, karşılaştırma ve mantık işleçleri, işleç önceliği ve kısa devre değerlendirme ele alınır. Son kesimler klavyeden Scanner ile veri okumayı, printf ile biçimli çıktı üretmeyi ve bu araçları birleştiren küçük programları içerir. Nottaki bütün Java kodları JDK 21 ile derlenip çalıştırılmıştır; verilen çıktılar programların gerçek çıktısıdır.

Bu bölümün kodu: `github.com/sercankulcu/computer-programming-java/Bolum06/src`

ÖĞRENME ÇIKTILARI

Değişken
Değişken bildirmek, ilk değer vermek ve değişkenin değerini güncellemek.

Tip
İlkel tipleri ve değer aralıklarını tanımak; taşma ve kayan nokta hatası riskine göre uygun tipi seçmek.

İfadeler
Atama, tip dönüşümü ve işleçleri doğru kullanmak; bir ifadenin değerini öncelik ve tip kurallarıyla adım adım hesaplamak.

Girdi/Çıktı
Scanner ile klavyeden girdi almak, printf ile biçimli çıktı üretmek.

6.1 Değişkenler

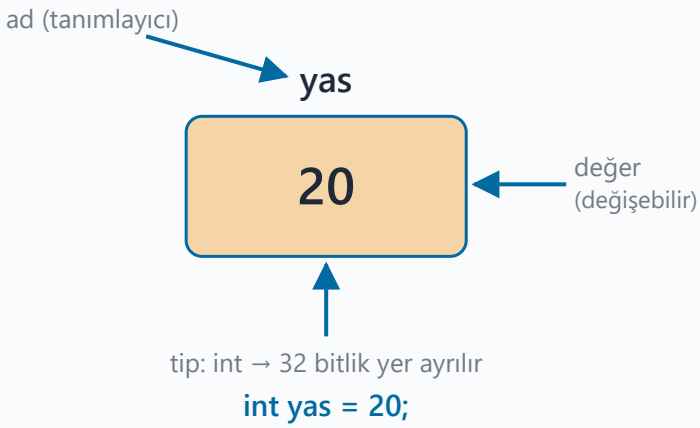
Bir hesap makinesiyle 70, 85 ve 90 notlarının ortalamasını alan biri, önce üç sayıyı toplar, ara sonucu ekranda tutar, sonra 3'e böler. Ara sonucu bir kâğıda yazmak da mümkündür. Bilgisayar programı da aynı ihtiyacı duyar: okuduğu sayıları, ara sonuçları ve son cevabı bir yerde saklamalıdır. Bu saklama

yeri bellektir. Program, bellekte ayırdığı her yere bir ad verir ve o yere adıyla ulaşır.

6.1.1 Değişken nedir?

Tanım 6.1

Değişken (*variable*), belirli tipteki bir değer için bellekte ayrılmış adlı bir kutudur. Değişkenin üç özelliği vardır: **adı**, **tipi** (*type*) ve o anki **değeri**. Ad ve tip program boyunca sabit kalır; değer ise değişebilir.



Şekil 6.1: `int yas = 20;` satırının bellekteki karşılığı. Kutunun adı `yas`, tipi `int`, değeri 20'dir. int tipi için 32 bitlik yer ayrılır.

Şekil 6.1 bu üç özelliği bir arada gösterir. Kutunun üstündeki `yas` adı, programcının bu kutuya verdiği addır; Java'daki adıyla bir **tanımlayıcıdır** (*identifier*). Kutunun içindeki 20 değeri program çalışırken değişebilir: yeni bir yıl başladığında yaş 21 olur. Kutunun altındaki `int` ise tiptir. Tip iki şeyi belirler. Birincisi, kutunun boyutudur: int için 32 bit, yani 4 bayt ayrılır. İkincisi, kutuya konabilecek değerlerdir: int bir tam sayı tipidir, bu kutuya 20 konabilir ama 20,5 ya da "yirmi" konamaz.

"Değişken" adı matematikten gelir ama anlamı biraz farklıdır. Matematikte $x = x + 1$ çözümü olmayan bir denklemdir. Programlamada ise bu satır bir komuttur: "x kutusundaki değeri oku, 1 ekle, sonucu yine x kutusuna yaz." Değişken, zamanla farklı değerler tutabilen bir kutudur; eşitlik işareti de bir eşitlik iddiası değil, bir yazma işlemidir.

6.1.2 Bildirim ve ilk değer

Java'da bir değişken kullanılmadan önce bildirilmelidir. Bildirim, derleyiciye "bu adla, bu tipte bir kutu istiyorum" demektir.

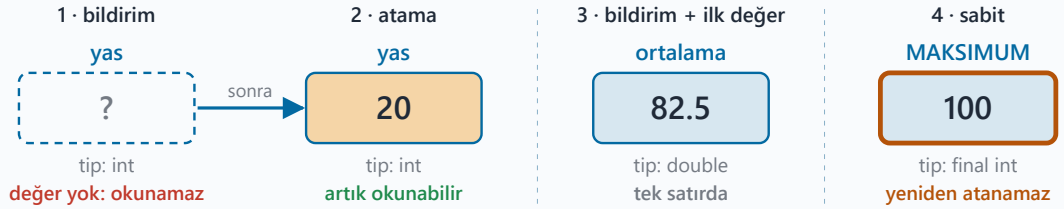
Tanım 6.2

Bildirim (*declaration*), bir değişkenin tipini ve adını belirten deyimdir: `int yas;`. **İlk değer verme** (*initialization*) ise değişkenin başlangıç değerini belirtir. İkisi tek deyimde de yazılabilir: `double ortalama = 82.5;`.

Sunumdaki dört satır, bildirimin farklı biçimlerini gösterir:

```
int yas;          // bildirim
yas = 20;         // atama
double ortalama = 82.5; // birlikte
final int MAKSIMUM = 100; // sabit
```

Birinci satır yalnızca bildirimdir: `yas` adlı int kutusu ayrılır ama içine henüz bir değer konmamıştır. İkinci satır bir **atamadır** (*assignment*): 20 değeri `yas` kutusuna yazılır. Üçüncü satır bildirimle ilk değeri birleştirir; `ortalama` kutusu ayrılır ve aynı anda 82.5 değerini alır. Dördüncü satırdaki `final` sözcüğü kutuyu bir **sabite** çevirir: `MAKSIMUM` bir kez 100 değerini alır ve program boyunca bu değeri korur.



Şekil 6.2: Dört satırın bellekteki etkisi. Bildirimden sonra `yas` kutusunun değeri yoktur ve okunamaz; atamadan sonra 20'dir. `ortalama` tek satırda bildirilip değer alır. `final` ile bildirilen `MAKSIMUM` yeniden atanamaz.

Java, bir metodun içinde bildirilen değişkenlere **yerel değişken** (*local variable*) der. Yerel değişkenler için katı bir kural vardır: değişken okunmadan önce kesin olarak bir değer almış olmalıdır. Derleyici bu kuralı programı çalıştırmadan, kodu okuyarak denetler. Aşağıdaki iki satırlık parça bu yüzden derlenmez:

```
int yas;
System.out.println(yas);
```

```
error: variable yas might not have been initialized
System.out.println(yas);
                   ^
1 error
```

İleti "yas değişkenine değer verilmemiş olabilir" der. Derleyici `yas` kutusunun boş olduğunu görür ve programı hiç üretmez. Bu kural bir güvenlik önlemidir. Kural olmasaydı program, kutuda o an rastlantıyla bulunan bir değeri okuyabilir ve anlamsız sonuçlar üretebilirdi. Bazı eski dillerde (örneğin C'de) bu durum gerçekten yaşanır.

`final` değişkenler için de benzer bir denetim vardır. Değer bir kez verildikten sonra ikinci atama derleme hatasıdır:

```
final int MAKSIMUM = 100;
MAKSIMUM = 200;
```

```
error: cannot assign a value to final variable MAKSIMUM
MAKSIMUM = 200;
^
1 error
```

Sabitler, programda birden çok yerde geçen ve değişmemesi gereken değerler için kullanılır: en yüksek not, KDV oranı, bir dizinin boyutu. Sayıyı her yerde ayrı ayrı yazmak yerine bir sabit tanımlamak iki yarar sağlar. Kod okunur hâle gelir; `puan <= MAKSIMUM` ifadesi `puan <= 100` ifadesinden daha çok şey anlatır. Ayrıca değer değişirse yalnızca tek satır düzeltilir.

Sık yapılan hata

Bildirilen bir değişkenin otomatik olarak 0 değerini aldığını sanmak. Java'da yerel değişkenler kendiliğinden değer almaz; `int toplam;` satırından sonra `toplam += 5;` yazmak "might not have been initialized" derleme hatası verir. Toplam tutacak bir değişken açıkça sıfırla başlatılır: `int toplam = 0;`.

6.1.3 Değişkenler bellekte adım adım

Değişkenlerin nasıl değer aldığını ve değiştirdiğini görmenin en iyi yolu, bir programı satır satır izlemektir. Sunumdaki örnek program şudur:

```
int yas;                  // bildirim
yas = 20;                 // ilk atama
yas = yas + 1;            // güncelleme
double ortalama = 82.5;   // bildirim + ilk değer
ortalama += 5;            // bileşik atama
final int MAKSIMUM = 100; // sabit
System.out.println(yas);
System.out.println(ortalama);
```

Program iki satır çıktı üretir:

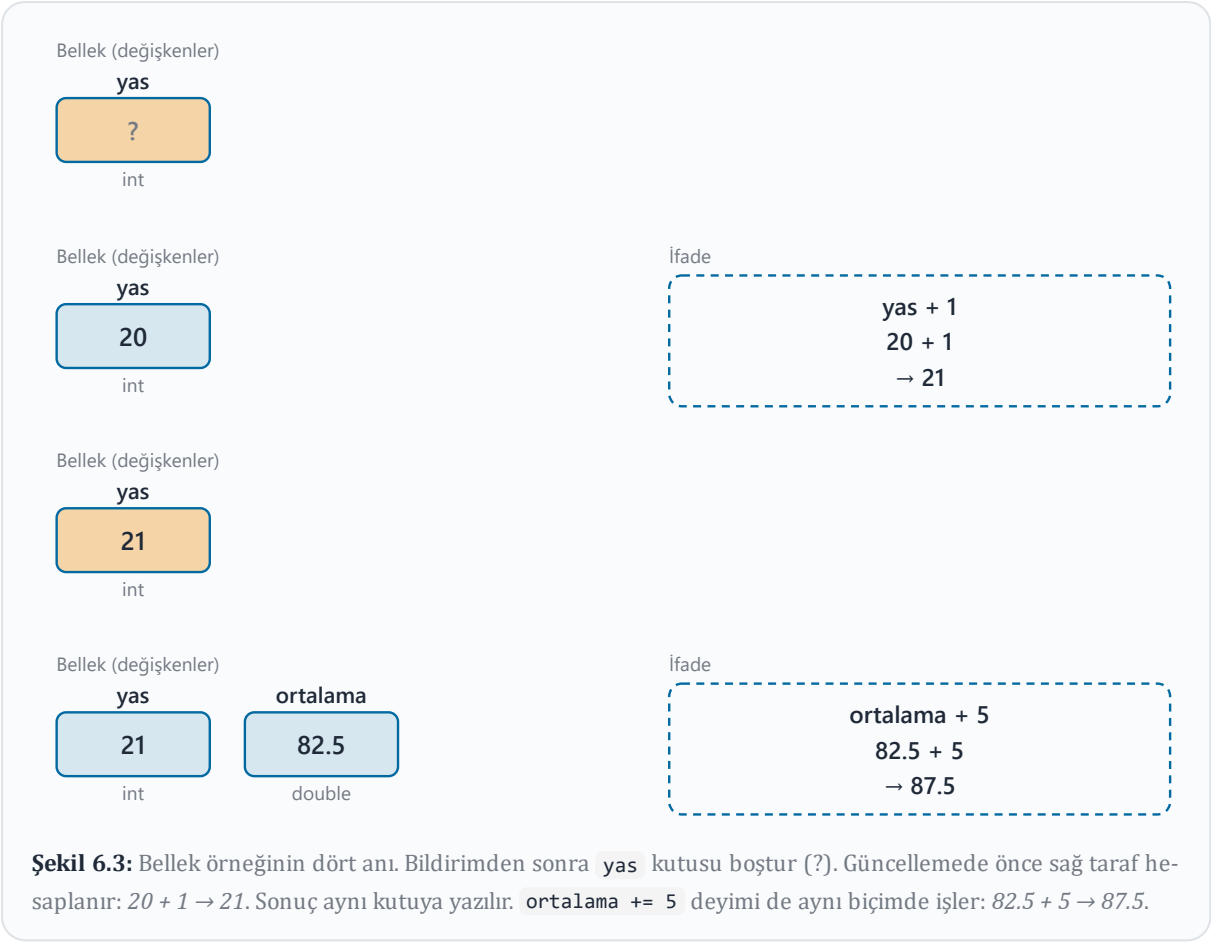
```
21
87.5
```

Tablo 6.1, her satır çalıştıktan sonra kutuların durumunu verir. Soru işareti, kutunun ayrıldığını ama henüz değer almadığını; tire, kutunun henüz hiç ayrılmadığını gösterir.

Satır	Deyim	yas	ortalama	MAKSIMUM	Çıktı
1	`int yas;`	?	–	–	
2	`yas = 20;`	20	–	–	
3	`yas = yas + 1;`	21	–	–	
4	`double ortalama = 82.5;`	21	82.5	–	
5	`ortalama += 5;`	21	87.5	–	
6	`final int MAKSIMUM = 100;`	21	87.5	100	
7	`System.out.println(yas);`	21	87.5	100	21
8	`System.out.println(ortalama);`	21	87.5	100	87.5

Tablo 6.1: Bellek örneğinin izleme tablosu. Her satır, o satırdaki deyim çalıştıktan sonraki durumu gösterir.

Üçüncü satır dikkatle okunmalıdır. `yas = yas + 1;` deyimi iki aşamada çalışır. Önce eşittir işaretinin sağındaki ifade hesaplanır: `yas` kutusundaki 20 okunur ve 1 eklenir, sonuç 21'dir. Bu aşamada kutu henüz değişmemiştir. İkinci aşamada 21 değeri soldaki kutuya yazılır; eski 20 silinir. Bir kutu aynı anda yalnızca bir değer tutar ve yeni değer eskisinin üzerine yazılır.



Şekil 6.3: Bellek örneğinin dört anı. Bildirimden sonra `yas` kutusu boştur (?). Güncellemede önce sağ taraf hesaplanır: *20 + 1 → 21*. Sonuç aynı kutuya yazılır. `ortalama += 5` deyimi de aynı biçimde işler: *82.5 + 5 → 87.5*.

Yedinci ve sekizinci satırlar kutuları yalnızca okur. Okuma bir değeri kutudan kopyalar ama kutuyu değiştirmez; `System.out.println(yas)` satırından sonra `yas` hâlâ 21'dir. Son durumda bellekte üç kutu vardır ve ekranda iki satır çıktı görünür. Altıncı satırdan sonra `MAKSIMUM = 200;` yazılsaydı program derlenmezdi; final değişkenin değeri bir kez verilir.

6.1.4 Atama ve güncelleme

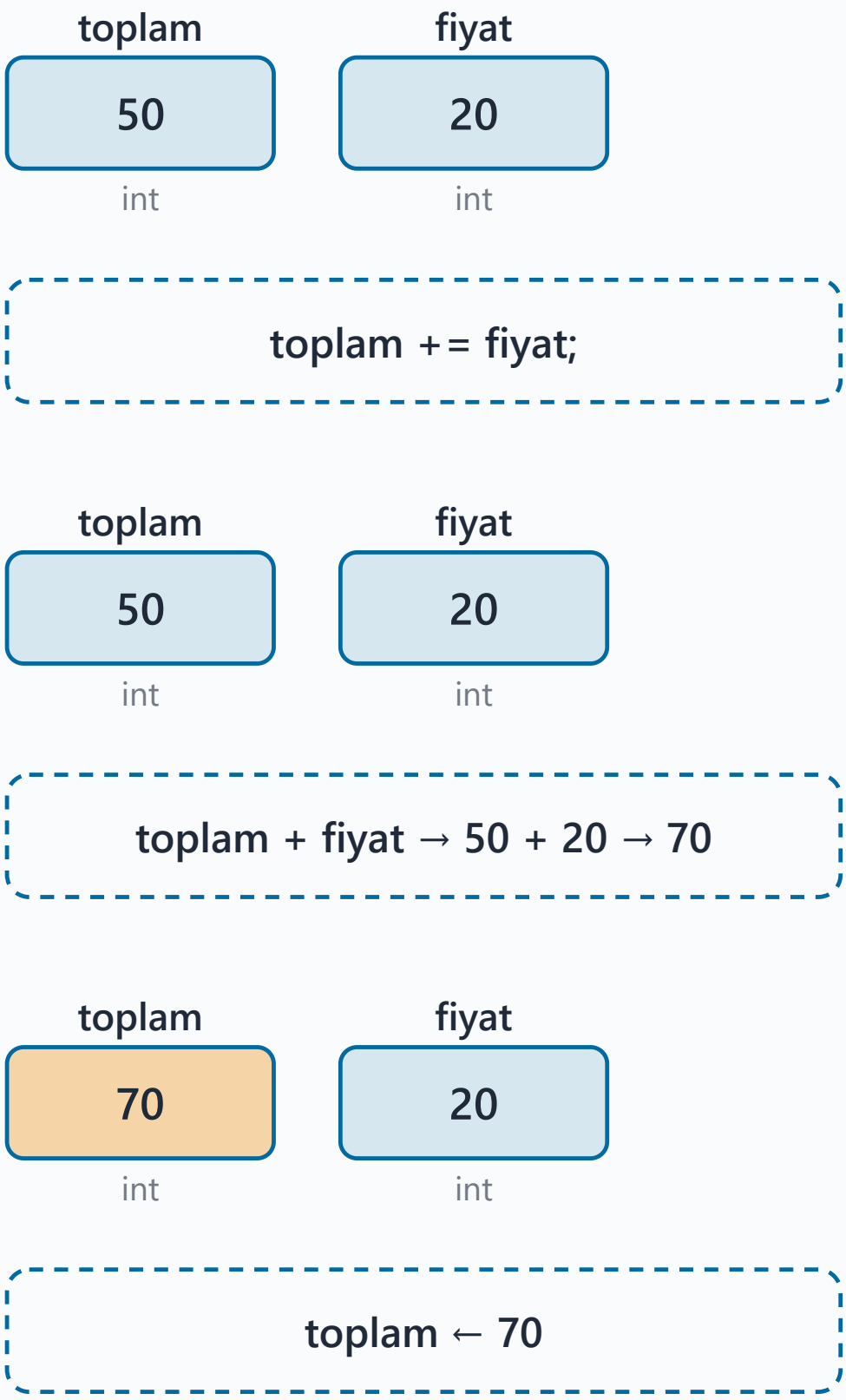
Atama işleci `=`, Java'nın en çok kullanılan işlecidir ve her zaman aynı sırayla çalışır: önce sağ taraf hesaplanır, sonra değer soldaki değişkene yazılır. Sol tarafta her zaman tek bir değişken bulunur. `5 = x;` ya da `x + 1 = 7;` gibi satırlar anlamsızdır, çünkü 5 ya da `x + 1` bir kutu değildir.

Atanan değer, değişkenin tipiyle uyumlu olmalıdır. int bir kutuya `"Ada"` dizgisi atanamaz; derleyici bunu tip uyuşmazlığı hatası olarak bildirir.

Programlarda bir değişkenin değerini, kendi eski değerini kullanarak değiştirmek çok sık yapılır: toplama yeni bir fiyat eklemek, sayacı bir artırmak, bakiyeden bir tutar düşmek. Bu iş için Java'da kısa bir yazım vardır.

Tanım 6.3

Bileşik atama (*compound assignment*), bir aritmetik işlemi atamayla birleştiren işleçtir: `+=`, `-=`, `*=`, `/=`, `%=`. `toplam += fiyat` deyimi `toplam = toplam + fiyat` anlamına gelir.



Şekil 6.4: `toplam += fiyat;` deyiminin üç adımı. Başta toplam = 50, fiyat = 20'dir. Önce iki kutu okunur ve *50 + 20 → 70* hesaplanır. Sonuç soldaki kutuya yazılır: toplam = 70. fiyat değişmez.

Şekil 6.4 işlemin adımlarını gösterir. Sağdaki değişken (`fiyat`) yalnızca okunur, soldaki değişken (`toplam`) hem okunur hem yazılır. Bileşik atamanın bir inceliği daha vardır: sağ taraf, işlemden önce bütünüyle hesaplanır. `i *= 3 + 7` deyimi `i = i * 3 + 7` değil, `i = i * (3 + 7)` demektir. i = 5 için sonuç 22 değil 50'dir.

Örnek 6.1

Bir alışveriş sepetinde toplam 0 TL'den başlar. Sırayla 45 TL'lik ve 30 TL'lik iki ürün eklenir, sonra 10 TL indirim uygulanır, en son tutar 2 taksite bölünür. Hangi deyimler yazılır ve her adımdan sonra `toplam` kaç olur?

Deyim	Hesap	toplam
`int toplam = 0;`	ilk değer	0
`toplam += 45;`	0 + 45	45
`toplam += 30;`	45 + 30	75
`toplam -= 10;`	75 − 10	65
`toplam /= 2;`	65 / 2 (tam sayı bölmesi)	32

Son satırdaki sonuç 32,5 değil 32'dir. `toplam` bir int olduğu için bölme tam sayı bölmesidir ve kesir atılır. Taksit tutarının kuruşuyla hesaplanması gerekiyorsa değişken baştan `double` seçilmelidir. İki tam sayının bölümü Java'da her zaman tam sayıdır.

6.1.5 Tanımlayıcı kuralları ve adlandırma

Değişkenlere, metotlara ve sınıflara verilen adların hepsi tanımlayıcıdır. Java, bir tanımlayıcının nasıl yazılabileceğini birkaç kesin kuralla sınırlar. Bu kurallara uymayan bir ad derleme hatası verir.

Kural	Geçerli	Geçersiz
Harf, _ veya $ ile başlar	`ogrenci1`	`1ogrenci`
Boşluk içermez	`toplamPuan`	`toplam puan`
Anahtar sözcük değildir	`sinifNo`	`class`
Tek alt çizgi ad olamaz	`_deger`	`_`
Büyük-küçük harfe duyarlı	`puan` ve `Puan`	Aynı değişken sanmak

Tablo 6.2: Tanımlayıcı kuralları (sunumdaki tablo).

Tanımlayıcı bir harfle, alt çizgiyle (_) ya da dolar işaretiyle ($) başlar; sonraki karakterler bunlara ek olarak rakam da olabilir. Rakamla başlayan bir ad derleyiciyi şaşırtır. `int 1ogrenci = 5;` satırında derleyici `int`'ten sonra bir ad bekler ama 1 rakamıyla karşılaşır ve satırı anlamlandıramaz:

```
error: not a statement
int 1ogrenci = 5;
^
error: ';' expected
int 1ogrenci = 5;
   ^
2 errors
```

İletiler sorunu doğrudan söylemez; "bu bir deyim değil" ve "noktalı virgül bekleniyordu" der. Derleyici hatayı bulduğu yerde, kendi anladığı biçimde bildirir. Bu yüzden bir derleme iletisi okunurken iletinin işaret ettiği satırın tamamına bakılmalıdır.

Anahtar sözcükler (*keyword*), Java'nın kendi işleri için ayırdığı sözcüklerdir: `int`, `double`, `class`, `if`, `final`, `new`, `return` bunlardan birkaçıdır. Bu sözcükler ad olarak kullanılamaz; `int class = 5;` satırı üç ayrı hata iletisi üretir. Tek başına alt çizgi de JDK 21'de ad değildir. `int _ = 3;` satırını derleyici "adsız değişkenler önizleme özelliğidir" iletisiyle reddeder. Daha yeni Java sürümlerinde `_` "adsız değişken" anlamına gelir; bu durumda da değişkene adıyla ulaşılamaz.

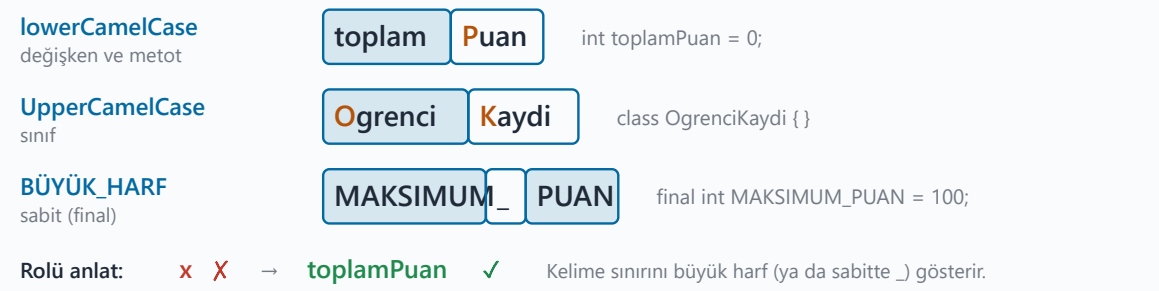
Son kural en sinsi olanıdır. Java büyük ve küçük harfi farklı karakter sayar. `puan` ve `Puan` iki ayrı değişkendir ve aynı programda yan yana yaşayabilir:

```
int puan = 70;
int Puan = 90;
System.out.println(puan + " " + Puan);
```

```
70 90
```

Program hatasız derlenir ve iki farklı değer yazar. Programcı aynı değişkeni kullandığını sanıyorsa hata sessizce oluşur. Bu yüzden büyük-küçük harf kullanımı rastgele bırakılmaz; adlandırma kuralları bu tutarlılığı sağlar.

Derleyicinin denetlediği kuralların yanında, Java programcılarının uyduğu yazılı olmayan **adlandırma kuralları** (*naming convention*) da vardır. Bu kurallara uymayan kod derlenir ama okuyan herkesi yorar.



Şekil 6.5: Üç adlandırma biçimi. Değişken ve metot adları küçük harfle başlar ve her yeni kelime büyük harfle gelir (`toplamPuan`). Sınıf adları büyük harfle başlar (`OgrenciKaydi`). Sabitler büyük harflerle yazılır ve kelimeler alt çizgiyle ayrılır (`MAKSIMUM_PUAN`).

- **lowerCamelCase**: değişken ve metot adları. İlk kelime küçük harfle başlar, sonraki her kelimenin ilk harfi büyüktür: `toplamPuan` , `ogrenciSayisi` , `ortalamaHesapla` . Kelimeler arasındaki büyük harfler, devenin hörgüçleri gibi kelime sınırını gösterir.
- **UpperCamelCase**: sınıf adları. Her kelime büyük harfle başlar: `OgrenciKaydi` , `Scanner` , `String` .
- **BÜYÜK_HARF_VE_ALT_ÇİZGİ**: `final` sabitler. `MAKSIMUM_PUAN` , `KDV_ORANI` .

Bir adın en önemli niteliği, değerin rolünü anlatmasıdır. `x` , `a1` , `temp2` gibi adlar, kodu yazan kişiye o an anlamlı gelir; bir hafta sonra kimseye bir şey söylemez. `toplamPuan` adı ise hem tipini hem görevini sezdirir. Türkçe karakterler (ç, ğ, ı, ö, ş, ü) Java tanımlayıcılarında geçerlidir, ama derste ve çoğu projede ASCII harflerle yazmak tercih edilir: `ogrenci` , `sinif` , `yas` . Bu tercih, farklı karakter kodlamalı bilgisayarlar arasında dosya taşınırken çıkabilecek sorunları önler.

> **Sık yapılan hata**
>
> Bir yerde `toplamPuan` , başka bir yerde `toplampuan` yazmak. İkinci ad bildirilmemişse derleyici “cannot find symbol” (simge bulunamadı) hatası verir. Daha kötüsü, iki ad da bildirilmişse program derlenir ve değerler farklı kutulara gider. Adlandırma kurallarına her yerde aynı biçimde uymak, bu tür hataların çoğunu baştan önler.

6.2 Veri tipleri

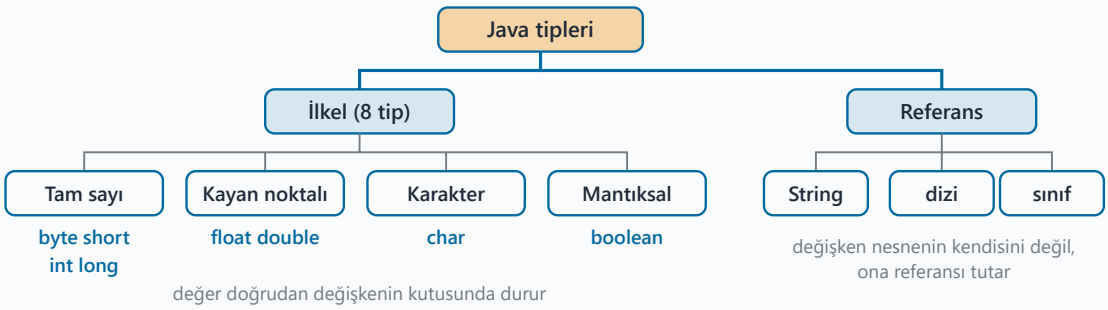
Bellek, en alt düzeyde yalnızca bitlerden, yani 0 ve 1'lerden oluşur. 01000001 bit dizisi bir yerde 65 sayısı, başka bir yerde 'A' harfi anlamına gelebilir. Bitlerin nasıl yorumlanacağını tip belirler. Java, her değişkenin tipinin derleme sırasında bilinmesini ister. Bu yüzden Java **katı tipli** bir dildir: derleyici her işlemin tiplere uygun olup olmadığını program çalışmadan önce denetler ve pek çok hatayı daha ilk derlemede yakalar.

6.2.1 Temel tip grupları

Java tipleri iki büyük aileye ayrılır. **İlkel tipler** (*primitive type*) dilin içine gömülü sekiz temel tiptir. İlkel bir değişkenin kutusunda değerin kendisi durur. **Referans tipleri** ise dizgiler (String), diziler ve sınıflardır. Referans tipinde bir değişkenin kutusunda nesnenin kendisi değil, bellekte başka bir yerde duran nesneye giden bir **referans** (*reference*) bulunur.

Grup	Örnek	Kullanım
Tam sayı	`byte`, `short`, `int`, `long`	Sayma ve ayrık değer
Kayan noktalı	`float`, `double`	Ölçüm ve yaklaşık hesap
Karakter	`char`	Tek UTF-16 kod birimi
Mantıksal	`boolean`	`true` / `false`
Referans	String, dizi, sınıf	Nesneye referans

Tablo 6.3: Temel tip grupları (sunumdaki tablo).



Şekil 6.6: Java tiplerinin iki ailesi. Sekiz ilkel tip dört gruba ayrılır; bunlarda değer doğrudan değişkenin kutusunda durur. String, diziler ve sınıflar referans tipleridir; değişken nesnenin kendisini değil, ona giden referansı tutar.

Gruplar, verinin doğasına göre seçilir. Bir sınıftaki öğrenci sayısı, bir dizideki konum, bir sayaç gibi **sayılan** değerler tam sayıdır. Boy, sıcaklık, fiyat gibi **ölçülen** değerler kayan noktalı sayıyla tutulur. Tek bir harf için char, evet/hayır cevabı için boolean kullanılır. Ad, adres, cümle gibi metinler ise String nesneleridir. Bu notta ağırlıklı olarak ilkel tipler ele alınır; String yalnızca birleştirme ve girdi okuma bağlamında görünür.

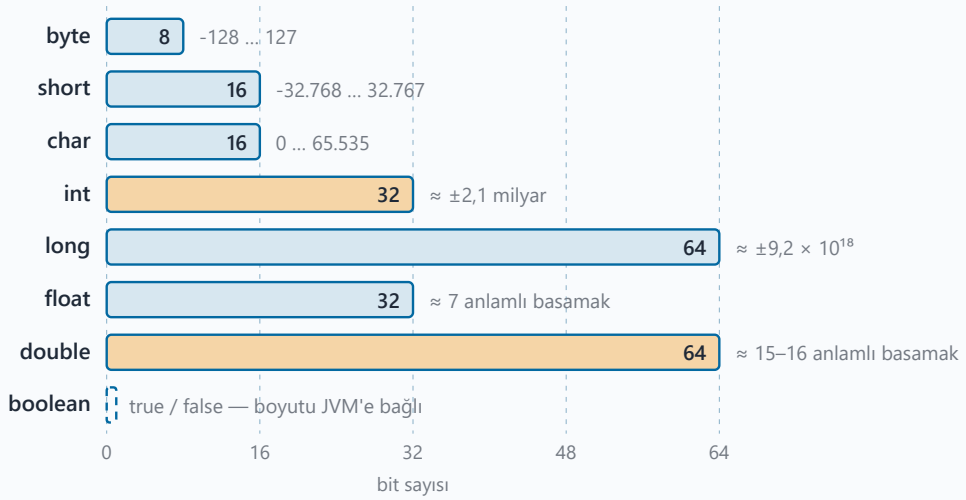
6.2.2 İlkel tipler ve boyutları

Sekiz ilkel tipin her biri bellekte sabit sayıda bit kaplar. Bu sayı Java'da her bilgisayarda ve her işletim sisteminde aynıdır. C gibi bazı dillerde int'in boyutu makineye göre değişebilir; Java'da int her yerde 32 bittir. Bu kararlılık, aynı programın her yerde aynı sonucu vermesini sağlar.

Tip	Bit	En küçük	En büyük	Sabit örneği
`byte`	8	−128	127	`byte seviye = 7;`
`short`	16	−32.768	32.767	`short yil = 2024;`

Tip	Bit	En küçük	En büyük	Sabit örneği
`int`	32	−2.147.483.648	2.147.483.647	`int adet = 42;`
`long`	64	−9.223.372.036.854.775.808	9.223.372.036.854.775.807	`long nufus = 8_000_000L;`
`float`	32	yaklaşık ±3,4 × 10^{38}; ≈ 7 anlamlı basamak		`float oran = 1.5F;`
`double`	64	yaklaşık ±1,8 × 10^{308}; ≈ 15–16 anlamlı basamak		`double fiyat = 19.90;`
`char`	16	0	65.535	`char not = 'A';`
`boolean`	JVM'e bağlı	`true` ya da `false`		`boolean aktif = true;`

Tablo 6.4: İlkel tipler, boyutları ve sınırları. Sınır değerleri JDK 21'de `Byte.MAX_VALUE` gibi sabitler yazdırılarak alınmıştır.



Şekil 6.7: İlkel tiplerin bit sayıları. Vurgulu çubuklar varsayılan seçimlerdir: tam sayı için int, kayan noktalı sayı için double. 1 bayt = 8 bittir.

Bir tipin aralığı bit sayısından hesaplanır. n bitle 2^n farklı bit dizisi yazılabilir. byte için $2^8 = 256$ farklı değer vardır. Java tam sayı tipleri işaretlidir: bu değerlerin yarısı negatif sayılara, yarısı sıfır ve pozitif sayılara ayrılır. Sıfır pozitif tarafta yer kapladığı için aralık −128'den 127'ye uzanır; 128 sayısı byte'a sığmaz. Aynı hesap int için $-2^{31} \ldots 2^{31} - 1$ aralığını verir:

> 2^{31} = 2.147.483.648, dolayısıyla int aralığı −2.147.483.648 … 2.147.483.647

Kabaca akılda tutmak için int'in sınırı “yaklaşık ±2,1 milyar”, long'unki “yaklaşık ±9,2 kentilyon” ($9{,}2 \times 10^{18}$) olarak hatırlanabilir. char tipi 16 bitlik olmasına karşın işaretsizdir; negatif değeri yoktur ve 0 ile 65.535 arasındaki kodları tutar.

Programda doğrudan yazılan değerlere **sabit** (*literal*) denir: `42`, `19.90`, `'A'`, `true`. Sabitlerin de tipi vardır ve bu tip yazılış biçiminden anlaşılır:

- Noktasız bir tam sayı sabiti (`42`) int'tir. Sonuna `L` eklenirse long olur (`42L`). Küçük `l` de geçerlidir ama 1 rakamına çok benzediği için kullanılmaz.
- Noktalı bir sabit (`19.90`) ya da bilimsel gösterimle yazılmış bir sabit (`3.9e10` , yani $3{,}9 \times 10^{10}$) double'dır. Sonuna `F` eklenirse float olur (`1.5F`).
- Uzun sayılarda basamakları gruplamak için alt çizgi kullanılabilir: `8_000_000_000L` . Alt çizgiler yalnızca okumayı kolaylaştırır; değeri değiştirmez.

Java kodunda ondalık ayırıcı her zaman noktadır. `19.90` Java'da on dokuz virgül doksandır; `19,90` ise sözdizimi hatasıdır. Bu not, Türkçe metin içinde ondalık virgül kullanır (19,90 TL), ama kod parçalarında ve program çıktılarında Java'nın yazdığı biçimi aynen korur.

Örnek 6.2

Dünya nüfusu (yaklaşık 8 milyar) bir değişkende tutulacak. `int nufus = 8_000_000_000;` satırı neden derlenmez? `long nufus = 8_000_000_000;` satırı neden yine derlenmez?

8 milyar, int'in en büyük değeri olan 2.147.483.647'den büyüktür; değer int kutusuna sığmaz. Ama asıl sorun daha önce başlar: noktasız ve L'siz yazılan `8_000_000_000` sabitinin kendisi int tipindedir. Derleyici bu sabiti int olarak okumaya çalışırken sınırı aşar ve sol taraftaki tipe bakmadan hata verir:

```
error: integer number too large
long nufus = 8_000_000_000;
             ^
1 error
```

Hedef long olsa bile ileti aynıdır. Doğru yazım, sabitin kendisini L ile long yapmaktır: `long nufus = 8_000_000_000L;` . Bu satır derlenir ve `System.out.println(nufus)` 8000000000 yazar.

Kayan noktalı sabitlerde de benzer bir kural vardır. `1.5` bir double sabitidir; `float oran = 1.5;` satırı "possible lossy conversion from double to float" (double'dan float'a olası kayıplı dönüşüm) iletisiyle derlenmez. float istendiğinde sabit `1.5F` yazılır. Çoğu programda bu soru hiç doğmaz, çünkü varsayılan seçim zaten double'dır.

6.2.3 Tam sayılar bellekte nasıl durur?

Tam sayı tipi seçerken çoğu durumda int yeterlidir ve varsayılan seçim odur. long, int'in yetmediği büyük değerler için kullanılır: dünya nüfusu, milisaniye cinsinden zaman, banka hesabındaki kuruş tutarı. byte ve short ise genellikle büyük veri dizilerinde bellekten kazanmak için seçilir. Bir hesapta

beklenen en büyük değer önceden kestirilmelidir; int'in sınırına yaklaşabilecek bir değer baştan long seçilir. Bu kararın önemi, sınır aşıldığında olanlarda yatar. Bunu anlamak için tam sayıların bitlerle nasıl gösterildiğine kısaca bakmak gerekir.

Java tam sayıları **ikiye tümleyen** (*two's complement*) gösterimle saklar. Bu gösterimde en soldaki bit **işaret bitidir** (*sign bit*): 0 ise sayı sıfır ya da pozitif, 1 ise negatiftir. byte üzerinde birkaç örnek:

Bitler	İşaretsiz okunuş	byte değeri
`0000 0000`	0	0
`0000 0001`	1	1
`0111 1111`	127	127 (en büyük)
`1000 0000`	128	128 − 256 = −128 (en küçük)
`1000 0010`	130	130 − 256 = −126
`1100 1000`	200	200 − 256 = −56
`1111 1111`	255	255 − 256 = −1

Tablo 6.5: Bazı byte değerlerinin bit gösterimi. İşaret biti 1 olan bit dizisinin değeri, işaretsiz değerden 256 çıkarılarak bulunur.

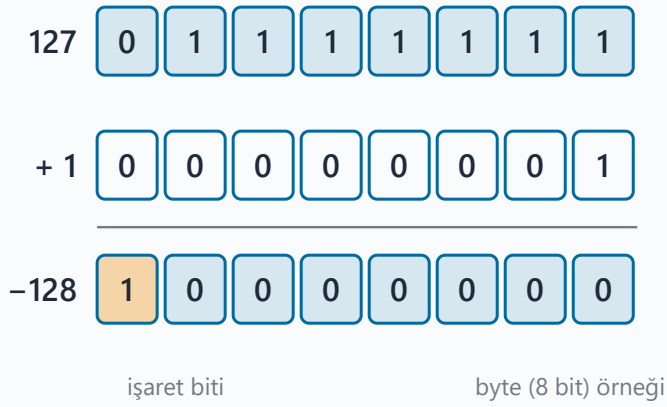
Tablonun son sütunu basit bir kurala dayanır: işaret biti 1 olan 8 bitlik bir dizi, işaretsiz değerinden 256 çıkarılarak okunur. Aynı kural int için 2^{32} ile, long için 2^{64} ile geçerlidir. Bu kural, taşmayı ve daraltma dönüşümlerini hesaplamak için yeterlidir.

6.2.4 Taşma

Bir tam sayı işleminin sonucu tipin aralığına sığmadığında ne olur? Bir arabanın kilometre sayacı 999.999'dan sonra 000.000'a döner. Java tam sayıları da benzer davranır: sonuç aralığın bir ucundan taşar ve öbür uçtan geri girer.

Tanım 6.4

Taşma (*overflow*), bir tam sayı işleminin sonucunun tipin aralığını aşmasıdır. Java'da taşma hata vermez. Sonucun yalnızca tipe sığan alt bitleri saklanır; değer sessizce aralığın öbür ucuna sarar.



Şekil 6.8: byte için taşma. 127'nin bitleri 0111 1111'dir; 1 eklenince 1000 0000 olur. İşaret biti 1 olduğu için bu dizi −128 diye okunur.

Sunumdaki örnek program iki tipte de taşmayı gösterir:

```
int x = Integer.MAX_VALUE;
x++;
byte b = 127;
b++;
System.out.println(x + " " + b);
```

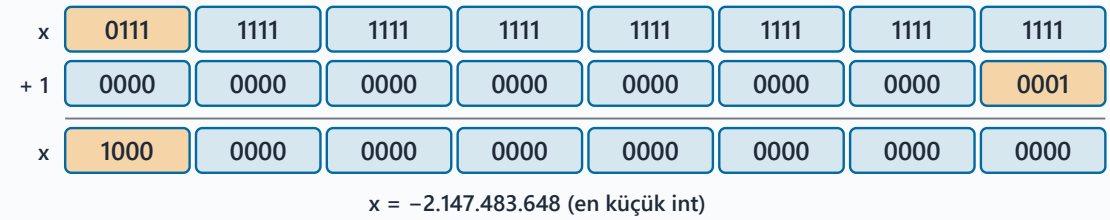
```
-2147483648 -128
```

Program satır satır şöyle çalışır:

Satır	Deyim	x	b	Açıklama
1	`int x = Integer.MAX_VALUE;`	2147483647	–	bitler 0111 1111 … 1111
2	`x++;`	−2147483648	–	bitler 1000 0000 … 0000
3	`byte b = 127;`	−2147483648	127	bitler 0111 1111
4	`b++;`	−2147483648	−128	bitler 1000 0000
5	`System.out.println(x + " " + b);`	−2147483648	−128	çıktı: -2147483648 -128

Tablo 6.6: Taşma örneğinin izleme tablosu.





Şekil 6.9: int için taşma. Integer.MAX_VALUE'da işaret biti 0, öteki 31 bit 1'dir. 1 eklenince elde bütün bitler boyunca ilerler, alttaki 31 bit 0 olur ve işaret biti 1'e döner. Sonuç en küçük int'tir: −2.147.483.648.

İkili toplama, onluk sistemdeki elde mantığıyla çalışır. 0111…1 dizisine 1 eklenince en sağdaki 1 ile 1'in toplamı 0 olur ve 1 elde kalır. Elde sola doğru bütün 1'leri sıfırlayarak ilerler ve en sonunda işaret bitine ulaşır. Sonuç 1000…0'dır. Matematiksel olarak doğru cevap 2.147.483.648'dir, ama bu değer int aralığının dışındadır. İşaret biti 1 olan dizinin değeri *2.147.483.648 − 2^{32} = −2.147.483.648* çıkar. Kilometre sayacı benzetmesiyle, sayı aralığın tepesinden taşar ve dibinden geri girer.

Taşmanın tehlikesi sessiz olmasıdır. Program durmaz, uyarı vermez; yanlış sayıyla hesaba devam eder. Taşma yalnızca sınıra bir eklemekle de olmaz. İki makul görünen sayının çarpımı bile int'i aşabilir:

```
int a = 50000;
int b = 50000;
System.out.println(a * b);
System.out.println((long) a * b);
System.out.println(a * b / 1000);
long yanlis = a * b;
System.out.println(yanlis);
```

```
-1794967296
2500000000
-1794967
-1794967296
```

50.000 × 50.000 = 2.500.000.000'dır ve int sınırını aşar. İlk satır bu yüzden negatif bir sayı yazar: *2.500.000.000 − 2^{32} = −1.794.967.296*. İkinci satırda `(long) a` dönüşümü çarpmadan önce yapılır; işlem long ile yürür ve doğru sonuç çıkar. Üçüncü satır, taşmanın sonraki işlemlere de bulaştığını gösterir: yanlış çarpım 1000'e bölünür ve yanlış bir bölüm elde edilir. Dördüncü satır en yanıltıcı olanıdır. Sonuç long bir değişkene atanır ama çarpma yine int ile yapılır, çünkü işlemin iki **işleneni** (*operand*), yani üzerinde işlem yapılan iki değer de int'tir. Sonuç taşmış hâliyle long'a kopyalanır. Hedefin long olması, işlemin tipini değiştirmez.

Sık yapılan hata

`long sonuc = a * b;` yazınca çarpmanın long ile yapılacağını sanmak. Java bir ifadenin tipini yalnızca işlenenlerine bakarak belirler; sonucun nereye atanacağına bakmaz. İki int'in çarpımı int'tir ve gerekirse taşar. Doğrusu, işlenenlerden birini işlemden önce long yapmaktır: `long sonuc = (long) a * b;` ya da `a * 1L * b`.

6.2.5 Sınır sabitleri ve Math.addExact

Tiplerin sınır değerlerini elle yazmak hem zor hem tehlikelidir; 2147483647 sayısında tek bir basamak yanlış yazılabilir. Java bu değerleri hazır sabitler olarak sunar:

Sabit	Değer
`Integer.MIN_VALUE`	-2147483648
`Integer.MAX_VALUE`	2147483647
`Long.MIN_VALUE`	-9223372036854775808
`Long.MAX_VALUE`	9223372036854775807

Tablo 6.7: int ve long sınır sabitleri (sunumdaki tablo).

Benzer sabitler öteki tipler için de vardır: `Byte.MAX_VALUE`, `Short.MIN_VALUE`, `Double.MAX_VALUE`. Bu adlar hem okunaklıdır hem de yanlış yazılamaz.

Taşmanın sessizce geçmesi istenmiyorsa `Math` sınıfının "kesin" (*exact*) metotları kullanılır. `Math.addExact`, `Math.subtractExact` ve `Math.multiplyExact` işlemi yapar; sonuç tipe sığmıyorsa sessizce sarmak yerine bir **istisna** (*exception*) fırlatır. İstisna, programın olağan akışını durduran bir hata bildirimidir.

```
int x = Integer.MAX_VALUE;
System.out.println(x + 1);
// -2147483648 (sessiz taşma)
System.out.println(
    Math.addExact(x, 1));
// ArithmeticException:
// integer overflow
```

```
-2147483648
Exception in thread "main" java.lang.ArithmeticException: integer overflow
```

İlk `println` sessiz taşmayı gösterir ve −2147483648 yazar. İkinci satırda `Math.addExact(x, 1)` çağrılır; sonuç int'e sığmadığı için `ArithmeticException` fırlatılır ve program "integer overflow" iletisiyle durur. long değerlerde ileti "long overflow" olur. Programın durması ilk bakışta kötü görünür, ama yanlış bir sonuçla devam etmesinden çok daha iyidir: hata hemen fark edilir ve yeri bellidir.

Taşmanın şaşırtıcı bir örneği de mutlak değerdedir. `Math.abs(Integer.MIN_VALUE)`, beklenen 2.147.483.648 yerine yine −2147483648 verir. Nedeni aralığın simetrik olmamasıdır: en küçük int'in mutlak değeri olan 2^{31} int'e sığmaz. Negatifini almak taşmaya yol açar ve sayı kendisine geri döner. Aynı durum `-Integer.MIN_VALUE` ifadesinde de görülür.

6.2.6 Kayan noktalı sayılar

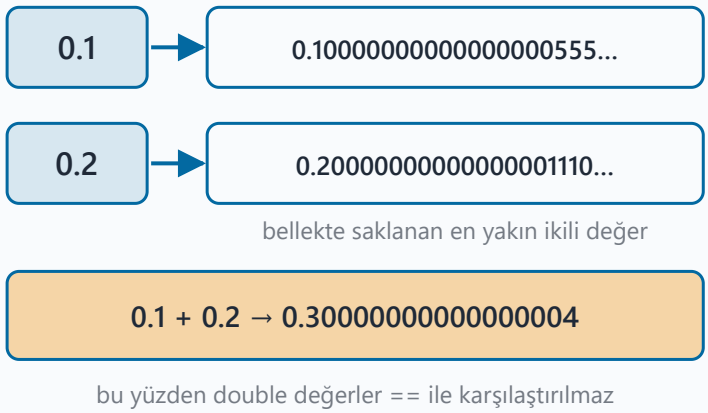
Ondalık kesirli sayılar Java'da **kayan noktalı sayı** (*floating-point number*) olarak saklanır. Ad, sayının bilimsel gösterime benzer bir biçimde tutulmasından gelir: bir dizi anlamlı basamak ve virgülün yerini belirten bir üs. *12.345,678 = 1,2345678 × 10^4* yazılışında virgül, üsse göre "kayar". Bu sayede aynı 64 bit hem çok küçük (10^{-300}) hem çok büyük (10^{300}) sayıları tutabilir.

double, çoğu yaklaşık hesap için varsayılan tiptir; yaklaşık 15–16 anlamlı basamağı vardır. float yarı boyuttadır ve yaklaşık 7 anlamlı basamak tutar. Aradaki fark kolayca görülür:

```
System.out.println(0.1 + 0.2);
System.out.println(1.0 - 0.9);
System.out.println(0.1 + 0.2 == 0.3);
System.out.println(0.1 + 0.2 - 0.3);
System.out.println(Math.abs(0.1 + 0.2 - 0.3) < 1e-9);
System.out.println(0.1f + 0.2f);
System.out.println(1.1f * 3);
System.out.println(1.1 * 3);
System.out.println(100.0 / 3);
System.out.println(100.0f / 3);
System.out.println(0.5 + 0.25 == 0.75);
System.out.println(new java.math.BigDecimal(0.1));
System.out.println(new java.math.BigDecimal(0.2));
System.out.println(new java.math.BigDecimal(0.1 + 0.2));
System.out.println(new java.math.BigDecimal(0.3));
```

```
0.30000000000000004
0.09999999999999998
false
5.551115123125783E-17
true
0.3
3.3000002
3.3000000000000003
33.333333333333336
33.333332
true
0.1000000000000000055511151231257827021181583404541015625
0.2000000000000000111022302462515654042363166809082031 25
0.3000000000000000444089209850062616169452667236328125
0.29999999999999998889776975374843459576368331909179 6875
```

Bu çıktının satırları tek tek okunduğunda kayan noktalı sayıların temel sorunu ortaya çıkar. Sorun, ikili tabanın bazı ondalık kesirleri tam gösterememesidir. Onluk sistemde 1/3 sonsuz bir kesirdir: 0,3333… Hiçbir sonlu basamak sayısı onu tam yazamaz. İkili sistemde aynı sorun 1/10 için yaşanır. 0,1'in ikili açılımı 0,0001100110011… biçiminde sonsuza kadar sürer. Bellekteki 52 bitlik kesir alanı bu açılımı bir yerde keser ve sayı, 0,1'e çok yakın ama ondan biraz farklı bir değer olarak saklanır.



Şekil 6.10: 0.1 ve 0.2 bellekte tam saklanamaz; en yakın ikili değerler saklanır. Bu küçük hatalar toplandığında sonuç 0.30000000000000004 yazdırılır.

Çıktının son dört satırı, `BigDecimal` sınıfıyla bellekteki değerlerin tam ondalık açılımını gösterir. Saklanan "0.1", gerçekte 0,1000000000000000055511…'dir. Saklanan "0.2" de 0,2'den biraz büyüktür. İki küçük fazlalık toplanınca sonuç, 0,3 için saklanabilecek en yakın double değerine değil, bir sonrakine düşer. `println` bu değeri 0.30000000000000004 diye yazar. 0,3 için saklanan değer ise 0,29999999999999998889…'dur. İki sayı farklı olduğu için `0.1 + 0.2 == 0.3` ifadesi false verir.

Aynı satırlar başka iki dersi de taşır. `1.1f * 3` float ile 3.3000002 verir, double ile 3.3000000000000003 verir. float'taki hata yedinci basamakta, double'daki hata on altıncı basamakta görünür. Buna karşılık `0.5 + 0.25 == 0.75` true'dur. 0,5 ve 0,25 ikinin kuvvetleriyle yazılabildiği için (1/2 ve 1/4) tam saklanır. Hatanın kaynağı kayan noktalı sayıların kendisi değil, onluk kesirlerin ikili tabanda sonsuz açılmasıdır.

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

```
0.30000000000000004
false
true
```

Sunumdaki bu kısa program, karşılaştırmanın doğru yolunu gösterir. İki kayan noktalı sayı `==` ile karşılaştırılmaz. Bunun yerine aralarındaki farkın mutlak değeri küçük bir **eşikle** (*epsilon*) karşılaştırılır: `Math.abs(a - 0.3) < 1e-9`. Buradaki fark yaklaşık $5{,}55 \times 10^{-17}$'dir ve 10^{-9}'dan çok küçüktür; karşılaştırma true verir. Eşiğin büyüklüğü probleme göre seçilir. Milimetre düzeyinde ölçümlerde 10^{-6} yeterlidir; çok küçük fiziksel büyüklüklerde daha küçük bir eşik gerekir.

Küçük hatalar tekrarlanan işlemlerde birikir. 0,1'i on kez toplayan bir döngü 1,0 değil 0.9999999999999999 bulur. Para gibi kuruşu kuruşuna doğru olması gereken hesaplarda bu birikim kabul edilemez. Böyle hesaplarda ya tutarlar kuruş cinsinden tam sayı (long) olarak tutulur ya da `BigDecimal` sınıfı kullanılır. BigDecimal, sayıyı onluk basamaklarıyla saklar:

```
BigDecimal a = new BigDecimal("0.1");
BigDecimal b = new BigDecimal("0.2");
System.out.println(a.add(b));
System.out.println(a.add(b).compareTo(new BigDecimal("0.3")) == 0);
BigDecimal fiyat = new BigDecimal("19.90");
BigDecimal adet = new BigDecimal("3");
System.out.println(fiyat.multiply(adet));
```

```
0.3
true
59.70
```

BigDecimal nesnesi dizgiyle (`"0.1"`) oluşturulur; `new BigDecimal(0.1)` yazılırsa hatalı double değeri olduğu gibi kopyalanır. Toplama `+` yerine `add`, çarpma `multiply` metoduyla yapılır. 19,90 TL'lik üç ürünün tutarı tam olarak 59,70 çıkar.

Sık yapılan hata

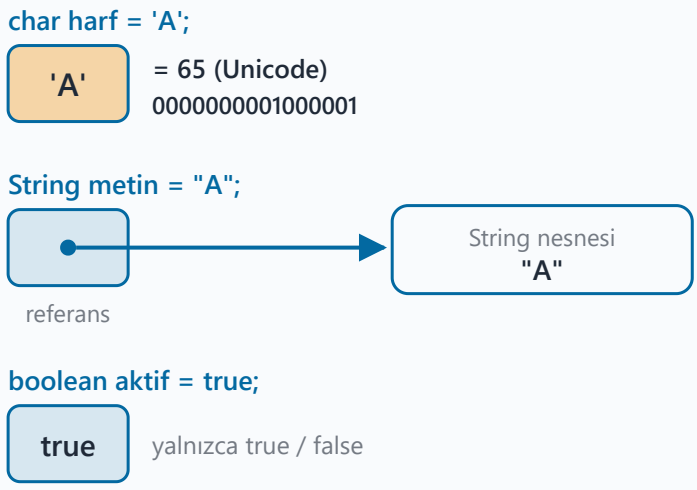
İki double değeri `==` ile karşılaştırıp şaşırmak: `if (toplam == 1.0)` koşulu, toplam 0,1'lerin birikmesiyle oluştuysa hiç sağlanmaz. Kayan noktalı sayılar her zaman bir eşikle karşılaştırılır. Ayrıca kuruş hesabı double ile yapılmaz; tam sayı kuruş ya da BigDecimal kullanılır.

6.2.7 char ve boolean

char tipi tek bir karakter tutar ve sabiti tek tırnakla yazılır: `'A'`, `'7'`, `'ç'`. Bilgisayar harfleri doğrudan saklayamaz; her karaktere bir sayı karşılık gelir ve kutuda o sayı durur. Bu eşleme Unicode standardının işidir. 'A' harfinin kodu 65, 'a' harfininki 97, '0' rakamınınki 48, boşluk karakterininki 32'dir. Java char'ı, Unicode metnini 16 bitlik birimlerle saklayan UTF-16 kodlamasının bir **kod birimidir**. Türkçe harflerin hepsi tek bir char'a sığar: 'ç' 231, 'ğ' 287, 'ı' 305, 'İ' 304, 'ş' 351 kodlarıyla saklanır. (Bazı emojiler gibi çok az kullanılan karakterler iki char gerektirir; bu ayrıntı başlangıç programlarında karşılaşılmaz.)

Tek tırnak ile çift tırnak farklı tipler üretir. `'A'` bir char'dır; kutuda 65 sayısı durur. `"A"` ise tek harflik bir **dizgidir** (*string*), yani bir String nesnesidir; değişken bu nesneye giden referansı tutar. İkisi birbirinin yerine kullanılamaz. `char c = "A";` satırı "String cannot be converted to char" iletisiyle derlenmez.

boolean tipi yalnızca iki değer alır: `true` (doğru) ve `false` (yanlış). Karşılaştırmaların ve koşulların sonucu boolean'dır. C dilinde 0 yanlış, sıfırdan farklı her sayı doğru sayılır; Java'da bu eşleme yoktur. `boolean aktif = 1;` satırı "int cannot be converted to boolean" hatası verir. Bu katılık, koşullarda sık yapılan bir hatayı derleme sırasında yakalar.



Şekil 6.11: Üç farklı kutu. char kutusunda 'A' harfinin kodu olan 65 durur (16 bit). String değişkeni bellekteki bir nesneye referans tutar. boolean kutusu yalnızca true ya da false alır.

6.2.8 char aritmetiği

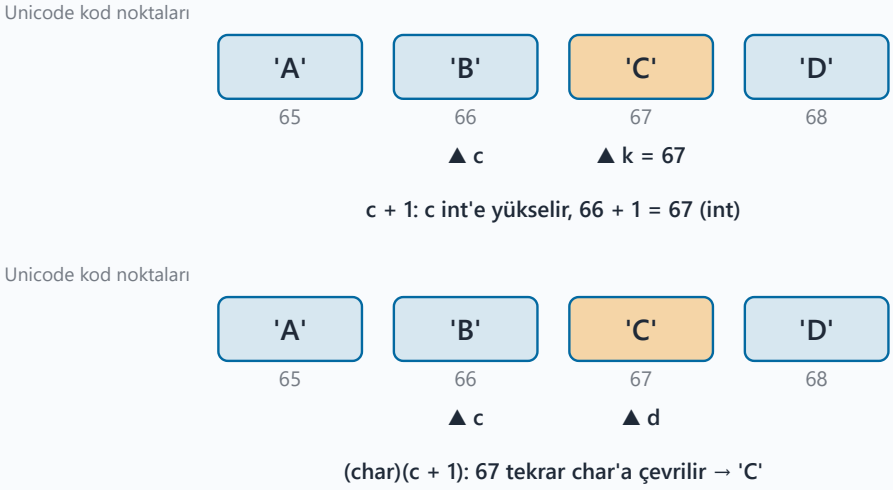
char kutusunda bir sayı durduğu için char değerleriyle aritmetik yapılabilir. Sunumdaki örnek bu ilişkiyi adım adım gösterir:

```
char c = 'A';
c++;
int k = c + 1;
char d = (char)(c + 1);
System.out.println(c + " " + k + " " + d);
```

```
B 67 C
```

Satır	Deyim	c (char)	k (int)	d (char)	Açıklama
1	`char c = 'A';`	'A' (65)	–	–	kutuda 65 durur
2	`c++;`	'B' (66)	–	–	kod 1 artar; tip char kalır
3	`int k = c + 1;`	'B' (66)	67	–	`c + 1` int'tir: 66 + 1 = 67
4	`char d = (char)(c + 1);`	'B' (66)	67	'C' (67)	67 yeniden char'a çevrilir
5	`System.out.println(…);`	'B'	67	'C'	çıktı: B 67 C

Tablo 6.8: char aritmetiği örneğinin izleme tablosu. Parantez içinde kutudaki sayısal kod verilmiştir.



Şekil 6.12: Unicode kod noktaları doğrusunda c, k ve d. `c + 1` ifadesinde c int'e yükselir ve sonuç 67 sayısıdır. `(char)(c + 1)` dönüşümü aynı 67'yi yeniden harf olarak yorumlar: 'C'.

Örneğin can alıcı noktası üçüncü satırdır. `c + 1` ifadesinde bir char ile bir int toplanır. Java, aritmetik işlemlerde char'ı önce int'e çevirir; sonuç int olur. `k` bu yüzden 'C' harfini değil 67 sayısını tutar. Sonucu harf olarak saklamak için açık bir `(char)` dönüşümü gerekir. Dönüşüm yazılmazsa satır der-

lenmez:

```
char c = 'A';
char d = c + 1;
```

```
error: incompatible types: possible lossy conversion from int to char
char d = c + 1;
           ^
1 error
```

İkinci satırdaki `c++` ise sorunsuz derlenir; `++` işleci değişkenin kendi tipini korur. Son satırda yazdırma, tipe göre yapılır: char değer harf olarak (B, C), int değer sayı olarak (67) görünür.

char aritmetiğinin pratik kullanımları vardır:

```
System.out.println((int) 'A' + " " + (int) 'a' + " " + (int) '0' + " " + (int) ' ');
System.out.println('a' - 'A');
System.out.println((char) ('a' - 32));
System.out.println('7' - '0');
System.out.println('A' + 'B');
System.out.println("" + 'A' + 'B');
System.out.println((int) 'ç' + " " + (int) 'ğ' + " " + (int) 'İ' + " " + (int) 'ı' + " " +
(int) 'ş');
char c = 'A';
c += 2;
System.out.println(c);
char h = 66;
System.out.println(h);
```

```
65 97 48 32
32
A
7
131
AB
231 287 304 305 351
C
B
```

Küçük ve büyük harf arasındaki fark İngiliz alfabesinde sabittir: *'a' − 'A' = 97 − 65 = 32*. Küçük bir harften 32 çıkarılıp sonuç char'a çevrilince büyük harf elde edilir. Bir rakam karakterinden '0' çıkarılınca rakamın sayısal değeri bulunur: *'7' − '0' = 55 − 48 = 7*. Bu yöntem, metin olarak okunan bir sayının ba-

samaklarını tek tek işlemek için kullanılır. Türkçe harflerde bu kısa yol işe yaramaz: 'ı' (305) ile 'I' (73) arasındaki fark 32 değildir. Türkçe metinlerde büyük-küçük harf dönüşümü için Java'nın dile duyarlı hazır metotları tercih edilir.

Çıktının beşinci ve altıncı satırları bir tuzağı gösterir. `'A' + 'B'` ifadesinde iki char toplanır; ikisi de int'e çevrilir ve sonuç *65 + 66 = 131* sayısıdır, "AB" değil. Harfleri yan yana yazmak için işlemi bir dizgiyle başlatmak gerekir: `"" + 'A' + 'B'` "AB" verir. Kural basittir: `+` işleminin işlenenlerinden biri dizgiyse işlem toplama değil birleştirmedir. Son iki satır ise char'ın tip korumasını gösterir. `c += 2` 'A'yı 'C'ye ilerletir; bileşik atama sonucu kendiliğinden char'a çevirir. `char h = 66;` satırı da derlenir, çünkü 66 sabiti char aralığındadır ve derleyici bu dönüşümü güvenli bulur; h, 'B' harfini tutar.

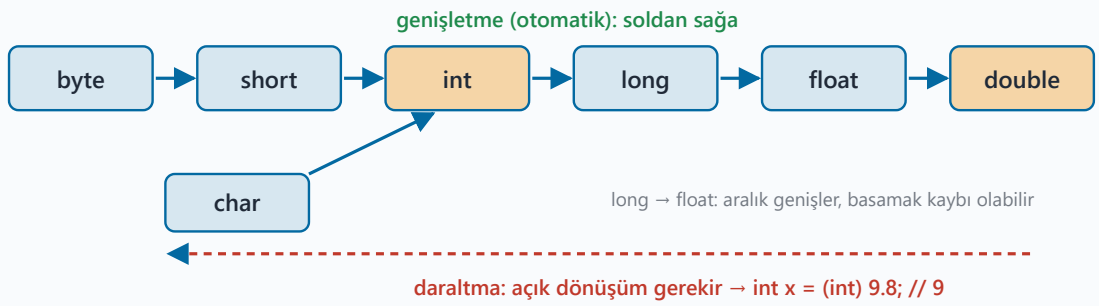
6.3 Tip dönüşümü

Bir programda farklı tipler sık sık karşılaşır. Bir öğrenci sayısı (int) bir ortalamayla (double) çarpılır, bir harfin koduna (char) bir sayı (int) eklenir, bir double sonuç int bir kutuya yazılmak istenir. Her karşılaşmada Java'nın bir değeri bir tipten öbürüne çevirmesi gerekir. Bu işleme **tip dönüşümü** (*type conversion*) denir. Bazı dönüşümleri Java kendiliğinden yapar, bazıları ise programcının açıkça istemesini gerektirir. Ayrımın ölçütü tektir: dönüşümde bilgi kaybı riski var mı?

6.3.1 Atama uyumluluğu: genişletme ve daraltma

Tanım 6.5

Genişletme (*widening conversion*), bir değeri daha geniş aralıklı bir tipe çevirmektir (örneğin int'ten double'a). Java genişletmeyi kendiliğinden, yani **örtük** olarak yapar. **Daraltma** (*narrowing conversion*) ise daha dar aralıklı bir tipe çevirmektir (örneğin double'dan int'e). Daraltma bilgi kaybettirebileceği için **açık dönüşüm** (*cast*) ister: hedef tip parantez içinde değerin önüne yazılır, `(int) 9.8`.



Şekil 6.13: Genişletme zinciri: byte → short → int → long → float → double. char da int'e genişler. Soldan sağa dönüşümler otomatiktir; ters yön daraltmadır ve açık dönüşüm gerektirir.

Genişletme güvenlidir, çünkü dar tipin her değeri geniş tipte de vardır. 7 sayısı int'te de double'da da gösterilebilir; double'a çevrilince 7.0 olur ve hiçbir şey kaybolmaz. Aşağıdaki atamaların hepsi derlenir:

```
int adet = 7;
double genis = adet;
long uzun = adet;
float f = 10;
double d = 'A';
long l = 5;
System.out.println(genis + " " + uzun + " " + f + " " + d + " " + l);
long buyuk = 123456789123L;
float ff = buyuk;
System.out.println(ff);
int i2 = 123456789;
float f2 = i2;
System.out.println(f2 + " " + (int) f2);
```

```
7.0 7 10.0 65.0 5
1.2345679E11
1.2345679E8 123456792
```

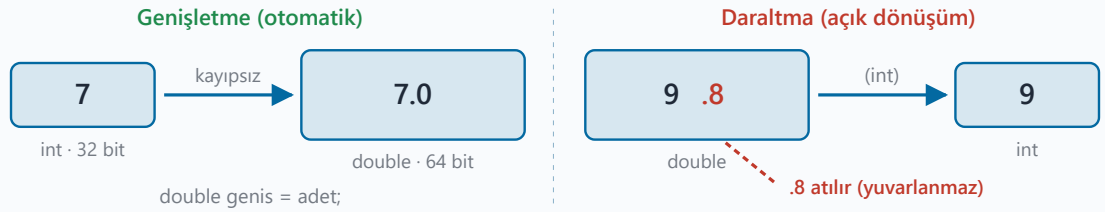
İlk satırdaki değerler beklenen dönüşümleri gösterir: int 7 double'da 7.0, long'da 7 olur; 10 float'ta 10.0; 'A' double'da 65.0 olur. Zincirin bir halkasında ise ince bir kayıp vardır. long'dan float'a ve int'ten float'a geçiş “genişletme” sayılır, çünkü aralık genişler. Ama float yalnızca yaklaşık 7 anlamlı basamak tutar. 123.456.789 sayısı float'a çevrilip geri alınınca 123.456.792 çıkar; son basamaklar yuvarlanmıştır. Java bu dönüşümde uyarı vermez. Büyük tam sayılarla çalışırken float yerine double seçmek bu sorunu büyük ölçüde ortadan kaldırır.

Daraltmada ise bilgi kaybı olağan bir durumdur. double 9.8 bir int kutusuna sığmaz; ondalık kısmın atılması gerekir. Java bu kararı programcıya bırakır ve açık dönüşüm yazılmadıkça atamayı reddeder:

```
double oran = 9.8;
int dar = oran;
```

```
error: incompatible types: possible lossy conversion from double to int
int dar = oran;
          ^
1 error
```

İletideki “possible lossy conversion” ifadesi “olası kayıplı dönüşüm” demektir. Derleyici kaybın gerçekten olup olmayacağını bilmez; yalnızca olabileceğini bildirir. `int dar = (int) oran;` yazıldığında programcı kaybı kabul ettiğini açıkça söylemiş olur ve satır derlenir. Aynı kural sabitler için de geçerlidir: `int x = 5L;` satırı, 5 sayısı int'e rahatça sığdığı hâlde “possible lossy conversion from long to int” verir, çünkü sabitin tipi long'dur.



Şekil 6.14: Genişletme ve daraltma. int 7, double kutuya kayıpsız olarak 7.0 biçiminde geçer. double 9.8 ise `(int)` ile 9'a çevrilir: .8 atılır, yuvarlama yapılmaz.

Dönüşüm, kaynak değişkeni değiştirmez. `(int) oran` ifadesi, `oran` kutusundaki 9.8'i okuyup 9 değerini üretir; `oran` hâlâ 9.8'dir. Dönüşüm yalnızca üretilen sonucu etkiler. Bir dönüşümü değişkenin tipini kalıcı olarak değiştirmek sanmak yaygın bir yanılgıdır; Java'da bir değişkenin tipi bildirildiği anda belirlenir ve bir daha değişmez.

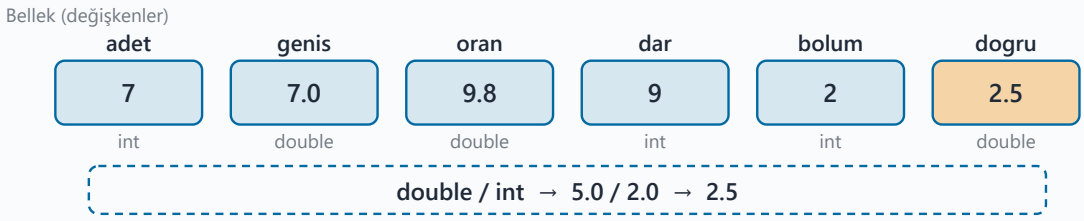
6.3.2 Tip dönüşümü adım adım

Sunumdaki örnek, genişletmeyi, daraltmayı ve iki bölme biçimini tek programda gösterir:

```
int adet = 7;
double genis = adet;       // 7.0
double oran = 9.8;
int dar = (int) oran;      // 9
int bolum = 5 / 2;         // 2
double dogru = 5.0 / 2;    // 2.5
```

Satır	Deyim	Yeni kutu	Değer	Ne oldu?
1	`int adet = 7;`	adet (int)	7	32 bitlik int kutusu
2	`double genis = adet;`	genis (double)	7.0	genişletme: otomatik, kayıpsız
3	`double oran = 9.8;`	oran (double)	9.8	64 bitlik double kutusu
4	`int dar = (int) oran;`	dar (int)	9	daraltma: kesir atıldı; oran hâlâ 9.8
5	`int bolum = 5 / 2;`	bolum (int)	2	int / int: tam sayı bölmesi
6	`double dogru = 5.0 / 2;`	dogru (double)	2.5	double / int: 2, 2.0'a genişler

Tablo 6.9: Tip dönüşümü örneğinin izleme tablosu. Her satırda yalnızca yeni ayrılan kutu verilmiştir; öncekiler değişmez.



Şekil 6.15: Programın sonunda bellek. Altı kutunun beşi önceki satırlarda doldu; son satırda *5.0 / 2.0 → 2.5* hesaplanır ve `dogru` kutusuna yazılır.

Dördüncü satırdaki dönüşüm yuvarlama değil **kesmedir** (*truncation*): 9.8 en yakın tam sayı olan 10'a değil, 9'a çevrilir. Ondalık kısım ne olursa olsun atılır; 9.99 da 9 olur. Beşinci ve altıncı satırlar ise bir sonraki kesimin konusunu önceden gösterir: aynı 5 ve 2 sayıları, yazılış biçimlerine göre 2 ya da 2.5 sonucunu verir.

6.3.3 Dönüşümlerde taşma ve kesme

Açık dönüşüm, derleyiciye "kaybı kabul ediyorum" demektir. Kaybın ne kadar olacağı ise dönüşümün türüne bağlıdır. Sunumdaki tablo, sık karşılaşılan durumları bir arada verir. Değerler JDK 21'de çalıştırılarak doğrulanmıştır.

İfade	Sonuç	Neden
`(byte) 200`	-56	yalnız alt 8 bit kalır: 200 − 256
`(int) 3000000000L`	-1294967296	long → int: üst bitler atılır
`(int) 3.9`	3	ondalık kısım kesilir, yuvarlanmaz
`(int) -2.7`	-2	sıfıra doğru kesilir
`(int) 3.9e10`	2147483647	double → int: int sınırında sıkışır
`Math.round(3.9)`	4	en yakın tam sayı (long döner)
`Math.round(-2.7)`	-3	en yakın tam sayı
`Math.round(3.9e10)`	39000000000	long olduğu için sığar

Tablo 6.10: Daraltma dönüşümlerinin ve Math.round'un sonuçları (sunumdaki tablo).

Tablodaki satırlar üç ayrı davranışı gösterir.

Tam sayıdan daha dar tam sayıya. Dönüşüm, kaynak sayının yalnızca hedef tipe sığan alt bitlerini tutar ve üst bitleri atar. 200'ün ikili gösterimi 1100 1000'dir. Bu sekiz bit byte'a aynen kopyalanır. İşaret biti 1 olduğu için değer *200 − 256 = −56* olarak okunur. 3.000.000.000 için hesap int boyutunda yapılır: *3.000.000.000 − 2^{32} = 3.000.000.000 − 4.294.967.296 = −1.294.967.296*. Taşmadaki sarma kuralıyla aynı kuraldır. Birkaç örnek daha: `(byte) 130` −126, `(byte) 256` 0, `(byte) -129` 127, `(short) 40000` −25536 verir.

Kayan noktalıdan tam sayıya. Ondalık kısım atılır ve sayı **sıfıra doğru** kesilir: pozitif sayılar aşağı, negatif sayılar yukarı gider. 3.9 → 3 ve −2.7 → −2 bu yüzdendir. −2.7'nin −3'e değil −2'ye gitmesi şaşırtıcı gelebilir; −2, −3'ten daha büyüktür ve sıfıra daha yakındır. Sayı hedef tipin aralığını aşıyorsa bitler kesilmez; sonuç aralığın ucunda sıkışır. $3{,}9 \times 10^{10}$ int'e sığmaz ve Integer.MAX_VALUE olan 2147483647 çıkar. Aynı biçimde `(int) -3.9e10` en küçük int'i, `(int) Double.NaN` ise 0 verir.

Yuvarlama. Kesme yerine en yakın tam sayı isteniyorsa `Math.round` kullanılır. 3.9 → 4 ve −2.7 → −3 olur. Math.round, double alınca long döndürür; bu yüzden $3{,}9 \times 10^{10}$ sorunsuz sığar. Sonuç int bir kutuya yazılacaksa ikinci bir dönüşüm gerekir: `int ri = (int) Math.round(x);`. Dönüşüm yazılmazsa `int r = Math.round(2.7);` satırı "possible lossy conversion from long to int" hatası verir.

> **Sık yapılan hata**
>
> `(int)` dönüşümünün sayıyı yuvarladığını sanmak. `(int) 9.8` 10 değil 9'dur; `(int) -2.7` −3 değil −2'dir. Yuvarlama için `Math.round`, int sonuç için `(int) Math.round(x)` yazılır. Daraltma dönüşümü hata vermez; veri sessizce kaybolur.

6.3.4 Tam sayı bölmesi: 5 / 2 ile 5.0 / 2

Java'da bölme işleci `/` iki farklı iş yapar ve hangisini yapacağına işlenenlerin tipine bakarak karar verir. İki işlenen de tam sayıysa **tam sayı bölmesi** (*integer division*) yapılır: sonuç bölümün tam kısmıdır, kalan atılır. İşlenenlerden en az biri kayan noktalıysa öbürü de kayan noktalıya yükseltilir ve olağan bölme yapılır.

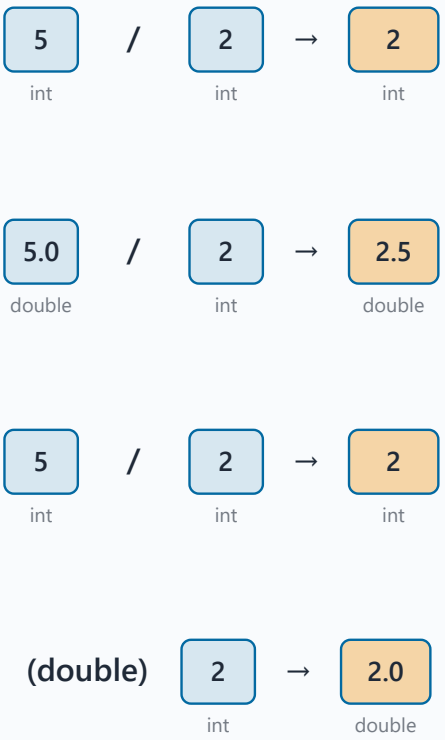
```
System.out.println(5 / 2);
System.out.println(5.0 / 2);
System.out.println(5 / 2.0);
System.out.println((double) 5 / 2);
System.out.println((double) (5 / 2));
System.out.println(5 % 2);
```

```
2
2.5
2.5
2.5
2.0
1
```

İfade	İşlenen tipleri	Yapılan işlem	Çıktı
`5 / 2`	int, int	tam sayı bölmesi	2
`5.0 / 2`	double, int	2 → 2.0; 5.0 / 2.0	2.5
`5 / 2.0`	int, double	5 → 5.0; 5.0 / 2.0	2.5

İfade	İşlenen tipleri	Yapılan işlem	Çıktı
`(double) 5 / 2`	double, int	önce dönüşüm: 5.0; sonra 5.0 / 2.0	2.5
`(double) (5 / 2)`	int, int	önce 5 / 2 = 2; sonra 2 → 2.0	2.0
`5 % 2`	int, int	kalan: 5 = 2 · 2 + 1	1

Tablo 6.11: Altı satırın işlenen tipleri ve sonuçları.



Şekil 6.16: Bölme ifadelerinin işlenen tipleri. İki int: sonuç 2 (int). Bir double: int işlenen double'a genişler ve sonuç 2.5 olur. `(double) (5 / 2)` ifadesinde önce parantez içi int olarak hesaplanır (2), dönüşüm ancak sonra gelir (2.0).

Dördüncü ve beşinci satırlar arasındaki fark yalnızca bir çift paranteze dayanır ama sonuç değişir. Tip dönüşümü bölmeden önce gelen bir tekli işleçtir. `(double) 5 / 2` ifadesinde önce 5 double'a çevrilir, sonra bölme kayan noktalı yapılır. `(double) (5 / 2)` ifadesinde ise parantez önce hesaplanır; bölme iki int arasında yapılır ve 0,5 kaybolur. Sonradan yapılan dönüşüm 2'yi 2.0 yapar ama kaybolan kesri geri getirmez.

Örnek 6.3

70, 85 ve 90 notlarının ortalaması dört farklı ifadeyle hesaplanıyor. Hangisi doğru sonucu verir?

```
int n1 = 70, n2 = 85, n3 = 90;
System.out.println((n1 + n2 + n3) / 3);
System.out.println((n1 + n2 + n3) / 3.0);
System.out.println((double) (n1 + n2 + n3) / 3);
System.out.printf("%.2f%n", (n1 + n2 + n3) / 3.0);
```

```
81
81.66666666666667
81.66666666666667
81.67
```

Toplam *70 + 85 + 90 = 245*'tir. İlk satırda 245 ve 3 int olduğu için tam sayı bölmesi yapılır: *245 / 3 = 81*, kalan 2 atılır. İkinci satırda 3.0 double'dır; 245 de double'a yükselir ve *245 / 3 = 81,666...* bulunur. Üçüncü satırda dönüşüm bölmeden önce yapılır ve sonuç yine doğrudur. Son satır sonucu iki ondalık basamakla yazar: 81.67. Doğru ortalama için bölmenin en az bir işleneni kayan noktalı olmalıdır.

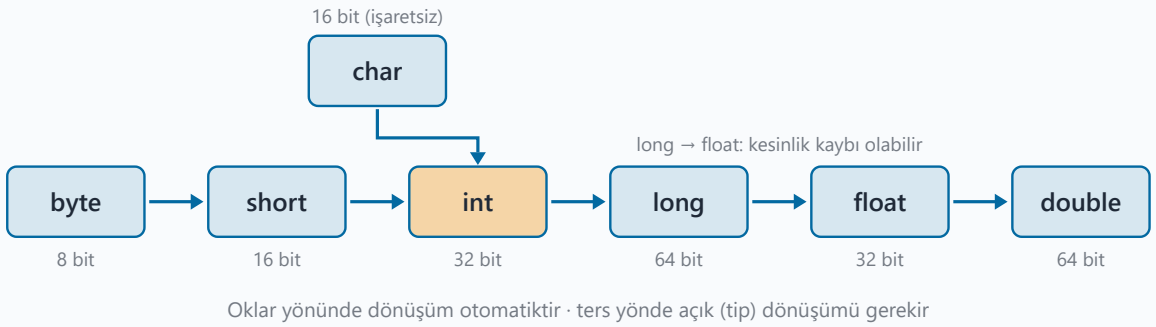
Tam sayı bölmesi her zaman bir hata değildir; çoğu zaman tam da istenen işlemdir. 17 elma 5'erli paketlere konursa *17 / 5 = 3* tam paket çıkar. 3725 saniyede *3725 / 3600 = 1* tam saat vardır. Bu tür hesaplarda tam sayı bölmesi, kalanı veren `%` işleciyle birlikte kullanılır. Hata, ortalama ya da oran gibi kesirli bir sonuç beklenirken iki tam sayının bölünmesidir.

6.3.5 Tip yükseltme

Bölmede görülen kural, bütün ikili aritmetik işleçlerde geçerlidir.

Tanım 6.6

Tip yükseltme (*type promotion*), karma tipli bir ifadede dar tipteki işlenenin, işlem yapılmadan önce geniş tipe otomatik çevrilmesidir. Sonuç geniş tiptedir: int + double → double, int * long → long. Ayrıca byte, short ve char işlenenler her aritmetik işlemde önce int'e yükseltilir.



Şekil 6.17: Tip yükseltme sırası. Oklar yönündeki dönüşümler otomatiktir; ters yönde açık dönüşüm gerekir. char 16 bitlik işaretsiz bir tiptir ve int'e yükselir. long → float geçişinde kesinlik kaybı olabilir.

Kuralın ikinci yarısı şaşırtıcıdır: iki byte'ın toplamı bile int'tir. Java bu küçük tiplerle doğrudan aritmetik yapmaz; işlemciler 32 bitlik işlemlerde daha verimli çalışır ve ara sonucun taşma riski azalır. Bunun sonucu olarak aşağıdaki satırlar derlenmez:

```
short a = 1, b = 2;
short c = a + b;
```

```
error: incompatible types: possible lossy conversion from int to short
short c = a + b;
          ^
1 error
```

`a + b` int'tir ve int'ten short'a daraltma açık dönüşüm ister: `short c = (short) (a + b);`. Aynı nedenle `byte b = 10; b = b + 5;` de derlenmez. Bu davranış, char aritmetiğinde görülen `c + 1` ifadesinin neden int olduğunu da açıklar.

Bir ifadede birden çok işlem varsa yükseltme her işlem için ayrı ayrı yapılır. `1 / 2 * 10.0` ifadesinde önce `1 / 2` iki int arasında hesaplanır ve 0 olur; ancak ondan sonra 0, 10.0 ile çarpılırken double'a yükselir. Sonuç 5.0 değil 0.0'dır. Bir double işlenenin ifadede bir yerde bulunması, ifadenin her yerini kayan noktalı yapmaz; yükseltme yalnızca o işlenenin katıldığı işlemde gerçekleşir.

6.4 İşleçler

Tanım 6.7

İşleç (*operator*), bir ya da daha fazla değer üzerinde işlem yapan simgedir: `+`, `*`, `<`, `&&`. İşlecin üzerinde işlem yaptığı değerlere **işlenen** (*operand*) denir. İşleçlerin, değişkenlerin ve sabitlerin bir değer üreten birleşimine **ifade** (*expression*) denir: `a + b * 2`.

İşleçler, aldıkları işlenen sayısına göre adlandırılır. `-x`, `!aktif`, `i++` gibi tek işlenenli işleçler **tekli**, `a + b` gibi iki işlenenli olanlar **ikili** işleçlerdir. Her ifadenin bir değeri ve bir tipi vardır. `7 / 2` ifadesinin değeri 3, tipi int'tir; `7 > 2` ifadesinin değeri true, tipi boolean'dır.

6.4.1 Aritmetik işleçler ve kalan

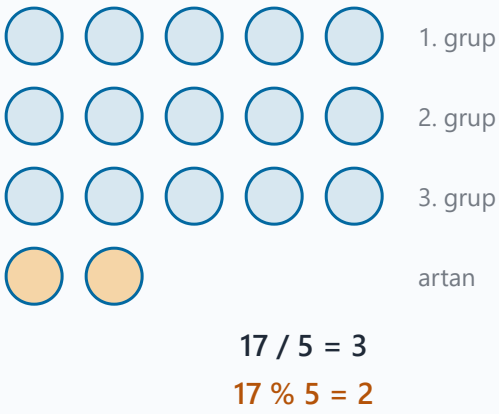
İşleç	Anlam	Örnek
`+ -`	Toplama / çıkarma	`a + b`
`* /`	Çarpma / bölme	`a * b`
`%`	Kalan	`sayi % 2`
`++ --`	Bir artır / azalt	`i++`
`+= -=`	Bileşik atama	`toplam += deger`

Tablo 6.12: Aritmetik işleçler (sunumdaki tablo).

Toplama, çıkarma ve çarpma matematikteki gibidir; tek fark, sonucun tipin aralığına sığmak zorunda olmasıdır. Bölme işlecinin iki yüzü tam sayı bölmesi kesiminde görüldü. Bu kesimin asıl konusu `%` işlecidir.

Tanım 6.8

Kalan (*remainder*) işleci `%`, tam sayı bölmesinden artan değeri verir. a ve b tam sayıları için *a = (a / b) · b + (a % b)* eşitliği her zaman sağlanır. Bölmede bölünen sayıya **bölünen**, bölme sonucunun tam kısmına **bölüm** denir.



Şekil 6.18: 17 nesne beşerli gruplanır. 3 tam grup oluşur (*17 / 5 = 3*) ve 2 nesne artar (*17 % 5 = 2*).

Şekil 6.18, 17 nesnenin beşerli gruplanmasını gösterir: `/` kaç tam grup oluştuğunu, `%` kaç nesne arttığını söyler. Eşitlik kontrol edilebilir: *3 · 5 + 2 = 17*. Kalan işleci, ilk bakışta sanıldığından çok daha sık kullanılır:

- **Çift-tek denetimi.** `sayi % 2 == 0` ise sayı çifttir. Genel olarak `a % b == 0`, a'nın b'ye tam bölündüğünü gösterir.
- **Basamak ayırma.** `sayi % 10` son basamağı verir; `sayi / 10` son basamağı atar. 1234 için *1234 % 10 = 4* ve *1234 / 10 = 123*'tür.
- **Birim dönüşümü.** Saniyeyi dakika ve saniyeye, kuruşu lira ve kuruşa ayırmak: 1575 kuruş, *1575 / 100 = 15* lira ve *1575 % 100 = 75* kuruştur.
- **Döngüsel sayma.** Haftanın günü, saat kadranı gibi başa dönen değerler: bugün çarşambaysa (2) 10 gün sonrası *(2 + 10) % 7 = 5*, yani cumartesidir (pazartesi 0 kabul edilirse).

Negatif sayılarda kalanın işareti bir kurala bağlıdır: Java'da kalan her zaman **bölünenin** işaretini alır. Bu kural, tam sayı bölmesinin sıfıra doğru kesmesinden doğar. Aşağıdaki programın ilk dört satırı negatif işlenenleri, sonraki satırları ise bu kesimde ve sıfıra bölme kesiminde geçen öteki durumları sınar:

```
System.out.println(7 / 2 + " " + 7 % 2);
System.out.println(-7 / 2 + " " + -7 % 2);
System.out.println(7 / -2 + " " + 7 % -2);
System.out.println(-7 / 3 + " " + -7 % 3 + " " + Math.floorMod(-7, 3) + " " +
Math.floorDiv(-7, 3));
System.out.println(17 / 5 + " " + 17 % 5);
System.out.println(5.5 % 2);
System.out.println(1 / 2 * 10 + " " + 10 * 1 / 2);
System.out.println(1.0 / 0 + " " + -1.0 / 0 + " " + 0.0 / 0);
double nan = 0.0 / 0;
System.out.println((nan == nan) + " " + (nan != nan) + " " + Double.isNaN(nan));
System.out.println(Math.sqrt(-1) + " " + (1.0 / 0 - 1.0 / 0) + " " + (1.0 / 0 + 5));
System.out.println(1 / 0.0 > 1e308);
```

```
3 1
-3 -1
-3 1
-2 -1 2 -3
3 2
1.5
0 5
Infinity -Infinity NaN
false true true
NaN NaN Infinity
true
```

Çıktının ilk dört satırı bu kuralı gösterir. *−7 / 2 = −3*'tür (−3,5 sıfıra doğru kesilir), kalan ise *−7 − (−3 · 2) = −1*'dir. *7 / −2 = −3* ve *7 % −2 = 1*'dir; bölünen 7 pozitif olduğu için kalan da pozitiftir. *−7 % 3 = −1*'dir; matematikte alışılan "0 ile 2 arasındaki kalan" 2'dir. Bu değer gerekiyorsa `Math.floorMod(-7, 3)` kullanılır. Bu yüzden bir sayının tek olup olmadığı `sayi % 2 == 1` ile değil `sayi % 2 != 0` ile denetlenir; −7 için ilki yanlış sonuç verir.

Beşinci satır şekildeki 17 nesne örneğidir. % işleci kayan noktalı sayılarla da çalışır: *5,5 = 2 · 2 + 1,5* olduğu için `5.5 % 2` 1.5 verir. Çıktının yedinci satırı ise işlem sırasının tam sayı bölmesindeki etkisini gösterir: `1 / 2 * 10` önce *1 / 2 = 0* hesaplar ve 0 bulur; `10 * 1 / 2` önce *10 · 1 = 10* hesaplar ve 5 bulur. Matematikte iki ifade eşittir; Java'da tam sayı bölmesi yüzünden değildir.

6.4.2 Sıfıra bölme ve özel değerler

Matematikte sıfıra bölme tanımsızdır. Java bu durumu tam sayılarda ve kayan noktalı sayılarda farklı ele alır.

İfade	Sonuç	Açıklama
`1 / 0`	ArithmeticException: / by zero	tam sayı bölmesinde sıfıra bölme çalışma zamanı hatasıdır
`1.0 / 0`	Infinity	ondalıkta hata yok: sonsuz

İfade	Sonuç	Açıklama
`-1.0 / 0`	-Infinity	eksi sonsuz
`0.0 / 0`	NaN	Not a Number: tanımsız sonuç
`NaN == NaN`	false	NaN kendine bile eşit değildir
`NaN != NaN`	true	NaN denetimi için `Double.isNaN(x)`
`-7 % 3`	-1	kalanın işareti bölünenin işaretidir
`5.5 % 2`	1.5	% kayan noktalı sayılarda da çalışır

Tablo 6.13: Sıfıra bölme ve özel değerler (sunumdaki tablo).

Tam sayılarda sıfıra bölme programı durdurur. Derleme başarılıdır, çünkü derleyici çoğu zaman bölenin sıfır olacağını önceden bilemez. Hata program çalışırken ortaya çıkar:

```
int a = 1;
int b = 0;
System.out.println("önce");
System.out.println(a / b);
System.out.println("sonra");
```

```
önce
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

"önce" yazılır, bölme satırında JVM bir `ArithmeticException` fırlatır ve program orada biter; "sonra" hiç yazılmaz. Aynı hata kalan işleminde de görülür: `5 % 0` da "/ by zero" iletisiyle durur.

Kayan noktalı sayılarda ise sıfıra bölme hata vermez. Kayan nokta standardı (IEEE 754) bu durumlar için özel değerler tanımlar. Pozitif bir sayının sıfıra bölümü `Infinity` (sonsuz), negatifinki `-Infinity` olur. *0,0 / 0* gibi anlamı olmayan işlemler `NaN` (*Not a Number*, "sayı değil") üretir. `Math.sqrt(-1)` ve ∞ − ∞ da NaN verir. Bu değerler programı durdurmaz, sonraki işlemlere taşınır: ∞ + *5* = ∞, NaN'lı her aritmetik işlem yine NaN'dır. Bir hesabın en sonunda beklenmedik biçimde NaN çıkması, genellikle çok önceki bir satırda sıfıra bölündüğünü gösterir.

NaN'ın en tuhaf özelliği kendine eşit olmamasıdır: `x == x`, x NaN ise false verir. Bu yüzden NaN denetimi `x == Double.NaN` ile yapılamaz; her zaman false döner. Doğru yol `Double.isNaN(x)` metodudur.

Sık yapılan hata

Kayan noktalı bölmede bölenin sıfır olabileceğini unutmak. Tam sayı bölmesi bu hatayı istisnayla hemen gösterir; kayan noktalı bölme ise Infinity ya da NaN üretip sessizce devam eder. Bölen sıfır olabiliyorsa bölmeden önce denetlenir.

6.4.3 Artırma ve azaltma

Bir değişkeni 1 artırmak ya da 1 azaltmak o kadar sık yapılır ki Java bunun için ayrı işleçler sunar: `++` ve `--`. Tek başına bir deyim olarak yazıldıklarında `i++;`, `i = i + 1;` ile aynıdır. İnce bir fark, işleç bir ifadenin içinde kullanıldığında ortaya çıkar.

Tanım 6.9

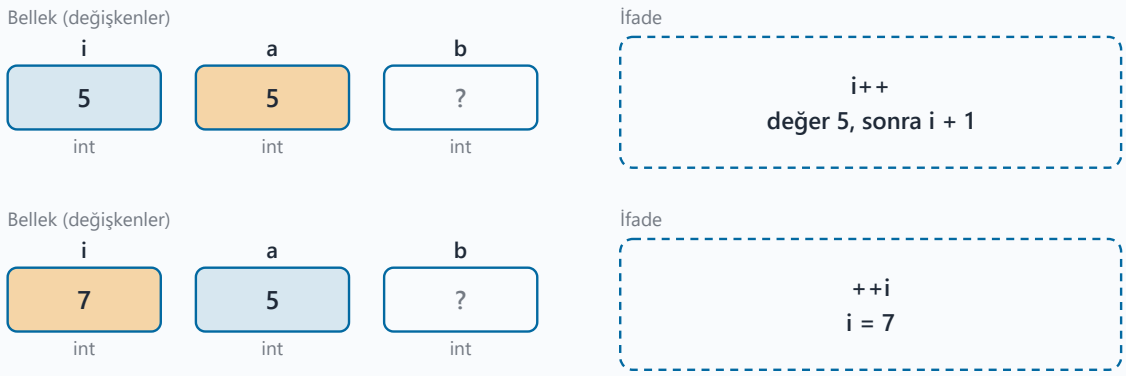
Sonek (*postfix*) biçiminde (`i++`) ifadenin değeri değişkenin **eski** değeridir; değişken bu değer kullanıldıktan sonra artırılır. **Önek** (*prefix*) biçiminde (`++i`) önce değişken artırılır; ifadenin değeri **yeni** değerdir.

```
int i = 5;
int a = i++;   // önce kullan, sonra artır
int b = ++i;   // önce artır, sonra kullan
i--;
System.out.println(i + " " + a + " " + b);
```

```
6 5 7
```

Satır	Deyim	İfadenin değeri	i	a	b
1	`int i = 5;`	–	5	–	–
2	`int a = i++;`	`i++` → 5 (eski değer)	6	5	–
3	`int b = ++i;`	`++i` → 7 (yeni değer)	7	5	7
4	`i--;`	tek başına deyim	6	5	7
5	`System.out.println(…);`	çıktı: 6 5 7	6	5	7

Tablo 6.14: Önek ve sonek artırma örneğinin izleme tablosu.



Şekil 6.19: İkinci ve üçüncü satır. Sonek `i++` : ifadenin değeri 5'tir, a = 5 olur, sonra i 6'ya çıkar. Önek `++i` : önce i = 7 olur, ifadenin değeri yeni değer 7'dir ve b = 7 olur.

İkinci satırda `i++` önce eski değer olan 5'i üretir; bu değer a'ya yazılır. Artırma da aynı satırda gerçekleşir ve i 6 olur. Üçüncü satırda `++i` önce i'yi 7 yapar, sonra bu yeni değeri üretir; b = 7 olur. Dördüncü satırda `i--` tek başına bir deyimdir. Değeri hiçbir yerde kullanılmadığı için önek ve sonek arasındaki fark burada önemsizdir: `--i;` yazılsaydı da i 6 olurdu.

Önek ve sonek farkı tuzaklara açıktır. `j = j++;` satırı j'yi artırmaz: sağ taraf eski değer olan 5'i üretir, j bir an 6 olur, sonra atama 5'i yeniden yazar. Bir ifadede aynı değişkenin birden çok kez artırılması da okunması zor kod üretir. `k++ + k++` ifadesi k = 1 için *1 + 2 = 3* verir ve k'yi 3 yapar. Sonuçlar Java'da kesin kurallarla belirlidir ama okuyan herkesi düşündürür. ++ ve --'yi yalnızca tek başına bir deyim olarak kullanmak kodu açık tutar.

6.4.4 Bileşik atama ve örtük daraltma

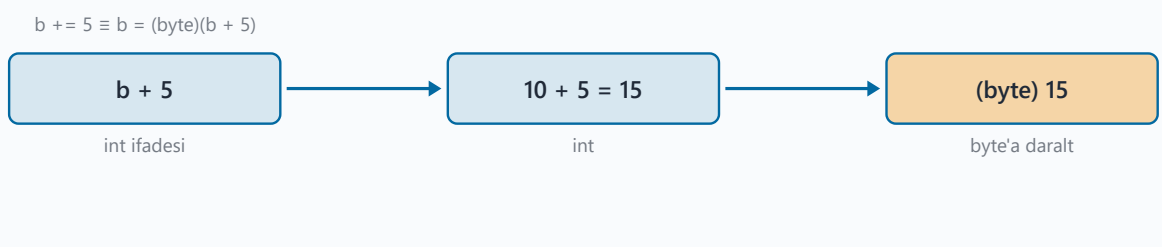
Bileşik atama işleçleri, `+=`, `-=`, `*=`, `/=` ve `%=`, yalnızca kısa bir yazım değildir. Sonucu soldaki değişkenin tipine kendiliğinden daraltırlar. Java dil tanımı, `x op= y` deyimini `x = (T) (x op y)` diye tanımlar; buradaki T, x'in tipidir.

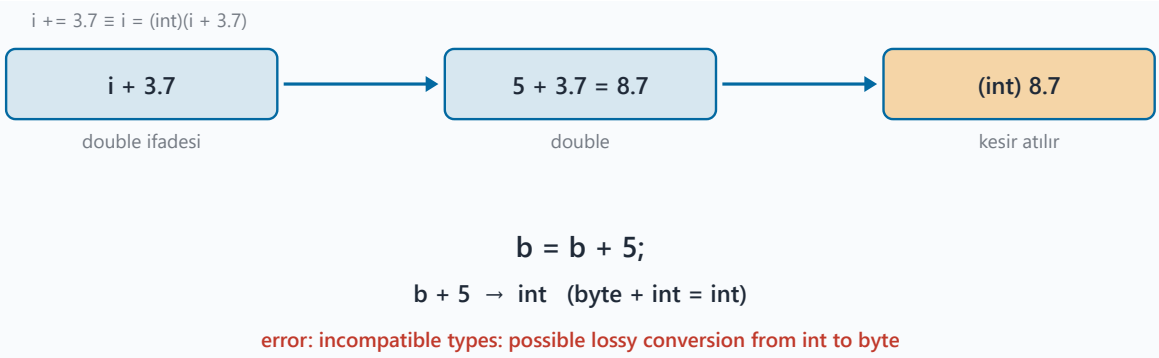
```
byte b = 10;
b += 5;
int i = 5;
i += 3.7;
// b = b + 5;   // derlenmez: int → byte
System.out.println(b + " " + i);
```

```
15 8
```

Satır	Deyim	Derleyicinin gördüğü	b	i
1	`byte b = 10;`		10	–
2	`b += 5;`	`b = (byte) (b + 5)` : 10 + 5 = 15 (int) → 15 (byte)	15	–
3	`int i = 5;`		15	5
4	`i += 3.7;`	`i = (int) (i + 3.7)` : 5 + 3.7 = 8.7 (double) → 8	15	8
5	`// b = b + 5;`	yorum: derlenmeyeceği için devre dışı	15	8
6	`System.out.println(b + " " + i);`	çıktı: 15 8	15	8

Tablo 6.15: Bileşik atama örneğinin izleme tablosu.





Şekil 6.20: Bileşik atamanın örtük dönüşümü. `b += 5`, int toplamı byte'a daraltır. `i += 3.7`, double toplamı int'e daraltır ve kesri atar. Aynı işi `b = b + 5;` biçiminde yazmak derleme hatası verir.

Beşinci satırdaki yorum önemlidir. `b = b + 5;` satırında `b + 5` int'tir ve int'ten byte'a atama açık dönüşüm ister; satır derlenmez. Aynı işi yapan `b += 5;` ise derlenir, çünkü dönüşüm bileşik atamanın tanımında gizlidir. Bu kolaylık bir tehlike de taşır: dönüşüm sessizdir. `i += 3.7` sonucu 8.7'yi uyarı vermeden 8'e indirir. byte sınırı aşıldığında taşma da sessizce olur:

```
int i = 5;
i *= 3 + 7;
System.out.println(i);
int x = 17;
x %= 5;
System.out.println(x);
int y = 20;
y /= 3;
System.out.println(y);
double z = 20;
z /= 3;
System.out.println(z);
byte b = 120;
b += 10;
System.out.println(b);
short s = 1;
s++;
System.out.println(s);
char c = 'a';
c -= 32;
System.out.println(c);
```

```
50
2
6
6.666666666666667
-126
2
A
```

Çıktının satırları sırasıyla okunur. `i *= 3 + 7` deyiminde sağ taraf bütünüyle önce hesaplanır: *5 · (3 + 7) = 50*. `x %= 5`, 17'yi kalan olan 2'ye çevirir. `y /= 3` int'te tam sayı bölmesidir: *20 / 3 = 6*; double z ile aynı işlem 6.666666666666667 verir. `b += 10`, 120 + 10 = 130'u byte'a sığdırmak için 256 çıkarır: −126. Son iki satır, `++` ve `-=` işleçlerinin de değişkenin tipini koruduğunu gösterir: short s 2 olur, char c 'a'dan 32 çıkarılarak 'A' olur.

Sık yapılan hata

`x += y` ile `x = x + y` deyimlerini her durumda aynı sanmak. İki fark vardır. Birincisi, bileşik atama örtük daraltma yapar; `x = x + y` ise tipler uymadığında derlenmez. İkincisi, bileşik atamada sağ taraf bir bütündür: `x *= a + b`, `x = x * (a + b)` demektir, `x = x * a + b` değil.

6.4.5 Karşılaştırma ve mantık işleçleri

Programlar karar verir: not 60'tan büyükse geçti yaz, yaş 18'den küçükse uyar. Bu kararların temelinde, sonucu true ya da false olan ifadeler vardır.

İşleç	Anlam	Örnek
`== !=`	Eşit / farklı	`a == b`
`< <= > >=`	Sıra karşılaştırması	`not >= 60`
`&&`	İki koşul da doğru	`yas >= 18 && belgeVar`
`||`	En az biri doğru	`haftaSonu || tatil`
`!`	Mantıksal değil	`!aktif`

Tablo 6.16: Karşılaştırma ve mantık işleçleri (sunumdaki tablo).

Karşılaştırma işleçleri (*comparison operator*) iki değeri karşılaştırır ve her zaman boolean bir sonuç üretir. Eşitlik iki eşittir işaretiyle yazılır: `==`. Tek eşittir (`=`) atamadır; ikisini karıştırmak en sık yapılan hatalardandır. `!=` "eşit değil" demektir. Matematikteki ≤ ve ≥ işaretleri `<=` ve `>=` diye iki karakterle yazılır.

Mantık işleçleri (*logical operator*) boolean değerleri birleştirir. `&&` (ve) iki koşul da doğruysa doğrudur. `||` (veya) en az biri doğruysa doğrudur. `!` (değil) tek işlenenlidir ve değeri tersine çevirir. Dört olası durumun tamamı bir doğruluk tablosunda gösterilir:

p	q	p && q	p || q	!p
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Tablo 6.17: && ve || için doğruluk tablosu.

```
int yas = 20;
boolean belgeVar = true;
System.out.println(yas >= 18 && belgeVar);
System.out.println(!belgeVar);
System.out.println(yas > 65 || yas < 12);
System.out.println(5 > 3 == true);
int not = 75;
System.out.println(not >= 60 && not <= 100);
System.out.println(true ^ true);
```

```
true
false
false
true
true
false
```

Yaş 20 ve belge varken `yas >= 18 && belgeVar` doğrudur. `yas > 65 || yas < 12` ifadesi ise iki koşul da yanlış olduğu için yanlıştır. Bir değerin bir aralıkta olup olmadığı iki karşılaştırmanın `&&` ile birleşimiyle yazılır: `not >= 60 && not <= 100`. (`not` Java'da anahtar sözcük değildir; değişken adı olarak kullanılabilir.) Son satırdaki `^` "ya o ya bu" (özel veya) işlecidir; iki işlenen farklıysa doğrudur.

Matematikte alışılan *0 < x < 10* yazılışı Java'da çalışmaz:

```
error: bad operand types for binary operator '<'
System.out.println(0 < x < 10);
                   ^
  first type:  boolean
  second type: int
1 error
```

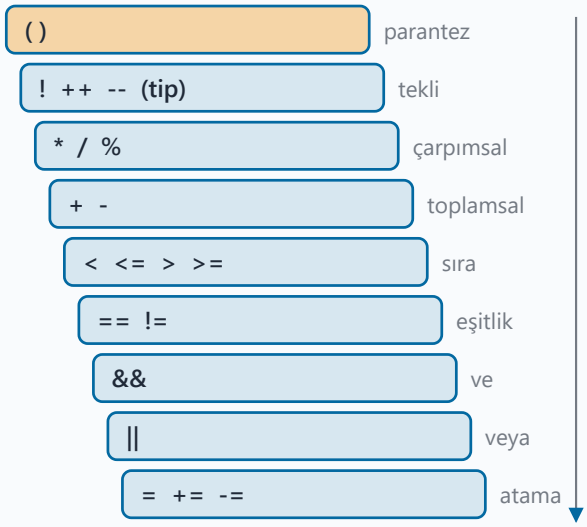
`0 < x` önce hesaplanır ve boolean bir değer üretir. Ardından bu boolean değer 10 ile karşılaştırılmak istenir; boolean ile int arasında `<` tanımlı değildir. Doğru yazılış `0 < x && x < 10`'dur.

6.4.6 İşleç önceliği ve ifade ağacı

`2 + 3 * 4` ifadesinin değeri 20 mi, 14 mü? Matematikte çarpmanın toplamadan önce yapıldığı bilinir ve cevap 14'tür. Java da aynı kuralı izler ve bunu bütün işleçlere genişletir.

Tanım 6.10

İşleç önceliği (*operator precedence*), bir ifadede farklı işleçlerin hangi sırayla uygulanacağını belirleyen kurallar dizisidir. Aynı öncelikteki işleçler soldan sağa uygulanır; atama işleçleri ise sağdan sola uygulanır.

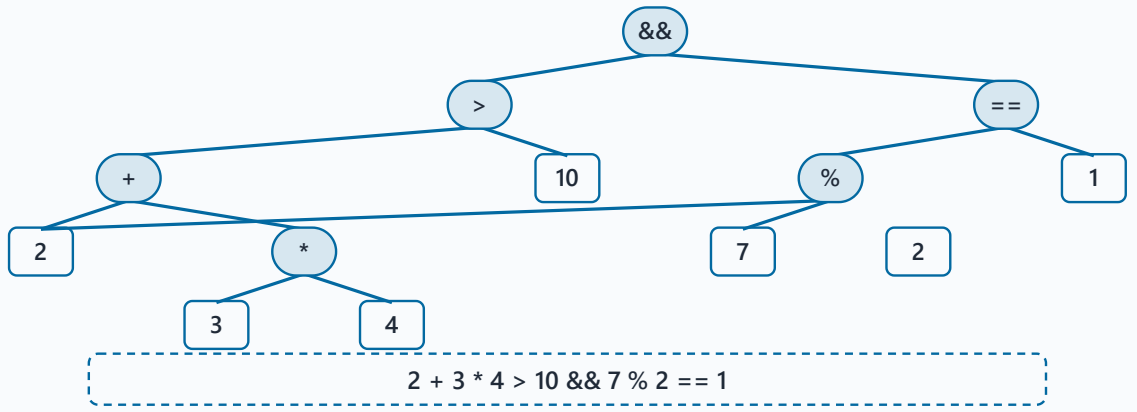


Şekil 6.21: İşleç önceliği merdiveni. Üstteki basamak önce uygulanır: parantez, tekli işleçler (`!`, `++`, `--`, tip dönüşümü), çarpımsal, toplamsal, sıra karşılaştırması, eşitlik, `&&`, `||` ve en son atama.

Merdiven, sunumdaki dört kurala karşılık gelir. Parantez içi her zaman önce değerlendirilir; tekli `!` ve tip dönüşümü de erken uygulanır. Çarpma, bölme ve kalan; toplama ve çıkarmadan önce gelir. Karşılaştırmalar aritmetikten sonra, `&&` ve `||` ise karşılaştırmalardan sonra değerlendirilir; `&&`, `||`'den önce gelir. Aynı düzeydeki işleçlerde soldan sağa gidilir. Bu yüzden `10 - 4 - 3` ifadesi *(10 − 4) − 3 = 3*'tür, *10 − (4 − 3) = 9* değil. `100 / 10 / 5` de *(100 / 10) / 5 = 2*'dir. Atama ise sağdan sola işler: `a = b = 4;` önce b'ye 4'ü yazar, sonra bu değeri a'ya yazar.

Derleyici bir ifadeyi okurken onu, önceliğe göre bir **ifade ağacına** (*expression tree*) dönüştürür. Ağacın yapraklarında sabitler ve değişkenler, iç düğümlerinde işleçler bulunur. En alttaki işlemler önce, kökteki işlem en son hesaplanır. Sunumdaki ifade bu süreci adım adım gösterir:

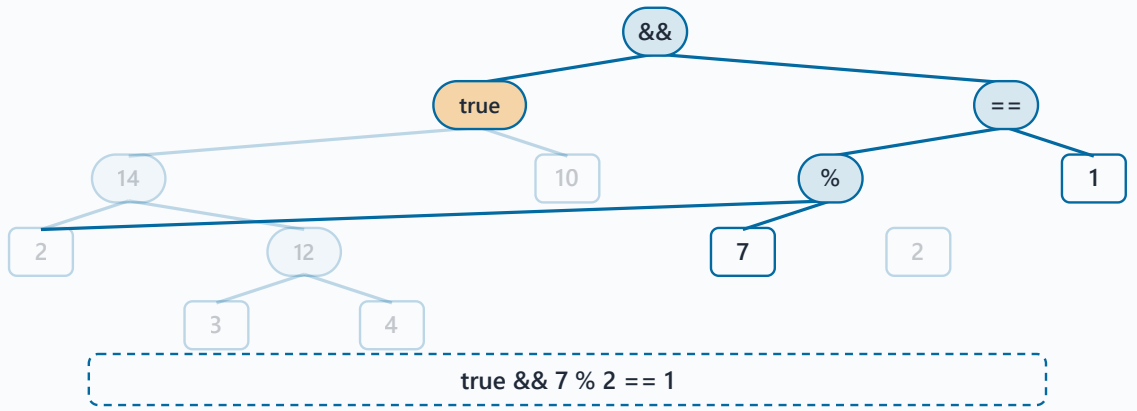
```
2 + 3 * 4 > 10 && 7 % 2 == 1
```

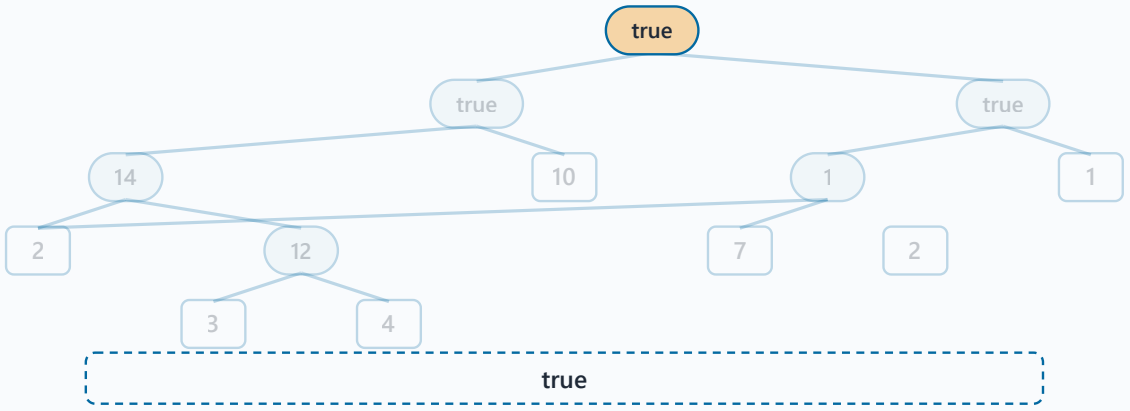


Şekil 6.22: `2 + 3 * 4 > 10 && 7 % 2 == 1` ifadesinin ağacı. Kökte en düşük öncelikli işleç `&&` bulunur; onun iki alt ağacı iki karşılaştırmadır.

Adım	Hesaplanan işlem	İfadenin yeni hâli
0	–	`2 + 3 * 4 > 10 && 7 % 2 == 1`
1	`3 * 4 = 12` (*, +'dan önce)	`2 + 12 > 10 && 7 % 2 == 1`
2	`2 + 12 = 14`	`14 > 10 && 7 % 2 == 1`
3	`14 > 10` → true	`true && 7 % 2 == 1`
4	`7 % 2 = 1`	`true && 1 == 1`
5	`1 == 1` → true	`true && true`
6	`true && true` → true	`true`

Tablo 6.18: İfadenin adım adım sadeleşmesi. Her adımda en derindeki işlem hesaplanır.





Şekil 6.23: Ağacın 3. ve 6. adımı. Sol alt ağaç true değerine indirgenmiştir. Sol taraf true olduğu için `&&` sağ tarafı da hesaplar; son adımda kök true olur.

Üçüncü adımda bir tip değişimi olur: `14 > 10` karşılaştırması iki int alır ama boolean üretir. Ağacın üst katlarında artık sayılar değil mantıksal değerler dolaşır. `&&` işlecinin iki işleneni de boolean olmalıdır. Sol taraf true olduğu için sonuç henüz belli değildir; sağ taraf da hesaplanır. Sol taraf false olsaydı sağ tarafa hiç bakılmazdı.

```
System.out.println(2 + 3 * 4);
System.out.println((2 + 3) * 4);
System.out.println(10 - 4 - 3);
System.out.println(100 / 10 / 5);
System.out.println(2 + 3 * 4 > 10 && 7 % 2 == 1);
System.out.println(10 - 4 / 2 * 3 + 7 % 4);
System.out.println(-3 * 2 + 8 / 3 % 2);
int a, b;
a = b = 4;
System.out.println(a + " " + b);
boolean p = true, q = false, r = true;
System.out.println(p || q && !r);
System.out.println((p || q) && !r);
```

```
14
20
3
2
true
7
-6
4 4
true
false
```

Son iki satır mantık işleçlerinin önceliğini gösterir. `p || q && !r` ifadesinde önce `!r`, sonra `&&`, en son `||` uygulanır: *true || (false && false) = true*. Parantezle sıra değiştirilince sonuç da değişir: *(true || false) && false = false*. `-3 * 2 + 8 / 3 % 2` ifadesinde ise tekli eksi önce gelir, sonra soldan sağa *, / ve % uygulanır: *−6 + (8 / 3) % 2 = −6 + 2 % 2 = −6 + 0 = −6*.

Öncelik kurallarının hepsini ezberlemek gerekmez. Okuyanın aklında soru işareti bırakacak her yerde parantez kullanmak, hem hatayı önler hem de niyeti görünür kılar. `(a + b) / 2` yazmak, `a + b / 2` 'nin ne anlama geldiğini düşünmekten kolaydır.

6.4.7 Karma ifade: öncelik ve tip birlikte

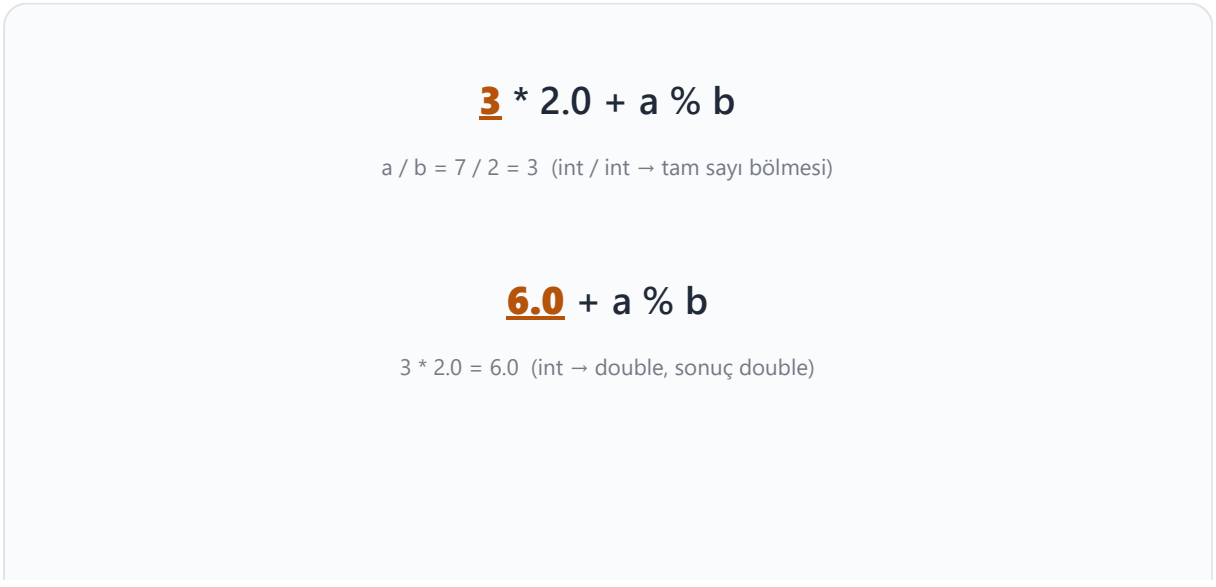
Gerçek ifadelerde öncelik ve tip kuralları birlikte çalışır. Sunumdaki örnek ikisinin etkileşimini gösterir:

```
int a = 7, b = 2;
double r = a / b * 2.0 + a % b;
System.out.println(r);
```

```
7.0
```

Adım	İşlem	Tipler	İfadenin yeni hâli
1	`a / b` = 7 / 2 = 3	int / int → int (tam sayı bölmesi)	`3 * 2.0 + a % b`
2	`3 * 2.0` = 6.0	int * double → double	`6.0 + a % b`
3	`a % b` = 7 % 2 = 1	int % int → int	`6.0 + 1`
4	`6.0 + 1` = 7.0	double + int → double	`7.0`

Tablo 6.19: `a / b * 2.0 + a % b` ifadesinin a = 7, b = 2 için adım adım hesaplanması.



7.0

6.0 + 1 = 7.0 (1 → 1.0, double)

Şekil 6.24: Karma ifadenin sadeleşmesi. `/`, `*` ve `%` aynı öncelikte olduğu için soldan sağa işlenir; `+` hepsinden sonra gelir.

/, * ve % aynı öncelik düzeyindedir ve soldan sağa işlenir; + ise hepsinden sonra gelir. İlk işlem `a / b`'dir ve iki int olduğu için 3.5 değil 3 verir. İfadede 2.0 gibi bir double bulunması bu ilk işlemi etkilemez; 2.0 ancak ikinci işlemde devreye girer. Sonuç 7.0'dır. `a / b` yerine `a / 2.0` yazılsaydı ilk işlem 3.5 verir ve sonuç *3,5 · 2,0 + 1 = 8,0* olurdu. Aynı sayılarla iki farklı cevap çıkması, sonucu işleçlerin sırasının ve işlenen tiplerinin birlikte belirlediğini gösterir.

6.4.8 Kısa devre değerlendirme

`&&` ve `||` işleçlerinin işlerini hızlandıran ve programları güvenli kılan bir özelliği vardır.

Tanım 6.11

Kısa devre değerlendirme (*short-circuit evaluation*), mantık ifadesinin sonucu sol işlenenden belli olduğunda sağ işlenenin hiç hesaplanmamasıdır. `&&` için sol taraf false ise sonuç false'tur; `||` için sol taraf true ise sonuç true'dur. Bu durumlarda sağ taraf çalıştırılmaz.

```
int a = 10, b = 0;
if (b != 0 && a / b > 2) {
    System.out.println("Oran büyük");
}
b = 2;
if (b != 0 && a / b > 2) {
    System.out.println("Oran büyük");
}
```

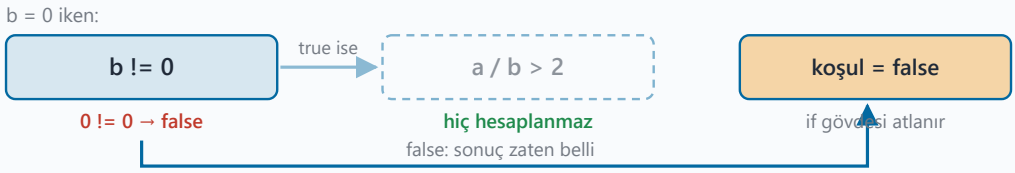
```
Oran büyük
```

Program aynı koşulu iki farklı b değeriyle sınar. Koşulun sol tarafı `b != 0`, sağ tarafı `a / b > 2`'dir. (if deyimi, koşul doğruysa süslü parantez içindeki satırları çalıştırır; yanlışsa atlar.)

b	Sol: `b != 0`	Sağ: `a / b > 2`	Koşul	Çıktı
0	0 != 0 → false	hesaplanmaz	false	yok (blok atlanır)

b	Sol: `b != 0`	Sağ: `a / b > 2`	Koşul	Çıktı
2	2 != 0 → true	10 / 2 = 5; 5 > 2 → true	true	Oran büyük

Tablo 6.20: Kısa devre örneğinin izleme tablosu.



Şekil 6.25: b = 0 iken kısa devre. Sol koşul false olduğu için `&&` ifadesinin sonucu bellidir; `a / b > 2` hiç hesaplanmaz ve sıfıra bölme hatası oluşmaz.

b = 0 iken sağ taraf çalıştırılsaydı `a / b` tam sayıyı sıfıra bölerdi ve program `ArithmeticException` ile çökerdi. Kısa devre bunu önler: sol taraf false olduğunda sonuç ne olursa olsun false'tur ve Java sağ tarafa bakmaz. Bu yüzden koşulların sırası önemlidir. Koşullar ters yazılırsa koruma ortadan kalkar:

```
int a = 10, b = 0;
if (a / b > 2 && b != 0) {
    System.out.println("Oran büyük");
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

Burada önce `a / b` hesaplanır ve program, `b != 0` denetimine sıra gelmeden durur. Koruyucu koşul her zaman sola, korunan işlem sağa yazılır. `||` için aynı fikir tersinden işler: `b == 0 || a / b > 2` ifadesinde b = 0 ise sol taraf true'dur ve bölme yapılmaz.

```
int a = 10, b = 0;
if (b == 0 || a / b > 2) {
    System.out.println("b sıfır ya da oran büyük");
}
int sayac = 0;
boolean s1 = false && (++sayac > 0);
boolean s2 = true || (++sayac > 0);
boolean s3 = false & (++sayac > 0);
System.out.println(sayac + " " + s1 + " " + s2 + " " + s3);
```

```
b sıfır ya da oran büyük
1 false true false
```

İkinci kısımdaki sayaç, hangi sağ tarafların gerçekten çalıştığını sayar. `false && (++sayac > 0)` ve `true || (++sayac > 0)` ifadelerinde sağ taraf kısa devre yüzünden çalışmaz. Tek `&` ise kısa devre yapmayan "ve" işlecidir; iki tarafı da her zaman hesaplar. Sayacın 1 olması, yalnızca bu üçüncü ifadede artırmanın gerçekleştiğini gösterir. Günlük programlamada koşullar için hep `&&` ve `||` kullanılır.

6.4.9 Dizgi birleştirme

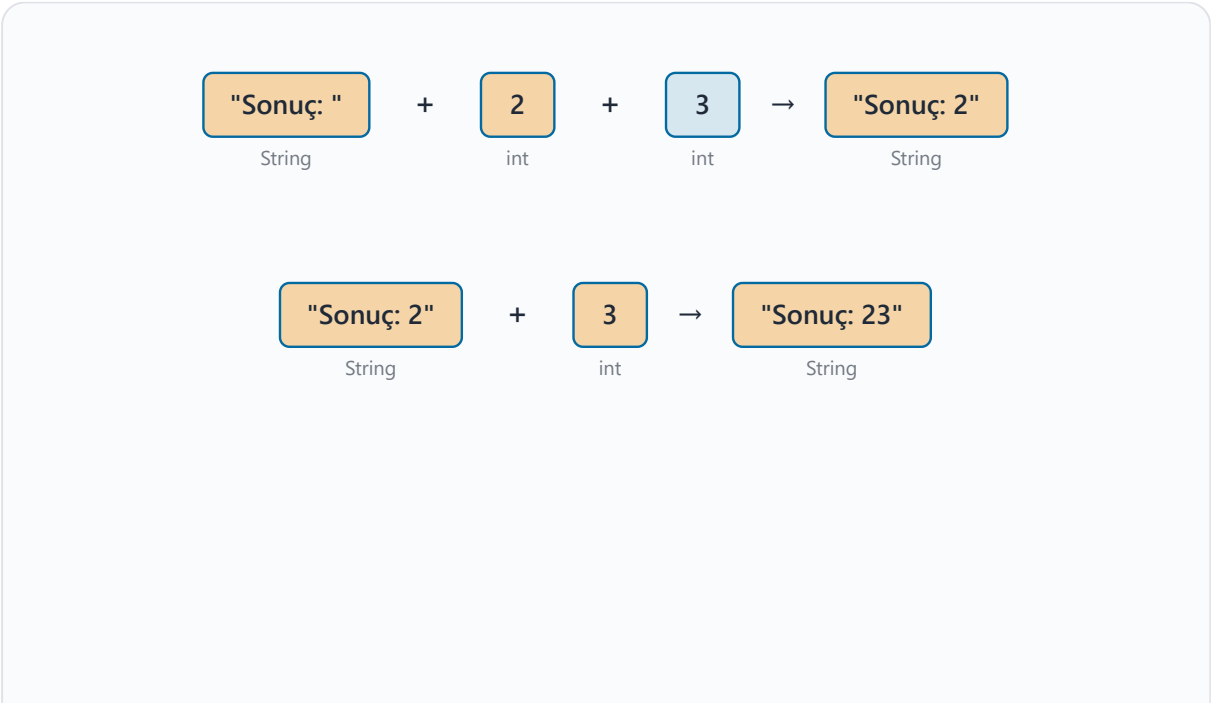
`+` işlecinin de iki yüzü vardır. İki işlenen sayıysa toplama yapar. İşlenenlerden en az biri bir dizgiyse (String) **dizgi birleştirme** (*string concatenation*) yapar: öteki işlenen metne çevrilir ve iki metin uç uca eklenir. İşlecin davranışını işlenenlerin tipi belirler.

```
System.out.println(2 + 3);       // 5
System.out.println("2" + "3");   // 23
System.out.println("Sonuç: " + 2 + 3); // Sonuç: 23
System.out.println("Sonuç: " + (2 + 3)); // Sonuç: 5
```

Dört satırın sonuçlarını açıklamak için iki kural yeterlidir. Birincisi, `+` soldan sağa işler. İkincisi, her adımda o adımdaki iki işlenene bakılır: ikisi de sayıysa toplama, biri dizgiyse birleştirme yapılır.

İfade	1. adım	2. adım	Çıktı
`2 + 3`	int + int → toplama: 5	–	5
`"2" + "3"`	String + String → birleştirme: "23"	–	23
`"Sonuç: " + 2 + 3`	String + int → "Sonuç: 2"	String + int → "Sonuç: 23"	Sonuç: 23
`"Sonuç: " + (2 + 3)`	parantez: int + int → 5	String + int → "Sonuç: 5"	Sonuç: 5

Tablo 6.21: Dört `println` satırının adımları.





Şekil 6.26: `"Sonuç: " + 2 + 3` ifadesinde önce `"Sonuç: " + 2` birleştirilir ve "Sonuç: 2" olur; sonra 3 de metne eklenir. Parantezli yazılışta önce *2 + 3 = 5* toplanır, sonra "Sonuç: 5" oluşur.

Üçüncü satır en çok şaşırtan satırdır. Okuyan kişi 2 ile 3'ün toplanacağını bekler ama Java soldan sağa gider: ilk adımda “Sonuç: ” dizgisiyle 2 birleşir. Bu adımın sonucu bir dizgidir; ikinci adımda dizgiyle 3 birleşir. 2 + 3 hiç toplanmaz. Parantez toplamanın önce yapılmasını sağlar. Kural, işlemin başında da geçerlidir:

```
System.out.println(2 + 3);
System.out.println("2" + "3");
System.out.println("Sonuç: " + 2 + 3);
System.out.println("Sonuç: " + (2 + 3));
System.out.println(2 + 3 + " elma");
System.out.println("Sonuç: " + 2 * 3);
System.out.println("Sonuç: " + 7 / 2);
System.out.println('A' + 1 + "B");
System.out.println("A" + 'B' + 1);
System.out.println(1 + 'A');
System.out.println("x" + 1.5 + true);
String s = "Sonuç: ";
s += 5;
System.out.println(s);
```

```
5
23
Sonuç: 23
Sonuç: 5
5 elma
Sonuç: 6
Sonuç: 3
66B
AB1
66
x1.5true
Sonuç: 5
```

- `2 + 3 + " elma"` : soldaki iki işlenen sayıdır ve önce toplanır; sonuç “5 elma”dır.
- `"Sonuç: " + 2 * 3` ve `"Sonuç: " + 7 / 2` : * ve /, +'dan önceliklidir. Önce sayısal işlem yapılır (6 ve 3), sonra birleştirme.

- `'A' + 1 + "B"`: ilk adımda char ile int toplanır (*65 + 1 = 66*), sonra "B" eklenir: "66B". `"A" + 'B' + 1` ise baştan dizgiyle başladığı için "AB1" verir.
- `"x" + 1.5 + true`: double ve boolean değerler de metne çevrilir: "x1.5true".
- `s += 5`: bileşik atama dizgilerde de çalışır ve metnin sonuna ekler.

Dizgilerle ilgili bir uyarı da karşılaştırmayla ilgilidir. İki dizginin içeriğinin aynı olup olmadığı `==` ile değil `equals` metoduyla denetlenir. `==`, String gibi referans tiplerinde iki değişkenin aynı nesneyi gösterip göstermediğine bakar, içeriğe bakmaz:

```
String s1 = "Ada";
String s2 = new String("Ada");
System.out.println(s1 == s2);
System.out.println(s1.equals(s2));
```

```
false
true
```

`s1` ve `s2` aynı "Ada" metnini taşır ama bellekte iki ayrı nesnedir. `==` false, `equals` true verir.

6.5 Girdi ve çıktı

Şimdiye kadarki programlar hep aynı sayılarla çalıştı; değerler kodun içine yazılmıştı. Bir sıcaklık dönüştürücüsü ya da not ortalaması hesaplayan bir program ise her çalıştığında farklı değerlerle çalışmalıdır. Bu değerler kullanıcıdan, çoğunlukla klavyeden alınır. Java'da klavyeden okumanın en kolay yolu `Scanner` sınıfıdır. Sonuçların düzgün hizalanmış, belirli sayıda ondalık basamakla yazılması için de `printf` metodu kullanılır.

6.5.1 Scanner ile girdi

```
Scanner klavye = new Scanner(System.in);
System.out.print("Yaş: ");
int yas = klavye.nextInt();
klavye.nextLine(); // satır sonunu tüket
System.out.print("Ad: ");
String ad = klavye.nextLine();
```

Scanner, Java'nın `java.util` paketindedir. Kullanılmadan önce dosyanın en başında, sınıf tanımından önce `import java.util.Scanner;` satırı yazılır. Bu satır derleyiciye Scanner adının nereden geleceğini söyler; yazılmazsa "cannot find symbol" hatası alınır.

1. `Scanner klavye = new Scanner(System.in);` : `new` ile bir Scanner nesnesi oluşturulur ve standart girdiye, yani klavyeye (`System.in`) bağlanır. `klavye` bu nesneye referans tutan değişkendir; adı serbestçe seçilebilir.
2. `System.out.print("Yaş: ");` : kullanıcıya ne girmesi gerektiğini söyleyen bir **istem** (*prompt*) yazılır. `println` değil `print` kullanıldığı için satır sonu eklenmez; kullanıcı aynı satırda yazar.
3. `int yas = klavye.nextInt();` : program durur ve kullanıcının bir tam sayı yazıp Enter'a basmasını bekler. Okunan sayı `yas` değişkenine yazılır.
4. `klavye.nextLine();` : satır sonunda kalan Enter karakteri tüketilir. `nextInt()` sayının arkasındaki Enter'ı okumaz; bu çağrı onu tampondan alır.
5. `String ad = klavye.nextLine();` : kullanıcının yazdığı satırın tamamı, boşluklar dahil, okunur.

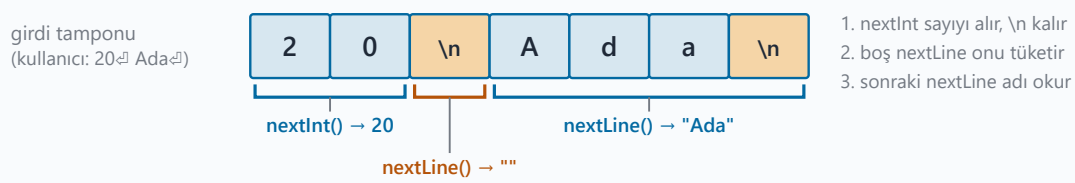
Scanner'ın okuma metotları, beklenen tipe göre seçilir: `nextInt()` int, `nextLong()` long, `nextDouble()` double, `nextBoolean()` boolean okur. `next()` boşluğa kadar tek bir kelime, `nextLine()` ise satırın sonuna kadar her şeyi okur. Sayı ve kelime okuyan metotlar, girdideki boşlukları ve satır sonlarını ayırıcı sayar ve atlar. Kullanıcı 20, 30 ve 40 sayılarını ister aynı satıra boşluklarla, ister ayrı satırlara yazsın; üç `nextInt()` çağrısı üçünü de okur.

Programın ekranda görünen hâli, kullanıcının yazdıklarıyla birlikte şöyledir (kullanıcı 20 ve Ada Yılmaz yazar):

```
Yaş: 20
Ad: Ada Yılmaz
```

6.5.2 Girdi tamponu adım adım

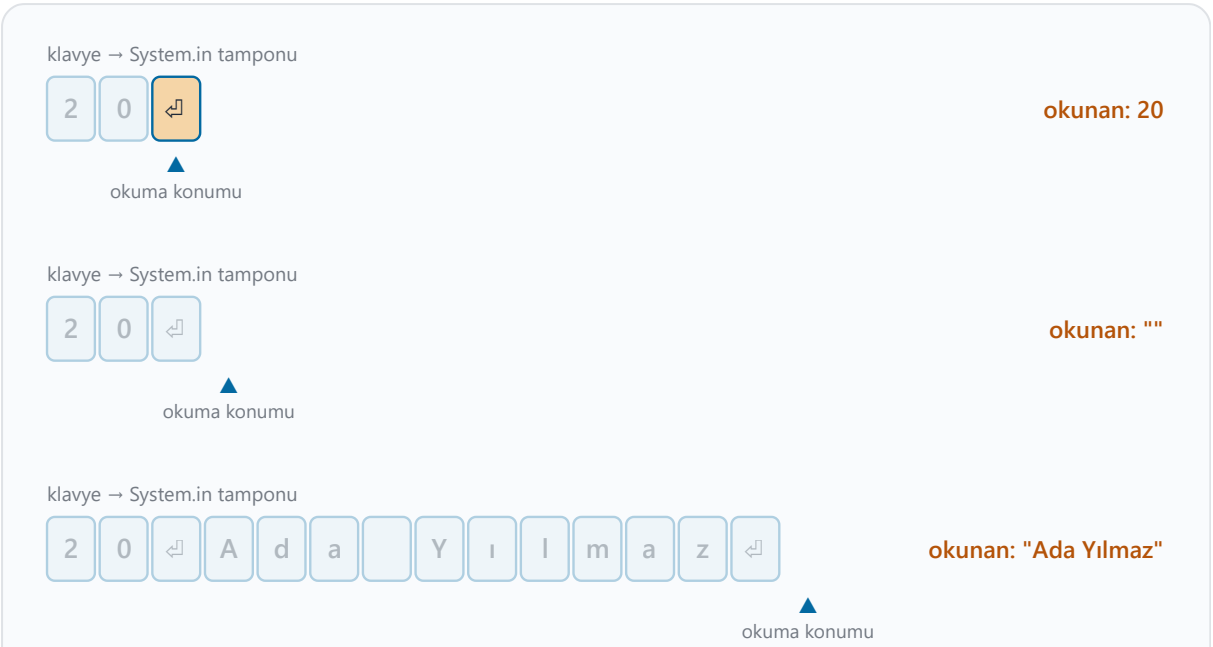
Kullanıcının yazdıkları Scanner'a doğrudan değil, bir **tampon** (*buffer*) üzerinden ulaşır. Tampon, klavyeden gelen karakterlerin okunana kadar beklediği bir sıradır. Kullanıcı Enter'a bastığında yazdığı satır, sonundaki satır sonu karakteriyle birlikte tampona girer. Satır sonu, Java'da `\n` ile, şekillerde ⏎ simgesiyle gösterilir. Scanner'ın her okuma metodu tampondan belirli karakterleri alır, gerisini sonraki okumalara bırakır.



Şekil 6.27: Girdi tamponu: kullanıcı önce 20⏎, sonra Ada⏎ yazar. `nextInt()` 20'yi okur ve satır sonunu tamponda bırakır. Boş `nextLine()` bu satır sonunu tüketir. Sonraki `nextLine()` adı okur.

Deyim	Ne olur?	Okunan	Tampon
`new Scanner(System.in)`	Scanner klavyeye bağlanır	–	boş
`print("Yaş: ")`	istem yazılır; imleç aynı satırda kalır	–	boş
`nextInt()`	tampon boş: bekler. Kullanıcı 20⏎ yazar	20	⏎
`nextLine()`	⏎'e kadar okur, ⏎'i tüketir	"" (boş dizgi)	boş
`print("Ad: ")`	istem yazılır	–	boş
`nextLine()`	tampon boş: bekler. Kullanıcı Ada Yılmaz⏎ yazar	"Ada Yılmaz"	boş

Tablo 6.22: Scanner örneğinin izleme tablosu. Tampon sütunu, deyim çalıştıktan sonra okunmayı bekleyen karakterleri gösterir.



Şekil 6.28: Tamponun üç anı. `nextInt` yalnızca 20'yi okur; okuma konumu ⏎'in önünde durur. `nextLine` ⏎'e kadar okur ve boş dizgi döndürür. Son `nextLine` satırın tamamını okur: "Ada Yılmaz". Okunan karakterler soluk gösterilmiştir.

Tablonun üçüncü satırı her şeyi açıklar. `nextInt()` sayının rakamlarını okur ve rakam olmayan ilk karakterde durur. Bu karakter kullanıcının bastığı Enter'dır ve tamponda kalır. `nextLine()` ise "satırın geri kalanını, satır sonuna kadar" okur. Satır sonu karakteri ise hemen okuma konumundadır; bu yüzden ilk `nextLine()` hiç beklemeden boş dizgi döndürür ve satır sonunu tüketir. Bu çağrının döndürdüğü değer bir yere atanmaz; çağrının tek amacı tamponu temizlemektir. Son `nextLine()` boş tamponla karşılaşır ve kullanıcıyı bekler. Kullanıcının yazdığı "Ada Yılmaz" boşluğuyla birlikte okunur; satır sonu tüketilir ama döndürülen dizgiye eklenmez.

6.5.3 Satır sonu tüketilmezse

Tamponu temizleyen satır unutulursa ne olur? Sunumdaki ikinci program bu durumu gösterir:

```
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 + "]");
```

Kullanıcı 20 yazıp Enter'a bastığında ekranda şu görünür ve program biter:

```
Yaş: 20
Ad: []
```



Şekil 6.29: Satır sonu tüketilmezse: `nextInt`'ten kalan ⏎ tampondadır. `nextLine` beklemeden bu ⏎'e kadar okur ve boş dizgi döndürür; kullanıcıya ad sorulamaz.

Program adı sormadan bitmiş gibi görünür. Gerçekte "Ad: " istemi yazılmış, `nextLine()` çağrılmış ve tamponda bekleyen satır sonunu bularak hemen dönmüştür. `ad` değişkeni boş dizgidir; köşeli parantezlerin arasında hiçbir şey yoktur. Kullanıcı ikinci bir şey yazmaya fırsat bulamaz. Çözüm, sunumdaki ilk programdaki gibi `nextInt()`'ten sonra fazladan bir `klavye.nextLine();` çağırmaktır. Bir başka çözüm, sayıları da satır olarak okuyup dönüştürmektir: `int yas = Integer.parseInt(klavye.nextLine());`.

Aynı etki, kullanıcı sayıyla adı aynı satıra yazdığında da görülür. Girdi "20 Ada Yılmaz ⏎ " ise `nextInt()` 20'yi, sonraki `nextLine()` ise satırın geri kalanını okur: " Ada Yılmaz". Baştaki boşluk da dizgiye dahildir. `next()` ile `nextLine()` arasındaki fark da benzer biçimde görülür: "Ada Yılmaz⏎" girdisinde `next()` yalnızca "Ada"yı, ardından gelen `nextLine()` ise " Yılmaz"ı okur.

Sık yapılan hata

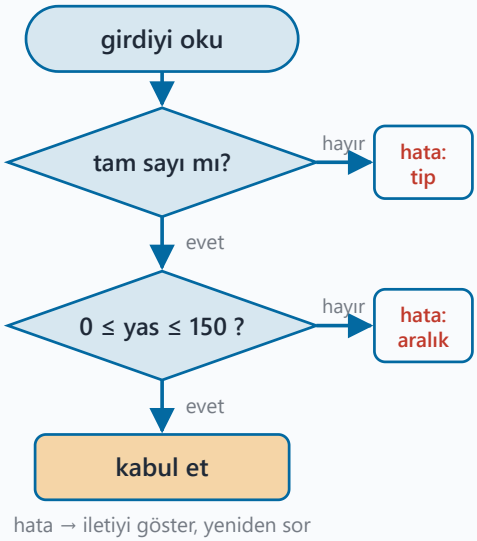
`nextInt()`, `nextDouble()` ya da `next()` çağrısından hemen sonra `nextLine()` ile bir satır okumaya çalışmak. İlk `nextLine()`, önceki satırdan kalan satır sonunu okur ve boş dizgi döndürür. Sayı okuduktan sonra satır okunacaksa araya tamponu temizleyen bir `klavye.nextLine();` konur.

6.5.4 Girdi doğrulama

Kullanıcının her zaman beklenen değeri gireceği varsayılamaz. Yaş sorulan bir programa "yirmi" yazılabilir, bir hata sonucu −5 ya da 200 girilebilir. `nextInt()`, sayı beklediği yerde sayı olmayan bir kelimeyle karşılaşınca programı durdurur:

```
Yaş: yirmi
Exception in thread "main" java.util.InputMismatchException
```

Programın bu tür girdilere dayanıklı olması için **girdi doğrulama** (*input validation*) yapılır. Doğrulama iki katmanlıdır. Önce **tip** denetlenir: girilen şey gerçekten bir tam sayı mı? Sonra **aralık ve iş kuralı** denetlenir: tam sayı makul bir yaş mı?



Şekil 6.30: Girdi doğrulama akışı. Önce girdinin tipi, sonra aralığı denetlenir. İki denetimden geçen değer kabul edilir; geçemeyen için hata iletisi gösterilir ve değer yeniden sorulur.

Scanner, tip denetimi için `hasNextInt()` gibi metotlar sunar. Bu metot girdiyi okumadan bakar: sıradaki parça bir tam sayıysa true, değilse false döndürür. Aşağıdaki program iki denetimi de `if` deyimleriyle yapar. `if (koşul) { … } else { … }` yapısı, koşul doğruysa ilk bloğu, yanlışsa `else`'ten sonraki bloğu çalıştırır.

```
Scanner klavye = new Scanner(System.in);
System.out.print("Yaş: ");
if (klavye.hasNextInt()) {
    int yas = klavye.nextInt();
    if (yas >= 0 && yas <= 150) {
        System.out.println("Kabul: " + yas);
    } else {
        System.out.println("Hata: yaş 0 ile 150 arasında olmalı.");
    }
} else {
    System.out.println("Hata: tam sayı bekleniyor (örnek: 20).");
}
```

Program üç farklı girdiyle çalıştırıldığında şu çıktıları verir:

```
Yaş: yirmi
Hata: tam sayı bekleniyor (örnek: 20).

Yaş: 200
Hata: yaş 0 ile 150 arasında olmalı.

Yaş: 20
Kabul: 20
```

"yirmi" girildiğinde `hasNextInt()` false döndürür; `nextInt()` hiç çağrılmadığı için program çökmez ve anlaşılır bir ileti yazar. 200 girildiğinde tip denetimi geçilir ama aralık denetimi başarısız olur. 20 her iki denetimi de geçer. Hata iletisinin beklenen biçimi açıkça söylediğine dikkat edilmelidir: "Geçersiz girdi" yerine "tam sayı bekleniyor (örnek: 20)" yazmak, kullanıcıya ne yapacağını gösterir. Gerçek bir programda yanlış girdiden sonra değer yeniden sorulur; bu tekrar, bir döngüyle yapılır.

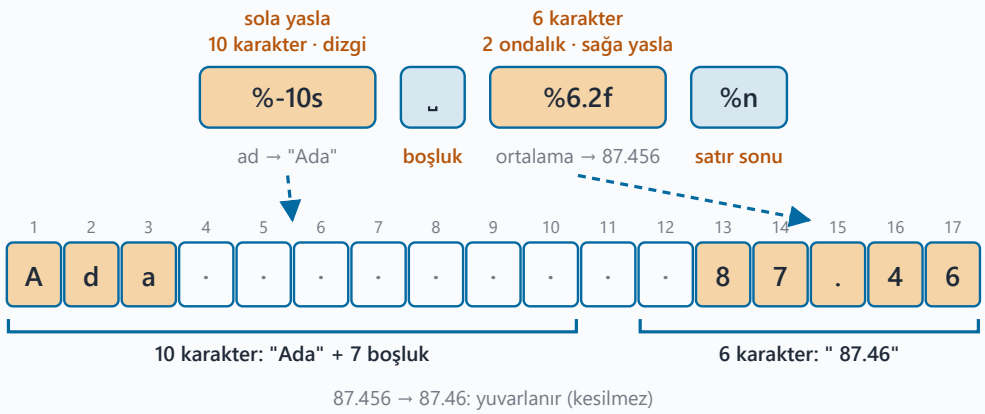
6.5.5 Biçimli çıktı: printf

`println` değerleri olduğu gibi yazar: 87.456 değeri 87.456, 2.0/3 değeri 0.6666666666666666 olarak görünür. Bir not çizelgesinde ise ortalamaların iki basamakla ve sütunlar hizalı yazılması istenir. Bu iş için **biçimli çıktı** (*formatted output*) üreten `printf` metodu kullanılır.

```
String ad = "Ada";
double ortalama = 87.456;
System.out.printf("%-10s %6.2f%n", ad, ortalama);
```

```
Ada         87.46
```

`printf`'in ilk argümanı bir **biçim dizgisidir** (*format string*). Biçim dizgisindeki sıradan karakterler aynen yazılır. % ile başlayan parçalar ise **biçim belirteçleridir** (*format specifier*): her biri, sıradaki argümanın nasıl yazılacağını tarif eder ve argümanları sırayla tüketir.



Şekil 6.31: `%-10s %6.2f%n` biçim dizgisinin çözümlenmesi. `%-10s` "Ada"yı 10 karakterlik alana sola yaslı yazar (3 harf + 7 boşluk). Sonra bir boşluk gelir. `%6.2f`, 87.456'yı iki basamağa yuvarlar ve 6 karakterlik alana sağa yaslar: " 87.46".

Bir biçim belirteci % işaretiyle başlar ve bir **dönüşüm harfiyle** biter. Aradaki üç parça isteğe bağlıdır. **Bayraklar** hizalamayı ve dolguyu belirler: `-` sola yaslar, `0` boşluk yerine sıfırla doldurur, `,` binlik ayırıcı ekler. **Genişlik**, alanın en az kaç karakter olacağını söyler. **Kesinlik** nokta ile yazılır ve ondalık basamak sayısını verir. En sık kullanılan dönüşüm harfleri şunlardır:

Belirteç	Anlam	Örnek	Çıktı
`%d`	tam sayı	`printf("[%d]", 42)`	`[42]`
`%5d`	5 karakter, sağa yaslı	`printf("[%5d]", 42)`	`[ 42]`
`%-5d`	5 karakter, sola yaslı	`printf("[%-5d]", 42)`	`[42 ]`
`%05d`	5 karakter, soldan sıfırlı	`printf("[%05d]", 42)`	`[00042]`
`%,d`	binlik ayırıcılı	`printf("[%,d]", 1234567)`	`[1,234,567]`
`%f`	kayan noktalı, 6 basamak	`printf("[%f]", Math.PI)`	`[3.141593]`
`%.3f`	3 ondalık basamak	`printf("[%.3f]", 2.0005)`	`[2.001]`
`%e`	bilimsel gösterim	`printf("[%e]", 12345.678)`	`[1.234568e+04]`
`%s`	dizgi (her değer metne çevrilir)	`printf("[%s]", 3.5)`	`[3.5]`
`%c`	karakter	`printf("[%c]", 'Z')`	`[Z]`
`%b`	boolean	`printf("[%b]", false)`	`[false]`
`%n`	satır sonu	`printf("a%nb")`	a ve b ayrı satırda
`%%`	% işaretinin kendisi	`printf("%%%d indirim", 20)`	`%20 indirim`

Tablo 6.23: Sık kullanılan biçim belirteçleri. Çıktılar köşeli parantez içinde, alan genişliği görünsün diye verilmiştir (JDK 21, Locale.US).

Tablodaki örnekler aşağıdaki programdan alınmıştır. Program aynı zamanda iki ayrıntıyı daha gösterir. Genişlik bir alt sınırdır: `%3s` belirteci 8 harflik bir adı kesmez, olduğu gibi yazar; `%2d` de 12345'i kesmez. Ayrıca `String.format` metodu aynı biçimlendirmeyi yapar ama sonucu yazmak yerine bir dizgi olarak döndürür.

```
String ad = "Ada";
double ortalama = 87.456;
System.out.printf("%-10s %6.2f%n", ad, ortalama);
System.out.printf("[%-10s][%6.2f]%n", ad, ortalama);
System.out.printf("%-8s|%6.2f|%05d%n", "Ada", 3.14159, 42);
System.out.printf("%-6s|%7.3f|%04d%n", "Can", 2.71828, 7);
System.out.printf("[%5d][%-5d][%05d][%d]%n", 42, 42, 42, 42);
System.out.printf("[%.0f][%.1f][%.3f][%f]%n", 2.5, 2.45, 2.0005, Math.PI);
System.out.printf("[%e][%.2e]%n", 12345.678, 0.000123);
System.out.printf("[%,d][%,.2f]%n", 1234567, 1234567.891);
System.out.printf("[%s][%s][%s][%b][%c]%n", 42, 3.5, true, false, 'Z');
System.out.printf("%%%d indirim%n", 20);
System.out.printf("[%3s][%2d]%n", "UzunİSim", 12345);
System.out.printf("[%.1f][%.1f][%.1f][%.1f]%n", 0.25, 0.35, 0.05, 2.25);
String s = String.format("%d:%02d:%02d", 1, 2, 5);
System.out.println(s);
```

```
Ada         87.46
[Ada       ][ 87.46]
Ada     |  3.14|00042
Can   |  2.718|0007
[   42][42   ][00042][42]
[3][2.5][2.001][3.141593]
[1.234568e+04][1.23e-04]
[1,234,567][1,234,567.89]
[42][3.5][true][false][Z]
%20 indirim
[UzunİSim][12345]
[0.3][0.4][0.1][2.3]
1:02:05
```

`%.1f` ile yazılan 0.25, 0.35, 0.05 ve 2.25 değerlerinin hepsi yukarı yuvarlanır: 0.3, 0.4, 0.1, 2.3. printf yuvarlarken sayının ondalık yazılışına bakar ve yarımları yukarı yuvarlar.

Sunumdaki ikinci örnek üç belirteci tek satırda kullanır:

```
System.out.printf("%-8s|%6.2f|%05d%n",
                  "Ada", 3.14159, 42);
```

```
Ada     |  3.14|00042
```

Parça	Argüman	Kural	Üretilen
`%-8s`	"Ada"	8 karakter, `-` sola yaslar: 3 harf + 5 boşluk	`Ada␣␣␣␣␣`
`|`	–	düz karakter, aynen yazılır	`|`
`%6.2f`	3.14159	2 basamağa yuvarla (3.14), 6 karaktere sağa yasla: 2 boşluk	`␣␣3.14`
`|`	–	düz karakter	`|`
`%05d`	42	5 karakter, `0` bayrağı soldan sıfırla doldurur	`00042`
`%n`	–	satır sonu	⏎

Tablo 6.24: `"%-8s|%6.2f|%05d%n"` biçim dizgisinin soldan sağa çözümlenmesi.



Şekil 6.32: Üç biçim belirtecinin ürettiği karakterler. ␣ simgesi boşluğu gösterir. Her % belirteci bir argüman tüketir; dikey çizgiler düz metindir.

Belirteç sayısı ile argüman sayısı ve tipleri uyuşmalıdır. Uyuşmazlık derleme sırasında değil, program çalışırken fark edilir. `printf("%d%n", 3.5)` çağrısı `IllegalFormatConversionException` fırlatır, çünkü %d tam sayı bekler ve 3.5 bir double'dır. `printf("%s ve %s%n", "Ada")` çağrısı ise ilk belirteci doldurur, ikinci belirteç için argüman bulamaz ve `MissingFormatArgumentException` ile durur.

Biçimli çıktının son bir ayrıntısı **yerel ayardır** (*locale*). printf, ondalık ayırıcıyı ve binlik ayırıcıyı bilgisayarın dil ve bölge ayarına göre seçer. Türkçe yerel ayarlı bir bilgisayarda aynı program farklı çıktı verir:

```
String ad = "Ada";
double ortalama = 87.456;
System.out.printf("%-10s %6.2f%n", ad, ortalama);
System.out.printf("%-8s|%6.2f|%05d%n", "Ada", 3.14159, 42);
System.out.printf("[%,d][%,.2f]%n", 1234567, 1234567.891);
System.out.printf(Locale.US, "%6.2f%n", ortalama);
System.out.println(ortalama);
```

```
Ada         87,46
Ada     |  3,14|00042
[1.234.567][1.234.567,89]
 87.46
87.456
```

Türkçe ayarda ondalık ayırıcı virgül, binlik ayırıcı noktadır: 87,46 ve 1.234.567,89. İlk argüman olarak `Locale.US` verilirse ayar o çağrı için geçersiz kılınır ve nokta kullanılır. `println` ise yerel ayardan etkilenmez; her zaman nokta kullanır. Derste ve sunumlarda çıktılar Locale.US biçiminde verilir.

Yerel ayar Scanner'ı da etkiler ve bu etki daha can sıkıcıdır. Türkçe yerel ayarda `nextDouble()`, ondalık sayının virgülle yazılmasını bekler. Kullanıcı “37.5” yazarsa `InputMismatchException` fırlatılır; “37,5” yazarsa 37.5 okunur. Locale.US ayarında durum tam tersidir. Programın her bilgisayarda aynı davranması isteniyorsa Scanner oluşturulduktan sonra ayar sabitlenir: `klavye.useLocale(Locale.US);`. Bu satırdan sonra Türkçe ayarlı bir bilgisayarda da “37.5” girdisi okunur.

Sık yapılan hata

Ödev programında ondalık sayı girildiğinde `InputMismatchException` alıp kodu hatalı sanmak. Kod doğru olabilir; sorun, bilgisayarın Türkçe yerel ayarda nokta yerine virgül beklemesidir. Ya girdi virgülle yazılır ya da Scanner'a `useLocale(Locale.US)` uygulanır. (Locale sınıfı da `java.util` paketindedir.)

6.6 Örnekler

Bu kesimdeki küçük programlar, notun bütün araçlarını bir arada kullanır: değişkenler, tipler, dönüşümler, işleçler, Scanner ve printf. Her programda önce kod, sonra adım adım hesap ve en son gerçek çıktı verilir.

6.6.1 Math sınıfı

Karekök, üs alma, mutlak değer gibi işlemler için Java'nın hazır `Math` sınıfı kullanılır. Math, `java.lang` paketindedir. Bu paket her Java programına kendiliğinden dahil edildiği için `import` satırı gerekmez. Math'in metotları bir nesne oluşturmadan, sınıf adıyla çağrılır: `Math.sqrt(16)`.

İfade	Sonuç	Açıklama
`Math.PI`	3.141592653589793	π sabiti
`Math.sqrt(16)`	4.0	karekök (double)
`Math.pow(2, 10)`	1024.0	üs alma (double)
`Math.abs(-7)`	7	mutlak değer
`Math.max(3, 8)`	8	büyük olan (min: küçük)
`Math.round(2.5)`	3	en yakın tam sayı (long)
`Math.round(-2.5)`	-2	yarım değer yukarı yuvarlanır

Tablo 6.25: Sık kullanılan Math öğeleri (sunumdaki tablo; değerler JDK 21 ile doğrulanmıştır).

`sqrt` ve `pow` her zaman double döndürür; tam kare olan 16'nın karekökü bile 4 değil 4.0'dır. `abs`, `max` ve `min` ise argümanın tipini korur: `Math.abs(-7)` int 7, `Math.abs(-7.5)` double 7.5 verir. Argümanlardan biri double ise sonuç da double'dır: `Math.max(3, 8.0)` 8.0 verir.



Şekil 6.33: Math.round sayı doğrusunda. Tam yarım değerler her zaman pozitif sonsuz yönüne yuvarlanır: 2.5 → 3, −2.5 → −2.

`Math.round`, en yakın tam sayıyı bulur. Sayı iki tam sayının tam ortasındaysa (2.5 gibi) yukarıya, yani pozitif sonsuz yönüne yuvarlar. Bu yüzden 2.5 → 3 ama −2.5 → −2 olur; −2, −3'ten büyüktür. Okulda öğretilen "yarımı sıfırdan uzağa yuvarla" kuralı −2.5 için −3 verir; Java'nın kuralı bundan farklıdır. Math sınıfının başka metotları da aşağıdaki programda görülür:

```
System.out.println(Math.PI);
System.out.println(Math.E);
System.out.println(Math.sqrt(16));
System.out.println(Math.pow(2, 10));
System.out.println(Math.abs(-7) + " " + Math.abs(-7.5));
System.out.println(Math.max(3, 8) + " " + Math.min(3, 8));
System.out.println(Math.round(2.5) + " " + Math.round(-2.5) + " " + Math.round(2.4) + " " +
Math.round(-2.6));
System.out.println(Math.floor(2.7) + " " + Math.ceil(2.1) + " " + Math.floor(-2.7) + " " +
Math.ceil(-2.1));
System.out.println(Math.pow(2, 0.5) + " " + Math.sqrt(2));
System.out.println(Math.pow(2, 31) + " " + (int) Math.pow(2, 31));
System.out.println(Math.max(3, 8.0));
System.out.println(Math.sqrt(2) * Math.sqrt(2));
System.out.println(Math.hypot(3, 4));
```

```
3.141592653589793
2.718281828459045
4.0
1024.0
7 7.5
8 3
3 -2 2 -3
2.0 3.0 -3.0 -2.0
1.4142135623730951 1.4142135623730951
2.147483648E9 2147483647
8.0
2.0000000000000004
5.0
```

`Math.floor` sayıyı aşağıya (negatif sonsuza doğru), `Math.ceil` yukarıya (pozitif sonsuza doğru) yuvarlar ve sonucu double olarak verir: *floor(−2,7) = −3,0*, *ceil(−2,1) = −2,0*. `(int)` dönüşümünün sıfıra doğru kestiği hatırlanırsa, negatif sayılarda floor ile (int)'in farklı sonuç verdiği görülür. `Math.pow(2, 31)` 2.147483648E9'dur; bu değer int'e çevrilince sınırda sıkışır ve 2147483647 olur. $\sqrt{2} \cdot \sqrt{2}$ ise kayan nokta hatası yüzünden tam 2 değil 2.0000000000000004'tür. `Math.hypot(3, 4)` dik üçgenin hipotenüsünü verir: $\sqrt{(9 + 16)} = 5$.

6.6.2 İki değişkeni takas etme

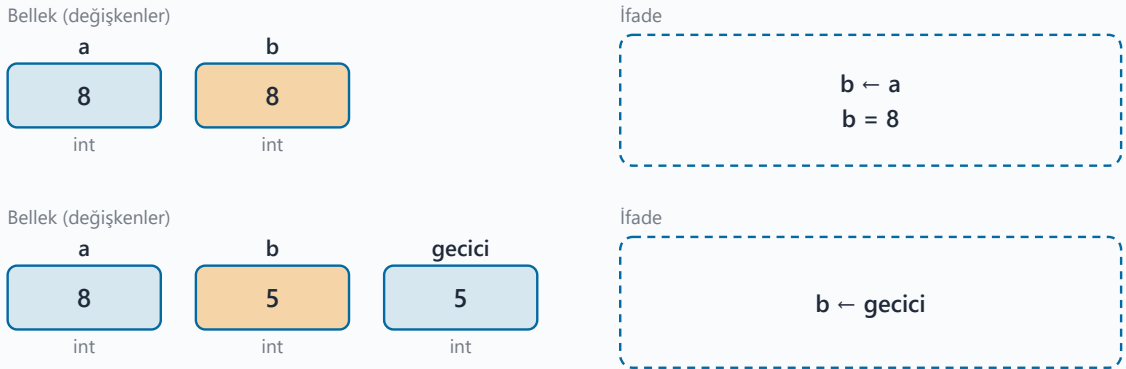
İki değişkenin değerlerini değiştirmek, yani **takas** (*swap*) etmek, sıralama gibi pek çok algoritmanın temel adımıdır. İlk akla gelen çözüm yanlıştır:

```
int a = 5, b = 8;
a = b;                // yanlış deneme
b = a;
System.out.println(a + " " + b);
a = 5; b = 8;         // baştan
int gecici = a;       // doğru yöntem
a = b;
b = gecici;
System.out.println(a + " " + b);
```

```
8 8
8 5
```

Deyim	a	b	gecici	Açıklama
`int a = 5, b = 8;`	5	8	–	hedef: a = 8, b = 5
`a = b;`	8	8	–	a'nın eski değeri 5 silindi
`b = a;`	8	8	–	5 artık hiçbir kutuda yok
`println`	8	8	–	çıktı: 8 8 (başarısız)
`a = 5; b = 8;`	5	8	–	baştan
`int gecici = a;`	5	8	5	a'nın değeri güvende
`a = b;`	8	8	5	
`b = gecici;`	8	5	5	
`println`	8	5	5	çıktı: 8 5 (tamam)

Tablo 6.26: Takas örneğinin izleme tablosu. İlk dört satır yanlış denemeyi, sonraki satırlar doğru yöntemi gösterir.



Şekil 6.34: Yanlış ve doğru takas. Üstte: `a = b` ve `b = a` sonrasında iki kutu da 8'dir. Altta: geçici değişken 5'i korur ve `b = gecici` takası tamamlar.

Yanlış denemenin sorunu, atamanın eski değeri silmesidir. `a = b` çalıştığında a'daki 5 kaybolur; sonraki `b = a` yalnızca 8'i geri kopyalar. İki bardaktaki suyu yer değiştirmek için üçüncü bir bardak gerektiği gibi, takas da üçüncü bir kutu ister. Bu kutuya **geçici değişken** (*temporary variable*) denir. Doğru yöntem üç atama ve bir geçici değişken kullanır.

Sayılarda geçici değişken olmadan da takas yapılabilir: `a = a + b; b = a - b; a = a - b;`. a = 5, b = 8 için a önce 13 olur, sonra b = 13 − 8 = 5, en son a = 13 − 5 = 8 bulunur. Bu yöntem taşmaya karşı şaşırtıcı biçimde dayanıklıdır:

```
int a = 5, b = 8;
a = a + b;
b = a - b;
a = a - b;
System.out.println(a + " " + b);
int x = 2_000_000_000, y = 1_500_000_000;
x = x + y;
System.out.println("ara: " + x);
y = x - y;
x = x - y;
System.out.println(x + " " + y);
```

```
8 5
ara: -794967296
1500000000 2000000000
```

İki milyar ile bir buçuk milyarın toplamı int'i taşar ve ara değer −794.967.296 olur. Yine de sonuç doğru çıkar: taşma sırasında eklenen 2^{32}, sonraki çıkarmalarda geri alınır. Buna karşın bu hile okunabilirliği bozar ve yalnızca sayılarla çalışır; günlük programlamada geçici değişkenli takas tercih edilir.

6.6.3 Celsius'tan Fahrenheit'a

Celsius cinsinden bir sıcaklığı Fahrenheit'a çeviren formül *F = C · 9 / 5 + 32*'dir. Program, sıcaklığı klavyeden okur ve sonucu bir ondalık basamakla yazar:

```
Scanner klavye = new Scanner(System.in);
double c = klavye.nextDouble();
double f = c * 9 / 5 + 32;
System.out.printf("%.1f C = %.1f F%n", c, f);
```

Girdi	c	c * 9	… / 5	… + 32 = f	Çıktı
100	100.0	900.0	180.0	212.0	`100.0 C = 212.0 F`
37	37.0	333.0	66.6	98.6	`37.0 C = 98.6 F`

Girdi	c	c * 9	... / 5	... + 32 = f	Çıktı
-40	-40.0	-360.0	-72.0	-40.0	`-40.0 C = -40.0 F`

Tablo 6.27: Üç çalıştırmada formülün soldan sağa hesaplanması (değerler programa ara çıktı eklenerek JDK 21'de alınmıştır).



Şekil 6.35: İkinci çalıştırma: c = 37.0 için *333.0 / 5 + 32 → 98.6* hesaplanır ve f kutusuna yazılır.

Kullanıcı 100 yazsa da `nextDouble()` değeri double olarak okur: c = 100.0. Formül soldan sağa hesaplanır. `c * 9` double ile int'in çarpımıdır ve double'dır; bölme de toplama da kayan noktalı yapılır. %.1f belirteci her iki değeri bir ondalık basamakla yazar. Üçüncü çalıştırma ilginç bir gerçeği gösterir: −40 derece iki ölçekte de aynı sıcaklıktır.

Formülün yazılış sırası önemlidir. Aynı formül `9 / 5 * c + 32` biçiminde yazılsaydı ilk işlem `9 / 5` olurdu. İki int'in bölümü 1,8 değil 1'dir. Sonuç c = 37 için 98.6 yerine *1 · 37 + 32 = 69* çıkar:

```
double c = 37;
double f1 = 9 / 5 * c + 32;
double f2 = 9.0 / 5 * c + 32;
double f3 = c * 9 / 5 + 32;
System.out.println(f1 + " " + f2 + " " + f3);
System.out.println(9 / 5 + " " + 9.0 / 5);
```

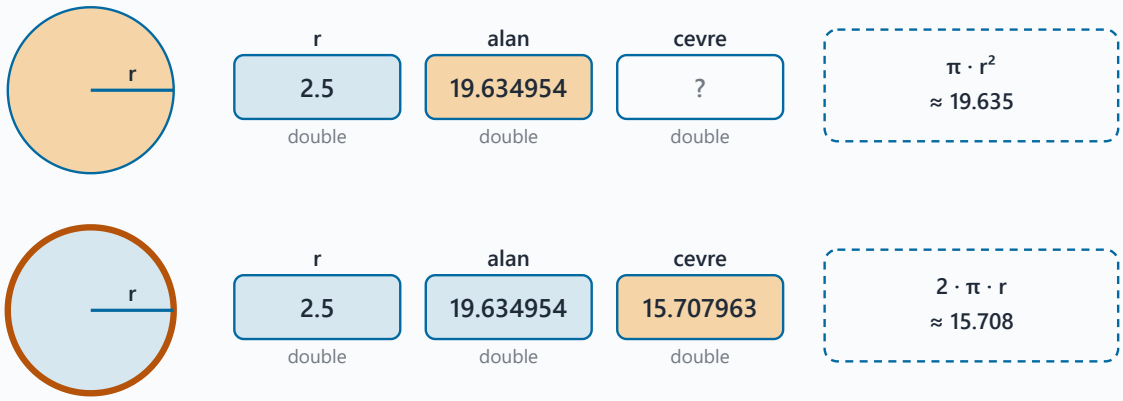
```
69.0 98.60000000000001 98.6
1 1.8
```

`9.0 / 5` yazılınca bölme kayan noktalı olur ve sonuç doğru çıkar. Bu yazılışta değer 98.60000000000001'dir; işlem sırası farklı olduğu için kayan nokta hatası başka bir basamakta görünür. printf ile bir basamağa yuvarlanınca iki yazılış da 98.6 verir.

6.6.4 Dairenin alanı ve çevresi

```
double r = 2.5;
double alan = Math.PI * r * r;
double cevre = 2 * Math.PI * r;
System.out.printf("Alan: %.2f%n", alan);
System.out.printf("Çevre: %.2f%n", cevre);
```

```
Alan: 19.63
Çevre: 15.71
```



Şekil 6.36: r = 2.5 için alan $\pi \cdot r^2 \approx 19{,}635$ ve çevre $2 \cdot \pi \cdot r \approx 15{,}708$ hesaplanır ve kutulara yazılır.

Alan $\pi \cdot r \cdot r = 3{,}14159... \cdot 2{,}5 \cdot 2{,}5$'tir ve `alan` kutusunda 19.634954084936208 değeri durur. Çevre $2 \cdot \pi \cdot r$'dir: 15.707963267948966. π değeri elle 3.14 diye yazılmaz; `Math.PI` 16 anlamlı basamaklı hazır bir sabittir. %.2f belirteci değerleri iki basamağa yuvarlar: 19.634… → 19.63 ve 15.7079… → 15.71. Yuvarlama yalnızca yazdırmayı etkiler; değişkenlerdeki değerler değişmez ve sonraki hesaplarda tam değerler kullanılır. Kare almak için `Math.pow(r, 2)` da yazılabilir; `r * r` hem daha kısa hem daha hızlıdır.

6.6.5 Saniyeyi saate çevirme

Bir süre saniye cinsinden verildiğinde onu saat:dakika:saniye biçimine çevirmek, tam sayı bölmesi ile kalan işlecinin birlikte kullanıldığı tipik bir problemdir.

```
int toplam = 3725;
int saat = toplam / 3600;
int kalan = toplam % 3600;
int dakika = kalan / 60;
int saniye = kalan % 60;
System.out.printf("%d:%02d:%02d%n",
                  saat, dakika, saniye);
```

```
1:02:05
```

Deyim	Hesap	Sonuç	Anlamı
`saat = toplam / 3600`	3725 / 3600	1	kaç tam saat
`kalan = toplam % 3600`	3725 % 3600	125	saatlerden artan saniye
`dakika = kalan / 60`	125 / 60	2	kaç tam dakika
`saniye = kalan % 60`	125 % 60	5	dakikalardan artan saniye
`printf("%d:%02d:%02d%n", …)`	1, 2, 5	1:02:05	%02d: en az 2 hane, soldan 0

Tablo 6.28: toplam = 3725 için izleme tablosu.



Şekil 6.37: Programın sonunda beş kutu: toplam = 3725, saat = 1, kalan = 125, dakika = 2, saniye = 5.

Yöntem büyük birimden küçüğe ilerler. Önce kaç tam saat olduğu bulunur, sonra kalan saniyeler dakika ve saniyeye ayrılır. Sonuç bir sağlamayla doğrulanır: *1 · 3600 + 2 · 60 + 5 = 3725*. %02d belirteci, tek basamaklı dakika ve saniyeleri önüne 0 koyarak iki haneye tamamlar; çıktı 1:2:5 yerine 1:02:05 olur. Aynı yöntem başka sürelerde de çalışır: 86399 saniye 23:59:59, 7384 saniye 2:03:04'tür. Formüller tek satırda da yazılabilir. `toplam % 3600 / 60` ifadesinde % ve / aynı önceliktedir ve soldan sağa işlenir; önce kalan, sonra dakika bulunur.

Örnek 6.4

Kuruş cinsinden verilen bir tutar lira ve kuruşa nasıl ayrılır? 1575 ve 1005 kuruş için sonuç nedir?

```
int tutar = 1575;
System.out.println(tutar / 100 + " lira " + tutar % 100 + " kuruş");
tutar = 1005;
System.out.println(tutar / 100 + " lira " + tutar % 100 + " kuruş");
System.out.printf("%d,%02d TL%n", tutar / 100, tutar % 100);
```

```
15 lira 75 kuruş
10 lira 5 kuruş
10,05 TL
```

1 lira 100 kuruştur. *1575 / 100 = 15* tam lira, *1575 % 100 = 75* kuruş artar. 1005 kuruşta *1005 / 100 = 10* ve *1005 % 100 = 5* bulunur. Tutar Türkçe biçimde yazılırken kuruş iki haneye tamamlanmalıdır; aksi hâlde 10,05 TL yerine 10,5 TL yazılır ve okuyan 50 kuruş anlar. Son satırdaki %02d bu tamamlamayı yapar. Tutarı kuruş cinsinden int ya da long olarak tutmak, double'ın kayan nokta hatalarını da baştan önler.

6.7 Yaygın hatalar

Bu bölümün konularında yapılan hataların çoğu birkaç kalıba girer. Sunumdaki tablo bunların beşini bir araya getirir:

Hata	Örnek	Ne olur?
= ile == karışır	`if (x = 5)`	Derleme hatası: int, boolean değil
Tam sayı bölmesi	`double o = 7 / 2;`	o = 3.0, 3.5 değil
Kapsam dışı kullanım	`{ int k = 1; } k++;`	k blok dışında tanımsız
Yanlış noktalı virgül	`if (x > 0);`	Boş deyim: koşul hiçbir şeyi denetlemez
String ve ==	`s1 == s2`	İçeriği değil, referansı karşılaştırır → `s1.equals(s2)`

Tablo 6.29: Yaygın hatalar (sunumdaki tablo).

= ile == karışması. Koşulda eşitlik denetlenmek istenirken tek eşittir yazılırsa ifade bir atama olur. `x = 5` ifadesinin değeri 5'tir ve tipi int'tir; if ise boolean bir koşul ister. Java bu hatayı derleme sırasında yakalar:

```
error: incompatible types: int cannot be converted to boolean
if (x = 5) {
      ^
1 error
```

C gibi dillerde bu satır derlenir ve koşul her zaman doğru sayılır. Java'nın int ile boolean'ı karıştırmaması, bu hatayı zararsız bir derleme hatasına çevirir.

Tam sayı bölmesi. `double o = 7 / 2;` satırında hedef double'dır ama bölme iki int arasında yapılır: *7 / 2 = 3*. Sonuç double'a genişler ve o = 3.0 olur. Taşma kesimindeki `long sonuc = a * b` hatasıyla aynı kökten gelir: ifadenin tipi, sonucun yazılacağı yere değil, işlenenlere bağlıdır.

Kapsam dışı kullanım. Bir değişken, bildirildiği süslü parantez bloğu boyunca geçerlidir. Bu alana değişkenin **kapsamı** (*scope*) denir. Blok kapanınca değişken ortadan kalkar:

```
{
    int k = 1;
}
k++;
```

```
error: cannot find symbol
k++;
^
  symbol:   variable k
  location: class Kapsam
1 error
```

Yanlış noktalı virgül. if koşulundan hemen sonra konan noktalı virgül, Java için kendi başına bir deyimdir: hiçbir şey yapmayan **boş deyim** (*empty statement*). Koşul yalnızca bu boş deyimi denetler; arkasındaki blok koşuldan bağımsız olarak her zaman çalışır:

```
int x = -4;
if (x > 0);
{
    System.out.println("pozitif");
}
```

```
pozitif
```

x = −4 olduğu hâlde “pozitif” yazılır. Program derlenir ve hiçbir uyarı vermez; bu yüzden bu hata zor bulunur.

String ve ==. Dizgi birleştirme kesiminde görüldüğü gibi, `==` iki String değişkeninin aynı nesneyi gösterip göstermediğine bakar. İçerik karşılaştırması `equals` ile yapılır.

Notun önceki kesimlerinde ayrıntılı ele alınan beş hata da bu listeye eklenebilir:

- tipe sığmayan sonucun sessizce taşması,

- kayan noktalı sayıların `==` ile karşılaştırılması,
- `(int)` dönüşümünün yuvarlama sanılması,
- `"Sonuç: " + 2 + 3` ifadesinde toplama yapılacağının sanılması,
- `nextInt()` 'ten sonra satır sonunun tüketilmemesi.

6.8 Özet

- **Değişken**, belirli tipteki bir değer için bellekte ayrılmış adlı bir kutudur; adı, tipi ve değeri vardır. **Bildirim** tipi ve adı belirtir, **ilk değer verme** başlangıç değerini verir. **Yerel değişken** okunmadan önce kesin olarak değer almış olmalıdır; `final` değişken yalnızca bir kez atanır.
- **Atama** önce sağ tarafı hesaplar, sonra sonucu soldaki değişkene yazar; eski değer silinir. **Bileşik atama** (`+=`, `-=` ...) değişkenin eski değerini kullanarak günceller ve sağ tarafı bir bütün olarak hesaplar.
- **Tanımlayıcı** harf, _ ya da $ ile başlar; boşluk içermez, anahtar sözcük olamaz ve büyük-küçük harfe duyarlıdır. Değişkenler lowerCamelCase, sınıflar UpperCamelCase, sabitler BÜYÜK_HARF ile adlandırılır.
- Java'nın sekiz **ilkel tipi** vardır: byte (8), short (16), int (32), long (64), float (32), double (64), char (16) ve boolean. Varsayılan seçimler int ve double'dır. long sabitleri L, float sabitleri F soneki ister.
- Tam sayılar **ikiye tümleyen** gösterimle saklanır. Aralığı aşan sonuç **taşar** ve sessizce öbür uca sarar: `Integer.MAX_VALUE + 1` → −2147483648. Taşmayı yakalamak için `Math.addExact` gibi metotlar istisna fırlatır. İfadenin tipi işlenenlerden belirlenir; `long s = a * b` yine int ile çarpar.
- **Kayan noktalı sayılar** 0.1 gibi onluk kesirleri tam saklayamaz: `0.1 + 0.2` 0.30000000000000004'tür. double değerler `==` ile değil bir **eşikle** karşılaştırılır. Para hesabında tam sayı kuruş ya da BigDecimal kullanılır.
- char bir UTF-16 kod birimini sayı olarak tutar ('A' = 65); aritmetikte int'e yükselir. boolean yalnızca true ya da false alır ve sayılarla yer değiştirmez.
- **Genişletme** otomatik ve kayıpsızdır (int → double). **Daraltma** açık dönüşüm ister: `(int) 9.8` 9 verir; kesir yuvarlanmaz, sıfıra doğru kesilir. Tam sayı daraltmasında alt bitler kalır: `(byte) 200` −56'dır. Yuvarlama için `Math.round` kullanılır.
- İki tam sayının bölümü **tam sayı bölmesidir**: `5 / 2` 2'dir. İşlenenlerden biri double ise öbürü **yükseltilir** ve `5.0 / 2` 2.5 olur. `(double) (5 / 2)` ise kaybolan kesri geri getirmez. byte, short ve char işlemlerde önce int'e yükselir.
- `%` kalanı verir ve işaretini bölünenden alır. Tam sayıyı sıfıra bölmek `ArithmeticException` fırlatır; kayan noktalı bölme ise Infinity ya da NaN üretir.
- **Sonek** `i++` eski değeri, **önek** `++i` yeni değeri üretir. Bileşik atama sonucu sol tarafın tipine örtük olarak daraltır: `b += 5` derlenir, `b = b + 5` derlenmez.

- Karşılaştırma işleçleri boolean üretir; `&&`, `||` ve `!` boolean değerleri birleştirir. **İşleç önceliği**: parantez, tekli, çarpımsal, toplamsal, sıra, eşitlik, &&, ||, atama. Aynı düzeyde soldan sağa gidilir. Derleyici ifadeyi bir **ifade ağacına** çevirir.
- **Kısa devre**: `&&`'de sol false ise, `||`'de sol true ise sağ taraf hesaplanmaz. Koruyucu koşul (`b != 0`) sola yazılır.
- `+` işlenenlerden biri dizgiyse **birleştirme** yapar ve soldan sağa işler: `"Sonuç: " + 2 + 3` "Sonuç: 23", `"Sonuç: " + (2 + 3)` "Sonuç: 5" verir. Dizgi içeriği `equals` ile karşılaştırılır.
- **Scanner** klavyeden okur; girdi bir **tamponda** bekler. `nextInt()` satır sonunu tamponda bırakır; ardından satır okunacaksa önce bir `nextLine()` ile tüketilir. Girdi önce tip, sonra aralık açısından doğrulanır.
- **printf**, **biçim dizgisindeki biçim belirteçlerini** (%d, %f, %s, %n ...) argümanlarla sırayla doldurur; genişlik, kesinlik ve bayraklar (-, 0, ,) çıktıyı hizalar. Ondalık ayırıcı **yerel ayara** bağlıdır; Türkçe ayarda printf virgül yazar, nextDouble virgül bekler.

6.9 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

değişken *variable* · tip *type* · bildirim *declaration* · ilk değer verme *initialization* · atama *assignment* · yerel değişken *local variable* · sabit *constant / literal* · bileşik atama *compound assignment* · tanımlayıcı *identifier* · anahtar sözcük *keyword* · adlandırma kuralı *naming convention* · ilkel tip *primitive type* · referans *reference* · ikiye tümleyen *two's complement* · işaret biti *sign bit* · taşma *overflow* · istisna *exception* · kayan noktalı sayı *floating-point number* · eşik *epsilon (tolerance)* · dizgi *string* · tip dönüşümü *type conversion* · genişletme *widening conversion* · daraltma *narrowing conversion* · açık dönüşüm *cast* · örtük dönüşüm *implicit conversion* · kesme *truncation* · tam sayı bölmesi *integer division* · tip yükseltme *type promotion* · işleç *operator* · işlenen *operand* · ifade *expression* · kalan *remainder* · bölünen *dividend* · bölüm *quotient* · artırma / azaltma *increment / decrement* · önek *prefix* · sonek *postfix* · karşılaştırma işleci *comparison operator* · mantıksal işleç *logical operator* · işleç önceliği *operator precedence* · ifade ağacı *expression tree* · kısa devre değerlendirme *short-circuit evaluation* · dizgi birleştirme *string concatenation* · istem *prompt* · tampon *buffer* · satır sonu *newline* · girdi doğrulama *input validation* · biçimli çıktı *formatted output* · biçim dizgisi *format string* · biçim belirteci *format specifier* · yerel ayar *locale* · ondalık ayırıcı *decimal separator* · takas *swap* · geçici değişken *temporary variable* · kapsam *scope* · boş deyim *empty statement*

6.10 Kendini Dene

1. `int x = 1_000_000; int y = x * 3000; long z = x * 3000L;` satırlarından sonra `System.out.println(y + " " + z);` ne yazar? y neden yanlıştır?

   Cevap

   Çıktı `-1294967296 3000000000`'dır. *1.000.000 · 3.000 = 3.000.000.000*, int sınırı 2.147.483.647'yi aşar. İki işlenen de int olduğu için çarpma int ile yapılır ve taşar: *3.000.000.000 − 2^{32} = −1.294.967.296*. z'de 3000L bir long sabitidir; x long'a yükselir ve çarpma long ile yapılır.

2. `(byte) 300`, `(int) -7.9`, `Math.round(-7.5)` ve `(int) Math.round(7.5)` ifadelerinin değerleri nedir?

   Cevap

   Sırasıyla 44, −7, −7 ve 8. 300'ün alt 8 biti tutulur: *300 − 256 = 44* (işaret biti 0 olduğu için pozitiftir). `(int) -7.9` sıfıra doğru keser: −7. Math.round yarımları pozitif yöne yuvarlar: −7.5 → −7, 7.5 → 8; ikincisi long döndürdüğü için int'e çevrilmiştir.

3. `int a = 7, b = 8; double o = (a + b) / 2;` satırından sonra o kaçtır? Doğru ortalamayı veren iki yazılış nedir?

   Cevap

   o = 7.0'dır. `a + b` 15'tir ve int'tir; *15 / 2* tam sayı bölmesiyle 7 olur, sonra double'a genişleyerek 7.0 yazılır. Doğru yazılışlar `(a + b) / 2.0` ve `(double) (a + b) / 2`'dir; ikisi de 7.5 verir. `(double) ((a + b) / 2)` ise yine 7.0 verir, çünkü dönüşüm bölmeden sonra yapılır.

4. `int i = 10; int a = i-- - 3; int b = --i * 2;` satırlarından sonra i, a ve b kaçtır?

   Cevap

   i = 8, a = 7, b = 16. Sonek `i--` eski değeri (10) üretir: *a = 10 − 3 = 7*; ardından i 9 olur. Önek `--i` önce i'yi 8 yapar, sonra yeni değeri kullanır: *b = 8 · 2 = 16*.

5. `short s = 10; s *= 2.6;` derlenir mi? Derlenirse s kaç olur? `s = s * 2.6;` için ne söylenebi-

lir?

Cevap

Derlenir ve s = 26 olur. Bileşik atama `s = (short) (s * 2.6)` anlamına gelir: *10 · 2,6 = 26,0* (double) hesaplanır ve short'a daraltılır. `s = s * 2.6;` ise derlenmez; sağ taraf double'dır ve double'dan short'a örtük dönüşüm yoktur ("possible lossy conversion from double to short").

6. `5 + 4 % 3 * 2 - 8 / 3` ve `!(3 > 2) || 4 >= 4 && 1 != 1` ifadelerinin değerleri nedir? Hesap hangi sırayla yapılır?

Cevap

İlki 5'tir. *, / ve % aynı düzeydedir ve soldan sağa işlenir: *4 % 3 = 1*, *1 · 2 = 2*, *8 / 3 = 2*; sonra *5 + 2 − 2 = 5*. İkincisi false'tur. Parantez içi *3 > 2* true'dur ve ! ile false olur. `&&`, `||` 'den önce gelir: *4 >= 4* true, *1 != 1* false, *true && false = false*. Sonuç *false || false = false*.

7. `System.out.println(1 + 2 + "3" + 4 + 5);` ve `System.out.println('a' + 2 * 3 + "");` ne yazar?

Cevap

İlki "3345"tir: soldan sağa önce iki int toplanır (3), sonra "3" dizgisiyle birleşir ("33"); bundan sonra her işlem birleştirmedir: "334", "3345". İkincisi "103"tür: *2 · 3 = 6* önce hesaplanır; 'a' int'e yükselir ve *97 + 6 = 103* bulunur; en son boş dizgiyle birleşir.

8. Kullanıcı ilk satıra `3 4.5`, ikinci satıra `Ali Veli` yazıyor (Locale.US). Program sırasıyla `nextInt()`, `nextDouble()`, `nextLine()` ve `nextLine()` çağırıyor. Dört çağrı ne döndürür?

Cevap

3, 4.5, "" (boş dizgi) ve "Ali Veli". nextInt ve nextDouble aynı satırdaki iki sayıyı okur; ilk satırın sonundaki satır sonu tamponda kalır. İlk nextLine bu satır sonuna kadar okur ve boş dizgi döndürür. İkinci nextLine ikinci satırı okur. Türkçe yerel ayarda ise nextDouble "4.5" girdisinde InputMismatchException fırlatırdı.

9. `System.out.printf("%-5s|%8.3f|%,d%n", "Kalem", 12.34567, 1500000);` satırı Locale.US aya-

rında ne yazar? Her alan kaç karakter tutar?

Cevap

`Kalem|  12.346|1,500,000`. “Kalem” tam 5 harftir; `%-5s` boşluk eklemez. `%8.3f` 12.34567'yi üç basamağa yuvarlar (12.346, 6 karakter) ve 8 karaktere tamamlamak için önüne 2 boşluk koyar. `%,d` binlik ayırıcı ekler; Türkçe ayarda bu alan 1.500.000, ikinci alan 12,346 olurdu.

6.11 İleri okuma

- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), 1.2 Built-in Types of Data: Java'nın ilkel tiplerini, tip dönüşümünü ve kayan nokta sorunlarını kısa programlarla anlatır; kitabın sitesinde (introcs.cs.princeton.edu) bölümün özeti ve alıştırmaları vardır.
- Oracle, *The Java Tutorials*, Language Basics: Variables ve Operators: ilkel tipleri, sabitlerin yazılışını, işleç öncelik tablosunu ve bileşik atamayı resmî kaynaktan, örneklerle verir.
- Y. D. Liang, *Introduction to Java Programming and Data Structures* (13. baskı, Pearson, 2024), Bölüm 2 Elementary Programming ve Bölüm 4: Scanner ile girdi, sayısal tipler, printf ve Math sınıfı üzerine çok sayıda çözülmüş örnek içerir.
- Oracle, *Java Language Specification*, Java SE 21: Bölüm 5 (Conversions and Contexts) ve Bölüm 15 (Expressions): bu notta anlatılan dönüşüm, yükseltme ve değerlendirme sırası kurallarının kesin tanımlarıdır; ileri düzeyde merak edilen bir ayrıntının son sözü buradadır.

6.12 Kaynaklar

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11. baskı, Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13. baskı, Pearson, 2024.
3. Oracle, *The Java Tutorials*: Variables, Operators.
4. Oracle, *Java Language Specification, Java SE 21 Edition*: Bölüm 4, Types, Values, and Variables; Bölüm 15, Expressions.
5. Oracle, *Java SE 21 API*: java.util.Scanner, java.lang.Math, java.util.Formatter (biçim belirteçleri).
6. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 1.2 Built-in Types of Data.

KISIM III · BÖLÜM 7

Koşullu İfadeler

Şimdiye kadar yazılan Java programları her çalıştırmada aynı satırları aynı sırayla yürütür. Gerçek programlar ise duruma göre davranır. Not 60'tan küçükse “Kaldı”, değilse “Geçti” yazılır. Bölen sıfırsa bölme yapılmaz. Kullanıcı 13. ay girerse uyarı verilir. Bu not, programın bir koşula bakarak farklı yollardan birini seçmesini sağlayan yapıları anlatır. Önce sıralı yürütme ve koşul kavramı ele alınır. Ardından `if`, `if-else`, `else-if` zinciri ve iç içe kararlar gelir. Mantıksal işleçler, kısa devre değerlendirme, koşul işleci ve dizgi karşılaştırma kararları daha kısa ve güvenli yazmayı sağlar. `switch` yapısı tek bir değerin ayrık seçeneklerine göre seçim yapar. Son kesimdeki örnekler bütün bu yapıları gerçek problemlerde bir araya getirir. Her program JDK 21 ile derlenip çalıştırılmıştır; verilen çıktılar programların gerçek çıktılarıdır.

Bu bölümün kodu: `github.com/sercankulcu/computer-programming-java/Bolum07/src`

ÖĞRENME ÇIKTILARI

Akış
Sıralı, karar ve döngü yapılarını ayırt etmek; bir programın satırlarının hangi sırayla ve hangi koşulda çalıştığını izleme tablosuyla göstermek.

Koşul
Karşılaştırma ve mantıksal işleçlerle boolean değer üreten koşullar yazmak; kısa devre değerlendirmeyi, De Morgan kurallarını ve işleç önceliğini doğru kullanmak.

Karar
`if`, `if-else`, `else-if` zinciri ve iç içe kararları doğru sırada kurmak; bağımsız if ile zincir arasındaki farkı, sarkan else sorununu ve süslü parantezin rolünü açıklamak.

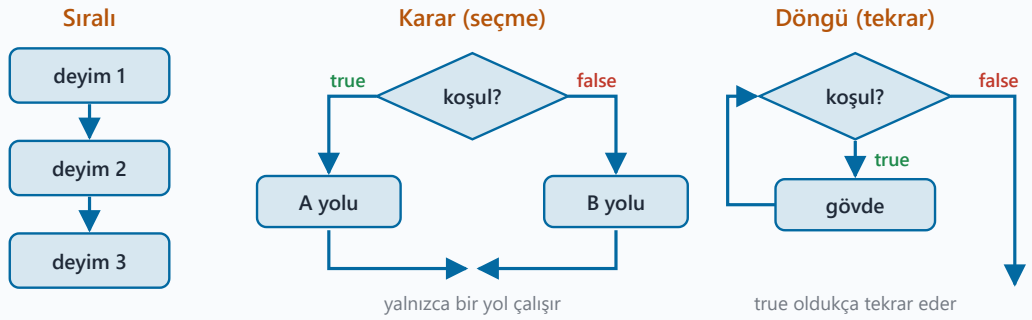
switch
Tek bir değerin ayrık seçeneklerinde `switch` ifadesi ve deyimi kullanmak; `default`, `yield`, `break` ve aşağı düşme davranışını açıklamak; if-else ile switch arasında seçim yapmak.

7.1 Akış yapıları

Bir program, bilgisayara verilen deyimlerin listesidir. Deyim (*statement*), programın tek bir işi yapan en küçük tam parçasıdır: bir değişken bildirmek, bir değer atamak, ekrana bir satır yazmak. Java'da her basit deyim noktalı virgülle biter. Bilgisayarın bu deyimleri hangi sırayla çalıştıracağına **akış** (*control flow*) denir. Akışı belirleyen dil yapılarının genel adı da **kontrol yapısıdır** (*control structure*).

7.1.1 Üç temel akış

Ne kadar büyük olursa olsun her algoritma yalnızca üç akış biçimiyle yazılabilir. Şekil 7.1 bu üç biçimi yan yana gösterir.



Şekil 7.1: Üç temel akış. Solda sıralı yapı: deyimler yazıldıkları sırayla birer kez çalışır. Ortada karar yapısı: koşulun değerine göre iki yoldan yalnızca biri çalışır. Sağda döngü yapısı: koşul doğru kaldıkça gövde tekrar eder.

> **Tanım 7.1**
>
> **Sıralı yapı** (*sequence structure*), deyimlerin yazıldıkları sırayla ve her birinin tam bir kez çalıştığı akıştır. **Karar yapısı** (*selection structure*), bir koşulun değerine göre birden çok yoldan birinin seçildiği akıştır; koşullu yapı adı da kullanılır. **Döngü yapısı** (*repetition structure*), bir kod bloğunun bir koşul sağlandığı sürece tekrar tekrar çalıştığı akıştır.

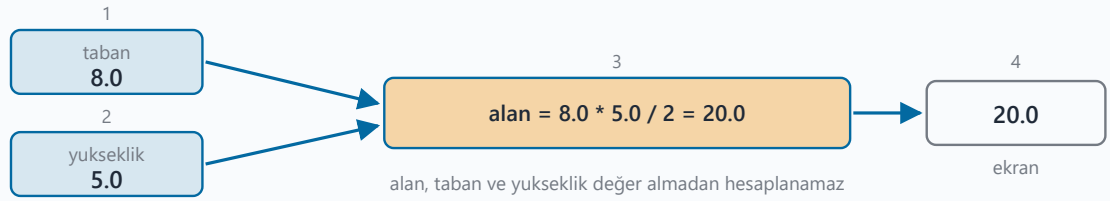
Bu üç yapının her birinin tek bir girişi ve tek bir çıkışı vardır. Bu yüzden yapılar birbirinin içine yerleştirilebilir. Bir kararın dalında sıralı deyimler bulunabilir, bir döngünün gövdesinde bir karar bulunabilir. Bu not karar yapısını ayrıntılı olarak ele alır. Sıralı yapı kararın yapı taşıdır; ilk önce o incelenir.

7.1.2 Sıralı yapı

Aşağıdaki dört satır bir üçgenin alanını hesaplar. Taban 8, yükseklik 5 birimdir.

```
double taban = 8;
double yukseklik = 5;
double alan = taban * yukseklik / 2;
System.out.println(alan);
```

Satırlar yukarıdan aşağıya çalışır. İlk satır `taban` değişkenini oluşturur ve içine 8 koyar. Değişkenin tipi `double` olduğu için tam sayı 8 kendiliğinden 8.0'a dönüşür. Küçük bir tipten büyük bir tipe yapılan bu dönüşüme **genişletme** (*widening conversion*) denir. İkinci satır `yukseklik` için aynı işi yapar. Üçüncü satır önceki iki değişkenin değerini kullanır ve alanı hesaplar. Dördüncü satır sonucu ekrana yazar.



Şekil 7.2: Sıralı işlemler. 1 ve 2 numaralı satırlar taban ve yukseklik değişkenlerine değer verir. 3 numaralı satır bu iki değeri kullanarak alanı 20.0 olarak hesaplar. 4 numaralı satır sonucu ekrana yazar.

Şekil 7.2 satırlar arasındaki bağımlılığı oklarla gösterir. Üçüncü satır, birinci ve ikinci satırın ürettiği değerlere muhtaçtır. Her deyim, kendinden önceki deyimlerin bıraktığı durumu kullanabilir. Sıralı yapının özü budur.

7.1.3 Sıralı işlemleri adım adım izlemek

Bir programı elle yürütmenin en güvenilir yolu izleme tablosudur. **İzleme tablosunda** (*trace table*) her satır bir adımı, her sütun bir değişkeni gösterir. Bir hücre, o adım bittiğinde değişkenin değerini taşır. Henüz var olmayan değişken için "—" yazılır.

Adım	Çalışan satır	taban	yukseklik	alan	Ekran
1	`double taban = 8;`	8.0	—	—	
2	`double yukseklik = 5;`	8.0	5.0	—	
3a	`taban * yukseklik`	8.0	5.0	— (ara sonuç 40.0)	
3b	`40.0 / 2` → `alan`	8.0	5.0	20.0	
4	`System.out.println(alan);`	8.0	5.0	20.0	20.0

Tablo 7.1: Alan hesabının izleme tablosu. Her satır ilgili deyim bittiğindeki durumu gösterir.

Üçüncü satır iki adımda incelenmiştir. `*` ve `/` aynı önceliktedir ve soldan sağa hesaplanır. Önce *8.0 × 5.0 = 40.0* bulunur, sonra *40.0 / 2 = 20.0* olur. Ancak bu iki adım bittiğinde `alan` kutusuna değer yazılır. İfadenin ortasında ara sonuç yalnızca işlemcinin içinde bulunur; hiçbir değişkende saklanmaz.

Program çalıştırıldığında şu çıktıyı verir:

```
20.0
```

Sıralı yapıda satırların yeri sonucu belirler. Üçüncü satır en başa taşınırsa `taban` ve `yukseklik` o noktada henüz yoktur. Java bu hatayı program çalışmadan, derleme sırasında yakalar:

```java
public class SiraHatasi {
    public static void main(String[] args) {
        double alan = taban * yukseklik / 2;
        double taban = 8;
        double yukseklik = 5;
        System.out.println(alan);
    }
}
```

```
SiraHatasi.java:3: error: cannot find symbol
        double alan = taban * yukseklik / 2;
                      ^
  symbol:   variable taban
  location: class SiraHatasi
SiraHatasi.java:3: error: cannot find symbol
        double alan = taban * yukseklik / 2;
                              ^
  symbol:   variable yukseklik
  location: class SiraHatasi
2 errors
```

İleti "sembol bulunamadı" anlamına gelir. Derleyici 3. satırda `taban` adında bir değişken arar ve bulamaz. Aynı ileti `yukseklik` için de verilir. Java bir değişkenin, bildirildiği satırdan önce kullanılmasına izin vermez.

Sık yapılan hata

Matematikteki gibi "alan = taban × yükseklik / 2" formülünü bir tanım sanmak. Java'da `=` bir eşitlik bildirimi değil, bir atamadır: sağdaki ifade o anda hesaplanır ve sonuç soldaki kutuya yazılır. Atamadan sonra `taban` değişse bile `alan` kendiliğinden güncellenmez. Yeni alan için atamanın yeniden çalışması gerekir.

7.2 Koşul ve if

Karar yapısının merkezinde bir soru vardır: “Not 60 ya da daha büyük mü?”, “Sayı negatif mi?”, “Bölen sıfır mı?” Programın hangi yoldan gideceği bu sorunun cevabına bağlıdır. Java'da bu soru bir koşul olarak yazılır.

7.2.1 Koşul: boolean değer üreten ifade

> **Tanım 7.2**
>
> **Koşul** (*condition*), değeri `true` (doğru) ya da `false` (yanlış) olan ifadedir. Bu iki değeri taşıyan Java tipine **boolean** (mantıksal tip) denir. Bir `if` yapısının parantezi içinde yalnızca boolean değer üreten bir ifade bulunabilir.

Koşulların çoğu iki değeri karşılaştırarak kurulur. Java'nın altı **karşılaştırma işleci** (*comparison operator*) Tablo 7.2'de verilmiştir. Her biri iki sayıyı alır ve bir boolean değer üretir.

İşleç	Anlamı	Örnek	Değeri	Tersi
`==`	eşit	`x == 5`	false	`!=`
`!=`	eşit değil	`x != 5`	true	`==`
`<`	küçük	`x < 7`	false	`>=`
`<=`	küçük ya da eşit	`x <= 7`	true	`>`
`>`	büyük	`x > 5`	true	`<=`
`>=`	büyük ya da eşit	`x >= 9`	false	`<`

Tablo 7.2: Karşılaştırma işleçleri. Örnek sütununda x = 7'dir.

Tablonun son sütunu her işlecin tersini verir. `>`'ün tersi `<` değil, `<=`'dir. Çünkü “7'den büyük değil” demek “7 ya da daha küçük” demektir; 7'nin kendisi de bu kümeye girer. Bu ayrıntı, koşulları tersine çevirirken sık karşılaşılan bir yanılgının kaynağıdır.

Eşitlik sınaması iki eşittir işaretiyle (`==`) yapılır. Tek eşittir (`=`) atamadır. İkisini karıştırmak yeni başlayanların en sık yaptığı hatalardan biridir. Neyse ki Java bu karışıklığı derleme sırasında yakalar:

```
public class AtamaKosul {
    public static void main(String[] args) {
        int x = 7;
        if (x = 5) {
            System.out.println("beş");
        }
    }
}
```

```
AtamaKosul.java:4: error: incompatible types: int cannot be converted to boolean
        if (x = 5) {
              ^
1 error
```

`x = 5` bir atamadır. Atamanın değeri atanan sayıdır, yani 5. Bu değerin tipi `int`'tir. `if` ise boolean bekler. Derleyici bu yüzden "int, boolean'a dönüştürülemez" iletisini verir. C gibi bazı dillerde sıfır olmayan her sayı "doğru" sayılır ve bu satır sessizce derlenir. Java'da sayı ile boolean ayrı tiplerdir; bu katılık bir hata sınıfını tümüyle ortadan kaldırır.

Matematikteki *10 < x < 20* yazımı da Java'da çalışmaz. İfade soldan sağa okunur: önce `10 < sayi` hesaplanır ve bir boolean değer çıkar. Ardından bu boolean değer 20 ile karşılaştırılmaya çalışılır. Boolean bir değer için "küçüktür" anlamsızdır:

```
public class ZincirKarsilastirma {
    public static void main(String[] args) {
        int sayi = 15;
        if (10 < sayi < 20) {
            System.out.println("arada");
        }
    }
}
```

```
ZincirKarsilastirma.java:4: error: bad operand types for binary operator '<'
        if (10 < sayi < 20) {
                      ^
  first type:  boolean
  second type: int
1 error
```

İleti, `<` işlecinin iki yanında bir boolean ile bir int bulunduğunu söyler. Doğru yazım iki karşılaştırmayı mantıksal VE ile birleştirir: `sayi > 10 && sayi < 20`. Mantıksal işleçler Tablo 7.9'da ayrıntılı olarak ele alınır.

7.2.2 Tek yollu if

En basit karar, koşul doğruysa bir işi yapmak, yanlışsa hiçbir şey yapmamaktır. Örneğin negatif bir sayıyı pozitife çevirmek için yalnızca sayı negatifken bir işlem gerekir:

```
if (sayi < 0) {
    sayi = -sayi;
}
```

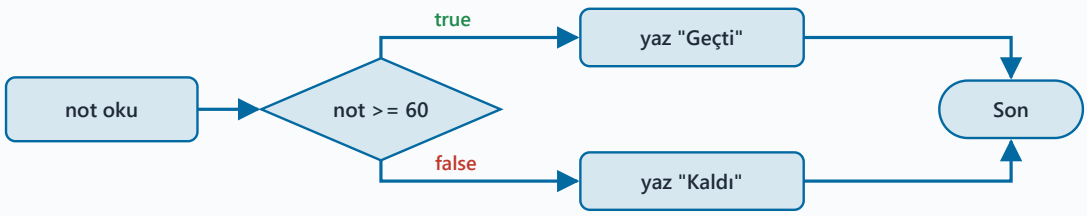
Yürütme `if` satırına geldiğinde önce parantezin içi hesaplanır. Sonuç `true` ise süslü parantezlerin arasındaki deyimler çalışır. Sonuç `false` ise bu deyimler atlanır ve yürütme kapanan süslü parantezden sonraki satırdan devam eder. Her iki durumda da program bir sonraki satıra ulaşır; yalnızca aradaki iş yapılır ya da yapılmaz. Bu kalıp mutlak değer örneğinde ayrıntılı olarak izlenir.

7.2.3 if-else yapısı

Çoğu kararda iki durumun her biri için ayrı bir iş vardır. Not 60 ya da daha büyükse “Geçti”, değilse “Kaldı” yazılmalıdır. Bunun için `if`'e bir `else` dalı eklenir.

> **Tanım 7.3**
>
> **if-else yapısı** (*if-else statement*) iki daldan oluşur. Koşul `true` ise `if`'in gövdesi, `false` ise `else`'in gövdesi çalışır. Her çalıştırmada dallardan tam olarak biri çalışır; ikisi birden ya da hiçbiri çalışmaz. **Dal** (*branch*), karar sonrasında izlenebilecek yollardan her biridir.

```
if (not >= 60) {
    System.out.println("Geçti");
} else {
    System.out.println("Kaldı");
}
```



Şekil 7.3: if-else yapısının akış diyagramı. Not okunduktan sonra `not >= 60` koşulu sorulur. true dalı “Geçti”, false dalı “Kaldı” yazar. İki dal Son'da birleşir.

Şekil 7.3'te karar kutusundan iki ok çıkar. Biri true, öteki false diye etiketlenmiştir. Bir not için ikisinden yalnızca biri izlenebilir. Okların sonunda iki yol yeniden birleşir; program kararın ardından aynı noktadan devam eder.

Koşuldaki sınır değeri dikkat ister. `not >= 60` yazıldığında 60 "Geçti" tarafına düşer. `not > 60` yazılsaydı tam 60 alan öğrenci kalırdı. Bir kararın en sık hatası sınırın yanlış tarafa konmasıdır. Bu yüzden bir karar her zaman sınır değeriyle ve sınırın iki komşusuyla (59 ve 61 gibi) denenir. Her olası değer yalnızca bir dala gider: 0'dan 59'a kadar olan notlar else dalına, 60 ve üstü if dalına.

7.2.4 if-else'i adım adım izlemek

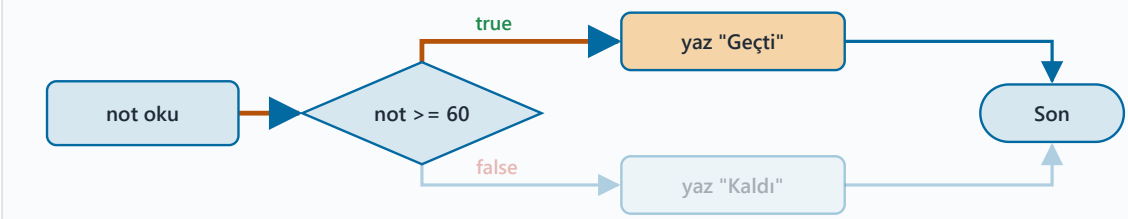
Aşağıdaki program notu klavyeden okur. `Scanner` sınıfı `java.util` paketindedir; kullanılabilmesi için dosyanın başında `import java.util.Scanner;` satırı bulunur. Notta bundan sonra kod parçaları kısa tutulmak için yalnızca `main` 'in gövdesi verilir. Tam program ise ayrıca gösterilir.

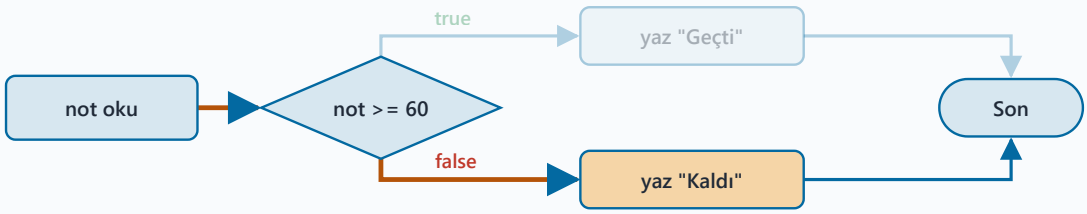
```
Scanner klavye = new Scanner(System.in);
int not = klavye.nextInt();
if (not >= 60) {
    System.out.println("Geçti");
} else {
    System.out.println("Kaldı");
}
```

Program üç kez çalıştırılsın: kullanıcı sırasıyla 75, 60 ve 45 girsin. Tablo 7.3 her çalıştırmanın adımlarını verir.

Girdi	Satır 2: not	Satır 3: koşul	Çalışan dal	Atlanan dal	Çıktı
75	75	*75 >= 60* → true	satır 4	satır 6	Geçti
60	60	*60 >= 60* → true	satır 4	satır 6	Geçti
45	45	*45 >= 60* → false	satır 6	satır 4	Kaldı

Tablo 7.3: Geçti/Kaldı programının üç çalıştırması. Satır numaraları yukarıdaki kodun satırlarıdır.





Şekil 7.4: İki çalıştırmada izlenen yol. (a) not = 75: koşul true, "Geçti" kutusu çalışır, "Kaldı" kutusu soluk kalır. (b) not = 45: koşul false, yalnızca "Kaldı" kutusu çalışır.

İkinci çalıştırma sınır değerini dener. *60 >= 60* doğrudur, çünkü `>=` eşitliği de kapsar. Şekil 7.4 iki çalıştırmada izlenen yolu renkle gösterir. Her çalıştırmada dallardan biri hiç çalışmaz. Tam program ve 75, 60, 45 girdileriyle elde edilen çıktılar şöyledir:

```
import java.util.Scanner;

public class GectiKaldi {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        int not = klavye.nextInt();
        if (not >= 60) {
            System.out.println("Geçti");
        } else {
            System.out.println("Kaldı");
        }
    }
}
```

```
Geçti
```

```
Geçti
```

```
Kaldı
```

7.2.5 Blok, süslü parantez ve boş deyim

Süslü parantezlerin arasındaki deyim grubuna **blok** (*block*) denir. Java'nın kuralı şudur: `if`'in ve `else`'in gövdesi tek bir deyimdir. Bir blok, içinde kaç deyim olursa olsun tek bir deyim sayılır. Gövde tek bir basit deyimse süslü parantez yazılmayabilir:

```
if (not >= 60)
    System.out.println("Geçti");
else
    System.out.println("Kaldı");
```

Bu yazım kısadır ama iki tuzağı vardır. Birincisi, gövdeye ikinci bir satır eklendiğinde ortaya çıkar. Girinti, eklenen satırın da `if`'e ait olduğunu düşündürür. Oysa derleyici girintiye bakmaz. Süslü parantez yoksa `if` yalnızca hemen ardından gelen tek deyimi denetler; ikinci satır her durumda çalışır.

```
public class SusluParantezsiz {
    public static void main(String[] args) {
        int x = 2;
        if (x > 5)
            System.out.println("A");
            System.out.println("B");
        System.out.println("C");
    }
}
```

```
B
C
```

x = 2 için *2 > 5* yanlıştır. Yalnızca `println("A")` atlanır. "B" satırı girintili yazılmış olsa da `if`'in dışındadır ve çalışır. Programcının niyeti A ile B'yi birlikte denetlemekse iki satır süslü parantez içine alınmalıdır.

İkinci tuzak, `if` satırının sonuna yanlışlıkla konan noktalı virgüldür. Tek başına duran noktalı virgül de Java'da bir deyimdir: hiçbir iş yapmayan **boş deyim** (*empty statement*). `if (x > 0);` yazıldığında `if`'in gövdesi bu boş deyim olur. Alttaki satır artık koşula bağlı değildir.

```
public class BosDeyim {
    public static void main(String[] args) {
        int x = -3;
        if (x > 0);
            System.out.println("Pozitif");
        System.out.println("Bitti");
    }
}
```

```
Pozitif
Bitti
```

x = −3 olmasına karşın “Pozitif” yazılır. *−3 > 0* yanlıştır ve boş deyim atlanır; ama atlanacak başka bir şey yoktur. Derleyici bu satırı hata saymaz, çünkü kural dışı bir şey yoktur. Hata yalnızca programın çıktısında görülür.

Sık yapılan hata

Gövdesi tek satır olan bir `if` 'e sonradan ikinci bir satır eklemek ve süslü parantezi unutmak. Girinti doğru göründüğü için hata gözden kaçar. Bu yüzden pek çok kodlama kılavuzu gövde tek satır olsa bile süslü parantezi zorunlu tutar. Süslü parantez ayrıca `if (...);` hatasını da görünür kılar: `if (x > 0); {` satırı okunurken göze batar.

7.3 Çok yollu karar

İki seçenekli kararlar if-else ile kurulur. Seçenek sayısı ikiden fazla olduğunda ne olur? Bir notu harf notuna çevirmek dört seçenek ister: 90 ve üstü A, 80–89 B, 70–79 C, 70'in altı F. Bu tür bir karara **çok yollu karar** (*multi-way decision*) denir.

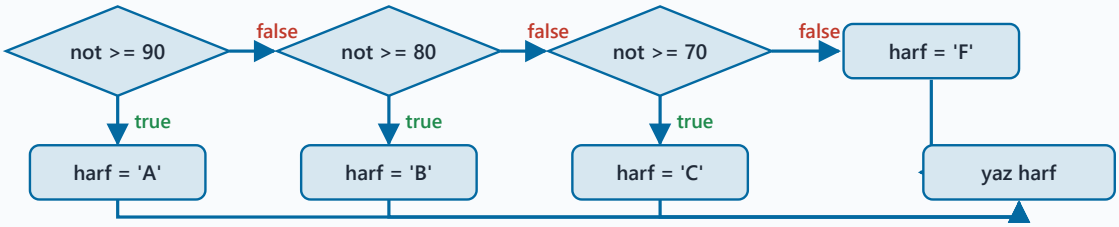
7.3.1 else-if zinciri

Tanım 7.4

else-if zinciri (*else-if chain*), her `else` 'in gövdesine yeni bir `if` konarak kurulan çok yollu karardır. Koşullar yukarıdan aşağıya sırayla denenir. İlk `true` çıkan koşulun dalı çalışır; zincirin geri kalanı denetlenmeden atlanır. Hiçbir koşul tutmazsa, varsa son `else` çalışır.

```
if (not >= 90) {
    harf = 'A';
} else if (not >= 80) {
    harf = 'B';
} else if (not >= 70) {
    harf = 'C';
} else {
    harf = 'F';
}
```

Zincirin aslında iç içe if-else'lerden oluştuğunu görmek kuralı anlamayı kolaylaştırır. İlk `else` 'in gövdesi ikinci `if` 'tir; onun `else` 'inin gövdesi üçüncü `if` 'tir. Kod her düzeyde biraz daha sağa kayacağı için Java programcıları `else if` 'i aynı satıra yazar ve bütün dalları aynı hizada tutar. Dil açısından `else if` diye ayrı bir sözcük yoktur; yalnızca bir yazım alışkanlığıdır.



Şekil 7.5: else-if zincirinin akış diyagramı. 90, 80 ve 70 eşikleri soldan sağa sırayla denenir. Bir koşul true olduğunda o dalın ataması yapılır ve yürütme doğrudan "yaz harf" kutusuna gider. Üç koşul da false ise harf = 'F' olur.

Şekil 7.5 zincirin biçimini gösterir. Her karar kutusunun false oku bir sonraki karara gider. true okları ise aşağıya, atama kutularına iner. Bütün yollar sonunda "yaz harf" kutusunda birleşir. Bir çalıştırmada en fazla bir atama kutusu çalışır.

7.3.2 Zinciri adım adım izlemek

Harf notu programının kısa yazımı aşağıdadır. Dallar tek deyimli olduğu için süslü parantez kullanılmamıştır; koşullar ve atamalar hizalanarak okunurluk korunmuştur.

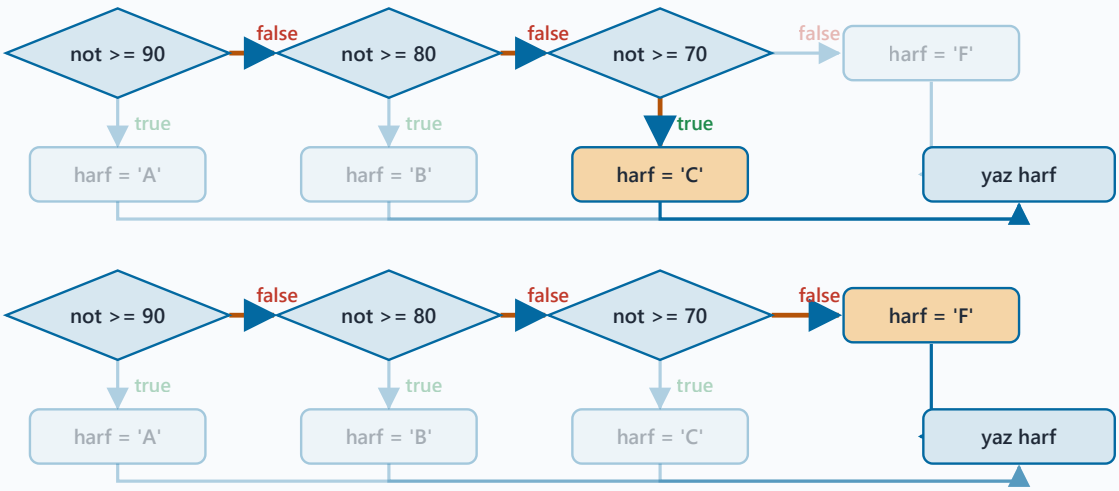
```
int not = klavye.nextInt();
char harf;
if (not >= 90)      harf = 'A';
else if (not >= 80) harf = 'B';
else if (not >= 70) harf = 'C';
else                harf = 'F';
System.out.println(harf);
```

İkinci satırdaki `char harf;` değişkeni bildirir ama değer vermez. Değeri, zincirin dallarından biri atar. Program 75, 95 ve 50 için yürütülsün.

not	`not >= 90`	`not >= 80`	`not >= 70`	Çalışan dal	Denenen koşul	Çıktı
75	false	false	true	`harf = 'C'`	3	C
95	true	atlandı	atlandı	`harf = 'A'`	1	A
50	false	false	false	`harf = 'F'` (else)	3	F

Tablo 7.4: else-if zincirinin üç çalıştırması. "atlandı", koşulun hiç hesaplanmadığını gösterir.

75 için ilk iki koşul yanlıştır: *75 >= 90* ve *75 >= 80*. Üçüncü koşul *75 >= 70* doğrudur; `harf = 'C'` yapılır. Zincirde başka koşul kalmasa bile kural aynıdır: seçilen daldan sonra yürütme doğrudan zincirin sonuna atlar. 95 için ilk koşul doğrudur; öteki iki koşul hiç hesaplanmaz. 50 için üç koşul da yanlıştır ve `else` çalışır.



Şekil 7.6: Zincirde izlenen yol. (a) not = 75: iki false okundan sonra üçüncü koşul true olur ve harf = 'C' kutusu çalışır. (b) not = 50: üç koşul da false; yol en sağdaki harf = 'F' kutusuna ulaşır.

Zincirin sınırları da denenmelidir. Program 90, 89, 80, 70 ve 69 için de çalıştırıldı. Sırasıyla A, B, B, C ve F yazdı. 90 ile 89 arasındaki, 70 ile 69 arasındaki tek birimlik fark harfi değiştirir; her eşik kendi değerini kapsar.

7.3.3 Eşiklerin sırası

Zincirdeki koşulların her biri tek başına bir alt sınır koyar: “en az 80”, “en az 70”. Üst sınır yazılmaz. B dalının koşulu `not >= 80 && not < 90` olmak zorunda değildir. Çünkü yürütme B'nin koşuluna ancak A'nın koşulu yanlışsa ulaşır; o noktada notun 90'dan küçük olduğu zaten bellidir. Zincirde her koşul, önceki koşulların yanlış olduğu bilgisini kendiliğinden taşır.

Bu kolaylık bir sorumluluk getirir: eşikler büyükten küçüğe yazılmalıdır. Sıra ters çevrildiğinde ne olduğunu aşağıdaki program gösterir.

```
import java.util.Scanner;

public class TersEsik {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        int not = klavye.nextInt();
        if (not >= 60) System.out.println("D");
        else if (not >= 70) System.out.println("C");
        else if (not >= 80) System.out.println("B");
        else if (not >= 90) System.out.println("A");
        else System.out.println("F");
    }
}
```

```
D
```

85 için ilk koşul *85 >= 60* doğrudur ve D yazılır. Zincirin geri kalanı atlanır. Program 95 için de D yazar. 60 ve üstündeki her not D alır. C, B ve A dallarına hiçbir girdiyle ulaşılamaz. Böyle bir kod derleme hatası vermez; hata ancak doğru çıktı bilinen bir girdiyle denendiğinde görülür.

> **Sık yapılan hata**
>
> Alt sınırla kurulan bir zincirde eşikleri küçükten büyüğe dizmek. Genel kural şudur: daha dar koşul daha önce sorulur. “En az 90” koşulunu sağlayan her not “en az 70” koşulunu da sağlar. Geniş koşul önce sorulursa dar koşula sıra gelmez.

7.3.4 Bağımsız if ile else-if zinciri

Aynı üç koşul, `else` sözcükleri olmadan alt alta yazılırsa bambaşka bir program ortaya çıkar. Buna **bağımsız if'ler** denir: hiçbiri bir öncekinin `else` dalında değildir ve her biri ötekilerden bağımsız olarak denetlenir.

```
int not = 95;
if (not >= 90) System.out.println("A");
if (not >= 80) System.out.println("B");
if (not >= 70) System.out.println("C");
```

```
A
B
C
```

not = 95 için üç koşul da doğrudur ve üç satır yazılır. İlk `if` 'in A yazması ikinci `if` 'i etkilemez; ikinci `if` yine denenir. Bir öğrenciye tek harf verilmesi beklenirken üç harf çıkar. Aynı koşullar `else if` ile bağlandığında ise yalnızca ilk doğru dal çalışır:

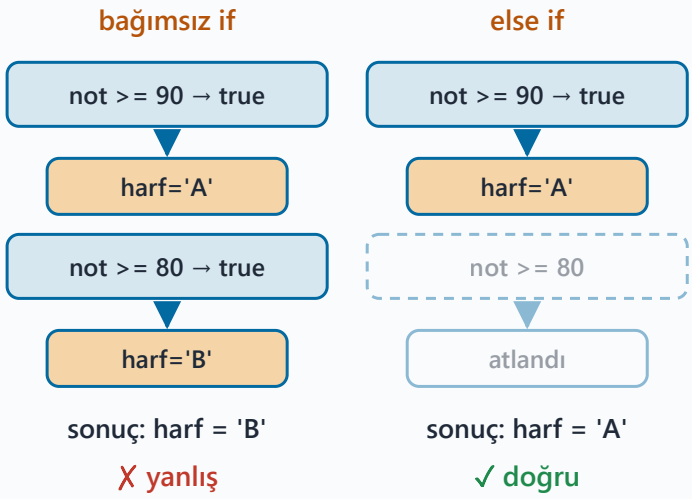
```
int not = 95;
if (not >= 90) System.out.println("A");
else if (not >= 80) System.out.println("B");
else if (not >= 70) System.out.println("C");
```

```
A
```

Koşul	Bağımsız if	else-if zinciri
`not >= 90`	denenir: true → A	denenir: true → A
`not >= 80`	denenir: true → B	denenmez
`not >= 70`	denenir: true → C	denenmez
Çıktı	3 satır: A, B, C	1 satır: A

Tablo 7.5: not = 95 için iki yazımın karşılaştırması.

Dallar ekrana yazmak yerine bir değişkene atama yaptığında hata daha sinsi olur. Şekil 7.7'de her dal `harf`'e bir değer atar. Bağımsız yazımda sonraki atama öncekinin üzerine yazar. Ekranda tek bir harf görüldüğü için çıktı "doğru biçimde" görünür, ama harf yanlıştır.



Şekil 7.7: not = 95 için iki yazım. Solda bağımsız if'ler: iki koşul da true, ikinci atama birincinin üzerine yazar ve harf = 'B' kalır. Sağda else-if: ilk koşul true olunca ikinci koşul atlanır ve harf = 'A' kalır.

```
public class BagimsizHarf {
    public static void main(String[] args) {
        int not = 95;
        char harf = 'F';
        if (not >= 90) harf = 'A';
        if (not >= 80) harf = 'B';
        System.out.println(harf);
    }
}
```

```
B
```

Bağımsız if her zaman yanlış değildir. Birbirinden bağımsız soruların her biri ayrı ayrı sorulacaksa doğru yapı odur. Örneğin bir başvuru formunda “ad boş mu?”, “e-posta geçersiz mi?”, “yaş 18'den küçük mü?” soruları için ayrı ayrı uyarı yazılır. Birinin doğru olması ötekilerin sorulmasını engellememelidir. Seçim kuralı şöyledir: seçeneklerden yalnızca biri geçerliyse `else if`, her biri ayrı ayrı geçerli olabilirse bağımsız `if`.

7.3.5 Son else ve değer almamış değişken

Zincirin sonundaki `else` hiçbir koşulun tutmadığı durumu yakalar. Bu dal zorunlu değildir. Ama harf notu programında silinirse Java programı derlemez:

```
public class AtanmamisHarf {
    public static void main(String[] args) {
        int not = 75;
        char harf;
        if (not >= 90)      harf = 'A';
        else if (not >= 80) harf = 'B';
        else if (not >= 70) harf = 'C';
        System.out.println(harf);
    }
}
```

```
AtanmamisHarf.java:8: error: variable harf might not have been initialized
        System.out.println(harf);
                           ^
1 error
```

İleti “harf değişkenine değer verilmemiş olabilir” anlamına gelir. Derleyici bütün yolları inceler. Üç koşul da yanlış olduğunda `harf`'e hiçbir değer atanmaz. O yoldan `println(harf)` satırına gelinirse yazılacak bir değer yoktur. Java, değer almamış bir yerel değişkenin okunmasına izin vermez. Bu kural, “değişkende ne olduğu belli olmayan” programları daha derleme aşamasında engeller.

Sorun iki yolla çözülür. Ya son `else` geri konur, ya da değişkene bildirim sırasında bir başlangıç değeri verilir (`char harf = 'F';`). İlk yol daha açıktır: her olası notun hangi harfe gittiği kodda tek tek görülür.

7.4 İç içe kararlar

Bazı sorular ancak başka bir sorunun cevabı belli olduktan sonra anlam kazanır. Araç kullanma izni için önce yaşa bakılır. Kişi 18 yaşından küçükse ehliyetinin olup olmadığını sormanın anlamı yoktur. Yaş yeterliyse ikinci soru sorulur: ehliyeti var mı?

7.4.1 Dal içinde karar

Tanım 7.5

İç içe karar (*nested decision*), bir karar yapısının dallarından birinin içinde başka bir karar yapısının bulunmasıdır. İçteki koşul yalnızca dıştaki koşul o dalı seçtiğinde denetlenir.

```
int yas = klavye.nextInt();
boolean ehliyet = klavye.nextBoolean();
if (yas >= 18) {
  if (ehliyet)
    System.out.println("Araç kullanabilir");
  else
    System.out.println("Önce ehliyet almalı");
} else {
  System.out.println("Yaş yetersiz");
}
```



Şekil 7.8: İç içe karar. Önce `yas >= 18` sorulur. true dalında ikinci bir karar, `ehliyet`, bulunur ve iki ayrı ileti yazar. false dalı doğrudan "Yaş yetersiz" yazar.

Şekil 7.8 yapıyı gösterir. Dış kararın true dalında ikinci bir karar kutusu vardır. Bu kutuya yalnızca yaş koşulu sağlandığında ulaşılır. Programın üç olası çıktısı vardır ve her biri ayrı bir yolun sonundadır.

Koddaki `nextBoolean()`, klavyeden `true` ya da `false` sözcüğünü okur ve boolean değere çevirir. `if (ehliyet)` yazımı da dikkat çeker. `ehliyet` zaten boolean bir değişkendir; `if (ehliyet == true)` yazmaya gerek yoktur. Değişkenin kendisi koşuldur.

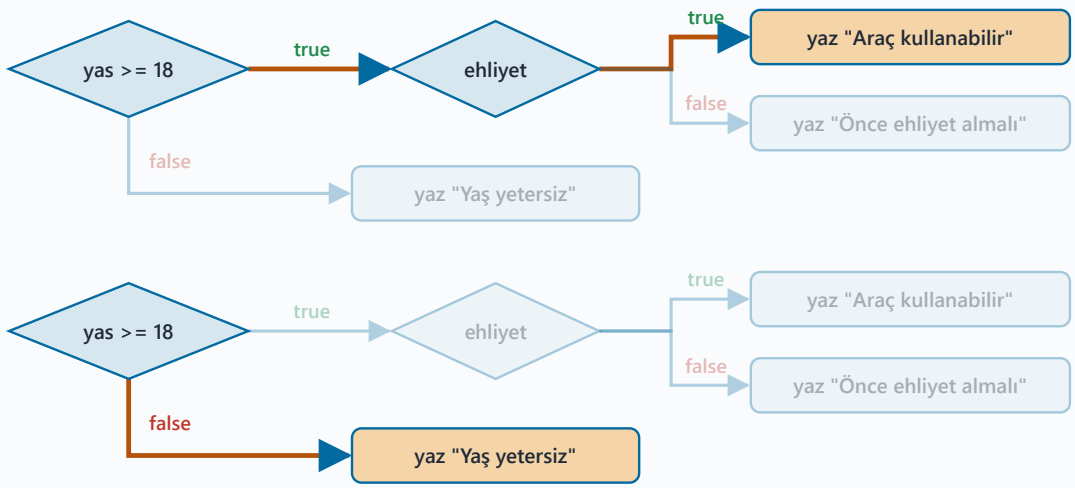
7.4.2 İç içe kararı adım adım izlemek

Program üç girdi çiftiyle çalıştırılsın: (20, true), (20, false) ve (15, true).

yas	ehliyet	Dış koşul `yas >= 18`	İç koşul `ehliyet`	Çıktı
20	true	*20 >= 18* → true	true	Araç kullanabilir
20	false	*20 >= 18* → true	false	Önce ehliyet almalı

yas	ehliyet	Dış koşul `yas >= 18`	İç koşul `ehliyet`	Çıktı
15	true	*15 >= 18* → false	denetlenmez	Yaş yetersiz

Tablo 7.6: İç içe kararın üç çalıştırması.



Şekil 7.9: İç içe kararda izlenen yol. (a) yas = 20, ehliyet = true: iki koşul da true, "Araç kullanabilir" yazılır. (b) yas = 15: dış koşul false; iç karar kutusu soluk kalır, çünkü hiç denetlenmez.

Üçüncü çalıştırma iç içeliğin anlamını gösterir. 15 yaşındaki kişinin ehliyeti `true` olarak girilmiştir; gerçek hayatta bu imkânsızdır ama program için önemsizdir. Dış koşul yanlış olduğunda iç `if` satırına hiç gelinmez. Tam programın gerçek çıktıları:

```
import java.util.Scanner;

public class Ehliyet {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        int yas = klavye.nextInt();
        boolean ehliyet = klavye.nextBoolean();
        if (yas >= 18) {
          if (ehliyet)
            System.out.println("Araç kullanabilir");
          else
            System.out.println("Önce ehliyet almalı");
        } else {
          System.out.println("Yaş yetersiz");
        }
    }
}
```

```
Araç kullanabilir
```

```
Önce ehliyet almalı
```

```
Yaş yetersiz
```

7.4.3 İç içelik yerine &&

İç içe yapı her düzeyde kodu biraz daha sağa kaydırır. Üç dört düzey derinlikte hangi `else`'in hangi `if`'e ait olduğu zor izlenir. Pek çok durumda iç içe kararlar mantıksal VE (`&&`) ile birleştirilerek düz bir zincire çevrilebilir:

```
import java.util.Scanner;

public class EhliyetVe {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        int yas = klavye.nextInt();
        boolean ehliyet = klavye.nextBoolean();
        if (yas >= 18 && ehliyet)
            System.out.println("Araç kullanabilir");
        else if (yas >= 18)
            System.out.println("Önce ehliyet almalı");
        else
            System.out.println("Yaş yetersiz");
    }
}
```

İlk koşul "18 yaşında ya da büyük ve ehliyetli" durumunu yakalar. İkinci koşula yalnızca ilki yanlışsa gelinir. O noktada `yas >= 18` doğruysa ehliyet yok demektir. Kalan her durum yaş yetersizliğidir. Program dört girdi çifti için iç içe sürümle aynı çıktıyı verdi:

Girdi	Ehliyet (iç içe)	EhliyetVe (&&)
20 true	Araç kullanabilir	Araç kullanabilir
20 false	Önce ehliyet almalı	Önce ehliyet almalı
15 true	Yaş yetersiz	Yaş yetersiz
15 false	Yaş yetersiz	Yaş yetersiz

Tablo 7.7: İç içe sürüm ile && sürümünün dört girdi için çıktıları.

Hangi yazımın seçileceği okunabilirliğe bağlıdır. İkinci soru gerçekten ilkine bağlıysa, yani ilk koşul yanlışken ikinci soru anlamsızsa, iç içe yapı bu bağımlılığı açıkça gösterir. Koşullar eşit ağırlıktaysa `&&` ile tek satırda birleştirmek daha kısa ve daha düzdür.

7.4.4 Sarkan else

İç içe if'lerde süslü parantez kullanılmadığında bir belirsizlik ortaya çıkar. Aşağıdaki kodda tek bir `else` ama iki `if` vardır. `else` hangisine aittir?

```
int a = -1, b = 5, x = 0;
if (a > 0)
    if (b > 0)
        x = 1;
    else
        x = 2;
System.out.println(x);
```

Tanım 7.6

Sarkan else (*dangling else*), süslü parantezsiz iç içe if'lerde bir `else`'in hangi `if`'e bağlandığının yazımdan açıkça anlaşılmaması durumudur. Java'nın kuralı kesindir: `else`, kendisinden önce gelen ve henüz bir `else`'i olmayan *en yakın* `if`'e bağlanır. Girinti bu kuralı değiştirmez.

Kodda `else` iç `if`'in hizasında yazılmıştır; burada girinti kuralla uyumludur. Yine de programcı `else`'i dış `if`'e ait sanabilir. Başlangıç değerleri a = −1, b = 5, x = 0 olsun. Adımlar şöyledir:

1. `a > 0` hesaplanır: *−1 > 0* yanlıştır.
2. Dış `if`'in gövdesi atlanır. Bu gövde, iç `if` ile onun `else`'inden oluşan tek bir deyimdir. Dolayısıyla `else` da atlanır.
3. `x` hiç değişmez; ekrana 0 yazılır.

```
0
```

Programcı `else`'i dış `if`'e ait sanıyorsa 2 bekler. Gerçekte `else` iç `if`'e bağlıdır ve iç `if`'e hiç gelinmediği için o da çalışmaz. Aynı kod farklı girdilerle denendiğinde kural daha da açık görünür:

a	b	x (başta)	`a > 0`	`b > 0`	Çalışan atama	Çıktı
−1	5	0	false	denetlenmez	yok	0
−4	6	9	false	denetlenmez	yok	9
3	5	0	true	true	`x = 1`	1
3	−2	0	true	false	`x = 2` (iç else)	2

Tablo 7.8: Sarkan else kodunun dört girdiyle çalıştırılması. Program a, b ve x'in başlangıç değerlerini klavyeden okuyacak biçimde değiştirilmiştir.

`x = 2` ataması yalnızca a pozitif, b pozitif değilken çalışır. Bu, `else`'in iç `if`'e ait olduğunun kanıtıdır. Aynı sonuç, `else` satırı en sola, dış `if`'in hizasına kaydırılsa da değişmez. Derleyici boşlukları okumaz.

Programcının niyeti `else`'i dış `if`'e bağlamaksa iç `if` süslü parantezle kapatılır. Kapanan süslü parantez iç `if`'i bitirir; ardından gelen `else` artık ona bağlanamaz.

```
int a = -1, b = 5, x = 0;
if (a > 0) {
    if (b > 0) x = 1;
} else {
    x = 2;
}
System.out.println(x);
```

```
2
```

Aynı başlangıç değerleriyle *−1 > 0* yanlıştır. Bu kez `else` dış `if`'e aittir ve çalışır: `x = 2` olur, 2 yazılır. İki kod arasındaki tek fark bir çift süslü parantezdir; çıktılar ise 0 ve 2'dir.

Sık yapılan hata

Kodun ne yaptığını girintiye bakarak çıkarmak. Girinti yalnızca okura yöneliktir; Java derleyicisi için boşluk ve satır başı anlamsızdır. İç içe if'lerde her gövdeyi süslü parantezle yazmak belirsizliği tümüyle ortadan kaldırır. Geliştirme ortamlarının “kodu biçimlendir” komutu da girintiyi derleyicinin gördüğü yapıya göre yeniden düzenler; sarkan bir `else` biçimlendirmeden sonra gerçek yerine kayar.

7.5 Mantıksal işleçlerle karar

Gerçek kararlar çoğu zaman birden çok koşula bağlıdır. Bir başvuru, kişi 18 yaşında ya da büyükse *ve* belgesi varsa kabul edilir. Bir gün, cumartesi *ya da* pazarsa hafta sonudur. Bu “ve”, “ya da” ve “değil” bağlaçlarının Java'daki karşılıkları mantıksal işleçlerdir.

7.5.1 &&, || ve !

Tanım 7.7

Mantıksal işleçler (*logical operators*) boolean değerleri birleştirir ve yeni bir boolean değer üretir. `a && b` (VE) yalnızca iki taraf da `true` iken `true` 'dur. `a || b` (VEYA) en az bir taraf `true` iken `true` 'dur. `!a` (DEĞİL) değeri tersine çevirir.

a	b	a && b	a || b	!a
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Tablo 7.9: Mantıksal işleçlerin doğruluk tablosu. Dört satır, iki boolean değerin bütün birleşimleridir.

`||` 'nin anlamı günlük dildeki "ya ... ya ..."dan farklıdır. Günlük dilde "ya çay ya kahve" çoğu zaman ikisinden yalnızca birini anlatır. `||` ise iki taraf da doğruyken de `true` verir. Başvuru örneği `&&` ile şöyle yazılır:

```
boolean uygun = yas >= 18 && belgeVar;
if (uygun) {
    System.out.println("Başvuru alınabilir");
}
```



Şekil 7.10: `yas >= 18` ve `belgeVar` değerleri `&&` ile birleşerek `uygun` değerini oluşturur. Sağdaki tabloda dört birleşimden yalnızca ikisi de true olan hücre true verir.

Koşulun sonucu önce `uygun` adlı boolean bir değişkene atanmıştır. Bu bir zorunluluk değildir; koşul doğrudan `if` 'in parantezine de yazılabilirdi. Ama koşula bir ad vermek kodu okunur kılar: `if (uygun)` satırı kendini açıklar.

7.5.2 Kısa devre değerlendirme

`&&` ve `||` önce sol tarafı hesaplar. Sol taraf sonucu tek başına belirliyorsa sağ taraf hiç hesaplanmaz.

Tanım 7.8

Kısa devre değerlendirme (*short-circuit evaluation*): `a && b` ifadesinde a `false` ise sonuç ne olursa olsun `false`'tur ve b hesaplanmaz. `a || b` ifadesinde a `true` ise sonuç `true`'dur ve b hesaplanmaz. Sağ taraf yalnızca sonucu belirlemek için gerekiyorsa hesaplanır.

Başvuru programı üç girdi için izlensin:

```
int yas = klavye.nextInt();
boolean belgeVar = klavye.nextBoolean();
boolean uygun = yas >= 18 && belgeVar;
if (uygun) {
    System.out.println("Başvuru alınabilir");
}
```

yas	belgeVar	Sol: `yas >= 18`	Sağ: `belgeVar`	uygun	Çıktı
20	true	true	okunur: true	true	Başvuru alınabilir
20	false	true	okunur: false	false	(çıktı yok)
16	true	false	okunmaz	false	(çıktı yok)

Tablo 7.10: `uygun = yas >= 18 && belgeVar` ifadesinin üç girdi için değerlendirilmesi.

Üçüncü satırda sol taraf yanlıştır. Sağ tarafın değeri ne olursa olsun VE'nin sonucu yanlış olacaktır; Java `belgeVar`'a hiç bakmaz. Bu örnekte kısa devre yalnızca küçük bir zaman kazancıdır. Asıl önemi, sağ tarafın ancak sol taraf doğruyken güvenle hesaplanabildiği durumlarda ortaya çıkar. Bölmede bölenin sıfır olmaması böyle bir koşuldur:

```
if (b != 0 && a / b > 1) { … }
```

b = 0 ise sol taraf yanlıştır ve `a / b` hiç hesaplanmaz. Tam sayılarda sıfıra bölme programı durduran bir hatadır. Kısa devre, koşulun sağ tarafını sol tarafın koruması altına alır. Sıra önemlidir: `a / b > 1 && b != 0` yazılsaydı bölme denetimden önce yapılırdı.

7.5.3 De Morgan kuralları

Başvurunun reddedileceği durumu yazmak gerekirse en doğrudan yol kabul koşulunun başına `!` koymaktır: `!(yas >= 18 && belgeVar)`. Bu yazım doğrudur ama okunması zordur. Okur önce parantezin içini, sonra tersini düşünmek zorunda kalır. De Morgan kuralları `!`'i parantezin içine dağıtarak daha düz bir koşul elde etmeyi sağlar.

Teorem 7.1 (De Morgan kuralları)

Her a ve b boolean değeri için

> !(a && b) == (!a || !b) ve !(a || b) == (!a && !b)

eşitlikleri sağlanır. `!` parantezin içine dağıtılırken `&&` ile `||` yer değiştirir ve her terim tersine döner.

Kanıt doğruluk tablosuyla yapılır. a ve b'nin alabileceği yalnızca dört değer birleşimi vardır; dördünde de iki taraf aynı değeri verirse eşitlik her durumda doğrudur. Tablo 7.11 dört satırı verir. Tablo, bir Java programının dört birleşimi tek tek hesaplamasıyla üretilmiştir.

a	b	`!(a && b)`	`!a || !b`	`!(a || b)`	`!a && !b`
false	false	true	true	true	true
false	true	true	true	false	false
true	false	true	true	false	false
true	true	false	false	false	false

Tablo 7.11: De Morgan kurallarının doğruluk tablosu. Üçüncü ve dördüncü sütun her satırda aynıdır; beşinci ve altıncı sütun da öyle.

Kural başvuru koşuluna uygulandığında `!(yas >= 18)` terimi de sadeleşir. “18'den büyük ya da eşit değil” demek “18'den küçük” demektir. Böylece ret koşulu şu biçimi alır:

```
// Başvuru reddedilsin: 18 yaşından küçük ya da belge yok
if (!(yas >= 18 && belgeVar)) { … }
// De Morgan ile aynı koşul:
if (yas < 18 || !belgeVar) { … }
```

İki koşulun eşdeğerliği de bir programla sınandı: yaş 0'dan 40'a kadar, `belgeVar` iki değeriyle, toplam *41 × 2 = 82* birleşim denendi. 82 birleşimin hepsinde iki koşul aynı sonucu verdi. Karşılaştırmaların tersleri Tablo 7.2'nin son sütunundadır: `>=` ↔ `<`, `>` ↔ `<=`, `==` ↔ `!=`.

Örnek 7.1

Bir sıcaklık değerinin 15 ile 30 arasında (iki sınır dahil) *olmadığı* durum, `!` kullanmadan nasıl yazılır?

Çözüm. Aralığın içi `s >= 15 && s <= 30`'dur. Dışı bunun tersidir: `!(s >= 15 && s <= 30)`. De Morgan kuralıyla `&&`, `||`'ye döner ve her terim tersine çevrilir: `!(s >= 15)` → `s < 15`, `!(s <= 30)` → `s > 30`. Sonuç: `s < 15 || s > 30`. Bir sıcaklık aynı anda 15'ten küçük ve 30'dan büyük olamayacağı için koşulda `&&` kalsaydı hiçbir değer onu sağlamazdı.

7.5.4 Kısa devresiz & ve |

Java'da tek karakterli `&` ve `|` işleçleri de boolean değerlerle kullanılabilir. Mantık sonuçları `&&` ve `||` ile aynıdır. Fark değerlendirme biçimindedir: tek karakterli işleçler iki tarafı da *her zaman* hesaplar. Bu fark, sağ taraf bir hata ya da bir yan etki taşıdığında görünür olur.

```
public class KisaDevre {
    public static void main(String[] args) {
        int a = 10, b = 0;
        if (b != 0 && a / b > 1) {
            System.out.println("büyük");
        }
        System.out.println("&& güvenli");
        if (b != 0 &  a / b > 1) {
            System.out.println("büyük");
        }
        System.out.println("buraya gelinmez");
    }
}
```

```
&& güvenli
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at KisaDevre.main(KisaDevre.java:8)
```

`&&`'li satır güvenle geçilir ve "&& güvenli" yazılır. `&`'li satırda ise b = 0 olduğu hâlde `a / b` hesaplanır. Java bir **istisna** (*exception*) fırlatır: `ArithmeticException: / by zero`, yani "sıfıra bölme". Program 8. satırda durur ve son `println`'e ulaşılmaz.

Yan etkiyi görmenin bir yolu, her çağrıldığında bir sayacı artıran küçük bir metot yazmaktır:

```
static int sayac = 0;
static boolean kontrol(boolean v) { sayac++; return v; }
```

İfade	Sonuç	sayac	Açıklama
`kontrol(false) && kontrol(true)`	false	1	Sol false; sağ çağrılmaz.
`kontrol(false) & kontrol(true)`	false	2	İki taraf da çağrılır.
`kontrol(true) || kontrol(true)`	true	1	Sol true; sağ çağrılmaz.
`kontrol(true) | kontrol(true)`	true	2	İki taraf da çağrılır.

Tablo 7.12: Her ifadeden önce sayac = 0 yapılır. Mantık sonucu aynı kalırken sayaç, kaç tarafın hesaplandığını gösterir.

Tablodaki değerler aşağıdaki programın çıktısıdır. Her satırda önce sonuç, sonra sayaç yazılır.

```
public class Sayac {
    static int sayac = 0;
    static boolean kontrol(boolean v) { sayac++; return v; }

    public static void main(String[] args) {
        boolean r;
        sayac = 0; r = kontrol(false) && kontrol(true); System.out.println(r + " " +
sayac);
        sayac = 0; r = kontrol(false) &  kontrol(true); System.out.println(r + " " +
sayac);
        sayac = 0; r = kontrol(true)  || kontrol(true); System.out.println(r + " " +
sayac);
        sayac = 0; r = kontrol(true)  |  kontrol(true); System.out.println(r + " " +
sayac);
    }
}
```

```
false 1
false 2
true 1
true 2
```

Koşullarda neredeyse her zaman `&&` ve `||` kullanılır. `&` ve `|`'nin asıl kullanım alanı tam sayılar üzerindeki bit işlemleridir.

Sık yapılan hata

Koşulun içine değişken değiştiren bir işlem (`++k` gibi) koymak. Kısa devre yüzünden bu işlemin çalışıp çalışmadığı sol tarafın değerine bağlı hâle gelir. `if (k > 0 && ++k > 5)` satırında k yalnızca pozitifken artar. Böyle bir kodun davranışını izlemek zordur; artırma koşuldan önce ayrı bir satırda yapılmalıdır.

7.5.5 İşleç önceliği

Bir koşulda aritmetik, karşılaştırma ve mantıksal işleçler bir arada bulunabilir. Hangisinin önce hesaplanacağını **işleç önceliği** (*operator precedence*) belirler. Koşullarda sık geçen işleçlerin önceliği yüksekten düşüğe Tablo 7.13'te verilmiştir.

Sıra	İşleçler	Tür
1	`!`, tekli `-`, `++`, `--`	tekli
2	`*` `/` `%`	çarpma türü
3	`+` `-`	toplama türü
4	`<` `<=` `>` `>=`	sıralama karşılaştırması
5	`==` `!=`	eşitlik
6	`&`	kısa devresiz VE
7	`|`	kısa devresiz VEYA
8	`&&`	VE
9	`||`	VEYA
10	`? :`	koşul işleci
11	`=` `+=` ...	atama

Tablo 7.13: Koşullarda sık geçen işleçlerin önceliği (üstteki önce hesaplanır).

Bu sıralama sayesinde `yil % 4 == 0 && yil % 100 != 0` parantezsiz yazılabilir. Önce `%` işlemleri, sonra `==` ve `!=` karşılaştırmaları, en son `&&` hesaplanır. `&&`'nin `||`'den önce gelmesi de önemlidir: `a || b && c` ifadesi `a || (b && c)` olarak okunur. `!`'in önceliği en yüksektir; bu yüzden bir karşılaştırmanın tersi alınırken parantez gerekir: `!(x > 5)`. Parantezsiz `!x > 5` yazımı `(!x) > 5` demektir ve x bir sayıysa derlenmez. Öncelikten emin olunamayan her durumda parantez kullanmak hem hatayı önler hem de okura yardım eder.

7.6 Koşul işleci ?:

Bir kararın tek işi bir değişkene iki değerden birini vermekse if-else dört satır tutar:

```
int enBuyuk;
if (a > b) enBuyuk = a;
else       enBuyuk = b;
```

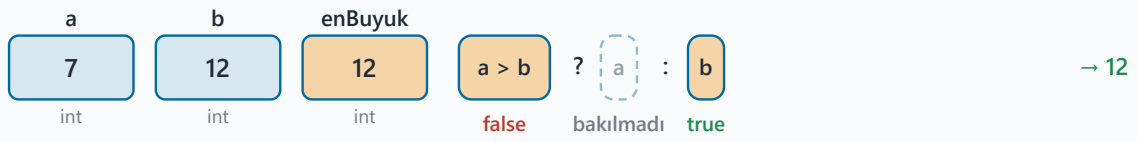
Java bu kalıp için daha kısa bir yazım sunar.

Tanım 7.9

Koşul işleci (*conditional operator*, üç terimli olduğu için *ternary operator* da denir) `koşul ? değer1 : değer2` biçiminde yazılır. Koşul `true` ise ifadenin değeri değer1, `false` ise değer2'dir. Yalnızca seçilen değer hesaplanır.

if-else bir deyimdir: bir iş yapar ama bir değer üretmez. Koşul işleci ise bir **ifadedir** (*expression*): bir değer üretir. Bu yüzden atamanın sağ tarafında, bir metodun argümanında ya da başka bir ifadenin içinde kullanılabilir.

```
int a = 7, b = 12;
int enBuyuk = (a > b) ? a : b;
System.out.println(enBuyuk);
```



Şekil 7.11: `(a > b) ? a : b` ifadesinin a = 7, b = 12 için değerlendirilmesi. Koşul false olduğu için iki noktadan sonraki b seçilir; a'ya bakılmaz. enBuyuk = 12 olur.

Adımlar şöyledir. Önce koşul hesaplanır: *7 > 12* yanlıştır. Koşul yanlış olduğu için iki noktadan (`:`) sonraki değer, yani b seçilir. İfadenin değeri 12'dir ve `enBuyuk`'a atanır. Program 12 yazar:

```
12
```

Koşul işleci iç içe de yazılabilir. Bir sayının işaretini "+", "-" ya da "0" olarak bulan ifade şöyledir:

```
String isaret = n > 0 ? "+" : n < 0 ? "-" : "0";
```

Koşul işleci sağdan gruplanır. İfade `n > 0 ? "+" : (n < 0 ? "-" : "0")` olarak okunur. n = −5 için ilk koşul yanlıştır; parantezin içi hesaplanır ve *−5 < 0* doğru olduğundan "-" seçilir. Kısa ve doğru bir ifadedir; ama üç dört düzey iç içe koşul işleci okunmaz hâle gelir. Koşul işleci en çok iki seçenekli, kısa değer seçimlerinde işe yarar. Seçenekler çoğaldığında else-if zinciri ya da switch daha açıktır.

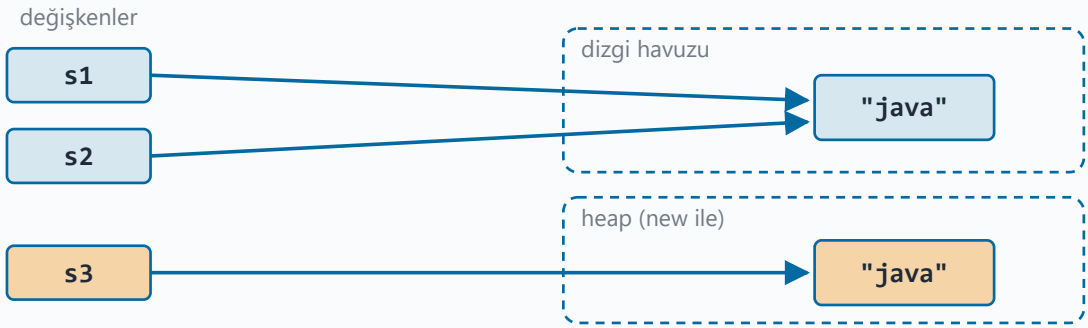
Koşul işlecinin iki değeri uyumlu tipte olmalıdır. `int m = a < b ? a : "yok";` derlenmez, çünkü ifadenin bir dalı int, öteki dizgidir ve sonuç bir int değişkene konamaz.

7.7 Dizgileri karşılaştırmak: == ve equals

Sayılar `==` ile karşılaştırılır. Dizgiler (`String`) için durum farklıdır ve Java'ya yeni başlayanların en sık takıldığı noktalardan biridir. Bir dizgi değişkeni metnin kendisini değil, bellekte metni tutan nesnenin yerini, yani bir **referansı** (*reference*) saklar. `==` iki referansı karşılaştırır: iki değişken *aynı nesneyi* mi gösteriyor? İçerik karşılaştırması için `equals` metodu kullanılır.

```
String s1 = "java";
String s2 = "java";
String s3 = new String("java");
System.out.println(s1 == s2);
System.out.println(s1 == s3);
System.out.println(s1.equals(s3));
```

```
true
false
true
```



Şekil 7.12: s1 ve s2, dizgi havuzundaki aynı "java" nesnesini gösterir. s3, `new` ile heap'te oluşturulan ayrı bir "java" nesnesini gösterir. İki nesnenin içeriği aynı, yerleri farklıdır.

Şekil 7.12 üç değişkenin bellekteki durumunu gösterir. Kodda tırnak içinde yazılan bir metne **dizgi sabiti** (*string literal*) denir. Java, dizgi sabitlerini **dizgi havuzu** (*string pool*) adı verilen ortak bir bölgede tutar. Aynı sabit ikinci kez yazıldığında yeni nesne oluşturulmaz; havuzdaki nesne paylaşılır. Bu yüzden s1 ve s2 aynı nesneyi gösterir. `new String("java")` ise her zaman yeni bir nesne oluşturur. Bu nesne, `new` ile oluşturulan nesnelerin tutulduğu **heap** bölgesindedir.

İfade	Neyi karşılaştırır?	Sonuç	Neden?
`s1 == s2`	referans	true	İkisi de havuzdaki aynı nesneyi gösterir.
`s1 == s3`	referans	false	s3 heap'teki ayrı nesneyi gösterir.

İfade	Neyi karşılaştırır?	Sonuç	Neden?
`s1.equals(s3)`	içerik	true	İki nesnenin karakterleri aynıdır: j, a, v, a.

Tablo 7.14: Üç karşılaştırmanın açıklaması.

İlk satırın `true` çıkması tehlikelidir, çünkü `==`'nin içerik karşılaştırdığı izlenimini verir. Oysa bu sonuç yalnızca havuzun bir yan ürünüdür. Kullanıcıdan okunan bir dizgi havuza girmez; `Scanner` onu yeni bir nesne olarak oluşturur. Aşağıdaki program bunu gösterir:

```
import java.util.Scanner;

public class DizgiGirdi {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        String cevap = klavye.next();
        System.out.println(cevap == "evet");
        System.out.println(cevap.equals("evet"));
        System.out.println(cevap.equalsIgnoreCase("EVET"));
    }
}
```

Kullanıcı “evet” yazdığında çıktı şudur:

```
false
true
true
```

Kullanıcı tam olarak “evet” yazmış olsa da `cevap == "evet"` yanlıştır: biri klavyeden okunan yeni nesne, öteki havuzdaki sabittir. `equals` doğru sonucu verir. Kullanıcı “Evet” yazdığında ise `equals` de yanlış verir, çünkü büyük-küçük harfe duyarlıdır: “E” ile “e” farklı karakterlerdir. Harf büyüklüğünü önemsemeyen karşılaştırma `equalsIgnoreCase` ile yapılır:

```
false
false
true
```

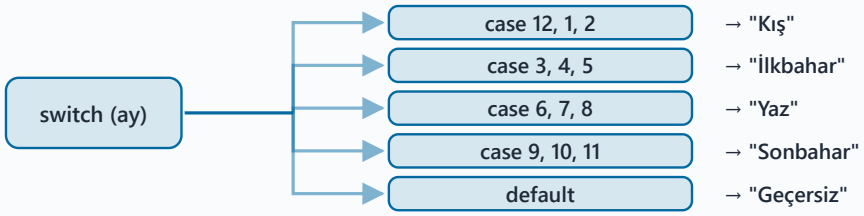
Sık yapılan hata

Dizgileri `==` ile karşılaştırmak. Sabitlerle yapılan küçük denemelerde program doğru çalışıyor gibi görünür; kullanıcıdan okunan ya da program içinde birleştirilen dizgilerde ise sonuç yanlış çıkar. Kural basittir: dizgiler her zaman `equals` ile karşılaştırılır. Dizgi değişkeni `null` olabiliyorsa sabit başa yazılır: `"evet".equals(cevap)`. Bu yazım, cevap `null` olduğunda hata vermez, yalnızca `false` döndürür.

7.8 switch

else-if zinciri her türlü çok yollu kararı kurabilir. Ancak bazı kararlarda bütün koşullar aynı değişkenin aynı biçimde sınanmasıdır: `ay == 1`, `ay == 2`, … `ay == 12`. On iki ayı mevsimlere ayırmak için uzun bir `||` zinciri gerekir. `switch` yapısı bu tür kararları daha kısa ve okunur kılar.

7.8.1 switch ne zaman kullanılır?



Şekil 7.13: switch ifadesi ay değerine göre doğrudan eşleşen case'e gider. Her case bir ya da birkaç sabit değer listeler; hiçbiri eşleşmezse default seçilir.

`switch`, tek bir ifadenin değerini bir dizi sabitle karşılaştırır ve eşleşen seçeneğe gider. Şekil 7.13'te ay değeri beş seçenekten birine yönlenir. Seçenekler **case etiketleriyle** (*case label*) yazılır. Her etiket bir ya da birkaç sabit değer taşır. Hiçbir etiket eşleşmezse `default` seçilir.

else-if zincirinde koşullar sırayla denenir. `switch`'te ise program, değerin eşleştiği case'e doğrudan gider; arada kalan etiketler "denenip geçilmez". Bu yüzden switch'in uygun olduğu yer, tek bir değişkenin ayrık değerlerine göre yapılan seçimlerdir: ay numarası, gün numarası, menü seçeneği, işlem işareti. Aralık koşulları ("not 85'ten büyükse") ve birden çok değişkene bağlı karmaşık koşullar if-else ile yazılır.

Bir case etiketindeki değer derleme sırasında bilinen bir **sabit** olmalıdır: `4`, `'A'`, `"+"` gibi. Bir değişken etiket olamaz:

```
public class DegiskenCase {
    public static void main(String[] args) {
        int ay = 4, sinir = 4;
        switch (ay) {
            case sinir -> System.out.println("eşit");
            default -> System.out.println("farklı");
        }
    }
}
```

```
DegiskenCase.java:5: error: constant expression required
            case sinir -> System.out.println("eşit");
                 ^
1 error
```

switch'in sınadığı ifade `int`, `char`, `String` gibi tiplerden olabilir. `double` ve `boolean` değerler switch ile sınanamaz; bunlar için if-else kullanılır.

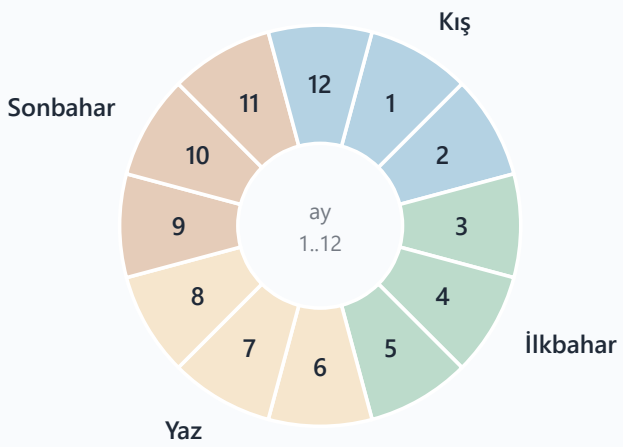
7.8.2 switch ifadesi

Java 14 ile dile giren yeni yazımda her case bir ok (`->`) ile yazılır ve switch bir değer üretebilir.

Tanım 7.10

switch ifadesi (*switch expression*), seçilen case'in değerini sonuç olarak üreten switch'tir: `x = switch (…) { … };`. **switch deyimi** (*switch statement*) ise bir değer üretmez; yalnızca seçilen case'in kodunu çalıştırır. Okla yazılan her case'e **ok biçimi** denir; ok biçiminde yalnızca seçilen case çalışır.

```
String mevsim = switch (ay) {
    case 12, 1, 2 -> "Kış";
    case 3, 4, 5 -> "İlkbahar";
    case 6, 7, 8 -> "Yaz";
    case 9, 10, 11 -> "Sonbahar";
    default -> "Geçersiz";
};
```



Şekil 7.14: Mevsim çarkı. 12, 1 ve 2 Kış; 3, 4 ve 5 İlkbahar; 6, 7 ve 8 Yaz; 9, 10 ve 11 Sonbahar'dır. 1–12 dışındaki değerler default'a, "Geçersiz"e gider.

Koddaki her satır bir case'tir. `case 12, 1, 2 -> "Kış";` satırı "ay 12, 1 ya da 2 ise değer "Kış"tır" demektir. Bir case'te virgülle ayrılmış birden çok değer listelenebilir. Bu sayede on iki ay beş satırda karşılanır (Şekil 7.14). switch ifadesinin bütünü bir değerdir; bu değer `mevsim` değişkenine atanır. Atama deyimi olduğu için kapanan süslü parantezden sonra noktalı virgül gelir: `};`.

Bir switch ifadesi her olası değer için bir sonuç üretmek zorundadır. `int` 4 milyardan fazla değer alabilir; case'ler bunların yalnızca on ikisini listeler. Bu yüzden `default` zorunludur. Silindiğinde Java programı derlemez:

```
DefaultYok.java:4: error: the switch expression does not cover all possible input values
        String mevsim = switch (ay) {
                        ^
1 error
```

İleti "switch ifadesi bütün olası girdi değerlerini kapsamıyor" anlamına gelir. switch deyiminde ise `default` isteğe bağlıdır; hiçbir case eşleşmezse switch hiçbir şey yapmadan biter. Aynı değer iki case'te birden geçerse de derleyici hata verir, çünkü değerin hangi dala gideceği belirsiz kalır:

```
YinelenenCase.java:6: error: duplicate case label
            case 2, 3, 4, 5 -> "İlkbahar";
            ^
1 error
```

7.8.3 switch ifadesini adım adım izlemek

```
int ay = klavye.nextInt();
String mevsim = switch (ay) {
    case 12, 1, 2 -> "Kış";
    case 3, 4, 5 -> "İlkbahar";
    case 6, 7, 8 -> "Yaz";
    case 9, 10, 11 -> "Sonbahar";
    default -> "Geçersiz";
};
System.out.println(mevsim);
```

ay	Eşleşen case	switch'in değeri	mevsim	Çıktı
4	`case 3, 4, 5`	"İlkbahar"	"İlkbahar"	İlkbahar
12	`case 12, 1, 2`	"Kış"	"Kış"	Kış
13	yok → `default`	"Geçersiz"	"Geçersiz"	Geçersiz

Tablo 7.15: Mevsim programının üç çalıştırması.

ay = 4 için program, 4'ün listede bulunduğu `case 3, 4, 5` satırına gider. Bu case'in değeri "İlkbahar"dır. switch ifadesinin değeri olarak `mevsim`'e atanır ve yazılır. Başka hiçbir case'in kodu çalışmaz. ay = 13 hiçbir listede yoktur; `default` seçilir. Program ayrıca 1, 7, 10, 0 ve −3 için denendi: Kış, Yaz, Sonbahar, Geçersiz ve Geçersiz yazdı.

```
İlkbahar
```

```
Kış
```

```
Geçersiz
```

7.8.4 Bloklu dal ve yield

Ok biçiminde bir case'in sağ tarafı tek bir ifadedir. Bazen bir dal, değeri üretmeden önce başka işler de yapmak zorundadır: bir ileti yazmak, bir ara değer hesaplamak. Böyle bir dal süslü parantezle bir bloğa alınır ve değer `yield` sözcüğüyle verilir.

Tanım 7.11

yield, bir switch ifadesinin bloklu dalında o dalın ürettiği değeri belirten deyimdir. `yield` çalıştığında blok biter ve verilen değer switch ifadesinin değeri olur.

```
int yil = 2024, ay = 2;
int gun = switch (ay) {
    case 4, 6, 9, 11 -> 30;
    case 2 -> {
        System.out.println("Şubat");
        yield yil % 4 == 0 ? 29 : 28;
    }
    default -> 31;
};
System.out.println(gun);
```

```
Şubat
29
```

ay = 2 olduğu için bloklu dal seçilir. Blok önce “Şubat” yazar. Ardından `yield` satırı çalışır. Buradaki değer bir koşul işleciyle hesaplanır: *2024 % 4 = 0* olduğundan `yil % 4 == 0` doğrudur ve 29 seçilir. switch ifadesinin değeri 29 olur, `gun`'e atanır ve yazılır. Çıktı iki satırdır: “Şubat” ve “29”.

Aynı program farklı girdilerle de çalıştırıldı. 2023 yılının şubatı için “Şubat” ve 28 yazdı (*2023 % 4 = 3*). Nisan için yalnızca 30, ocak için yalnızca 31 yazdı; bu iki çalıştırmada blok seçilmediği için “Şubat” satırı da yazılmadı. Bu örnekte artık yıl kuralı basitleştirilmiştir: yalnızca 4'e bölünebilmeye bakılır. Bu yüzden program 1900 için de 29 yazar; oysa 1900 artık yıl değildir. Tam kural artık yıl örneğinde verilmiştir.

Bloklu dal `yield` olmadan biterse dalın bir değeri olmaz. Java bunu derleme sırasında yakalar:

```
public class YieldYok {
    public static void main(String[] args) {
        int yil = 2024, ay = 2;
        int gun = switch (ay) {
            case 4, 6, 9, 11 -> 30;
            case 2 -> {
                System.out.println("Şubat");
            }
            default -> 31;
        };
        System.out.println(gun);
    }
}
```

```
YieldYok.java:8: error: switch rule completes without providing a value
            }
            ^
  (switch rules in switch expressions must either provide a value or throw)
1 error
```

İleti “switch kuralı bir değer vermeden tamamlanıyor” anlamına gelir; ek satır, switch ifadelerindeki kuralların ya bir değer vermesi ya da bir istisna fırlatması gerektiğini hatırlatır.

7.8.5 Klasik switch, break ve aşağı düşme

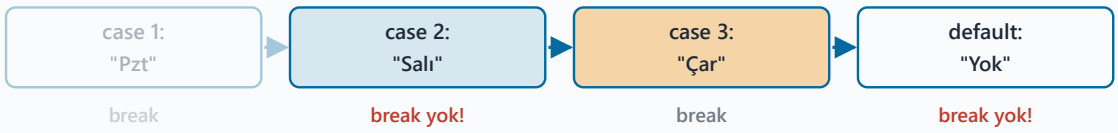
Ok biçiminden önce Java'da switch yalnızca iki nokta (`:`) ile yazılırdı. Bu klasik yazım bugün de geçerlidir ve eski kodlarda sıkça görülür. Klasik switch'te case etiketleri yalnızca birer giriş noktasıdır. Program eşleşen etiketten içeri girer ve `break` görene ya da switch'in sonuna gelene kadar alttaki bütün kodu çalıştırır.

Tanım 7.12

Aşağı düşme (*fall-through*), klasik switch'te bir case'in kodu `break` olmadan bittiğinde yürütmenin bir sonraki case'in koduna geçmesidir. Bu sırada sonraki case'in etiketi denetlenmez. `break` deyimi switch'ten çıkar.

```java
int gun = 2;
switch (gun) {
  case 1: System.out.println("Pzt"); break;
  case 2: System.out.println("Salı");
  case 3: System.out.println("Çar"); break;
  default: System.out.println("Yok");
}
```

```
Salı
Çar
```



Şekil 7.15: gun = 2 için klasik switch. Yürütme doğrudan case 2'ye girer ve “Salı” yazar. case 2'de break olmadığı için case 3'ün koduna düşer ve “Çar” yazar. case 3'teki break switch'ten çıkarır; default çalışmaz.

Adımlar şöyledir. gun = 2'dir. Program doğrudan `case 2:` etiketine gider; case 1 denetlenmez. “Salı” yazılır. Bu satırda `break` yoktur. Yürütme alttaki satıra, `case 3:`'ün koduna geçer. Etiketin 3 olması önemsizdir; etiket yalnızca giriş noktasıdır. “Çar” yazılır. Bu satırdaki `break` switch'ten çıkar ve `default` çalışmaz (Şekil 7.15). Aynı program gun = 1 için yalnızca “Pzt”, 3 için yalnızca “Çar”, 9 için “Yok” yazdı.

Aşağı düşme bazen bilerek kullanılır: birkaç case'in aynı işi yapması için etiketler alt alta yazılır ve kodu yalnızca sonuncusuna konur. Ok biçiminde bu ihtiyaç virgüllü liste ile karşılanır. Aşağıdaki program hafta içi ve hafta sonu günlerini aşağı düşmeye gerek kalmadan ayırır:

```
import java.util.Scanner;

public class GunTuru {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        int gun = klavye.nextInt();
        switch (gun) {
            case 1, 2, 3, 4, 5 -> System.out.println("Hafta içi");
            case 6, 7 -> System.out.println("Hafta sonu");
            default -> System.out.println("Geçersiz gün");
        }
    }
}
```

```
Hafta içi
```

```
Hafta sonu
```

```
Geçersiz gün
```

Ok biçiminde aşağı düşme hiçbir zaman olmaz: seçilen case'in kodu çalışır ve switch biter. `break` yazmaya gerek kalmaz. Yeni kodda ok biçimini kullanmak, unutulan bir `break`'in yol açtığı hatayı baştan önler.

Sık yapılan hata

Klasik switch'te `break`'i unutmak. Program derlenir, çoğu girdide doğru da çalışabilir; hata yalnızca `break`'i eksik case seçildiğinde, fazladan yazılan satırlarla ortaya çıkar. Bir başka yanılgı da `default`'un her zaman en son çalıştığını sanmaktır. `default` switch'in ortasında da yazılabilir ve aşağı düşme kuralı onun için de geçerlidir: `break`'i yoksa altındaki case'in koduna düşülür.

7.8.6 if-else mi, switch mi?



Şekil 7.16: Solda if-else ile sayı aralıkları: 50'nin altı Kaldı, 50–85 Geçti, 85 ve üstü Pekiyi. Sağda switch ile tek değişkenin ayrık değerleri: 1–5 hafta içi, 6–7 hafta sonu.

Şekil 7.16 iki yapının doğal kullanım alanlarını yan yana koyar. Not aralıkları süreklidir: 50 ile 85 arasında sonsuz sayıda ondalık değer vardır ve hepsini case olarak listelemek imkânsızdır. Günler ise ayrıktır: yalnızca yedi değer vardır ve her biri adıyla listelenebilir. Tablo 7.16 iki yapıyı birkaç ölçüte göre karşılaştırır.

Ölçüt	if-else	switch
Aralık koşulu	Uygun	Doğrudan uygun değil
Tek değere göre çok seçenek	Kullanılabilir	Genellikle daha açık
Karmaşık mantık	Uygun	Sınırlı
Her dal sonuç üretir	Atama gerekir	switch ifadesi değer döndürebilir

Tablo 7.16: if-else ile switch'in karşılaştırılması.

Son satır önemli bir avantajı gösterir. else-if zincirinde her dal aynı değişkene ayrı ayrı atama yapar; bir dalda atamanın unutulması kolaydır. switch ifadesinde ise değer tek bir yerde atanır ve derleyici her olası değerin bir sonucu olduğunu denetler.

7.9 Örnekler

Bu kesimdeki örnekler, karar yapılarını gerçek problemlerde bir araya getirir. Her örnekte önce problem ve çözüm fikri, sonra kod, ardından birkaç girdi için izleme ve programın gerçek çıktısı verilir. Örneklerin çoğunda kodun kendisi kısadır; asıl iş, hangi girdilerin özel davrandığını bulmaktır.

7.9.1 Tek mi çift mi?

Bir tam sayı 2'ye kalansız bölünüyorsa çifttir. Java'da kalan `%` işleciyle bulunur. **Kalan** (*remainder*), bölme işleminden sonra artan miktardır: 17'yi 2'ye bölünce bölüm 8, kalan 1'dir, çünkü $17 = 2 \times 8 + 1$.

```
int sayi = klavye.nextInt();
if (sayi % 2 == 0)
    System.out.println("Çift");
else
    System.out.println("Tek");
```

Koşulda önce `%` hesaplanır, sonra sonuç 0 ile karşılaştırılır; `%`'nin önceliği `==`'den yüksektir. Program 17, −4 ve −7 için izlensin.

sayi	`sayi % 2`	`sayi % 2 == 0`	Çıktı
17	1	false	Tek
−4	0	true	Çift
−7	−1	false	Tek

Tablo 7.17: Tek/çift programının üç çalıştırması.

```
Tek
```

```
Çift
```

```
Tek
```

Üçüncü satır dikkat ister: `-7 % 2` değeri 1 değil, −1'dir. Java'da tam sayı bölmesi sıfıra doğru yuvarlar: *−7 / 2 = −3*'tür, −4 değil. Kalan da bu bölüme göre hesaplanır: *−7 = 2 × (−3) + (−1)*. Sonuçta kalanın işareti her zaman bölünen sayının işaretiyle aynıdır. Şekil 7.17 −4'ten 4'e kadar sayıların 2'ye göre kalanlarını gösterir.



Şekil 7.17: −4'ten 4'e kadar sayıların `sayi % 2` değerleri. Çift sayılarda kalan 0'dır. Pozitif tek sayılarda kalan 1, negatif tek sayılarda −1'dir.

Şekildeki değerler bir programla da üretildi. İkinci satır −7 için iki koşulu, `-7 / 2` bölümünü ve `7 % -2`'yi yazar:

```
public class KalanTablosu {
    public static void main(String[] args) {
        for (int sayi = -4; sayi <= 4; sayi++) {
            System.out.print(sayi % 2 + " ");
        }
        System.out.println();
        System.out.println((-7 % 2 == 1) + " " + (-7 % 2 != 0) + " " + (-7 / 2) + " " + (7
% -2));
    }
}
```

```
0 -1 0 -1 0 1 0 1 0
false true -3 1
```

`-7 % 2 == 1` yanlış, `-7 % 2 != 0` doğrudur. `7 % -2` 'nin 1 çıkması da kuralı doğrular: bölen negatif olsa da kalan, bölünen 7'nin işaretini alır.

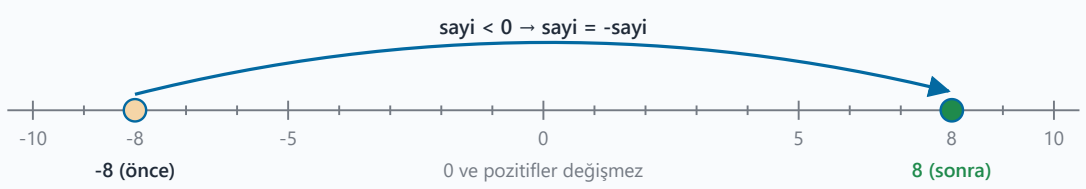
Sık yapılan hata

Tek sayıyı `sayi % 2 == 1` koşuluyla sınamak. Bu koşul pozitif tek sayılarda doğru çalışır, negatif tek sayılarda ise yanlış sonuç verir: −7 "tek değil" sayılır. Güvenli koşullar kalanı 0 ile karşılaştırır: çift için `sayi % 2 == 0`, tek için `sayi % 2 != 0`. Bu iki koşul her int için doğrudur.

7.9.2 Mutlak değer ve taşma

Bir sayının **mutlak değeri** (*absolute value*), sayının sıfıra uzaklığıdır: *|−8| = 8*, *|5| = 5*. Sayı negatifse işareti değiştirilir, değilse olduğu gibi bırakılır. Bu, tek yollu `if` 'in tipik örneğidir: negatif olmayan sayılar için yapılacak bir iş yoktur.

```
int sayi = klavye.nextInt();
if (sayi < 0) {
    sayi = -sayi;
}
System.out.println(sayi);
```



Şekil 7.18: Mutlak değer sayı doğrusunda. −8, sıfıra göre yansıtılarak 8 olur. 0 ve pozitif değerler değişmez.

Girdi	`sayi < 0`	`sayi = -sayi`	Çıktı
−8	true	*−(−8) = 8*	8
5	false	atlanır	5
−2147483648	true	*−(−2147483648)* → taşma → −2147483648	−2147483648

Tablo 7.18: Mutlak değer programının üç çalıştırması. Üçüncü girdi Integer.MIN_VALUE'dur.

```
8
```

```
5
```

```
-2147483648
```

İlk iki satır beklenen sonucu verir. Üçüncü satırdaki sonuç ise şaşırtıcıdır: bir sayının mutlak değeri negatif çıkmıştır. Nedeni int tipinin sınırlarıdır. Bir int 32 bitle saklanır ve $-2^{31} = -2.147.483.648$ ile $2^{31} - 1 = 2.147.483.647$ arasındaki değerleri alabilir. En küçük değerin, `Integer.MIN_VALUE`'nun, pozitif karşılığı 2.147.483.648'dir. Bu sayı en büyük int'ten bir fazladır ve int'e sığmaz.

Tanım 7.13

Taşma (*overflow*), bir işlemin sonucunun, sonucu saklayacak tipin alabileceği değer aralığının dışına çıkmasıdır. Java'da int ve long işlemlerinde taşma bir hata vermez; sonuç aralığın öbür ucuna "sarar". En büyük int'e 1 eklenince en küçük int elde edilir.

−(−2.147.483.648) işleminin doğru sonucu 2.147.483.648'dir. Bu değer en büyük int'i 1 aşar ve aralığın öbür ucuna sarar: sonuç yine −2.147.483.648 olur. Java'nın hazır `Math.abs` metodu da aynı davranır. Bu özel durumu doğru ele almanın iki yolu vardır. Birincisi, hesabı daha geniş bir tipte, long'da yapmaktır. İkincisi, Java 15'ten beri bulunan `Math.absExact` metodunu kullanmaktır; bu metot sessizce yanlış sonuç vermek yerine bir istisna fırlatır.

```
public class MutlakOzel {
    public static void main(String[] args) {
        int sayi = Integer.MIN_VALUE;
        System.out.println(Integer.MAX_VALUE);
        System.out.println(Math.abs(sayi));
        long uzun = sayi;
        if (uzun < 0) uzun = -uzun;
        System.out.println(uzun);
        System.out.println(Math.absExact(sayi));
    }
}
```

```
2147483647
-2147483648
2147483648
Exception in thread "main" java.lang.ArithmeticException: Overflow to represent absolute
value of Integer.MIN_VALUE
        at java.base/java.lang.Math.absExact(Math.java:1903)
        at MutlakOzel.main(MutlakOzel.java:9)
```

İlk satır en büyük int'tir. İkinci satır `Math.abs`'ın da negatif sonuç verdiğini gösterir. Üçüncü satırda değer long'a kopyalanmıştır; long'un aralığı çok daha geniştir ve doğru sonuç, 2147483648, elde edilir. Son satırda `Math.absExact` taşmayı fark eder ve `ArithmeticException` fırlatır. İleti "Integer.MIN_VALUE'nun mutlak değerini göstermek taşmaya yol açar" anlamındadır.

7.9.3 Artık yıl

Dünya Güneş'in çevresindeki bir turu yaklaşık 365,2425 günde tamamlar. Takvim bu fazlalığı artık yıllarla, yani 366 günlük yıllarla dengeler. Gregoryen takvimin kuralı üç basamaklıdır:

1. 4'e bölünebilen yıllar artık yıldır;
2. ama 100'e bölünebilen yıllar artık yıl değildir;
3. ama 400'e bölünebilen yıllar yine artık yıldır.

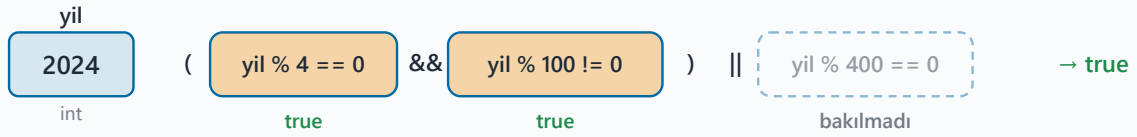
Kural tek bir boolean ifadede toplanabilir: "4'e bölünür ve 100'e bölünmez" ya da "400'e bölünür".

```
int yil = klavye.nextInt();
boolean artik =
    (yil % 4 == 0 && yil % 100 != 0)
    || yil % 400 == 0;
if (artik)
    System.out.println(yil + " artık yıl");
else
    System.out.println(yil + " normal yıl");
```

Parantez, `&&`'li kısmın bir bütün olduğunu gösterir. `&&`'nin önceliği `||`'den yüksek olduğu için parantez olmasa da sonuç değişmezdi; ama parantez kuralın yapısını okura açıkça gösterir. Program dört yıl için izlensin.

yil	`yil % 4 == 0`	`yil % 100 != 0`	Parantez	`yil % 400 == 0`	artik
2024	0 → true	24 → true	true	bakılmadı	true
2023	3 → false	bakılmadı	false	23 → false	false
1900	0 → true	0 → false	false	300 → false	false
2000	0 → true	0 → false	false	0 → true	true

Tablo 7.19: Artık yıl koşulunun dört yıl için değerlendirilmesi. “bakılmadı”, kısa devre yüzünden hesaplanmayan terimdir.



Şekil 7.19: yil = 2024 için koşulun değerlendirilmesi. Parantezin içindeki iki terim de true'dur. Sol taraf true olduğu için `||` kısa devre yapar; `yil % 400 == 0` terimine bakılmaz.

Her satırda iki kısa devre fırsatı vardır. 2024'te parantezin içi doğru çıkar ve `||`'nin sağına bakılmaz (Şekil 7.19). 2023'te ilk terim yanlıştır; `&&` ikinci terimi hesaplamaz. Parantezin içi yanlış olduğu için bu kez `||`'nin sağı hesaplanır. 1900 ile 2000 kuralın ince noktalarıdır. İkisi de 100'e bölünür ve parantezin içi yanlış çıkar. Fark son terimdedir: *1900 % 400 = 300*, *2000 % 400 = 0*. Bu yüzden 2000 artık yıldır, 1900 değildir. Program 2100 için “normal yıl”, 1600 için “artık yıl” yazdı.

```
2024 artık yıl
```

```
2023 normal yıl
```

```
1900 normal yıl
```

```
2000 artık yıl
```

7.9.4 İkinci dereceden denklem

$ax^2 + bx + c = 0$ denkleminin reel köklerinin sayısı diskriminanta bağlıdır: $\Delta = b^2 - 4ac$. Δ pozitifse iki farklı reel kök, sıfırsa tek (çakışık) kök vardır; negatifse reel kök yoktur. Kökler şu formülle bulunur:

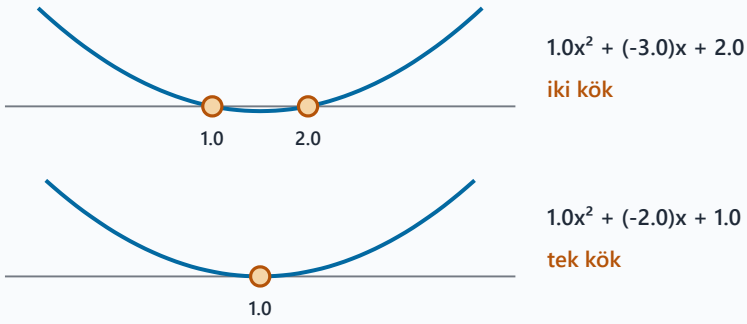
$$x_{1,2} = (-b \pm \sqrt{\Delta}) / (2a)$$

Üç durum üç dallı bir else-if zinciriyle ayrılır. Burada a'nın 0 olmadığı varsayılır; a = 0 olsaydı denklem ikinci dereceden olmazdı.

```
double a = klavye.nextDouble();
double b = klavye.nextDouble();
double c = klavye.nextDouble();
double d = b * b - 4 * a * c;
if (d > 0) {
  double x1 = (-b + Math.sqrt(d)) / (2 * a);
  double x2 = (-b - Math.sqrt(d)) / (2 * a);
  System.out.println(x1 + " ve " + x2);
} else if (d == 0) {
  System.out.println(-b / (2 * a));
} else {
  System.out.println("Reel kök yok");
}
```

a, b, c	d = b² − 4ac	Çalışan dal	Hesap	Çıktı
1.0, −3.0, 2.0	*9.0 − 8.0 = 1.0*	`d > 0`	*x1 = (3.0 + 1.0) / 2.0 = 2.0, x2 = (3.0 − 1.0) / 2.0 = 1.0*	2.0 ve 1.0
1.0, −2.0, 1.0	*4.0 − 4.0 = 0.0*	`d == 0`	*−b / 2a = 2.0 / 2.0 = 1.0*	1.0
1.0, 0.0, 1.0	*0.0 − 4.0 = −4.0*	`else`	—	Reel kök yok

Tablo 7.20: Denklem programının üç çalıştırması. Her satırda farklı bir dal çalışır.



1.0x² + (-3.0)x + 2.0

iki kök

1.0x² + (-2.0)x + 1.0

tek kök



$1.0x^2 + (0.0)x + 1.0$

kök yok

Şekil 7.20: Üç çalıştırmanın parabolleri. (a) Δ = 1.0: parabol x eksenini 1.0 ve 2.0'da keser. (b) Δ = 0.0: parabol eksene 1.0'da teğettir. (c) Δ = −4.0: parabol ekseni kesmez.

```
2.0 ve 1.0
```

```
1.0
```

```
Reel kök yok
```

Şekil 7.20 her çalıştırmanın geometrik anlamını verir. Program $2x^2 - 7x + 3 = 0$ için de denendi: $\Delta = 49 - 24 = 25$, $\sqrt{\Delta} = 5$; kökler $(7 + 5) / 4 = 3.0$ ve $(7 - 5) / 4 = 0.5$'tir. Program "3.0 ve 0.5" yazdı.

`else` dalı yalnızca bir ileti yazmakla kalmaz; programı bir hatadan da korur. Negatif bir sayının kare-kökü reel sayılarda tanımsızdır. Java'da `Math.sqrt(-4.0)` bir istisna fırlatmaz, `NaN` (*Not a Number*, "sayı değil") değerini döndürür. Ondalık sayılarda sıfıra bölme de hata vermez; sonsuzluk (`Infinity`) üretir:

```
public class OndalikOzel {
    public static void main(String[] args) {
        System.out.println(Math.sqrt(-4.0));
        System.out.println(7.0 / 0);
        System.out.println(-7.0 / 0);
        System.out.println(0.0 / 0);
        System.out.println(0.1 + 0.2 == 0.3);
        System.out.println(0.1 + 0.2);
    }
}
```

```
NaN
Infinity
-Infinity
NaN
false
0.30000000000000004
```

`NaN` ile yapılan her hesap yine `NaN` verir ve hata, sonuçlar ekrana yazılana kadar fark edilmez. Kararların sırası bu yüzden önemlidir: `Math.sqrt(d)` yalnızca *d > 0* olduğu bilinen dalda çağrılır.

Son iki satır ondalık sayıların başka bir inceliğini gösterir. `0.1 + 0.2` tam olarak 0.3 değildir; ikili sistemde 0.1 ve 0.2 tam gösterilemez ve küçük bir yuvarlama farkı kalır. `d == 0` karşılaştırması bu örnekte güvenlidir, çünkü girdiler tam sayılardır ve $b^2 - 4ac$ tam olarak hesaplanır. Ölçümden gelen ondalık değerlerde ise `==` yerine küçük bir tolerans kullanılır: `Math.abs(d) < 1e-9`.

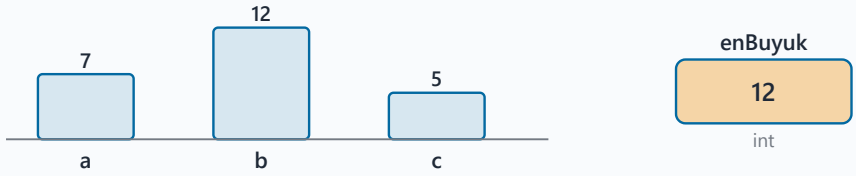
7.9.5 Üç sayının en büyüğü

Üç sayının en büyüğünü bulmanın bir yolu her olasılığı ayrı bir koşulla yazmaktır: “a, b'den ve c'den büyükse …”. Bu yaklaşım üç sayıda bile karmaşıklaşır, beş sayıda içinden çıkılmaz hâle gelir. Daha iyi bir yol “aday ve karşılaştır” kalıbıdır: ilk sayı aday kabul edilir, sonra her sayı adayla karşılaştırılır ve adaydan büyükse aday olur.

```
int a = klavye.nextInt();
int b = klavye.nextInt();
int c = klavye.nextInt();
int enBuyuk = a;
if (b > enBuyuk)
    enBuyuk = b;
if (c > enBuyuk)
    enBuyuk = c;
System.out.println("En büyük: " + enBuyuk);
```

Girdi (a, b, c)	Adım	Koşul	enBuyuk
7, 12, 5	`enBuyuk = a`	—	7
	`b > enBuyuk`	*12 > 7* → true	12
	`c > enBuyuk`	*5 > 12* → false	12
3, 3, 9	`enBuyuk = a`	—	3
	`b > enBuyuk`	*3 > 3* → false	3
	`c > enBuyuk`	*9 > 3* → true	9
8, 2, 4	`enBuyuk = a`	—	8
	`b > enBuyuk`	*2 > 8* → false	8
	`c > enBuyuk`	*4 > 8* → false	8

Tablo 7.21: “Aday ve karşılaştır” kalıbının üç çalıştırması. Son sütun her adımdan sonraki aday değeridir.



Şekil 7.21: (7, 12, 5) girdisinde b ile karşılaştırmadan sonraki durum. Çubuklar a, b ve c'nin büyüklüğünü gösterir; aday enBuyuk = 12 olmuştur.

İkinci girdide *3 > 3* yanlıştır; eşit bir değer adayı değiştirmez. Sonuç yine doğrudur, çünkü eşit değerlerden hangisinin aday olduğu fark etmez. Kalıbın gücü genellenebilmesindedir: n sayı için tam olarak n − 1 karşılaştırma gerekir ve her karşılaştırma aynı biçimdedir. Bu kalıp bir döngünün gövdesine konduğunda istenen sayıda sayının en büyüğünü bulur. Java'nın hazır metoduyla aynı iş tek satırda da yapılabilir: `Math.max(a, Math.max(b, c))`. Tam program iki yöntemin sonucunu da yazar:

```
import java.util.Scanner;

public class EnBuyuk {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        int a = klavye.nextInt();
        int b = klavye.nextInt();
        int c = klavye.nextInt();
        int enBuyuk = a;
        if (b > enBuyuk)
            enBuyuk = b;
        if (c > enBuyuk)
            enBuyuk = c;
        System.out.println("En büyük: " + enBuyuk);
        System.out.println(Math.max(a, Math.max(b, c)));
    }
}
```

```
En büyük: 12
12
```

```
En büyük: -1
-1
```

Sık yapılan hata

Adayı ilk sayı yerine 0 ile başlatmak: `int enBuyuk = 0;`. Girdiler pozitifken bu da doğru sonuç verir. Ama (−2, −7, −1) girdisinde hiçbir sayı 0'dan büyük değildir; aday hiç değişmez ve girdilerden biri bile olmayan 0 yazılır. Doğru program bu girdide −1 yazar. Aday her zaman girdilerden biriyle başlatılır.

7.9.6 switch ile hesap makinesi

Kullanıcı iki sayı ile aralarına bir işlem işareti girsin: `8.0 * 2.5`. Program işareti dizgi olarak okur ve switch ile ilgili işlemi seçer. `Scanner`'ın `next()` metodu boşluğa kadar olan sözcüğü dizgi olarak okur; bu yüzden girdide sayılar ile işaret arasında boşluk bulunur.

```
double x = klavye.nextDouble();
String op = klavye.next();
double y = klavye.nextDouble();
switch (op) {
  case "+" -> System.out.println(x + y);
  case "-" -> System.out.println(x - y);
  case "*" -> System.out.println(x * y);
  case "/" -> {
    if (y == 0) System.out.println("Hata!");
    else System.out.println(x / y);
  }
  default -> System.out.println("Geçersiz");
}
```

Girdi	op	Eşleşen case	Ek koşul	Çıktı
8.0 * 2.5	"*"	`case "*"`	—	20.0
7.0 / 0.0	"/"	`case "/"`	`y == 0` → true	Hata!
5.0 % 3.0	"%"	yok → `default`	—	Geçersiz

Tablo 7.22: Hesap makinesinin üç çalıştırması.



Şekil 7.22: 8.0 * 2.5 girdisinde switch, op = "*" değerini taşıyan case'e gider ve 20.0 yazar.

```
20.0
```

```
Hata!
```

```
Geçersiz
```

Bölme dalı tek bir ifade değil, içinde if-else bulunan bir bloktur. Bu bir switch deyimidir, değer üretmez; bu yüzden blokta `yield` gerekmez. Bloktaki denetim gereklidir, çünkü ondalık sayılarda sıfıra bölme bir hata vermez: `7.0 / 0` ifadesi `Infinity` üretir. Kullanıcıya "Infinity" yazmak yerine anlaşılır bir ileti vermek için bölmeden önce bölen sınanır. Üçüncü çalıştırmada "%" işareti hiçbir case'le eşleşmez; `default` beklenmeyen girdiyi yakalar. String üzerinde switch, eşleşmeyi `equals` ile yapar; dizgilerde `==` kullanmanın yol açtığı referans sorunu burada ortaya çıkmaz.

Program ayrıca `9 - 4`, `7 / 2` ve `3 + 4.5` için denendi; 5.0, 3.5 ve 7.5 yazdı. x ve y double olduğu için `7 / 2` tam sayı bölmesi değildir ve 3.5 verir.

7.9.7 Alıştırmalardan çözümlü örnekler

Aşağıdaki örnekler sunumun alıştırmalarından seçilmiştir. Her biri bu notta anlatılan bir kalıbı uygular.

Örnek 7.2

Girilen tam sayının pozitif, negatif ya da sıfır olduğunu yazan program.

Çözüm. Üç seçenek birbirini dışlar; bir sayı yalnızca birine girer. Bu, else-if zincirinin işidir. Üçüncü durum için koşul yazmaya gerek yoktur: sayı ne pozitif ne negatifse sıfırdır.

```
import java.util.Scanner;

public class Isaret {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        int sayi = klavye.nextInt();
        if (sayi > 0)
            System.out.println("Pozitif");
        else if (sayi < 0)
            System.out.println("Negatif");
        else
            System.out.println("Sıfır");
    }
}
```

12 için "Pozitif", −3 için "Negatif", 0 için "Sıfır" yazar. 0 bir sınır değeridir ve ayrıca denenmiştir: *0 > 0* ve *0 < 0* yanlıştır, else çalışır.

Örnek 7.3

Boy (metre) ve kilodan vücut kitle indeksini, *VKİ = kilo / boy²*, hesaplayıp sınıfını yazan program. Sınırlar: 18,5'in altı zayıf, 18,5 ile 25 arası normal, 25 ve üstü fazla kilolu.

Çözüm. Sınıflar sürekli bir aralığı böler; switch uygun değildir. Eşikler küçükten büyüğe sıralandığı için koşullar `<` ile yazılır ve zincir en küçük eşikten başlar.

```
import java.util.Scanner;

public class Vki {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        double boy = klavye.nextDouble();
        double kilo = klavye.nextDouble();
        double vki = kilo / (boy * boy);
        if (vki < 18.5)
            System.out.println("Zayıf");
        else if (vki < 25)
            System.out.println("Normal");
        else
            System.out.println("Fazla kilolu");
    }
}
```

Boy 1,75 m ve kilo 70 kg için *VKİ = 70 / 3,0625 ≈ 22,86*'dır ve "Normal" yazılır. 1,60 m ve 45 kg için *45 / 2,56 ≈ 17,58*, "Zayıf"; 1,80 m ve 90 kg için *90 / 3,24 ≈ 27,78*, "Fazla kilolu" yazılır. Programın girdisi, Java kaynak kodundaki gibi ondalık noktayla verilmiştir: 1.75. Bilgisayarın bölge ayarı Türkçe ise `Scanner` ondalık virgül bekleyebilir; bu durumda 1,75 yazılır.

Örnek 7.4

Üç kenar uzunluğu bir üçgen oluşturuyor mu? Oluşturuyorsa üçgen eşkenar mı, ikizkenar mı, çeşitkenar mı?

Çözüm. Üç kenarın bir üçgen oluşturması için her iki kenarın toplamı üçüncüden büyük olmalıdır (üçgen eşitsizliği). Önce bu koşulun tersi denetlenir: herhangi bir toplam üçüncü kenara eşit ya da ondan küçükse üçgen yoktur. De Morgan kuralına göre "üç eşitsizliğin hepsi doğru" koşulunun tersi "en az biri yanlış"tır; bu yüzden terimler `||` ile bağlanır. Ardından üçgen türü dar koşuldan geniş koşula doğru sınanır: önce üç kenarın eşitliği, sonra herhangi iki kenarın eşitliği.

```java
import java.util.Scanner;

public class Ucgen {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        int a = klavye.nextInt(), b = klavye.nextInt(), c = klavye.nextInt();
        if (a + b <= c || a + c <= b || b + c <= a)
            System.out.println("Üçgen değil");
        else if (a == b && b == c)
            System.out.println("Eşkenar");
        else if (a == b || b == c || a == c)
            System.out.println("İkizkenar");
        else
            System.out.println("Çeşitkenar");
    }
}
```

Program (3, 3, 3) için Eşkenar, (3, 3, 5) ve (5, 3, 5) için İkizkenar, (3, 4, 5) için Çeşitkenar yazdı. (1, 2, 3) için *1 + 2 = 3* olduğundan "Üçgen değil" yazdı; bu kenarlar düz bir çizgi oluşturur. (2, 9, 4) için de *2 + 4 = 6 < 9* olduğundan aynı sonuç çıktı. Eşkenar koşulu ikizkenardan önce sorulmalıdır, çünkü her eşkenar üçgen ikizkenar koşulunu da sağlar.

Örnek 7.5

1–7 arasındaki gün numarasını gün adına çeviren, geçersiz girdide uyarı veren program.

Çözüm. Tek bir tam sayının yedi ayrık değeri vardır; switch ifadesi tam bu iş içindir. Geçersiz girdi `default` ile yakalanır. switch ifadesi olduğu için `default` zaten zorunludur.

```
import java.util.Scanner;

public class GunAdi {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        int gun = klavye.nextInt();
        String ad = switch (gun) {
            case 1 -> "Pazartesi";
            case 2 -> "Salı";
            case 3 -> "Çarşamba";
            case 4 -> "Perşembe";
            case 5 -> "Cuma";
            case 6 -> "Cumartesi";
            case 7 -> "Pazar";
            default -> "Geçersiz gün numarası";
        };
        System.out.println(ad);
    }
}
```

1 için Pazartesi, 5 için Cuma, 7 için Pazar yazdı; 8 ve 0 için “Geçersiz gün numarası” yazdı.

Örnek 7.6

İki sayının küçüğünü koşul işleciyle bulan, sonra bunu üç sayıya genişleten program.

Çözüm. İki sayının küçüğü `a < b ? a : b`'dir. Üç sayı için önce ilk ikisinin küçüğü bulunur, sonra bu değer üçüncüyle karşılaştırılır. İki ayrı değişken kullanmak tek satırlık iç içe bir koşul işlecinden daha okunurdur.

```
import java.util.Scanner;

public class EnKucuk {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        int a = klavye.nextInt(), b = klavye.nextInt(), c = klavye.nextInt();
        int ikili = a < b ? a : b;
        int uclu = ikili < c ? ikili : c;
        System.out.println(ikili + " " + uclu);
    }
}
```

(7, 12, 5) için ilk ikisinin küçüğü 7, üçünün küçüğü 5'tir; program “7 5” yazdı. (4, 4, 9) için “4 4”, (−1, 8, −6) için “−1 −6” yazdı.

Örnek 7.7

Harf notu programına "not 0–100 aralığının dışında" denetimi eklenecek. Bu denetim zincirin neresine yazılmalıdır?

Çözüm. Zincirin en başına. Denetim sona konsaydı geçersiz bir not, örneğin 105, ilk koşulu (*105 >= 90*) sağlar ve A alırdı; denetime hiç sıra gelmezdi. −5 de benzer biçimde F alırdı. Geçersiz durumları zincirin başında ayıklamak, sonraki koşulların yalnızca geçerli notlarla çalışmasını sağlar.

```
import java.util.Scanner;

public class HarfNotuSinir {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        int not = klavye.nextInt();
        if (not < 0 || not > 100)
            System.out.println("Geçersiz not");
        else if (not >= 90)
            System.out.println('A');
        else if (not >= 80)
            System.out.println('B');
        else if (not >= 70)
            System.out.println('C');
        else
            System.out.println('F');
    }
}
```

Program 105 ve −5 için "Geçersiz not", 100 ve 95 için A, 75 için C, 0 için F yazdı. Sınır değerleri 0 ve 100 geçerli kabul edilir; koşulda `<` ve `>` kullanıldığı için bu iki değer "geçersiz" dalına düşmez.

7.10 Özet

- Her algoritma üç akışla yazılabilir: **sıralı yapı**, **karar yapısı** ve **döngü yapısı**. Sıralı yapıda deyimler yazıldıkları sırayla birer kez çalışır; her deyim öncekilerin bıraktığı değerleri kullanır.
- **Koşul**, `true` ya da `false` değeri üreten boolean bir ifadedir. Eşitlik `==` ile sınanır; `=` atamadır ve `if (x = 5)` derlenmez. `10 < x < 20` yazımı da derlenmez; doğrusu `x > 10 && x < 20`'dir.
- `if` koşul doğruysa gövdesini çalıştırır; **if-else** iki daldan tam olarak birini çalıştırır. Süslü parantez yoksa gövde tek deyimdir; derleyici girintiye bakmaz. `if (…);` satırındaki noktalı virgül **boş deyimdir**.

- **else-if zincirinde** koşullar yukarıdan aşağıya denenir; ilk doğru dal çalışır, gerisi atlanır. Alt sınırla kurulan zincirde eşikler büyükten küçüğe yazılır. Bağımsız if'lerin ise hepsi denetlenir.
- Son `else` silinirse, dallarda değer alan bir değişken "değer verilmemiş olabilir" hatasına yol açar. Java değer almamış yerel değişkenin okunmasına izin vermez.
- **İç içe kararda** iç koşul yalnızca dış koşul o dalı seçtiğinde denetlenir. Derin iç içelik `&&` ile düzleştirilebilir. **Sarkan else** en yakın, else'i olmayan `if`'e bağlanır; belirsizlik süslü parantezle giderilir.
- `&&` iki tarafın da, `||` en az bir tarafın doğru olmasını ister; `!` tersine çevirir. İkisi de **kısa devre** yapar: sonuç sol taraftan belliyse sağ taraf hesaplanmaz. Bu sayede `b != 0 && a / b > 1` güvenlidir. Tek karakterli `&` ve `|` iki tarafı da her zaman hesaplar.
- **De Morgan kuralları**: `!(a && b)` = `!a || !b`, `!(a || b)` = `!a && !b`. Karşılaştırmaların tersleri: `>=` ↔ `<`, `>` ↔ `<=`, `==` ↔ `!=`.
- **Koşul işleci** `koşul ? d1 : d2` bir değer üreten ifadedir; kısa, iki seçenekli değer seçimlerinde kullanılır ve sağdan gruplanır.
- Dizgilerde `==` referansı, `equals` içeriği karşılaştırır. Klavyeden okunan bir dizgi, aynı metni taşıyan sabitle `==` karşılaştırmasında `false` verir. Dizgiler her zaman `equals` ile karşılaştırılır.
- **switch** tek bir değerin ayrık seçeneklerine göre seçim yapar; case etiketleri sabittir. **switch ifadesi** bir değer üretir ve bütün değerleri kapsamak zorundadır; int ve String için `default` zorunludur. Bloklu dalın değeri `yield` ile verilir.
- Klasik switch'te `break` yoksa yürütme sonraki case'in koduna **düşer**. Ok biçiminde (`->`) aşağı düşme olmaz. Aralık ve karmaşık koşullarda if-else, tek değerin ayrık seçeneklerinde switch uygundur.
- Örneklerin dersleri: çift sınaması `% 2 == 0` ile yapılır, çünkü Java'da `-7 % 2 = -1`'dir. `Integer.MIN_VALUE`'nun mutlak değeri int'te **taşar**. `Math.sqrt` negatif sayıda `NaN`, ondalık sıfıra bölme `Infinity` verir. "Aday ve karşılaştır" kalıbında aday girdilerden biriyle başlatılır.

7.11 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

akış *control flow* · kontrol yapısı *control structure* · sıralı yapı *sequence structure* · karar yapısı *selection structure* · döngü yapısı *repetition structure* · deyim *statement* · ifade *expression* · genişletme *widening conversion* · izleme tablosu *trace table* · koşul *condition* · boolean (mantıksal tip) *boolean* · karşılaştırma işleci *comparison operator* · if-else yapısı *if-else statement* · dal *branch* · blok *block* · süslü parantez *curly braces* · boş deyim *empty statement* · çok yollu karar *multi-way decision* · else-if zinciri *else-if chain* · iç içe karar *nested decision* · sarkan else *dangling else* · mantıksal işleç *logical operator* · kısa devre değerlendirme *short-circuit evaluation* · De Morgan kuralları *De Morgan's laws* · işleç önceliği *operator precedence* · istisna *exception*

koşul işleci *conditional (ternary) operator* · referans *reference* · dizgi sabiti *string literal* · dizgi havuzu *string pool* · heap *heap* · switch ifadesi *switch expression* · switch deyimi *switch statement* · case etiketi *case label* · ok biçimi *arrow form (->)* · aşağı düşme *fall-through* · kalan *remainder* · mutlak değer *absolute value* · taşma *overflow* · derleme hatası *compile error*

7.12 Kendini Dene

1. p = 4 ve q = 9 iken `if (p > 3 || ++q > 0) p = p * 2;` satırı çalışıyor. Ardından p ve q kaçtır?

   Cevap

   Sol taraf *4 > 3* doğrudur. `||` kısa devre yapar ve `++q` hiç çalışmaz; q 9 kalır. Koşul doğru olduğu için *p = 4 × 2 = 8* olur. Sonuç: p = 8, q = 9.

2. t = 15 iken `t % 3 == 0 && t % 5 == 0 ? "FizzBuzz" : t % 3 == 0 ? "Fizz" : "diğer"` ifadesinin değeri nedir? t = 9 ve t = 7 için ne olur?

   Cevap

   Koşul işleci sağdan gruplanır; ifade “ilk koşul doğruysa FizzBuzz, değilse (ikinci koşul doğruysa Fizz, değilse diğer)” biçiminde okunur. 15 hem 3'e hem 5'e bölünür: “FizzBuzz”. 9, 3'e bölünür ama 5'e bölünmez: ilk koşul yanlış, ikinci doğru, “Fizz”. 7 ikisine de bölünmez: “diğer”.

3. Aşağıdaki kod puan = 72 için ne atar? puan = 90 ve puan = 40 için ne olur?

```
if (puan >= 50) {
    if (puan >= 85) d = "Pekiyi";
    else d = "Geçti";
} else d = "Kaldı";
```

   Cevap

   72 için dış koşul *72 >= 50* doğrudur, iç koşul *72 >= 85* yanlıştır: d = "Geçti". 90 için iki koşul da doğrudur: "Pekiyi". 40 için dış koşul yanlıştır, iç if hiç denetlenmez: "Kaldı". Süslü parantez iç `else`'i iç `if`'e, dıştaki `else`'i dış `if`'e bağlar.

4. `if (!(sicaklik < 15 || yagmur))` koşulu, `!` işleci yalnızca boolean değişkenin önünde kalacak

biçimde nasıl yazılır? sicaklik = 30, yagmur = false için koşul doğru mudur?

Cevap

De Morgan kuralıyla `||`, `&&`'ye döner ve her terim tersine çevrilir: `sicaklik >= 15 && !yagmur`. `<`'nin tersi `>=`'dir. 30 ve false için *30 >= 15* doğru, `!false` doğrudur; koşul doğrudur. Özgün yazımda da *30 < 15* yanlış, yagmur yanlış, VEYA'ları yanlış, tersi doğrudur.

5. Klasik bir switch'te `case 1: puan += 10;`, `case 2: puan += 20; break;`, `case 3: puan += 30;`, `default: puan += 5;` satırları var. puan her seferinde 0'dan başlıyor. k = 1, 2, 3 ve 4 için puan kaç olur?

Cevap

k = 1: case 1'de break yoktur, case 2'ye düşülür: *10 + 20 = 30*, sonra break. k = 2: yalnızca 20. k = 3: case 3'te break yoktur, default'a düşülür: *30 + 5 = 35*. k = 4: hiçbir case eşleşmez, default: 5. Sonuç: 30, 20, 35, 5.

6. `char not = 'B';` iken `switch (not) { case 'A', 'B' -> "iyi"; case 'C' -> "orta"; default -> "zayıf"; }` ifadesinin değeri nedir? Bu ifadeden `default` satırı silinirse ne olur?

Cevap

'B', ilk case'in listesindedir: değer "iyi"dir. Ok biçiminde aşağı düşme olmaz; ikinci case çalışmaz. default silinirse program derlenmez: char 65 binden fazla değer alabilir ve switch ifadesi bütün değerleri kapsamalıdır. Derleyici “the switch expression does not cover all possible input values” iletisini verir.

7. a = 5, b = 0 iken `if (b == 0 || a / b > 2)` ve `if (b == 0 | a / b > 2)` satırları ne yapar?

Cevap

İlk satırda sol taraf doğrudur; `||` kısa devre yapar ve bölme hiç hesaplanmaz. Koşul doğrudur ve gövde çalışır. İkinci satırda `|` iki tarafı da hesaplar: `5 / 0` tam sayı bölmesi `ArithmeticException: / by zero` istisnasını fırlatır ve program durur.

8. Kullanıcı klavyeden “Ada” yazıyor ve bu metin `ad` değişkenine okunuyor. `ad == "Ada"`, `ad.equals("ada")` ve `ad.equalsIgnoreCase("ada")` ifadelerinin değerleri nedir?

Cevap

`ad == "Ada"` false'tur: klavyeden okunan dizgi yeni bir nesnedir, sabit ise havuzdadır; referanslar farklıdır. `ad.equals("ada")` false'tur: içerikler büyük-küçük harf açısından farklıdır (“A” ≠ “a”). `ad.equalsIgnoreCase("ada")` true'dur: harf büyüklüğü göz ardı edilir.

7.13 İleri okuma

- Oracle, *The Java Tutorials*, “The if-then and if-then-else Statements” ve “The switch Statement” sayfaları: Java'nın karar yapıları resmî örneklerle ve kısa açıklamalarla anlatılır.
- Oracle, *Java Language Specification, Java SE 21 Edition*, Bölüm 14 (“Blocks, Statements, and Patterns”) ve 15.28 (“switch Expressions”): sarkan else kuralının, switch ifadesinin kapsama zorunluluğunun ve `yield`'in kesin tanımları.
- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), Kesim 1.3: koşullar ve döngüler kısa, bilimsel hesap örnekleriyle anlatılır; ikinci dereceden denklem örneği burada da işlenir.
- A. B. Downey, C. Mayfield, *Think Java* (2. baskı, O'Reilly, 2019), “Conditionals and Logic” bölümü: koşullar, mantıksal işleçler ve iç içe kararlar programlamaya yeni başlayanlar için sade bir dille anlatılır; kitabın çevrim içi sürümü ücretsizdir.

7.14 Kaynaklar

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11. baskı, Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13. baskı, Pearson, 2024.
3. Oracle, *The Java Tutorials: The if-then and if-then-else Statements; The switch Statement*.
4. Oracle, *Java Language Specification, Java SE 21 Edition*, Bölüm 14: Blocks, Statements, and Patterns.
5. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 1.3 Conditionals and Loops.
6. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019 (ücretsiz çevrim içi sürüm).

KISIM III · BÖLÜM 8

Döngüler

Bilgisayarın asıl gücü, aynı işi yorulmadan binlerce kez yapabilmesinden gelir. Bir programda bu tekrarı döngüler sağlar. Not, önce her döngüde bulunan dört parçayı tanıtır: ilk değer verme, koşul, gövde ve güncelleme. Ardından Java'nın üç döngüsü gelir: `for`, `while` ve `do-while`. Her biri küçük bir programla tur tur izlenir. Döngünün akışını değiştiren `break` ve `continue` deyimleri, sonsuz döngü ve sınır hataları gibi tipik yanlışlar da ayrıntılı ele alınır. Notun ikinci yarısı örneklere ayrılmıştır. Sayının rakamlarıyla çalışan döngüler, bölen arayan döngüler, kullanıcıdan girdi okuyan döngüler ve iç içe döngülerle çizilen desenler birkaç ortak kalıpta toplanır. Bütün Java programları JDK 21 ile derlenip çalıştırılmıştır; notta verilen çıktılar bu çalıştırmalardan alınmıştır.

Bu bölümün kodu: `github.com/sercankulcu/computer-programming-java/Bolum08/src`

ÖĞRENME ÇIKTILARI

Döngü
Bir döngünün ilk değer verme, koşul, gövde ve güncelleme parçalarını kodda göstermek; `for`, `while` ve `do-while` arasından probleme uygun olanı seçmek; `break`, `continue` ve etiketli `break` deyimlerinin akışı nasıl değiştirdiğini açıklamak.

Kalıplar
Sayaç ve biriktirici kullanarak toplam, çarpım ve sayım yapmak; rakam kalıbıyla (`% 10`, `/ 10`) basamak sayısı, rakamlar toplamı, ters çevirme ve palindrom problemlerini çözmek; bölen kalıbıyla asal ve mükemmel sayıları bulmak.

İç içe
İç içe döngülerle satır-sütun düzenleri ve yıldız ya da sayı desenleri üretmek; bir desenin her satırındaki boşluk ve karakter sayısını satır numarasına bağlayan formülü bulmak.

İzleme
Bir döngüyü izleme tablosuyla tur tur yürütmek; sonsuz döngü, sınır hatası ve `double` sayaç hatası gibi yanlışları bulmak; döngüyü normal, sıfır, bir, negatif ve uç değerlerle test etmek.

8.1 Döngü nedir?

8.1.1 Tekrar neden gerekir?

1'den 4'e kadar sayıların toplamını yazdıran bir program düşünelim. Tek satırlık bir çözüm akla hemen gelir: `toplam = 1 + 2 + 3 + 4;`. Aynı iş 1'den 1000'e kadar istendiğinde bu yol işlemez. Bin terimlik bir satır yazmak hem zahmetlidir hem de hataya açıktır. Üstelik üst sınır çoğu zaman program yazılırken bilinmez; kullanıcı onu çalışma sırasında girer.

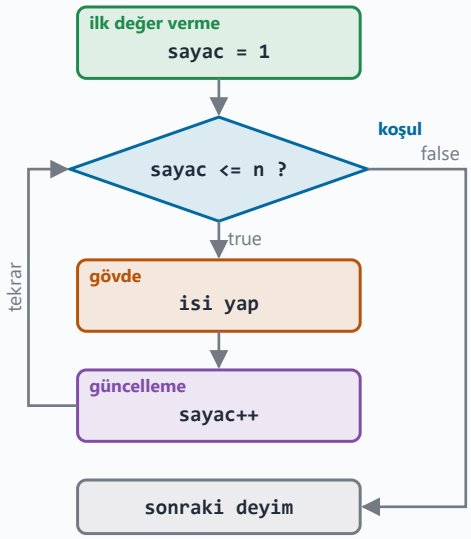
Bu problemlerin ortak yanı, aynı küçük işin art arda yapılmasıdır: "toplama bir sayı daha ekle, sonra sıradaki sayıya geç". Programlama dillerinde bu tür tekrarı anlatan yapıya döngü denir. Programcı tekrarlanacak işi bir kez yazar ve ne zaman duracağını söyler. Gerisini bilgisayar yapar.

> **Tanım 8.1**
>
> **Döngü** (*loop*), bir grup deyimi bir koşul doğru olduğu sürece tekrar tekrar çalıştıran yapıdır. Tekrarlanan deyimlere **döngü gövdesi** (*loop body*), gövdenin bir kez çalışmasına **tur** (*iteration*) denir.

Java'da üç döngü vardır: `for`, `while` ve `do-while`. Üçü de aynı işi yapabilir; aralarındaki fark, koşulun ne zaman sorulduğu ve parçaların nereye yazıldığıdır. Bu yüzden önce her döngüde ortak olan parçaları tanımak gerekir.

8.1.2 Döngünün dört parçası

Her döngü dört parçadan oluşur. Sunumda ve bu notta her parça hep aynı renkle gösterilir. Şekil 8.1 bu parçaları genel bir akış diyagramında toplar.



Şekil 8.1: Genel döngü akışı. İlk değer verme bir kez çalışır. Koşul doğru oldukça gövde ve güncelleme çalışır, ardından ok koşula döner. Koşul yanlış olunca döngüden sonraki deyime geçilir.

1. **İlk değer verme** (*initialization*): döngüden önce bir kez çalışır ve döngünün kullanacağı değişkenlere başlangıç değeri verir. Şekildeki `sayac = 1` böyle bir adımdır.
2. **Koşul**: sonucu `true` ya da `false` olan bir mantıksal ifadedir. `for` ve `while` döngülerinde her turdan önce, `do-while` döngüsünde her turdan sonra sorulur. Sonuç `false` ise döngüden çıkılır.
3. **Gövde**: her turda yapılan iştir. Şekilde "işi yap" kutusuyla gösterilmiştir.
4. **Güncelleme**: koşulda kullanılan değişkeni değiştirir. Örneğin `sayac++` sayacı 1 artırır. Güncellemenin görevi koşulu her turda sona biraz daha yaklaştırmaktır. Güncelleme unutulursa koşul hep aynı kalır ve döngü bitmez.

Şekildeki koşul `sayac <= n ?` sorusudur. sayac 1'den başlar ve her turda 1 artar. Sonunda n'yi geçer ve koşul yanlış olur. Döngünün bitmesini güvenceye alan şey bu ilerlemedir. Bir döngü yazılırken dört parçanın her birinin nerede olduğu açıkça görülebilmelidir. Parçalardan biri eksik ya da yanlışsa döngü ya hiç çalışmaz ya bir tur eksik/fazla çalışır ya da hiç bitmez.

Tanım 8.2

Sayaç (*counter*), her turda sabit bir miktar (çoğunlukla 1) değişen ve kaç tur döndüğünü ya da sıradaki elemanı gösteren değişkendir. **Biriktirici** (*accumulator*), her turda yeni bir değeri kendisine ekleyen ya da kendisiyle çarpan değişkendir. Toplam biriktiricisi 0'dan, çarpım biriktiricisi 1'den başlar.

Başlangıç değerlerinin nedeni basittir. 0, toplamada etkisiz elemandır: 0'a bir sayı eklemek o sayıyı verir. Çarpmada etkisiz eleman 1'dir. Çarpım biriktiricisi yanlışlıkla 0 ile başlatılırsa her tur 0 ile çarptığı için sonuç hep 0 çıkar.

8.2 for döngüsü

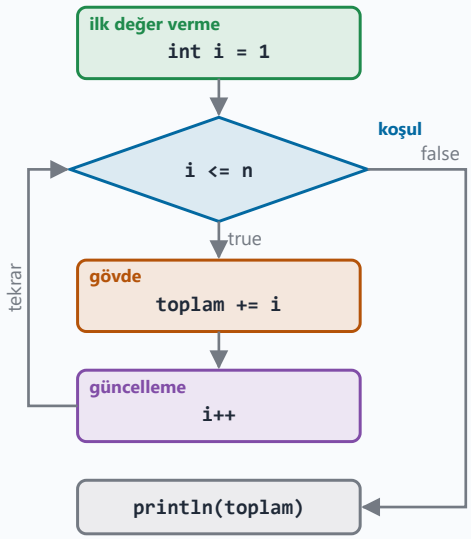
8.2.1 Yazım ve çalışma sırası

Tekrar sayısı baştan belli olduğunda en uygun döngü `for` döngüsüdür. Dört parçanın üçü, yani ilk değer verme, koşul ve güncelleme, aynı satırda, `for` sözcüğünden sonraki parantezin içinde yazılır. Parçalar noktalı virgülle ayrılır. Gövde süslü parantezlerin arasına gelir. Aşağıdaki program 1'den n'ye kadar sayıları toplar:

```
int n = 4;
int toplam = 0;
for (int i = 1; i <= n; i++) {
    toplam += i;
}
System.out.println(toplam);
```

Program çalıştırıldığında ekrana yalnızca toplam yazılır:

```
10
```



Şekil 8.2: for döngüsünün akış diyagramı. Başlıktaki üç parça diyagramda ayrı kutulara dağılır: `int i = 1` ilk değer verme, `i <= n` koşul, `i++` güncellemedir. Gövde `toplam += i` deyimidir.

Şekil 8.2 bu programın akışını gösterir. for döngüsünün parçaları kodda yan yana yazılsa da yazıldıkları sırayla çalışmaz. Gerçek çalışma sırası şöyledir:

1. `int i = 1` çalışır. Bu adım yalnızca bir kez, döngüye girerken yapılır.

2. Koşul `i <= n` sorulur. Doğruysa gövdeye girilir, yanlışsa döngü biter.
3. Gövde `toplam += i` çalışır.
4. Güncelleme `i++` çalışır ve akış 2. adıma, yani koşula döner.

Kısaca sıra "ilk değer verme → koşul → gövde → güncelleme → koşul → gövde → güncelleme …" biçimindedir. Güncelleme parantezin içinde gövdeden önce yazılır ama her zaman gövdeden sonra çalışır. Bu ayrıntı döngüyü elle izlerken çok önemlidir.

Başlıkta tanımlanan `i` değişkeninin bir özelliği daha vardır: yalnızca döngünün içinde yaşar. Bir değişkenin kullanılabildiği program bölgesine **kapsam** (*scope*) denir. `for (int i = 1; …)` biçiminde tanımlanan i'nin kapsamı döngünün kendisidir. Döngüden sonra `System.out.println(i);` yazılırsa derleyici i'yi tanımaz ve hata verir. Döngüden sonra i'nin değeri gerekiyorsa i döngüden önce tanımlanmalıdır: `int i; for (i = 1; i <= n; i++) …`.

8.2.2 for döngüsünü adım adım izlemek

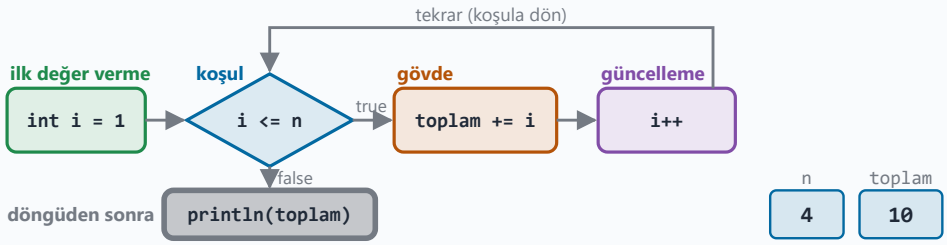
Bir döngünün ne yaptığını anlamanın en güvenilir yolu, onu kâğıt üzerinde bilgisayar gibi çalıştırmaktır. Her turda değişkenlerin aldığı değerler bir tabloya yazılır. Bu tabloya **izleme tablosu** (*trace table*) denir. Tablo 8.1 yukarıdaki programın izleme tablosudur. Tablo, programın aynısı JDK 21 ile çalıştırılıp her turda değişkenler yazdırılarak üretilmiştir.

Tur	i	i <= n	toplam (gövdeden sonra)
1	1	`true`	1
2	2	`true`	3
3	3	`true`	6
4	4	`true`	10
—	5	`false`	10 (döngü biter)

Tablo 8.1: `for` ile 1 + 2 + 3 + 4 (n = 4). Son satır, döngüyü bitiren koşul denetimidir.

Tablo şöyle okunur. Döngüye girilirken `int i = 1` çalışır. 1. turda koşul *1 <= 4* doğrudur ve gövde *toplam = 0 + 1 = 1* yapar. Ardından `i++` ile i 2 olur ve akış koşula döner. 2. turda *2 <= 4* doğrudur ve *toplam = 1 + 2 = 3* olur. Aynı biçimde 3. turda toplam 6, 4. turda 10 olur. 4. turun sonundaki `i++` i'yi 5 yapar. Koşul *5 <= 4* yanlıştır ve döngü biter. Döngüden sonraki `println` 10 yazar.

Tablodan iki sayım çıkar. Gövde 4 kez çalıştı, koşul ise 5 kez soruldu. Koşulun son, yanlış sonuçlanan denetimi de bir değerlendirmedir. Genel olarak, `i` 1'den n'ye kadar artan bir for döngüsünde gövde n kez, koşul n + 1 kez çalışır. İlk değer verme ise yalnızca 1 kez çalışır.



Şekil 8.3: İzlemenin son adımı. Koşul yanlış olduğu için akış "döngüden sonra" kutusuna geçmiştir; n = 4 ve toplam = 10'dur. i artık görünmez, çünkü kapsamı döngüyle birlikte sona ermiştir.

Sık yapılan hata

Biriktiriciyi döngünün içinde tanımlamak. `int toplam = 0;` satırı gövdenin içine taşınırsa iki sorun çıkar. Birincisi, her tur toplamı yeniden 0 yapar ve önceki turların emeği kaybolur. İkincisi, toplamın kapsamı gövdeyle sınırlı kalır ve döngüden sonraki `println(toplam)` derlenmez. Biriktirici her zaman döngüden önce tanımlanır ve başlangıç değerini orada alır.

8.2.3 for başlığının esnek biçimleri

for başlığının üç parçası her zaman "1'den başla, 1 artır" biçiminde olmak zorunda değildir. Sayaç herhangi bir değerden başlayabilir, ikişer artabilir ya da geriye doğru sayabilir. İlk değer verme bölümünde aynı tipte birden çok değişken tanımlanabilir; güncelleme bölümüne de virgülle ayrılmış birden çok ifade yazılabilir. Tablo 8.2 üç örnek başlığı ve ürettikleri değerleri verir.

Başlık	Çalışma
`for (int i = 0, j = 9; i < j; i++, j--)`	i ve j birlikte ilerler: (0,9) (1,8) (2,7) (3,6) (4,5); 5 tur
`for (int i = 0; i <= 10; i += 2)`	0 2 4 6 8 10
`for (int i = 5; i >= 1; i--)`	geri sayım: 5 4 3 2 1

Tablo 8.2: for başlığının esnek biçimleri (değerler JDK 21 ile çalıştırılarak alınmıştır).

İlk satırda iki sayaç vardır: i baştan, j sondan ilerler. Her turda i 1 artar, j 1 azalır. Koşul `i < j`, iki sayaç karşılaştığında döngüyü durdurur. (4,5) turundan sonra i = 5 ve j = 4 olur; *5 < 4* yanlıştır. Bu kalıp, bir dizinin iki ucundan ortaya doğru ilerleyen algoritmalarda sık kullanılır.

Üç bölümün her biri isteğe bağlıdır. Hatta üçü birden boş bırakılabilir. Koşul bölümü boşsa Java onu `true` sayar ve döngü kendiliğinden bitmez. Böyle bir döngüde çıkışı gövdedeki bir `break` sağlamalıdır:

```
int k = 0;
for (;;) {            // üç bölüm de boş: koşul true sayılır
    k++;
    if (k == 3) break;  // çıkış koşulu şart
}
System.out.println(k);   // 3
```

k her turda 1 artar ve 3 olduğunda `break` döngüyü bitirir. Program 3 yazar. Bölümlerin boş bırakılabilmesi bir kolaylıktır ama okunabilirliği azaltır. Sayaçlı bir iş için üç bölümü dolu, sıradan bir for başlığı her zaman daha anlaşılırdır.

8.2.4 i++ ile ++i

Java'da bir değişkeni 1 artırmanın iki kısa yolu vardır: `i++` (sonra artırma) ve `++i` (önce artırma). İkisi de i'yi 1 artırır. Fark, ifadenin *değerinde*dir. `i++` ifadesi i'nin eski değerini verir, sonra i'yi artırır. `++i` önce i'yi artırır, sonra yeni değeri verir. Tablo 8.3 bu farkı dört küçük örnekle gösterir.

Başlangıç	İfade	Sonuç	Sonra
a = 5	`b = a++`	b = 5	a = 6
a = 5	`b = ++a`	b = 6	a = 6
idx = 0	`d[idx++]`	10	idx = 1
idx = 0	`d[++idx]`	20	idx = 1

Tablo 8.3: Sonra artırma ve önce artırma (d = {10, 20, 30}; değerler JDK 21 ile doğrulanmıştır).

İlk iki satırda a her iki durumda da 6 olur; değişen yalnızca b'ye giden değerdir. Son iki satırda idx her iki durumda da 1 olur. `d[idx++]` eski indeksi (0) kullanır ve 10'u verir. `d[++idx]` önce indeksi 1'e çıkarır ve 20'yi verir.

for başlığının güncelleme bölümünde ise ifadenin değeri hiçbir yerde kullanılmaz; yalnızca yan etkisi, yani i'nin artması önemlidir. Bu yüzden aşağıdaki iki döngü arasında hiçbir fark yoktur:

```
for (int i = 0; i < 3; i++) System.out.print(i + " ");   // 0 1 2
for (int i = 0; i < 3; ++i) System.out.print(i + " ");   // 0 1 2
```

İkisi de `0 1 2` yazar. Okunabilirlik açısından sayaçlarda alışılmış biçim olan `i++` tercih edilir. İki yazım arasındaki fark ancak artırma ifadesinin değeri başka bir ifadenin içinde kullanıldığında ortaya çıkar. Bu tür iç içe kullanımlar kodu okumayı zorlaştırdığı için de kaçınılması gereken bir alışkanlıktır.

8.3 while döngüsü

8.3.1 Yazım ve çalışma sırası

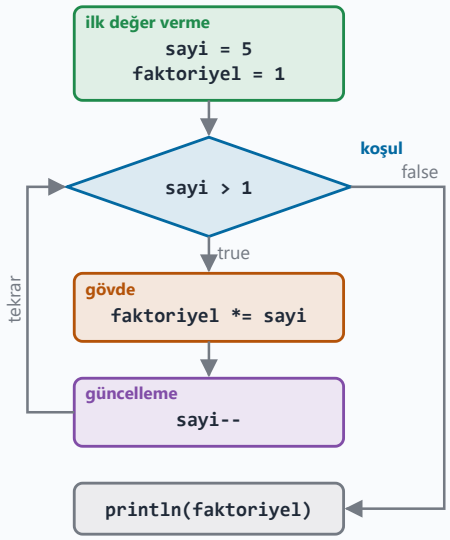
for döngüsü tekrar sayısı belli olduğunda rahat bir yazım sunar. Bazı problemlerde ise kaç tur gerektiği baştan bilinmez. Bir sayının rakamlarını tek tek işlemek böyledir: 7 için bir, 725 için üç tur gerekir. Kullanıcı doğru tahmini yapana kadar soru sormak da böyledir. Bu durumlarda `while` döngüsü daha doğal bir seçimdir. while sözcüğünden sonra parantez içinde yalnızca koşul yazılır. İlk değer verme döngüden önce, güncelleme gövdenin içinde yer alır.

Tanım 8.3

`while` **döngüsü**, `while (koşul) { gövde }` biçiminde yazılan ve koşulu her turdan *önce* soran döngüdür. Koşul doğru olduğu sürece gövde çalışır. Koşulu gövdeden önce sorulan döngülere **ön koşullu döngü** (*pre-test loop*) denir; for döngüsü de ön koşullludur.

Aşağıdaki program 5! değerini, yani *5 × 4 × 3 × 2 × 1* çarpımını hesaplar:

```
int sayi = 5;
int faktoriyel = 1;
while (sayi > 1) {
    faktoriyel *= sayi;
    sayi--;
}
System.out.println(faktoriyel);
```



Şekil 8.4: while döngüsünün akış diyagramı. İlk değer verme (`sayi = 5` , `faktoriyel = 1`) döngüden önce,

güncelleme (`sayi--`) gövdenin sonunda yer alır.

Şekil 8.4 for diyagramıyla aynı biçimdedir. Değişen yalnızca kodun yazılışıdır: for döngüsünde başlıkta toplanan parçalar burada üç ayrı yere dağılmıştır. Bu dağınıklık while döngüsünün tek riskidir. Güncelleme gövdenin içinde unutulabilir. Böyle olursa koşul hiç değişmez ve döngü bitmez.

Koşul gövdeden önce sorulduğu için bir while döngüsünün gövdesi hiç çalışmayabilir. Yukarıdaki programda `sayi = 1` verilirse ilk denetim *1 > 1* yanlış olur. Gövdeye hiç girilmez ve 1 yazılır. Bu sonuç doğrudur, çünkü *1! = 1*'dir. Aynı biçimde `sayi = 0` için de 1 yazılır ve *0! = 1* tanımıyla uyuşur.

8.3.2 5!'i adım adım hesaplamak

Tur	sayi (koşulda)	sayi > 1	faktoriyel	sayi (güncellemeden sonra)
1	5	`true`	5	4
2	4	`true`	20	3
3	3	`true`	60	2
4	2	`true`	120	1
—	1	`false`	120	—

Tablo 8.4: while ile 5!. “sayi (koşulda)” sütunu koşul sorulurken sayi'nin değeridir.

Tablo 8.4 her turu bir satırda gösterir. Çarpım biriktiricisi `faktoriyel` 1'den başlar. 1. turda *5 > 1* doğrudur; *faktoriyel = 1 × 5 = 5* olur ve `sayi--` sayi'yi 4'e indirir. 2. turda *5 × 4 = 20*, 3. turda *20 × 3 = 60*, 4. turda *60 × 2 = 120* hesaplanır. 4. turun sonunda sayi 1 olur. Koşul *1 > 1* yanlıştır ve döngü biter. Program 120 yazar.

Gövde 4 kez çalıştı. 1 ile çarpmak sonucu değiştirmediği için koşul `sayi > 1` seçilmiştir; `sayi >= 1` de aynı sonucu verirdi, yalnızca bir tur fazla dönerdi. Burada sayaç azalarak ilerler. Güncelleme `sayi--` olduğu için sayi her turda 1 azalır ve sonunda 1'e iner. Döngünün bitmesini sağlayan bu ilerlemedir.

Aynı hesap for ile de yazılabilir: `for (int i = 2; i <= 5; i++) faktoriyel *= i;`. İki yazım da doğrudur. Tekrar sayısı baştan belli olduğu için for burada biraz daha okunaklıdır. Faktöriyel çok hızlı büyür: *12! = 479.001.600* int'e sığar, *13! = 6.227.020.800* sığmaz. int en çok 2.147.483.647 tutabilir. 13! int ile hesaplanırsa Java hata vermez; değer sessizce sarar ve 1932053504 gibi yanlış bir sayı yazılır. Bu yüzden faktöriyel hesabında `long` kullanılır. long 20!'e kadar yeter; 21! long'a da sığmaz.

8.4 do-while döngüsü

8.4.1 Yazım ve çalışma sırası

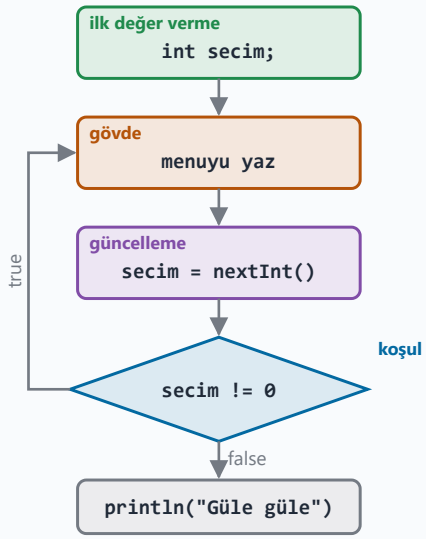
Bazı işlerde gövdenin en az bir kez çalışması gerekir. Bir menü programı düşünelim. Menü ekrana yazılmadan kullanıcı seçim yapamaz; seçim yapılmadan da "çıkılsın mı?" sorusunun cevabı bilinmez. Bu tür işler için Java'da `do-while` döngüsü vardır.

> **Tanım 8.4**
>
> `do-while` **döngüsü**, `do { gövde } while (koşul);` biçiminde yazılan ve koşulu her turdan *sonra* soran döngüdür. Gövde koşula bakılmadan bir kez çalışır; sonraki turlar koşula bağlıdır. Böyle döngülere **son koşullu döngü** (*post-test loop*) denir.

Aşağıdaki program kullanıcıdan bir seçim okur. Seçim 1 ise "Merhaba!" yazar. Kullanıcı 0 girene kadar bu işi tekrarlar:

```
Scanner klavye = new Scanner(System.in);
int secim;
do {
  System.out.print("Seçim (1/0): ");
  secim = klavye.nextInt();
  if (secim == 1)
    System.out.println("Merhaba!");
} while (secim != 0);
System.out.println("Güle güle");
```

Klavyeden okuma için `Scanner` sınıfı kullanılır. Dosyanın başına `import java.util.Scanner;` satırı yazılmalıdır. `klavye.nextInt()` çağrısı kullanıcı bir tam sayı girip Enter tuşuna basana kadar bekler ve girilen sayıyı döndürür.



Şekil 8.5: do-while döngüsünün akış diyagramı. Gövde (menüyü yaz) ve güncelleme (`secim = nextInt()`) koşuldan önce gelir. Koşul doğruysa ok yukarı, gövdenin başına döner.

Şekil 8.5 ile for ve while diyagramları karşılaştırıldığında fark hemen görülür: karar kutusu döngünün başında değil, sonundadır. `secim` değişkeni döngüden önce bildirilmiş ama değer almamıştır. Java, değer verilmemiş bir yerel değişkenin okunmasına izin vermez. Burada sorun çıkmaz, çünkü koşul ilk kez gövdeden sonra sorulur ve o ana kadar `secim = klavye.nextInt();` satırı mutlaka çalışmıştır.

Yazımda iki ayrıntı dikkat ister. Birincisi, `while (secim != 0)` satırının sonunda noktalı virgül vardır; for ve while döngülerinde bu yerde noktalı virgül olmaz. İkincisi, koşulda kullanılan değişken döngüden önce bildirilmelidir. secim gövdenin içinde bildirilseydi kapsamı gövdeyle sınırlı kalır ve `while` satırında kullanılamazdı.

8.4.2 Menü döngüsünü izlemek

Kullanıcı sırayla 1, 2 ve 0 girsin. Tablo 8.5 döngünün üç turunu gösterir.

Tur	Okunan secim	secim == 1	Yazılan	secim != 0
1	1	`true`	Seçim (1/0): ve Merhaba!	`true`
2	2	`false`	Seçim (1/0):	`true`
3	0	`false`	Seçim (1/0):	`false`

Tablo 8.5: do-while menüsü, girdi 1, 2, 0. Koşul her turun sonunda sorulur.

1. turda gövdeye koşula bakılmadan girilir. Menü yazılır, kullanıcı 1 girer, *secim == 1* doğrudur ve “Merhaba!” yazılır. Turun sonunda *1 != 0* doğrudur ve gövdenin başına dönülür. 2. turda 2 okunur. if koşulu yanlıştır ve hiçbir şey yazılmaz; *2 != 0* doğru olduğu için döngü sürer. 3. turda 0 okunur ve *0 != 0* yanlıştır. Döngü biter ve “Güle güle” yazılır.

Gövde 3 kez çalıştı, koşul da 3 kez soruldu. Ön koşullu bir döngüde koşul, gövdeden bir fazla sorulur. Son koşullu döngüde ikisinin sayısı eşittir. Program aynı girdilerle JDK 21'de çalıştırıldığında aşağıdaki çıktıyı verir. Girdiler bir dosyadan yönlendirildiği için kullanıcının yazdığı sayılar ekranda görünmez; istem satırları bu yüzden yan yana düşmüştür:

```
Seçim (1/0): Merhaba!
Seçim (1/0): Seçim (1/0): Güle güle
```

Kullanıcı ilk seçimde 0 girerse çıktı yalnızca bir istem satırı ve "Güle güle"den oluşur. Menü yine de bir kez yazılmıştır. do-while döngüsünün doğal kullanım alanı budur: menüler ve girdi doğrulama. Örneğin kullanıcıdan 1 ile 5 arasında bir not istenirken soru en az bir kez sorulmalıdır. Girilen değer aralık dışındaysa soru tekrarlanır: `do { not = sc.nextInt(); } while (not < 1 || not > 5);`. Girdi 7, 0, 3 olduğunda döngü üç tur döner ve not = 3 ile çıkar.

Sık yapılan hata

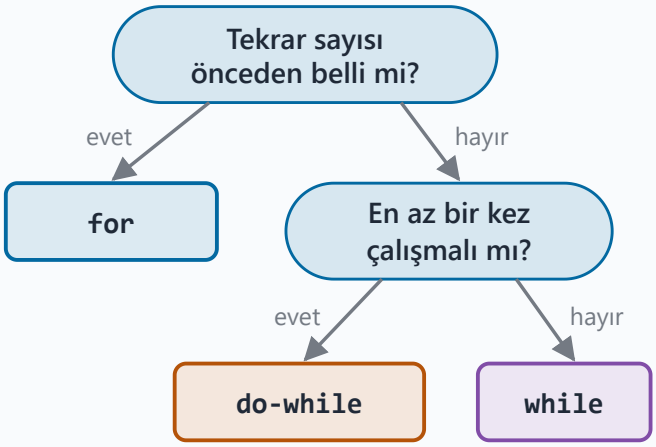
Gövdenin hiç çalışmaması gereken bir yerde do-while kullanmak. Koşul ilk denemede yanlışsa while ve do-while farklı davranır. Aşağıdaki iki döngüde koşul baştan yanlıştır (n = m = 5, sınır 3). do-while yine de bir kez çalışır, 5 yazar ve n'yi 6 yapar. while döngüsünün gövdesi hiç çalışmaz ve m 5 kalır. Çıktı `5 | 6 5` olur. Gövdenin sıfır kez çalışabileceği her durumda ön koşullu döngü seçilir.

```
int n = 5;
do {
    System.out.print(n + " ");
    n++;
} while (n < 3);

int m = 5;
while (m < 3) {
    System.out.print(m + " ");
    m++;
}
System.out.println("| " + n + " " + m);
```

8.4.3 Hangi döngü seçilir?

Üç döngü birbirine çevrilebilir. Her for döngüsü bir while döngüsüyle, her while döngüsü de bir for döngüsüyle yazılabilir. do-while da bir while döngüsü ve önüne eklenen bir gövde kopyasıyla yazılabilir. Yani seçim bir yetenek meselesi değil, okunabilirlik meselesidir. Doğru döngü, okuyana "bu döngü ne zaman ve kaç kez döner?" sorusunun cevabını en kısa yoldan verir.



Şekil 8.6: Döngü seçimi için karar ağacı. Tekrar sayısı önceden belliyse for seçilir. Belli değilse gövdenin en az bir kez çalışması gerekip gerekmediğine bakılır: gerekiyorsa do-while, gerekmiyorsa while.

Durum	Tercih
1'den n'ye say, n kez tekrarla	`for`
Sayı 0 olana kadar basamak işle	`while`
Menü göster, geçerli girdi gelene kadar sor	`do-while`
Kullanıcı “0” yazana kadar topla	`while` / `do-while`

Tablo 8.6: Tipik durumlar ve uygun döngü.

Son satırda iki seçenek vardır, çünkü iki çözüm de doğal görünür. while ile yazılan çözüm ilk sayıyı döngüden önce okur, döngünün sonunda bir sonrakini okur. do-while ile yazılan çözüm okumayı gövdenin başına koyar ama 0'ın toplama eklenmemesi için gövdede ayrıca bir denetim ister. Hangisi daha açık görünüyorsa o seçilir.

8.5 break ve continue

8.5.1 Turu atlamak ve döngüden çıkmak

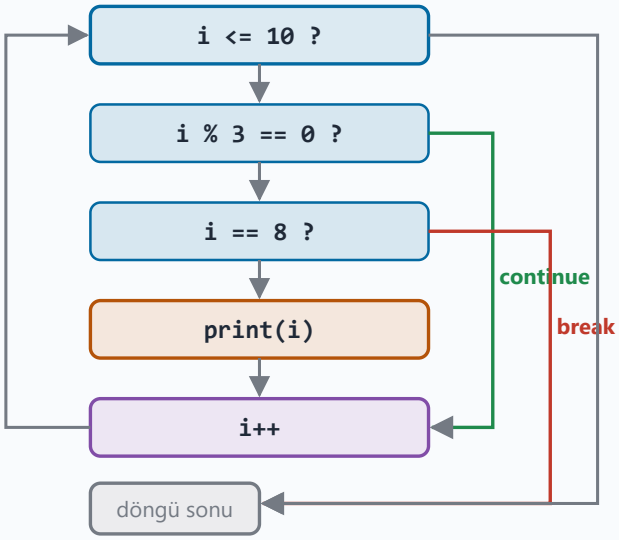
Bir döngü normalde yalnızca koşulu yanlış olduğunda biter ve her turda gövdenin tamamı çalışır. İki deyim bu düzeni değiştirir.

Tanım 8.5

`break`, içinde bulunduğu en yakın döngüyü hemen bitirir; akış döngüden sonraki ilk deyimden sürer. `continue`, yalnızca o turun kalanını atlar. for döngüsünde akış güncellemeye (`i++`), while ve do-while döngüsünde koşula geçer; döngü sürer.

Aşağıdaki döngü 1'den 10'a kadar sayar. 3'ün katlarını atlar ve 8'e gelince durur:

```
for (int i = 1; i <= 10; i++) {
    if (i % 3 == 0) continue;
    if (i == 8) break;
    System.out.print(i + " ");
}
System.out.println("| bitti");
```



Şekil 8.7: break ve continue okları. continue oku `print(i)` kutusunu atlayıp doğrudan `i++` 'ya gider. break oku döngünün dışına, "döngü sonu"na çıkar.

8.5.2 break ve continue'yu izlemek

i	i % 3 == 0	i == 8	Sonuç	Çıktı
1	`false`	`false`	1 yazılır	1
2	`false`	`false`	2 yazılır	1 2
3	`true`	sorulmaz	continue: yazdırma atlanır	1 2
4	`false`	`false`	4 yazılır	1 2 4
5	`false`	`false`	5 yazılır	1 2 4 5
6	`true`	sorulmaz	continue: yazdırma atlanır	1 2 4 5
7	`false`	`false`	7 yazılır	1 2 4 5 7
8	`false`	`true`	break: döngü biter	1 2 4 5 7

Tablo 8.7: break ve continue içeren döngünün izlemesi. "Çıktı" sütunu o ana kadar yazılanlardır.

Tablo 8.7 her i için iki denetimin sonucunu verir. i = 1, 2, 4, 5, 7 için iki koşul da yanlıştır ve sayı yazılır. i = 3 ve i = 6 için *i % 3 == 0* doğrudur. continue çalışır, print atlanır ve akış doğrudan `i++`'ya geçer. Bu turlarda ikinci if hiç sorulmaz. i = 8'de ilk koşul yanlıştır (*8 % 3 = 2*), ikincisi doğrudur ve break döngüyü bitirir. 9 ve 10 hiç denenmez. Akış döngüden sonraki satırdan sürer ve çıktı şöyle olur:

```
1 2 4 5 7 | bitti
```



Şekil 8.8: 1'den 10'a sayıların akıbeti. Yeşil kutular yazılan sayılar, kesikli kutular continue ile atlanan 3 ve 6'dır. Kırmızı 8'de break çalışmıştır; 9 ve 10'a hiç gelinmemiştir.

İki deyimin sırası da önemlidir. Aynı yapıda koşullar `i % 4 == 0` ve `i == 10` olsaydı ve döngü 12'ye kadar sürseydi, 4 ve 8 atlanır, 10'da break çalışırdı. Çıktı `1 2 3 5 6 7 9 | son` olurdu. Bu döngüde i = 8 için önce continue denetimi yapıldığından, break'e o turda hiç sıra gelmez.

break ve continue döngüleri kısaltabilir ama akışı izlemeyi de zorlaştırır. Gövdenin ortasından çıkan bir döngüde, döngünün ne zaman bittiğini anlamak için başlığa bakmak yetmez; bütün gövdenin okunması gerekir. Bu yüzden bu deyimler ölçülü kullanılır. Aynı iş bir koşulla ya da bir bayrak değişkeniyle açıkça yazılabiliyorsa o yol tercih edilir. Asal sayı testindeki `asal && i <= n / i` koşulu böyle bir örnektir.

8.5.3 Etiketli break

Düz break yalnızca içinde bulunduğu en yakın döngüyü bitirir. İç içe iki döngüde iç döngüdeki bir break, dış döngüyü durdurmaz. İki döngüden birden çıkmak gerektiğinde Java'nın **etiketli break** (*labeled break*) deyimi kullanılır. Dış döngünün önüne bir ad ve iki nokta yazılır; `break ad;` o adı taşıyan döngüyü bitirir.

```
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 + " " + j);
    }
}
System.out.println("bitti");
```

Program 1'den 3'e kadar i ve j çiftlerini yazar, çarpımı 4 olan ilk çiftte durur. Tablo 8.8 aynı döngünün etiketli ve düz break ile nasıl davrandığını yan yana gösterir.

i	j	i * j	break dis ile	düz break ile
1	1	1	“1 1” yazılır	“1 1” yazılır
1	2	2	“1 2” yazılır	“1 2” yazılır
1	3	3	“1 3” yazılır	“1 3” yazılır
2	1	2	“2 1” yazılır	“2 1” yazılır
2	2	4	iki döngüden çıkılır	iç döngü biter
2	3	6	denenmez	denenmez
3	1	3	denenmez	“3 1” yazılır
3	2	6	denenmez	“3 2” yazılır
3	3	9	denenmez	“3 3” yazılır

Tablo 8.8: Her (i, j) çifti için etiketli ve düz break'in sonucu.

i = 1 iken j = 1, 2, 3 için çarpım 4 değildir ve üç satır yazılır. i = 2, j = 1 için çarpım 2'dir ve “2 1” yazılır. i = 2, j = 2'de *2 · 2 = 4* olur. Etiketli `break dis` iki döngüden birden çıkar; i = 3 hiç denenmez. Akış dış döngünün ardındaki satıra atlar ve “bitti” yazılır. Çıktı 4 sayı satırı ve “bitti”dir:

```
1 1
1 2
1 3
2 1
bitti
```

Etiket kaldırılıp düz `break` yazılırsa i = 2, j = 2'de yalnızca iç döngü biter. Dış döngü i++ ile 3'e geçer ve iç döngü j = 1'den yeniden başlar. 3 için çarpımlar 3, 6, 9'dur; hiçbiri 4 değildir ve üç satır daha yazılır. Düz break ile 7 sayı satırı çıkar:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
bitti
```

Etiketli break, iç içe döngülerde bir şey arandığında ve bulunduğu anda bütün aramanın bitmesi gerektiğinde işe yarar. Aynı iş bir bayrakla da yapılabilir: bulununca `bulundu = true` yapılır ve dış döngünün koşuluna `!bulundu &&` eklenir. Etiket adı herhangi bir geçerli Java adı olabilir; yaygın seçimler

`dis` ya da `arama` gibi işi anlatan adlardır.

8.6 Döngü hataları ve doğrulama

Döngü hataları çoğu zaman derleyiciye takılmaz. Program derlenir, çalışır ve yanlış bir sonuç verir ya da hiç bitmez. Bu kesimde dört tipik sorun ele alınır: sonsuz döngü, `double` sayaç, döngü değişmezinin bozulması ve sınır hataları.

8.6.1 Sonsuz döngü

Tanım 8.6

Sonsuz döngü (*infinite loop*), koşulu hiçbir zaman yanlış olmadığı için kendiliğinden bitmeyen döngüdür. Program "donmuş" görünür: ya hiçbir şey yazmaz ya da aynı satırları durmadan yazar.




```
for (int i = 0; i < 5; i--) → i = 0, -1, -2, … koşul hep true

while (i < 5) { … } // i++ yok → i hep 0
```

Şekil 8.9: Sayaç hedefin tersine gidiyor. Koşul `i < 5` iken sayaç `i--` ile 0, −1, −2, … diye azalır ve hedeften uzaklaşır. Alttaki while döngüsünde güncelleme hiç yoktur ve i hep 0 kalır.

Şekil 8.9 iki tipik nedeni gösterir. İlkinde güncelleme yanlış yöndedir. `for (int i = 0; i < 5; i--)` döngüsünde i her turda azalır ve *i < 5* hep doğru kalır. İkincisinde güncelleme hiç yazılmamıştır. `while (i < 5) { … }` gövdesinde i'yi değiştiren bir deyim yoksa koşul ilk turdaki değerini korur.

Birinci döngü kuramsal olarak sonsuzdur. Gerçekte ise int sınırlı bir tiptir. i, −2.147.483.648'e indikten sonra bir kez daha azaltılınca **taşma** (*overflow*) olur ve değer en büyük int'e, 2.147.483.647'ye sarar. Bu değer 5'ten küçük değildir ve döngü biter. JDK 21 ile sayılan tur sayısı aşağıdadır:

Döngü	Tur sayısı	Son işlenen i
for (int i = 0; i < 5; i--)	2147483649	−2147483648

Tablo 8.9: Yanlış yönde güncellenen döngünün tur sayısı (long bir sayaçla sayılmıştır).

i, 0'dan −2.147.483.648'e kadar her değeri bir kez alır. Bu, $2^{31} + 1 = 2.147.483.649$ tur, yani yaklaşık 2,1 milyar turdur. Modern bir bilgisayar bu kadar turu birkaç saniyede bitirebilir; gövdede ekrana yazma varsa saatler sürer. Pratikte bu döngü de sonsuz sayılır ve doğru bir program değildir.

Bazen döngü bilerek sonsuz yazılır: `while (true)`. Sunucular, oyun döngüleri ve bekçi değerle okuma bu biçimi kullanır. Böyle bir döngünün gövdesinde mutlaka bir `break` koşulu olmalıdır. Program bir sonsuz döngüye takılırsa komut satırında Ctrl+C tuşlarıyla durdurulur; geliştirme ortamlarında bunun için bir durdurma düğmesi vardır.

> **Sık yapılan hata**
>
> Süslü parantezleri unutmak. `int i = 0; while (i < 5) System.out.print(i); i++;` satırında gövde yalnızca `print(i)` deyimidir. Girinti ya da aynı satırda yazılmış olmak, `i++` 'yı gövdeye katmaz. i hep 0 kalır ve ekrana durmadan 0 yazılır. Gövdesi tek deyim olsa bile döngülerde süslü parantez kullanmak bu hatayı baştan önler.

8.6.2 double sayaç tuzağı

Aşağıdaki döngü x'i 0'dan başlatıp 0,1'er artırıyor. Amaç 10 tur dönmektir: x = 0,0; 0,1; ...; 0,9.

```
int n = 0;
for (double x = 0; x < 1.0; x += 0.1) {
    n++;
}
System.out.println(n);
```

Program 10 değil, 11 yazar:

```
11
```

Nedeni, 0,1 sayısının ikili tabanda tam gösterilememesidir. Onluk tabanda 1/3 = 0,333... sonsuza kadar sürer. Aynı biçimde ikili tabanda 1/10 = $0{,}0001100110011\ldots_2$ sonsuza kadar sürer. double bu açılımı 53 bitte keser. Her toplamada küçük bir yuvarlama hatası oluşur ve bu hatalar birikir. Tablo 8.10 x'in her turdaki gerçek değerini Java'nın yazdığı biçimde verir.

Tur	x (koşul sorulurken)	x < 1.0	n (gövdeden sonra)
1	0.0	`true`	1
2	0.1	`true`	2
3	0.2	`true`	3
4	0.30000000000000004	`true`	4
5	0.4	`true`	5

Tur	x (koşul sorulurken)	x < 1.0	n (gövdeden sonra)
6	0.5	`true`	6
7	0.6	`true`	7
8	0.7	`true`	8
9	0.7999999999999999	`true`	9
10	0.8999999999999999	`true`	10
11	0.9999999999999999	`true`	11
—	1.0999999999999999	`false`	11 (döngü biter)

Tablo 8.10: `x += 0.1` ile ilerleyen double sayaç. Değerler JDK 21'in `println` çıktısıdır.

4. turda ilk sapma görülür: x = 0,30000000000000004. 9. turda x = 0,7999999999999999 olur. 10 toplamadan sonra x tam 1,0 değil, 0,9999999999999999'dur. Bu değer 1,0'dan küçük olduğu için koşul doğru çıkar ve döngü bir tur fazla döner. x 1,0999999999999999 olunca döngü biter.

Koşul `x != 1.0` yazılsaydı durum daha kötü olurdu. x hiçbir turda tam 1,0 olmadığı için koşul hep doğru kalır ve döngü bitmez. Denemede 1000 turdan sonra x yaklaşık 99,9999999999986'dır. Ders açıktır: döngü sayacı tam sayı olmalıdır. Gerçek sayı gerekiyorsa her turda sayaçtan hesaplanır: `for (int k = 0; k < 10; k++) { double x = k * 0.1; … }`. Bu döngü tam 10 tur döner.

Her ondalık adım hatalı değildir. 0,25 = 1/4 ikili tabanda tam gösterilir ($0{,}01_2$). `for (double d = 0; d != 1.0; d += 0.25)` döngüsü tam 4 tur döner ve biter. Ancak hangi sayının tam gösterilebildiğini her seferinde düşünmek hataya açıktır. Güvenli kural yine int sayaçtır.

8.6.3 Döngü değişmezi

Bir döngünün doğru çalıştığını birkaç girdiyle denemek yararlıdır ama her girdiyi denemek mümkün değildir. Döngünün her girdide doğru sonuç verdiğini görmenin bir yolu, her turda değişmeden doğru kalan bir özellik bulmaktır.

> **Tanım 8.7**
>
> **Döngü değişmezi** (*loop invariant*), döngünün her turunun başında doğru olan bir özelliktir. Döngüye girerken doğrudur; bir tur onu bozmaz; döngü bittiğinde de doğrudur ve o anda istenen sonucu verir.

1'den n'ye toplam döngüsünde değişmez şudur: her turun başında *toplam = 1 + 2 + … + (i − 1)*. Yani toplam, o ana kadar işlenmiş sayıların toplamıdır. Tablo 8.11 n = 4 için bu özelliği her turun başında denetler.

Turun başında i	toplam	1 + … + (i − 1)	Değişmez
1	0	0 (boş toplam)	doğru

Turun başında i	toplam	1 + … + (i − 1)	Değişmez
2	1	1 = 1	doğru
3	3	1 + 2 = 3	doğru
4	6	1 + 2 + 3 = 6	doğru
5	10	1 + 2 + 3 + 4 = 10	doğru

Tablo 8.11: Toplam döngüsünde değişmezin denetimi (n = 4). Son satır, koşulun yanlış çıktığı andır.



Şekil 8.10: Değişmezin iki anı. Solda i = 3 turunun başı: toplam = 3 = 1 + 2. Sağda i = 5, koşulun yanlış çıktığı an: toplam = 10 = 1 + 2 + 3 + 4.

Değişmezin doğruluğu üç adımda gösterilir:

1. **Başlangıç.** İlk turdan önce i = 1 ve toplam = 0'dır. *1 + … + 0* hiç terim içermeyen boş bir toplamdır ve değeri 0'dır. Değişmez baştan doğrudur.
2. **Koruma.** Bir turun başında *toplam = 1 + … + (i − 1)* olsun. Gövde toplama i'yi ekler: *toplam = 1 + … + (i − 1) + i*. Güncelleme i'yi 1 artırır. Yeni i ile yazıldığında bu yine *1 + … + (i − 1)* biçimindedir. Değişmez bir sonraki turun başında da doğrudur.
3. **Bitiş.** Döngü *i = n + 1* olduğunda biter. Değişmezde i yerine n + 1 yazılınca *toplam = 1 + 2 + … + n* çıkar. Bu tam olarak istenen sonuçtur.

Üç adımın hiçbiri n'nin belirli bir değerine bağlı değildir. Bu yüzden akıl yürütme her n için geçerlidir. Üs alma döngüsünde değişmez "her turun başında *sonuc = taban*^{i}", faktöriyel döngüsünde "faktoriyel, o ana kadar çarpılan sayıların çarpımıdır" biçimindedir. Bir döngü yazarken değişmezi tek cümleyle söyleyebilmek, döngünün ne yaptığını gerçekten bilmek demektir.

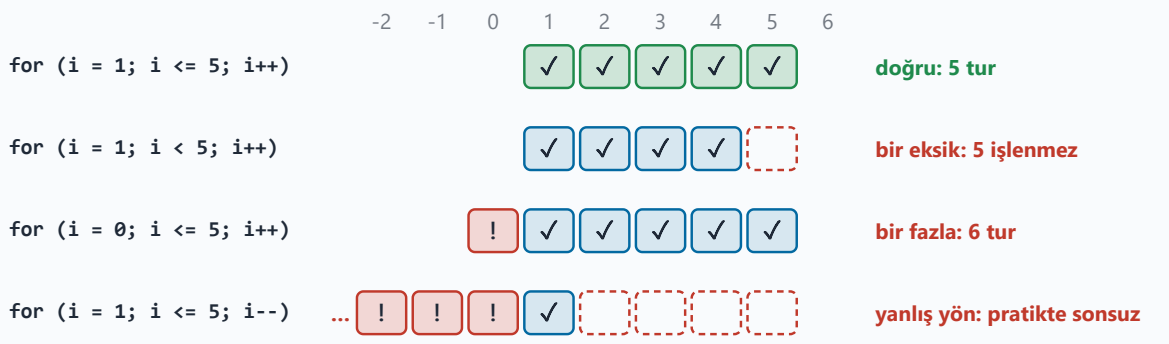
8.6.4 Sınır hataları

En sık döngü hatası bir tur eksik ya da bir tur fazla dönmektir. Sayaç 0'dan mı 1'den mi başlamalı, koşul `<` mı `<=` mı olmalı? Bu küçük seçimlerdeki bir yanlış, döngünün ilk ya da son elemanı atlamasına veya fazladan bir eleman işlemesine yol açar.

Tanım 8.8

Sınır hatası (*off-by-one error*), bir döngünün doğru sayıdan bir eksik ya da bir fazla tur dönmesiyle ortaya çıkan hatadır. Çoğunlukla başlangıç değerinin (0 ya da 1) veya karşılaştırma işlecinin (`<` ya da `<=`) yanlış seçilmesinden doğar.

Görev 1'den 5'e kadar saymak olsun. Şekil 8.11 ve Tablo 8.12 dört başlığın hangi değerleri işlediğini gösterir.



Şekil 8.11: Dört for başlığının işlediği sayılar. Yalnızca ilki tam 1–5'i işler. İkincisi 5'i kaçırır, üçüncüsü fazladan 0'ı işler, dördüncüsü yanlış yöne gider.

Başlık	İşlenen i değerleri	Tur sayısı
for (i = 1; i <= 5; i++)	1 2 3 4 5	5
for (i = 1; i < 5; i++)	1 2 3 4	4
for (i = 0; i <= 5; i++)	0 1 2 3 4 5	6
for (i = 1; i <= 5; i--)	1 0 −1 −2 … −2147483648	2147483650

Tablo 8.12: Dört başlık ve tur sayıları (JDK 21 ile sayılmıştır).

Sonuncu başlık, sonsuz döngü kesimindeki taşmanın aynısıdır. i 1'den başlayıp −2.147.483.648'e kadar iner; bu 2.147.483.650 tur eder. Ardından değer sarar ve döngü biter.

Sık karşılaşılan bir başka kalıp 0'dan başlayan saymadır. Diziler ve dizgiler 0'dan numaralanır. Bu yüzden Java programlarında `for (int i = 0; i < 5; i++)` biçimi çok yaygındır. Bu döngü i = 0, 1, 2, 3, 4 için tam 5 tur döner:

```
int sayac = 0;
for (int i = 0; i < 5; i++) {
    System.out.print(i + " ");
    sayac++;
}
System.out.println("| tur: " + sayac);
```

```
0 1 2 3 4 | tur: 5
```

Koşul `i <= 5` yazılırsa i = 5 de işlenir ve döngü 6 tur döner. Sayaç 5 yerine 6 olur:

```
0 1 2 3 4 5 | tur: 6
```

Sınır hatasını önlemenin yolu basit bir denetimdir. Bir döngü yazıldıktan sonra üç soru sorulur: ilk turda sayaç hangi değeri alıyor, son turda hangi değeri alıyor ve toplam kaç tur dönüyor? `for (int i = a; i < b; i++)` biçiminde tur sayısı $b - a$, `for (int i = a; i <= b; i++)` biçiminde $b - a + 1$'dir. Bu iki formül çoğu sınır hatasını kod çalıştırılmadan yakalar.

8.7 Rakam kalıbı

Döngülerle çözülen problemlerin çoğu birkaç ortak kalıba dayanır. Kalıp bir kez anlaşılınca aynı iskelet farklı problemlerde küçük değişikliklerle yeniden kullanılır. Bu kesimdeki örneklerin hepsi bir tam sayının rakamlarını tek tek işler. Basamak sayısı, rakamlar toplamı, ters çevirme, palindrom, Armstrong sayısı ve onluktan ikiliye çevirme aynı döngünün çeşitlemeleridir.

Tanım 8.9

Kalıp (*pattern*), birçok problemde tekrar eden bir çözüm iskeletidir. Bir döngü kalıbı, döngünün türünü, ilk değerlerini, koşulunu ve güncellemesini belirler; problemden probleme yalnızca gövdedeki işlem değişir.

8.7.1 % 10 ve / 10

Kalıbın temelinde iki tam sayı işlemi vardır. Pozitif bir tam sayı için `sayi % 10`, sayının 10'a bölümünden kalanı, yani **son rakamı** verir: *725 % 10 = 5*. `sayi / 10` ise tam sayı bölmesidir; kesirli kısım atılır ve **son rakam silinir**: *725 / 10 = 72*. İki işlem art arda uygulandığında sayı sağdan sola doğru rakam rakam "tüketilir":

725 → (5) → 72 → (2) → 7 → (7) → 0

Sayı 0 olduğunda işlenecek rakam kalmamıştır. Kalıbın iskeleti bu gözleme dayanır:

```
while (kalan > 0) {
    int rakam = kalan % 10;   // son rakamı al
    // ... rakamla bir iş yap ...
    kalan /= 10;              // son rakamı sil
}
```

Döngü, sayının basamak sayısı kadar tur döner. Altı basamaklı bir sayı için 6 tur yeter. Bu yüzden rakam kalıbı çok büyük sayılarda bile hızlıdır: en büyük int on basamaklıdır ve döngü en çok 10 tur döner.

İki ayrıntı her örnekte tekrarlanır. Birincisi, döngü sayının kendisini değil bir kopyasını (`kalan`) tüketir. Döngü bittiğinde kalan 0 olur ama özgün sayı korunur; palindrom ve Armstrong testinde sonuç onunla karşılaştırılır. İkincisi, negatif sayıların işareti önce atılır. Java'da *−725 % 10 = −5*'tir; kalan, bölünenin işaretini alır. İşaret atılmazsa `while (kalan > 0)` koşulu baştan yanlış çıkar ve döngü hiç çalışmaz.

8.7.2 Basamak sayısı

```
int sayi = -507;
long kalan = sayi;
if (kalan < 0) kalan = -kalan;
int basamak = 0;
do {
    basamak++;
    kalan /= 10;
} while (kalan > 0);
System.out.println(basamak);
```



Şekil 8.12: Basamak sayısı. kalan 507, 50, 5, 0 olurken basamak 0, 1, 2, 3 olur. kalan 0 olunca koşul yanlış çıkar; sonuç 3 basamaktır.

Tur	kalan (turun başında)	basamak	kalan / 10	kalan > 0
1	507	1	50	`true`
2	50	2	5	`true`
3	5	3	0	`false`

Tablo 8.13: Basamak sayısı, sayi = −507. İşaret atıldıktan sonra kalan = 507'dir.

Burada rakamın kendisi gerekmez; yalnızca kaç kez bölündüğü sayılır. Bu yüzden gövdede `% 10` yoktur. basamak bir sayaçtır ve her turda 1 artar. Program 3 yazar.

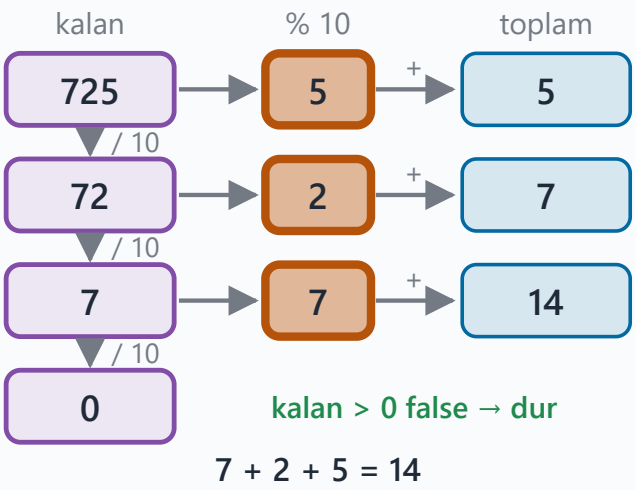
Döngünün do-while olması bilinçli bir seçimdir. sayi = 0 için while döngüsü hiç çalışmaz ve 0 basamak bulur. Oysa 0 bir basamaklı bir sayıdır. do-while gövdeyi en az bir kez çalıştırdığı için 0 için de 1 bulur; ayrıca bir özel durum yazmak gerekmez. Aynı sonuç while döngüsüyle ve önüne eklenen `if (sayi == 0) return 1;` denetimiyle de alınabilir.

Kopyanın `long` olmasının da bir nedeni vardır. int'in en küçük değeri −2.147.483.648'dir ama en büyük değeri 2.147.483.647'dir. Bu yüzden −2.147.483.648'in pozitif karşılığı int'e sığmaz. `Math.abs(-2147483648)` bile yine −2.147.483.648 döndürür. Kopya long olunca işaret değiştirme güvenle yapılır ve uç değer için doğru sonuç (10) çıkar. Kopya int olsaydı bu girdi için yöntem 1 bulurdu.

8.7.3 Rakamlar toplamı

Rakamlar toplamında her turda son rakam bir biriktiriciye eklenir:

```
int sayi = 725;
long kalan = sayi;
if (kalan < 0) kalan = -kalan;
int toplam = 0;
while (kalan > 0) {
    toplam += (int) (kalan % 10);
    kalan /= 10;
}
System.out.println(toplam);
```



Şekil 8.13: Rakamlar toplamı. 725'ten `% 10` ile sırasıyla 5, 2, 7 alınır; toplam 5, 7, 14 olur.

Tur	kalan	kalan % 10	toplam	kalan / 10
1	725	5	5	72

Tur	kalan	kalan % 10	toplam	kalan / 10
2	72	2	7	7
3	7	7	14	0

Tablo 8.14: Rakamlar toplamı, sayi = 725.

Program 14 yazar: *7 + 2 + 5 = 14*. Burada while yeterlidir. sayi = 0 için gövde hiç çalışmaz ve toplam 0 kalır; bu sonuç doğrudur. `(int)` dönüşümü de gereklidir. kalan long olduğu için `kalan % 10` da long'dur ve int bir değişkene eklenirken açıkça dönüştürülmelidir. Bileşik atama `+=` bu dönüşümü kendisi de yapardı; açık yazım okuyana ne olduğunu gösterir. Aynı döngü −725 için de 14, 1001 için 2 verir.

8.7.4 Ters çevirme ve palindrom

Bir sayıyı ters çevirmek için rakamlar sağdan sola alınır ve yeni bir sayı soldan sağa kurulur. Her turda eldeki ters sayı 10 ile çarpılır; böylece eski rakamlar bir basamak sola kayar ve birler basamağı boşalır. Boşalan yere yeni rakam eklenir.

```
int sayi = 1221;
int kalan = sayi, ters = 0;
while (kalan > 0) {
    int rakam = kalan % 10;
    ters = ters * 10 + rakam;
    kalan /= 10;
}
System.out.println(ters);
System.out.println(ters == sayi);  // palindrom mu?
```

Tur	kalan	rakam	ters = ters × 10 + rakam	kalan / 10
1	1221	1	0 × 10 + 1 = 1	122
2	122	2	1 × 10 + 2 = 12	12
3	12	2	12 × 10 + 2 = 122	1
4	1	1	122 × 10 + 1 = 1221	0
ters == sayi	`true`			

Tablo 8.15: 1221'in tersi. Son satır palindrom denetimidir.

Tanım 8.10

Rakamları tersten okunduğunda da aynı kalan sayıya **palindrom** (*palindrome*) denir. 1221, 7 ve 12321 palindromdur; 123 değildir.

Program önce 1221'i, sonra `true` yazar. Palindrom testi, ters sayının özgün sayıya eşit olup olmadığına bakar. Bu yüzden döngü özgün sayıyı değil, kopyasını tüketir. 123 için tablo şöyle olur:

Tur	kalan	rakam	ters = ters × 10 + rakam	kalan / 10
1	123	3	0 × 10 + 3 = 3	12
2	12	2	3 × 10 + 2 = 32	1
3	1	1	32 × 10 + 1 = 321	0
ters == sayi	`false`			

Tablo 8.16: 123'ün tersi 321'dir; 321 ≠ 123 olduğundan 123 palindrom değildir.

Sık yapılan hata

Sondaki sıfırları unutmak. 120'nin tersi bu yöntemle 21 çıkar, çünkü 021 diye bir int yoktur. 1200 de 21'e döner. Palindrom testi bu sayılar için yine doğru sonuç verir (120 ≠ 21). Ancak tersi ekranda "021" biçiminde göstermek gerekiyorsa sayı değil, karakter dizisi kurulmalıdır.

8.7.5 Armstrong sayısı

Tanım 8.11

Rakamlarının basamak sayısı kadar kuvvetleri toplamı kendisine eşit olan sayıya **Armstrong sayısı** (*Armstrong number*) denir. Üç basamaklı bir sayı için kuvvet 3'tür: $153 = 1^3 + 5^3 + 3^3 = 1 + 125 + 27$.

```
int sayi = 153;
int kalan = sayi, toplam = 0;
while (kalan > 0) {
    int r = kalan % 10;
    toplam += r * r * r;   // 3 basamak: küp
    kalan /= 10;
}
System.out.println(toplam == sayi);
```

Tur	kalan	r	r × r × r	toplam
1	153	3	27	27
2	15	5	125	152
3	1	1	1	153

Tablo 8.17: 153'ün Armstrong denetimi.

Kalıp rakamlar toplamının aynısıdır; yalnızca rakam yerine rakamın küpü eklenir. Toplam 153'e ulaşır ve program `true` yazar. Bu kod yalnızca üç basamaklı sayılar için doğrudur, çünkü kuvvet sabit 3 alınmıştır. Genel bir çözüm önce basamak sayısını bulur, sonra her rakamın o kuvvetini alır. 100'den 999'a kadar bütün sayılar denendiğinde dört Armstrong sayısı çıkar: 153, 370, 371 ve 407.

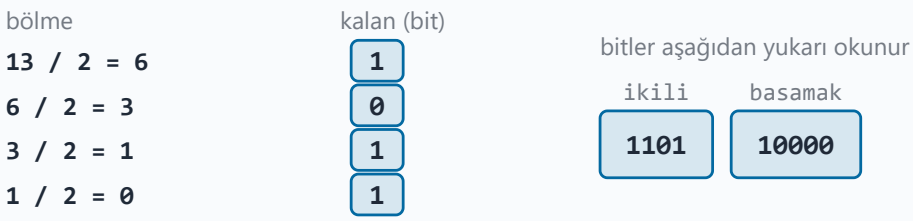
8.7.6 Onluktan ikiliye çevirme

Kalıbın en ilginç çeşitlemesi taban değiştirmedir. 10 yerine 2 kullanılınca `n % 2` sayının ikili tabandaki son bitini, `n / 2` ise o biti silinmiş sayıyı verir. Bitler sağdan sola elde edilir. Aşağıdaki program ikili gösterimi ondalık rakamlarla (1 ve 0) bir int'te kurar. Her yeni bit bir öncekinin bir basamak soluna yazılır; bunun için `basamak` değişkeni 1, 10, 100, … diye büyür.

```
int n = 13;
int ikili = 0, basamak = 1;
while (n > 0) {
    ikili += (n % 2) * basamak;
    basamak *= 10;
    n /= 2;
}
System.out.println(ikili);   // 1101
```

Tur	n	n % 2	basamak	ikili	n / 2
1	13	1	1	1	6
2	6	0	10	1	3
3	3	1	100	101	1
4	1	1	1000	1101	0

Tablo 8.18: 13'ün ikili gösterimi.



Şekil 8.14: Bölme merdiveni. 13 sürekli 2'ye bölünür; kalanlar 1, 0, 1, 1'dir. Bitler aşağıdan yukarı okunur: 1101.

Program 1101 yazar. Denetim: *1 · 8 + 1 · 4 + 0 · 2 + 1 · 1 = 13*. Bu yöntemin bir sınırı vardır. İkili gösterim ondalık rakamlarla kurulduğu için sonuç çabuk büyür. 1023'ün ikili gösterimi on tane 1'dir (1111111111) ve int'e sığar. 1024'ün ikili gösterimi 10000000000, yani on bir basamaklıdır ve int'e

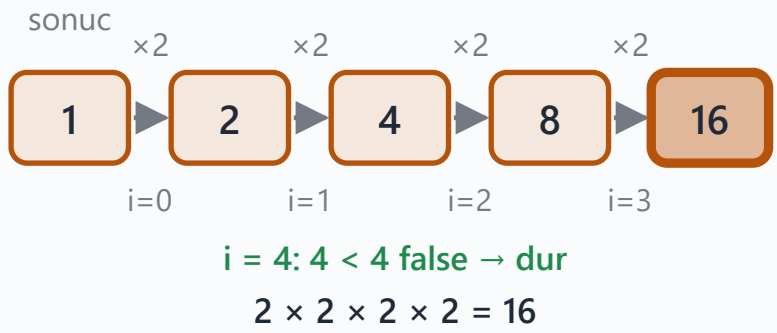
sığmaz. Program bu girdi için sessizce taşar ve 1410065408 yazar. Büyük sayılar için sonuç karakter dizisi olarak kurulur ya da Java'nın `Integer.toBinaryString` metodu kullanılır.

8.8 Çarpım ve bölen kalıpları

8.8.1 Üs alma

Bir tabanın tam sayı kuvveti, tabanın kendisiyle tekrar tekrar çarpılmasıdır: $2^4 = 2 \times 2 \times 2 \times 2$. Tekrar sayısı (üs) bellidir; bu yüzden for döngüsü uygundur.

```
int taban = 2, us = 4;
int sonuc = 1;
for (int i = 0; i < us; i++) {
    sonuc *= taban;
}
System.out.println(sonuc);
```



Şekil 8.15: Üs alma. sonuc 1'den başlar ve 4 turda ikişer kat artarak 2, 4, 8, 16 olur. i = 4'te koşul yanlış çıkar.

Tur	i	i < us	sonuc
1	0	`true`	2
2	1	`true`	4
3	2	`true`	8
4	3	`true`	16
—	4	`false`	16 (döngü biter)

Tablo 8.19: 2^4'ün hesabı.

Döngü tam `us` kez çalışır ve her turda bir kez tabanla çarpar. Değişmez "her turun başında *sonuc* = 2^i" biçimindedir. i = 4'te döngü biter ve sonuc = 2^4 = 16 olur. Çarpım biriktiricisi 1'den başlar. us = 0 ise döngü hiç çalışmaz ve sonuç 1 kalır; bu da $2^0 = 1$ ile uyuşur.

Kod negatif üsleri kapsamaz; us negatifse döngü hiç dönmez ve yanlış olarak 1 bulunur. Büyük sonuçlar da int sınırını aşar. 2^{30} = 1.073.741.824 sığar. 2^{31} int'te −2.147.483.648, 2^{32} ise 0 çıkar. Taşma bir hata iletisi üretmez; sonuç sessizce yanlış olur.

8.8.2 Mükemmel sayı

Bölen kalıbında döngü, sayının olası bölenlerini tek tek dener. i, n'yi bölüyorsa *n % i == 0* olur.

> **Tanım 8.12**
>
> Kendisi hariç pozitif bölenlerinin toplamı kendisine eşit olan sayıya **mükemmel sayı** (*perfect number*) denir. En küçük örnek *6 = 1 + 2 + 3*'tür.

```
int n = 28;
int toplam = 0;
for (int i = 1; i <= n / 2; i++) {
    if (n % i == 0) toplam += i;
}
System.out.println(toplam == n);
```

n'nin kendisi hariç en büyük böleni en fazla n / 2'dir. n / 2'den büyük ve n'den küçük bir sayı n'yi bölemez. Bu yüzden döngü n / 2'de durur ve gereksiz turlardan kaçınır. Tablo 8.20 28 için 14 turu gösterir.

i	28 % i	Bölen mi?	toplam
1	0	evet	1
2	0	evet	3
3	1	hayır	3
4	0	evet	7
5	3	hayır	7
6	4	hayır	7
7	0	evet	14
8	4	hayır	14
9	1	hayır	14
10	8	hayır	14
11	6	hayır	14
12	4	hayır	14
13	2	hayır	14

i	28 % i	Bölen mi?	toplam
14	0	evet	28

Tablo 8.20: 28'in bölenleri. toplam yalnızca bölen bulunan turlarda artar.



Şekil 8.16: 28'in bölen adayları. Yeşil kutular bölenlerdir: 1, 2, 4, 7, 14. Toplamları 28'dir.

Bölenler 1, 2, 4, 7 ve 14'tür. Toplamları *1 + 2 + 4 + 7 + 14 = 28* olduğu için program `true` yazar. 10.000'e kadar denendiğinde yalnızca dört mükemmel sayı çıkar: 6, 28, 496 ve 8128.

8.8.3 Asal sayı testi

> **Tanım 8.13**
>
> 1'den büyük olup 1 ve kendisinden başka pozitif böleni olmayan tam sayıya **asal sayı** (*prime number*) denir. 0 ve 1 asal değildir; 2 tek çift asaldır.

```
int n = 29;
boolean asal = n >= 2;
for (int i = 2; asal && i <= n / i; i++) {
    if (n % i == 0) asal = false;
}
System.out.println(asal);
```

Kodun her satırında bir fikir vardır. `asal = n >= 2` ataması 0, 1 ve negatif sayıları baştan eler. Döngü 2'den başlayarak bölen arar. Bir bölen bulununca `asal = false` olur ve koşuldaki `asal &&` döngüyü durdurur. Burada break yerine bir bayrak kullanılmıştır. Bayrak, durumu bir mantıksal değişkende tutar ve döngünün çıkış koşulunu başlıkta görünür kılar.

İkinci fikir aramanın sınırıdır. n'nin 1 ve kendisinden başka bir böleni varsa, bu bölenlerden en az biri $\sqrt{n}$'den büyük değildir. Çünkü $n = a \times b$ ise a ve b'nin ikisi birden $\sqrt{n}$'den büyük olamaz; olsaydı çarpımları n'yi geçerdi. Bu yüzden i'yi $\sqrt{n}$'ye kadar denemek yeter. Koşuldaki `i <= n / i`, $i \times i \le n$ ile aynı anlamdadır ama `i * i` çok büyük i için taşabilir; bölme taşmaz.

n	i	n / i	asal && i <= n / i	n % i	asal
29	2	14	`true`	1	`true`
29	3	9	`true`	2	`true`
29	4	7	`true`	1	`true`
29	5	5	`true`	4	`true`
29	6	4	`false`	—	`true`
21	2	10	`true`	1	`true`
21	3	7	`true`	0	`false`
21	4	hesaplanmaz	`false`	—	`false`

Tablo 8.21: 29 ve 21 için asal testi. Her n için son satır döngüyü bitiren koşul denetimidir.



Şekil 8.17: 29 için i = 2, 3, 4, 5 denenir; kalanlar 1, 2, 1, 4'tür. i = 6'da *6 ≤ 29 / 6 = 4* yanlıştır, döngü biter ve 29 asaldır.

29 için 4 bölen denenir ve hiçbiri 29'u bölmez. i = 6'da *6 <= 29 / 6* yani *6 <= 4* yanlıştır. √29 ≈ 5,39 geçilmiştir ve daha büyük bölen aramaya gerek yoktur. Program `true` yazar.

21 için i = 3'te *21 % 3 = 0* olur ve asal = false yapılır. Güncelleme i'yi 4'e çıkarır. Koşulun ilk parçası `asal` artık yanlıştır. `&&` işleci kısa devre yapar: ilk işlenen yanlışsa ikincisi hiç hesaplanmaz. `i <= n / i` denenmeden döngü biter ve program `false` yazar. Yalnızca 2 bölen denenmiştir.

√n sınırının kazancı büyüktür. 97 için yalnızca 8 aday (2…9) denenir. En büyük int olan 2.147.483.647 asaldır; bu sayı için 46.339 deneme yeter. 2'den n − 1'e kadar arayan bir döngü aynı iş için iki milyardan fazla tur dönerdi.

8.8.4 1'den n'ye asal sayılar

Asal testi bir dış döngünün içine konunca 2'den n'ye kadar bütün asallar listelenir. Her k için iç döngü asal testini yapar:

```
int n = 20;
for (int k = 2; k <= n; k++) {
    boolean asal = true;
    for (int i = 2; i <= k / i; i++) {
        if (k % i == 0) { asal = false; break; }
    }
    if (asal) System.out.print(k + " ");
}
```

Burada iç döngü break ile biter. `asal` değişkeni her k için yeniden `true` yapılır; bu atama dış döngünün içinde olmalıdır. Döngünün dışına taşınırsa ilk bileşik sayıdan sonra bayrak hep false kalır ve 2, 3'ten sonra hiçbir sayı yazılmaz. Tablo 8.22 her k için denenen bölenleri verir.

k	Denenen i değerleri	Sonuç
2	yok (2 > 2 / 2)	asal, yazılır
3	yok (2 > 3 / 2)	asal, yazılır
4	2 (4 % 2 = 0, break)	asal değil
5	2	asal, yazılır
6	2 (6 % 2 = 0, break)	asal değil
7	2	asal, yazılır
8	2 (8 % 2 = 0, break)	asal değil
9	2, 3 (9 % 3 = 0, break)	asal değil
10	2 (10 % 2 = 0, break)	asal değil
11	2, 3	asal, yazılır
12	2 (12 % 2 = 0, break)	asal değil
13	2, 3	asal, yazılır
14	2 (14 % 2 = 0, break)	asal değil
15	2, 3 (15 % 3 = 0, break)	asal değil
16	2 (16 % 2 = 0, break)	asal değil
17	2, 3, 4	asal, yazılır
18	2 (18 % 2 = 0, break)	asal değil
19	2, 3, 4	asal, yazılır
20	2 (20 % 2 = 0, break)	asal değil

Tablo 8.22: 2'den 20'ye asal testi (iç döngüde denenen i değerleri).

2 ve 3 için iç döngü hiç çalışmaz, çünkü *2 <= k / 2* yanlıştır (2 / 2 = 1 ve 3 / 2 = 1). Çift sayıların hepsi ilk denemede, i = 2'de elenir. 9 ve 15 ikinci denemede, i = 3'te elenir. 17 ve 19 için 2, 3 ve 4 denenir; hiçbiri bölmez. Program şunu yazar:

```
2 3 5 7 11 13 17 19
```



Şekil 8.18: 2..20 arasındaki sayıların durumu. Yeşil kutular asal sayılardır: 2, 3, 5, 7, 11, 13, 17, 19.

2..20 arasında 8 asal vardır; 2..100 arasında 25. Çok sayıda asal gerektiğinde her sayıyı ayrı ayrı test etmek yerine Eratosthenes eleği gibi daha hızlı yöntemler kullanılır. Bu yöntem bir asal bulduğunda onun bütün katlarını tek geçişte işaretler.

8.8.5 Fibonacci dizisi

Tanım 8.14

Fibonacci dizisi (*Fibonacci sequence*) 0 ve 1 ile başlayan, sonraki her terimi kendisinden önceki iki terimin toplamı olan dizidir: 0, 1, 1, 2, 3, 5, 8, 13, …

Diziyi üretmek için yalnızca son iki terimi tutmak yeter. a sıradaki terimi, b ondan sonrakini tutar. Her turda a yazılır, sonraki terim hesaplanır ve iki değişken bir adım “kaydırılır”:

```
int adet = 5;
long a = 0, b = 1;
for (int i = 0; i < adet; i++) {
    System.out.print(a + " ");
    long sonraki = a + b;
    a = b;
    b = sonraki;
}
```

i	Yazılan a	sonraki = a + b	Yeni a	Yeni b
0	0	1	1	1
1	1	2	1	2
2	1	3	2	3
3	2	5	3	5
4	3	8	5	8

Tablo 8.23: İlk 5 Fibonacci sayısı.



Şekil 8.19: Kayan pencere. Her turda a (turuncu) ve b (mavi) bir adım sağa kayar; a sırasıyla 0, 1, 1, 2, 3 değerlerini alır.

Çıktı `0 1 1 2 3`'tür. Kaydırmanın sırası önemlidir: önce `a = b`, sonra `b = sonraki`. Sıra ters yazılırsa (önce `b = sonraki`, sonra `a = b`) a, b'nin eski değerini değil yeni değerini alır. İki değişken hep eşit olur ve her terim bir öncekinin iki katı çıkar: `0 1 2 4 8`. `sonraki` değişkeni tam da bu nedenle vardır; eski değeri kaybetmeden yeni değeri tutar.

Dizi üstel hızla büyür. Bu yüzden değişkenler long seçilmiştir. long bile sonsuz değildir. 93. terim 7.540.113.804.746.346.429'dur ve long'a sığar. 94. terim long'un en büyük değerini (9.223.372.036.854.775.807) aşar; program onu negatif bir sayı olarak yazar.

8.9 Girdiyle çalışan döngüler

Şimdiye kadarki döngülerin tur sayısı programın içindeki sayılara bağlıydı. Kullanıcıdan girdi okuyan döngülerde ise döngünün ne zaman biteceğine kullanıcı karar verir. Bu döngülerde tur sayısı baştan bilinmez ve doğal seçim while ya da do-while olur.

8.9.1 Sayı tahmin oyunu

Bilgisayar 1 ile 100 arasında gizli bir sayı tutar. Kullanıcı tahmin eder; program “Yukarı” ya da “Aşağı” diyerek yol gösterir. Doğru tahmin yapılınca kaçıncı denemede bilindiği yazılır.

```
Random rnd = new Random();
Scanner klavye = new Scanner(System.in);
int hedef = rnd.nextInt(100) + 1;  // 1..100
int tahmin = 0, deneme = 0;
while (tahmin != hedef) {
  tahmin = klavye.nextInt();
  deneme++;
  if (tahmin < hedef) System.out.println("Yukarı");
  if (tahmin > hedef) System.out.println("Aşağı");
}
System.out.println(deneme + ". denemede bildin");
```

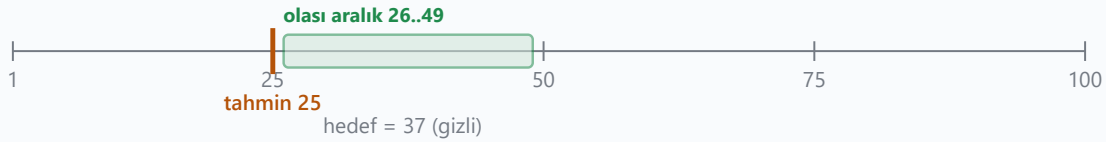
Rastgele sayı üretmek için `java.util.Random` sınıfı kullanılır; dosyanın başına `Random` ve `Scanner` için iki import satırı gerekir. `rnd.nextInt(100)` 0 ile 99 arasında bir sayı verir; 1 eklenince aralık 1..100 olur. Her çalıştırmada farklı bir hedef çıkar.

Döngüden önce `tahmin = 0` verilmesinin nedeni koşuldur. Hedef 1..100 arasında olduğu için 0 hiçbir zaman hedef olamaz. Böylece `tahmin != hedef` ilk denetimde kesinlikle doğru çıkar ve döngüye girilir. Aynı iş do-while ile de yazılabilirdi; o durumda bu yapay başlangıç değerine gerek kalmazdı.

Hedefin 37 çıktığı bir çalıştırmada kullanıcı sırayla 50, 25 ve 37 girsin. Tablo 8.24 oyunun üç turunu gösterir. Tablo, rastgele sayı üreticisine 37'yi verecek bir başlangıç tohumu verilerek JDK 21 ile alınmıştır.

Tur	tahmin	deneme	İleti	Olası aralık
1	50	1	Aşağı	1…49
2	25	2	Yukarı	26…49
3	37	3	yok	bulundu

Tablo 8.24: Sayı tahmin oyunu, hedef = 37, tahminler 50, 25, 37.



Şekil 8.20: İkinci tahminden sonra olası aralık 26..49'dur. Her tahmin, hedefin bulunabileceği aralığı daraltır.

İlk tahmin 50'dir. *50 > 37* olduğu için "Aşağı" yazılır; hedef artık 1..49 aralığındadır. İkinci tahmin 25'tir; "Yukarı" yazılır ve aralık 26..49'a iner. Üçüncü tahmin 37'dir. İki if de yanlıştır ve hiçbir ileti yazılmaz. Koşul *37 != 37* yanlış çıkar ve döngü biter. Program "3. denemede bildin" yazar.

Her seferinde aralığın ortasını söylemek, aralığı yaklaşık yarıya indirir. 100 sayılık bir aralık 7 yarılamada tek sayıya iner; bu yüzden bu stratejiyle her hedef en çok 7 denemede bulunur. Bu düşünce ikili aramanın temelidir.

8.9.2 Bekçi değerle okuma

Kullanıcı kaç sayı gireceğini önceden söylemezse verinin bittiği nasıl anlaşılır? Yaygın çözüm, veri olamayacak özel bir değeri "bitti" işareti olarak kullanmaktır.

> **Tanım 8.15**
>
> **Bekçi değer** (*sentinel value*), bir veri dizisinin sonunu işaretleyen ve kendisi veriye katılmayan özel değerdir. Bekçi değer, gerçek verinin hiçbir zaman alamayacağı bir değer olmalıdır.

```
Scanner sc = new Scanner(System.in);
int toplam = 0;
while (true) {
    int sayi = sc.nextInt();
    if (sayi == 0) break;
    toplam += sayi;
}
System.out.println("Toplam: " + toplam);
```

Döngü bilerek sonsuz yazılmıştır: `while (true)`. Çıkış gövdenin ortasındaki break ile sağlanır. Sayı okunur okunmaz bekçi değer olup olmadığına bakılır; bekçi değilse toplama eklenir. Tablo 8.25 girdi 5, 3, 0 için turları gösterir.

Okunan sayi	sayi == 0	toplam
5	`false`	5
3	`false`	8
0	`true`	8 (break)

Tablo 8.25: Bekçi değerle okuma, girdi 5, 3, 0.

Program "Toplam: 8" yazar. 0 toplama eklenmemiştir. Kullanıcı ilk sayı olarak 0 girerse döngü ilk turda biter ve "Toplam: 0" yazılır. Girdi 7, −2, 4, 0 için toplam 9'dur; negatif sayılar bu programda geçerli veridir, çünkü bekçi yalnızca 0'dır.

Sık yapılan hata

Bekçiyi işlemden sonra denetlemek ya da veri olabilecek bir değeri bekçi seçmek. `toplam += sayi;` satırı if'ten önce yazılırsa bekçi de toplanır. Bekçi 0 iken bu fark etmez ama bekçi −1 olsaydı toplam 1 eksik çıkardı. Not ortalaması gibi bir programda ise 0 gerçek bir not olabilir. Bekçi değer 0 seçilirse 0 alan öğrencinin notu "veri bitti" diye yorumlanır. Böyle durumlarda −1 gibi veri aralığının dışında bir bekçi seçilir.

Bekçi değerli okumanın tipik bir uzantısı ortalama hesabıdır. Toplamın yanında bir sayaç tutulur ve döngüden sonra ortalama *toplam / adet* olarak hesaplanır. İki ayrıntı unutulmamalıdır: hiç sayı girilmemişse (adet = 0) bölme yapılmaz ve bölmeden önce `(double)` dönüşümü yapılır. 80, 64, 90, 0 girdisi için toplam 234, ortalama 78,0 çıkar; 6, 9, 0 için ortalama 7,5'tir. Dönüşüm yapılmasaydı tam sayı bölmesi *15 / 2 = 7* verirdi.

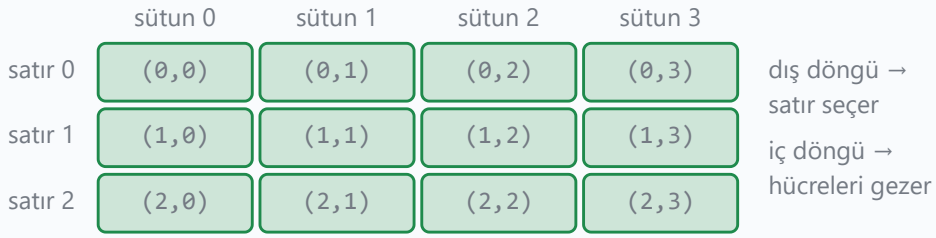
8.10 İç içe döngüler

8.10.1 Satır ve sütun

Bir döngünün gövdesinde başka bir döngü bulunabilir. Bu yapıya **iç içe döngü** (*nested loop*) denir. Dıştaki döngüye dış döngü, içtekine iç döngü denir. Tablolar, ızgaralar, çarpım tabloları ve ekran desenleri satır-sütun düzenindedir; bu tür çıktıların doğal aracı iç içe döngüdür.

```
for (int satir = 0; satir < 3; satir++) {
  for (int sutun = 0; sutun < 4; sutun++) {
    System.out.printf("(%d,%d) ", satir, sutun);
  }
  System.out.println();
}
```

```
(0,0) (0,1) (0,2) (0,3)
(1,0) (1,1) (1,2) (1,3)
(2,0) (2,1) (2,2) (2,3)
```



Şekil 8.21: 3 × 4 ızgara. Dış döngü satırı seçer, iç döngü o satırın hücrelerini soldan sağa gezer.

İç içe döngünün temel kuralı şudur: iç döngü, dış döngünün *her* turunda baştan sona tamamlanır. satir = 0 iken iç döngü sutun = 0, 1, 2, 3 için dört kez çalışır ve (0,0)'dan (0,3)'e kadar dört hücre yazar. sutun 4 olunca iç döngü biter. Ardından `println()` satırı sonlandırır. Dış döngü satir++ ile 1'e geçer ve iç döngü sutun = 0'dan *yeniden* başlar. sutun değişkeni iç döngünün başlığında tanımlandığı için her dış turda yeniden oluşturulur ve 0 değerini alır.

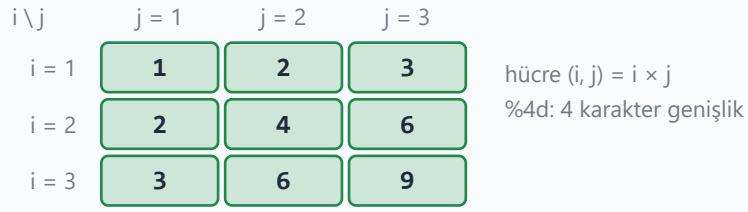
İç gövde toplam *satır sayısı × sütun sayısı = 3 × 4 = 12* kez çalışır. `printf` içindeki `%d` yer tutucuları, virgülden sonra verilen değerlerle sırayla doldurulur. `println()` ise dış döngünün gövdesinde ama iç döngünün *dışında* yer alır. Bu konum her satırın sonunda bir kez alt satıra geçilmesini sağlar.

İç döngünün sınırı dış döngünün değişkenine bağlı olabilir. O durumda iç gövdenin çalışma sayısı bir çarpım değil, bir toplam olur. Örneğin i 1'den 4'e, j de i'den 4'e kadar dönerse iç gövde *4 + 3 + 2 + 1 = 10* kez çalışır. Artan üçgen deseni bu yapının bir örneğidir.

8.10.2 Çarpım tablosu

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        System.out.printf("%4d", i * j);
    }
    System.out.println();
}
```

```
   1   2   3
   2   4   6
   3   6   9
```



Şekil 8.22: 3 × 3 çarpım tablosu. (i, j) hücresine i × j yazılır; `%4d` her sayıyı 4 karakterlik bir alana sağa yaslar.

Dış döngü i = 1 iken iç döngü j = 1, 2, 3 için 1, 2, 3 yazar. j = 4'te iç koşul yanlış olur ve println satırı bitirir. Aynı iş i = 2 ve i = 3 için tekrarlanır. `%4d` biçimi sayıları sütunlar hâlinde hizalar: tek basamaklı sayıların önüne üç, iki basamaklı sayıların önüne iki boşluk eklenir. Sınırlar 10 yapıldığında 10 × 10'luk klasik çarpım tablosu çıkar; en büyük sayı 100 olduğu için 4 karakterlik alan yine yeter.

8.11 Desenler

Yıldızlardan ya da sayılardan oluşan ekran desenleri iç içe döngü alıştırmasının klasik konusudur. Desenlerin kendisi önemli değildir; önemli olan, bir çıktının düzenini sayılarla ifade edip o sayıları döngü sınırlarına çevirmektir. Bu beceri tablolar, raporlar ve matrislerle çalışırken de gerekir.

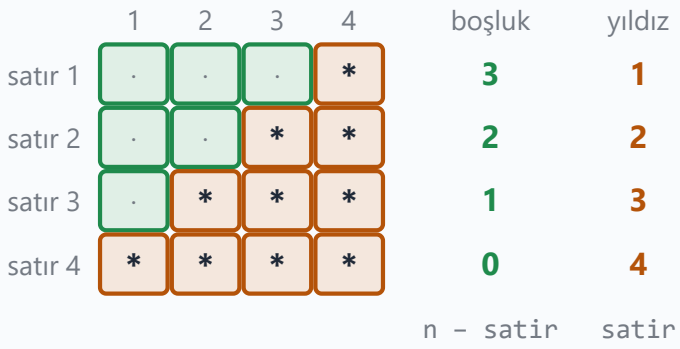
Tanım 8.16

Desen (*pattern*), satır satır yazılan ve her satırdaki boşluk ve karakter sayısı satır numarasına göre düzenli değişen çıktıdır. Dış döngü satırları, iç döngüler ise bir satırdaki boşluk ve karakter parçalarını üretir.

8.11.1 Desen problemini çözme yöntemi

Bir desen dört adımda koda dönüştürülür:

1. **Çiz.** Beklenen çıktı küçük bir n için (örneğin n = 4) kareli kâğıda çizilir. Her kare bir karakterdir; boşluklar da kare sayılır.
2. **Say.** Her satırdaki boşluk ve karakter sayıları bir tabloya yazılır.
3. **İlişkilendir.** Sayılar satır numarasıyla ilişkilendirilir. Örneğin 3, 2, 1, 0 dizisi n − satır, 1, 2, 3, 4 dizisi satır, 1, 3, 5, 7 dizisi 2 × satır − 1'dir.
4. **Kodla.** Dış döngü satır sayısı kadar döner. Satırdaki her parça (boşluklar, yıldızlar) için bir iç döngü yazılır; iç döngünün sınırı 3. adımda bulunan formüldür. Satır sonunda println çağrılır.



Şekil 8.23: n = 4 sağa yaslı üçgen kareli kâğıtta. Boşluk sütunu 3, 2, 1, 0 (n − satır), yıldız sütunu 1, 2, 3, 4'tür (satır).

Tablo 8.26 dört temel desenin formüllerini toplar. Sağdaki sayılar, desenler JDK 21 ile n = 4 için gerçekten üretilip her satırdaki boşluk ve yıldızlar sayılarak elde edilmiştir.

Desen	Boşluk	Karakter
Artan üçgen	0	satır
Azalan üçgen	0	n − satır + 1
Sağa yaslı	n − satır	satır
Piramit	n − satır	2 × satır − 1

Tablo 8.26: Satıra göre boşluk ve karakter sayıları.

Satır	Artan	Azalan	Sağa yaslı	Piramit
1	0 + 1	0 + 4	3 + 1	3 + 1
2	0 + 2	0 + 3	2 + 2	2 + 3
3	0 + 3	0 + 2	1 + 3	1 + 5
4	0 + 4	0 + 1	0 + 4	0 + 7

Tablo 8.27: n = 4 için her satırın “boşluk + yıldız” sayısı.



Şekil 8.24: n = 4 için dört desen: artan üçgen, azalan üçgen, sağa yaslı üçgen ve piramit. Kesikli kareler boşluk, dolu kareler karakterdir.

8.11.2 Artan üçgen ve sayı üçgeni

Artan üçgende satır r'de r yıldız vardır ve boşluk yoktur. Tek bir iç döngü yeter; üst sınırı dış döngünün değişkenidir:

```
int n = 4;
for (int satir = 1; satir <= n; satir++) {
    for (int sutun = 1; sutun <= satir; sutun++) {
        System.out.print("*");
    }
    System.out.println();
}
```

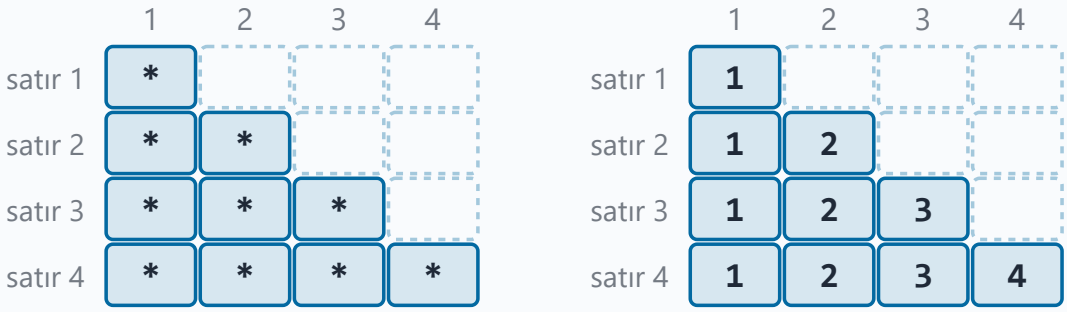
```
*
**
***
****
```

satir = 1 iken iç döngü yalnızca sutun = 1 için çalışır ve bir yıldız yazar. sutun = 2'de *2 <= 1* yanlıştır ve println alt satıra geçer. satir = 2 iken iç döngü iki kez, satir = 3 iken üç kez, satir = 4 iken dört kez döner. satir = 5'te dış koşul yanlış olur. Toplam *1 + 2 + 3 + 4 = 10* yıldız yazılır. Genel olarak n satırlık artan üçgende *n(n + 1) / 2* yıldız vardır; n = 4 için *4 · 5 / 2 = 10*.

Aynı iskelet karakter yerine sayı da yazabilir. Sayı üçgeninde her hücreye iç döngünün değişkeni yazılır:

```
int n = 4;
for (int satir = 1; satir <= n; satir++) {
    for (int deger = 1; deger <= satir; deger++) {
        System.out.print(deger + " ");
    }
    System.out.println();
}
```

```
1
1 2
1 2 3
1 2 3 4
```



Şekil 8.25: n = 4 için artan üçgen ve sayı üçgeni. İki desenin yapısı aynıdır; yalnızca hücreye yazılan değer değişir.

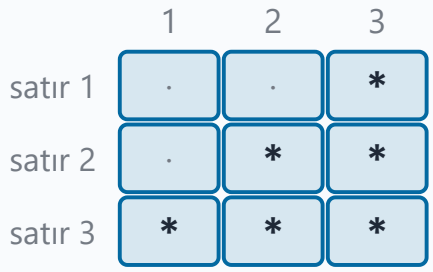
Dış döngü yine satırı, iç döngünün üst sınırı satırdaki eleman sayısını belirler. Fark yalnızca `print` çağrısının içindedir: `"*"` yerine `deger + " "` yazılır. Her sayıdan sonra bir boşluk bırakıldığı için satırların sonunda da bir boşluk vardır; ekranda görünmez. İç döngünün değişkeni yerine dış döngünün değişkeni yazılsaydı her satırda aynı sayı tekrarlanırdı: 1, 2 2, 3 3 3, 4 4 4 4.

8.11.3 Boşluklu desenler: sağa yaslı üçgen ve piramit

Sağa yaslı üçgende her satır iki parçadan oluşur: önce boşluklar, sonra yıldızlar. Boşluk sayısı azalırken yıldız sayısı artar. Her parça için ayrı bir iç döngü yazılır:

```
int n = 3;
for (int satir = 1; satir <= n; satir++) {
    for (int bosluk = 0; bosluk < n - satir; bosluk++)
        System.out.print(" ");
    for (int yildiz = 0; yildiz < satir; yildiz++)
        System.out.print("*");
    System.out.println();
}
```

```
  *
 **
***
```



Şekil 8.26: n = 3 sağa yaslı üçgen. Satır 1'de 2 boşluk ve 1 yıldız, satır 3'te hiç boşluk yok ve 3 yıldız vardır.

n = 3 için satir = 1'de boşluk döngüsü *n − satir = 2* kez döner, yıldız döngüsü 1 kez. satir = 2'de 1 boşluk ve 2 yıldız yazılır. satir = 3'te boşluk döngüsünün koşulu *0 < 3 − 3* baştan yanlıştır; döngü hiç çalışmaz ve satır doğrudan 3 yıldızla başlar. Her satır tam n = 3 karakterdir. Bu denetim, boşluk ve yıldız formüllerinin birbiriyle tutarlı olduğunu gösterir: *(n − satır) + satır = n*.

Piramit aynı iskeletin bir çeşitlemesidir. Boşluk sayısı yine n − satır, yıldız sayısı ise 2 × satır − 1'dir. Yıldızlar her satırda ikişer artar; boşluklar birer azalır ve yıldız grubu ortada kalır. Aşağıdaki program n = 4 için önce piramidi, sonra karşılaştırma için azalan üçgeni çizer:

```
public class Piramit {
    public static void main(String[] args) {
        int n = 4;
        for (int satir = 1; satir <= n; satir++) {
            for (int b = 0; b < n - satir; b++) System.out.print(" ");
            for (int y = 0; y < 2 * satir - 1; y++) System.out.print("*");
            System.out.println();
        }
        for (int satir = 1; satir <= n; satir++) {
            for (int y = 1; y <= n - satir + 1; y++) System.out.print("*");
            System.out.println();
        }
    }
}
```

```
   *
  ***
 *****
*******
****
***
**
*
```

Çıktının ilk dört satırı piramittir. Son dört satır aynı programın devamındaki azalan üçgendir: yıldız sayısı *n − satır + 1*, yani 4, 3, 2, 1'dir. Piramitteki toplam yıldız sayısı *1 + 3 + 5 + 7 = 16 = 4²*'dir. İlk n tek sayının toplamı her zaman n^2'dir.

8.11.4 İçi boş kare

Bazı desenlerde bir hücreye ne yazılacağı yalnızca satıra değil, hem satıra hem sütuna bağlıdır. İçi boş karede kenardaki hücreler yıldız, içteki hücreler boşluktur. İç döngünün gövdesine bir if-else girer:

```
int n = 4;
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    if (i == 0 || i == n - 1 || j == 0 || j == n - 1)
      System.out.print("*");
    else
      System.out.print(" ");
  }
  System.out.println();
}
```

```
****
*  *
*  *
****
```

Bir hücre dört kenardan birindeyse yıldızdır: ilk satır (`i == 0`), son satır (`i == n - 1`), ilk sütun (`j == 0`) ya da son sütun (`j == n - 1`). Dört koşul `||` ile bağlanmıştır; biri doğruysa yeter. Kısa devre nedeniyle ilk doğru koşuldan sonrakiler hesaplanmaz. Tablo 8.28 her hücrede hangi koşulun karar verdiğini gösterir.

i	j = 0	j = 1	j = 2	j = 3
0	* (i == 0)	* (i == 0)	* (i == 0)	* (i == 0)
1	* (j == 0)	boşluk	boşluk	* (j == n - 1)
2	* (j == 0)	boşluk	boşluk	* (j == n - 1)
3	* (i == n - 1)	* (i == n - 1)	* (i == n - 1)	* (i == n - 1)

Tablo 8.28: İçi boş kare (n = 4); her hücrede doğru çıkan ilk koşul.

16 hücrenin 12'si kenar, 4'ü iç hücredir. else dalındaki boşluk yazımı zorunludur. Bu satır atlanırsa iç hücrelere hiçbir şey yazılmaz. Sağ kenardaki yıldız sola kayar ve ortadaki satırlar `**` olarak görünür; kare bozulur.

8.11.5 Desenlerde sık hatalar



Şekil 8.27: Solda doğru artan üçgen. Sağda println iç döngüye yazıldığında her yıldız ayrı bir satıra düşer.

Sık yapılan hata

`println` çağrısını iç döngünün içine koymak. Bu durumda her yıldızdan sonra satır biter. n = 3 için artan üçgen yerine alt alta *1 + 2 + 3 = 6* satır, her satırda tek yıldız çıkar. println iç döngünün kapanış parantezinden sonra, dış döngünün içine yazılmalıdır.

Desenlerdeki diğer tipik yanlışlar şunlardır. `<` ile `<=` karıştırılırsa her satırda bir eksik ya da bir fazla karakter çıkar; sayaç 0'dan başlıyorsa `<`, 1'den başlıyorsa `<=` kullanılır. Boşluk döngüsü yıldız döngüsünden sonra yazılırsa boşluklar satırın sonuna düşer, görünmez ve desen sola yaslanır. Dış döngünün değişkeni iç döngüde yanlışlıkla değiştirilirse (örneğin iç başlıkta `satir++` yazılırsa) satırlar atlanır ya da döngü bitmez. Her desen, küçük bir n için izleme tablosuyla satır satır denetlenerek bu hatalardan korunur.

8.12 Test tasarımı

Bir döngünün tek bir girdiyle doğru çalışması, her girdide doğru çalışacağını göstermez. Döngü hataları genellikle sınırlarda ortaya çıkar: hiç dönmemesi gereken, tek tur dönen ya da en büyük değeri işleyen durumlarda. İyi bir test kümesi bu durumların her birinden bir örnek içerir.

Tanım 8.17

Test durumu (*test case*), bir program için seçilen girdi ile o girdide beklenen çıktının ikilisidir. **Sınır durumu** (*edge case*), girdi aralığının ucundaki ya da özel davranış gerektiren değerdir: 0, 1, negatif sayılar ve tipin en büyük ya da en küçük değeri gibi.

Bir test kümesi şu adımlarla hazırlanır:

1. Her problem için sıradan, “normal” bir değer seçilir.

2. 0, 1, negatif değerler ve üst sınıra yakın değerler ayrıca seçilir.
3. Her girdi için beklenen sonuç, kod çalıştırılmadan elle hesaplanır.
4. Program çalıştırılır. Sonuç beklenenden farklıysa değişkenler tur tur izlenir ve ilk sapmanın olduğu tur bulunur.

Basamak sayısı programı için dört test yeterli bir başlangıçtır. Tablo 8.29 bu testleri ve JDK 21 ile alınan sonuçları verir.

Basamak sayısı testi	sayı	beklenen	JDK 21 sonucu
Normal	4096	4	4
Sıfır	0	1	1
Negatif	−507	3	3
Uç değer	−2147483648	10	10

Tablo 8.29: Basamak sayısı testleri (long kopyalı do-while çözümü).

Her test belirli bir hatayı yakalamak için seçilmiştir. 0 testi, döngü while ile yazılmışsa 0 sonucunu verir ve hatayı gösterir. Negatif test, işaret atılmamışsa döngünün hiç çalışmadığını ortaya çıkarır. Uç değer testi, kopya int olarak tutulmuşsa yanlış sonuç (1) verir. Normal test ise kodun temel mantığını doğrular. Dört testin dördünden de geçen bir çözüm, bu hataların hiçbirini içermez.

Aynı düşünce her döngüye uygulanır. Bir toplam döngüsü n = 0 (hiç tur), n = 1 (tek tur) ve büyük bir n ile denenir. Bir asal testi 0, 1, 2, 4, bir kare sayı (örneğin 25) ve büyük bir asal ile denenir. Bir desen programı n = 1 ve n = 0 ile de çalıştırılır. Döngünün sıfır kez, bir kez ve çok kez döndüğü durumların her biri en az bir testle kapsanmalıdır.

8.13 Özet

- Her döngü dört parçadan oluşur: ilk değer verme, koşul, gövde ve güncelleme. Güncelleme koşulu sona yaklaştırır; unutulursa döngü bitmez.
- `for` döngüsünde üç parça başlıkta toplanır ve sıra “ilk değer verme → koşul → gövde → güncelleme → koşul …” biçimindedir. 1'den n'ye dönen bir for döngüsünde gövde n, koşul n + 1 kez çalışır.
- `while` koşulu gövdeden önce, `do-while` gövdeden sonra sorar. while gövdesi hiç çalışmayabilir; do-while gövdesi en az bir kez çalışır.
- Tekrar sayısı belliyse for, koşul sürdükçe while, en az bir tur gerekiyorsa do-while seçilir. Üçü birbirine çevrilebilir; seçim okunabilirlik içindir.
- `break` en yakın döngüyü bitirir, `continue` yalnızca o turun kalanını atlar. Etiketli break iç içe döngülerden birden çıkar.
- Sonsuz döngü yanlış yönde ya da hiç yapılmayan güncellemeden doğar. double sayaç yuvarlama hatası biriktirir; döngü sayacı int olmalıdır.

- Döngü değişmezi her turun başında doğru kalan bir özelliktir; döngü bittiğinde sonucun doğruluğunu gösterir. Sınır hatalarına karşı ilk tur, son tur ve tur sayısı denetlenir.
- Rakam kalıbı (`% 10` son rakamı verir, `/ 10` onu siler) basamak sayısı, rakamlar toplamı, ters çevirme, palindrom, Armstrong sayısı ve taban değiştirmede kullanılır.
- Bölen kalıbı `n % i == 0` denetimine dayanır. Mükemmel sayıda n / 2'ye, asal testinde √n'ye kadar aramak yeter.
- Bekçi değer veri olamayacak bir değerdir; veri sonunu işaretler ve kendisi işlenmez.
- İç içe döngüde iç döngü dış döngünün her turunda baştan sona tamamlanır. Desenlerde dış döngü satırı, iç döngüler satırdaki boşluk ve karakterleri üretir; sayılar satır numarasına bağlanır.
- Döngüler normal, 0, 1, negatif ve uç değerlerle test edilir; beklenen sonuç önceden elle hesaplanır.

8.14 Kavram listesi

döngü *loop* · döngü gövdesi *loop body* · tur *iteration* · ilk değer verme *initialization* · koşul *condition* · döngü değişkeni *loop variable* · kapsam *scope* · sayaç *counter* · biriktirici *accumulator* · artırma / azaltma *increment / decrement* · ön koşullu döngü *pre-test loop* · son koşullu döngü *post-test loop* · menü *menu* · girdi doğrulama *input validation* · break *break statement* · etiketli break *labeled break* · sonsuz döngü *infinite loop* · taşma *overflow* · kayan noktalı sayı *floating-point number* · döngü değişmezi *loop invariant* · sınır hatası *off-by-one error* · kalıp *pattern* · rakam *digit* · tam sayı bölmesi *integer division* · palindrom *palindrome* · Armstrong sayısı *Armstrong number* · üs / taban *exponent / base* · mükemmel sayı *perfect number* · bölen *divisor* · asal sayı *prime number* · bayrak *flag* · kısa devre değerlendirme *short-circuit evaluation* · Fibonacci dizisi *Fibonacci sequence* · rastgele sayı *random number* · bekçi değer *sentinel value* · iç içe döngü *nested loop* · iç döngü / dış döngü *inner loop / outer loop* · desen *pattern (output)* · kod izleme *code tracing* · izleme tablosu *trace table* · test durumu *test case* · sınır durumu *edge case*

8.15 Kendini Dene

1. `for (int i = 2; i <= 20; i *= 3) System.out.print(i + " ");` döngüsü ne yazar? Gövde ve

koşul kaçar kez çalışır?

Cevap

i sırasıyla 2, 6, 18 değerlerini alır ve `2 6 18` yazılır. 18'den sonra *i = 18 × 3 = 54* olur ve *54 <= 20* yanlıştır. Gövde 3 kez, koşul 4 kez çalışır. Güncelleme her zaman 1 artırma olmak zorunda değildir; burada sayaç çarpılarak ilerler.

2. `int x = 1; while (x < 100) x *= 2;` döngüsünden sonra x kaçtır? Döngü kaç tur döner?

Cevap

x sırasıyla 2, 4, 8, 16, 32, 64, 128 olur. 64 < 100 doğru olduğu için bir tur daha döner ve x = 128 olur; *128 < 100* yanlıştır. Döngü 7 tur döner ve x = 128'dir. Döngü, koşulu yanlış yapan ilk değerde biter; bu değer sınırı aşmış olabilir.

3. `int k = 10; do { k -= 4; } while (k > 0);` çalıştıktan sonra k kaçtır? Gövde kaç kez çalışır? Aynı döngü `while (k > 0) k -= 4;` ve başlangıç değeri k = −1 ile yazılsaydı ne olurdu?

Cevap

k sırasıyla 6, 2, −2 olur. Her turun sonunda koşul sorulur; 6 ve 2 için doğru, −2 için yanlıştır. Gövde 3 kez çalışır, k = −2'dir. k = −1 ile başlayan while döngüsünde *−1 > 0* baştan yanlıştır; gövde hiç çalışmaz ve k = −1 kalır. Aynı başlangıçla do-while bir kez çalışır ve k = −5 olurdu.

4. Aşağıdaki döngü ne yazar? `for (int i = 1; i <= 10; i++) { if (i % 2 == 0) continue; if (i > 7) break; System.out.print(i + " "); }`

Cevap

Çift sayılar continue ile atlanır. Tek sayılardan 1, 3, 5 ve 7 yazılır. i = 9'da 9 tektir, ilk koşul yanlıştır, *9 > 7* doğrudur ve break döngüyü bitirir. Çıktı `1 3 5 7` 'dir. i = 8 continue ile atlandığı için break'e 8'de değil 9'da sıra gelir.

5. Rakam kalıbıyla 4096'nın rakamlar toplamı ve tersi nedir? 1200'ün tersi ve palindrom testi ne verir? 0 için ters sayı ve palindrom testi ne verir?

Cevap

4096: rakamlar sağdan sola 6, 9, 0, 4'tür. Toplam *6 + 9 + 0 + 4 = 19*, ters 6904'tür; 6904 ≠ 4096 olduğundan palindrom değildir. 1200: rakamlar 0, 0, 2, 1'dir ve ters sayı *0 → 0 → 2 → 21* diye kurulur; sonuç 21'dir, palindrom değildir. 0 için while döngüsü hiç çalışmaz; ters = 0 kalır ve *0 == 0* olduğundan test `true` verir. 0 tek rakamlı bir sayı olarak gerçekten palindromdur.

6. Aşağıdaki iç içe döngüden sonra sayac kaçtır? `for (int i = 1; i <= 4; i++) for (int j = i + 1; j <= 4; j++) sayac++;` (sayac başta 0)

Cevap

i = 1 için j = 2, 3, 4 (3 tur); i = 2 için j = 3, 4 (2 tur); i = 3 için j = 4 (1 tur); i = 4 için iç döngü hiç dönmez (*5 <= 4* yanlış). Toplam *3 + 2 + 1 + 0 = 6*'dır. Bu sayı, 4 elemandan seçilebilecek farklı ikililerin sayısıdır: *4 · 3 / 2 = 6*.

7. n = 4 için aşağıdaki desen isteniyor. Satır r'deki boşluk ve yıldız sayıları nedir? İç döngülerin sınırları ne olmalıdır?

```
****
 ***
  **
   *
```

Cevap

Satır 1'de 0 boşluk ve 4 yıldız, satır 2'de 1 boşluk ve 3 yıldız, satır 3'te 2 boşluk ve 2 yıldız, satır 4'te 3 boşluk ve 1 yıldız vardır. Boşluk sayısı *r − 1*, yıldız sayısı *n − r + 1*'dir. Her satır *(r − 1) + (n − r + 1) = n* karakterdir. Kod: `for (int b = 0; b < satir - 1; b++)` boşluklar için, `for (int y = 0; y < n - satir + 1; y++)` yıldızlar için.

8. `for (double d = 0; d != 1.0; d += 0.25)` döngüsü biter mi? Kaç tur döner? Aynı döngü `d += 0.1` ile yazılsaydı ne olurdu?

Cevap

0,25 = 1/4 ikili tabanda tam gösterilir ($0{,}01_2$). d sırasıyla 0, 0,25, 0,5, 0,75 ve tam 1,0 olur. Döngü 4 tur döner ve biter. 0,1 ise ikili tabanda tam gösterilemez; 10 toplamadan sonra d = 0,9999999999999999 olur ve hiçbir turda tam 1,0'a eşit olmaz. `!=` koşuluyla bu döngü bitmez. Güvenli yol int sayaçtır.

9. Asal testi (`asal && i <= n / i` koşuluyla) n = 97 için çalıştırılıyor. Kaç bölen adayı denenir? Döngü hangi i değerinde ve neden biter?

Cevap

i = 2, 3, ..., 9 denenir; hiçbiri 97'yi bölmez. Bu 8 adaydır. i = 9'da *9 <= 97 / 9 = 10* doğrudur. i = 10'da *10 <= 97 / 10 = 9* yanlıştır ve döngü biter. √97 ≈ 9,85 olduğundan daha büyük bölen aramaya gerek yoktur. Program `true` yazar.

8.16 İleri okuma

- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), Kesim 1.3: koşullar ve döngüler Java'da, izleme tablolarıyla ve kuvvetler, harmonik sayılar, Newton yöntemi ve ikili gösterim gibi kısa programlarla anlatılır; kesimin çevrim içi sürümü introcs.cs.princeton.edu adresindedir.
- A. B. Downey, C. Mayfield, *Think Java* (2. baskı, O'Reilly, 2019): while ve for döngüleri ile iç içe döngüler programlamaya yeni başlayanlar için sade bir dille anlatılır; kitabın çevrim içi sürümü ücretsizdir.
- Oracle, *The Java Tutorials*: while ve do-while, for ve dallanma deyimleri sayfaları; etiketli break ve continue'nun resmî örnekleri bu sayfalardadır.

8.17 Kaynaklar

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11. baskı, Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13. baskı, Pearson, 2024.
3. Oracle, *The Java Tutorials: while and do-while, for, Branching Statements*.
4. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 1.3 Conditionals and Loops.
5. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019.

KISIM III · BÖLÜM 9

Metotlar

Program büyüdükçe aynı işler farklı yerlerde tekrar tekrar yazılmaya başlar ve `main` metodu okunamayacak kadar uzar. Metot, bir görevi bir ad altında toplar. Görev bir kez yazılır, sonra adıyla istenen yerden çağrılır. Not önce bir metodun nasıl tanımlandığını ve çağrıldığını anlatır. Ardından bir çağrı sırasında bellekte olanlar gelir: argümanların parametrelere kopyalanması, her çağrıya açılan çerçeve ve bu çerçevelerin oluşturduğu çağrı yığını. Dönüş değeri, `void` metotlar ve erken dönüş ayrı ayrı ele alınır. Java'nın argümanları neden her zaman değer ile aktardığı, swap örneğiyle adım adım gösterilir. Aşırı yükleme, değişken sayıda argüman, kapsam ve gölgeleme konularında derleyicinin verdiği gerçek hata iletileri de incelenir. Son kesimler iyi metot tasarımını ve tek/çift, basamak sayısı, asal sayı, EBOB, EKOK gibi örnek metotları kapsar. Nottaki bütün programlar JDK 21 ile derlenip çalıştırılmıştır.

Bu bölümün kodu: `github.com/sercankulcu/computer-programming-java/Bolum09/src`

ÖĞRENME ÇIKTILARI

Tanımlama
Bir metodun başlığını ve gövdesini yazmak; metodu `main` 'den ve başka metotlardan çağırmak; `main` metodunun imzasını ve `static` niteleyicisinin anlamını açıklamak.

Veri akışı
Parametre ile argümanı ayırt etmek; dönüş değerini bir ifadede kullanmak; `void` metodu ve erken dönüşü doğru yerde kullanmak; Java'da argümanların değer ile aktarıldığını izleyerek göstermek; aşırı yükleme ve değişken sayıda argümanla çalışmak.

Kapsam
Yerel değişkenin nerede görünür olduğunu ve ne kadar yaşadığını açıklamak; çağrı yığınındaki çerçeveleri adım adım izlemek; gölgelemeyi ve kapsamla ilgili derleme hatalarını tanımak.

Tasarım
Bir problemi tek görevli metotlara bölmek; metodun sözleşmesini yazmak; sonucu ekrana yazmak yerine döndüren, kısa, odaklı ve test edilebilir metotlar kurmak.

9.1 Metot nedir?

Şimdiye kadar yazılan Java programlarının bütün işi tek bir yerde, `main` metodunun içinde yapıldı. Kısa programlarda bu yeterlidir. Ama program birkaç yüz satıra ulaşınca aynı hesap üç dört yerde yeniden yazılır. Bir hata düzeltildiğinde öteki kopyalar unutulur. `main`'in neyi hangi sırayla yaptığı da okunamaz hâle gelir. Metotlar bu sorunun çözümüdür.

9.1.1 Bir görevi adlandırmak

Matematikte bir fonksiyon, girdisine karşılık bir çıktı üretir: $f(x) = x^2$ için $f(3) = 9$'dur. Programlamadaki fonksiyon da benzer bir birimdir. Belirli bir görevi yapan kod parçasına bir ad verilir. Bu kod girdilerini dışarıdan alır, işini yapar ve gerekiyorsa bir sonuç verir. Java'da her şey bir sınıfın içinde yazıldığı için fonksiyonlar da sınıfın içinde tanımlanır ve bunlara **metot** denir.

Tanım 9.1

Metot (*method*), bir sınıfın içinde tanımlanan, adı olan ve belirli bir görevi yapan deyimler bütünüdür. Metot girdilerini **parametreleri** aracılığıyla alır, sonucunu **dönüş değeri** olarak verir. Metodun adıyla birlikte parantez içinde değerler yazılarak metot çalıştırılır; buna **çağrı** (*method call*) denir.

Java'nın kendi kütüphanesindeki metotlar bu fikri iyi gösterir. `Math.sqrt(16)` çağrısı 16'nın karekökünü, yani `4.0`'ı verir. Karekökün hangi yöntemle hesaplandığını bilmek gerekmez. Bilinmesi gereken üç şey vardır: metodun adı (`sqrt`), ne istediği (bir sayı) ve ne verdiği (karekökü). Metodu kullanan için iç ayrıntı gizlidir. Şekil 9.1 bu durumu bir kara kutu olarak gösterir.



Şekil 9.1: Metot bir kara kutu gibidir. `topla` iki girdi (5 ve 3) alır ve bir sonuç (8) verir. Girdiler parametrelerle içeri girer, sonuç dönüş değeri olarak çıkar; gövdedeki ayrıntı dışarıdan görünmez.

Şekildeki kutunun solunda iki girdi, sağında bir çıktı vardır. Kutunun içinde ne olduğu ancak metodun gövdesine bakılınca görülür. Metodu çağıran kod yalnızca "bu iki sayıyı ver, toplamı al" diye düşünür. Bu ayrım, metotların en önemli kazancıdır: bir işin *ne* yaptığı ile *nasıl* yaptığı birbirinden ayrılır.

9.1.2 Neden metot yazılır?

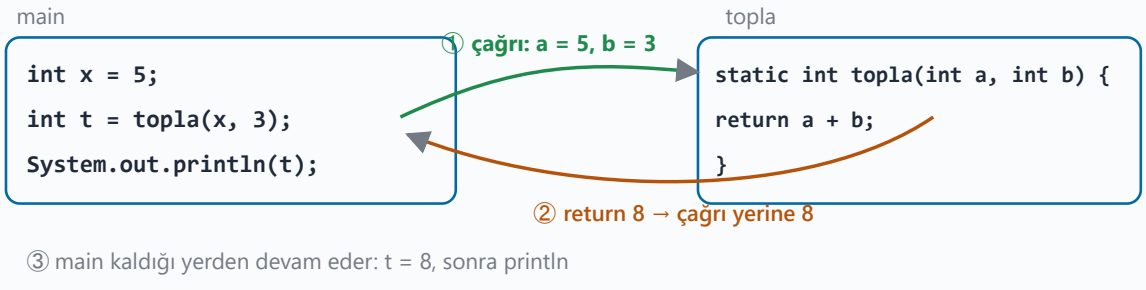
Bir öğrenci üç dersin notlarını ekrana yazdıran bir program yazıyor olsun. Her dersin başına ve sonuna çizgi çekmek istiyor. Metot olmadan çizgiyi çizen döngü altı kez kopyalanır. Çizginin uzunluğu değişecek olsa altı yerin hepsi düzeltilmelidir; biri unutulursa çıktı tutarsız olur. Bu döngü `cizgiCiz` adlı bir metoda konursa altı yerin her birinde tek satırlık bir çağrı kalır: `cizgiCiz(5);`. Uzunluk değişirse yalnızca argüman değişir.

Metot yazmanın kazançları şöyle sıralanabilir:

- **Tekrarın önlenmesi.** Aynı kod bir kez yazılır. Bir hata bulunduğunda tek yerde düzeltilir ve düzeltme her çağrıya yansır.
- **Okunabilirlik.** İyi adlandırılmış bir metot çağrısı ne yapıldığını söyler. `if (asalMi(n))` satırı, bölen arayan bir döngüden çok daha kolay okunur.
- **Test edilebilirlik.** Küçük bir metot, programın geri kalanından bağımsız olarak birkaç örnek girdiyle denenebilir. `basamakSayisi(-507)` 'nin 3 vermesi tek satırla kontrol edilir.
- **Yeniden kullanım** (*reuse*). Bir kez yazılan metot başka programlarda, başka metotların içinde de kullanılır. EKOK hesaplayan bir metot, EBOB metodunu çağırarak işini kısaltır.
- **İş bölümü.** Büyük bir problem küçük alt görevlere bölünür. Her alt görev ayrı bir metot olur ve ayrı ayrı yazılıp denenir.

9.1.3 Bir çağrıda neler olur?

Bir metot çağrıldığında programın akışı geçici olarak çağrıyı yapan yerden ayrılır, metodun gövdesine gider ve metot bitince kaldığı yere döner. Şekil 9.2 bu gidiş dönüşü gösterir.



Şekil 9.2: Bir çağrının yolculuğu. (1) `main`, `topla(x, 3)` 'ü çağırır; kontrol metoda geçer ve a = 5, b = 3 olur. (2) `return` 8'i geri gönderir; 8 çağrının yerine geçer. (3) `main` kaldığı yerden sürer: t = 8 olur, sonra `println` çalışır.

Bu yolculuk her çağrıda aynı beş adımdan oluşur:

1. **Argümanlar değerlendirilir.** Çağrıdaki parantezin içindeki ifadelerin değeri bulunur. `topla(x, 3)` 'te x'in o anki değeri (5) okunur.
2. **Değerler parametrelere kopyalanır.** Metot için bellekte yeni bir alan açılır. Birinci değer birinci parametreye, ikinci değer ikinci parametreye yazılır: a = 5, b = 3.
3. **Gövde çalışır.** Metodun küme parantezleri arasındaki deyimler sırayla çalışır.
4. **Değer döndürülür.** `return` deyimine gelindiğinde metot biter ve `return` 'ün yanındaki değer çağıran yere gönderilir.
5. **Çağıran kod devam eder.** Çağrı ifadesinin yerine dönen değer geçer. Program, çağrının bulunduğu satırdan sürer.

Bu adımlarda gözden kaçmaması gereken bir nokta vardır: her çağrı kendi parametreleri ve kendi yerel değişkenleriyle çalışır. Aynı metot iki kez çağrılırsa iki ayrı bellek alanı açılır. İkinci çağrı, birincinin değişkenlerini görmez; birinci çağrı bittiğinde onun değişkenleri zaten silinmiştir. Değer ile aktarım, kapsam ve çağrı yığını bu kurala dayanır.

9.2 Metot tanımı ve çağrısı

Bir metodu kullanmak için önce onu *tanımlamak*, yani adını, girdilerini, sonucunun tipini ve gövdesini yazmak gerekir. Tanımlanmış metot daha sonra istenen yerden ve istenen sayıda *çağrılır*. Tanımlama bir kez yapılır, çağrı ise defalarca.

9.2.1 İlk metot: topla

İki tam sayının toplamını veren `topla` metodu ve onu kullanan iki satır şöyledir:

```
static int topla(int a, int b) {
    int sonuc = a + b;
    return sonuc;
}

int toplam = topla(5, 3);
System.out.println(toplam);
```

İlk dört satır tanımlamadır. `static int topla(int a, int b)` satırı metodun *başlığıdır*. `int` , metodun bir tam sayı döndüreceğini söyler. `topla` metodun adıdır. Parantezin içindeki `int a, int b` , metodun iki tam sayı beklediğini ve bunlara metodun içinde `a` ve `b` adıyla erişileceğini bildirir. Küme parantezleri arasındaki iki satır metodun *gövdesidir*: önce `sonuc` adlı bir değişkene a + b yazılır, sonra bu değer `return` ile geri verilir. Son iki satır ise çağrıdır: `topla(5, 3)` metodu 5 ve 3 ile çalıştırır, dönen değer `toplam` 'a atanır ve ekrana yazılır.

Bu parçalar tek başına derlenmez; bir sınıfın içinde olmaları gerekir. Tam program aşağıdadır. Metot `main` ile aynı sınıfın içinde, `main`'in dışında yazılır. Metotlar iç içe tanımlanamaz: `topla`'yı `main`'in küme parantezlerinin içine yazmak derleme hatasıdır.

```
public class Topla {
    static int topla(int a, int b) {
        int sonuc = a + b;
        return sonuc;
    }

    public static void main(String[] args) {
        int toplam = topla(5, 3);
        System.out.println(toplam);
    }
}
```

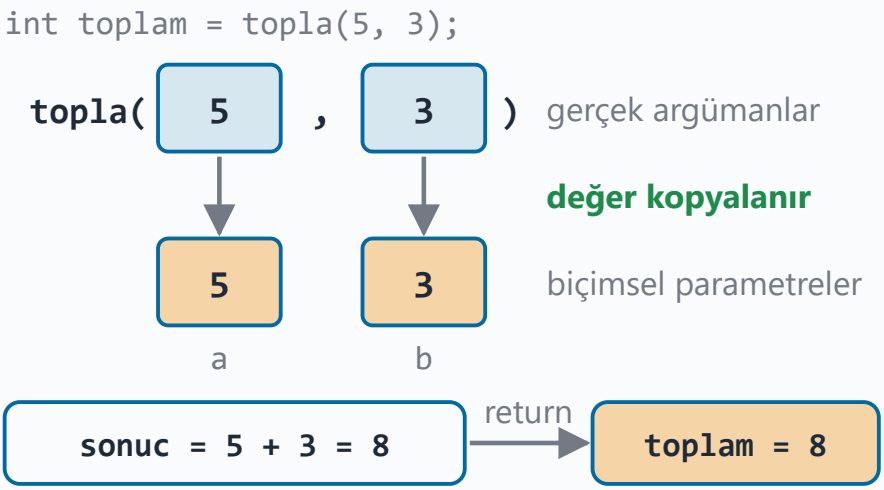
```
8
```

Program çalışmaya her zaman `main`'den başlar. `topla` sınıfta `main`'den önce yazılmış olsa da kendiliğinden çalışmaz; yalnızca çağrıldığında çalışır. Metotların sınıf içindeki sırası önemli değildir. `topla`, `main`'in altına da yazılabilirdi.

Tanımlamadaki `a` ve `b` ile çağrıdaki `5` ve `3` farklı rolleri üstlenir ve farklı adlar alır.

Tanım 9.2

Parametre (*parameter*), ya da tam adıyla **biçimsel parametre** (*formal parameter*), metot başlığında tipiyle birlikte bildirilen değişkendir; metodun hangi girdileri beklediğini söyler. **Argüman** (*argument*), ya da **gerçek argüman** (*actual argument*), çağrı sırasında parantezin içine yazılan ve parametreye kopyalanan değerdir.



Şekil 9.3: `int toplam = topla(5, 3);` çağrısında gerçek argümanlar (5 ve 3) biçimsel parametrelere (a ve b) kopyalanır. Gövdede sonuc = 5 + 3 = 8 hesaplanır ve `return` 8'i `toplam`'a gönderir.

Parametre bir değişkendir; argüman ise bir değerdir ya da değeri hesaplanacak bir ifadedir. Parametreler metot her çağrıldığında yeniden doğar ve o çağrının argümanlarıyla dolar. Argümanlar parametrelere *sırayla* eşlenir: birinci argüman birinci parametreye, ikinci argüman ikinci parametreye. Argüman sayısı parametre sayısıyla aynı olmalı, her argümanın tipi de karşılık gelen parametreye uymalıdır. `topla(5)` ya da `topla(5, 3, 1)` çağrıları derlenmez, çünkü `topla` tam olarak iki tam sayı bekler.

```
public class EksikArguman {
    static int topla(int a, int b) { return a + b; }
    public static void main(String[] args) {
        System.out.println(topla(5));
    }
}
```

```
EksikArguman.java:4: error: method topla in class EksikArguman cannot be applied to given
types;
        System.out.println(topla(5));
                           ^
  required: int,int
  found:    int
  reason: actual and formal argument lists differ in length
1 error
```

Derleyici bu kez daha uzun bir ileti verir: `topla` metodu verilen tiplere uygulanamaz. `required: int,int` satırı metodun beklediği parametre tiplerini, `found: int` satırı çağrıda bulunan argümanları gösterir. Son satır nedeni söyler: gerçek ve biçimsel argüman listelerinin uzunlukları farklıdır.

Sık yapılan hata

Çağrıda argümanın tipini yeniden yazmak: `topla(int 5, int 3)` . Tipler yalnızca tanımlamada, parametrelerin yanında yazılır. Çağrıda yalnızca değerler ya da ifadeler bulunur: `topla(5, 3)` . Tersine, tanımlamada parametre tipini unutmak da hatadır: `static int topla(a, b)` derlenmez. Her parametrenin kendi tipi ayrı ayrı yazılır; `int a, b` biçimi parametre listesinde geçerli değildir.

9.2.2 Çağrının sonucunu kullanmak

Değer döndüren bir metodun çağrısı, Java'da bir *ifadedir*. Çağrı çalıştıktan sonra yerine dönen değer geçer. Bu yüzden çağrı, bir değerin kullanılabildiği her yerde kullanılabilir: bir atamanın sağında, bir aritmetik işlemin içinde, bir koşulda ya da başka bir metodun argümanı olarak. Aşağıdaki program dört farklı kullanımı bir arada gösterir:

```
public class ToplaKullan {
    static int topla(int a, int b) {
        return a + b;
    }

    public static void main(String[] args) {
        int x = 5;
        int t = topla(x, 3);                    // değişken ve sabit argüman
        System.out.println(t);
        System.out.println(topla(10, 20) * 2);   // çağrı bir ifadenin parçası
        System.out.println(topla(topla(1, 2), 4)); // çağrının sonucu başka çağrıya argüman
        topla(7, 8);                            // dönen değer kullanılmadı
    }
}
```

```
8
60
7
```

İlk çağrı bir değişkeni ve bir sabiti argüman olarak alır. x'in değeri 5 olduğundan *topla(5, 3) = 8* hesaplanır. İkinci çağrıda `topla(10, 20)` önce 30'u döndürür, ardından bu değer 2 ile çarpılır ve 60 yazılır. Üçüncü satırda bir çağrının sonucu başka bir çağrıya argüman olur. Java önce içteki `topla(1, 2)` 'yi çalıştırır, 3'ü bulur; sonra dıştaki çağrı `topla(3, 4)` olarak çalışır ve 7 yazılır.

Son satırdaki `topla(7, 8);` da geçerli bir deyimdir. Metot 15'i döndürür ama bu değer hiçbir yerde kullanılmadığı için atılır. Java, dönen değerin kullanılmasını zorunlu tutmaz. Ancak bu satır programa hiçbir şey katmaz: `topla` yalnızca hesap yapar, ekrana yazmaz, hiçbir değişkeni değiştirmez. Değer döndüren bir metot çağrılıp sonucu atılıyorsa bu genellikle bir unutkanlığın işaretidir.

Bir ifadede birden fazla çağrı varsa Java argümanları ve işlenenleri soldan sağa değerlendirir. Aşağıdaki küçük metot, çağrıldığını anlamak için argümanını köşeli parantez içinde yazıyor ve iki katını döndürüyor:

```
public class SiraOrnek {
    static int iki(int x) {
        System.out.print("[" + x + "] ");
        return 2 * x;
    }

    public static void main(String[] args) {
        int r = iki(3) - iki(1);
        System.out.println(r);
        System.out.println(iki(iki(1)));
    }
}
```

```
[3] [1] 4
[1] [2] 4
```

`iki(3) - iki(1)` ifadesinde önce soldaki çağrı çalışır ve `[3]` yazar, sonra sağdaki çalışır ve `[1]` yazar. Çıkarma iki değer de hazır olunca yapılır: *6 − 2 = 4*. Son satırdaki iç içe çağrıda ise içteki `iki(1)` önce çalışmak zorundadır, çünkü dıştaki çağrının argümanı onun sonucudur. Önce `[1]`, sonra `[2]` yazılır ve 4 basılır. Ekrana yazdırmak ile değer döndürmek birbirinden bağımsızdır. Metot ekrana ne yazarsa yazsın, ifadeye katılan yalnızca dönüş değeridir.

9.2.3 topla(5, 3) adım adım

Bir çağrıda olup biteni ayrıntısıyla görmek için kod satır satır yürütülür. Satır numaraları aşağıdaki koda göredir; 7. ve 8. satırlar `main`'in içindedir.

```
static int topla(int a, int b) {
    int sonuc = a + b;
    return sonuc;
}

// main içinde:
int toplam = topla(5, 3);
System.out.println(toplam);
```

Tablo 9.1 her adımda hangi satırın çalıştığını, bellekte hangi çerçevelerin açık olduğunu ve en üstteki çerçevedeki değişkenleri verir. Çerçeve, bir çağrıya ayrılmış bellek alanıdır; kesin tanımı çağrı yığını ile birlikte verilir.

Adım	Satır	Çağrı yığını (üstteki önce)	Değişkenler (üstteki çerçeve)	Çıktı
1	7	`main()`	—	—
2	1	`topla(5, 3) ← main()`	a=5, b=3	—
3	2	`topla(5, 3) ← main()`	a=5, b=3, sonuc=8	—
4	3	`topla(5, 3) ← main()`	a=5, b=3, sonuc=8	—
5	7	`main()`	toplam=8	—
6	8	`main()`	toplam=8	8

Tablo 9.1: `topla(5, 3)` çağrısının izleme tablosu. Çağrı yığını sütununda en üstteki çerçeve önce yazılmıştır.

Yürütme `main`'deki 7. satırla başlar. Atamanın sağ tarafı hesaplanmadan `toplam`'a bir değer yazılamaz. Sağ tarafta bir çağrı olduğu için önce o çalışmalıdır. 2. adımda `topla` için yeni bir çerçeve açılır ve argümanlar kopyalanır: a = 5, b = 3. 3. adımda yerel değişken `sonuc` oluşur ve 8 değerini alır. Bu değişken yalnızca `topla`'nın çerçevesinde yaşar; `main` onu göremez. 4. adımda `return sonuc;` çalışır, 8 çağırana gönderilir ve `topla`'nın çerçevesi kapanır. 5. adımda yürütme 7. satıra döner. Çağrının yerine 8 geçmiştir ve atama tamamlanır: toplam = 8. Bu noktada a, b ve sonuc artık yoktur. 6. adımda 8 ekrana yazılır.



Şekil 9.4: `topla(5, 3)` çağrısının üç anı. (a) `topla`'nın çerçevesi `main`'in üstüne açıldı: a = 5, b = 3. (b) `return 8` : çerçeve kapanıyor. (c) `main`'e dönüldü: toplam = 8; a, b ve sonuc artık yok.

İzleme tablosunda dikkat çeken bir ayrıntı vardır: 7. satır iki kez görünür. İlk kez çağrı başlarken, ikinci kez çağrı bittikten sonra atama tamamlanırken. Bir satırın "yarıda bırakılıp" sonra sürdürülmesi, metot çağrılarının doğal bir sonucudur. Yürütme bir çağrıya rastladığında bulunduğu satırı askıya alır, metodu bitirir ve aynı satıra geri gelir.

9.3 Çağrı yığını

`topla` örneğinde yalnızca iki metot vardı: `main` ve `topla`. Gerçek programlarda bir metot başka bir metodu, o da bir başkasını çağırır. Bir anda birden çok metot "yarıda" bekler. Java hangi metodun nerede kaldığını ve hangi değişkenlerin kime ait olduğunu bir yığın yapısıyla izler.

9.3.1 Yığın ve çerçeve

Yemekhanede üst üste konmuş tabaklar düşünülsün. Yeni tabak hep en üste konur, alınan tabak da hep en üstten alınır. En son konan tabak ilk alınır. Bu düzene *son giren ilk çıkar* denir ve böyle davranan yapıya **yığın** (*stack*) adı verilir. Metot çağrıları da tam olarak bu düzende açılır ve kapanır.

Tanım 9.3

Çerçeve (*stack frame*), bir metot çağrısı için bellekte ayrılan alandır. Çerçevede o çağrının parametreleri, yerel değişkenleri ve metot bitince yürütmenin nereye döneceği bilgisi tutulur. **Çağrı yığını** (*call stack*), o anda devam eden bütün çağrıların çerçevelerinden oluşan yığındır. Her çağrı yığının üstüne yeni bir çerçeve koyar; metot bitince en üstteki çerçeve kaldırılır.

Program başladığında yığında tek bir çerçeve vardır: `main`'in çerçevesi. `main` bir metodu çağırınca o metodun çerçevesi `main`'in üstüne konur. O metot da bir başkasını çağırırsa yığın bir kat daha yükselir. Bu düzenden iki kural çıkar. Birincisi, yalnızca en üstteki çerçevenin metodu çalışır; alttakiler, üstlerindeki çağrının bitmesini bekler. İkincisi, en son çağrılan metot ilk biter. Bir metot, kendisinin çağırdığı metot bitmeden bitemez.

9.3.2 ortalama(9, 7) adım adım

Aşağıdaki örnekte `ortalama` metodu iki sayının ortalamasını bulmak için `topla` metodunu çağırır. Böylece yığında üç çerçeve birden bulunur:

```
static int topla(int a, int b) {
    return a + b;
}
static int ortalama(int x, int y) {
    int t = topla(x, y);
    return t / 2;
}
int o = ortalama(9, 7);  // main içinde
System.out.println(o);
```

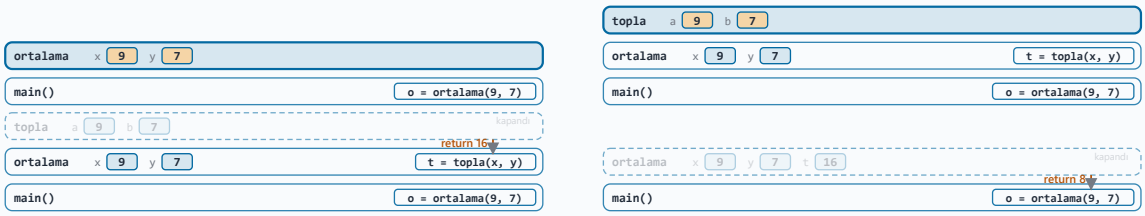
Satır numaraları: 1–3 `topla`, 4–7 `ortalama`, 8–9 `main`'deki iki satırdır. Tablo 9.2 dokuz adımı verir.

Adım	Satır	Çağrı yığını (üstteki önce)	Değişkenler (üstteki çerçeve)	Çıktı
1	8	`main()`	—	—
2	4	`ortalama(9, 7) ← main()`	x=9, y=7	—
3	5	`ortalama(9, 7) ← main()`	x=9, y=7	—
4	1	`topla(9, 7) ← ortalama(9, 7) ← main()`	a=9, b=7	—
5	2	`topla(9, 7) ← ortalama(9, 7) ← main()`	a=9, b=7	—
6	5	`ortalama(9, 7) ← main()`	x=9, y=7, t=16	—
7	6	`ortalama(9, 7) ← main()`	x=9, y=7, t=16	—
8	8	`main()`	o=8	—
9	9	`main()`	o=8	8

Tablo 9.2: `ortalama(9, 7)` çağrısının izleme tablosu. 4. ve 5. adımlarda yığında üç çerçeve vardır.

1. adımda `main`, `ortalama(9, 7)`'yi çağırır; yığında henüz tek çerçeve vardır. 2. adımda `ortalama`'nın çerçevesi `main`'in üstüne konur ve argümanlar kopyalanır: x = 9, y = 7. 3. adımda `ortalama`, 5. satırda kendisi bir çağrı yapar: `topla(x, y)`. Gönderilen, x ve y değişkenlerinin kendisi değil, değerleridir: 9 ve 7. 4. adımda `topla`'nın çerçevesi en üste konur ve yığında üç çerçeve olur. Bu anda yalnızca `topla` çalışır. `ortalama` 5. satırda, `main` de 8. satırda bekler.

5. adımda `topla` 16'yı döndürür ve çerçevesi kaldırılır. Son açılan çerçeve ilk kapanmıştır. 6. adımda yürütme `ortalama`'nın 5. satırına döner: çağrının yerine 16 geçer ve t = 16 olur. 7. adımda `return t / 2;` çalışır. İki tam sayının bölümü tam sayı bölmesidir: *16 / 2 = 8*. `ortalama`'nın çerçevesi de kapanır. 8. adımda `main`'e dönülür ve o = 8 olur. Bu anda x, y, t, a ve b değişkenlerinin hiçbiri artık yoktur. 9. adımda 8 yazılır.



Şekil 9.5: Çağrı yığınının dört anı. (a) `ortalama`'nın çerçevesi `main`'in üstünde: x = 9, y = 7. (b) `topla` en üstte; yığında üç çerçeve var. (c) `topla` 16 döndürüp kapanıyor. (d) `ortalama` 8 döndürüp kapanıyor; yığında yine yalnızca `main` kalacak.

Programın tamamı ve gerçek çıktısı aşağıdadır. İkinci satırdaki `ortalama(9, 8)` çağrısı tam sayı bölmesinin etkisini gösterir: *9 + 8 = 17* ve *17 / 2 = 8*'dir. Gerçek ortalama 8,5 olsa da dönüş tipi `int` ve bölme iki `int` arasında olduğu için kesir atılır.

```
public class Ortalama {
    static int topla(int a, int b) {
        return a + b;
    }

    static int ortalama(int x, int y) {
        int t = topla(x, y);
        return t / 2;
    }

    public static void main(String[] args) {
        int o = ortalama(9, 7);
        System.out.println(o);
        System.out.println(ortalama(9, 8));
    }
}
```

```
8
8
```

Sık yapılan hata

Metotlardaki aynı adları aynı değişken sanmak. `ortalama`'nın x'i ile `main`'deki bir x, ya da `topla`'nın a'sı ile başka bir metodun a'sı birbirinden tamamen bağımsızdır. Her biri kendi çerçevesindeki ayrı bir bellek kutusudur. Değer bir metottan ötekine yalnızca iki yolla geçer: çağrıda argüman olarak ya da `return` ile dönüş değeri olarak.

9.3.3 Yığının kuralları

Çağrı yığını Java'ya özgü değildir; C, Python ve hemen her programlama dilindeki metot ya da fonksiyon çağrıları aynı biçimde işler. İzleme sırasında şu kurallar akılda tutulursa çoğu sorunun cevabı kendiliğinden çıkar:

- Her çağrı yeni bir çerçeve açar. Aynı metot iki kez çağrılırsa iki ayrı çerçeve açılır, ikinci çağrı birincinin değişkenlerini görmez.
- Yalnızca en üstteki çerçeve çalışır. Alttaki her çerçeve, bir çağrının ortasında, o çağrının sonucunu bekler.
- Metot bitince çerçevesi bütün içeriğiyle silinir. Yerel değişkenler ve parametreler bir çağrıdan ötekine değer taşımaz.
- Dönen değer, çağrının yapıldığı ifadede çağrının yerine geçer ve hesap oradan sürer.
- Son açılan çerçeve ilk kapanır. Çerçeveler açıldıkları sıranın tersiyle kapanır.

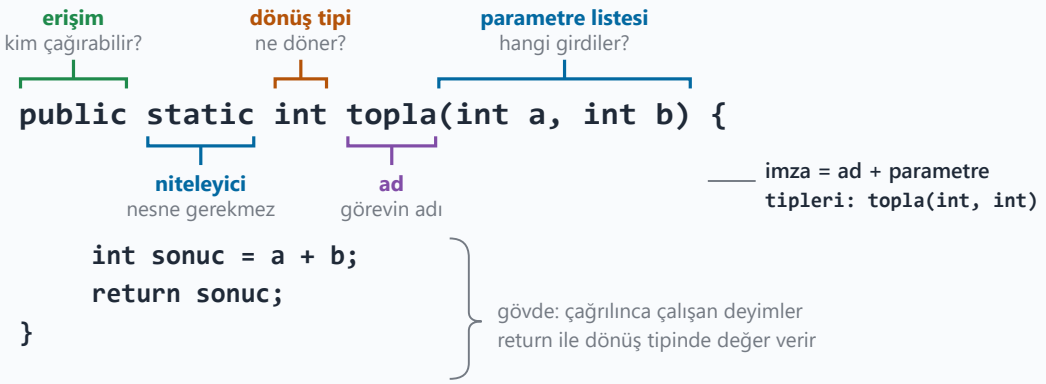
Yığının bir sınırı da vardır. Java her iş parçacığına sınırlı büyüklükte bir yığın ayırır. Birbirini sonu gelmeden çağıran metotlar yığını doldurursa program `StackOverflowError` hatasıyla durur. Sıradan programlarda yığın derinliği birkaç ya da birkaç düzine çerçeveyi geçmez. Metot Çağrı Yığını simülatörü, çerçevelerin açılıp kapanışını adım adım canlandırır.

9.4 Metot tanımlamasının parçaları

Bir metot tanımlaması iki kısımdan oluşur: başlık ve gövde. Başlık metodun dış dünyaya görünen yüzüdür. Kimin çağırabileceğini, ne döndüreceğini, adını ve hangi girdileri beklediğini söyler. Gövde ise metodun işini yapan deyimleri içerir.

9.4.1 Başlık ve gövde

Şekil 9.6 `public static int topla(int a, int b)` başlığını parçalarına ayırır.



Şekil 9.6: Bir metot tanımlamasının parçaları: erişim belirteci (`public`), niteleyici (`static`), dönüş tipi (`int`), ad (`topla`), parametre listesi (`int a, int b`) ve küme parantezleri arasındaki gövde. Ad ile parametre tipleri birlikte metodun imzasını oluşturur: `topla(int, int)` .

Parça	Örnek	Cevapladığı soru	Açıklama
erişim belirteci	`public`	Kim çağırabilir?	`public` her yerden, `private` yalnızca aynı sınıftan çağrılabilir. Yazılmazsa aynı paketteki sınıflar çağırabilir.
niteleyici	`static`	Nesne gerekir mi?	`static` metot sınıfa aittir; çağırmak için nesne oluşturmak gerekmez.
dönüş tipi	`int`	Ne döner?	Metodun vereceği değerin tipi. Değer döndürmüyorsa `void` .
ad	`topla`	Görevin adı nedir?	Metodun ne yaptığını söyleyen tanımlayıcı.
parametre listesi	`(int a, int b)`	Hangi girdiler?	Virgülle ayrılmış “tip ad” çiftleri; girdi yoksa boş parantez `()` .
gövde	`{ … }`	Nasıl yapılır?	Çağrılınca çalışan deyimler; `return` ile dönüş tipinde bir değer verir.

Tablo 9.3: Metot başlığının parçaları ve cevapladıkları sorular.

Başlıktaki parçaların sırası sabittir: önce erişim belirteci ve niteleyici, sonra dönüş tipi, sonra ad ve parametre listesi. `int static topla(…)` gibi bir sıralama derlenmez. Erişim belirteci ve `static` isteğe bağlıdır; dönüş tipi, ad ve parantezler ise zorunludur. Bu nottaki örneklerde metotlar `main` ile aynı sınıfta ve `static` olarak yazılır. Erişim belirteci çoğunlukla yazılmaz, çünkü metotlar zaten yalnızca aynı sınıftan çağrılır.

Tanım 9.4

İmza (*method signature*), bir metodun adı ile parametrelerinin tiplerinin sırayla yazılmış listesidir. `static int topla(int a, int b)` metodunun imzası `topla(int, int)` 'dir. Dönüş tipi, parametre adları ve niteleyiciler imzaya dahil değildir. Bir sınıfta aynı imzalı iki metot bulunamaz.

İmza, derleyicinin bir çağrıyı hangi metoda bağlayacağına karar verirken baktığı bilgidir. `topla(5, 3)` çağrısını gören derleyici, adı `topla` olan ve iki `int` kabul eden bir metot arar. Parametre adlarının (`a` , `b`) imzada yer almaması doğaldır, çünkü çağrıda bu adlar hiç görünmez. Dönüş tipi de imzada yer almaz, çünkü çağrının kendisinde dönen değerin tipine dair bir ipucu yoktur.

9.4.2 Metot adları

Metot adları değişken adlarıyla aynı kurallara uyar: harfle başlar, harf, rakam ve alt çizgi içerebilir, Java'nın ayrılmış sözcüklerinden biri olamaz. Java geleneğinde metot adları küçük harfle başlar ve birden çok sözcük birleşirse sonraki sözcüklerin ilk harfi büyük yazılır: `basamakSayisi`, `yazdirPozitif`, `enBuyuk3`. Türkçe karakterler (ç, ğ, ı, ö, ş, ü) Java'da geçerlidir ama dosyaların farklı bilgisayarlarda sorunsuz açılması için çoğu zaman kullanılmaz.

İyi bir metot adı metodun ne yaptığını söyler. Bir iş yapan metotlar bir eylemle adlandırılır: `cizgiCiz`, `raporYaz`, `notlariOku`. Bir değer hesaplayan metotlar o değerin adını alır: `ortalama`, `alan`, `ebob`. `boolean` döndüren metotlar ise bir soru gibi okunur: `ciftMi`, `asalMi`. Böyle adlandırılan bir metot bir koşulda kullanıldığında satır neredeyse düz Türkçe gibi okunur: `if (asalMi(n))`, “n asal mı?”.

9.4.3 main metodu ve static

Her Java programında bulunan `main` de bir metottur ve başlığındaki her sözcüğün bir nedeni vardır.

Parça	Neden?
`public`	JVM (Java Sanal Makinesi) metodu sınıfın dışından çağırabilsin.
`static`	Metot, hiçbir nesne üretilmeden çağrılabilsin.
`void`	`main` bir değer döndürmez.
`main`	JVM'nin programı başlatmak için aradığı ad.
`String[] args`	Program çalıştırılırken komut satırından verilen argümanlar.

Tablo 9.4: `public static void main(String[] args)` başlığının parçaları.

`static` niteleyicisinin anlamı şöyledir. Java nesneye yönelik bir dildir; çoğu metot bir nesneye aittir ve o nesne üzerinden çağrılır. `static` metot ise bir nesneye değil, sınıfın kendisine aittir. Çağırmak için önce bir nesne oluşturmak gerekmez. JVM programı başlattığında henüz hiçbir nesne yoktur. `main`'i çalıştırabilmesi için bu metodun nesne gerektirmemesi, yani `static` olması zorunludur.

Bu durumun bir sonucu vardır: `static` bir metot, aynı sınıftaki `static` olmayan bir metodu ya da alanı doğrudan kullanamaz, çünkü ortada o metodun ait olacağı bir nesne yoktur. Aşağıda bu hatayı gösteren örneğin tam bir sınıf içindeki hâli ve JDK 21 derleyicisinin verdiği ileti yer alıyor:

```
public class Ornek {
    void selamla() { }
    public static void main(String[] args) {
        selamla();   // hata
    }
}
```

```
Ornek.java:4: error: non-static method selamla() cannot be referenced from a static
context
        selamla();   // hata
        ^
1 error
```

İleti, "`static` olmayan `selamla()` metoduna `static` bir bağlamdan başvurulamaz" der. Derleyici hatanın satırını (`Ornek.java:4`) ve `^` işaretiyle yerini gösterir. Bu notun kapsamında çözüm basittir: çağrılan metodu da `static` yapmak.

```
public class StaticCozum {
    static void selamla() {
        System.out.println("Merhaba");
    }

    public static void main(String[] args) {
        selamla();
    }
}
```

```
Merhaba
```

`main`'in parametresi `String[] args`, program çalıştırılırken komut satırına yazılan sözcükleri bir dizi olarak alır. Aşağıdaki program kaç argüman geldiğini ve bu argümanları yazar.

```
public class MainArgs {
    public static void main(String[] args) {
        System.out.println(args.length);
        for (String a : args) System.out.println(a);
    }
}
```

`java MainArgs` komutuyla argümansız çalıştırıldığında `args` boş bir dizidir ve program yalnızca 0 yazar. `java MainArgs Ali 42` ile çalıştırıldığında çıktı şudur:

```
2
Ali
42
```

Komut satırındaki her sözcük ayrı bir `String` olarak gelir. `42` de bir sayı değil, iki karakterlik bir metindir.

Sık yapılan hata

`main`'in başlığını eksik ya da farklı yazmak. `static` unutulursa ya da ad `Main` yazılırsa program derlenebilir, ama çalıştırılırken JVM başlangıç noktasını bulamaz ve programı başlatmaz. Java büyük-küçük harf ayrımı yapar: `main` ile `Main` iki ayrı addır.

9.5 Dönüş değeri

Bir metot işini bitirince çağırana bir sonuç verebilir ya da hiçbir şey vermeden yalnızca bir iş yapabilir. Hangisinin olacağını başlıktaki dönüş tipi belirler. Sonuç `return` deyimiyle verilir.

9.5.1 return deyimi

Tanım 9.5

Dönüş tipi (*return type*), metodun vereceği değerin tipidir ve başlıkta adın önüne yazılır. **Dönüş değeri** (*return value*), metodun `return` deyimiyle çağırana gönderdiği değerdir. `return ifade;` deyimi ifadenin değerini hesaplar, metodu hemen bitirir ve bu değeri çağrının yapıldığı yere **döndürür**.

`return` iki iş birden yapar: bir değer gönderir ve metodu bitirir. `return`'den sonra aynı yolda yazılmış deyimler çalışmaz. Derleyici, `return`'den hemen sonra gelen ve hiçbir yolla ulaşılamayacak bir deyimi “ulaşılamaz deyim” (`unreachable statement`) hatasıyla reddeder.

Döndürülen değerin tipi dönüş tipine uymalıdır. `int` döndüren bir metot `return 8;` yazabilir, `return "sekiz";` yazamaz. Dönüş tipi `double` ise `return 8;` geçerlidir; `int` değer kendiliğinden `double`'a genişletilir ve çağırana `8.0` gider. Tersi geçerli değildir: `int` döndüren metot `return 2.5;` yazamaz, çünkü kesirli kısım kaybolurdu.

9.5.2 void metotlar

Her metodun bir değer hesaplaması gerekmez. Ekrana bir çizgi çizen, bir rapor yazan ya da bir dosyaya kayıt ekleyen metodun çağırana gönderecek bir sonucu yoktur. Bu metotların dönüş tipi `void`'dir. `void` "boş" demektir: metot çağrıya bir değer döndürmez.

```
static void cizgiCiz(int n) {
    for (int i = 0; i < n; i++)
        System.out.print("-");
    System.out.println();
}
// main içinde:
cizgiCiz(5);          // -----
int x = cizgiCiz(5);  // HATA: void değer vermez
```

`cizgiCiz` metodu n tane tire yazar ve satırı bitirir. Metodun işi bir **yan etkidir**: ekranın durumunu değiştirir ama çağırana bir değer vermez. Tam program ve çıktısı:

```
public class CizgiCiz {
    static void cizgiCiz(int n) {
        for (int i = 0; i < n; i++)
            System.out.print("-");
        System.out.println();
    }

    public static void main(String[] args) {
        cizgiCiz(5);
        System.out.println("Metotlar");
        cizgiCiz(8);
        cizgiCiz(0);
        System.out.println("son");
    }
}
```

```
-----
Metotlar
--------

son
```

`cizgiCiz(5)` beş, `cizgiCiz(8)` sekiz tire yazar. `cizgiCiz(0)` çağrısında döngü hiç dönmez; yalnızca `println()` çalışır ve çıktıda boş bir satır oluşur. Örnekteki son satır, `int x = cizgiCiz(5);`, derlenmez. Metot bir değer vermediği için atamanın sağ tarafında kullanılamaz:

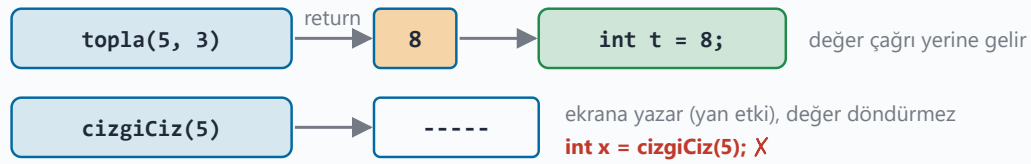
```
VoidAtama.java:9: error: incompatible types: void cannot be converted to int
        int x = cizgiCiz(5);   // HATA: void değer vermez
                         ^
1 error
```

İleti "uyumsuz tipler: `void`, `int`'e dönüştürülemez" der. Aynı nedenle `void` bir çağrı başka bir metoda argüman da olamaz:

```java
public class VoidYaz {
    static void cizgiCiz(int n) {
        System.out.println("-".repeat(n));
    }
    public static void main(String[] args) {
        System.out.println(cizgiCiz(5));
    }
}
```

```
VoidYaz.java:6: error: 'void' type not allowed here
        System.out.println(cizgiCiz(5));
                                    ^
1 error
```

Bu kez derleyici "`void` tipine burada izin verilmez" der. Kural tektir: `void` bir çağrı yalnızca kendi başına bir deyim olarak, sonuna noktalı virgül konarak yazılabilir. Bir ifadenin parçası olamaz.



Şekil 9.7: Değer döndüren ve döndürmeyen metot. `topla(5, 3)`'ün döndürdüğü 8 çağrının yerine gelir ve atamada kullanılır. `cizgiCiz(5)` yalnızca ekrana yazar; sonucu bir değişkene atanamaz.

`void` bir metotta `return` zorunlu değildir. Metot, gövdesinin kapanan küme parantezine ulaşınca kendiliğinden biter. Yine de `void` metotta değersiz bir `return;` deyimi yazılabilir. Bu deyim bir değer göndermez, yalnızca metodu o noktada bitirir. Erken dönüş kalıbı bu deyime dayanır.

Sık yapılan hata

Bir sonucu hem yazdırmak hem döndürmek gerektiğini sanmak ya da ikisini karıştırmak. `System.out.println(x)` değeri ekrana yazar ama çağırana bir şey göndermez. `return x;` ise değeri çağırana gönderir ama ekrana hiçbir şey yazmaz. Hesap yapan metotlar sonucu

döndürmeli, yazdırma kararını çağırana bırakmalıdır.

9.5.3 Erken dönüş

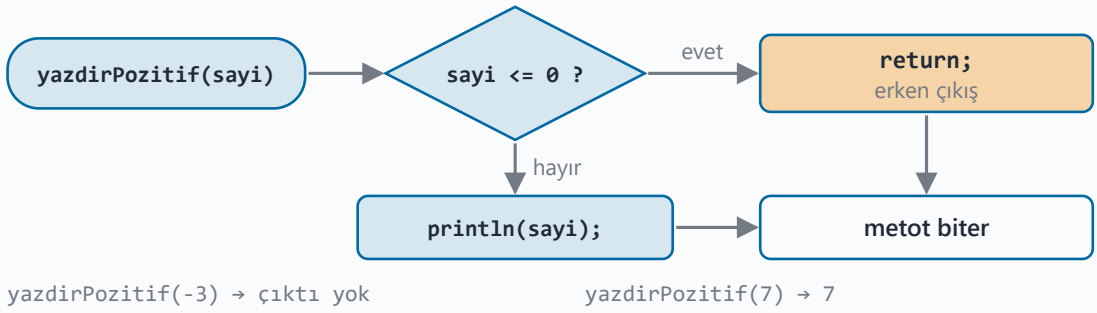
Bazı girdiler için metodun yapacak bir işi yoktur. Bu durum metodun başında yakalanır ve metottan hemen çıkılır. Böylece asıl iş, iç içe `if` blokları içine gömülmeden düz bir sırayla yazılır.

Tanım 9.6

Erken dönüş (*early return*), özel bir durumu metodun başında bir koşulla yakalayıp `return` ile metodu gövdenin sonuna gelmeden bitirmektir. `void` metotta değersiz `return;`, değer döndüren metotta ise o durum için uygun bir değerle `return` kullanılır.

```
static void yazdirPozitif(int sayi) {
    if (sayi <= 0) return;
    System.out.println(sayi);
}
```

`yazdirPozitif`, yalnızca pozitif sayıları yazar. Sayı 0 ya da negatifse `return;` metodu hemen bitirir ve `println`'e hiç gelinmez. Şekil 9.8 bu iki yolu akış diyagramıyla gösterir.



Şekil 9.8: `yazdirPozitif`'in akışı. `sayi <= 0` ise Evet dalı `return;` ile metottan erken çıkar; değilse sayı yazılır ve metot sona ulaşır. `yazdirPozitif(-3)` hiçbir şey yazmaz, `yazdirPozitif(7)` 7 yazar.

Aşağıdaki programda metot iki kez çağrılır: önce −3, sonra 7 ile. Satır numaraları: 1–4 metot, 6 ve 7 `main`'deki iki çağrıdır. Tablo 9.5 iki çağrıyı arka arkaya izler.

```
static void yazdirPozitif(int sayi) {
    if (sayi <= 0) return;
    System.out.println(sayi);
}
// main içinde:
yazdirPozitif(-3);
yazdirPozitif(7);
```

Adım	Satır	Çağrı yığını (üstteki önce)	Değişkenler (üstteki çerçeve)	Çıktı
1	6	`main()`	—	—
2	1	`yazdirPozitif(-3) ← main()`	sayi=−3	—
3	2	`yazdirPozitif(-3) ← main()`	sayi=−3	—
4	7	`main()`	—	—
5	1	`yazdirPozitif(7) ← main()`	sayi=7	—
6	2	`yazdirPozitif(7) ← main()`	sayi=7	—
7	3	`yazdirPozitif(7) ← main()`	sayi=7	7
8	4	`yazdirPozitif(7) ← main()`	sayi=7	7

Tablo 9.5: `yazdirPozitif(-3)` ve `yazdirPozitif(7)` çağrılarının izleme tablosu.

İlk çağrıda çerçeve açılır ve sayi = −3 olur. 2. satırda *−3 <= 0* doğrudur ve `return;` çalışır. Metot hemen biter; 3. satırdaki `println`'e hiç gelinmez ve çıktı boş kalır. `main` bir sonraki satırdan, yani ikinci çağrıdan devam eder. İkinci çağrı yepyeni bir çerçeve açar: sayi = 7. Önceki çağrının sayi değişkeni çoktan silinmiştir; iki çağrının hiçbir ortak değişkeni yoktur. Bu kez *7 <= 0* yanlıştır, erken dönüş olmaz ve 7 yazılır. Son adımda yürütme 4. satırdaki kapanan küme parantezine gelir; `void` metot burada kendiliğinden döner. Programın tam hâline bir de `yazdirPozitif(0)` çağrısı eklenince çıktı değişmez, çünkü 0 da erken dönüşe takılır:

```
public class YazdirPozitif {
    static void yazdirPozitif(int sayi) {
        if (sayi <= 0) return;
        System.out.println(sayi);
    }

    public static void main(String[] args) {
        yazdirPozitif(-3);
        yazdirPozitif(7);
        yazdirPozitif(0);
    }
}
```

```
7
```

Erken dönüş olmadan aynı metot `if (sayi > 0) { System.out.println(sayi); }` biçiminde yazılırdı. Tek satırlık bir işte iki yazım arasında fark azdır. Ama asıl iş on satır sürüyorsa ve üç farklı özel durum varsa, erken dönüşsüz yazım üç kat iç içe `if` doğurur. Erken dönüşle her özel durum metodun başında bir satırda elenir ve asıl iş girintisiz kalır.

Değer döndüren metotlarda da erken dönüş sık kullanılır. Bu durumda `return`'ün yanında o özel durum için doğru değer yazılır. Basamak sayısı metodundaki `if (kalan == 0) return 1;` ve asal sayı metodundaki `if (n < 2) return false;` satırları birer erken dönüştür.

9.5.4 return ve tip hataları

Derleyici, değer döndüren bir metodun her durumda gerçekten bir değer döndürdüğünü denetler. Aşağıdaki ilk örnekte metot yalnızca n pozitifken bir değer döndürür:

```
public class EksikReturn {
    static int yaz(int n) {
        if (n > 0) {
            return 1;
        }
    }
}
```

```
EksikReturn.java:6: error: missing return statement
    }
    ^
1 error
```

İleti “eksik `return` deyimi” der ve metodun kapanan küme parantezini gösterir. Derleyici şöyle düşünür: n pozitif değilse `if`'in içine girilmez ve yürütme metodun sonuna ulaşır. Bu yolda döndürülecek bir değer yoktur. `void` olmayan bir metotta *her yol* bir `return` ile bitmelidir. Çözüm, eksik yolu kapatmaktır: metodun sonuna `return 0;` eklemek ya da `if`'e bir `else` dalı yazıp orada da bir değer döndürmek.

Derleyici bu denetimi koşulun değerine bakmadan yapar. `if (n > 0) return 1; if (n <= 0) return 0;` yazılsa bile sonuç değişmez. İki koşul insan için bütün durumları kapsar, ama derleyici yalnızca iki ayrı `if` görür ve aynı hatayı verir. Derleyiciyi ikna etmenin yolu `if-else` kullanmak ya da metodun sonuna koşulsuz bir `return` koymaktır.

İkinci hata türü, döndürülen değerin dönüş tipine sığmamasıdır:

```
public class KayipDonusum {
    static int f() {
        return 2.5;
    }
}
```

```
KayipDonusum.java:3: error: incompatible types: possible lossy conversion from double to
int
        return 2.5;
               ^
1 error
```

İleti "uyumsuz tipler: `double`'dan `int`'e dönüşümde kayıp olabilir" der. Java, bilgi kaybına yol açabilecek **daraltma** (*narrowing conversion*) dönüşümlerini kendiliğinden yapmaz. Programcı kesirin atılmasını gerçekten istiyorsa bunu açık dönüşümle belirtir: `return (int) 2.5;` derlenir ve 2 döner. Kesirli sonuç isteniyorsa dönüş tipi `double` yapılır.

`return` ile dönüş tipinin uyuşmadığı iki durum daha vardır. `void` bir metot bir değer döndüremez; değer döndüren bir metotta da değersiz `return;` yazılamaz:

```
public class VoidReturn {
    static void g() {
        return 1;
    }
}
```

```
VoidReturn.java:3: error: incompatible types: unexpected return value
        return 1;
               ^
1 error
```

```
public class DonusDegerYok {
    static int h() {
        return;
    }
}
```

```
DonusDegerYok.java:3: error: incompatible types: missing return value
        return;
        ^
1 error
```

Durum	İleti	Çözüm
Bir yolda `return` yok	`missing return statement`	eksik dala ya da metodun sonuna `return` eklemek
`int` metotta `return 2.5;`	`incompatible types: possible lossy conversion from double to int`	`(int)` ile açık dönüşüm ya da dönüş tipini `double` yapmak
`void` metotta `return 1;`	`incompatible types: unexpected return value`	değeri silmek ya da dönüş tipini değiştirmek
`int` metotta `return;`	`incompatible types: missing return value`	`return`'e bir değer yazmak
`void` çağrıyı atamak	`incompatible types: void cannot be converted to int`	metodu değer döndürecek biçimde yeniden yazmak
`void` çağrıyı argüman yapmak	`'void' type not allowed here`	çağrıyı ayrı bir deyim olarak yazmak

Tablo 9.6: `return` ile ilgili derleme hataları (JDK 21 iletileri) ve çözümleri.

Örnek 9.1

Bir sayının işaretini −1, 0 ya da 1 olarak döndüren `isaret` metodu şöyle yazılmış: `static int isaret(int x) { if (x > 0) return 1; else if (x < 0) return -1; }`. Derlenir mi?

Derlenmez. x = 0 için iki koşul da yanlıştır ve yürütme metodun sonuna, `return`'süz olarak ulaşır. Derleyici `missing return statement` hatası verir. Zincirin sonuna `else return 0;` ya da metodun sonuna `return 0;` eklenince her yol bir değerle biter: isaret(5) = 1, isaret(−3) = −1, isaret(0) = 0.

9.6 Parametre aktarımı

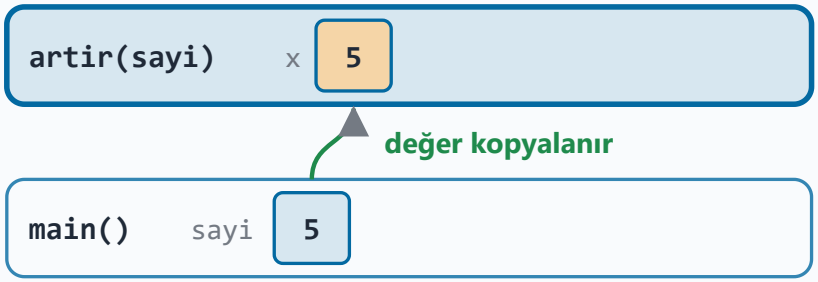
Bir çağrıda argümanın değeri parametreye geçer. Peki metot parametresini değiştirirse çağıranın değişkeni de değişir mi? Bu sorunun cevabı Java'da kesindir ve birçok yeni programcıyı şaşırtır.

9.6.1 Değer ile aktarım

Tanım 9.7

Değer ile aktarım (*pass by value*), bir çağrıda argümanın değerinin kopyalanıp parametreye yazılmasıdır. Parametre, argümandan ayrı bir bellek kutusudur. Metot parametresine ne yaparsa yapsın, çağıranın değişkeni etkilenmez. Java argümanları her zaman değer ile aktarır.

`int`, `double`, `boolean`, `char` gibi ilkel tiplerde aktarılan şey sayının ya da karakterin kendisidir. Çağıranın değişkeni ile metodun parametresi iki ayrı kutudur; çağrı anında ikincisine birincinin içindekinin bir kopyası yazılır. Kopyalamadan sonra iki kutu birbirinden bağımsızdır.



Şekil 9.9: `artir(sayi)` çağrısında `main`'deki sayi kutusunun değeri (5) `artir`'ın çerçevesindeki x kutusuna kopyalanır. İki ayrı bellek kutusu vardır.

Nesnelerde, örneğin dizilerde, durum ilk bakışta farklı görünür. Bir dizi değişkeninin içinde dizinin kendisi değil, dizinin bellekteki yerini gösteren bir **referans** bulunur. Diziyi bir metoda gönderirken kopyalanan bu referanstır. Kopya referans da aynı diziyi gösterdiği için metot dizinin öğelerini değiştirebilir. Ama bu da değer ile aktarımdır: kopyalanan değer bir referanstır. Metot parametresine yeni bir dizi atarsa çağıranın değişkeni yine eski diziyi gösterir.

9.6.2 artir(sayi) adım adım

İlk örnek, bir sayıyı artırmaya çalışan bir `void` metottur:

```
static void artir(int x) {
    x++;
}
// main içinde:
int sayi = 5;
artir(sayi);
System.out.println(sayi);   // 5
```

Satır numaraları: 1–3 `artir`, 5–7 `main`'deki satırlardır. Program çalıştırılınca 6 değil 5 yazar:

```
5
```

Adım	Satır	Çağrı yığını (üstteki önce)	Değişkenler (üstteki çerçeve)	Çıktı
1	5	`main()`	sayi=5	—
2	6	`main()`	sayi=5	—
3	1	`artir(5) ← main()`	x=5	—
4	2	`artir(5) ← main()`	x=6	—
5	3	`artir(5) ← main()`	x=6	—
6	7	`main()`	sayi=5	5

Tablo 9.7: `artir(sayi)` çağrısının izleme tablosu. x değişir, sayi değişmez.

1. adımda `main`'in çerçevesinde sayi = 5 olur. 2. adımda `artir(sayi)` çağrılır. Gönderilen şey sayi değişkeni değil, onun değeri olan 5'tir. 3. adımda yeni çerçevede ayrı bir kutu açılır: x = 5. Bu bir kopyadır. 4. adımda `x++` çalışır ve x = 6 olur. Değişen yalnızca `artir`'ın kutusudur; `main`'deki sayi hâlâ 5'tir. 5. adımda metot biter, çerçeve ve içindeki x (6) silinir. `artir` bir `void` metot olduğu için 6 değeri hiçbir yere gönderilmez ve tamamen kaybolur. 6. adımda sayi'nin değeri olan 5 yazılır.



Şekil 9.10: (a) Çağrı anı: `artir`'ın çerçevesinde x = 5, sayi'nin kopyası. (b) `x++` sonrası: x = 6, ama `main`'deki sayi hâlâ 5.

9.6.3 swap tuzağı

İki değişkenin değerlerini yer değiştirmek, yani takas etmek, programlamada sık yapılan bir iştir. Takas bir geçici değişkenle üç adımda yapılır. Bu üç adımı bir metoda koymak akla yatkın görünür:

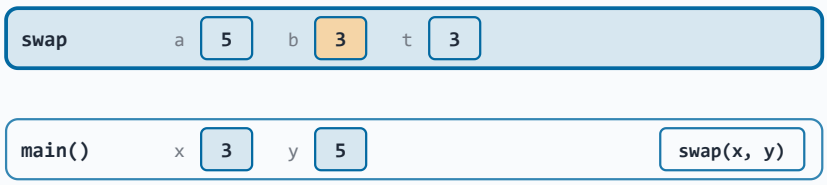
```
static void swap(int a, int b) {
    int t = a;
    a = b;
    b = t;
}
// main içinde:
int x = 3, y = 5;
swap(x, y);
System.out.println(x + " " + y);   // 3 5
```

Yorum satırında da görüldüğü gibi program `5 3` değil `3 5` yazar. Metodun içindeki takas doğrudur; ama takas edilen, x ve y'nin kendisi değil kopyalarıdır. Tablo 9.8 durumu adım adım gösterir. Satır numaraları: 1–5 `swap`, 7–9 `main`'deki satırlardır.

Adım	Satır	Çağrı yığını (üstteki önce)	Değişkenler (üstteki çerçeve)	Çıktı
1	7	`main()`	x=3, y=5	—
2	8	`main()`	x=3, y=5	—
3	1	`swap(3, 5) ← main()`	a=3, b=5	—
4	2	`swap(3, 5) ← main()`	a=3, b=5, t=3	—
5	3	`swap(3, 5) ← main()`	a=5, b=5, t=3	—
6	4	`swap(3, 5) ← main()`	a=5, b=3, t=3	—
7	9	`main()`	x=3, y=5	3 5

Tablo 9.8: `swap(x, y)` çağrısının izleme tablosu. a ve b takas edilir, x ve y değişmez.

Çağrı anında a = 3 ve b = 5 olur; bunlar x ve y'nin kopyalarıdır ve onlarla hiçbir bağları yoktur. `int t = a;` geçici kutuya 3'ü koyar. `a = b;` ile a = 5 olur; a'nın eski değeri artık yalnızca t'de saklıdır. `b = t;` ile b = 3 olur. Bu noktada `swap` 'in kendi kutuları gerçekten yer değiştirmiştir: a = 5, b = 3. Ancak metot bitince a, b ve t'nin hepsi silinir. `main` 'deki x ve y'ye hiç dokunulmamıştır: x = 3, y = 5.



Şekil 9.11: `swap` 'in son satırından sonra: `swap` 'in çerçevesinde a = 5, b = 3, t = 3; alttaki `main` çerçevesinde ise x = 3, y = 5. Takas yalnızca kopyalarda olmuştur.

Bunu metodun içine birkaç yazdırma satırı ekleyerek de görmek mümkündür. Aşağıdaki sürüm, metodun başında ve sonunda kendi değişkenlerini, dönüşten sonra da `main` 'in değişkenlerini yazar:

```
public class SwapIz {
    static void swap(int a, int b) {
        System.out.println("swap başı: a=" + a + " b=" + b);
        int t = a;
        a = b;
        b = t;
        System.out.println("swap sonu: a=" + a + " b=" + b + " t=" + t);
    }

    public static void main(String[] args) {
        int x = 3, y = 5;
        swap(x, y);
        System.out.println("main: x=" + x + " y=" + y);
    }
}
```

```
swap başı: a=3 b=5
swap sonu: a=5 b=3 t=3
main: x=3 y=5
```

Çıktının ikinci satırı takasın metodun içinde başarılı olduğunu, üçüncü satırı ise bunun dışarıya hiçbir etkisi olmadığını gösterir. Java'da iki ilkel değişkeni takas eden bir metot yazılamaz. Takas ya çağıran kodun içinde, geçici bir değişkenle yapılır ya da değerler bir dizi içinde tutulur ve dizinin öğeleri takas edilir.

Sık yapılan hata

Parametreyle argümanın adı aynı olunca metodun çağıranın değişkenini değiştireceğini sanmak. `static void artir(int sayi) { sayi++; }` yazılsa ve `main`'de `artir(sayi);` çağrılsa da sonuç değişmez. Parametrenin adı `sayi` olsa bile o, `artir`'ın çerçevesindeki ayrı bir kutudur. Adların aynı olması iki değişkeni birbirine bağlamaz.

9.6.4 Sonucu çağırana iletmek

Bir metot, çağıranın bir değişkenini değiştirmek istiyorsa yeni değeri döndürür; çağıran da bu değeri kendi değişkenine atar. Düzeltilmiş sürüm şudur:

```
public class ArtirDondur {
    static int artir(int x) {
        return x + 1;
    }

    public static void main(String[] args) {
        int sayi = 5;
        sayi = artir(sayi);
        System.out.println(sayi);   // 6
    }
}
```

```
6
```

Burada da değer ile aktarım geçerlidir: x, sayi'nin kopyasıdır ve `main`'deki sayi'yi değiştiren metot değildir. Değişikliği `main`'in kendisi, `sayi = artir(sayi);` atamasıyla yapar. Bu yazım ne olduğunu açıkça gösterir: okuyan, sayi'nin bu satırda değiştiğini atamadan anlar. Bir metodun çağıranın değişkenlerini gizlice değiştirememesi bir kısıt gibi görünse de programları anlaşılır kılan bir güvencedir. `hesapla(x)` gibi bir çağrının x'i bozmayacağı, metodun gövdesine bakmadan bilinir.

9.7 Aşırı yükleme

Karenin alanı tek bir kenarla, dikdörtgenin alanı iki kenarla hesaplanır. İki işe `kareAlani` ve `dikdortgenAlani` gibi ayrı adlar verilebilir. Ama ikisi de aslında “alan” hesaplar. Java, aynı işi farklı girdilerle yapan metotlara aynı adı vermeye izin verir.

9.7.1 Aynı ad, farklı parametre listesi

Tanım 9.8

Aşırı yükleme (*method overloading*), aynı sınıfta aynı adlı birden çok metot tanımlamaktır. Bu metotların parametre listeleri farklı olmalıdır: parametre sayısı, parametre tipleri ya da tiplerin sırası. Kısacası imzaları farklı olmalıdır. Derleyici bir çağrıyı, argümanların sayısına ve tiplerine bakarak bu metotlardan birine bağlar.

```
static int alan(int kenar) {
    return kenar * kenar;
}
static int alan(int en, int boy) {
    return en * boy;
}
// HATA: yalnız dönüş tipi farklı
static double alan(int kenar) { … }
```

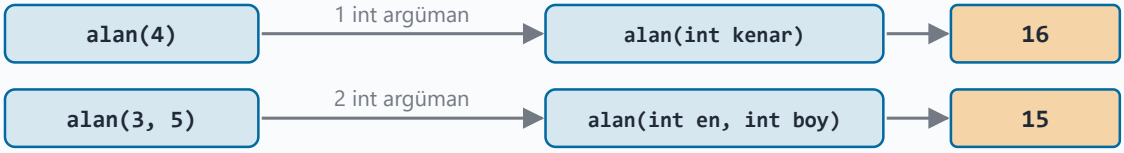
İlk iki metot geçerli bir aşırı yüklemedir. `alan(int kenar)` bir parametre, `alan(int en, int boy)` iki parametre alır; imzaları `alan(int)` ve `alan(int, int)` 'dir. Üçüncü metot ise hatalıdır: parametre listesi birinciyle aynıdır, yalnızca dönüş tipi farklıdır.

```
public class Alan {
    static int alan(int kenar) {
        return kenar * kenar;
    }

    static int alan(int en, int boy) {
        return en * boy;
    }

    public static void main(String[] args) {
        System.out.println(alan(4));
        System.out.println(alan(3, 5));
    }
}
```

```
16
15
```



Derleyici argümanların sayı ve tipine bakar; dönüş tipine bakmaz.

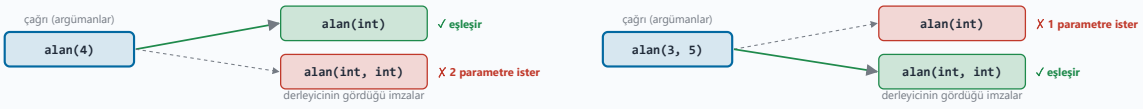
Şekil 9.12: `alan(4)` 'te tek `int` argüman vardır ve tek parametreli sürüm seçilir (16). `alan(3, 5)` 'te iki `int` argüman vardır ve iki parametreli sürüm seçilir (15). Derleyici argümanların sayısına ve tipine bakar; dönüş tipine bakmaz.

Aşırı yüklemenin en tanıdık örneği `System.out.println`'dir. `println(5)`, `println(2.5)`, `println("Merhaba")` ve `println(true)` çağrılarının her biri ayrı bir metoda gider: `println(int)`, `println(double)`, `println(String)`, `println(boolean)`. Programcı her tip için ayrı bir ad ezberlemek zorunda kalmaz. `Math.abs` ve `Math.max` da `int`, `long`, `float` ve `double` için aşırı yüklenmiştir.

9.7.2 Derleyici hangi metodu seçer?

Aşırı yüklenmiş metotlar arasında seçim program çalışmadan önce, derleme sırasında yapılır. Derleyici çağrıdaki her argümanın tipini bilir ve bu tiplere uyan imzayı arar. Aşağıdaki program bu kararı iki çağrı için gösterir:

```
static int alan(int kenar) {
    return kenar * kenar;
}
static int alan(int en, int boy) {
    return en * boy;
}
System.out.println(alan(4));      // main
System.out.println(alan(3, 5));   // main
```



Şekil 9.13: Derleyicinin seçimi. (a) `alan(4)` : `alan(int)` eşleşir; `alan(int, int)` iki parametre ister. (b) `alan(3, 5)` : `alan(int)` bir parametre ister; `alan(int, int)` eşleşir.

`alan(4)` çağrısında tek bir `int` argüman vardır. `alan(int)` sayı ve tip bakımından tam uyar; `alan(int, int)` iki argüman istediği için elenir. Çalışma sırasında `alan(int)` çağrılır, kenar = 4 olur ve *4 × 4 = 16* döner. `alan(3, 5)` çağrısında iki `int` argüman vardır ve bu kez yalnızca `alan(int, int)` uyar: en = 3, boy = 5 ve *3 × 5 = 15* döner.

Argüman tipi hiçbir parametreye tam uymadığında derleyici **genişletme** (*widening conversion*) dönüşümlerini dener. Bir tip, bilgi kaybetmeden daha geniş bir tipe dönüştürülebilir: `char` → `int` → `long` → `float` → `double`. Derleyici önce tam eşleşen bir imza arar. Bulamazsa argümanı genişleterek uyan imzalara bakar ve en yakın genişlemeyi gerektireni seçer. Daraltma ise hiçbir zaman kendiliğinden yapılmaz. Aşağıdaki programda `alan`'a bir de `double` sürüm eklenmiştir:

```
public class AlanDouble {
    static int alan(int kenar) {
        return kenar * kenar;
    }

    static int alan(int en, int boy) {
        return en * boy;
    }

    static double alan(double kenar) {
        return kenar * kenar;
    }

    public static void main(String[] args) {
        System.out.println(alan(4));      // alan(int)
        System.out.println(alan(2.5));    // alan(double)
        System.out.println(alan(3, 5));   // alan(int, int)
        System.out.println(alan(4L));     // long → double: alan(double)
        System.out.println(alan('B'));    // char → int: alan(int)
    }
}
```

```
16
6.25
15
16.0
4356
```

Çağrı	Argüman tipleri	Seçilen metot	Neden?	Sonuç
`alan(4)`	`int`	`alan(int)`	tam eşleşme	`16`
`alan(2.5)`	`double`	`alan(double)`	tam eşleşme	`6.25`
`alan(3, 5)`	`int, int`	`alan(int, int)`	tam eşleşme	`15`
`alan(4L)`	`long`	`alan(double)`	`long`, `int`'e daraltılamaz; `double`'a genişler	`16.0`
`alan('B')`	`char`	`alan(int)`	`char` → `int` en yakın genişleme; 'B' = 66	`4356`

Tablo 9.9: `AlanDouble` programındaki çağrılar için derleyicinin seçimi.

Son iki satır dikkat ister. `4L`, `long` tipinde bir sabittir. `alan(int)`'e verilemez, çünkü `long`'dan `int`'e geçiş bir daraltmadır. `long`, `double`'a genişleyebildiği için `alan(double)` seçilir ve sonuç `16.0` olarak, ondalık noktasıyla yazılır. `'B'` ise bir `char`'dır. Hem `int`'e hem `double`'a genişleyebilir; derleyici daha yakın olan `int`'i seçer. 'B' karakterinin kodu 66'dır ve *66 × 66 = 4356* yazılır. Seçim tamamen derleme sırasında yapılır; program çalışırken argümanın değerine bakılmaz.

9.7.3 Hatalı aşırı yükleme

Yukarıdaki üçüncü metot, `static double alan(int kenar)`, birinci metotla aynı imzayı taşır: `alan(int)`. Yalnızca dönüş tipi farklıdır. Derleyici bunu kabul etmez:

```
public class AlanTekrar {
    static int alan(int kenar) {
        return kenar * kenar;
    }
    static double alan(int kenar) {
        return kenar * kenar;
    }
}
```

```
AlanTekrar.java:5: error: method alan(int) is already defined in class AlanTekrar
    static double alan(int kenar) {
                  ^
1 error
```

İleti "`alan(int)` metodu bu sınıfta zaten tanımlı" der. Dönüş tipinin imzaya dahil olmamasının nedeni burada açıkça görülür. `alan(4);` gibi sonucu kullanılmayan bir çağrıyla karşılaşan derleyici, hangi sürümün kastedildiğini bilemezdi. `System.out.println(alan(4));` çağrısında da durum aynıdır, çünkü `println` hem `int` hem `double` kabul eder. Çağrının biçimi yalnızca ad ve argümanlardan oluşur; seçim de yalnızca bunlara dayanabilir.

Uyan hiçbir imza yoksa derleme yine başarısız olur. `alan(2.5)` çağrısı, yalnızca `alan(int)` ve `alan(int, int)` tanımlıyken derlenmez:

```
public class AlanHata {
    static int alan(int kenar) {
        return kenar * kenar;
    }
    static int alan(int en, int boy) {
        return en * boy;
    }
    public static void main(String[] args) {
        System.out.println(alan(2.5));
    }
}
```

```
AlanHata.java:9: error: incompatible types: possible lossy conversion from double to int
        System.out.println(alan(2.5));
                           ^
Note: Some messages have been simplified; recompile with -Xdiags:verbose to get full
output
1 error
```

JDK 21 bu durumda kısaltılmış bir ileti verir. Tek parametreli tek aday `alan(int)` olduğu için, sorunu doğrudan `double`'dan `int`'e daraltma olarak bildirir. İletinin altındaki not, ayrıntılı çıktının `-Xdiags:verbose` seçeneğiyle alınabileceğini söyler. Bu seçenekle derleyici "`alan(double)` için uygun metot bulunamadı" (`no suitable method found for alan(double)`) der ve iki adayın neden uymadığını ayrı ayrı sıralar: birinde kayıplı dönüşüm, ötekinde argüman sayısı farkı.

Üçüncü bir hata türü de belirsizliktir. İki ayrı imza da genişletmeyle eşit derecede uyuyorsa derleyici bir seçim yapamaz:

```
public class BelirsizCagri {
    static void f(int a, double b) { }
    static void f(double a, int b) { }
    public static void main(String[] args) {
        f(1, 2);
    }
}
```

```
BelirsizCagri.java:5: error: reference to f is ambiguous
        f(1, 2);
        ^
  both method f(int,double) in BelirsizCagri and method f(double,int) in BelirsizCagri
match
1 error
```

`f(1, 2)` çağrısında iki argüman da `int`'tir. `f(int, double)` için ikinci argümanın, `f(double, int)` için birinci argümanın genişletilmesi gerekir. İki seçenek de birer genişletme ister ve hiçbiri ötekinden daha iyi değildir. Derleyici "`f`'ye başvuru belirsiz" der ve iki adayı da gösterir. Argümanlardan biri açıkça `double` yazılırsa belirsizlik kalkar:

```
public class Belirsiz {
    static void f(int a, double b) {
        System.out.println("f(int, double)");
    }

    static void f(double a, int b) {
        System.out.println("f(double, int)");
    }

    public static void main(String[] args) {
        f(1, 2.0);
        f(1.0, 2);
    }
}
```

```
f(int, double)
f(double, int)
```

9.8 Değişken sayıda argüman

İki sayıyı toplayan bir metot, üç sayıyı toplayan bir başkası, dört sayıyı toplayan bir üçüncüsü... Aşırı yüklemeyle bu yol sonsuza kadar uzar. Java, bir metodun istenen sayıda argüman almasına izin veren bir sözdizimi sunar.

9.8.1 int... sözdizimi ve kuralları

Tanım 9.9

Değişken sayıda argüman (*varargs, variable-length arguments*), parametre tipinden sonra üç nokta yazılarak bildirilen ve sıfır ya da daha fazla argüman kabul eden parametredir: `int... s`. Derleyici çağrıdaki değerleri bir diziye paketler; metodun içinde `s` bir `int[]` dizisidir.

```
static int topla(int... s) {
    int t = 0;
    for (int v : s) t += v;
    return t;
}
topla(1, 2, 3)   // 6
topla()          // 0
topla(new int[]{4, 5})   // 9
```

Metodun içinde `s` sıradan bir dizi gibi kullanılır. `s.length` kaç argüman geldiğini verir. `for (int v : s)` biçimindeki **gelişmiş for** döngüsü dizinin öğelerini sırayla v'ye koyar; her turda bir öğe işlenir. Tam program ve çıktısı:

```
public class Varargs {
    static int topla(int... s) {
        int t = 0;
        for (int v : s) t += v;
        return t;
    }

    public static void main(String[] args) {
        System.out.println(topla(1, 2, 3));
        System.out.println(topla());
        System.out.println(topla(new int[]{4, 5}));
        System.out.println(topla(10));
    }
}
```

```
6
0
9
10
```

`topla(1, 2, 3)` için derleyici `new int[]{1, 2, 3}` dizisini kendisi oluşturur ve metoda bu diziyi geçirir. `topla()` çağrısında hiç argüman yoktur; derleyici bu kez boş bir dizi geçirir ve sonuç 0'dır. Üçüncü çağrıda dizi doğrudan verilmiştir: varargs parametresi bir dizi de kabul eder, sonuç *4 + 5 = 9*'dur. Tek argümanlı `topla(10)` da geçerlidir ve 10 döner.

Varargs parametresinin iki kuralı vardır. Parametre listesinin en sonunda bulunmalıdır, çünkü derleyici çağrıdaki değerlerden hangilerinin diziye gireceğini ancak böyle anlayabilir. Bir metotta da yalnızca bir tane bulunabilir. İki kural da aynı derleyici iletisiyle denetlenir:

```
public class VarargsSon {
    static int topla(int... s, int x) {
        return x;
    }
}
```

```
VarargsSon.java:2: error: varargs parameter must be the last parameter
    static int topla(int... s, int x) {
                             ^
1 error
```

İki varargs parametresi yazıldığında da (`int... s, int... t`) aynı ileti çıkar, çünkü birincisi en sonda değildir. Varargs'tan önce sıradan parametreler bulunabilir. Aşağıdaki metot önce bir başlık, ardından istenen sayıda not alır:

```
public class Etiket {
    static void yaz(String baslik, int... notlar) {
        System.out.print(baslik + ":");
        for (int n : notlar) System.out.print(" " + n);
        System.out.println(" (" + notlar.length + " not)");
    }

    public static void main(String[] args) {
        yaz("Ali", 70, 85, 90);
        yaz("Ayşe", 100);
        yaz("Can");
        System.out.printf("%d + %d = %d%n", 2, 3, 2 + 3);
    }
}
```

```
Ali: 70 85 90 (3 not)
Ayşe: 100 (1 not)
Can: (0 not)
2 + 3 = 5
```

“Can” için hiç not verilmemiştir; dizi boştur, döngü dönmez ve `notlar.length` 0'dır. Programın son satırındaki `printf`, Java kütüphanesinde varargs kullanan bir metottur: imzası `printf(String, Object...)`'tır. Biçim dizgisinden sonra istenen sayıda değer alabilmesi bundandır.

9.8.2 topla(1, 2, 3) adım adım

Aşağıdaki `main`'de `topla` iki kez çağrılır:

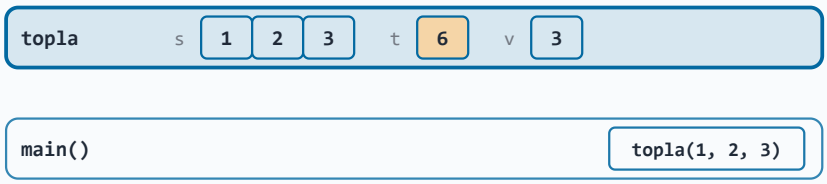
```
static int topla(int... s) {
    int t = 0;
    for (int v : s) t += v;
    return t;
}
// main içinde:
System.out.println(topla(1, 2, 3));
System.out.println(topla());
```

İlk çağrının izlemesi, metoda her turda değişkenleri yazdıran satırlar eklenmiş bir sürüm çalıştırılarak elde edildi. Tablo 9.10 bu çıktıdan oluşturulmuştur.

Tur	v	İşlem	t
giriş	—	`s = {1, 2, 3}`, `s.length = 3`	0
1. tur	1	0 + 1 = 1	1
2. tur	2	1 + 2 = 3	3
3. tur	3	3 + 3 = 6	6

Tablo 9.10: `topla(1, 2, 3)` çağrısında gelişmiş for döngüsünün turları.

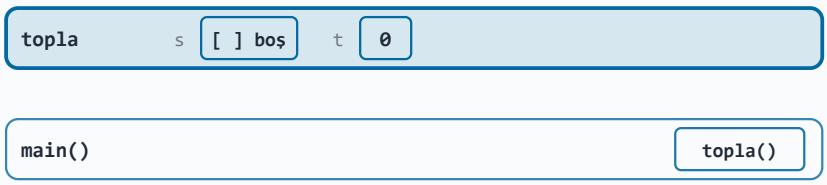
Çağrı anında derleyicinin hazırladığı dizi s parametresine geçer: s = {1, 2, 3} ve s.length = 3. `int t = 0;` toplam için yerel bir kutu açar. Gelişmiş for, 1. turda v = 1 yapar ve *t = 0 + 1 = 1* olur. 2. turda v = 2 ve *t = 1 + 2 = 3*, 3. turda v = 3 ve *t = 3 + 3 = 6* olur. Dizide başka öğe kalmadığı için döngü biter. `return t;` 6'yı döndürür. v değişkeni döngünün içinde tanımlandığı için döngüyle birlikte kapsam dışına çıkmıştır. `topla`'nın çerçevesi (s ve t) silinir ve `main` 6'yı yazar.



Şekil 9.14: 3. turun sonu: `topla`'nın çerçevesinde s dizisi {1, 2, 3}, t = 6 ve v = 3.

9.8.3 Boş çağrı

İkinci çağrı `topla()`'dır ve hiç argüman içermez. Derleyici `null` değil, boş bir dizi (`new int[0]`) geçirir; s.length = 0'dır. `t = 0` atanır. Dizide hiç öğe olmadığı için gelişmiş for'un gövdesi bir kez bile çalışmaz ve t 0 kalır. Metot 0 döndürür, program 0 yazar.



Şekil 9.15: `topla()` çağrısında s boş bir dizidir. Döngü gövdesi hiç çalışmaz, t = 0 kalır.

Boş çağrının bir hata olmadığına ve sonucun 0 çıktığına dikkat edilmelidir. 0 toplamanın **etkisiz elemanıdır**: bir sayıya 0 eklemek onu değiştirmez. Hiç sayı yokken toplamın 0 olması bu yüzden matematikte de doğal kabul edilir. Aynı mantıkla çarpım hesaplayan bir varargs metodunda biriktirici 1 ile başlar ve boş çağrı 1 döndürür. Buna karşılık "en büyük öğe" ya da "ortalama" gibi hesapların boş girdi için anlamlı bir cevabı yoktur. Böyle metotlarda boş dizi başta ayrıca ele alınmalıdır. Örneğin tam sayı ortalamasında *0 / 0* bölmesi program çalışırken `ArithmeticException` hatasına yol açar.

9.9 Kapsam

Bir değişken bildirildiği her yerden kullanılamaz. Java her değişkene, programın belirli bir bölgesinde görünme hakkı verir. Bu bölgenin sınırlarını küme parantezleri çizer.

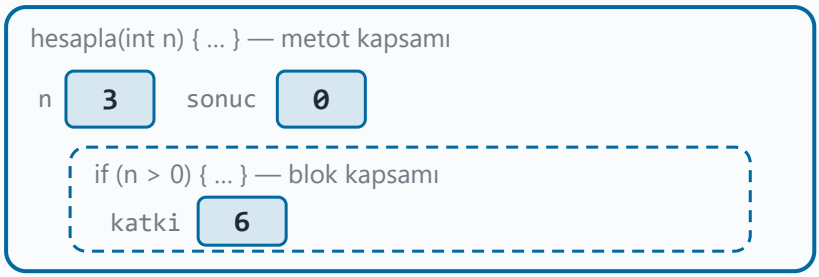
9.9.1 Yerel değişken ve kapsam

Tanım 9.10

Kapsam (*scope*), bir değişkenin adıyla kullanılabildiği program bölgesidir. Bir metodun içinde bildirilen değişkenlere ve metodun parametrelerine **yerel değişken** (*local variable*) denir. Yerel değişkenin kapsamı, bildirildiği satırdan başlar ve bildirildiği bloğun kapanan küme parantezinde biter. Parametrelerin kapsamı metodun bütün gövdesidir.

Kapsamın dört kuralı vardır:

- Parametreler ve metot içinde bildirilen değişkenler yereldir.
- Bir blokta, yani bir çift küme parantezi `{ }` arasında bildirilen değişken yalnızca o blokta ve onun içindeki bloklarda görünür.
- Aynı kapsamda aynı adlı iki yerel değişken bildirilemez.
- Bir metot, başka bir metodun yerel değişkenine erişemez.



Şekil 9.16: İç içe kapsamlar. Dıştaki kutu `hesapla`'nın metot kapsamıdır: n ve sonuc burada yaşar. İçteki kesikli kutu `if` bloğunun kapsamıdır: katki yalnızca burada vardır.

Son kural, çağrı yığınıyla birlikte düşünülünce kendiliğinden anlaşılır. Bir metodun yerel değişkenleri o metodun çerçevesinde durur. Başka bir metot ya o çerçeve henüz yokken ya da başka bir çerçevenin içinde çalışırken o değişkene ulaşamaz. Derleyici bu erişimi baştan reddeder:

```
public class BaskaMetot {
    static void birinci() {
        int gizli = 42;
    }
    static void ikinci() {
        System.out.println(gizli);
    }
}
```

```
BaskaMetot.java:6: error: cannot find symbol
        System.out.println(gizli);
                           ^
  symbol:   variable gizli
  location: class BaskaMetot
1 error
```

İleti "sembol bulunamadı" der ve aranan sembolün `gizli` adlı bir değişken olduğunu belirtir. `ikinci` metodunun içinde bu adla bilinen hiçbir şey yoktur. Değeri `birinci`'den `ikinci`'ye taşımanın yolu, `birinci`'nin değeri döndürmesi ve çağıranın bu değeri `ikinci`'ye argüman olarak vermesidir.

9.9.2 hesapla(3) adım adım

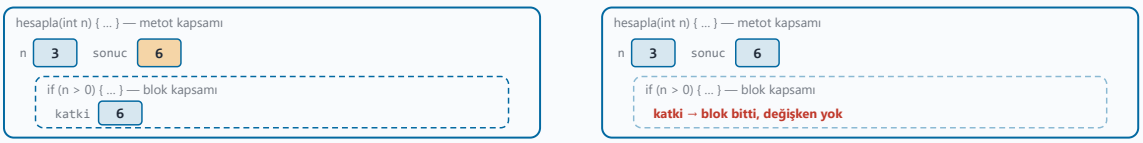
`hesapla` metodu iki iç içe kapsam içerir: metodun kendi kapsamı ve `if` bloğunun kapsamı.

```
static int hesapla(int n) {
    int sonuc = 0;
    if (n > 0) {
        int katki = n * 2;
        sonuc += katki;
    }
    return sonuc;
}
System.out.println(hesapla(3));  // main
```

Adım	Satır	Çağrı yığını (üstteki önce)	Değişkenler (üstteki çerçeve)	Çıktı
1	9	`main()`	—	—
2	1	`hesapla(3) ← main()`	n=3	—
3	2	`hesapla(3) ← main()`	n=3, sonuc=0	—
4	3	`hesapla(3) ← main()`	n=3, sonuc=0	—
5	4	`hesapla(3) ← main()`	n=3, sonuc=0, katki=6	—
6	5	`hesapla(3) ← main()`	n=3, sonuc=6, katki=6	—
7	6	`hesapla(3) ← main()`	n=3, sonuc=6	—
8	7	`hesapla(3) ← main()`	n=3, sonuc=6	—
9	9	`main()`	—	6

Tablo 9.11: `hesapla(3)` çağrısının izleme tablosu. katki yalnızca 5. ve 6. adımlarda vardır.

`main`, `hesapla(3)`'ü çağırır ve metot kapsamı açılır: n = 3. 2. satırda `sonuc` bildirilir. Metodun en dış bloğunda bildirildiği için metodun sonuna kadar görünür. 3. satırda *3 > 0* doğrudur ve `if` bloğuna girilir; burada yeni bir iç kapsam açılır. 4. satırda `katki` bildirilir ve *3 × 2 = 6* değerini alır. Bu değişken yalnızca `if` bloğunun içinde vardır. 5. satırda iç bloktan dış kapsamdaki `sonuc`'a erişilir: *sonuc = 0 + 6 = 6*. İç blok dıştaki değişkenleri görür; tersi doğru değildir. 6. satırda blok kapanır ve `katki` yok olur. 7. satırda `return sonuc;` 6'yı döndürür ve metot kapsamı da kapanır. Dönen 6, `println`'e argüman olur ve yazılır.



Şekil 9.17: (a) 5. satırdan sonra: iç blokta katki = 6, dış kapsamda sonuc = 6. (b) Blok kapandıktan sonra: katki artık yok; n ve sonuc duruyor.

Metodu negatif bir sayıyla çağırmak `if` bloğunu hiç açmaz: `hesapla(-2)` için katki hiç oluşmaz ve 0 döner.

```
public class Hesapla {
    static int hesapla(int n) {
        int sonuc = 0;
        if (n > 0) {
            int katki = n * 2;
            sonuc += katki;
        }
        return sonuc;
    }

    public static void main(String[] args) {
        System.out.println(hesapla(3));
        System.out.println(hesapla(-2));
    }
}
```

```
6
0
```

Blok kapandıktan sonra `katki`'yi kullanmaya çalışmak derleme hatasıdır. Aşağıdaki sürüm, `return` satırında katki'yi de toplamaya çalışıyor:

```
public class KatkiDisarida {
    static int hesapla(int n) {
        int sonuc = 0;
        if (n > 0) {
            int katki = n * 2;
            sonuc += katki;
        }
        return sonuc + katki;
    }
}
```

```
KatkiDisarida.java:8: error: cannot find symbol
        return sonuc + katki;
                       ^
  symbol:   variable katki
  location: class KatkiDisarida
1 error
```

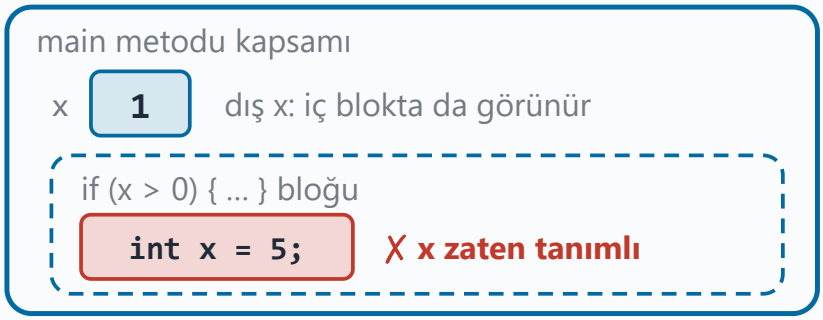
Derleyici 8. satırdaki `katki`'yi tanımaz. Bu satır `if` bloğunun dışındadır ve orada bu adla bir değişken yoktur. Derleyicinin bakış açısından durum, değişkenin hiç bildirilmemiş olmasından farksızdır. Değişkenin bloktan sonra da gerekmesi durumunda bildirimi bloktan önceye, dış kapsama taşınır.

9.9.3 Aynı adlı yerel değişkenler

İç blok, dış kapsamdaki değişkenleri gördüğüne göre iç blokta aynı adla yeni bir değişken bildirmek ne olur? Java'da bu bir hatadır:

```
public class AyniAd {
    public static void main(String[] args) {
        int x = 1;
        if (x > 0) {
            int x = 5;   // hata
        }
    }
}
```

```
AyniAd.java:5: error: variable x is already defined in method main(String[])
            int x = 5;   // hata
                ^
1 error
```



Şekil 9.18: `main`'in kapsamındaki x = 1, iç `if` bloğunda da görünür. İç blokta `int x = 5;` bildirmek hatadır: x zaten tanımlıdır.

İleti "x değişkeni `main(String[])` metodunda zaten tanımlı" der. Bazı dillerde iç bloktaki yeni değişken dıştakini geçici olarak gizler. Java bunu yerel değişkenler arasında bilerek yasaklar: iki x aynı anda görünseydi okuyan, bir satırdaki x'in hangisi olduğunu karıştırabilirdi.

Parametreler de yerel değişkendir ve metodun en dış kapsamında bulunur. Bu yüzden metot gövdesinde parametreyle aynı adlı bir değişken bildirilemez:

```
public class ParametreAyniAd {
    static int topla(int a, int b) {
        int a = 3;
        return a + b;
    }
}
```

```
ParametreAyniAd.java:3: error: variable a is already defined in method topla(int,int)
        int a = 3;
            ^
1 error
```

İletide metodun imzası parametre tipleriyle birlikte görünür: `topla(int,int)`. Parametreye yeni bir değer *atamak* (`a = 3;`) ise geçerlidir; hata yalnızca tipin yeniden yazılmasından, yani yeniden bildirimden doğar.

Aynı ad, kapsamı bitmiş bir değişkenden sonra ise yeniden kullanılabilir. Blok kapanınca içindeki değişken silinir ve ad serbest kalır:

```
public class BlokSonrasi {
    public static void main(String[] args) {
        for (int i = 0; i < 3; i++) {
            System.out.print(i + " ");
        }
        int i = 7;                 // önceki i'nin kapsamı bitti
        System.out.println(i);
        {
            int k = 1;
            System.out.println(k);
        }
        {
            int k = 2;             // ayrı blok, ayrı k
            System.out.println(k);
        }
    }
}
```

```
0 1 2 7
1
2
```

`for` başlığında bildirilen i'nin kapsamı döngüyle sınırlıdır. Döngü bitince `int i = 7;` yazılabilir. İki ayrı blokta bildirilen iki k de birbirini görmez; her biri kendi bloğu bitince yok olur. Aynı adlı iki değişkenin kapsamları üst üste binmiyorsa çakışma da yoktur.

9.9.4 Gölgeleme: static alan ve yerel değişken

Değişkenler yalnızca metotların içinde bildirilmez. Bir değişken sınıfın içinde ama bütün metotların dışında da bildirilebilir. Böyle bir değişkene **alan** (*field*) denir. `static` bir alan sınıfa aittir, sınıftaki bütün metotlar onu görür ve program boyunca yaşar. Yerel değişkenlerin aksine bir çağrının sonunda silinmez.

> **Tanım 9.11**
>
> **Gölgeleme** (*shadowing*), bir metodun içinde, sınıftaki bir alanla aynı adlı yerel değişken bildirilmesidir. Yerel değişkenin kapsamı içinde bu ad yerel değişkeni gösterir; alan gölgede kalır ve değişmez. Alana yine de `SinifAdi.alanAdi` yazımıyla ulaşılabilir.

```
public class AlanGolge {
    static int sayac = 100;          // static alan

    static void dene() {
        int sayac = 1;               // yerel değişken alanı gölgeler
        sayac += 5;
        System.out.println("yerel sayac = " + sayac);
        System.out.println("alan sayac  = " + AlanGolge.sayac);
    }

    public static void main(String[] args) {
        dene();
        System.out.println("main: " + sayac);
    }
}
```

```
yerel sayac = 6
alan sayac  = 100
main: 100
```

`dene` metodunda `int sayac = 1;` yazıldığı anda yerel bir sayac doğar. Bundan sonra metodun içinde `sayac` adı bu yerel değişkeni gösterir. `sayac += 5;` yerel değişkeni 6 yapar. Alanın değeri 100'de kalır; `AlanGolge.sayac` yazımı alanın kendisine ulaşır ve 100 yazılır. `dene` bitince yerel sayac silinir ve `main`'deki `sayac` adı yine alanı gösterir: 100.

Gölgelemenin derlenmesi, yerel değişkenler arasındaki yasakla çelişmez. Yasak, iki yerel değişkenin aynı adla aynı anda görünmesini önler. Alan ile yerel değişken ise farklı türlerdendir ve alan, sınıf adıyla her zaman ayırt edilebilir. Yine de gölgeleme çoğu zaman bir dikkatsizlik sonucudur. Programcı alanı değiştirdiğini sanırken yerel bir kopyayı değiştirir ve hata uzun süre fark edilmez. Alanlara ve yerel değişkenlere farklı adlar vermek en güvenli yoldur.

9.9.5 Yerel değişkenin ömrü ve ilk değeri

Kapsam bir değişkenin *nerede* görünür olduğunu söyler. Bir de değişkenin *ne zaman* var olduğu sorusu vardır. Yerel değişken, bildirim satırı çalışınca doğar ve bloğu bitince ölür. Bir metot her çağrıldığında yerel değişkenleri yeniden doğar. Önceki çağrıdan hiçbir değer hatırlanmaz. `yazdirPozitif`'in ikinci çağrısındaki sayi'nin birinci çağrıdaki −3'ten habersiz olması bundandır.

Bir çağrıdan ötekine değer taşıması gereken bir sayaç bu yüzden yerel olamaz; `static` bir alan olmalıdır. Örneğin bir metodun kaç kez çağrıldığını sayan bir sayaç, metodun içinde `int sayac = 0;` diye bildirilirse her çağrıda 0'dan başlar ve 1'den öteye geçemez.

Alanlarla yerel değişkenler arasında bir fark daha vardır. Bir `static int` alan, değer verilmezse kendiliğinden 0 ile başlar. Yerel değişkenin ise kendiliğinden bir ilk değeri yoktur. Derleyici, değer verilmemiş bir yerel değişkenin okunmasına izin vermez:

```
public class IlkDegerYok {
    static int toplam(int n) {
        int t;
        for (int i = 1; i <= n; i++) t += i;
        return t;
    }
}
```

```
IlkDegerYok.java:4: error: variable t might not have been initialized
        for (int i = 1; i <= n; i++) t += i;
                                     ^
IlkDegerYok.java:5: error: variable t might not have been initialized
        return t;
               ^
2 errors
```

İleti “t değişkenine bir ilk değer verilmemiş olabilir” der. Hata iki satırda bildirilir: `t += i` t'nin eski değerini okur, `return t` de onu okur. Çözüm biriktiriciyi `int t = 0;` ile başlatmaktır. Biriktirici toplama için 0, çarpma için 1 ile başlar.

9.10 Hazır metotlar: Math

Java'nın kütüphanesi binlerce hazır metot içerir. Bunların en sık kullanılanlarından biri `Math` sınıfıdır. `Math`'in metotları `static`'tir; nesne oluşturmadan sınıf adıyla çağrılır: `Math.sqrt(16)`. Bunlar da bu notta yazılan metotlar gibi parametre alır, değer döndürür ve birçoğu aşırı yüklenmiştir.

Çağrı	Sonuç	Dönüş tipi	Not
`Math.abs(-7)`	`7`	`int`	mutlak değer; `int`, `long`, `float`, `double` için aşırı yüklü

Çağrı	Sonuç	Dönüş tipi	Not
`Math.abs(-7.5)`	`7.5`	`double`	argüman `double` olunca `double` sürüm seçilir
`Math.max(3, 9)`	`9`	`int`	büyük olanı döndürür
`Math.max(3, 9.5)`	`9.5`	`double`	3 `double` 'a genişler, `max(double, double)` seçilir
`Math.min(-4, 2)`	`-4`	`int`	küçük olanı döndürür
`Math.sqrt(16)`	`4.0`	`double`	karekök; sonuç her zaman `double`
`Math.sqrt(-1)`	`NaN`	`double`	negatif sayının gerçel karekökü yoktur: “sayı değil”
`Math.pow(2, 10)`	`1024.0`	`double`	üs alma; `int` 'e atamak için `(int)` gerekir
`Math.round(2.5)`	`3`	`long`	en yakın tam sayıya yuvarlar
`Math.round(2.4)`	`2`	`long`	aşağı yuvarlanır
`Math.round(-2.5)`	`-2`	`long`	tam yarıda büyük olana, yani −2'ye yuvarlar

Tablo 9.12: Sık kullanılan `Math` metotları (sonuçlar JDK 21 çıktısıdır).

```
public class MathOrnek {
    public static void main(String[] args) {
        System.out.println(Math.abs(-7));
        System.out.println(Math.max(3, 9));
        System.out.println(Math.sqrt(16));
        System.out.println(Math.pow(2, 10));
        System.out.println(Math.round(2.5));
        System.out.println(Math.sqrt(-1));
        System.out.println(Math.abs(-7.5));
        System.out.println(Math.max(3, 9.5));
        System.out.println(Math.round(-2.5));
        System.out.println(Math.round(2.4));
        int p = (int) Math.pow(2, 10);
        System.out.println(p);
        System.out.println(Math.abs(Integer.MIN_VALUE));
        System.out.println(Math.min(-4, 2));
    }
}
```

```
7
9
4.0
1024.0
3
NaN
7.5
9.5
-2
2
1024
-2147483648
-4
```

Çıktıdaki `4.0` ve `1024.0` sayılarının ondalık noktası, dönüş tipinin `double` olmasından gelir. `Math.sqrt(16)`'nın sonucu matematikte tam sayı olsa da metot her zaman `double` döndürür. Bir metodu kullanırken sonucun değeri kadar tipi de önemlidir. Dönüş tipi `double` ya da `long` olan bir metodun sonucu bir `int` değişkene doğrudan atanamaz:

```
public class PowInt {
    public static void main(String[] args) {
        int p = Math.pow(2, 10);
    }
}
```

```
PowInt.java:3: error: incompatible types: possible lossy conversion from double to int
        int p = Math.pow(2, 10);
                        ^
1 error
```

```
public class RoundInt {
    public static void main(String[] args) {
        int r = Math.round(2.5);
    }
}
```

```
RoundInt.java:3: error: incompatible types: possible lossy conversion from long to int
        int r = Math.round(2.5);
                          ^
1 error
```

İki hata da daraltma hatasıdır. `Math.pow` `double`, `Math.round(double)` ise `long` döndürür. Programcı sonucun `int`'e sığacağından eminse açık dönüşüm yazar: `int p = (int) Math.pow(2, 10);` derlenir ve 1024 olur.

`Math.round(-2.5)`'in −3 değil −2 vermesi şaşırtıcı gelebilir. `Math.round`, sayıya 0,5 ekleyip sonucun tabanını alır gibi davranır: *−2,5 + 0,5 = −2*. Tam yarıdaki sayılar hep büyük olana, yani artı sonsuza doğru yuvarlanır: 2,5 → 3 ve −2,5 → −2. Yine `Math.abs(Integer.MIN_VALUE)` çağrısının sonucu da negatiftir (−2147483648). `int`'in en küçük değerinin mutlak değeri olan 2147483648, `int`'in en büyük değerinden (2147483647) bir fazladır ve `int`'e sığmaz. Basamak sayısını bulan metot bu yüzden sayıyı önce `long`'a çevirir.

Hazır bir metodu kullanmadan önce belgesinde parametreleri, dönüş tipi ve özel durumlar okunmalıdır. `Math.sqrt(-1)`'in bir hata fırlatmak yerine `NaN` döndürmesi ya da `Math.round`'un `long` döndürmesi, ancak belgeye bakılınca öğrenilir.

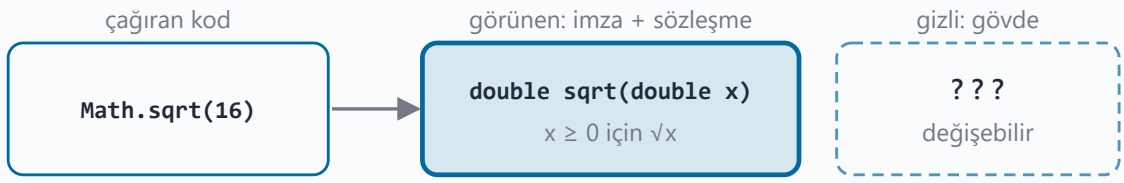
9.11 Metot tasarımı

Derlenen ve doğru sonuç veren bir metot her zaman iyi bir metot değildir. Yüz satırlık, beş iş birden yapan, sonucu hem ekrana yazıp hem bir alana kaydeden bir metot çalışabilir, ama okunması, denenmesi ve değiştirilmesi zordur. İyi bir metodu kötüsünden ayıran, nasıl bölündüğü ve nasıl adlandırıldığıdır.

9.11.1 Soyutlama

Tanım 9.12

Soyutlama (*abstraction*), bir işin ayrıntılarını gizleyip yalnızca o işin ne yaptığını gösteren bir ad ve arayüz sunmaktır. Bir metodun soyutlaması, imzası ve **sözleşmesidir**: ne aldığı, ne döndürdüğü ve hangi koşullarda doğru çalıştığı.



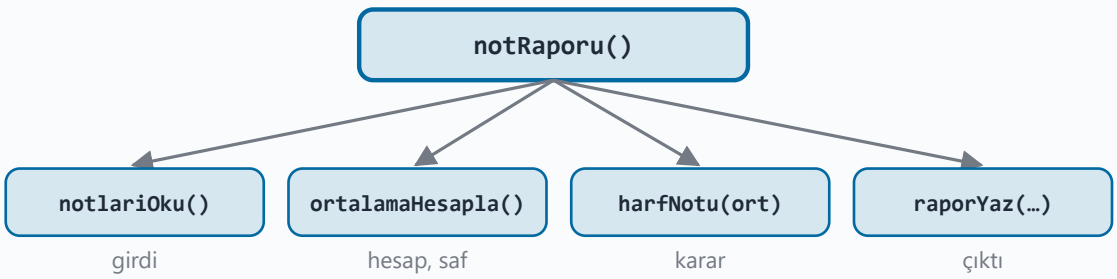
Şekil 9.19: Soyutlama. Çağıran kod yalnızca `Math.sqrt(16)` yazar. Görünen kısım imza (`double sqrt(double x)`) ve sözleşmedir (x ≥ 0 için √x). Gövde gizlidir ve değişebilir.

Metot, çağıran koddan uygulama ayrıntısını saklar. İyi adlandırılmış bir metot *ne* yapıldığını söyler; *nasıl* yapıldığı gövdede kalır. `Math.sqrt`'ü kullanan bir öğrenci, karekökün Newton yöntemiyle mi yoksa işlemcinin özel bir komutuyla mı hesaplandığını bilmez ve bilmesi de gerekmez.

Soyutlamanın önemli bir kazancı vardır: gövde değişse bile imza aynı kaldıkça çağıranlar etkilenmez. Asal sayı metodu önce 2'den n − 1'e kadar bütün bölenleri deneyen yavaş bir döngüyle yazılmış olsun. Sonra yalnızca √n'ye kadar bakan hızlı sürümle değiştirilsin. `asalMi(n)`'i çağıran yüz satırın hiçbiri değişmez; hepsi kendiliğinden hızlanır.

9.11.2 Ayrıştırma

Büyük bir görev, her biri tek bir sorumluluğu olan alt görevlere bölünür. Her alt görev için girdinin, çıktının ve hatalı girdide ne olacağının belirlenmesi gerekir. Üst düzey metot bu alt görevleri anlamlı bir sırada birleştirir. Böyle yazılmış bir üst düzey metot, okunurken bir içindekiler tablosu gibidir. Bu yaklaşıma **ayrıştırma** (*decomposition*) denir.



Şekil 9.20: Not raporu görevinin parçalanması. `notRaporu()` dört alt metodu çağırır: notları okuyan (girdi), ortalamayı hesaplayan (hesap, saf), harf notunu belirleyen (karar) ve raporu yazan (çıktı).

Aşağıdaki program, şekildeki ayrıştırmayı küçük bir örnekle uygular. Okuma işi kısa olduğu için ayrı bir metoda alınmamış, `notRaporu`'nun ilk üç satırına yazılmıştır. Harf notu sınırları örnek olarak seçilmiştir.

```java
import java.util.Scanner;

public class NotRaporu {
    static double ortalamaHesapla(int a, int b, int c) {
        return (a + b + c) / 3.0;
    }

    static String harfNotu(double ort) {
        if (ort >= 85) return "AA";
        if (ort >= 70) return "BB";
        if (ort >= 60) return "CC";
        if (ort >= 50) return "DD";
        return "FF";
    }

    static void raporYaz(double ort, String harf) {
        System.out.println("Ortalama: " + Math.round(ort * 100) / 100.0);
        System.out.println("Harf notu: " + harf);
    }

    static void notRaporu(Scanner giris) {
        int n1 = giris.nextInt();              // notları oku
        int n2 = giris.nextInt();
        int n3 = giris.nextInt();
        double ort = ortalamaHesapla(n1, n2, n3);
        raporYaz(ort, harfNotu(ort));
    }

    public static void main(String[] args) {
        notRaporu(new Scanner(System.in));
    }
}
```

Üç farklı girdiyle çıktılar:

Girdi	Ortalama (hesap)	Çıktı
`70 85 91`	246 / 3 = 82	`Ortalama: 82.0 Harf notu: BB`
`50 60 71`	181 / 3 ≈ 60,33	`Ortalama: 60.33 Harf notu: CC`

Girdi	Ortalama (hesap)	Çıktı
`40 45 52`	137 / 3 ≈ 45,67	`Ortalama: 45.67 Harf notu: FF`

Tablo 9.13: `NotRaporu` programının üç girdi için çıktısı.

Her metodun tek bir işi vardır ve her biri ayrı ayrı denenebilir. `ortalamaHesapla` yalnızca hesap yapar. Bölmenin `3.0` ile yapıldığına dikkat edilmelidir: `(a + b + c) / 3` yazılsaydı iki `int` arasında tam sayı bölmesi yapılırdı, 181 / 3 için 60,33 yerine 60 bulunurdu. Dönüş tipinin `double` olması bu bölmeyi kendiliğinden kesirli yapmaz; dönüşüm bölme bittikten sonra olur. `harfNotu` yalnızca bir karar verir; art arda erken dönüşlerle yazıldığı için her eşik bir satırdır. `raporYaz` yalnızca çıktıyla ilgilenir. Ortalamayı iki basamağa yuvarlamak için `Math.round(ort * 100) / 100.0` kullanılır: 60,333… × 100 = 6033,3… → 6033 → 60,33. Harf notu sınırları değişirse yalnızca `harfNotu`, raporun biçimi değişirse yalnızca `raporYaz` değişir.

9.11.3 İyi metot tasarımının ilkeleri

Bir metodun iyi tasarlanıp tasarlanmadığı birkaç soruyla sınanabilir:

İlke	Soru	Kötü örnek → iyi örnek
Tek görev	Metot tek cümleyle açıklanabiliyor mu?	hem okuyan hem hesaplayan hem yazan `isle()` → `notlariOku`, `ortalamaHesapla`, `raporYaz`
Anlamlı ad	Ad sonucu ya da eylemi söylüyor mu?	`f(x)`, `islem2()` → `basamakSayisi(x)`, `ciftMi(x)`
Az parametre	Girdilerin hepsi gerekli mi?	kullanılmayan bir `int mod` parametresi → parametreyi kaldırmak
Belirgin sonuç	Dönüş ve yan etkiler açık mı?	hem yazan hem döndüren metot → yalnızca döndüren metot
Test edilebilirlik	Metot bağımsız örneklerle denenebilir mi?	klavyeden okuyan hesap metodu → değeri parametre olarak alan metot

Tablo 9.14: İyi metot tasarımının ilkeleri ve her ilkenin sorusu.

Bu ilkeler birbirini destekler. Tek görevli bir metoda anlamlı bir ad vermek kolaydır. Az parametreli ve sonucu döndüren bir metot kolay denenir. Uzunluk için kesin bir sınır yoktur. Yine de bir metot bir ekran yüksekliğini aşıyorsa bölünmeye adaydır. Gövdesinde “şimdi şunu yap” diye ayrılan bölümler varsa, bu bölümlerin her biri büyük olasılıkla ayrı bir metot olmalıdır.

9.12 Örnek metotlar

Aşağıdaki örnekler, metotlarla ilgili kavramları küçük ve tanıdık problemler üzerinde bir araya getirir. Her metot önce yazılır, sonra çağrı yığınıyla ya da bir izleme tablosuyla adım adım yürütülür.

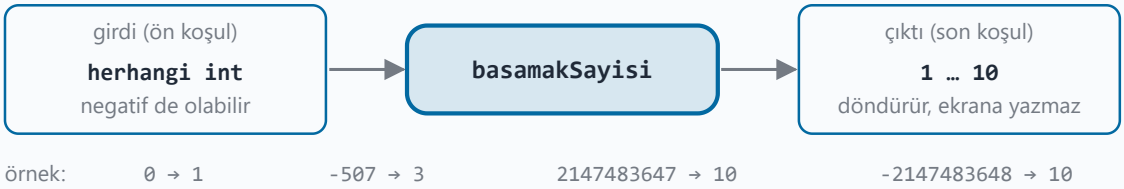
9.12.1 Önce sözleşme

Bir metot yazılmadan önce ne yapacağı açıkça belirlenmelidir. Metodun adı yaptığı işi söylemeli, parametrelerin geçerli değer aralığı tanımlanmalı, dönüş değerinin anlamı ve hatalı girdide ne olacağı bilinmelidir. Bu bilgilerin toplamına metodun sözleşmesi denir.

Tanım 9.13

Sözleşme (*contract*), bir metodun çağıran ile arasındaki anlaşmadır. **Ön koşul**, çağıranın sağlaması gereken koşuldur (ör. "n ≥ 0"). **Son koşul**, ön koşul sağlandığında metodun garanti ettiği sonuçtur (ör. "n'nin basamak sayısını döndürür").

```
// Görev : sayının onluk basamak sayısını döndürür.
// Girdi : herhangi bir int (negatif de olabilir).
// Çıktı : 1..10; 0 için 1, -507 için 3.
static int basamakSayisi(int sayi) { … }
```



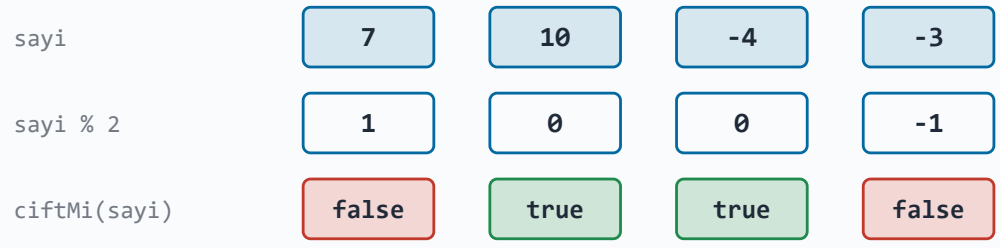
Şekil 9.21: `basamakSayisi`'nin sözleşmesi. Ön koşul: herhangi bir `int`, negatif de olabilir. Son koşul: 1 ile 10 arasında bir sayı döndürür, ekrana yazmaz. Örnekler: 0 → 1, −507 → 3, 2147483647 → 10, −2147483648 → 10.

Sözleşmedeki örnekler, metot yazılmadan önce belirlenmiş birer test durumudur. Sınır değerleri bilerek içerir: 0 tek basamaklıdır ama döngüsü hiç dönmeyecek bir girdidir; `int`'in en büyük ve en küçük değerleri ise en uzun sayılardır. Gövde yazıldıktan sonra bu örneklerin hepsi denenir. Sözleşme yazılmadan başlanan bir metotta "0 için ne dönmeli?" sorusu çoğu zaman ancak bir hata çıktığında sorulur.

9.12.2 Tek mi, çift mi?

```
static boolean ciftMi(int sayi) {
    return sayi % 2 == 0;
}
```

Bir sayı 2'ye bölündüğünde kalan 0 ise sayı çifttir. `ciftMi` bu karşılaştırmanın sonucunu, yani bir `boolean` değeri doğrudan döndürür. `if (sayi % 2 == 0) return true; else return false;` biçiminde yazmak da doğrudur, ama gereksiz uzundur: `sayi % 2 == 0` ifadesi zaten `true` ya da `false`'tur.



Java'da negatif sayının kalanı negatif olabilir (-3 % 2 = -1); bu yüzden == 0 ile bakılır.

Şekil 9.22: `ciftMi` dört girdi için: 7 % 2 = 1 → `false`; 10 % 2 = 0 → `true`; −4 % 2 = 0 → `true`; −3 % 2 = −1 → `false`.

Şekildeki son sütun önemli bir ayrıntıyı gösterir. Java'da negatif bir sayının kalanı negatif olabilir: kalanın işareti bölünenin işaretini izler ve *−3 % 2 = −1*'dir. Tek sayı denetimini `sayi % 2 == 1` diye yazmak bu yüzden yanlıştır; −3 için `false` verir ve −3'ü çift sayar. `== 0` ile çiftliğe bakmak her işarette doğru çalışır. Programın gerçek çıktısı, her satırda sayıyı, kalanı ve metodun sonucunu verir:

```
public class CiftMi {
    static boolean ciftMi(int sayi) {
        return sayi % 2 == 0;
    }

    public static void main(String[] args) {
        if (ciftMi(7)) System.out.println("çift");
        else System.out.println("tek");
        int[] denenecek = {7, 10, -4, -3, 0};
        for (int s : denenecek)
            System.out.println(s + " " + (s % 2) + " " + ciftMi(s));
    }
}
```

```
tek
7 1 false
10 0 true
-4 0 true
-3 -1 false
0 0 true
```

Metot ekrana yazmaz, sonucu döndürür. Bu yüzden `if` koşulunda doğrudan kullanılır. `ciftMi(7)` çağrısı izlendiğinde, çerçevede sayi = 7 olur, *7 % 2 = 1* ve *1 == 0* yanlış olduğundan `false` döner. Dönüş değeri çağrının yerine geçer: `if (false)`. Then dalı atlanır, `else` dalı "tek" yazar. Ne yazılacağına metot değil, çağıran karar vermiştir. Aynı `ciftMi`, başka bir programda çift sayıları saymak ya da bir listeyi süzmek için değiştirilmeden kullanılabilir.

9.12.3 Basamak sayısı

```
static int basamakSayisi(int sayi) {
    long kalan = Math.abs((long) sayi);
    if (kalan == 0) return 1;
    int adet = 0;
    while (kalan > 0) {
        adet++;
        kalan /= 10;
    }
    return adet;
}
System.out.println(basamakSayisi(-507));
```

Metot önce sayının mutlak değerini alır, çünkü −507 ile 507'nin basamak sayısı aynıdır. Ardından 0 için erken dönüş yapar. Sonra sayı 10'a bölündükçe bir basamak eksilir; sayı 0 olana kadar kaç kez bölündüğü sayılır. Sonuç ekrana yazılmaz, döndürülür; çağıran onu istediği gibi kullanır.

Tablo 9.15 `basamakSayisi(-507)` çağrısında döngünün turlarını verir. Döngüden önce *kalan = |−507| = 507* olur; *507 == 0* yanlış olduğu için erken dönüş yapılmaz ve adet = 0 ile döngüye girilir.

Tur	Koşul `kalan > 0`	adet	kalan
1. tur	507 > 0 → doğru	1	507 / 10 = 50
2. tur	50 > 0 → doğru	2	50 / 10 = 5
3. tur	5 > 0 → doğru	3	5 / 10 = 0
çıkış	0 > 0 → yanlış	3	—

Tablo 9.15: `basamakSayisi(-507)` çağrısında `while` döngüsünün turları. `kalan /= 10` tam sayı bölmesidir.

Her turda `kalan /= 10` sayının son basamağını atar: 507 → 50 → 5 → 0. Döngü üç kez döner ve 3 döndürülür. `main`'deki `println`'in argümanı için önce bu çağrı yapılmış, dönen 3 çağrının yerine geçmiş ve `println(3)` çalışmıştır.



Şekil 9.23: (a) `Math.abs`'ten sonra: sayi = −507, kalan = 507. (b) Döngü bitti: kalan = 0, adet = 3; `return 3` ile çerçeve kapanıyor.

Sözleşmedeki bütün örnekler gerçekten denenmiştir:

```
public class BasamakSayisi {
    static int basamakSayisi(int sayi) {
        long kalan = Math.abs((long) sayi);
        if (kalan == 0) return 1;
        int adet = 0;
        while (kalan > 0) {
            adet++;
            kalan /= 10;
        }
        return adet;
    }

    public static void main(String[] args) {
        System.out.println(basamakSayisi(-507));
        System.out.println(basamakSayisi(0));
        System.out.println(basamakSayisi(9));
        System.out.println(basamakSayisi(10));
        System.out.println(basamakSayisi(2147483647));
        System.out.println(basamakSayisi(-2147483648));
    }
}
```

```
3
1
1
2
10
10
```

Birinci satırdaki `(long)` dönüşümünün de bir nedeni vardır. `int`'in en küçük değeri olan −2147483648'in mutlak değeri `int`'e sığmaz ve `Math.abs` bu sayı için yine −2147483648 döndürür. Dönüşüm yapılmadan yazılmış bir sürüm bu yüzden yanlış sonuç verir:

```
public class BasamakHatali {
    static int basamakSayisi(int sayi) {
        int kalan = Math.abs(sayi);      // long'a çevrilmeden
        if (kalan == 0) return 1;
        int adet = 0;
        while (kalan > 0) {
            adet++;
            kalan /= 10;
        }
        return adet;
    }

    public static void main(String[] args) {
        System.out.println(Math.abs(-2147483648));
        System.out.println(basamakSayisi(-2147483648));
        System.out.println(basamakSayisi(-507));
    }
}
```

```
-2147483648
0
3
```

Hatalı sürümde kalan negatif kalır. 0'a eşit olmadığı için erken dönüş yapılmaz. *kalan > 0* koşulu ilk denemede yanlış olduğu için döngü hiç dönmez ve 0 döner. Oysa sözleşme "1 ile 10 arası" diyordu. Doğru sürüm sayıyı önce `long`'a çevirir. `long`'da 2147483648 rahatça temsil edilir ve `Math.abs`'in `long` sürümü doğru sonucu verir. Bu hata, yalnızca sınır değerle denendiğinde ortaya çıkan bir hatadır ve sözleşmeye sınır değerleri yazmanın değerini gösterir.

9.12.4 Asal mı?

1'den büyük olup kendisinden ve 1'den başka böleni olmayan sayıya asal sayı denir. Bir sayının asal olup olmadığını anlamak için olası bölenler sırayla denenir. Bir bölen bulunduğu anda cevap kesinleşir ve metot erken dönüş yapar.

```
static boolean asalMi(int n) {
    if (n < 2) return false;
    for (int i = 2; i <= n / i; i++)
        if (n % i == 0) return false;
    return true;
}
```

Döngü koşulundaki `i <= n / i` ifadesi ilk bakışta tuhaf görünebilir. Bu koşul, i'nin n'nin karekköküne kadar denenmesini sağlar. Gerekçe şudur: n iki sayının çarpımı olarak yazılabiliyorsa, yani $n = p \times q$ ise, p ile q'dan en az biri √n'den büyük olamaz. İkisi de √n'den büyük olsaydı çarpımları n'yi aşardı. Bu yüzden n'nin √n'den büyük bir böleni varsa, onun √n'den küçük bir eşi de vardır ve o eş daha önce denenmiş olur. Örneğin 91 = 7 × 13'tür ve √91 ≈ 9,54'tür. 13'e kadar beklemeye gerek yoktur; 7 denendiğinde bölen bulunur.

`i <= n / i` koşulu `i * i <= n` koşulunun tam sayı bölmesiyle yazılmış biçimidir. Bu biçim, n çok büyük olduğunda `i * i` çarpımının `int` sınırını aşmasını önler. Tablo 9.16 koşulun 37 için nasıl işlediğini gösterir. Tablonun değerleri, döngüye yazdırma satırı eklenmiş bir sürüm JDK 21 ile çalıştırılarak elde edilmiştir.

i	`n / i`	Koşul `i <= n / i`	`n % i`	Bölüyor mu?
2	37 / 2 = 18	2 ≤ 18 → doğru	37 % 2 = 1	hayır
3	37 / 3 = 12	3 ≤ 12 → doğru	37 % 3 = 1	hayır
4	37 / 4 = 9	4 ≤ 9 → doğru	37 % 4 = 1	hayır
5	37 / 5 = 7	5 ≤ 7 → doğru	37 % 5 = 2	hayır
6	37 / 6 = 6	6 ≤ 6 → doğru	37 % 6 = 1	hayır
7	37 / 7 = 5	7 ≤ 5 → yanlış	—	döngü biter: `return true`

Tablo 9.16: `asalMi(37)` : yalnızca i ≤ n / i olan i değerleri denenir (√37 ≈ 6,08).

i = 2'den 6'ya kadar beş değer denenir ve hiçbiri 37'yi bölmez. i = 7'de *37 / 7 = 5*'tir ve *7 ≤ 5* yanlıştır; döngü biter ve metot `true` döndürür. 2'den 36'ya kadar bütün sayıları deneyen basit bir sürüm 35 kalan hesaplardı; bu sürüm yalnızca 5 hesaplar. Sonuç aynıdır. Asal olmayan bir sayıda ise döngü ilk bölende biter. 49 = 7 × 7 için i = 7'de *7 ≤ 49 / 7 = 7* doğrudur ve *49 % 7 = 0* bulunur:

i	`n / i`	Koşul `i <= n / i`	`n % i`	Bölüyor mu?
2	49 / 2 = 24	2 ≤ 24 → doğru	49 % 2 = 1	hayır
3	49 / 3 = 16	3 ≤ 16 → doğru	49 % 3 = 1	hayır
4	49 / 4 = 12	4 ≤ 12 → doğru	49 % 4 = 1	hayır
5	49 / 5 = 9	5 ≤ 9 → doğru	49 % 5 = 4	hayır
6	49 / 6 = 8	6 ≤ 8 → doğru	49 % 6 = 1	hayır
7	49 / 7 = 7	7 ≤ 7 → doğru	49 % 7 = 0	evet: `return false`

Tablo 9.17: `asalMi(49)` : koşul i = 7 için hâlâ doğrudur, çünkü 7 × 7 = 49'dur.

Bu örnek koşulda neden `<` değil `<=` kullanıldığını gösterir. `i < n / i` yazılsaydı 7 hiç denenmez ve 49 asal sanılırdı. Tam kare sayılar bu tür sınır hatalarını yakalamak için iyi test girdileridir.

Metot 15 ve 7 için izlenebilir. `asalMi(15)` 'te n = 15 için önce *15 < 2* yanlıştır. i = 2 için *2 ≤ 15 / 2 = 7* doğrudur ama *15 % 2 = 1*'dir; i artar. i = 3 için *3 ≤ 15 / 3 = 5* doğrudur ve *15 % 3 = 0*'dır: `return false` çalışır, kalan turlar hiç çalışmaz. `asalMi(7)` 'de i = 2 için *2 ≤ 7 / 2 = 3* doğrudur, *7 % 2 = 1*'dir. i

= 3 için *3 ≤ 7 / 3 = 2* yanlıştır: √7'ye kadar bölen bulunmamıştır, döngü erken dönüş olmadan biter ve `true` döner.

```
public class AsalMi {
    static boolean asalMi(int n) {
        if (n < 2) return false;
        for (int i = 2; i <= n / i; i++)
            if (n % i == 0) return false;
        return true;
    }

    public static void main(String[] args) {
        System.out.println(asalMi(15));
        System.out.println(asalMi(7));
        System.out.println(asalMi(37));
        System.out.println(asalMi(1) + " " + asalMi(2) + " " + asalMi(4) + " " + asalMi(49)
+ " " + asalMi(97));
        for (int k = 1; k <= 30; k++)
            if (asalMi(k)) System.out.print(k + " ");
        System.out.println();
    }
}
```

```
false
true
true
false true false false true
2 3 5 7 11 13 17 19 23 29
```

Son satır, metodun 1'den 30'a kadar sayılarla bir döngüde çağrılmasıyla elde edilen asal sayılardır. `asalMi(1)`'in `false`, `asalMi(2)`'nin `true` döndüğüne dikkat edilmelidir. 1 asal değildir ve ilk satırdaki erken dönüşle elenir. 2 için döngü koşulu *2 ≤ 2 / 2 = 1* baştan yanlıştır; döngü hiç dönmez ve `true` döner.

9.12.5 EBOB: Öklid algoritması

İki sayının en büyük ortak böleni (EBOB), ikisini de kalansız bölen en büyük sayıdır. 1'den başlayıp küçük sayıya kadar bütün sayıları denemek yerine, yaklaşık 2300 yıllık Öklid algoritması çok daha hızlı bir yol sunar: *ebob(a, b) = ebob(b, a % b)* ve b = 0 olunca cevap a'dır.

```
static int ebob(int a, int b) {
    while (b != 0) {
        int r = a % b;
        a = b;
        b = r;
    }
    return a;
}
System.out.println(ebob(48, 18));  // main
```

Algoritmanın doğruluğu şu gözleme dayanır: a'yı ve b'yi bölen her sayı, $a \% b = a - k \times b$ farkını da böler. Tersine, b'yi ve a % b'yi bölen her sayı a'yı da böler. Bu yüzden (a, b) çiftinin ortak bölenleri ile (b, a % b) çiftinin ortak bölenleri aynıdır ve EBOB'ları da eşittir. Sayılar her turda küçülür. Sonunda b = 0 olur ve *ebob(a, 0) = a*'dır, çünkü 0'ı her sayı böler.

Tur	`r = a % b`	a	b	Koşul `b != 0`
başlangıç	—	48	18	18 != 0 → doğru
1. tur	48 % 18 = 12	18	12	12 != 0 → doğru
2. tur	18 % 12 = 6	12	6	6 != 0 → doğru
3. tur	12 % 6 = 0	6	0	0 != 0 → yanlış

Tablo 9.18: `ebob(48, 18)` 'in turları (değerler JDK 21 ile çalıştırılan izleme sürümünden alınmıştır).

Başlangıçta a = 48, b = 18 ve *18 != 0* doğrudur. 1. turda *r = 48 % 18 = 12* (48 = 2 × 18 + 12); a = 18, b = 12 olur. Bu, *ebob(48, 18) = ebob(18, 12)* demektir. 2. turda *r = 18 % 12 = 6*; a = 12, b = 6. 3. turda *r = 12 % 6 = 0*; a = 6, b = 0. Koşul *0 != 0* yanlıştır ve 6 döndürülür. Denetim: *48 = 6 × 8*, *18 = 6 × 3* ve 8 ile 3'ün 1'den başka ortak böleni yoktur. Yalnızca üç tur yetmiştir.



Şekil 9.24: (a) 1. turdan sonra `ebob` 'un çerçevesi: a = 18, b = 12, r = 12. (b) Döngü bitti: a = 6, b = 0; `return 6` .

Sayılar ters sırayla verilirse ilk tur onları kendiliğinden yer değiştirir: *18 % 48 = 18* olduğu için `ebob(18, 48)` ilk turdan sonra `ebob(48, 18)` durumuna gelir. Metot ayrıca 0 ile de doğru çalışır: `ebob(7, 0)` döngüye hiç girmez ve 7 döndürür.

```
public class Ebob {
    static int ebob(int a, int b) {
        while (b != 0) {
            int r = a % b;
            a = b;
            b = r;
        }
        return a;
    }

    static int ekok(int a, int b) {
        return a / ebob(a, b) * b;
    }

    public static void main(String[] args) {
        System.out.println(ebob(48, 18));
        System.out.println(ekok(12, 18));
        System.out.println(ebob(18, 48) + " " + ebob(17, 5) + " " + ebob(7, 0) + " " +
ebob(100, 75));
        System.out.println(ekok(4, 6) + " " + ekok(5, 7) + " " + ekok(8, 12));
    }
}
```

```
6
36
6 1 7 25
12 35 24
```

9.12.6 EKOK: metot içinden metot çağrısı

İki sayının en küçük ortak katı (EKOK) EBOB'la kolayca bulunur: *ekok(a, b) = a × b / ebob(a, b)*. EBOB'u hesaplayan metot zaten yazılmış olduğu için `ekok` metodu onu çağırır ve işi tek satırda biter:

```
static int ekok(int a, int b) {
    return a / ebob(a, b) * b;
}
// ebob: Öklid algoritmasıyla EBOB
System.out.println(ekok(12, 18));  // main
```

Adım	Satır	Çağrı yığını (üstteki önce)	Değişkenler (üstteki çerçeve)	Çıktı
1	5	`main()`	—	—
2	1	`ekok(12, 18) ← main()`	a=12, b=18	—
3	2	`ebob(12, 18) ← ekok(12, 18) ← main()`	a=12, b=18	—
4	2	`ebob(12, 18) ← ekok(12, 18) ← main()`	a=6, b=0	—

Adım	Satır	Çağrı yığını (üstteki önce)	Değişkenler (üstteki çerçeve)	Çıktı
5	2	`ekok(12, 18) ← main()`	a=12, b=18	—
6	2	`ekok(12, 18) ← main()`	a=12, b=18	—
7	5	`main()`	—	36

Tablo 9.19: `ekok(12, 18)` çağrısının izleme tablosu. 3. ve 4. adımlarda yığında üç çerçeve vardır.

`main`, `ekok(12, 18)`'i çağırır ve `ekok`'un çerçevesi açılır: a = 12, b = 18. `return` satırındaki ifadeyi hesaplamak için önce `ebob(a, b)` çağrılmalıdır. Yığında üç çerçeve olur. `ebob` kendi döngüsünü çalıştırır: (12, 18) → (18, 12) → (12, 6) → (6, 0) ve 6 döndürür. Adları aynı olsa da `ebob`'un a ve b'si, `ekok`'un a ve b'sinden ayrı kutulardır. `ebob`'un a'sı 6'ya inerken `ekok`'un a'sı 12'de kalır. Çağrının yerine 6 geçer ve ifade hesaplanır: *12 / 6 × 18 = 2 × 18 = 36*. `ekok` 36'yı döndürür ve 36 yazılır.



Şekil 9.25: `ebob`, `ekok`'un üstünde: `ebob`'un a = 6, b = 0 olmuştur ve 6 döndürmek üzeredir; alttaki `ekok`'un a = 12, b = 18 değişmemiştir.

Kodda formülün `a * b / ebob(a, b)` yerine `a / ebob(a, b) * b` biçiminde yazılmasının bir nedeni vardır. Önce bölmek, ara sonucun büyümesini önler. a × b çarpımı `int`'e sığmayacak kadar büyük olabilir. Oysa a, EBOB'a her zaman kalansız bölünür ve bölüm küçüktür. İki yazımın farkı büyük sayılarda ortaya çıkar:

```
public class EkokTasma {
    static int ebob(int a, int b) {
        while (b != 0) {
            int r = a % b;
            a = b;
            b = r;
        }
        return a;
    }

    static int ekokOnceCarp(int a, int b) {
        return a * b / ebob(a, b);
    }

    static int ekok(int a, int b) {
        return a / ebob(a, b) * b;
    }

    public static void main(String[] args) {
        System.out.println(ekokOnceCarp(12, 18) + " " + ekok(12, 18));
        System.out.println(ekokOnceCarp(100000, 300000) + " " + ekok(100000, 300000));
    }
}
```

```
36 36
-647 300000
```

12 ve 18 için iki yazım da 36 verir. 100000 ve 300000 için ise *100000 × 300000 = 30.000.000.000* çarpımı `int`'in üst sınırını (yaklaşık 2,1 milyar) aşar. Taşan çarpım anlamsız bir negatif sayıya dönüşür ve sonuç −647 çıkar. Önce bölen yazım *100000 / 100000 × 300000 = 300000* ile doğru cevabı verir. Bu hatanın sinsi yanı, derleyicinin de çalışma zamanının da hiçbir uyarı vermemesidir.

Metot içinden metot çağırmak kod tekrarını önler. EBOB'u hesaplayan döngü `ekok`'un içine bir daha yazılmamıştır. `ebob`'da bir hata bulunup düzeltilirse `ekok` da kendiliğinden düzelir.

9.12.7 Yeniden kullanım: enBuyuk3

Üç sayının en büyüğünü bulmak için yeni bir karşılaştırma zinciri yazmak gerekmez. İki sayının büyüğünü veren metot iki kez kullanılır: önce ilk iki sayının büyüğü bulunur, sonra bu sonuç üçüncü sayıyla karşılaştırılır.

```
static int enBuyuk(int a, int b) {
    if (a > b) return a;
    return b;
}
static int enBuyuk3(int a, int b, int c) {
    return enBuyuk(enBuyuk(a, b), c);
}
System.out.println(enBuyuk3(4, 9, 7));  // main
```

Adım	Satır	Çağrı yığını (üstteki önce)	Değişkenler (üstteki çerçeve)	Çıktı
1	8	`main()`	—	—
2	5	`enBuyuk3(4, 9, 7)` ← `main()`	a=4, b=9, c=7	—
3	6	`enBuyuk3(4, 9, 7)` ← `main()`	a=4, b=9, c=7	—
4	1, 2	`enBuyuk(4, 9)` ← `enBuyuk3(4, 9, 7)` ← `main()`	a=4, b=9	—
5	3	`enBuyuk(4, 9)` ← `enBuyuk3(4, 9, 7)` ← `main()`	a=4, b=9	—
6	6	`enBuyuk3(4, 9, 7)` ← `main()`	a=4, b=9, c=7	—
7	2	`enBuyuk(9, 7)` ← `enBuyuk3(4, 9, 7)` ← `main()`	a=9, b=7	—
8	6	`enBuyuk3(4, 9, 7)` ← `main()`	a=4, b=9, c=7	—
9	8	`main()`	—	9

Tablo 9.20: `enBuyuk3(4, 9, 7)` çağrısının izleme tablosu. `enBuyuk` iki kez, iki ayrı çerçeveyle çağrılır.

`enBuyuk3`'ün çerçevesinde a = 4, b = 9, c = 7 olur. `return enBuyuk(enBuyuk(a, b), c);` satırında iç içe çağrı vardır ve önce içteki hesaplanır: `enBuyuk(4, 9)`. Yeni çerçevede a = 4, b = 9 olur. *4 > 9* yanlış olduğu için ilk `return` atlanır ve `return b;` 9'u döndürür. İfade sadeleşmiştir: `enBuyuk(9, 7)`. Aynı metot ikinci kez, yepyeni bir çerçeveyle çağrılır: a = 9, b = 7. *9 > 7* doğrudur ve `return a;` 9'u döndürür. `enBuyuk3` de bu 9'u döndürür ve 9 yazılır.



Şekil 9.26: (a) İçteki çağrı: `enBuyuk(4, 9)`'un çerçevesi `enBuyuk3`'ün üstünde. (b) Dıştaki çağrı: `enBuyuk(9, 7)` 9 döndürüyor.

Bu tasarımda karşılaştırma mantığı tek bir yerdedir. Bir hata varsa, örneğin `>` yerine yanlışlıkla `<` yazılmışsa, tek bir yerde düzeltilir. Sayıların hangi sırada verildiği sonucu değiştirmez; eşit ve negatif sayılarla da doğru çalışır:

```
public class EnBuyuk3 {
    static int enBuyuk(int a, int b) {
        if (a > b) return a;
        return b;
    }

    static int enBuyuk3(int a, int b, int c) {
        return enBuyuk(enBuyuk(a, b), c);
    }

    public static void main(String[] args) {
        System.out.println(enBuyuk3(4, 9, 7));
        System.out.println(enBuyuk3(9, 4, 7) + " " + enBuyuk3(4, 7, 9) + " " + enBuyuk3(5,
5, 5) + " " + enBuyuk3(-2, -7, -1));
    }
}
```

```
9
9 9 5 -1
```

9.12.8 double döndüren metotlar: sıcaklık dönüşümü

Celsius ile Fahrenheit arasındaki dönüşüm *F = C × 9 / 5 + 32* ve *C = (F − 32) × 5 / 9* formülleriyle yapılır. Sonuçlar kesirli olabileceği için iki metot da `double` alır ve `double` döndürür.

```
static double fahrenheit(double c) {
    return c * 9 / 5 + 32;
}
static double celsius(double f) {
    return (f - 32) * 5 / 9;
}
double f = fahrenheit(25);          // main
System.out.println(f);
System.out.println(celsius(f));
```

Adım	Satır	Çağrı yığını (üstteki önce)	Değişkenler (üstteki çerçeve)	Çıktı
1	7	`main()`	—	—
2	1	`fahrenheit(25.0) ← main()`	c=25.0	—
3	2	`fahrenheit(25.0) ← main()`	c=25.0	—
4	7	`main()`	f=77.0	—
5	8	`main()`	f=77.0	77.0
6	9	`celsius(77.0) ← main()`	f=77.0	77.0
7	5	`celsius(77.0) ← main()`	f=77.0	77.0

Adım	Satır	Çağrı yığını (üstteki önce)	Değişkenler (üstteki çerçeve)	Çıktı
8	9	`main()`	f=77.0	77.0 / 25.0

Tablo 9.21: `fahrenheit(25)` ve `celsius(f)` çağrılarının izleme tablosu.

`fahrenheit(25)` çağrısında argüman `int`'tir ama parametre `double`'dır. Genişletme kendiliğinden yapılır ve c = 25.0 olur. Gövdede *25,0 × 9 = 225,0*; *225,0 / 5 = 45,0*; *45,0 + 32 = 77,0* hesaplanır. `main`'de f = 77.0 olur ve `double` olduğu için `77.0` biçiminde, ondalık noktasıyla yazılır. Sonra `celsius(f)` çağrılır. f'nin değeri, `celsius`'un kendi f parametresine kopyalanır; adların aynı olması yine bir bağ kurmaz. *(77,0 − 32) × 5 / 9 = 45,0 × 5 / 9 = 225,0 / 9 = 25,0* bulunur ve `25.0` yazılır. İki metot birbirinin tersidir. Birinin sonucunu ötekine verip başlangıç değerine dönmek, ikisini birlikte sınamanın kolay bir yoludur.

```
public class Sicaklik {
    static double fahrenheit(double c) {
        return c * 9 / 5 + 32;
    }

    static double celsius(double f) {
        return (f - 32) * 5 / 9;
    }

    public static void main(String[] args) {
        double f = fahrenheit(25);
        System.out.println(f);
        System.out.println(celsius(f));
        System.out.println(fahrenheit(100) + " " + fahrenheit(-40) + " " + celsius(98.6));
        System.out.println(5 / 9 * (f - 32));
        System.out.println((f - 32) * 5 / 9);
    }
}
```

```
77.0
25.0
212.0 -40.0 37.0
0.0
25.0
```

Üçüncü satır bilinen birkaç değeri doğrular: suyun kaynama noktası 100 °C = 212 °F, iki ölçeğin kesiştiği −40 ve insan vücut sıcaklığı 98,6 °F = 37 °C. Dördüncü satır sık yapılan bir hatayı gösterir. Formül `5 / 9 * (f - 32)` diye yazılırsa önce `5 / 9` hesaplanır. İki `int` arasında tam sayı bölmesi 0 verir ve sonuç ne olursa olsun `0.0` çıkar. Doğru yazımda ise önce `(f - 32)`, yani bir `double` hesaplanır; sonraki çarpma ve bölme `double` aritmetiğiyle yapılır ve doğru sonuç (`25.0`) elde edilir.

9.12.9 Saf metot ve yan etki

Tanım 9.14

Yan etki (*side effect*), bir metodun değer döndürmenin dışında dünyada bıraktığı her değişikliktir: ekrana yazmak, bir alanı değiştirmek, bir dosyaya kayıt eklemek gibi. **Saf metot** (*pure method*), aynı girdiler için her zaman aynı sonucu döndüren ve hiçbir yan etkisi olmayan metottur.

İki küçük metot farkı açıkça gösterir:

```
public class SafYan {
    static int kare(int x) {
        return x * x;
    }

    static void kareYaz(int x) {
        System.out.println(x * x);
    }

    public static void main(String[] args) {
        int t = kare(3) + kare(4);
        System.out.println(t);
        kareYaz(4);
        System.out.println(kare(kare(2)));
    }
}
```

```
25
16
16
```

`kare` saftır. `kare(4)` her zaman 16'dır ve her yerde kullanılabilir: bir toplamın içinde (*9 + 16 = 25*), başka bir çağrının argümanı olarak (`kare(kare(2))` = *kare(4) = 16*) ya da bir koşulda. `kareYaz` ise yalnızca ekrana yazar. Sonucu başka bir hesapta kullanılamaz. `kareYaz(3) + kareYaz(4)` derlenmez bile, çünkü `kareYaz` `void` 'dir.

Saf metotlar daha kolay test edilir. `kare(4) == 16` diye denetlemek yeter; ekran çıktısını yakalamaya ya da bir alanın önceki değerini hatırlamaya gerek yoktur. Bu yüzden hesaplama metodu ile ekrana yazma metodu birbirinden ayrılır. Hesap saf bir metotta yapılır, yazma işi ayrı bir metoda ya da `main` 'e bırakılır. Yan etki kaçınılmaz olduğunda, örneğin bir rapor gerçekten yazılacaksa, yan etki açıkça adlandırılır ve sınırlandırılır: `raporYaz` adı ne yaptığını söyler ve başka hiçbir şey yapmaz.

Bir metodun saf olmasını bozan yollar yalnızca ekrana yazmak değildir. Bir `static` alanı değiştiren metot da yan etkilidir. Her çağrıda bir sayacı artıran bir metodun sonucu, kaçıncı kez çağrıldığına göre değişebilir. Klavyeden okuyan ya da rastgele sayı üreten metotlar da saf değildir, çünkü aynı girdiyle çağrıldıklarında farklı sonuçlar verebilir.

9.13 Özet

- **Metot**, bir sınıfın içinde tanımlanan, adı olan ve bir görevi yapan deyimler bütünüdür. Girdisini **parametrelerle** alır, sonucunu **dönüş değeri** olarak verir. Metot bir kez tanımlanır, istenen yerden ve istenen sayıda **çağrılır**.
- Tanımlamada yazılan **biçimsel parametreler** değişkendir; çağrıda yazılan **gerçek argümanlar** değerdir. Argümanlar parametrelere sırayla eşlenir; sayıları ve tipleri uymalıdır.
- Bir çağrıda argümanlar değerlendirilir, parametrelere kopyalanır, gövde çalışır, `return` değeri gönderir ve çağıran kod çağrının yerine geçen değerle sürer.
- Her çağrı **çağrı yığınının** üstüne kendi **çerçevesini** açar. Yalnızca en üstteki çerçeve çalışır; metot bitince çerçevesi bütün yerel değişkenleriyle silinir. Son açılan çerçeve ilk kapanır.
- Başlık erişim belirteci, niteleyici, dönüş tipi, ad ve parametre listesinden oluşur. **İmza**, ad ile parametre tiplerinin listesidir; dönüş tipi imzaya dahil değildir.
- `main` `static`'tir, çünkü JVM onu hiçbir nesne yokken çağırır. `static` bir metot, `static` olmayan bir metodu doğrudan çağıramaz.
- `void` metot değer döndürmez; işi bir **yan etkidir**. `void` çağrı bir ifadenin parçası olamaz. **Erken dönüş** özel durumları metodun başında eler.
- `void` olmayan bir metotta her yol bir `return` ile bitmelidir. Döndürülen değer dönüş tipine sığmalıdır; daraltma kendiliğinden yapılmaz.
- Java argümanları her zaman **değer ile aktarır**. Metot parametresini değiştirse de çağıranın değişkeni değişmez; bu yüzden ilkel değişkenleri takas eden bir `swap` metodu yazılamaz. Yeni değer döndürülür ve çağıran onu atar.
- **Aşırı yüklemede** aynı adlı metotların parametre listeleri farklıdır. Derleyici seçimi derleme sırasında, önce tam eşleşmeye sonra en yakın genişletmeye bakarak yapar. Yalnızca dönüş tipi farklı iki metot ve eşit derecede uyan iki aday derleme hatasıdır.
- **Değişken sayıda argüman** (`int... s`) metodun içinde bir dizidir. Parametre listesinin en sonunda bulunur; argümansız çağrıda boş bir dizi geçer.
- **Kapsam**, bir değişkenin kullanılabildiği bölgedir ve küme parantezleriyle sınırlanır. Aynı kapsamda aynı adlı iki yerel değişken bildirilemez. Alanla aynı adlı yerel değişken alanı **gölgeler**. Yerel değişken bir ilk değer almadan okunamaz.
- `Math` metotları `static`'tir ve aşırı yüklenmiştir. `sqrt` ve `pow` `double`, `round(double)` `long` döndürür.

- İyi bir metot tek bir görev yapar, anlamlı bir ad taşır, az parametre alır, sonucunu döndürür ve bağımsız örneklerle denenebilir. **Soyutlama** ne yapıldığını nasıl yapıldığından ayırır; **ayrıştırma** büyük görevi alt metotlara böler; **sözleşme** ön ve son koşulları belirler.
- Metot başka metodu çağırabilir (`ekok` → `ebob`, `enBuyuk3` → `enBuyuk`); her çağrı yığında kendi çerçevesini açar. **Saf metot** aynı girdiye hep aynı sonucu verir ve yan etkisi yoktur; bu yüzden kolay test edilir ve her yerde yeniden kullanılır.

9.14 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

metot *method* · çağrı *method call* · parametre *parameter* · biçimsel parametre *formal parameter* · argüman *argument* · gerçek argüman *actual argument* · dönüş tipi *return type* · dönüş değeri *return value* · döndürmek *return* · imza *method signature* · static metot *static method* · main metodu *main method* · komut satırı argümanı *command-line argument* · yığın *stack* · çerçeve *stack frame* · çağrı yığını *call stack* · kod izleme *code tracing* · erken dönüş *early return* · yan etki *side effect* · değer ile aktarım *pass by value* · referans *reference* · aşırı yükleme *method overloading* · genişletme *widening conversion* · daraltma *narrowing conversion* · değişken sayıda argüman *varargs* · gelişmiş for *enhanced for loop* · kapsam *scope* · yerel değişken *local variable* · alan *field* · gölgeleme *shadowing* · derleme hatası *compile-time error* · soyutlama *abstraction* · ayrıştırma *decomposition* · sözleşme *contract* · yeniden kullanım *reuse* · saf metot *pure method* · tam sayı bölmesi *integer division* · taşma *overflow* · etkisiz eleman *identity element* · en büyük ortak bölen (EBOB) *greatest common divisor* · EKOK *least common multiple*

9.15 Kendini Dene

1. `static double daireAlani(double r) { return Math.PI * r * r; }` metodu `main`'de `double s = daireAlani(2 * yaricap);` diye çağrılıyor. Bu satırlarda parametre, argüman, dönüş tipi ve imza nedir?

   Cevap

   Biçimsel parametre `r`'dir; başlıkta tipiyle (`double r`) bildirilen değişkendir. Gerçek argüman `2 * yaricap` ifadesidir; önce değeri hesaplanır ve bu değer r'ye kopyalanır. Dönüş tipi `double`'dır. İmza `daireAlani(double)`'dır; dönüş tipi ve parametre adı imzaya girmez.

2. Üç metot şöyle tanımlanmış: `a()` önce “a1 ” yazar, sonra `b()`'yi çağırır, sonra “a2 ” yazar. `b()` “b1 ” yazar, `c()`'yi çağırır, “b2 ” yazar. `c()` yalnızca “c ” yazar. `main`, `a()`'yı çağırıyor. Çıktı nedir? Yığında en çok kaç çerçeve bulunur?

   Cevap

   Çıktı `a1 b1 c b2 a2`'dir. Her metot, çağırdığı metot bitene kadar bekler ve sonra kaldığı yerden sürer. `c` çalışırken yığında `main`, `a`, `b` ve `c` olmak üzere 4 çerçeve vardır. Çerçeveler açıldıkları sıranın tersiyle kapanır: önce `c`, sonra `b`, en son `a`.

3. `static void sifirla(int n) { n = 0; }` metodu için `int k = 9; sifirla(k); System.out.println(k);` ne yazar? k'yi gerçekten sıfırlamak için metot ve çağrı nasıl değiştirilir?

   Cevap

   9 yazar. Java değer ile aktarır: n, k'nin kopyasıdır ve `n = 0` yalnızca kopyayı değiştirir. Metot bitince n silinir. Metot bir değer döndürecek biçimde yazılır, `static int sifirla() { return 0; }`, ve çağıran atamayı kendisi yapar: `k = sifirla();`. Bu örnekte en sade çözüm ise metot kullanmadan `k = 0;` yazmaktır.

4. Bir sınıfta yalnızca `static void g(long x)` ve `static void g(double x)` var; ikisi de parametresinin tipini ve değerini yazıyor. `g(5)`, `g(5.5f)` ve `g('x')` çağrıları hangi metodu çalıştırır?

   Cevap

   Hiçbir çağrı için tam eşleşme yoktur; derleyici en yakın genişletmeyi seçer. `g(5)`: `int` → `long`, `double`'dan daha yakındır; `long 5` yazılır. `g(5.5f)`: `float`, `long`'a daraltılamaz ama `double`'a genişler; `double 5.5` yazılır. `g('x')`: `char` → `int` → `long` genişlemesiyle `long` sürüm seçilir; 'x' karakterinin kodu 120 olduğu için `long 120` yazılır.

5. `static int say(int... s) { return s.length; }` için `say()`, `say(5)`, `say(new int[]{1, 2, 3})` ve `say(1, 2)` ne döndürür? `static int say(int... s, int k)` başlığı neden derlenmez?

   Cevap

   Sırasıyla 0, 1, 3 ve 2. Argümansız çağrıda boş bir dizi geçer (uzunluğu 0); dizi doğrudan verilirse o dizi kullanılır. İkinci başlıkta varargs parametresi listenin sonunda değildir; derleyici `varargs parameter must be the last parameter` hatası verir, çünkü çağrıdaki değerlerden hangisinin k'ye gideceğini bilemez.

6. Aşağıdaki metotta hangi satır derleme hatası verir, neden? `static int toplamKare(int n) { for`

`(int i = 1; i <= n; i++) { int kare = i * i; } return kare; }`

Cevap

`return kare;` satırı `cannot find symbol` hatası verir. `kare`, `for` döngüsünün gövde bloğunda bildirilmiştir ve blok kapanınca kapsamı biter; döngünün dışında bu adla bir değişken yoktur. Ayrıca metot amacına da ulaşmıyor: kareler biriktirilmiyor. Doğrusu, döngüden önce `int toplam = 0;` bildirmek, gövdede `toplam += i * i;` yazmak ve `return toplam;` ile bitirmektir.

7. `ebob(105, 30)` Öklid algoritmasıyla kaç turda biter, ne döndürür? Aynı sayılar için `ekok` metodu ne döndürür?

Cevap

1. tur: *105 % 30 = 15* (105 = 3 × 30 + 15); a = 30, b = 15. 2. tur: *30 % 15 = 0*; a = 15, b = 0. Koşul yanlış olur ve 2 turda 15 döner. *ekok = 105 / 15 × 30 = 7 × 30 = 210*. Denetim: *210 = 2 × 105 = 7 × 30*.

8. `asalMi(91)` çağrısında döngü kaç tur döner ve ne döner? `asalMi(37)` kaç tur döner?

Cevap

91 = 7 × 13'tür. i = 2, 3, 4, 5, 6 için koşul (*i ≤ 91 / i*) doğrudur ve kalan 0 değildir. i = 7'de *7 ≤ 91 / 7 = 13* doğrudur ve *91 % 7 = 0*'dır: 6. turda `false` döner. 37 için i = 2…6 denenir (5 tur); i = 7'de *7 ≤ 37 / 7 = 5* yanlıştır ve `true` döner.

9. `Math.round(3.5)`, `Math.round(-3.5)`, `Math.abs(-0.5)` ve `Math.max(-1, -8)` ne döndürür? `int r = Math.round(3.5);` neden derlenmez?

Cevap

4, −3, 0.5 ve −1. Tam yarıdaki sayılar büyük olana yuvarlanır: 3,5 → 4, −3,5 → −3. `Math.abs` `double` argümanla `double` döndürür. −1, −8'den büyüktür. `Math.round(double)` `long` döndürdüğü için sonuç `int`'e doğrudan atanamaz (`possible lossy conversion from long to int`); `int r = (int) Math.round(3.5);` yazılmalıdır.

10. `static int ucKati(int x) { return 3 * x; }` için `ucKati(ucKati(2))` ve `ucKati(2) * ucKati(1)` ne verir? Aynı metot sonucu döndürmek yerine `System.out.println(3 * x)` ile yaz-

saydı bu iki ifade yazılabilir miydi?

Cevap

İlki: içteki çağrı 6 verir, dıştaki *ucKati(6) = 18*. İkincisi: *6 × 3 = 18*. İkisi de 18'dir. Metot yazdıran bir `void` metot olsaydı iki ifade de derlenmezdi: `void` bir çağrı ne başka bir çağrıya argüman ne de bir çarpmanın işleneni olabilir. Sonucu döndüren saf metot her türlü ifadede kullanılabilir.

9.16 İleri okuma

- Oracle, *The Java Tutorials*: "Defining Methods" ve "Passing Information to a Method or a Constructor" sayfaları; metot bildirimi, aşırı yükleme, varargs ve değer ile aktarım kısa örneklerle anlatılır.
- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), Kesim 2.1 "Static Methods": `static` metotlar, kapsam ve metotlarla modüler programlama, bilimsel hesaplardan örneklerle işlenir.
- A. B. Downey, C. Mayfield, *Think Java* (2. baskı, O'Reilly, 2019): metotlar, parametreler ve dönüş değerleri programlamaya yeni başlayanlar için yığın diyagramlarıyla anlatılır; kitabın çevrim içi sürümü ücretsizdir.
- Oracle, *Java Language Specification, Java SE 21*, §8.4 "Method Declarations" ve §15.12 "Method Invocation Expressions": imza, dönüş ve aşırı yükleme seçimini derleyicinin nasıl yaptığının kesin kuralları.

9.17 Kaynaklar

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11. baskı, Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13. baskı, Pearson, 2024.
3. Oracle, *The Java Tutorials: Defining Methods; Passing Information to a Method or a Constructor*.
4. Oracle, *Java Language Specification, Java SE 21 Edition*, Bölüm 8: Classes (§8.4 Method Declarations).
5. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 2.1 Static Methods.
6. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019.

KISIM III · BÖLÜM 10

Diziler

Bir sınıfın 40 öğrencisinin notlarını saklamak için 40 ayrı değişken açmak hem yorucu hem de hataya açıktır. Not sayısı değiştiğinde programın da baştan yazılması gerekir. Dizi bu sorunu çözer: aynı tipteki çok sayıda değeri tek bir ad altında, numaralı hücrelerde tutar. Bu not dizinin bellekteki yapısıyla başlar. Ardından dizi bildirmeyi, öğelere indisle erişmeyi ve diziyi döngülerle gezmeyi anlatır. Toplam, ortalama, en büyük ve en küçük değer, ters çevirme, frekans sayma ve Eratosthenes eleği gibi temel işlemler adım adım izlenir. Kopyalama, `Arrays` sınıfı, dizilerin metotlara aktarılması, arama ve sıralamanın Java'daki yazımı bunları izler. Son kesimler iki boyutlu dizileri, matris işlemlerini ve dizilerle ilgili yaygın hataları ele alır. Notta gösterilen bütün programlar JDK 21 ile derlenip çalıştırılmıştır; konsol çıktıları ve hata iletileri bu çalıştırmalardan birebir alınmıştır.

Bu bölümün kodu: `github.com/sercankulcu/computer-programming-java/Bolum10/src`

ÖĞRENME ÇIKTILARI

Oluşturma
Dizi bildirmek, `new` ve ilklendiriciyle dizi oluşturmak; dizi değişkeninin bir referans tuttuğunu ve hücrelerin varsayılan değer aldığını açıklamak.

Erişim
Öğelere indisle erişmek, `length` alanını doğru kullanmak; sınır dışı hatasının nedenini ve iletisini tanımak.

İşleme
Diziyi klasik ve gelişmiş `for` ile gezmek; toplam, ortalama, en büyük–en küçük, ters çevirme, sayma, kopyalama, arama ve sıralama işlemlerini Java'da yazmak ve maliyetlerini belirtmek.

Matris
İki boyutlu dizilerle satır–sütun işlemleri yapmak; düzensiz dizileri ve sığ kopya tuzağını açıklamak; transpoz ve matris çarpımını kodlamak.

10.1 Dizi kavramı

Şimdiye kadarki programlarda her değer kendi değişkeninde duruyordu: `int not1`, `int not2`, `int not3`. Üç not için bu yöntem işe yarar. Kırk not için kırk değişken gerekir; bunların ortalamasını bulmak için kırk terimli bir toplama yazılır. Not sayısı kırk bire çıkarsa program değiştirilmek zorundadır. Sorunun kaynağı, değişkenlerin birbirinden bağımsız olmasıdır. Program "üçüncü not" ya da "sıradaki not" diye bir şey söyleyemez, çünkü değişken adları sayı değil, metindir.

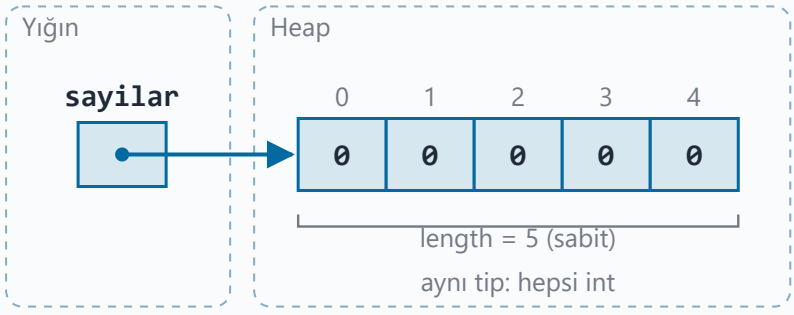
10.1.1 Dizi nedir?

Tanım 10.1

Dizi (*array*), aynı tipten sabit sayıda değeri tek bir nesnede, art arda dizilmiş **hücrelerde** (*cell*) saklayan veri yapısıdır. Hücrelerdeki değerlere **öğe** (*element*) denir. Her öğeye, hücrenin sıra numarası olan **indisle** (*index*) erişilir. Java'da indisler 0'dan başlar. Hücre sayısına dizinin **uzunluğu** (*length*) denir; uzunluk dizi oluşturulurken belirlenir ve sonradan değişmez.

Tanımdaki üç koşulun her birinin bir nedeni vardır. **Aynı tip** koşulu, her hücrenin aynı büyüklükte olmasını sağlar: bir `int` dizisinin her hücresi 4 bayt tutar. **Sabit sayı** koşulu, dizinin bellekte tek parça hâlinde yer almasına izin verir. **Art arda hücreler** ise herhangi bir öğenin yerinin tek bir hesapla bulunmasını mümkün kılar.

Java'da bir dizi bir **nesnedir** (*object*). Nesneler, belleğin **heap** adı verilen bölgesinde oluşturulur. Metotların yerel değişkenleri ise **yığın** (*stack*) adı verilen ayrı bir bölgede tutulur. Dizi değişkeni yığında durur, ama içinde öğeler yoktur. Değişken, heap'teki dizi nesnesinin **referansını** (*reference*), yani nesnenin bellekteki yerini gösteren bir değeri saklar. Şekillerde referans bir okla çizilir.

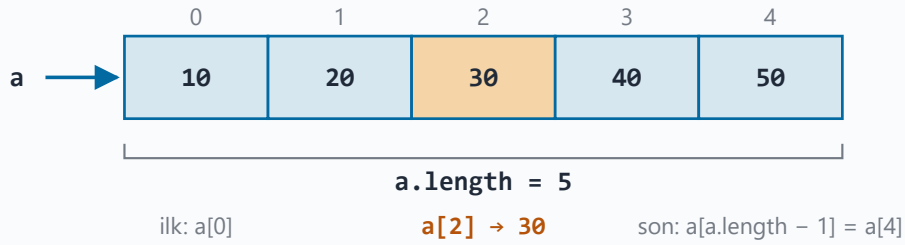


Şekil 10.1: `int[] sayilar = new int[5];` sonrasında bellek. Yığındaki `sayilar` değişkeni, heap'teki beş hücreli dizi nesnesini gösterir. Hücreler 0'dan 4'e kadar numaralıdır, hepsi aynı tiptedir (`int`) ve başlangıçta 0 değerini taşır. Uzunluk 5'tir ve değişmez.

Şekil 10.1 bu ayrımı gösterir. Soldaki kutu değişkenin kendisidir; içinde yalnızca bir ok, yani referans vardır. Sağdaki beş hücre dizi nesnesidir. Bu ayrım ilk bakışta gereksiz bir ayrıntı gibi görünür. Oysa dizilerle ilgili en şaşırtıcı davranışların çoğu buradan çıkar. İki değişken aynı diziyi gösterebilir; bir metoda dizi verildiğinde metot asıl diziyi değiştirebilir; `==` işleci iki dizinin içeriğini değil, referanslarını karşılaştırır. Bu davranışların her biri ilerleyen kesimlerde örnekle işlenir.

10.1.2 İndis ve adres hesabı

Dizinin öğelerine köşeli parantez içinde indis yazılarak erişilir. `a[0]` ilk öğe, `a[1]` ikinci öğedir. Beş öğeli bir dizide son öğe `a[4]` 'tür. Genel kural şudur: uzunluğu `n` olan bir dizinin ilk indisi 0, son indisi `n - 1` 'dir. Java'da dizinin uzunluğu `a.length` ile okunur; bu yüzden son öğe her zaman `a[a.length - 1]` diye yazılabilir.



Şekil 10.2: Beş öğeli `a` dizisi: 10, 20, 30, 40, 50. İndisler 0'dan 4'e gider ve `a.length` 5'tir. İlk öğe `a[0]`, son öğe `a[a.length - 1]`, yani `a[4]` 'tür. Vurgulu hücre `a[2]` 'dir ve değeri 30'dur.

İndisin 0'dan başlaması yeni başlayanlara garip gelir. Gündelik hayatta sayma 1'den başlar. Java'da, C'de ve birçok başka dilde ise indis aslında bir **uzaklıktır**: dizinin başından kaç hücre ileri gidileceğini söyler. İlk öğe başlangıçtadır, uzaklığı sıfırdır. `a[2]` ise başlangıçtan iki hücre ötededir.

Bu yorum, erişimin neden hızlı olduğunu da açıklar. Dizi bellekte art arda hücrelerden oluştuğu için `i` numaralı hücrenin adresi tek bir formülle hesaplanır:

> adres(a[i]) = başlangıç adresi + i × hücre boyu

Örnek 10.1

Bir `int` dizisinin heap'teki başlangıç adresi 1000 olsun. `int` 4 bayt tuttuğu için hücre boyu 4'tür. `a[3]` 'ün adresi nedir?

Formülde `i` = 3 ve hücre boyu 4'tür: 1000 + 3 × 4 = 1000 + 12 = 1012. İşlemci bu adrese doğrudan gider ve değeri okur. `a[0]`, `a[1]` ve `a[2]` hücrelerine hiç bakılmaz. Aynı hesap `a[999]` için de tek çarpma ve tek toplamadır: 1000 + 999 × 4 = 4996.

Hesabın uzunluğu dizinin boyuna bağlı değildir. Beş öğeli dizide de bir milyon öğeli dizide de tek bir çarpma ve tek bir toplama yapılır. Bu yüzden indisle erişimin maliyeti **sabit zamandır**, büyük O gösterimiyle **O(1)**. Büyük O gösterimi bir işlemin süresinin girdi büyüdükçe nasıl arttığını özetler. O(1) "girdi ne kadar büyük olursa olsun aynı süre", O(n) ise "süre öğe sayısıyla doğru orantılı artar" demektir. Gerçek Java programında başlangıç adresi programcıya gösterilmez ve JVM her erişimde ayrıca indisin geçerli olup olmadığını denetler. Ama hesap ilkesi aynıdır.

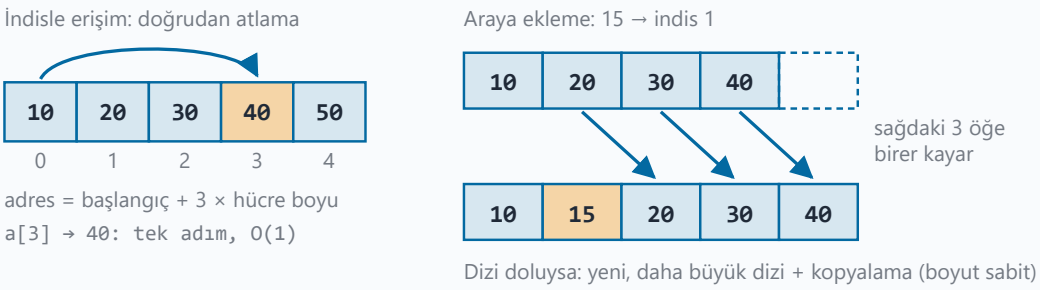
10.1.3 Dizinin güçlü ve zayıf yanları

Dizinin yapısı ona bazı üstünlükler kazandırır, bazı sınırlar da koyar. İkisi aynı kaynaktan gelir: sabit boyutlu, art arda hücreler.

Avantaj	Sınır
İndisle hızlı erişim	Boyut sabittir
Öğe tipi tutarlıdır	Araya ekleme maliyetlidir
Döngüyle kolay işlenir	Sınır aşımı çalışma zamanı hatasıdır
Bellekte düzenli yapı	Tek bir öğe tipi tutar

Tablo 10.1: Dizinin avantajları ve sınırları.

Tablodaki "öğe tipi tutarlıdır" ile "tek bir öğe tipi tutar" aynı özelliğin iki yüzüdür. Derleyici, bir `int` dizisine yanlışlıkla metin konmasını engeller; bu bir güvencedir. Buna karşılık bir öğrencinin adını, numarasını ve notunu aynı dizide tutmak mümkün değildir; bunun için üç ayrı dizi ya da başka bir yapı gerekir.



Şekil 10.3: Solda indisle erişim: `a[3]` 'e adres hesabıyla tek adımda gidilir. Sağda araya ekleme: 15'i 1 numaralı konuma koymak için 20, 30 ve 40 birer hücre sağa kaydırılır. Dizi doluysa daha büyük yeni bir dizi oluşturulup bütün öğeler kopyalanmalıdır.

Şekil 10.3'ün sağ yarısı araya eklemenin neden pahalı olduğunu gösterir. `{10, 20, 30, 40}` dizisinde 10 ile 20 arasına 15 konacak olsun. Hücreler art arda olduğu için "araya hücre sokmak" diye bir işlem yoktur. 40 bir sağa, sonra 30 bir sağa, sonra 20 bir sağa taşınır ve boşalan 1 numaralı hücreye 15 yazılır. Kaydırılan öğe sayısı, ekleme noktasından sonraki öğe sayısı kadardır. En kötü durumda,

başa eklemede, bütün öğeler kayar. Bu yüzden araya ekleme ve aradan silme O(n) maliyetlidir. Üstelik dizinin sonunda boş hücre yoksa ekleme hiç yapılamaz: uzunluk sabittir. Daha büyük yeni bir dizi oluşturulur ve eski öğeler oraya kopyalanır.

10.2 Dizi bildirme ve oluşturma

Bir diziyi kullanmak iki ayrı iş gerektirir. Önce dizi değişkeni bildirilir; bu, yığında bir referans kutusu açar. Sonra dizi nesnesi oluşturulur ve referansı bu kutuya yazılır. İki iş tek satırda da yapılabilir, ayrı satırlarda da.

10.2.1 Bildirim ve new

Java'da bir dizi tipi, öğe tipinin ardına köşeli parantez çifti eklenerek yazılır: `int[]` "int dizisi", `String[]` "String dizisi", `double[]` "double dizisi" demektir. Dizi nesnesi `new` işleciyle oluşturulur ve uzunluk köşeli parantez içinde verilir. Aşağıdaki program dört ayrı bildirimi bir arada gösterir:

```
public class Bildirim {
    public static void main(String[] args) {
        int[] sayilar = new int[5];
        String[] adlar = {"Ada", "Can", "Ece"};
        double[] olcumler;
        olcumler = new double[10];
        System.out.println(sayilar.length + " " + adlar.length + " " + olcumler.length);
    }
}
```

```
5 3 10
```

Satırları sırayla inceleyelim.

1. `int[] sayilar = new int[5];` bildirim ile oluşturmayı tek satırda yapar. `new int[5]` heap'te beş hücreli bir `int` dizisi oluşturur. Atama, bu dizinin referansını `sayilar` değişkenine yazar.
2. `String[] adlar = {"Ada", "Can", "Ece"};` diziyi değerleriyle birlikte oluşturur. Uzunluk yazılmaz; derleyici değerleri sayar ve uzunluğu 3 yapar.
3. `double[] olcumler;` yalnızca değişkeni bildirir. Bu noktada heap'te hiçbir dizi yoktur.
4. `olcumler = new double[10];` on hücreli diziyi sonradan oluşturur ve referansını değişkene yazar.

Uzunluk bir sabit olmak zorunda değildir; çalışma sırasında hesaplanan bir `int` de olabilir. `new int[n]`, `n`'nin o anki değeri kadar hücre açar. Uzunluk 0 olabilir; böyle bir diziye **boş dizi** denir ve hiç hücresi yoktur. Negatif uzunluk ise program çalışırken `NegativeArraySizeException` hatasına

yol açar. Dizi bir kez oluşturulduktan sonra uzunluğu değiştirilemez. `olcumler`'e daha sonra `new double[20]` atanırsa eski dizi büyümez; değişken yeni, ayrı bir diziyi göstermeye başlar.

Sık yapılan hata

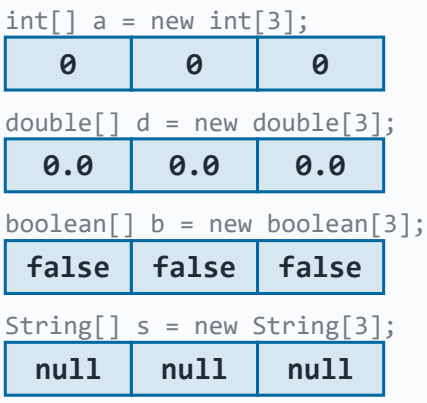
Değişkeni bildirip diziyi oluşturmadan kullanmak. `int[] sayilar;` satırından sonra `sayilar[0] = 7;` yazılırsa ortada hücre yoktur. Java bu durumu derleme sırasında yakalar:

```
Atanmamis.java:4: error: variable sayilar might not have been initialized
        sayilar[0] = 7;
        ^
1 error
```

İleti "sayilar değişkenine değer verilmemiş olabilir" der. Çözüm, kullanmadan önce `sayilar = new int[4];` gibi bir atama yapmaktır.

10.2.2 Varsayılan değerler

`new` ile oluşturulan bir dizinin hücreleri boş kalmaz. Java her hücreye öğe tipinin **varsayılan değerini** (*default value*) yazar. Sayı tiplerinde bu değer sıfırdır, `boolean`'da `false`'tur. Nesne tiplerinde, örneğin `String`'de, varsayılan değer `null`'dır. `null`, "bu referans hiçbir nesneyi göstermiyor" anlamına gelen özel değerdir.



Şekil 10.4: `new` ile oluşturulan üç hücreli dizilerin başlangıç değerleri: `int` için 0, `double` için 0.0, `boolean` için `false`, `String` için `null`.

Aşağıdaki program Şekil 10.4'teki değerleri konsola yazar. `char` dizisinin varsayılan değeri de eklenmiştir:

```
public class Varsayilan {
    public static void main(String[] args) {
        int[] a = new int[3];
        double[] d = new double[2];
        boolean[] b = new boolean[2];
        String[] s = new String[2];
        char[] c = new char[2];
        System.out.println(a[1] + " " + d[0] + " " + b[1] + " " + s[0]);
        System.out.println((int) c[0]);
        System.out.println(a.length + d.length + s.length);
    }
}
```

```
0 0.0 false null
0
7
```

İlk satırda dört farklı dizinin birer hücresi yazdırılır: 0, 0.0, `false` ve `null`. `char` dizisinin varsayılan değeri kodu 0 olan karakterdir. Bu karakter ekranda görünmediği için `(int)` dönüşümüyle sayı olarak yazdırılmıştır. Son satırda bir tuzak vardır. `a.length + d.length + s.length` ifadesinde hiç dizgi yoktur. Bu yüzden `+` toplama yapar ve sonuç 3 + 2 + 2 = 7 olur.

Varsayılan değer kuralı yalnızca dizi hücreleri ve sınıf alanları için geçerlidir. Bir metodun içinde bildirilen yerel değişkenlere Java kendiliğinden değer vermez. `int x; System.out.println(x);` derlenmez. Buna karşılık `new int[1]` ile oluşturulan dizinin hücresi her zaman 0 ile başlar.

10.2.3 İlklendirici

Öğelerin değerleri baştan biliniyorsa dizi, süslü parantez içinde değerler yazılarak oluşturulur. Bu yazıma **ilklendirici** (*array initializer*) denir: `int[] puanlar = {70, 85, 90};`. Uzunluk yazılmaz, çünkü derleyici değerleri sayar. Burada uzunluk 3'tür.

İlklendiricinin kısa biçimi yalnızca bildirim satırında kullanılabilir. Önce bildirilmiş bir değişkene sonradan bu biçimle atama yapılamaz:

```
public class IlkHata {
    public static void main(String[] args) {
        int[] a;
        a = {1, 2, 3};
    }
}
```

```
IlkHata.java:4: error: illegal start of expression
        a = {1, 2, 3};
            ^
IlkHata.java:4: error: not a statement
        a = {1, 2, 3};
             ^
IlkHata.java:4: error: ';' expected
        a = {1, 2, 3};
              ^
3 errors
```

Derleyici tek bir yanlış için üç ileti verir. Bu tür ileti yağmurunda ilk ileti esas alınır: "ifadenin geçersiz başlangıcı". Doğru yazım, ilklendiricinin önüne dizi tipini eklemektir: `a = new int[]{1, 2, 3};`. Bu uzun biçim, bir metoda doğrudan dizi vermek için de kullanılır: `ara(new int[]{2, 4, 6, 8}, 5)`.

Java dizi değişkeninin iki yazımını da kabul eder: `int[] a` ve `int a[]`. İkincisi C dilinden kalmadır. Bu notta ve sunumda birinci biçim kullanılır, çünkü tip bilgisi (`int[]`) tek parçada durur.

10.2.4 Dizi oluşturma adım adım

Aşağıdaki program bir diziyi önce bildirir, sonra oluşturur, iki hücresine değer yazar ve iki dizinin uzunluğunu yazdırır:

```
public class Olustur {
    public static void main(String[] args) {
        int[] sayilar;
        sayilar = new int[4];
        sayilar[0] = 7;
        sayilar[3] = 2;
        String[] adlar = {"Ada", "Can", "Ece"};
        System.out.println(sayilar.length + " "
                           + adlar.length);
    }
}
```

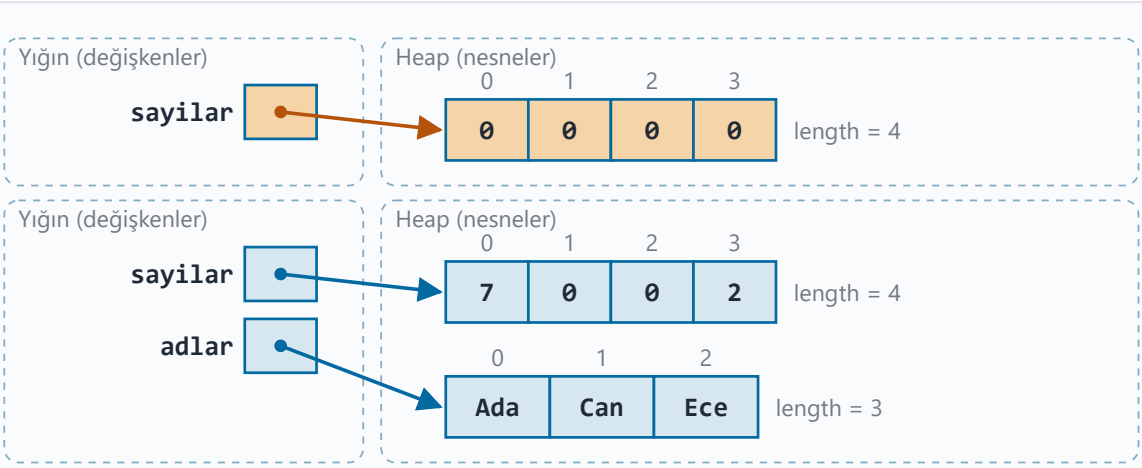
```
4 3
```

Programın her adımında yığın ve heap'te ne olduğunu Tablo 10.2 gösterir.

Satır	Yığın	Heap	Açıklama
`int[] sayilar;`	`sayilar` = ?	—	Yalnızca referans değişkeni açılır
`sayilar = new int[4];`	`sayilar` → dizi 1	dizi 1: [0, 0, 0, 0]	Dört hücre, hepsi 0
`sayilar[0] = 7;`	aynı	dizi 1: [7, 0, 0, 0]	Ok izlenir, 0. hücreye 7 yazılır

Satır	Yığın	Heap	Açıklama
`sayilar[3] = 2;`	aynı	dizi 1: [7, 0, 0, 2]	Son hücre: indis `length - 1` = 3
`String[] adlar = {…};`	`adlar` → dizi 2	dizi 2: [Ada, Can, Ece]	Üç değer, uzunluk 3
`println(…)`	aynı	aynı	Çıktı: `4 3`

Tablo 10.2: `Olustur` programının izleme tablosu. "?" henüz değer verilmemiş değişkeni gösterir.



Şekil 10.5: İzlemenin iki anı. Üstte `new int[4]` sonrası: heap'te dört sıfırlı hücre açılmış, `sayilar` değişkenine referans yazılmıştır. Altta programın sonu: `sayilar` [7, 0, 0, 2] dizisini, `adlar` da üç dizgili diziyi gösterir.

Tablonun ilk satırı dikkat ister: `int[] sayilar;` çalıştığında heap'te hiçbir şey olmaz. Şekil 10.5 bu boşluğu ve ardından gelen değişimi gösterir. İkinci satırdaki `new` heap'te nesneyi açar, atama işleci de referansı değişkene yazar. Bundan sonra `sayilar[0] = 7` gibi her erişim aynı yolu izler: değişkendeki ok takip edilir, heap'teki diziye gidilir, indisle hücre bulunur ve değer yazılır.

Son satırda `sayilar.length` ve `adlar.length` okunur. `length` bir metot değil, dizinin bir **alanıdır**; bu yüzden parantezsiz yazılır. `String` sınıfında ise uzunluk `length()` metoduyla alınır. Bu fark sık karıştırılır: dizide `a.length()`, dizgide `s.length` yazmak derleme hatasıdır. `length` alanı yalnızca okunur; `sayilar.length = 10;` da derlenmez.

10.3 Öğelere erişim

10.3.1 Okuma ve yazma

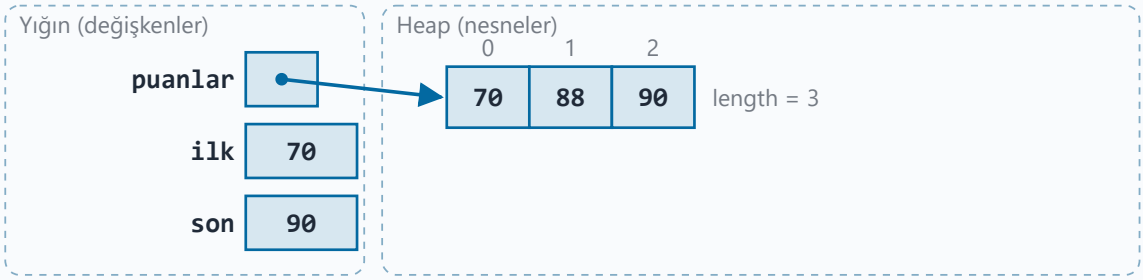
Bir dizi öğesi, sıradan bir değişken gibi kullanılır. `puanlar[1]` bir ifadenin içinde geçerse hücredeki değer okunur. Atamanın solunda geçerse hücrenin kendisi kastedilir ve oraya yeni değer yazılır. Köşeli parantezin içine sabit bir sayı yazmak şart değildir; `int` değer veren her ifade indis olabilir: `a[i]`, `a[i + 1]`, `a[a.length - 1]`.

```java
public class Okuma {
    public static void main(String[] args) {
        int[] puanlar = {70, 85, 90};
        int ilk = puanlar[0];
        puanlar[1] = 88;
        int son = puanlar[puanlar.length - 1];
        System.out.println(ilk + " " + son);
    }
}
```

```
70 90
```

Satır	`puanlar`	`ilk`	`son`	Açıklama
1	[70, 85, 90]	—	—	Üç öğe; indisler 0, 1, 2
2	[70, 85, 90]	70	—	Okuma: `puanlar[0]` → 70
3	[70, 88, 90]	70	—	Yazma: 85 silinir, 88 yazılır
4	[70, 88, 90]	70	90	`puanlar.length - 1` = 3 − 1 = 2
5	[70, 88, 90]	70	90	Çıktı: `70 90`

Tablo 10.3: `Okuma` programının izleme tablosu.



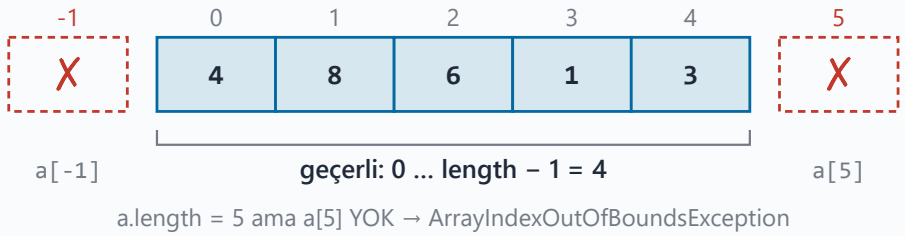
Şekil 10.6: `Okuma` programının sonunda bellek. `ilk` ve `son` sıradan `int` değişkenlerdir; değerleri diziden kopyalanmıştır. Dizinin 1 numaralı hücresinde artık 88 vardır.

4. satırdaki `puanlar[puanlar.length - 1]` yazımı, son öğeyi okumanın genel yoludur. Önce iç ifade hesaplanır: 3 − 1 = 2. Sonra `puanlar[2]` okunur. Dizi beş öğeli olsaydı aynı satır hiç değişmeden `puanlar[4]`'ü okurdu. Son indisi elle yazmak (`puanlar[2]`) ise dizi büyüdüğünde sessizce yanlış öğeyi verir.

Şekil 10.6 önemli bir ayrıntıyı da gösterir: `int ilk = puanlar[0];` diziden değeri kopyalar. Bu satırdan sonra `puanlar[0]` değişse bile `ilk` 70 kalır. Temel tiplerde (`int`, `double`, `boolean`, `char`) atama her zaman değeri kopyalar.

10.3.2 Sınır dışı hatası

Geçerli indisler 0 ile `length - 1` arasındaki tam sayılardır. Negatif bir indis ya da `length`'in kendisi geçersizdir. Geçersiz indisle erişim, Java'da programı `ArrayIndexOutOfBoundsException` ile durdurur. Bu bir **istisnadır** (*exception*): program çalışırken ortaya çıkan ve olağan akışı kesen bir hata durumu.



Şekil 10.7: Beş öğeli dizide geçerli indisler 0…4'tür. Dizinin solundaki `a[-1]` ve sağındaki `a[5]` hücreleri yoktur. `a.length` 5'tir ama `a[5]` diye bir hücre bulunmaz.

Tanım 10.2

Uzunluğu `n` olan bir dizide $0 \le i \le n - 1$ koşulunu sağlamayan bir `i` indisiyle erişim, **sınır dışı** (*out of bounds*) erişimdir. Java her dizi erişiminde bu koşulu denetler; koşul sağlanmazsa `ArrayIndexOutOfBoundsException` fırlatılır.

Derleyici bu hatayı genellikle yakalayamaz, çünkü indis çoğu zaman program çalışırken hesaplanır. Aşağıdaki program kusursuz derlenir:

```
public class Negatif {
    public static void main(String[] args) {
        int[] a = {4, 8, 6};
        System.out.println(a[-1]);
    }
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index -1 out of
bounds for length 3
        at Negatif.main(Negatif.java:4)
```

İleti üç bilgi verir. Hatanın türü `ArrayIndexOutOfBoundsException`'dır. Ayrıntı kısmı “Index -1 out of bounds for length 3” der, yani “indis −1, uzunluğu 3 olan dizinin sınırları dışında”. Son satır da hatanın yerini gösterir: `Negatif.java` dosyasının 4. satırı. Bu denetim C gibi dillerde yoktur; orada sınır dışı

yazma, belleğin başka bir değişkene ait bölümünü sessizce bozabilir. Java'nın denetimi bir maliyet getirir, ama hatayı tam yerinde yakalar.

10.3.3 Bir tur fazla: adım adım

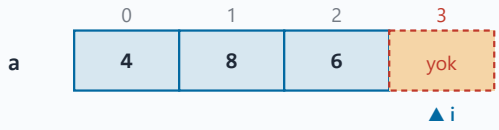
Sınır dışı hatasının en yaygın kaynağı, döngü koşulunda `<` yerine `<=` yazmaktır. Hataya **bir tur fazla** (ya da bir eksik) hatası denir:

```
public class Sinir {
    public static void main(String[] args) {
        int[] a = {4, 8, 6};
        for (int i = 0; i <= a.length; i++) {
            System.out.println(a[i]);
        }
    }
}
```

```
> java Sinir
4
8
6
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out of bounds
for length 3
        at Sinir.main(Sinir.java:5)
```

Tur	`i`	`i <= a.length`	Yapılan	Çıktı
1	0	0 <= 3 → true	`a[0]` yazdırılır	4
2	1	1 <= 3 → true	`a[1]` yazdırılır	8
3	2	2 <= 3 → true	`a[2]` yazdırılır	6
4	3	3 <= 3 → true	`a[3]` : hücre yok	istisna

Tablo 10.4: `Sinir` programının izleme tablosu (`a.length` = 3).



Şekil 10.8: Dördüncü turda `i` = 3 olur. Koşul `3 <= 3` doğru olduğu için gövdeye girilir, ama 3 numaralı hücre yoktur.

İlk üç tur sorunsuzdur ve 4, 8, 6 yazılır. Dördüncü turda `i` 3 olur. Koşul `3 <= 3` doğrudur, gövdeye girilir ve `a[3]` okunmaya çalışılır (Şekil 10.8). Program bu noktada durur. Dikkat edilirse hata, önceki çıktıları silmez: 4, 8 ve 6 konsolda kalır. Bu yüzden hatalı bir program "neredeyse doğru" görünebilir.

Çözüm, koşulu `i < a.length` yapmaktır. O zaman `i` 3 olduğunda `3 < 3` yanlıştır ve döngü gövdeye girmeden biter. Akılda tutulacak kural kısadır: indis 0'dan başlıyorsa koşul `< length` olur. `<= length - 1` da doğrudur, ama daha uzun ve hataya açıktır.

10.4 Döngülerle gezinme

Dizinin asıl gücü döngüyle birleştiğinde ortaya çıkar. İndis bir değişken olabildiği için tek bir döngü, dizinin bütün öğelerine sırayla uğrayabilir. Dizinin her öğesini bir kez ziyaret etmeye **gezinme** (*traversal*) denir. Java'da gezinmenin iki yolu vardır: klasik `for` ve gelişmiş `for`.

10.4.1 Klasik for

Klasik `for` döngüsünde bir sayaç, genellikle `i`, 0'dan başlar ve her turda 1 artar. Koşul `i < dizi.length` olduğu için sayaç tam olarak geçerli indisler üzerinde gezinir:

```
public class Klasik {
    public static void main(String[] args) {
        int[] puanlar = {70, 88, 90};
        for (int i = 0; i < puanlar.length; i++) {
          System.out.printf("%d: %d%n", i, puanlar[i]);
        }
    }
}
```

```
0: 70
1: 88
2: 90
```

`printf` biçimli yazdırma yapar: kalıptaki ilk `%d` yerine `i`, ikinci `%d` yerine `puanlar[i]` yazılır, `%n` de satırı bitirir. Döngünün izlemesi Tablo 10.5'te verilmiştir.

`i`	`i < 3`	`puanlar[i]`	Yazılan satır
0	true	70	`0: 70`
1	true	88	`1: 88`
2	true	90	`2: 90`

`i`	`i < 3`	`puanlar[i]`	**Yazılan satır**
3	false	—	döngü biter

Tablo 10.5: `Klasik` programının izleme tablosu.

Son satır önemlidir: `i` 3 olduğunda koşul yanlıştır ve `puanlar[3]`'e hiç erişilmez. Koşul her turun başında, gövdeden önce sınanır. Bu yüzden `i`'nin geçersiz bir değere ulaşması sorun değildir; sorun, geçersiz değerle gövdeye girmektir.

Klasik `for`'un yapısı serbesttir. Başlangıç, koşul ve adım değiştirilerek dizi farklı biçimlerde gezilebilir. `for (int i = a.length - 1; i >= 0; i--)` diziyi sondan başa doğru gezer. `i += 2` adımı bir öğe atlayarak ilerler. Bu esneklik indisi doğrudan kontrol etmekten gelir.

10.4.2 Gelişmiş for

Pek çok gezinmede indisin kendisine ihtiyaç yoktur; yalnızca öğelerin değerleri gerekir. Bu durumlar için Java'nın daha kısa bir döngüsü vardır.

> **Tanım 10.3**
>
> **Gelişmiş for** (*enhanced for loop*, *for-each*), `for (T x : dizi)` biçiminde yazılan döngüdür. Döngü dizinin öğelerinde baştan sona sırayla gezinir; her turda sıradaki öğenin **kopyasını** `x` değişkenine koyar. Döngüde indis yoktur.

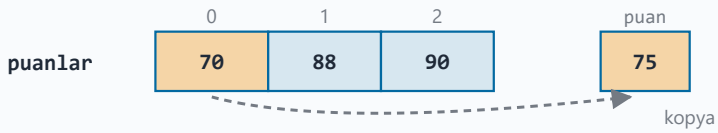
Yazım "dizideki her `x` için" diye okunur. İki noktanın solunda öğe tipiyle bir değişken, sağında dizi bulunur. Tanımdaki "kopya" sözcüğü aşağıdaki programın davranışını açıklar:

```
public class Gelismis {
    public static void main(String[] args) {
        int[] puanlar = {70, 88, 90};
        for (int puan : puanlar) {
            puan = puan + 5;
            System.out.println(puan);
        }
        System.out.println(puanlar[0]);
    }
}
```

```
75
93
95
70
```

Tur	Kopyalanan hücre	`puan` (önce → sonra)	`puanlar`	Çıktı
1	`puanlar[0]`	70 → 75	[70, 88, 90]	75
2	`puanlar[1]`	88 → 93	[70, 88, 90]	93
3	`puanlar[2]`	90 → 95	[70, 88, 90]	95
son	—	kapsam dışı	[70, 88, 90]	70

Tablo 10.6: `Gelismis` programının izleme tablosu. Dizi hiçbir adımda değişmez.



Şekil 10.9: İlk turda `puanlar[0]`'daki 70, `puan` değişkenine kopyalanır. `puan = puan + 5` yalnızca kopyayı 75 yapar; dizideki hücre 70 olarak kalır.

Her turda `puan` beş artırılıp yazdırılır: 75, 93, 95. Son satır ise 70 yazar. Dizi hiç değişmemiştir. Şekil 10.9 nedenini gösterir: `puan`, hücrenin kendisi değil, ayrı bir değişkendir. Hücredeki değer her turun başında oraya kopyalanır. Kopyaya yapılan atama diziye geri yazılmaz. Döngü bittiğinde `puan` değişkeni de kapsam dışına çıkar.

Diziyi gerçekten değiştirmek için hücreye indisle yazmak gerekir. Bu da klasik `for` ister:

```
import java.util.Arrays;

public class Artir {
    public static void main(String[] args) {
        int[] puanlar = {70, 88, 90};
        for (int i = 0; i < puanlar.length; i++) {
            puanlar[i] = puanlar[i] + 5;
        }
        System.out.println(Arrays.toString(puanlar));
    }
}
```

```
[75, 93, 95]
```

Bu programda atamanın solunda `puanlar[i]` vardır; değer doğrudan hücreye yazılır. `Arrays.toString` dizinin bütün öğelerini köşeli parantez içinde yazdırır. Bu metot `java.util.Arrays` sınıfındadır ve programın başında `import` edilir.

Sık yapılan hata

Gelişmiş `for` ile diziyi güncellemeye çalışmak. `for (int x : a) x = x * 2;` derlenir ve çalışır, ama dizi aynı kalır. Derleyici uyarı da vermez, çünkü döngü değişkenine atama dil kurallarına uygundur. Hata ancak çıktı incelenince fark edilir.

10.4.3 Hangi döngü ne zaman?

İki döngü birbirinin yerine her zaman kullanılamaz. Seçimi, işin indise ihtiyaç duyup duymadığı belirler.

İş	Klasik `for`	Gelişmiş `for`
Öğeleri okuyup toplamak, saymak, yazdırmak	Olur	Olur ve daha kısadır
Öğenin konumunu yazdırmak	Olur	Olmaz (indis yok)
Hücreye yeni değer yazmak	Olur	Olmaz (değişken kopyadır)
Komşu öğeye bakmak (`a[i + 1]`)	Olur	Olmaz
Tersten ya da atlayarak gezmek	Olur	Olmaz (hep baştan sona)
İki diziyi aynı indisle birlikte gezmek	Olur	Olmaz

Tablo 10.7: Klasik ve gelişmiş `for` karşılaştırması.

Gelişmiş `for` kısa olduğu kadar güvenlidir de: indis olmadığı için sınır dışı hatası yapılamaz. Bu yüzden yalnızca okuma yapılan gezinmelerde tercih edilir. İndis gerektiğinde klasik `for`'a geçilir.

10.5 Temel dizi işlemleri

Bu kesimdeki işlemler, dizilerle yazılan programların büyük bölümünün yapı taşlarıdır. Her birinde aynı soru sorulur: diziyi kaç kez gezmek gerekir ve gezerken hangi bilgi saklanmalıdır? Örneklerin hepsi O(n) zamanda çalışır, çünkü her öğeye sabit sayıda bakılır.

10.5.1 Toplam ve ortalama

Toplam bulmak için bir **biriktirici** değişken kullanılır. Biriktirici, toplamanın etkisiz elemanı olan 0 ile başlar ve her öğe ona eklenir:

```
public class Toplam {
    public static void main(String[] args) {
        int[] sayilar = {4, 7, 2, 6};
        int toplam = 0;
        for (int deger : sayilar) toplam += deger;
        double ortalama = 0;
        if (sayilar.length > 0)
            ortalama = (double) toplam / sayilar.length;
        System.out.println(toplam + " " + ortalama);
        System.out.println(toplam / sayilar.length);
    }
}
```

```
19 4.75
4
```

Adım	`deger`	`toplam`	`ortalama`
Başlangıç	—	0	—
1. tur	4	0 + 4 = 4	—
2. tur	7	4 + 7 = 11	—
3. tur	2	11 + 2 = 13	—
4. tur	6	13 + 6 = 19	—
`double ortalama = 0;`	—	19	0.0
`4 > 0` → bölme	—	19	19.0 / 4 = 4.75

Tablo 10.8: `Toplam` programının izleme tablosu.

Ortalamada iki ayrıntı vardır. Birincisi `(double)` dönüşümüdür. `toplam` ve `sayilar.length` ikisi de `int`'tir; iki `int`'in bölümü tam sayı bölmesidir ve kesir kısmı atılır. Programın son satırı bunu gösterir: `toplam / sayilar.length` 4 verir. Dönüşüm, bölmeden önce `toplam`'ı `double`'a çevirir; bölme de ondalıklı yapılır ve 4.75 bulunur. İkincisi `if (sayilar.length > 0)` denetimidir. Boş dizide uzunluk 0'dır. Ondalıklı bölmede 0.0 / 0 hata vermez ama `NaN` (*Not a Number*, sayı değil) üretir. Denetim bu durumda ortalamayı güvenli bir başlangıç değeri olan 0.0'da bırakır.

Maliyet: döngü her öğe için bir toplama yapar; n öğe için n toplama, yani O(n) zaman. Ek bellek olarak yalnızca iki değişken kullanılır; bu da O(1) ek alan demektir.

10.5.2 En küçük ve en büyük

En küçük ve en büyük öğeyi bulmak için iki **aday** değişken tutulur. Her öğe adayla karşılaştırılır; aday geçilirse yerini öğe alır. Adayların başlangıç değeri bu yöntemin en kritik noktasıdır:

```
public class MinMax {
    public static void main(String[] args) {
        int[] sayilar = {5, 3, 9, 1};
        int min = sayilar[0], max = sayilar[0];
        for (int deger : sayilar) {
            if (deger < min) min = deger;
            if (deger > max) max = deger;
        }
        System.out.println(min + " " + max);
    }
}
```

```
1 9
```

Tur	`deger`	`deger < min`	`deger > max`	`min`	`max`
Başlangıç	—	—	—	5	5
1	5	5 < 5 → false	5 > 5 → false	5	5
2	3	3 < 5 → true	3 > 5 → false	**3**	5
3	9	9 < 3 → false	9 > 5 → true	3	**9**
4	1	1 < 3 → true	1 > 9 → false	**1**	9

Tablo 10.9: `MinMax` programının izleme tablosu. Değişen değerler kalın yazılmıştır.

Adaylar dizinin ilk öğesiyle başlatılır. Aday her zaman dizide gerçekten bulunan bir değerdir; bu yüzden sonuç da dizinin bir öğesi olur. Adayı 0 ile başlatmak cazip görünür ama yanlıştır. Bütün öğeleri pozitif olan bir dizide `min` hiç güncellenmez ve dizide olmayan 0 bulunur. Bütün öğeleri negatif bir dizide aynı sorun `max`'ta çıkar.

Örnek 10.2

En büyüğü bulan bir metot adayını 0 ile başlatıyor ve `{-8, -3, -5}` dizisiyle çağrılıyor. Ne döner?

Aday 0'dır. Sırayla −8 > 0, −3 > 0, −5 > 0 sınanır; üçü de yanlıştır. Aday hiç değişmez ve metot 0 döndürür. JDK 21 ile çalıştırılan program da 0 yazar. Doğru sonuç −3'tür ve aday `dizi[0]` ile başlatılınca bulunur.

Programda dikkat çeken bir başka nokta: döngü ilk öğeyi de inceler, oysa adaylar zaten o öğeyle başlamıştır. İlk turdaki karşılaştırmalar (5 < 5, 5 > 5) yanlış çıkar ve zarar vermez. İsteyen döngüyü `i = 1`'den başlayan klasik `for` ile yazabilir; sonuç değişmez.

Sık yapılan hata

Boş dizide ilk öğeye erişmek. `int min = sayilar[0];` uzunluğu 0 olan bir dizide hücre bulamaz:

```
0
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 0 out of
bounds for length 0
        at BosDizi.main(BosDizi.java:5)
```

İleti "indis 0, uzunluğu 0 olan dizinin sınırları dışında" der. Gerçek programlarda önce `sayilar.length == 0` durumu ayrıca ele alınır.

10.5.3 Diziyi ters çevirme

Bir diziyi ters çevirmenin kolay yolu, yeni bir dizi açıp öğeleri sondan başa doğru oraya kopyalamaktır. Bu yol n hücrelik ek bellek ister. Daha tutumlu yol, diziyi **yerinde** (*in place*) çevirmektir: iki uçtaki öğeler yer değiştirir, sonra uçlar birer adım içeri kayar. Bu yöntemde iki indis kullanıldığı için yönteme **iki işaretçi** (*two pointers*) yöntemi denir.

```java
import java.util.Arrays;

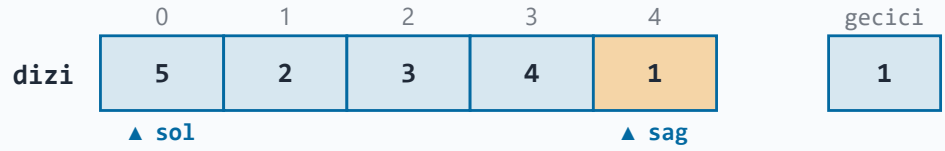
public class Ters {
    public static void main(String[] args) {
        int[] dizi = {1, 2, 3, 4, 5};
        int sol = 0, sag = dizi.length - 1;
        while (sol < sag) {
            int gecici = dizi[sol];
            dizi[sol] = dizi[sag];
            dizi[sag] = gecici;
            sol++;
            sag--;
        }
        System.out.println(Arrays.toString(dizi));
    }
}
```

```
[5, 4, 3, 2, 1]
```

Takas işlemi üç atamayla yapılır. `dizi[sol] = dizi[sag]` doğrudan yazılırsa soldaki değer kaybolur; bu yüzden önce `gecici` değişkeninde saklanır. Ardından sağdaki değer sola, saklanan değer sağa yazılır.

Adım	`sol`	`sag`	`sol < sag`	`gecici`	`dizi`
Başlangıç	0	4	—	—	[1, 2, 3, 4, 5]
1. takas	0	4	true	1	[5, 2, 3, 4, 1]
İşaretçiler kayar	1	3	—	—	[5, 2, 3, 4, 1]
2. takas	1	3	true	2	[5, 4, 3, 2, 1]
İşaretçiler kayar	2	2	false	—	[5, 4, 3, 2, 1]

Tablo 10.10: `Ters` programının izleme tablosu.



Şekil 10.10: İlk takasın sonu. `sol` = 0 ve `sag` = 4 hücreleri yer değiştirmiştir: 5 başa, 1 sona geçmiştir. `gecici` hâlâ 1'i tutar.

Beş öğeli dizide iki takas yapılır. Üçüncü turda iki işaretçi de 2 numaralı hücreyi gösterir; ortadaki öğenin yerini değiştirmeye gerek yoktur ve `2 < 2` yanlış olduğu için döngü biter. Genel olarak n öğeli bir dizide ⌊n/2⌋ takas yapılır; bu da O(n) zamandır. Ek bellek yalnızca üç değişkendir: O(1).

Sık yapılan hata

Takası bütün dizi boyunca yapmak. `for (int i = 0; i < a.length; i++)` ile `a[i]` ve `a[a.length - 1 - i]` takas edilirse, dizinin ilk yarısında yapılan takaslar ikinci yarıda geri alınır. `{1, 2, 3, 4}` dizisi programın sonunda yine `[1, 2, 3, 4]` olur. Döngü yalnızca orta-ya kadar, `i < a.length / 2` koşuluyla dönmelidir.

10.5.4 Ortalamanın üstündekileri sayma

Bazı sorular tek geçişte cevaplanamaz. “Kaç öğrencinin notu ortalamanın üstünde?” sorusu bunlardan biridir. Bir notun ortalamanın üstünde olup olmadığını bilmek için önce ortalama gerekir. Ortalama ise ancak bütün notlar okunduktan sonra bulunur. Bu yüzden dizi iki kez gezilir. Dizinin değerli bir yanı da budur: veriyi saklar ve ikinci kez okumaya izin verir.

```
public class Ustte {
    public static void main(String[] args) {
        int[] notlar = {60, 75, 90, 45, 80};
        int toplam = 0;
        for (int x : notlar) toplam += x;
        double ort = (double) toplam / notlar.length;
        int ustte = 0;
        for (int x : notlar)
          if (x > ort) ustte++;
        System.out.println(ort + " " + ustte);
    }
}
```

```
70.0 3
```

`x`	**1. geçiş:** `toplam`	**2. geçiş:** `x > 70.0`	`ustte`
60	60	false	0
75	135	true	1
90	225	true	2
45	270	false	2
80	350	true	3

Tablo 10.11: `Ustte` programının izleme tablosu. Birinci geçiş toplamı, ikinci geçiş sayacı günceller.

Birinci geçişin sonunda toplam 350'dir ve ortalama 350 / 5 = 70.0 bulunur. İkinci geçişte her not 70.0 ile karşılaştırılır; 75, 90 ve 80 üstündedir. Program `70.0 3` yazar. İki geçişin toplam maliyeti 2n karşılaştırma ve toplamadır. Büyük O gösteriminde sabit çarpanlar yazılmaz; maliyet yine O(n)'dir.

10.5.5 En büyük ikinci öğe

Dizideki en büyük ikinci öğe için diziyi sıralamak gerekmez. Tek geçişte iki aday tutulur: `enBuyuk` ve `ikinci`. Tekrarlanan değerler soruyu inceltir: `{7, 12, 5, 12, 9}` dizisinde en büyük 12'dir ve iki kez geçer; en büyük ikinci öğe ise 9'dur.

```
public class Ikinci {
    public static void main(String[] args) {
        int[] a = {7, 12, 5, 12, 9};
        int enBuyuk = Integer.MIN_VALUE;
        int ikinci = Integer.MIN_VALUE;
        for (int x : a) {
          if (x > enBuyuk) {
            ikinci = enBuyuk;
            enBuyuk = x;
          } else if (x > ikinci && x != enBuyuk) {
            ikinci = x;
          }
        }
        System.out.println(ikinci);
        System.out.println(Integer.MIN_VALUE);
    }
}
```

```
9
-2147483648
```

İki aday da `Integer.MIN_VALUE` ile başlar. Bu sabit `int` tipinin alabileceği en küçük değerdir, yani −2147483648; programın ikinci çıktı satırı onu gösterir. Böylece dizinin ilk öğesi, ne olursa olsun, adaylardan büyük çıkar. Her öğe için iki durum vardır:

1. `x > enBuyuk` ise yeni bir en büyük bulunmuştur. Eski en büyük boşa gitmez, ikinciliğe iner: önce `ikinci = enBuyuk`, sonra `enBuyuk = x`. Sıra ters olursa eski değer kaybolur.
2. Değilse ve `x` hem `ikinci`'den büyük hem de en büyükten farklıysa, `x` yeni ikincidir. `x != enBuyuk` koşulu, en büyüğün tekrarının ikinci sayılmasını engeller.

`x`	**Durum**	`enBuyuk`	`ikinci`
—	Başlangıç	MIN	MIN
7	7 > MIN: yeni en büyük	7	MIN
12	12 > 7: yeni en büyük, 7 ikinciye iner	12	7
5	5, ikinciden (7) büyük değil	12	7
12	12 == enBuyuk: tekrar, sayılmaz	12	7
9	9 > 7 ve 9 != 12: yeni ikinci	12	9

Tablo 10.12: `Ikinci` programının izleme tablosu. MIN, `Integer.MIN_VALUE`'dur.

Dizi tek kez gezilir: O(n) zaman, O(1) ek alan. Dizide yalnızca tek bir farklı değer varsa, örneğin `{4, 4}`, `ikinci` hiç güncellenmez ve `Integer.MIN_VALUE` kalır. Gerçek bir programda bu durum “ikinci en büyük yok” diye ayrıca bildirilmelidir.

10.5.6 Frekans sayma: zar histogramı

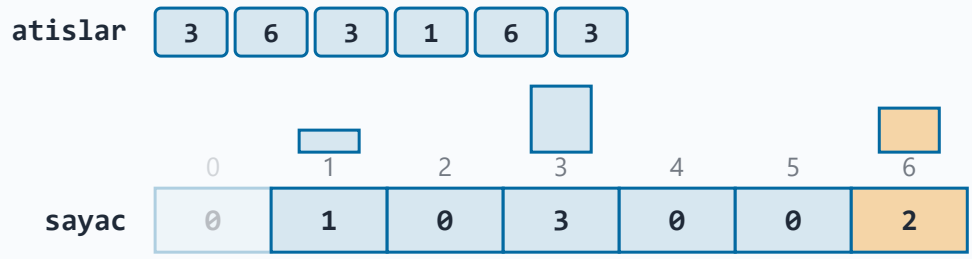
Bir zarın altı yüzünden her birinin kaç kez geldiğini saymak için altı sayaç gerekir. Altı ayrı değişken ve uzun bir `if-else` zinciri yerine tek bir sayaç dizisi kullanılır. Yöntemin püf noktası, **değeri indis olarak kullanmaktır**: zar 3 geldiyse `sayac[3]` bir artırılır.

```
public class Histogram {
    public static void main(String[] args) {
        int[] atislar = {3, 6, 3, 1, 6, 3};
        int[] sayac = new int[7];
        for (int zar : atislar)
          sayac[zar]++;
        for (int y = 1; y <= 6; y++) {
          System.out.print(y + ": ");
          for (int k = 0; k < sayac[y]; k++)
            System.out.print("*");
          System.out.println();
        }
    }
}
```

```
1: *
2:
3: ***
4:
5:
6: **
```

Sayaç dizisi 7 hücrelidir, çünkü zar değerleri 1 ile 6 arasındadır ve 6 numaralı hücreye ihtiyaç vardır. 0 numaralı hücre hiç kullanılmaz. Bir hücre boşa gider, ama karşılığında `sayac[zar]` yazımı doğrudan okunur. Atışlar örnekte sabit verilmiştir; gerçek bir programda `rastgele.nextInt(6) + 1` ile üretilir.

İlk döngü atışları sayar. `sayac[zar]++` tek bir deyimdir: önce `zar`'ın değeri okunur, o numaralı hücre bulunur ve hücredeki değer bir artırılır. Altı atıştan sonra sayaç dizisi `[0, 1, 0, 3, 0, 0, 2]` olur. İkinci döngü her yüz için bir satır yazar; iç döngü o satıra sayaç kadar yıldız koyar. 2, 4 ve 5 hiç gelmediği için o satırlarda yıldız yoktur.



Şekil 10.11: Bütün atışlar sayıldıktan sonra sayaç dizisi. Hücre numarası zar değeridir; 0 numaralı hücre soluk çizilmiştir, çünkü kullanılmaz. Hücrelerin üstündeki çubuklar sayaç değerleriyle orantılıdır.

Sayma döngüsü her atış için tek bir artırma yapar; m atış için O(m) zaman. `if-else` zinciri kullanılsaydı her atış için altıya kadar karşılaştırma gerekirdi. Aynı yöntem harflerin ya da sınav notlarının frekansını bulmakta da kullanılır; tek koşul, değerlerin küçük ve negatif olmayan tam sayılara karşılık gelmesidir.

10.5.7 Eratosthenes eleği

Belirli bir sayıya kadar bütün asal sayıları bulmanın eski ve hızlı bir yolu, Eratosthenes eleğidir. Yöntem asal sayıları aramaz; asal olmayanları eler. 2'den başlanır. Elenmemiş her sayı asaldır ve onun katları elenir. Elenmiş sayıların yerini tutmak için bir `boolean` dizisi kullanılır: `asalDegil[k]` true ise k elenmiştir.

```
public class Elek {
    public static void main(String[] args) {
        int n = 20;
        boolean[] asalDegil = new boolean[n + 1];
        for (int i = 2; i * i <= n; i++)
          if (!asalDegil[i])
            for (int k = i * i; k <= n; k += i)
              asalDegil[k] = true;
        for (int i = 2; i <= n; i++)
          if (!asalDegil[i]) System.out.print(i + " ");
        System.out.println();
    }
}
```

```
2 3 5 7 11 13 17 19
```

Dizi `n + 1` hücrelidir, çünkü indis olarak 20 de kullanılır. Başlangıçta bütün hücreler `false`'tur; bu, `boolean` dizisinin varsayılan değeridir ve "henüz elenmedi" anlamına gelir. İç döngü katları işaretler. Elemenin adımları JDK 21 ile ayrıca izlenmiş, her `i` için işaretlenen hücreler Tablo 10.13'te verilmiştir.

`i`	`i * i <= 20`	**Durum**	**İşaretlenen `k` değerleri**
2	4 <= 20 → true	Elenmemiş, asal	4, 6, 8, 10, 12, 14, 16, 18, 20
3	9 <= 20 → true	Elenmemiş, asal	9, 12 (zaten), 15, 18 (zaten)
4	16 <= 20 → true	Elenmiş; katlarına bakılmaz	—
5	25 <= 20 → false	Dış döngü biter	—

Tablo 10.13: n = 20 için elemenin adımları. “zaten” sözcüğü, hücrenin daha önce başka bir asalın katı olarak işaretlendiğini gösterir.



Şekil 10.12: Elemenin sonunda `asalDegil` dizisi. Üstü çizili sayılar elenmiştir. Onay işaretli sekiz sayı (2, 3, 5, 7, 11, 13, 17, 19) asaldır.

Yöntemde iki hızlandırma vardır. Birincisi, katlar `i * i`'den başlar. 3'ün 6 gibi daha küçük katları, daha küçük bir asalın (2'nin) katı olarak zaten elenmiştir. İkincisi, dış döngü `i * i <= n` koşuluyla durur. 20'ye kadar asal olmayan her sayının $\sqrt{20} \approx 4{,}47$'den küçük ya da ona eşit bir böleni vardır. Bu yüzden 5 ve sonrası için eleme yapmaya gerek kalmaz. Aynı program n = 100 ile çalıştırılınca 25 asal bulur.

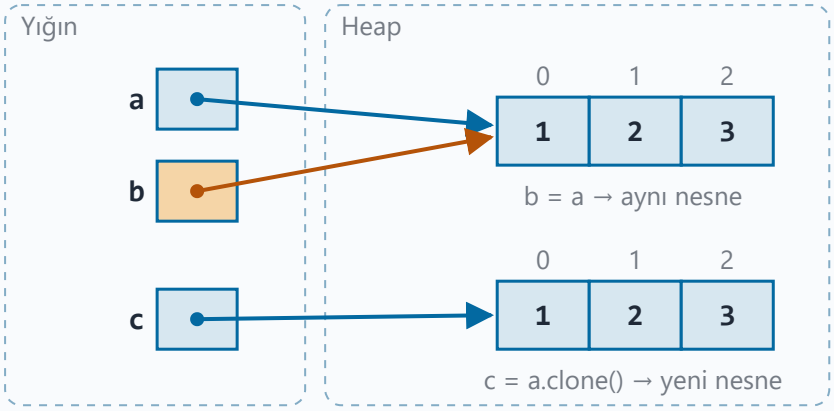
Elek, bir dizinin hem veri hem de “işaret tahtası” olarak kullanılabildiğini gösterir. Her sayının asal olup olmadığını bölme deneyerek sınamak, büyük n'ler için elekten çok daha yavaştır. Eleğin zaman maliyeti O(n log log n)'dir; pratikte n'ye çok yakın büyür. Bedeli ise n + 1 hücrelik bir `boolean` dizisidir.

10.6 Kopyalama, Arrays sınıfı ve metotlar

Dizi değişkeninin bir referans tuttuğu, en çok kopyalamada önem kazanır. Temel tipte `int y = x;` değeri kopyalar ve iki değişken birbirinden bağımsız olur. Dizilerde aynı yazım bambaşka bir sonuç verir.

10.6.1 Referans kopyası ve öğe kopyası

`int[] b = a;` deyimi yeni bir dizi oluşturmaz. `a`'daki referans `b`'ye kopyalanır; artık iki değişken aynı nesneyi gösterir. Bu duruma **referans kopyası** denir. Birinin üzerinden yapılan değişiklik ötekinden de görülür, çünkü ortada tek bir dizi vardır.



Şekil 10.13: Referans kopyası ve öğe kopyası. `b = a` sonrasında iki ok aynı nesneyi gösterir. `c = a.clone()` ise heap'te yeni bir dizi oluşturur ve öğeleri oraya kopyalar.

Bağımsız bir kopya için dizinin öğeleri yeni bir diziye aktarılmalıdır. Java bunun için üç hazır yol sunar:

- `a.clone()`: aynı uzunlukta yeni bir dizi oluşturur ve bütün öğeleri kopyalar.
- `Arrays.copyOf(a, a.length)`: aynı işi yapar; ikinci argümanla yeni dizinin uzunluğu da seçilebilir.
- `System.arraycopy(kaynak, kBas, hedef, hBas, adet)`: var olan bir hedef diziye, belirli bir konumdan başlayarak belirli sayıda öğe kopyalar.

Üçüncü yol en esnek olanıdır, ama hedef dizinin önceden oluşturulmuş olmasını ister:

```
import java.util.Arrays;

public class ArrayCopy {
    public static void main(String[] args) {
        int[] a = {1, 2, 3, 4, 5};
        int[] d = new int[7];
        System.arraycopy(a, 1, d, 2, 3);
        System.out.println(Arrays.toString(d));
    }
}
```

```
[0, 0, 2, 3, 4, 0, 0]
```

`System.arraycopy(a, 1, d, 2, 3)` çağrısı "`a`'nın 1 numaralı hücresinden başlayarak 3 öğeyi, `d`'nin 2 numaralı hücresinden başlayarak yaz" demektir. Kopyalanan öğeler 2, 3 ve 4'tür; `d`'nin 2, 3 ve 4 numaralı hücrelerine yazılırlar. Kalan hücreler varsayılan değer olan 0'da kalır.

Tanım 10.4

Sığ kopya (*shallow copy*), bir dizinin yalnızca hücrelerindeki değerlerin kopyalandığı kopyadır. Hücrelerde temel tipte değerler varsa sığ kopya tamamen bağımsızdır. Hücrelerde referanslar varsa yalnızca referanslar kopyalanır; gösterdikleri nesneler iki dizi arasında paylaşılır.

`clone`, `Arrays.copyOf` ve `System.arraycopy` sığ kopya yapar. `int` dizilerinde bu bir sorun yaratmaz, çünkü hücrelerde sayıların kendisi durur. Nesne dizilerinde ve iki boyutlu dizilerde ise nesneler kopyalanmaz, paylaşılır. Bu tuzak iki boyutlu dizilerle birlikte ayrıca işlenir.

10.6.2 Kopyalama adım adım

Aşağıdaki program iki kopya türünü yan yana gösterir. Son satır, iki referansı `==` ile karşılaştırır:

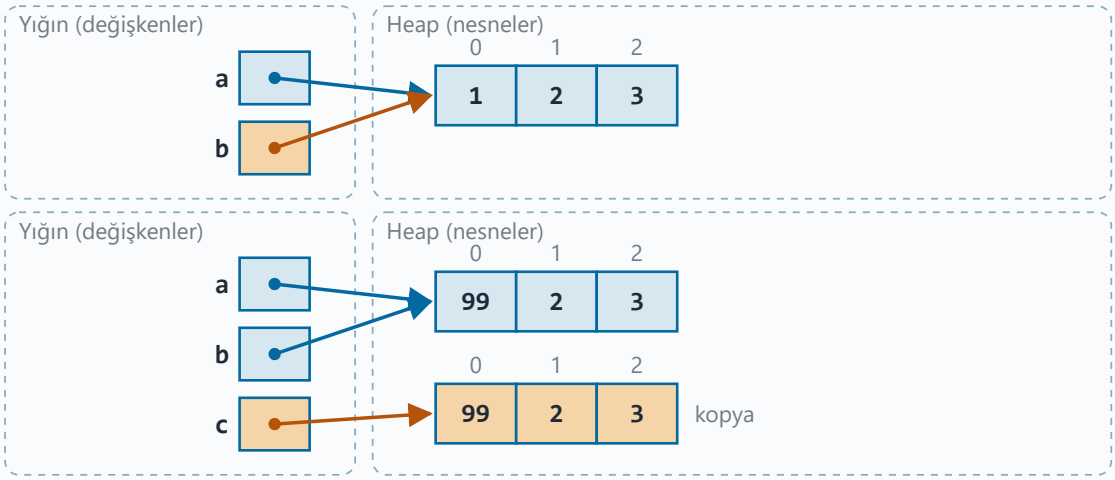
```
public class Kopya {
    public static void main(String[] args) {
        int[] a = {1, 2, 3};
        int[] b = a;
        b[0] = 99;
        System.out.println(a[0]);
        int[] c = a.clone();
        c[1] = 50;
        System.out.println(a[1] + " " + c[1]);
        System.out.println((a == b) + " " + (a == c));
    }
}
```

```
99
2 50
true false
```

Satır	a	b	c	Nesne 1	Nesne 2	Çıktı
`int[] a = {1, 2, 3};`	→ 1	—	—	[1, 2, 3]	—	
`int[] b = a;`	→ 1	→ 1	—	[1, 2, 3]	—	
`b[0] = 99;`	→ 1	→ 1	—	[99, 2, 3]	—	
`println(a[0])`	→ 1	→ 1	—	[99, 2, 3]	—	99
`int[] c = a.clone();`	→ 1	→ 1	→ 2	[99, 2, 3]	[99, 2, 3]	
`c[1] = 50;`	→ 1	→ 1	→ 2	[99, 2, 3]	[99, 50, 3]	

Satır	a	b	c	Nesne 1	Nesne 2	Çıktı
`println(a[1] + " " + c[1])`	→ 1	→ 1	→ 2	[99, 2, 3]	[99, 50, 3]	2 50

Tablo 10.14: `Kopya` programının izleme tablosu. “Nesne 1” ve “nesne 2” heap'teki iki ayrı diziyi gösterir.



Şekil 10.14: Üstte `b = a` sonrası: iki ok aynı nesnede. Altta `a.clone()` sonrası: `c`, o anki içeriğin (99, 2, 3) kopyası olan yeni nesneyi gösterir.

`b[0] = 99` satırı yalnızca `b`'yi değiştiriyor gibi görünür. Oysa `b`'nin gösterdiği nesne `a`'nın da nesnesidir ve `a[0]` 99 olur. `clone` çağrısı ise o anki içeriği yeni bir nesneye kopyalar. Bundan sonra `c` üzerindeki değişiklik `a`'yı etkilemez. Son satırdaki `true false` çıktısı iki ilişkiyi özetler. `a == b` doğrudur, çünkü iki değişken aynı referansı tutar. `a == c` yanlıştır; içerikleri aynı olsa da bunlar farklı nesnelerdir.

10.6.3 Arrays yardımcıları

`java.util.Arrays` sınıfı, dizilerle sık yapılan işler için hazır metotlar içerir. Bu metotlar `static`'tir ve sınıf adıyla çağrılır: `Arrays.toString(a)`. Programın başına `import java.util.Arrays;` satırı eklenir. Aşağıdaki program sunumdaki tablonun her satırını JDK 21 ile çalıştırır:

```
import java.util.Arrays;

public class Yardimci {
    public static void main(String[] args) {
        int[] a = {5, 2, 9}, r = {10, 20, 30, 40, 50}, p = {1, 2, 3}, q = {1, 2, 3}, f =
new int[4];
        int[][] m = {{1, 2}, {3, 4}};
        System.out.println(Arrays.toString(a));
        System.out.println(Arrays.deepToString(m));
        System.out.println(Arrays.toString(m).startsWith("[[I@"));
        Arrays.fill(f, 7);
        System.out.println(Arrays.toString(f));
        System.out.println((p == q) + " / " + p.equals(q));
        System.out.println(Arrays.equals(p, q));
        System.out.println(Arrays.toString(Arrays.copyOf(a, 5)));
        System.out.println(Arrays.toString(Arrays.copyOf(a, 2)));
        System.out.println(Arrays.toString(Arrays.copyOfRange(r, 1, 4)));
        System.out.println(a);
    }
}
```

```
[5, 2, 9]
[[1, 2], [3, 4]]
true
[7, 7, 7, 7]
false / false
true
[5, 2, 9, 0, 0]
[5, 2]
[20, 30, 40]
[I@548c4f57
```

Çağrı	Sonuç	Açıklama
`Arrays.toString(a)`	`[5, 2, 9]`	Tek boyutlu dizinin öğelerini yazar
`Arrays.deepToString(m)`	`[[1, 2], [3, 4]]`	İç içe dizileri de açarak yazar
`Arrays.toString(m)`	`[[I@…`	2B dizide satırların referanslarını yazar
`Arrays.fill(f, 7)`	`[7, 7, 7, 7]`	Bütün hücrelere aynı değeri yazar
`p == q` / `p.equals(q)`	`false / false`	İkisi de referans karşılaştırır
`Arrays.equals(p, q)`	`true`	İçeriği öğe öğe karşılaştırır
`Arrays.copyOf(a, 5)`	`[5, 2, 9, 0, 0]`	Uzun kopya: fazla hücreler 0
`Arrays.copyOf(a, 2)`	`[5, 2]`	Kısa kopya: sondan kesilir
`Arrays.copyOfRange(r, 1, 4)`	`[20, 30, 40]`	1, 2 ve 3 numaralı öğeler; bitiş dahil değil

Tablo 10.15: Sık kullanılan `Arrays` metotları ve `Yardimci` programındaki sonuçları.

Programın son satırı, diziyi doğrudan `println` ile yazdırmanın sonucunu gösterir: `[I@` ile başlayan bir metin. Baştaki `[I` "int dizisi" demektir. `@` işaretinden sonraki onaltılık sayı nesneye ait bir koddur ve her çalıştırmada değişebilir. Bu çıktı öğeler hakkında hiçbir bilgi vermez. İki boyutlu bir diziye `Arrays.toString` uygulandığında da aynı şey olur: dış dizinin hücreleri satır dizilerinin referanslarıdır ve her biri `[I@…` biçiminde yazılır. Program bu satırda yalnızca metnin `[[I@` ile başladığını sınar ve `true` yazar; kodun kendisi çalıştırmadan çalıştırmaya değiştiği için gösterilmemiştir.

`equals` konusu ayrıca dikkat ister. `String`'de `equals` içeriği karşılaştırır. Dizilerde ise `p.equals(q)`, `p == q` ile aynı işi yapar ve yalnızca referansa bakar. İçeriği karşılaştırmak için `Arrays.equals(p, q)`, iki boyutlu dizilerde `Arrays.deepEquals` kullanılır.

`copyOfRange`'teki aralık, Java'nın genel bir alışkanlığını izler: başlangıç indisi dahil, bitiş indisi hariçtir. `copyOfRange(r, 1, 4)` 1, 2 ve 3 numaralı öğeleri alır; kopyanın uzunluğu 4 − 1 = 3'tür. Aynı kural `String` sınıfının `substring` metodunda da geçerlidir.

10.6.4 Diziler ve metotlar

Bir metot dizi alabilir ve dizi döndürebilir. Java'da metoda verilen her argüman kopyalanarak aktarılır. Dizi argümanında kopyalanan şey referanstır, dizinin kendisi değil. Bu yüzden metottaki parametre, çağıran taraftaki diziyle **aynı** nesneyi gösterir. Sonuç iki kurala bağlanır:

1. Metot parametre üzerinden hücrelere yazarsa, değişiklik çağıranın dizisinde de görülür.
2. Metot parametreye yeni bir dizi atarsa, yalnızca kendi yerel kopyasındaki referans değişir; çağıranın değişkeni etkilenmez.

```
import java.util.Arrays;

public class Metot {
    static void ikiKatinaCikar(int[] dizi) {
        for (int i = 0; i < dizi.length; i++)
            dizi[i] = dizi[i] * 2;
    }

    static void yeniDiziAta(int[] dizi) {
        dizi = new int[]{0, 0, 0};
    }

    static int[] kareler(int n) {
        int[] sonuc = new int[n];
        for (int i = 0; i < n; i++)
            sonuc[i] = (i + 1) * (i + 1);
        return sonuc;
    }

    public static void main(String[] args) {
        int[] a = {1, 2, 3};
        ikiKatinaCikar(a);
        System.out.println(Arrays.toString(a));
        yeniDiziAta(a);
        System.out.println(Arrays.toString(a));
        System.out.println(Arrays.toString(kareler(4)));
    }
}
```

```
[2, 4, 6]
[2, 4, 6]
[1, 4, 9, 16]
```

`ikiKatinaCikar(a)` çağrısında `dizi` parametresi `a`'nın nesnesini gösterir ve hücrelere yazar; `a` `[2, 4, 6]` olur. `yeniDiziAta(a)` ise `dizi`'ye yeni bir nesne bağlar. Bu atama yalnızca metodun kendi değişkenini etkiler; `a` eski nesneyi göstermeyi sürdürür ve `[2, 4, 6]` kalır. `kareler(4)` metodu yeni bir dizi oluşturup referansını döndürür. Metot bittiğinde yerel `sonuc` değişkeni ortadan kalkar, ama heap'teki dizi yaşamaya devam eder, çünkü referansı çağırana verilmiştir.

Bu davranışın bir yan etkisi vardır. Diziyi sıralayan, ters çeviren ya da dolduran bir metot, çağıranın verisini kalıcı olarak değiştirir. Özgün sıranın korunması gerekiyorsa metot önce bir kopya almalıdır. Sıralama ve ortanca örneklerinde bu kural yeniden karşımıza çıkar. Bu arada `main` metodunun parametresi de bir dizidir: `String[] args`, komut satırından verilen argümanları tutar ve `args.length` argüman sayısını verir.

10.7 Arama ve sıralama

Doğrusal arama, ikili arama, kabarcık sıralaması ve eklemeli sıralama algoritmalarının fikri, sözde kodu ve ayrıntılı izlemeleri Algoritmalar ders notunda anlatılmıştır. Bu kesim aynı algoritmaların Java dizileriyle yazımına, kısa izlemelerine ve dizilere özgü noktalara odaklanır: indis sınırları, yerinde değişiklik, `Arrays` sınıfının hazır metotları.

10.7.1 Doğrusal arama

Doğrusal arama diziyi baştan sona gezer ve her öğeyi hedefle karşılaştırır. İlk eşleşmede indisi döndürür; hedef yoksa −1 döndürür. −1 geçerli bir indis olamayacağı için "bulunamadı" anlamında kullanılır.

```
public class Dogrusal {
    static int ara(int[] dizi, int hedef) {
        for (int i = 0; i < dizi.length; i++)
            if (dizi[i] == hedef) return i;
        return -1;
    }

    public static void main(String[] args) {
        int[] d = {8, 3, 6, 2};
        System.out.println(ara(d, 6));
        System.out.println(ara(d, 5));
    }
}
```

```
2
-1
```

`i`	`dizi[i]`	`== 6` ?
0	8	false
1	3	false
2	6	true → `return 2`

Tablo 10.16: `ara(d, 6)` çağrısının izlemesi.

Java yazımında iki nokta göze çarpar. Birincisi, `return i` döngünün içindedir; metot ilk eşleşmede biter ve kalan öğelere bakılmaz. `return -1` ise döngünün dışındadır ve ancak bütün öğeler denendikten sonra çalışır. Bu satırı yanlışlıkla döngünün içine, bir `else` koluna koymak, aramayı ilk öğede bitirir. İkincisi, `dizi` parametresi `d` ile aynı diziyi gösterir; çağrı sırasında hiçbir öğe kopyalanmaz. 5 dizide olmadığı için ikinci çağrı dört karşılaştırmadan sonra −1 döndürür. En kötü durumda n karşılaştırma yapılır: O(n).

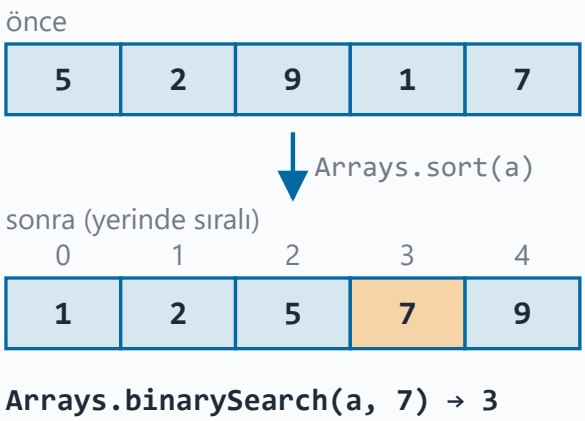
10.7.2 Arrays.sort ve Arrays.binarySearch

Java, sıralama ve ikili arama için hazır metotlar sunar. `Arrays.sort(a)` diziyi küçükten büyüğe sıralar. `Arrays.binarySearch(a, x)` sıralı bir dizide `x`'i ikili aramayla arar.

```
import java.util.Arrays;

public class Sirala {
    public static void main(String[] args) {
        int[] a = {5, 2, 9, 1, 7};
        Arrays.sort(a);
        System.out.println(Arrays.toString(a));
        System.out.println(Arrays.binarySearch(a, 7));
        System.out.println(Arrays.binarySearch(a, 6));
    }
}
```

```
[1, 2, 5, 7, 9]
3
-4
```



Şekil 10.15: `Arrays.sort` diziyi yerinde sıralar: aynı nesnenin hücreleri yeniden düzenlenir. Sıralı dizide 7'nin indisi 3'tür.

`Arrays.sort` yeni bir dizi döndürmez; dizinin kendisini **yerinde** değiştirir (Şekil 10.15). Özgün sıra kaybolur. Özgün sıra başka bir iş için gerekiyorsa önce kopya alınır ve kopya sıralanır.

`Arrays.binarySearch` hedefi bulursa indisini döndürür. Bulamazsa negatif bir değer döndürür: `-(eklenme noktası) - 1`. **Eklenme noktası**, hedefin sıralı dizide gireceği konumdur. Sıralı `[1, 2, 5, 7, 9]` dizisinde 6, 5 ile 7 arasına, yani 3 numaralı konuma girerdi. Dönüş değeri −3 − 1 = −4'tür. Formüldeki "− 1", eklenme noktası 0 olduğunda bile sonucun negatif çıkmasını sağlar; −0 ile 0 ayırt edilemezdi.

Sık yapılan hata

`Arrays.binarySearch`'ü sıralanmamış bir dizide çağırmak. Metot dizinin sıralı olduğunu varsayar ve bunu denetlemez. Sonuç anlamsız olabilir: dizide bulunan bir değer için bile "yok" cevabı alınabilir. Kendini Dene kesimindeki bir soru bu durumu somut bir dizide gösterir.

Her iki metot da `java.util.Arrays` sınıfındadır ve `import java.util.Arrays;` gerektirir. Temel tipteki diziler için `Arrays.sort`, ortalama O(n log n) zamanda çalışan bir hızlı sıralama türü kullanır. Bu, kabarcık ve eklemeli sıralamanın $O(n^2)$ maliyetinden büyük dizilerde çok daha iyidir.

10.7.3 Ortanca

Tanım 10.5

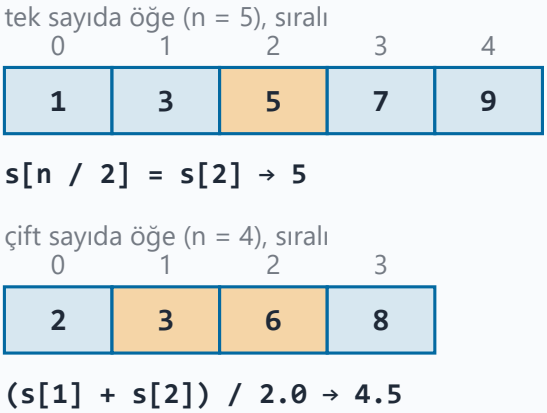
Ortanca (*median*), sıralanmış bir dizinin tam ortasındaki değerdir. Öğe sayısı n tekse ortanca `s[n / 2]`'dir. n çiftse ortadaki iki öğenin, yani `s[n/2 – 1]` ile `s[n/2]`'nin ortalamasıdır.

Ortancanın ortalamadan üstün bir yanı, uç değerlerden etkilenmemesidir. `{1, 2, 3, 4, 100}` dizisinin ortalaması 110 / 5 = 22.0'dır; bu değer öğelerin hiçbirine yakın değildir. Ortanca ise 3'tür. Gelir, ev fiyatı ya da yanıt süresi gibi verilerde birkaç aşırı değer bulunur; bu yüzden ortanca daha anlamlı bir "tipik değer" verir.

```
import java.util.Arrays;

public class Medyan {
    public static void main(String[] args) {
        int[] a = {7, 3, 9, 1, 5};
        int[] s = Arrays.copyOf(a, a.length);
        Arrays.sort(s);    // a bozulmaz
        int n = s.length;
        double m = (n % 2 == 1) ? s[n / 2]
            : (s[n/2 - 1] + s[n/2]) / 2.0;
        System.out.println(m);   // 5.0
        System.out.println(Arrays.toString(a) + " " + Arrays.toString(s));
    }
}
```

```
5.0
[7, 3, 9, 1, 5] [1, 3, 5, 7, 9]
```



Şekil 10.16: Üstte tek sayıda öğe (n = 5): ortanca `s[2]` = 5. Altta çift sayıda öğe (n = 4): ortanca (`s[1]` + `s[2]`) / 2.0 = (3 + 6) / 2.0 = 4.5.

Programda dizilere özgü üç nokta vardır. Birincisi, sıralama kopya üzerinde yapılır: `Arrays.copyOf` ile `s` oluşturulur ve `a`'nın özgün sırası korunur. Programın ikinci satırı bunu doğrular: `a` hâlâ `[7, 3, 9, 1, 5]`'tir. İkincisi, koşullu işleç (`? :`) tek ve çift durumu tek ifadede birleştirir. Üçüncüsü, çift durumda `2.0` ile bölünür. `2` ile bölünseydi iki `int`'in bölümü alınırdı ve (3 + 6) / 2 sonucu 4.5 yerine 4 çıkardı. Aşağıdaki program bu farkı ve uç değer örneğini JDK 21 ile doğrular:

```
import java.util.Arrays;

public class Medyan2 {
    static double ortanca(int[] a) {
        int[] s = Arrays.copyOf(a, a.length);
        Arrays.sort(s);
        int n = s.length;
        return (n % 2 == 1) ? s[n / 2] : (s[n/2 - 1] + s[n/2]) / 2.0;
    }

    public static void main(String[] args) {
        int[] u = {1, 2, 3, 4, 100};
        int t = 0;
        for (int x : u) t += x;
        System.out.println((double) t / u.length + " " + ortanca(u));
        System.out.println(ortanca(new int[]{8, 2, 6, 3}));
        System.out.println((3 + 6) / 2);
    }
}
```

```
22.0 3.0
4.5
4
```

Maliyeti sıralama belirler: O(n log n). Kopya için n hücrelik ek bellek gerekir. Sıralama yapmadan ortancayı ortalama O(n) zamanda bulan yöntemler de vardır, ama bu dersin kapsamı dışındadır.

> **Sık yapılan hata**
>
> Ortancayı bulan bir metodun diziyi kopyalamadan sıralaması. Metoda dizinin referansı gittiği için `Arrays.sort(dizi)` çağıranın dizisini de sıralar. Çağıran taraf, ortancayı aldıktan sonra kendi dizisinin sırasının değiştiğini görür. Çözüm, metodun içinde `dizi.clone()` ya da `Arrays.copyOf` ile alınan kopyayı sıralamaktır.

10.7.4 Kabarcık sıralaması

Kabarcık sıralaması komşu öğeleri karşılaştırır ve sırası bozuk olanları takas eder. Her geçişte kalan öğelerin en büyüğü, suyun yüzüne çıkan bir kabarcık gibi sona yerleşir. Java yazımı iki iç içe `for` döngüsüdür:

```
import java.util.Arrays;

public class Kabarcik {
    public static void main(String[] args) {
        int[] a = {5, 1, 4, 2};
        int n = a.length;
        for (int gecis = 0; gecis < n - 1; gecis++) {
          for (int j = 0; j < n - 1 - gecis; j++) {
            if (a[j] > a[j + 1]) {
              int gecici = a[j];
              a[j] = a[j + 1];
              a[j + 1] = gecici;
            }
          }
        }
        System.out.println(Arrays.toString(a));
    }
}
```

```
[1, 2, 4, 5]
```

Dış döngü geçişleri sayar: n öğe için n − 1 geçiş yeterlidir, çünkü n − 1 öğe yerine oturunca sonuncusu da kendiliğinden yerindedir. İç döngünün sınırı `j < n - 1 - gecis`'tir. Bu sınırda dizilere özgü iki ayrıntı vardır. Birincisi "`- 1`"dir: gövde `a[j + 1]`'e eriştiği için `j` en çok `n − 2` olabilir. Sınır `j < n` olsaydı son turda `a[n]` okunur ve sınır dışı hatası alınırdı. İkincisi "`- gecis`"tir: her geçiş sondaki bir öğeyi yerine oturttuğu için o bölgeye yeniden bakmaya gerek yoktur.

Geçiş	İç döngüdeki adımlar	Geçiş sonu dizi	Karşılaştırma	Takas
1	5↔1, 5↔4, 5↔2	[1, 4, 2, 5]	3	3
2	1–4 sıralı, 4↔2	[1, 2, 4, 5]	2	1
3	1–2 sıralı	[1, 2, 4, 5]	1	0

Tablo 10.17: `{5, 1, 4, 2}` dizisinde kabarcık sıralamasının geçişleri. Değerler, sayaç eklenmiş bir sürümün JDK 21 çıktısından alınmıştır.

Toplam karşılaştırma sayısı 3 + 2 + 1 = 6'dır. Genel olarak n öğe için (n − 1) + (n − 2) + … + 1 = n(n − 1)/2 karşılaştırma yapılır; bu $O(n^2)$ demektir. Üçüncü geçişte hiç takas olmaz, çünkü dizi ikinci geçişten sonra sıralanmıştır. Bu gözlem bir iyileştirmeye yol açar: bir geçişte takas yapılmazsa döngüden erken çıkılabilir. Bu iyileştirme sunumdaki alıştırmalardan biridir. Sıralama yerinde yapılır; ek bellek yalnızca `gecici` değişkenidir: O(1).

10.7.5 Eklemeli sıralama

Eklemeli sıralama, dizinin solunda sıralı bir bölüm büyütür. Her turda sıralı bölümün hemen sağındaki öğe **anahtar** olarak alınır. Sıralı bölümde anahtardan büyük olan öğeler birer hücre sağa kaydırılır ve anahtar açılan boşluğa yazılır. Elde iskambil kâğıdı dizmeye benzer.

```
import java.util.Arrays;

public class Eklemeli {
    public static void main(String[] args) {
        int[] a = {4, 3, 5, 1};
        for (int i = 1; i < a.length; i++) {
          int anahtar = a[i];
          int j = i - 1;
          while (j >= 0 && a[j] > anahtar) {
            a[j + 1] = a[j];
            j--;
          }
          a[j + 1] = anahtar;
        }
        System.out.println(Arrays.toString(a));
    }
}
```

```
[1, 3, 4, 5]
```

`i`	`anahtar`	Sağa kayanlar	Anahtarın yazıldığı indis	Tur sonu dizi
1	3	4	0	[3, 4, 5, 1]
2	5	—	2	[3, 4, 5, 1]

`i`	`anahtar`	Sağa kayanlar	Anahtarın yazıldığı indis	Tur sonu dizi
3	1	5, 4, 3	0	[1, 3, 4, 5]

Tablo 10.18: `{4, 3, 5, 1}` dizisinde eklemeli sıralamanın turları.

Kodda dizilere özgü üç nokta vardır. Birincisi, anahtar `int anahtar = a[i];` ile ayrı bir değişkende saklanır. Kaydırma `a[i]`'nin üzerine yazacağı için bu zorunludur. Üçüncü turun ara adımları bunu açıkça gösterir. Kaydırmalar sırasında dizi `[3, 4, 5, 5]`, `[3, 4, 4, 5]` ve `[3, 3, 4, 5]` hâllerinden geçer. Bu sırada 1 değeri yalnızca `anahtar`'da yaşar. İkincisi, `while` koşulundaki `j >= 0` denetimi önce gelir. `&&` işleci soldan sağa değerlendirilir ve sol taraf yanlışsa sağ tarafa hiç bakmaz. `j` −1 olduğunda `a[j]` okunmaz ve sınır dışı hatası önlenir. Koşulların sırası ters yazılsaydı (`a[j] > anahtar && j >= 0`) anahtar dizinin en küçüğü olduğunda program `a[-1]`'de dururdu. Üçüncüsü, döngü bitince anahtar `a[j + 1]`'e yazılır, çünkü `j` son kaydırılan öğenin bir solunda durur.

En kötü durumda, dizi ters sıralıysa, her anahtar bütün sıralı bölümü kaydırır ve maliyet $O(n^2)$ olur. Dizi zaten neredeyse sıralıysa kaydırma çok azdır ve maliyet $O(n)$'e yaklaşır. Bu yüzden eklemeli sıralama küçük ya da neredeyse sıralı dizilerde iyi bir seçimdir.

10.7.6 İkili arama

İkili arama yalnızca sıralı bir dizide çalışır. Arama aralığının ortasındaki öğeye bakılır. Hedef ortadakinden büyükse sol yarı, küçükse sağ yarı elenir. Aralık her adımda yarıya indiği için n öğeli bir dizide en çok yaklaşık $\log_2 n + 1$ tur gerekir. Java yazımında aralık iki indisle tutulur: `lo` (alt sınır) ve `hi` (üst sınır).

```
public class Ikili {
    public static void main(String[] args) {
        int[] a = {2, 5, 8, 12, 16, 23, 38, 56};
        int hedef = 23, lo = 0, hi = a.length - 1;
        int sonuc = -1;
        while (lo <= hi) {
          int mid = (lo + hi) / 2;
          if (a[mid] == hedef) { sonuc = mid; break; }
          if (a[mid] < hedef) lo = mid + 1;
          else hi = mid - 1;
        }
        System.out.println(sonuc);
    }
}
```

```
5
```

Tur	`lo`	`hi`	`mid`	`a[mid]`	Karar
1	0	7	3	12	12 < 23 → `lo` = 4

Tur	lo	hi	mid	a[mid]	Karar
2	4	7	5	23	bulundu: sonuc = 5

Tablo 10.19: {2, 5, 8, 12, 16, 23, 38, 56} dizisinde 23'ün aranması.

23 iki turda bulunur. Doğrusal arama aynı hedef için altı karşılaştırma yapardı. Küçük dizide fark az görünür. Bir milyon öğeli sıralı dizide ise ikili arama en çok 20 tur ister. Doğrusal arama aynı dizide bir milyona kadar karşılaştırma yapabilir.

Dizilere özgü noktalar şunlardır. Başlangıçta `hi = a.length - 1`'dir; `a.length` geçerli bir indis değildir. Döngü koşulu `lo <= hi`'dir: `lo` ile `hi` eşitken aralıkta hâlâ bir öğe vardır ve ona bakılmalıdır. `mid = (lo + hi) / 2` tam sayı bölmesidir ve aşağı yuvarlar: (0 + 7) / 2 = 3. Son olarak `lo = mid + 1` ve `hi = mid - 1` güncellemelerindeki ±1 zorunludur. `mid`'e zaten bakılmıştır. Güncelleme `lo = mid` diye yazılırsa aralık bazı durumlarda hiç daralmaz ve döngü sonsuza dek döner.

10.7.7 Başarısız ikili arama

Hedef dizide yoksa aralık sonunda boşalır: `lo`, `hi`'yi geçer. O anda `lo`, hedefin sıralı dizide gireceği konumu, yani eklenme noktasını gösterir. Aşağıdaki metot bu bilgiyi `Arrays.binarySearch` ile aynı kurala göre döndürür:

```
import java.util.Arrays;

public class IkiliYok {
    static int ara(int[] a, int x) {
      int lo = 0, hi = a.length - 1;
      while (lo <= hi) {
        int mid = (lo + hi) / 2;
        if (a[mid] == x) return mid;
        if (a[mid] < x) lo = mid + 1; else hi = mid - 1;
      }
      return -(lo + 1);
    }

    public static void main(String[] args) {
        System.out.println(ara(new int[]{2, 4, 6, 8}, 5));
        System.out.println(Arrays.binarySearch(new int[]{2, 4, 6, 8}, 5));
        System.out.println(ara(new int[]{2, 4, 6, 8}, 1) + " " + ara(new int[]{2, 4, 6, 8},
9));
    }
}
```

```
-3
-3
-1 -5
```

Tur	`lo`	`hi`	`mid`	`a[mid]`	Karar
1	0	3	1	4	4 < 5 → `lo` = 2
2	2	3	2	6	6 > 5 → `hi` = 1
—	2	1			`lo > hi`: döngü biter

Tablo 10.20: `{2, 4, 6, 8}` dizisinde 5'in aranması.

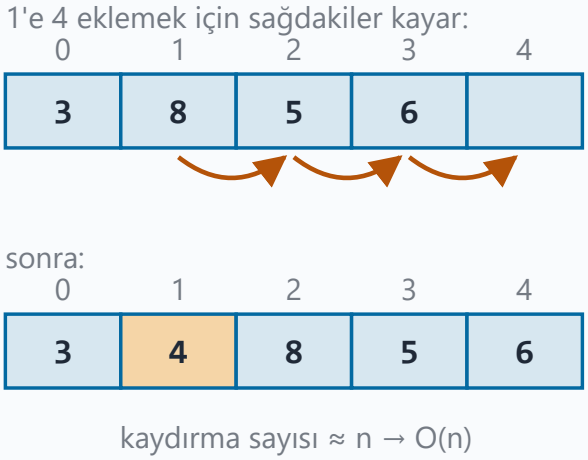
Döngü bittiğinde `lo` = 2'dir ve metot −(2 + 1) = −3 döndürür. Programın ikinci satırı, `Arrays.binarySearch`'ün aynı dizi ve aynı hedef için de −3 döndürdüğünü gösterir. Negatif sonuç "yok" demektir. Eklenme noktası sonuçtan geri hesaplanır: −(−3) − 1 = 2. Gerçekten de 5, 4 ile 6 arasına, 2 numaralı konuma girerdi. Üçüncü satır iki uç durumu sınar. Dizinin en küçüğünden küçük olan 1 için eklenme noktası 0, dönüş −1'dir. En büyüğünden büyük olan 9 için eklenme noktası 4, dönüş −5'tir.

10.7.8 Temel karmaşıklıklar

Bu notta görülen dizi işlemlerinin maliyetleri, dizinin yapısından doğrudan çıkar. Tablo 10.21 bunları bir arada verir.

İşlem	Zaman	Gerekçe
İndisle erişim	O(1)	Adres tek bir çarpma ve toplamayla hesaplanır
Doğrusal arama	O(n)	En kötü durumda her öğeye bakılır
Sona kadar gezinme	O(n)	Her öğe bir kez ziyaret edilir
Ortaya ekleme / silme	O(n)	Sonraki öğeler birer hücre kaydırılır
İkili arama (sıralı dizi)	O(log n)	Aralık her adımda yarıya iner
İki iç içe tam gezinme	$O(n^2)$	Her öğe için bütün diziye bakılır

Tablo 10.21: Temel dizi işlemlerinin zaman karmaşıklıkları (n: öğe sayısı).



Şekil 10.17: 4'ü 1 numaralı konuma eklemek için 8, 5 ve 6 birer hücre sağa kayar. Kaydırma sayısı öğe sayısıyla orantılıdır: O(n).

Şekil 10.17 ortaya eklemenin maliyetini somutlaştırır. Dizinin sonunda boş bir hücre bulunduğu varsayılmıştır. Ekleme noktasından sonraki her öğe bir sağa taşınır; kaydırma sağdan sola yapılır, yoksa öğeler birbirinin üzerine yazılır. Kabarcık ve eklemeli sıralama iki iç içe döngü kullandığı için $O(n^2)$ satırına girer. `Arrays.sort` ise O(n log n) maliyetle bu satırların arasında yer alır.

10.8 İki boyutlu diziler

Bir sınıftaki öğrencilerin üç sınavdaki notları, bir resmin piksel renkleri ya da bir oyun tahtasının kareleri satırlar ve sütunlar hâlinde düzenlenir. Bu tür veriler için iki indisli bir yapı daha doğaldır: biri satırı, öteki sütunu seçer.

10.8.1 Matris bildirimi ve erişim

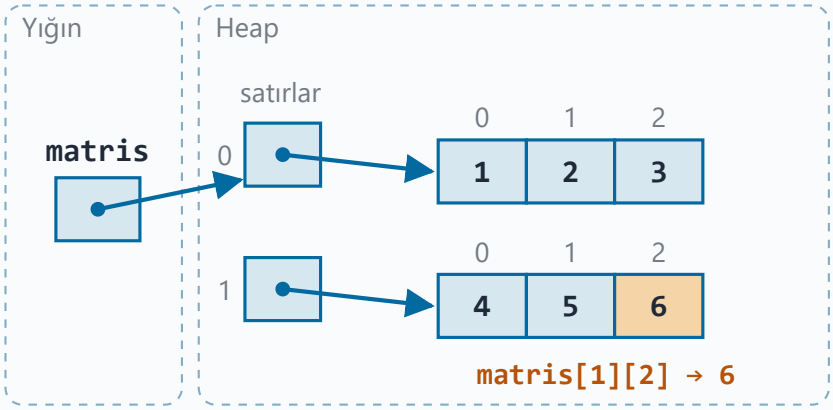
> **Tanım 10.6**
>
> **İki boyutlu dizi** (*two-dimensional array*), öğelerine iki indisle erişilen dizidir. Satır ve sütunlardan oluşan bu yapıya **matris** (*matrix*) da denir. Java'da iki boyutlu dizi, **dizilerin dizisidir**: dış dizinin her hücresi bir satır dizisinin referansını tutar.

Tipi `int[][]` diye yazılır. İlklendiricide her satır kendi süslü parantezi içinde verilir. Boş bir matris `new int[2][3]` ile oluşturulur: 2 satır, her satırda 3 sütun, bütün hücreler 0.

```
public class Matris {
    public static void main(String[] args) {
        int[][] matris = {
            {1, 2, 3},
            {4, 5, 6}
        };
        int deger = matris[1][2]; // 6
        System.out.println(deger + " " + matris.length + " " + matris[0].length);
        int[][] bos = new int[2][3];
        System.out.println(bos[1][2]);
    }
}
```

```
6 2 3
0
```

Erişimde ilk indis satırı, ikinci indis sütunu seçer. `matris[1][2]`, 1 numaralı satırın 2 numaralı sütunudur, yani 6. Okuma soldan sağa yapılır: önce `matris[1]` ile satır dizisi bulunur, sonra o dizinin 2 numaralı hücresine gidilir.



Şekil 10.18: `int[][] matris = {{1, 2, 3}, {4, 5, 6}};` bellekte. `matris` değişkeni iki hücreli dış diziyi gösterir. Dış dizinin her hücresi üç hücreli bir satır dizisini gösterir. Vurgulu hücre `matris[1][2]` = 6'dır.

Şekil 10.18 "dizilerin dizisi" fikrinin bellekteki karşılığını gösterir. Bu yapıdan iki önemli sonuç çıkar:

- `matris.length` dış dizinin uzunluğudur, yani **satır sayısıdır** (2). `matris[0].length` ise 0 numaralı satırın uzunluğudur, yani o satırdaki **sütun sayısıdır** (3).
- `matris[1]` tek başına geçerli bir ifadedir ve bir `int[]`, yani bütün bir satırdır. `Arrays.toString(matris[1])` o satırı `[4, 5, 6]` diye yazar.

10.8.2 Düzensiz diziler

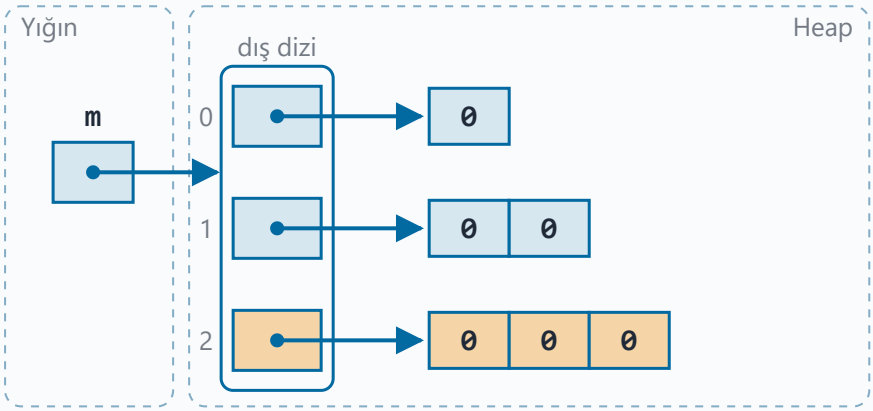
Satırlar ayrı nesneler olduğu için hepsinin aynı uzunlukta olması gerekmez. Satırları farklı uzunlukta olan iki boyutlu diziye **düzensiz dizi** (*jagged array*) denir. Böyle bir dizi oluşturmak için `new` ifadesinde yalnızca satır sayısı verilir; satırlar sonradan tek tek oluşturulur:

```
public class Duzensiz {
    public static void main(String[] args) {
        int[][] m = new int[3][];
        m[0] = new int[1];
        m[1] = new int[2];
        m[2] = new int[3];
        for (int i = 0; i < m.length; i++)
            System.out.print(m[i].length + " ");
        System.out.println();
    }
}
```

```
1 2 3
```

Satır	`m[0]`	`m[1]`	`m[2]`	Çıktı
`m = new int[3][];`	null	null	null	
`m[0] = new int[1];`	[0]	null	null	
`m[1] = new int[2];`	[0]	[0, 0]	null	
`m[2] = new int[3];`	[0]	[0, 0]	[0, 0, 0]	
döngü, `i` = 0, 1, 2	aynı	aynı	aynı	`1 2 3`

Tablo 10.22: `Duzensiz` programının izleme tablosu.



Şekil 10.19: Üç satır da oluşturulduktan sonra düzensiz dizi. Dış dizinin hücreleri uzunlukları 1, 2 ve 3 olan üç ayrı satır nesnesini gösterir.

`new int[3][]` yalnızca dış diziyi oluşturur. Dış dizinin hücreleri referans tuttuğu için varsayılan değerleri `null`'dır. Satırlar atanana kadar `m[1].length` gibi bir ifade çalıştırılırsa program `NullPointerException` ile durur. Çalışma sorularındaki bir soru tam olarak bu durumu sınar; JDK 21'in iletisi şöyledir:

```
> java Q20
6
Exception in thread "main" java.lang.NullPointerException: Cannot read the array length
because "<local1>[1]" is null
        at Q20.main(Q20.java:7)
```

İleti, "`<local1>[1]` null olduğu için dizi uzunluğu okunamıyor" der. `<local1>`, `m` değişkeninin derlenmiş koddaki adıdır. Kaynak, `javac -g` seçeneğiyle derlenirse ileti değişkenin gerçek adını gösterir:

```
> javac -g Q20.java
> java Q20
6
Exception in thread "main" java.lang.NullPointerException: Cannot read the array length
because "m[1]" is null
        at Q20.main(Q20.java:7)
```

Düzensiz diziler üçgen biçimli veriler için bellek kazandırır: Pascal üçgeni ya da bir mesafe tablosunun yarısı gibi.

10.8.3 Sığ kopya tuzağı

İki boyutlu bir dizide `clone` yalnızca dış diziyi kopyalar. Dış dizinin hücrelerinde satırların referansları vardır; kopyalanan da bu referanslardır. Sonuçta iki ayrı dış dizi, **aynı** satır nesnelerini gösterir.

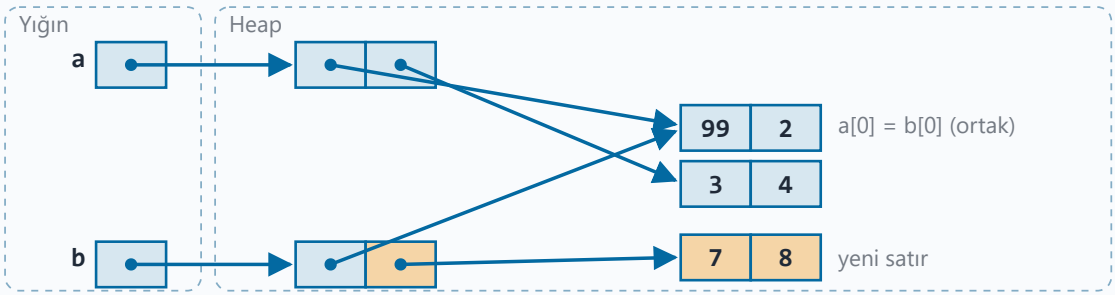
```
import java.util.Arrays;

public class Sig {
    public static void main(String[] args) {
        int[][] a = {{1, 2}, {3, 4}};
        int[][] b = a.clone();
        b[0][0] = 99;
        b[1] = new int[]{7, 8};
        System.out.println(Arrays.deepToString(a));
        System.out.println(Arrays.deepToString(b));
    }
}
```

```
[[99, 2], [3, 4]]
[[99, 2], [7, 8]]
```

Satır	`a`	`b`	Etki
`int[][] a = {{1, 2}, {3, 4}};`	[satır 1, satır 2]	—	Satır 1 = [1, 2], satır 2 = [3, 4]
`int[][] b = a.clone();`	[satır 1, satır 2]	[satır 1, satır 2]	Yeni dış dizi, satırlar ortak
`b[0][0] = 99;`	aynı	aynı	Satır 1 = [99, 2]; `a[0][0]` de 99
`b[1] = new int[]{7, 8};`	[satır 1, satır 2]	[satır 1, satır 3]	Yalnızca `b`'nin hücresi değişir

Tablo 10.23: `Sig` programının izleme tablosu. Satır 1 ve satır 2, başlangıçtaki iki satır nesnesidir; satır 3 sonradan oluşturulur.



Şekil 10.20: `b[1]`'e yeni satır bağlandıktan sonra bellek. `a[0]` ve `b[0]` aynı satırı (99, 2) gösterir. `a[1]` eski satırda (3, 4) kalmıştır; `b[1]` ise yeni satırı (7, 8) gösterir.

İki satırın farklı sonuç vermesi tuzağın özüdür. `b[0][0] = 99`, ortak satır nesnesinin içine yazar; bu yüzden `a`'dan da görülür. `b[1] = new int[]{7, 8}` ise bir satırın içine değil, `b`'nin dış dizisindeki hücreye yazar. O hücre yalnızca `b`'ye aittir ve `a` etkilenmez. Gerçek, yani **derin** bir kopya için her satır ayrıca kopyalanır:

```
import java.util.Arrays;

public class Derin {
    public static void main(String[] args) {
        int[][] a = {{1, 2}, {3, 4}};
        int[][] b = new int[a.length][];
        for (int i = 0; i < a.length; i++)
            b[i] = a[i].clone();
        b[0][0] = 99;
        System.out.println(Arrays.deepToString(a));
        System.out.println(Arrays.deepToString(b));
    }
}
```

```
[[1, 2], [3, 4]]
[[99, 2], [3, 4]]
```

Bu kez `b[0][0] = 99` yalnızca `b`'nin kendi satırını değiştirir; `a` `[[1, 2], [3, 4]]` olarak kalır. Döngüdeki `b[i] = a[i].clone()` her satır için ayrı bir nesne oluşturur.

10.8.4 Matriste satır satır gezinme

Bir matrisin bütün hücrelerini gezmek için iki iç içe döngü kullanılır. Dış döngü satırları, iç döngü o satırın sütunlarında gezinir:

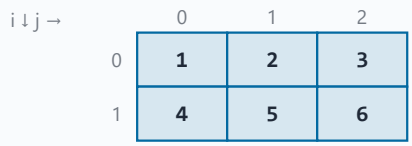
```
public class Gezinme {
    public static void main(String[] args) {
        int[][] matris = {{1, 2, 3}, {4, 5, 6}};
        for (int i = 0; i < matris.length; i++) {
          for (int j = 0; j < matris[i].length; j++) {
            System.out.print(matris[i][j] + " ");
          }
          System.out.println();
        }
    }
}
```

```
1 2 3
4 5 6
```

İç döngünün sınırı `matris[i].length`'tir; yani her satır kendi uzunluğu kadar gezilir. Bu yazım düzensiz dizilerde de doğru çalışır. Hücreler `print` ile aynı satıra, aralarında boşlukla yazılır. Bir satır bittiğinde iç döngüden çıkılır ve `println()` alt satıra geçer.

Adım	`i`	`j`	`matris[i][j]`	Konsoldaki satır
1	0	0	1	`1`
2	0	1	2	`1 2`
3	0	2	3	`1 2 3` ⏎
4	1	0	4	`4`
5	1	1	5	`4 5`
6	1	2	6	`4 5 6` ⏎

Tablo 10.24: `Gezinme` programında hücrelerin ziyaret sırası.



Şekil 10.21: 2 × 3 matris. `i` satırı aşağı doğru, `j` sütunu sağa doğru seçer. Satır satır gezinmede önce 0. satırın üç hücresi, sonra 1. satırın üç hücresi ziyaret edilir.

Toplam 2 × 3 = 6 hücre gezilir. Genel olarak r satır ve c sütunlu bir matriste maliyet O(r · c)'dir. Kare bir n × n matriste bu $O(n^2)$ olur. Satır toplamı, sütun toplamı ya da en büyük hücreyi bulmak gibi işlemlerin hepsi bu iskeletin üzerine kurulur. Örneğin her satırın toplamı için iç döngüden önce `int toplam = 0;` yazılır ve iç döngü bitince toplam yazdırılır.

10.8.5 Matris transpozu

Bir matrisin **transpozu**, satırların sütun, sütunların satır olduğu matristir. r × c boyutlu bir matrisin transpozu c × r boyutludur ve `t[j][i] = matris[i][j]` kuralıyla doldurulur.

```
import java.util.Arrays;

public class Transpoz {
    public static void main(String[] args) {
        int[][] matris = {{1, 2, 3}, {4, 5, 6}};
        int satirSayisi = matris.length;
        int sutunSayisi = matris[0].length;
        int[][] t = new int[sutunSayisi][satirSayisi];
        for (int i = 0; i < satirSayisi; i++)
            for (int j = 0; j < sutunSayisi; j++)
                t[j][i] = matris[i][j];
        System.out.println(Arrays.deepToString(t));
    }
}
```

```
[[1, 4], [2, 5], [3, 6]]
```

`i`	`j`	Atama	`t` (adım sonu)
0	0	`t[0][0] = matris[0][0]` = 1	[[1, 0], [0, 0], [0, 0]]
0	1	`t[1][0] = matris[0][1]` = 2	[[1, 0], [2, 0], [0, 0]]
0	2	`t[2][0] = matris[0][2]` = 3	[[1, 0], [2, 0], [3, 0]]
1	0	`t[0][1] = matris[1][0]` = 4	[[1, 4], [2, 0], [3, 0]]
1	1	`t[1][1] = matris[1][1]` = 5	[[1, 4], [2, 5], [3, 0]]
1	2	`t[2][1] = matris[1][2]` = 6	[[1, 4], [2, 5], [3, 6]]

Tablo 10.25: `Transpoz` programının izleme tablosu.



Şekil 10.22: 2 × 3 boyutlu `matris` ve 3 × 2 boyutlu transpozu `t`. Matrisin 0. satırı (1, 2, 3), `t`'nin 0. sütunu olmuştur.

Kodda boyutlar baştan okunur: `satirSayisi = matris.length` (2), `sutunSayisi = matris[0].length` (3). Hedef dizi ters boyutla oluşturulur: `new int[sutunSayisi][satirSayisi]`, yani 3 × 2. `matris[0].length` ile sütun sayısını almak, bütün satırların aynı uzunlukta olduğunu varsayar; transpoz zaten yalnızca düzenli matrisler için anlamlıdır. Her hücre bir kez kopyalanır: O(r · c) zaman ve hedef matris için r · c hücrelik ek bellek.

10.8.6 Matris çarpımı

İki matrisin çarpımı C = A × B şöyle tanımlanır: C'nin `i`. satır `j`. sütunundaki hücresi, A'nın `i`. satırı ile B'nin `j`. sütunundaki karşılıklı öğelerin çarpımlarının toplamıdır. A'nın sütun sayısı B'nin satır sayısına eşit olmalıdır.

> C[i][j] = A[i][0] · B[0][j] + A[i][1] · B[1][j] + … + A[i][m−1] · B[m−1][j]

```
import java.util.Arrays;

public class Carpim {
    public static void main(String[] args) {
        int[][] A = {{1, 2}, {3, 4}};
        int[][] B = {{5, 6}, {7, 8}};
        int[][] C = new int[2][2];
        for (int i = 0; i < 2; i++)
          for (int j = 0; j < 2; j++)
            for (int k = 0; k < 2; k++)
              C[i][j] += A[i][k] * B[k][j];
        System.out.println(Arrays.deepToString(C));
    }
}
```

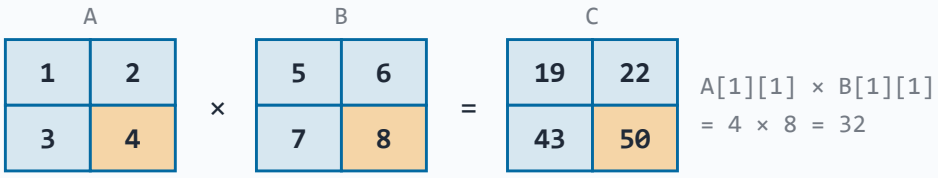
```
[[19, 22], [43, 50]]
```

Örnek 10.3

A = [[1, 2], [3, 4]] ve B = [[5, 6], [7, 8]] için C'nin dört hücresini adım adım hesaplayalım. Her hücre iki çarpımın toplamıdır, çünkü A'nın sütun sayısı 2'dir.

- C[0][0] = A[0][0] · B[0][0] + A[0][1] · B[1][0] = 1 · 5 + 2 · 7 = 5 + 14 = 19
- C[0][1] = A[0][0] · B[0][1] + A[0][1] · B[1][1] = 1 · 6 + 2 · 8 = 6 + 16 = 22
- C[1][0] = A[1][0] · B[0][0] + A[1][1] · B[1][0] = 3 · 5 + 4 · 7 = 15 + 28 = 43
- C[1][1] = A[1][0] · B[0][1] + A[1][1] · B[1][1] = 3 · 6 + 4 · 8 = 18 + 32 = 50

Sonuç [[19, 22], [43, 50]]'dir ve JDK 21 çıktısıyla aynıdır.

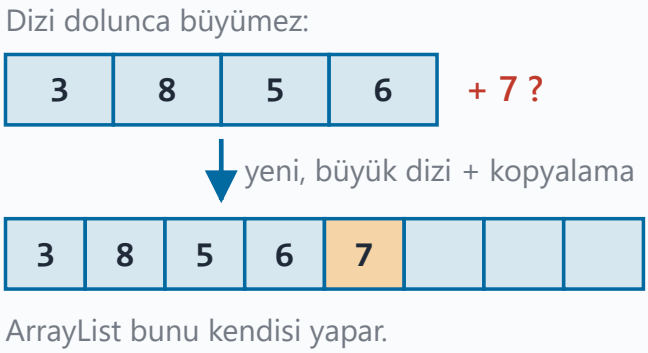


Şekil 10.23: Son hücrenin son adımı: C[1][1] += A[1][1] × B[1][1] = 4 × 8 = 32, böylece 18 + 32 = 50 olur. A'nın 1. satırı ve B'nin 1. sütunu kullanılmıştır.

Kodda üç iç içe döngü vardır. `i` ve `j` hesaplanacak hücreyi seçer; `k` toplamın terimleri üzerinde gezinir. `C[i][j] += A[i][k] * B[k][j]` deyimi her terimi hücreye ekler. `C` `new int[2][2]` ile oluşturulduğu için hücreler 0 ile başlar; ayrıca sıfırlamaya gerek yoktur. Kare n × n matrislerde her hücre için n çarpma, toplam n^2 hücre için n^3 çarpma yapılır: $O(n^3)$. 100 × 100 matrisler için bu bir milyon çarpmadır.

10.9 Dizi mi liste mi?

Dizinin uzunluğu sabittir. Öğe sayısı program çalışırken artıp azalıyorsa bu sınır can sıkar: her eklemede yeni ve daha büyük bir dizi oluşturmak, eski öğeleri oraya kopyalamak gerekir. Java'nın koleksiyon kütüphanesindeki `ArrayList` sınıfı bu işi kendisi yapar.



Şekil 10.24: Dolu bir diziye 7 eklenemez, çünkü dizi büyümez. Yeni ve daha büyük bir dizi oluşturulur, öğeler kopyalanır ve 7 boş hücreye yazılır. `ArrayList` bu adımları kendiliğinden yapar.

```
import java.util.ArrayList;

public class Liste {
    public static void main(String[] args) {
        ArrayList<Integer> liste = new ArrayList<>();
        liste.add(3);
        liste.add(8);
        liste.add(5);
        liste.add(7);
        System.out.println(liste + " " + liste.size() + " " + liste.get(1));
    }
}
```

```
[3, 8, 5, 7] 4 8
```

`ArrayList` içeride yine bir dizi kullanır. Dizi dolduğunda daha büyük bir dizi açar ve öğeleri oraya taşır (Şekil 10.24). Programcı yalnızca `add`, `get` ve `size` gibi metotları çağırır. Erişim için köşeli parantez yerine `liste.get(1)`, uzunluk için `length` yerine `liste.size()` yazılır.

Bu kolaylığın bir bedeli vardır. `int[]` sayıları hücrelerde doğrudan saklar. `ArrayList<Integer>` ise yalnızca nesne tutabilir; her sayı bir `Integer` nesnesinin içine konur ve listede bu nesnelerin referansları durur. Bu, daha fazla bellek ve bir miktar daha yavaş erişim demektir. Seçim için kaba bir kural verilebilir:

- Öğe sayısı baştan belli ve sabitse, örneğin haftanın günleri, bir matrisin hücreleri ya da bir sayaç tablosu, dizi yeterli ve hızlıdır.
- Öğe sayısı çalışma sırasında değişecekse, örneğin kullanıcının girdiği kayıtlar, `ArrayList` gibi bir koleksiyon daha kullanışlıdır.

Son karar kullanım biçimine, kodun okunabilirliğine ve gerekirse ölçülen performansa göre verilir. Bu notta öğrenilen gezinme, arama ve sıralama yöntemleri listelerde de aynen geçerlidir.

10.10 Yaygın hatalar

Dizilerle ilgili hataların çoğu dört gruba girer. Bu kesim her birini çalıştırılmış bir örnekle verir.



Şekil 10.25: İki yaygın hata. Solda bir tur fazla: `i <= a.length` son turda var olmayan `a[5]` 'e erişir. Sağda referans kopyası: `b = a` sonrasında `b[0] = 9` , tek ortak diziyi değiştirir ve `a[0]` da 9 olur.

Sık yapılan hata

Bir tur fazla. `i <= a.length` koşulu son turda `a[length]` 'e erişir ve `ArrayIndexOutOfBoundsException` fırlatılır. Doğrusu `i < a.length` 'tir. Aynı hatanın sessiz bir kardeşi de vardır: `i < a.length - 1` yazmak son öğeyi atlar ve hiçbir ileti vermez.

Sık yapılan hata

Boş dizide ilk öğe. `int min = a[0];` boş dizide "Index 0 out of bounds for length 0" iletisiyle durur. En küçük, en büyük ya da ortanca hesaplanmadan önce `a.length == 0` durumu ayrıca ele alınır.

Sık yapılan hata

Referans kopyası. `int[] b = a;` yeni bir dizi değildir; `b[0] = 9` `a` 'yı da değiştirir. Bağımsız kopya için `a.clone()` ya da `Arrays.copyOf` kullanılır. İki boyutlu dizilerde bunlar da yetmez; her satır ayrıca kopyalanır.

Sık yapılan hata

Düzensiz matriste sabit sütun sayısı. İç döngüde her satır için `matris[0].length` kullanmak, bütün satırların ilk satır kadar uzun olduğunu varsayar. Aşağıdaki programda ikinci satır yalnızca bir öğelidir:

```
public class DuzensizHata {
    public static void main(String[] args) {
        int[][] m = {{1, 2, 3}, {4}};
        int toplam = 0;
        for (int i = 0; i < m.length; i++)
            for (int j = 0; j < m[0].length; j++)
                toplam += m[i][j];
        System.out.println(toplam);
    }
}
```

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 1 out of
bounds for length 1
        at DuzensizHata.main(DuzensizHata.java:7)
```

`i` = 1 iken `j` 1 olur ve uzunluğu 1 olan satırda 1 numaralı hücre aranır. İç döngü sınırı `m[i].length` yapılınca program 10 yazar.

Bu dört hatanın ortak noktası, derleyicinin hiçbirini yakalamamasıdır. Hepsi ya çalışma sırasında bir istisnayla ya da hiç iletisiz yanlış bir sonuçla ortaya çıkar. Bu yüzden dizi kodu yazılırken üç sınır durumu her zaman denenir: boş dizi, tek öğeli dizi ve hedefin ilk ya da son konumda olduğu dizi.

10.11 Özet

- **Dizi**, aynı tipten sabit sayıda öğeyi art arda hücrelerde tutan bir nesnedir. İndisler 0'dan `length - 1`'e gider; `length` bir alandır ve oluşturulduktan sonra değişmez.
- Dizi değişkeni yığında durur ve heap'teki dizi nesnesinin **referansını** tutar. `new` ile oluşturulan hücreler varsayılan değeri alır: sayılarda 0, `boolean`'da `false`, nesnelerde `null`. İlklendirici `{…}` uzunluğu değer sayısından çıkarır.
- `a[i]` okuma ve yazma O(1) zamanda yapılır, çünkü adres *başlangıç + i × hücre boyu* formülüyle hesaplanır. Geçersiz indis `ArrayIndexOutOfBoundsException` ile programı durdurur; en yaygın nedeni `<=` ile yazılan döngü koşuludur.
- Klasik `for` indis gerektiğinde, gelişmiş `for` yalnızca okuma yapılacağında kullanılır. Gelişmiş `for`'daki döngü değişkeni öğenin kopyasıdır; ona yapılan atama diziyi değiştirmez.
- Toplam ve ortalama bir biriktiriciyle, en küçük ve en büyük dizinin ilk öğesiyle başlatılan adaylarla bulunur. Ters çevirme iki işaretçiyle yerinde yapılır. Frekans saymada değer indis olarak kullanılır. Eratosthenes eleği bir `boolean` dizisiyle asal olmayanları eler.
- `b = a` yalnızca referansı kopyalar. `clone`, `Arrays.copyOf` ve `System.arraycopy` öğeleri kopyalar, ama sığ kopya yapar. `==` ve `equals` referansı, `Arrays.equals` içeriği karşılaştırır. Metoda giden dizi, çağıranın dizisiyle aynı nesnedir.

- Doğrusal arama O(n), sıralı dizide ikili arama O(log n) zamandadır. `Arrays.sort` diziyi yerinde sıralar; `Arrays.binarySearch` bulamazsa `-(eklenme noktası) - 1` döndürür. Kabarcık ve eklemeli sıralama $O(n^2)$'dir.
- `int[][]` dizilerin dizisidir. `matris.length` satır, `matris[i].length` o satırın sütun sayısıdır. Satırlar farklı uzunlukta olabilir (düzensiz dizi). `clone` yalnızca dış diziyi kopyalar. Gezinme iki, matris çarpımı üç iç içe döngüyle yapılır.

10.12 Kavram listesi

dizi *array* · öğe *element* · hücre *cell* · indis *index* · uzunluk *length* · referans *reference* · heap *heap* · yığın *stack* · varsayılan değer *default value* · ilklendirici *array initializer* · null *null reference* · sınır dışı *out of bounds* · istisna *exception* · gezinme *traversal* · gelişmiş for *enhanced for loop* · ters çevirme *reversing* · işaretçi *pointer* · takas *swap* · frekans *frequency* · Eratosthenes eleği *Sieve of Eratosthenes* · kopya *copy* · sığ kopya *shallow copy* · doğrusal arama *linear search* · ikili arama *binary search* · ortanca *median* · kabarcık sıralaması *bubble sort* · eklemeli sıralama *insertion sort* · zaman karmaşıklığı *time complexity* · iki boyutlu dizi *two-dimensional array* · çok boyutlu dizi *multidimensional array* · matris *matrix* · düzensiz dizi *jagged array*

10.13 Kendini Dene

1. Aşağıdaki programın çıktısı nedir?

```
import java.util.Arrays;

public class KD1 {
    public static void main(String[] args) {
        int[] a = new int[4];
        a[a.length - 1] = a.length;
        a[a[3] - 3] = 9;
        System.out.println(Arrays.toString(a));
    }
}
```

Cevap

```
[0, 9, 0, 4]
```

Dizi dört sıfırla başlar. `a[a.length - 1] = a.length` son hücreye, yani `a[3]`'e 4 yazar. Sonraki satırda iç ifade önce hesaplanır: `a[3] - 3` = 4 − 3 = 1. Böylece `a[1]`'e 9 yazılır.

2. Aşağıdaki programda `sifirla` metodu çağrıldıktan sonra `a` ne olur?

```
import java.util.Arrays;

public class KD2 {
    static void sifirla(int[] d) {
        d[0] = 0;
        d = new int[]{5, 5, 5};
        d[1] = 0;
    }

    public static void main(String[] args) {
        int[] a = {1, 2, 3};
        sifirla(a);
        System.out.println(Arrays.toString(a));
    }
}
```

Cevap

```
[0, 2, 3]
```

Metoda `a`'nın referansı gider. `d[0] = 0` ortak diziye yazar ve `a[0]` 0 olur. `d = new int[]{5, 5, 5}` ise yalnızca metodun yerel değişkenini yeni bir diziye bağlar. Bundan sonraki `d[1] = 0` bu yeni diziyi değiştirir; `a` etkilenmez.

3. Aşağıdaki programın çıktısı nedir? Üç değerin her biri neden öyledir?

```java
import java.util.Arrays;

public class KD3 {
    public static void main(String[] args) {
        int[] p = {4, 5};
        int[] q = p.clone();
        int[] r = p;
        System.out.println((p == q) + " " + (p == r) + " " + Arrays.equals(p, q));
    }
}
```

Cevap

```
false true true
```

`p == q` yanlıştır, çünkü `clone` yeni bir nesne oluşturur. `p == r` doğrudur, çünkü `r = p` aynı referansı kopyalar. `Arrays.equals(p, q)` doğrudur, çünkü iki dizinin uzunluğu ve öğeleri aynıdır.

4. `Arrays.binarySearch` önce sıralanmamış, sonra sıralanmış bir dizide 1'i arıyor. Çıktı nedir ve ilk sonuç neden yanlıştır?

```java
import java.util.Arrays;

public class KD4 {
    public static void main(String[] args) {
        int[] a = {5, 1, 9, 3, 7};
        System.out.println(Arrays.binarySearch(a, 1));
        Arrays.sort(a);
        System.out.println(Arrays.binarySearch(a, 1));
    }
}
```

Cevap

```
-1
0
```

İlk aramada aralık 0…4'tür ve ortadaki öğe `a[2]` = 9'dur. 1 < 9 olduğu için sağ yarı elenir. Sonra `a[0]` = 5'e bakılır; 1 < 5 olduğu için yine sola gidilir ve aralık boşalır. Metot −1 döndürür, oysa 1 dizide, 1 numaralı konumdadır. Metot dizinin sıralı olduğunu varsaydığı için yanlış yarıyı elemiştir. Sıralamadan sonra dizi `[1, 3, 5, 7, 9]` olur ve 1 doğru biçimde 0 numaralı konumda bulunur.

5. Aşağıdaki program üçgen biçimli bir düzensiz dizi oluşturuyor. `t[3].length` ve toplam hücre sayısı kaçtır?

```
public class KD5 {
    public static void main(String[] args) {
        int[][] t = new int[4][];
        int hucre = 0;
        for (int i = 0; i < t.length; i++) {
            t[i] = new int[i + 1];
            hucre += t[i].length;
        }
        System.out.println(t[3].length + " " + hucre);
    }
}
```

Cevap

```
4 10
```

`i` numaralı satır `i + 1` hücrelidir: 1, 2, 3 ve 4. Son satırın uzunluğu 4, toplam hücre sayısı 1 + 2 + 3 + 4 = 10'dur. Düzenli bir 4 × 4 matris 16 hücre isterdi.

6. Ters sıralı `{4, 3, 2, 1}` dizisi kabarcık sıralamasıyla sıralanıyor. Kaç karşılaştırma ve kaç takas

yapılır?

```
import java.util.Arrays;

public class KD6 {
    public static void main(String[] args) {
        int[] a = {4, 3, 2, 1};
        int kars = 0, takas = 0;
        for (int gecis = 0; gecis < a.length - 1; gecis++)
            for (int j = 0; j < a.length - 1 - gecis; j++) {
                kars++;
                if (a[j] > a[j + 1]) {
                    int g = a[j]; a[j] = a[j + 1]; a[j + 1] = g;
                    takas++;
                }
            }
        System.out.println(Arrays.toString(a) + " " + kars + " " + takas);
    }
}
```

Cevap

```
[1, 2, 3, 4] 6 6
```

Karşılaştırma sayısı dizinin içeriğine bağlı değildir: 3 + 2 + 1 = 6. Ters sıralı dizide her karşılaştırma bozuk bir çift bulur ve takasla sonuçlanır; takas sayısı da 6'dır. Bu, n = 4 için en kötü durumdur: $n(n-1)/2 = 6$.

7. Aşağıdaki program bir dizide en sık geçen rakamı arıyor. Çıktı nedir? `say` dizisi neden 10 hücrelidir?

```
public class KD7 {
    public static void main(String[] args) {
        int[] a = {2, 7, 7, 3, 7};
        int[] say = new int[10];
        for (int x : a) say[x]++;
        int enCok = 0;
        for (int d = 1; d < say.length; d++)
            if (say[d] > say[enCok]) enCok = d;
        System.out.println(enCok + " " + say[enCok]);
    }
}
```

Cevap

```
7 3
```

Değerler indis olarak kullanılır: `say[x]++`. Rakamlar 0…9 arasında olduğu için 10 hücre gerekir. Sayma sonunda `say[2]` = 1, `say[3]` = 1, `say[7]` = 3'tür. İkinci döngü en büyük sayacın indisini arar; 7 üç kez geçtiği için sonuç `7 3` 'tür.

8. Aşağıdaki program 3 × 3 bir matrisi dolduruyor. `m[2][1]` ve `m[1][2]` kaçtır?

```
public class KD8 {
    public static void main(String[] args) {
        int[][] m = new int[3][3];
        for (int i = 0; i < 3; i++)
            for (int j = 0; j < 3; j++)
                m[i][j] = i * 3 + j;
        System.out.println(m[2][1] + " " + m[1][2]);
    }
}
```

Cevap

```
7 5
```

Her hücreye `i * 3 + j` yazılır; hücreler satır satır 0'dan 8'e kadar numaralanmış olur. `m[2][1]` = 2 · 3 + 1 = 7, `m[1][2]` = 1 · 3 + 2 = 5. İlk indis satır, ikinci indis sütun olduğu için iki ifade farklı hücreleri gösterir.

9. Bir programcı, uzunluğu `n` olan bir diziyi sondan başa yazdırmak için `for (int i = a.length;`

`i >= 0; i--)` döngüsünü yazıyor. Hangi hata oluşur ve döngü nasıl düzeltilir?

Cevap

İlk turda `i` = `a.length`'tir ve `a[a.length]` okunmaya çalışılır. Üç öğeli bir diziyle çalıştırılan program hiçbir şey yazdırmadan durur:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 3 out
of bounds for length 3
        at KD9.main(KD9.java:5)
```

Son geçerli indis `a.length - 1` olduğu için doğru başlangıç `int i = a.length - 1`'dir. Koşul `i >= 0` doğrudur, çünkü 0 numaralı hücre de yazdırılmalıdır.

10.14 İleri okuma

- Oracle, *The Java Tutorials*, "Arrays": dizi bildirimi, oluşturma, çok boyutlu diziler, kopyalama ve `java.util.Arrays` metotlarını kısa, çalıştırılabilir örneklerle anlatır.
- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), 1.4 Arrays: deste karıştırma, kupon toplayıcı problemi ve Eratosthenes eleği gibi dizi uygulamalarını içerir; kitabın sitesinde (introcs.cs.princeton.edu) program kodları ve alıştırmalar vardır.
- A. B. Downey, C. Mayfield, *Think Java* (2. baskı, O'Reilly, 2019), Bölüm 7 "Arrays and References": referans kavramını, dizilerin metotlara aktarılmasını ve gezinme kalıplarını yeni başlayanlar için sade bir dille anlatır; ücretsiz çevrim içi sürümü vardır.
- Oracle, *The Java Language Specification, Java SE 21 Edition*, Bölüm 10 "Arrays": dizi tiplerinin, ilklendiricilerin, varsayılan değerlerin ve `clone` davranışının kesin tanımını verir.

10.15 Kaynaklar

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11. baskı, Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13. baskı, Pearson, 2024.
3. Oracle, *The Java Tutorials*: Arrays.
4. Oracle, *Java SE 21 API*: java.util.Arrays; *The Java Language Specification, Java SE 21 Edition*: Chapter 10. Arrays.
5. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 1.4 Arrays.
6. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
7. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019 (ücretsiz çevrim içi sürüm).

KISIM III · BÖLÜM 11

Dizgiler

Programların işlediği verinin büyük bölümü metindir: öğrenci adları, e-posta adresleri, kullanıcının klavyeden yazdığı cümleler, ekrana basılan raporlar. Java'da metin `String` sınıfının nesneleriyle temsil edilir. Bu not dizgilerin bellekte nasıl tutulduğunu, neden değiştirilemediğini ve en sık kullanılan metotlarını anlatır. İndislerle karakterlere erişim, alt dizgi alma, arama, bölme ve değiştirme işlemleri tek tek ele alınır. Ardından dizgilerin karşılaştırılması gelir: `==` ile `equals` arasındaki fark, `compareTo` ile sözlüksel sıra ve Türkçe harflerin yol açtığı tuzaklar. Klavyeden metin okuma, `StringBuilder` ile verimli birleştirme ve `printf` ile biçimli çıktı da notun konuları arasındadır. Son kesimdeki uygulamalar (ters çevirme, palindrom, sesli harf ve kelime sayma, harf frekansı, Sezar şifresi) bütün bu araçları bir araya getirir. Her program JDK 21 ile derlenip çalıştırılmıştır; verilen çıktılar programların gerçek çıktılarıdır.

Bu bölümün kodu: `github.com/sercankulcu/computer-programming-java/Bolum11/src`

ÖĞRENME ÇIKTILARI

Temsil

`String` ile `char` arasındaki farkı, dizgi değişkeninin bir referans tuttuğunu ve dizgilerin değiştirilemez olduğunu açıklamak; boş dizgi ile `null`'ı ayırt etmek.

İşleme

İndis ve `charAt` ile karakterlere erişmek; `length`, `indexOf`, `substring`, `split`, `replace` gibi metotları doğru sınırlarla kullanmak; döngüyle dizgi taramak.

Karşılaştırma

İçerik karşılaştırmasını (`equals`) referans karşılaştırmasından (`==`) ayırmak; `compareTo` sonucunu hesaplamak; Türkçe harflerde yerel ayarın etkisini açıklamak.

Girdi ve çıktı

`Scanner` ile kelime ve satır okumak, tampon davranışını izlemek; `StringBuilder` ile verimli birleştirme yapmak; `printf` ile biçimli metin üretmek.

11.1 Dizgi nedir?

Şimdiye kadar yazılan programların çoğu sayılarla çalıştı. Bir sayı `int` ya da `double` değişkeninde tutulur ve aritmetik işlemlere girer. Metin ise başka bir yapıdır: harflerin, rakamların, boşlukların ve noktalama işaretlerinin sıralı bir dizisidir. Bu kesim metnin Java'daki iki yapı taşını, `char` ve `String` tiplerini tanıtır.

11.1.1 Karakter ve dizgi

> **Tanım 11.1**
>
> **Karakter** (*character*), metnin tek bir harfi, rakamı, boşluğu ya da işaretidir. Java'da bir karakter `char` tipindeki bir değişkende tutulur ve tek tırnakla yazılır: `'T'`, `'7'`, `' '`. **Dizgi** (*string*), sıfır ya da daha fazla karakterin sıralı dizisidir. Java'da dizgiler `String` sınıfının nesneleridir ve çift tırnakla yazılır: `"Tarkan"`, `"7"`, `""`.

Tırnak türü önemlidir. `'T'` tek bir karakterdir ve tipi `char`'dır. `"T"` ise uzunluğu 1 olan bir dizgidir ve tipi `String`'dir. İkisi ekranda aynı görünür ama Java için farklı tiplerdir. Bu yüzden bir dizgi bir `char` değişkenine atanamaz:

```
public class IlkelDegil {
    public static void main(String[] args) {
        char c = "T";
        System.out.println(c);
    }
}
```

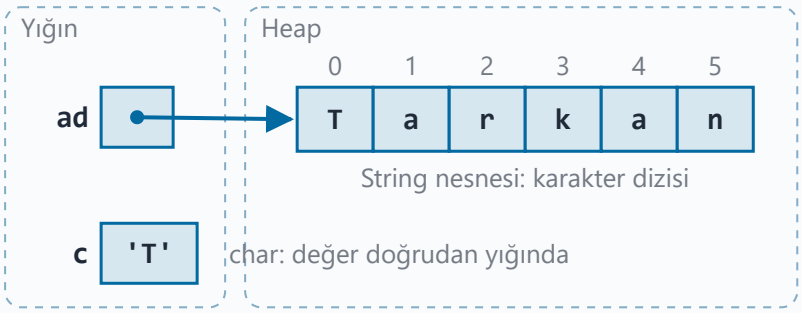
```
IlkelDegil.java:3: error: incompatible types: String cannot be converted to char
        char c = "T";
                 ^
1 error
```

İleti, `String` tipinin `char` tipine dönüştürülemeyeceğini söyler. Doğrusu `char c = 'T';` yazmaktır.

`String` sınıfı `java.lang` paketindedir. Bu paket her Java programına kendiliğinden eklenir; `Scanner` için gereken `import` satırı `String` için gerekmez. Programın içine doğrudan yazılan `"Tarkan"` gibi bir metne **dizgi sabiti** (*string literal*) denir.

11.1.2 Dizgi bir nesnedir

`char`, `int` ve `double` gibi **ilkel tipler** (*primitive type*) değerlerini doğrudan değişkenin içinde tutar. `String` ise ilkel bir tip değildir; bir sınıftır. Bir `String` değişkeni metnin kendisini değil, metni taşıyan nesnenin bellekteki yerini tutar. Bu yer bilgisine **referans** (*reference*) denir. Şekil 11.1 iki tipin bellekteki görünümünü karşılaştırır.



Şekil 11.1: `String ad = "Tarkan";` ve `char c = 'T';` bildirimlerinin bellekteki görünümü. `ad` değişkeni heap'teki altı karakterlik nesneyi gösteren bir ok (referans) tutar. `c` değişkeni ise `'T'` değerini doğrudan içinde taşır.

Şekildeki iki bellek bölgesinin adı ilerideki bütün çizimlerde geçer. **Yığın** (*stack*), metotların yerel değişkenlerinin tutulduğu bölgedir. **Heap** ise `new` ile oluşturulan nesnelerin ve dizgilerin bulunduğu bölgedir. Yığındaki bir `String` değişkeni heap'teki nesneyi bir okla gösterir. Aynı nesneyi birden çok değişken gösterebilir. Dizgilerin karşılaştırılmasında bu paylaşım belirleyicidir.

> **Tanım 11.2**
>
> Bir **nesne** (*object*), bir sınıftan üretilmiş, kendi verisini ve metotlarını taşıyan bellek parçasıdır. **Referans**, bir nesnenin bellekteki yerini gösteren değerdir. `String s = "Ada";` bildiriminde `s` değişkeni bir referanstır; `"Ada"` ise heap'teki nesnedir.

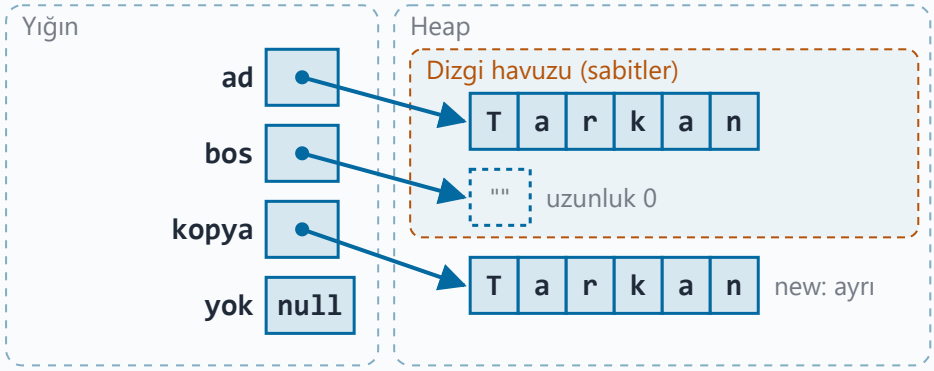
Nesne olmanın bir sonucu da metotlardır. Bir dizgiye ait işlemler nokta işaretiyle çağrılır: `ad.length()` dizginin uzunluğunu, `ad.charAt(0)` ilk karakterini verir. İlkel tiplerin metodu yoktur; `c.length()` gibi bir yazım derlenmez.

11.1.3 Dizgi oluşturma, boş dizgi ve null

Bir dizgi değişkenine dört farklı biçimde ilk değer verilebilir:

```
String ad = "Tarkan";
String bos = "";
String kopya = new String(ad);
String yok = null;
```

İlk satır en yaygın biçimdir: `ad`, `"Tarkan"` sabitini gösterir. İkinci satırdaki `""`, hiç karakter içermeyen **boş dizgidir**. Üçüncü satır `new` ile yeni bir nesne oluşturur ve `ad`'ın içeriğini ona kopyalar. Dördüncü satırdaki `null`, değişkenin hiçbir nesneyi göstermediği anlamına gelir. Şekil 11.2 dört değişkenin bellekteki durumunu gösterir.



Şekil 11.2: Dört dizgi değişkeni. `ad` ve `bos` dizgi havuzundaki sabitleri gösterir. `kopya`, `new` ile açılan ayrı bir nesneyi gösterir; içeriği aynı olsa da nesne farklıdır. `yok` hiçbir nesneyi göstermez.

Şekildeki **dizgi havuzu**, heap içinde aynı sabitlerin tek bir kopyasının tutulduğu özel bir bölgedir. Buradaki kural kısadır: sabitle oluşturulan dizgi havuzdan gelir, `new` ise her zaman yeni bir nesne açar. Uygulamada `new String(...)` yazmaya neredeyse hiç gerek yoktur; dizgi sabiti tercih edilir.

Boş dizgi ile `null` sık karıştırılır. Boş dizgi gerçek bir nesnedir; uzunluğu 0'dır ve bütün metotları çağrılabilir. `null` ise bir nesne değildir; bir değişkenin "henüz hiçbir şeyi göstermiyorum" durumudur. `null` bir değişken üzerinden metot çağrılırsa program `NullPointerException` istisnasıyla durur. Aşağıdaki program dört değişkeni yazdırır, sonra `yok.length()`'i çağırır:

```
public class Olusturma {
    public static void main(String[] args) {
        String ad = "Tarkan";
        String bos = "";
        String kopya = new String(ad);
        String yok = null;
        char c = 'T';
        System.out.println(ad + " " + ad.length());
        System.out.println("[" + bos + "] " + bos.length());
        System.out.println(kopya + " " + kopya.length());
        System.out.println(yok);
        System.out.println(c);
        System.out.println(yok.length());
    }
}
```

```
Tarkan 6
[] 0
Tarkan 6
null
T
Exception in thread "main" java.lang.NullPointerException: Cannot invoke "String.length()"
because "yok" is null
```

Çıktıda boş dizginin köşeli parantezler arasında hiçbir şey göstermediği ve uzunluğunun 0 olduğu görülür. `System.out.println(yok)` hata vermez, `null` yazar. Hata ancak `null` üzerinden bir metot çağrıldığında oluşur. JDK 21'in iletisi sorunun nedenini de söyler: `yok` null olduğu için `String.length()` çağrılamaz.

Sık yapılan hata

Değeri henüz belli olmayan bir dizgi değişkenine `null` verip sonra metot çağırmak. Değişken bir döngüde ya da koşulda doldurulacaksa ilk değeri `""` yapmak çoğu zaman daha güvenlidir. `null` gerçekten "değer yok" anlamı taşıyorsa metot çağrısından önce `s != null` denetimi yapılır.

11.1.4 + ile birleştirme

İki dizgiyi uç uca eklemeye **dizgi birleştirme** (*string concatenation*) denir. Java'da bunun için `+` işleci kullanılır. Aynı işleç sayılarda toplama yapar. Hangisinin yapılacağına işlenenlerin tipi karar verir: iki işlenenden en az biri `String` ise `+` birleştirmedir, ikisi de sayıysa toplamadır.

```java
public class Birlestirme {
    public static void main(String[] args) {
        String sayi1 = "12";
        String sayi2 = "34";
        String toplam = sayi1 + sayi2;
        System.out.println(toplam);
        int a = 12, b = 34;
        System.out.println(a + b);
        System.out.println(sayi1 + 3 + 4);
        System.out.println(3 + 4 + sayi1);
        System.out.println(sayi1 + (3 + 4));
        System.out.println("Toplam: " + a + b);
        System.out.println("Toplam: " + (a + b));
        System.out.println("x" + null + 'c' + 2.5 + true);
        System.out.println("Merhaba".concat(" dünya"));
    }
}
```

```
1234
46
1234
712
127
Toplam: 1234
Toplam: 46
xnullc2.5true
Merhaba dünya
```

Program satır satır şöyle çalışır. `sayi1 + sayi2` iki dizgiyi birleştirir; değişkenin adı "toplam" olsa da sonuç `"1234"` 'tür. `a + b` iki `int` 'i toplar: 46. Sonraki iki satır işlem sırasının etkisini gösterir. `+` soldan sağa işlenir:

- `sayi1 + 3 + 4` : önce *"12" + 3* hesaplanır. Bir işlenen dizgi olduğu için 3 dizgiye çevrilir: `"123"` . Sonra *"123" + 4* = `"1234"` .
- `3 + 4 + sayi1` : önce *3 + 4* hesaplanır. İkisi de sayıdır: 7. Sonra *7 + "12"* = `"712"` .
- `sayi1 + (3 + 4)` : parantez önce hesaplanır: 7. Sonra *"12" + 7* = `"127"` .

`"Toplam: " + a + b` satırı da aynı tuzağa düşer: soldaki dizgi yüzünden iki sayı da birer birer metne eklenir ve "Toplam: 1234" yazılır. Toplamın yazılması için parantez gerekir: `"Toplam: " + (a + b)` . Bir dizgiye `null` , `char` , `double` ve `boolean` değerleri de eklenebilir; her biri metin karşılığına çevrilir. `concat` metodu da iki dizgiyi birleştirir, ancak yalnızca dizgi kabul eder.

Örnek 11.1

`int x = 5, y = 2;` iken `"x+y=" + x + y` ve `x + y + "=x+y"` ifadelerinin değerleri nedir?

Çözüm. İlk ifadede en soldaki işlenen dizgidir. *"x+y=" + 5* = `"x+y=5"`, ardından *"x+y=5" + 2* = `"x+y=52"`. İkinci ifadede ilk iki işlenen sayıdır: *5 + 2 = 7*. Sonra *7 + "=x+y"* = `"7=x+y"`. Aynı üç işlenen, sıraları değişince farklı sonuç verir.

11.1.5 char bir sayıdır

Bilgisayar karakterleri sayıyla saklar. Her karakterin Unicode standardında bir kod numarası vardır: `'a'` 97, `'b'` 98, `'A'` 65, `'0'` 48'dir. Java'da `char` aslında 16 bitlik işaretsiz bir tam sayı tipidir. Bu yüzden bir `char` aritmetik işlemlere girebilir. Sonucun tipi kurala bağlıdır: `char` ile bir tam sayı ya da iki `char` toplanınca sonuç `int` olur.

```
public class CharAritmetik {
    public static void main(String[] args) {
        char k = 'a';
        System.out.println(k + 1);
        System.out.println((char) (k + 1));
        System.out.println("" + k + 1);
        System.out.println('a' + 'b');
        System.out.println("" + 'a' + 'b');
        k++;
        System.out.println(k);
    }
}
```

```
98
b
a1
195
ab
b
```

`k + 1` bir `int`'tir: *97 + 1 = 98*. `(char) (k + 1)` bu sayıyı yeniden karaktere çevirir: `b`. `"" + k + 1` ifadesinde en soldaki boş dizgi bütün ifadeyi birleştirmeye çevirir: önce `"a"`, sonra `"a1"`. `'a' + 'b'` ise iki sayının toplamıdır: *97 + 98 = 195*. İki karakteri yan yana yazdırmak için başa yine `""` konur. `k++` ise `char` değişkenini olduğu yerde bir artırır ve tipi değişmez: `b`.

Karakterlerin sayı olması dizgi işlemlerinin çoğunda işe yarar. İki küçük harfin farkı alfabedeki uzaklıklarını verir: *'k' − 'a' = 107 − 97 = 10*. Bir rakam karakterinden `'0'` çıkarılınca rakamın sayı değeri bulunur: *'7' − '0' = 55 − 48 = 7*. Harf frekansı ve Sezar şifresi uygulamaları bu iki fikre dayanır.

```
public class CharSayi {
    public static void main(String[] args) {
        System.out.println((int) 'a' + " " + (int) 'k' + " " + (int) 'v' + " " + (int) 'z'
+ " " + (int) 'A' + " " + (int) '0');
        System.out.println('k' - 'a');
        System.out.println((char) ('a' + 10));
        System.out.println((int) 'ç' + " " + ('ç' - 'a') + " " + (int) 'ı' + " " + (int)
'İ' + " " + (int) 'ş');
        System.out.println('7' - '0');
    }
}
```

```
97 107 118 122 65 48
10
k
231 134 305 304 351
7
```

Son satırlar Türkçe harflerin kodlarını da gösterir: `'ç'` 231, `'ı'` 305, `'İ'` 304, `'ş'` 351'dir. Bu kodlar İngiliz alfabesinin 97–122 aralığının dışındadır. *'ç' − 'a' = 134* olduğu için `'a'`'dan `'z'`'ye 26 hücrelik bir harf tablosunda `'ç'`'ye yer yoktur.

11.2 Değiştirilemezlik

Dizgilerin en önemli özelliği, oluşturulduktan sonra içeriklerinin değişmemesidir. Bu özellik ilk bakışta garip gelir: `s = s + "!"` yazınca `s` değişmiş gibi görünür. Gerçekte değişen nesne değil, değişkenin gösterdiği yerdir.

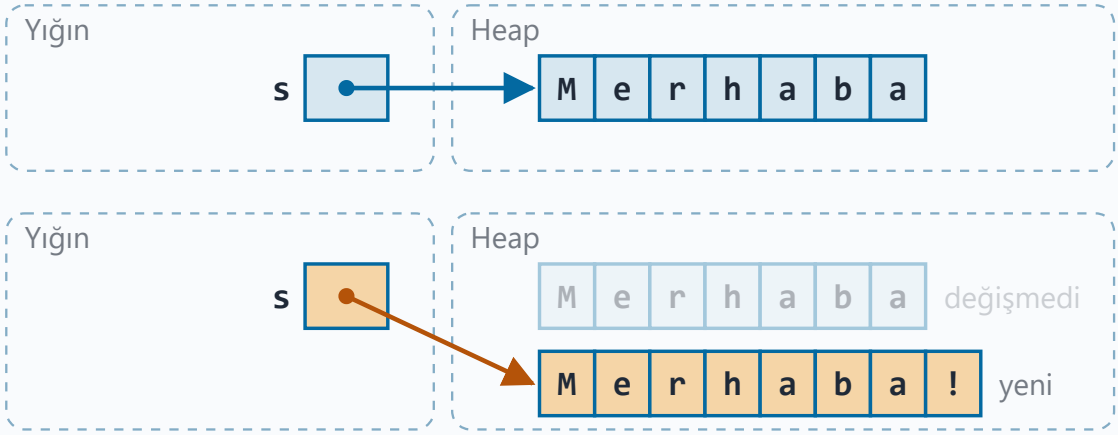
11.2.1 Yeni nesne, aynı eski nesne

Tanım 11.3

Oluşturulduktan sonra içeriği değiştirilemeyen nesneye **değiştirilemez** (*immutable*) nesne denir. `String` değiştirilemez bir sınıftır: hiçbir metodu nesnenin karakterlerini değiştirmez. Dizgiyi "değiştiren" her işlem (birleştirme, büyük harfe çevirme, alt dizgi alma, değiştirme) yeni bir `String` nesnesi üretir ve onu döndürür. Bu özelliğin adı **değiştirilemezliktir** (*immutability*).

Şekil 11.3 en basit örneği gösterir. `String s = "Merhaba";` satırından sonra `s`, yedi karakterlik nesneyi gösterir. `s = s + "!";` satırı bu nesneye ünlem eklemez. Önce `s + "!"` hesaplanır ve sekiz karakterlik yeni bir nesne oluşur. Sonra atama, `s`'nin okunu bu yeni nesneye taşır. Eski `"Merhaba"` nes-

nesi olduğu gibi kalır.



Şekil 11.3: `s = s + "!";` ataması. Üstte atamadan önceki durum: `s`, "Merhaba" nesnesini gösterir. Altta atamadan sonraki durum: "Merhaba!" adlı yeni bir nesne oluşmuş ve ok ona taşınmıştır; eski nesne değişmemiştir.

Bunu bir programla da görmek mümkündür. Atamadan önce eski nesneyi gösteren ikinci bir değişken (`eski`) tutulursa, atamadan sonra iki değişkenin farklı nesneleri gösterdiği anlaşılır:

```
public class Unlem {
    public static void main(String[] args) {
        String s = "Merhaba";
        String eski = s;
        s = s + "!";
        System.out.println(s);
        System.out.println(eski);
        System.out.println(s == eski);
    }
}
```

```
Merhaba!
Merhaba
false
```

`eski` hâlâ "Merhaba"yı gösterir. `s == eski` karşılaştırması, iki değişkenin aynı nesneyi gösterip göstermediğini sorar ve `false` verir: artık iki ayrı nesne vardır. Hiçbir değişkenin göstermediği nesneler zamanla gereksiz hâle gelir. Java'da bu nesneleri **çöp toplayıcı** (*garbage collector*) kendiliğinden temizler; programcının belleği elle geri vermesi gerekmez.

11.2.2 toUpperCase örneğini adım adım izlemek

Değiştirilemezliğin pratikteki sonucu şudur: bir dizgi metodunun sonucu kullanılmak isteniyorsa bir değişkene atanmalıdır. Aşağıdaki beş satır bu noktayı gösterir.

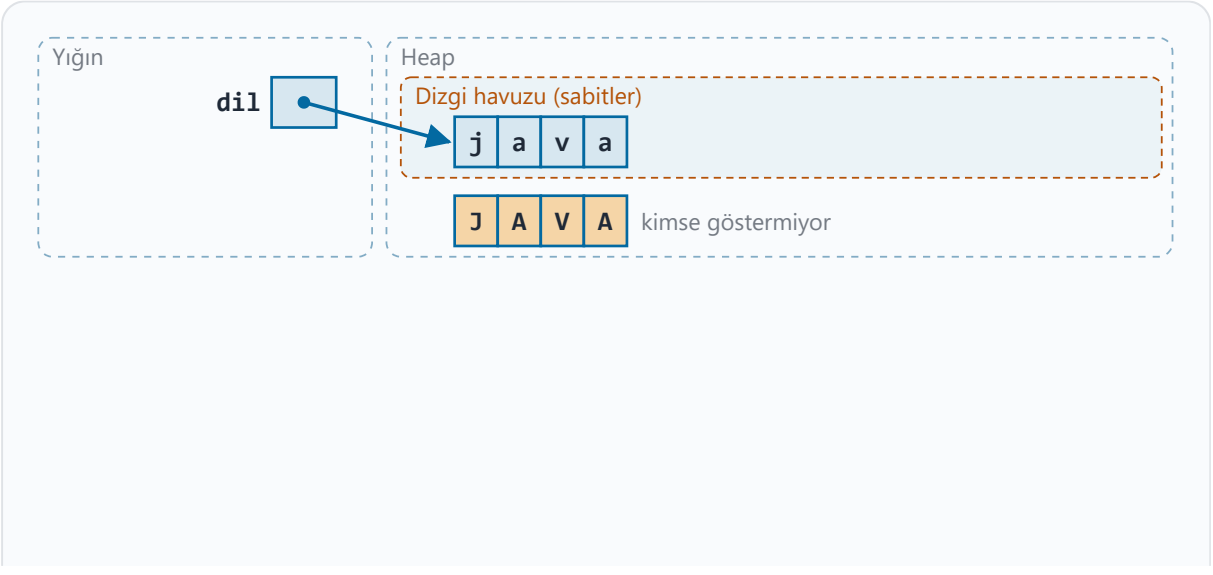
```
String dil = "java";
dil.toUpperCase();
System.out.println(dil); // java
dil = dil.toUpperCase();
System.out.println(dil); // JAVA
```

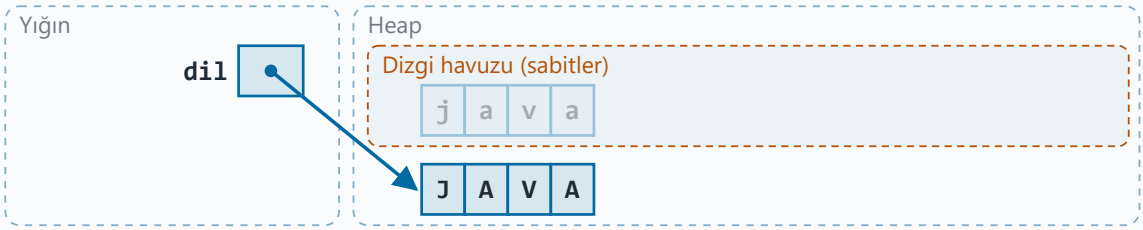
```
java
JAVA
```

İkinci satır `dil.toUpperCase()`'i çağırır ama dönen değeri hiçbir yere koymaz. Metot görevini yapar ve yeni bir `"JAVA"` nesnesi üretir. Ne var ki bu nesneyi gösteren bir değişken yoktur; sonuç kaybolur. `dil` eski nesneyi göstermeye devam ettiği için üçüncü satır `java` yazar. Dördüncü satırda aynı çağrının sonucu `dil`'e atanır; ok yeni nesneye taşınır ve beşinci satır `JAVA` yazar. Tablo 11.1 beş satırın izleme tablosudur; Şekil 11.4 ise kritik iki anı gösterir.

Satır	dil neyi gösterir?	Heap'teki nesneler	Çıktı
1	"java"	"java"	
2	"java"	"java", "JAVA" (kimse göstermiyor)	
3	"java"	"java", "JAVA" (çöp)	java
4	"JAVA" (yeni nesne)	"java", "JAVA"	
5	"JAVA"	"java", "JAVA"	JAVA

Tablo 11.1: toUpperCase örneğinin izleme tablosu. “Heap'teki nesneler” sütunu o satırdan sonra bellekte bulunan dizgi nesnelerini verir.





Şekil 11.4: Üstte 2. satırdan sonraki durum: `toUpperCase()` yeni bir "JAVA" nesnesi üretmiş, ama onu gösteren değişken yoktur. Altta 4. satırdan sonraki durum: dönen nesne `dil`'e atanmış, ok ona taşınmıştır. "java" nesnesi iki durumda da aynıdır.

Sık yapılan hata

`ad.trim();`, `ad.toUpperCase();` ya da `ad.replace('a', 'e');` gibi satırları tek başına yazıp `ad`'ın değiştiğini sanmak. Bu satırlar derlenir ve çalışır, ama sonuç atılır. Doğrusu `ad = ad.trim();` biçiminde sonucu yeniden atamaktır. Bazı geliştirme ortamları dönen değeri kullanılmayan çağrılar için uyarı gösterir.

Bir dizginin tek bir karakterini olduğu yerde değiştirmenin yolu da yoktur. Dizilerde `a[0] = 5;` yazılabilir, ama dizgilerde buna karşılık gelen bir işlem yoktur. `charAt` yalnızca okumak içindir; ona değer atanamaz:

```
public class SetCharAt {
    public static void main(String[] args) {
        String s = "java";
        s.charAt(0) = 'J';
        System.out.println(s);
    }
}
```

```
SetCharAt.java:4: error: unexpected type
        s.charAt(0) = 'J';
                ^
  required: variable
  found:    value
1 error
```

İleti, eşittir işaretinin solunda bir değişken beklendiğini ama bir değer bulunduğunu söyler. `s.charAt(0)` bir değer üretir, bir bellek yeri değildir. İlk harfi büyütmek için yeni bir dizgi kurulur: `s = Character.toUpperCase(s.charAt(0)) + s.substring(1);` Bu satır `"Java"` üretir.

11.2.3 Değiştirilemezlik neden yararlıdır?

Her değişiklikte yeni nesne üretmek israf gibi görünebilir. Java'nın tasarımcıları bu bedeli üç kazanç için ödemeyi seçmiştir:

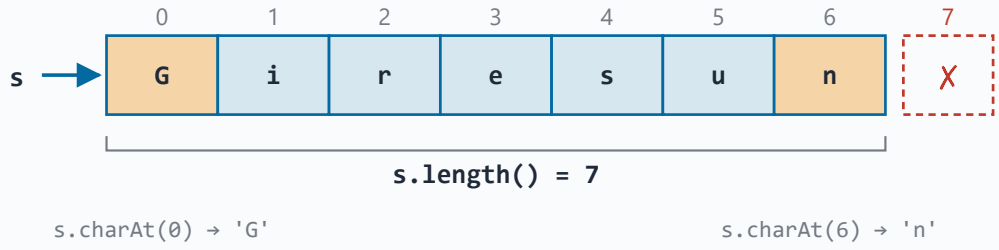
- **Güvenli paylaşım.** Aynı dizgi nesnesini birden çok değişken gösterebilir. Hiçbiri nesneyi değiştiremeyeceği için bir değişken üzerinden yapılan işlem ötekileri etkilemez. Dizgi havuzu bu sayede mümkündür: programdaki bütün `"Giresun"` sabitleri tek bir nesneyi paylaşır.
- **Öngörülebilirlik.** Bir dizgi bir metoda gönderildiğinde metot onu bozamaz. Çağıran taraf, metottan dönünce dizginin aynı kaldığından emindir.
- **Güvenlik.** Dosya adları, ağ adresleri ve parolalar dizgilerde taşınır. Denetimden geçmiş bir dosya adının sonradan değiştirilememesi önemli bir güvencedir.

Bedel ise çok sayıda birleştirmenin yavaşlamasıdır. Bir döngüde binlerce kez `s = s + x` yazılırsa her turda yeni nesne üretilir ve eski içerik kopyalanır. Bu durum için Java değiştirilebilir bir yardımcı sınıf sunar: `StringBuilder`.

11.3 İndisleme

Bir dizginin karakterleri sıralıdır ve her birinin bir sıra numarası vardır. Bu numaraya **indis** (*index*) denir. Diziler gibi dizgilerde de sayma 0'dan başlar.

> **Tanım 11.4**
>
> Uzunluğu n olan bir dizginin karakterleri 0, 1, ..., n − 1 **indislerindedir**. İlk karakter 0, son karakter n − 1 indisindedir. Bir karaktere indisiyle erişmeye **indisleme** (*indexing*) denir. Java'da `s.length()` dizginin uzunluğunu, `s.charAt(i)` i indisindeki karakteri bir `char` değeri biçiminde döndürür.



Şekil 11.5: `s = "Giresun"` dizgisinin indisleri. Yedi karakter 0'dan 6'ya kadar numaralanır. `s.charAt(0)` 'G', `s.charAt(6)` 'n' döndürür. 7 numaralı bir hücre yoktur; `s.charAt(7)` geçersizdir.

Şekil 11.5'teki dizgi için aşağıdaki program uzunluğu, ilk ve son karakteri yazar. Son karakterin indisi her dizgi için `s.length() - 1` ifadesiyle bulunur; böylece kod uzunluk ne olursa olsun çalışır.

```
public class Indis {
    public static void main(String[] args) {
        String s = "Giresun";
        System.out.println(s.length());
        System.out.println(s.charAt(0));
        System.out.println(s.charAt(s.length() - 1));
        int[] dizi = {4, 8, 15};
        System.out.println(dizi.length + " " + s.length());
        System.out.println(s.charAt(7));
    }
}
```

```
7
G
n
3 7
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: Index 7 out of
bounds for length 7
```

Son satır, var olmayan 7 numaralı karaktere erişmeye çalışır. Derleyici bu hatayı yakalamaz, çünkü indis genellikle program çalışırken hesaplanır. Hata çalışma zamanında `StringIndexOutOfBoundsException` istisnasıyla ortaya çıkar. İleti, 7 indisinin 7 uzunluklu bir dizginin sınırları dışında kaldığını söyler. Bu tür bir indise **sınır dışı** (*out of bounds*) indis denir.

Programdaki `dizi.length` ile `s.length()` arasındaki fark da önemlidir. Dizilerde `length` bir alandır ve parantezsiz yazılır. Dizgilerde `length()` bir metottur ve parantezle çağrılır. Parantez unutulursa program derlenmez:

```
public class StringMetotDegil {
    public static void main(String[] args) {
        String s = "Giresun";
        System.out.println(s.length);
    }
}
```

```
StringMetotDegil.java:4: error: cannot find symbol
        System.out.println(s.length);
                           ^
  symbol:   variable length
  location: variable s of type String
1 error
```

Derleyici, `String` tipinde `length` adlı bir değişken bulamadığını söyler. İletideki “symbol” sözcüğü değişken, metot ya da sınıf gibi adlandırılmış bir program öğesi anlamındadır.

Sık yapılan hata

Son karaktere `s.charAt(s.length())` ile erişmeye çalışmak. Uzunluk n ise son indis n − 1'dir; `s.charAt(s.length())` her zaman sınır dışıdır. Aynı hata döngülerde de görülür: `for (int i = 0; i <= s.length(); i++)` döngüsünün son turu `charAt` 'te istisna fırlatır. Doğru koşul `i < s.length()` 'tir.

11.4 Dizgi metotları

`String` sınıfının altmışı aşkın metodu vardır. Bu kesim gündelik programlarda en sık gerekenleri ele alır. Hepsinde ortak bir kural geçerlidir: hiçbir metot asıl dizgiyi değiştirmez. Bir sonuç gerekiyorsa metot onu yeni bir değerle döndürür.

11.4.1 length, indexOf, substring ve contains

Dört temel metot bir arada şu programda kullanılır:

```
public class TemelMetotlar {
    public static void main(String[] args) {
        String s = "Java Programlama";
        int uzunluk = s.length();
        int konum = s.indexOf("Programlama");
        String alt = s.substring(5, 16);
        boolean icerir = s.contains("Java");
        System.out.println(alt + " " + konum);
        System.out.println(uzunluk + " " + icerir + " " + s.indexOf("java") + " " +
s.indexOf('a') + " " + s.lastIndexOf('a'));
    }
}
```

```
Programlama 5
16 true -1 1 15
```

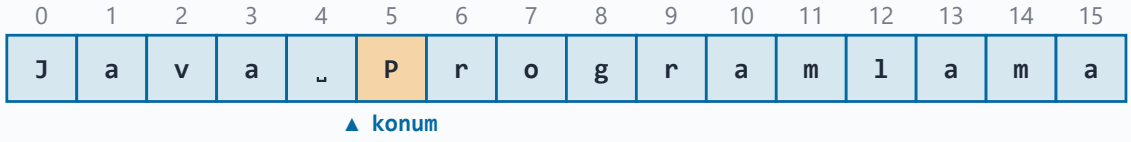
Programın son `println` satırı, ilk altı satıra eklenmiş bir denetim satırıdır; çıktısı aşağıda ayrıca açıklanır. İlk altı satır şöyle çalışır:

1. `s`, 16 karakterlik `"Java Programlama"` dizgisini gösterir. Aradaki boşluk da bir karakterdir ve 4 indisindedir.
2. `s.length()` 16 döndürür. İndisler 0'dan 15'e kadardır.
3. `s.indexOf("Programlama")`, aranan metnin *ilk karakterinin* indisini döndürür. “P” 5 indisindedir; `konum` 5 olur. Aranan metin hiç geçmeseydi sonuç −1 olurdu.

4. `s.substring(5, 16)`, 5 indisinden başlayıp 16 indisinden *önce* biten parçayı verir. Başlangıç dahil, bitiş hariçtir. Parçanın uzunluğu *16 − 5 = 11*'dir: `"Programlama"`.
5. `s.contains("Java")`, aranan metin dizginin herhangi bir yerinde geçiyorsa `true` döndürür.
6. `alt + " " + konum` birleştirilip yazılır: `Programlama 5`.

Satır	uzunluk	konum	alt	icerir	Çıktı
1					
2	16				
3	16	5			
4	16	5	"Programlama"		
5	16	5	"Programlama"	true	
6	16	5	"Programlama"	true	Programlama 5

Tablo 11.2: Temel metotlar programının izleme tablosu. Boş hücre, değişkenin henüz bildirilmediğini gösterir.



Şekil 11.6: `"Java Programlama"` dizgisinin on altı hücresi. `indexOf("Programlama")`, aranan metnin ilk karakteri olan "P"nin indisini, yani 5'i verir; ok bu hücreyi gösterir.

Programa eklenen denetim satırı üç ayrıntıyı daha gösterir. `s.indexOf("java")` −1 verir, çünkü arama büyük-küçük harfe duyarlıdır: dizgide "java" değil "Java" vardır. `s.indexOf('a')` 1 verir; `indexOf` tek bir karakteri de arayabilir ve ilk geçtiği yeri bulur. `s.lastIndexOf('a')` ise aramayı sondan yapar ve son geçtiği yeri, 15'i bulur.

Örnek 11.2

`String t = "bilgisayar";` için `t.indexOf('a')`, `t.lastIndexOf('a')`, `t.indexOf("say")` ve `t.substring(2, 5)` değerleri nedir?

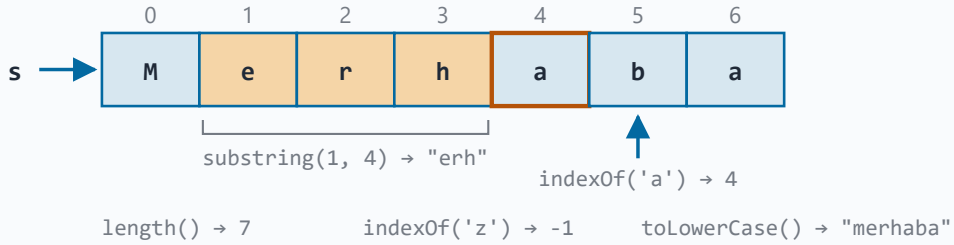
Çözüm. Önce indisler yazılır: b=0, i=1, l=2, g=3, i=4, s=5, a=6, y=7, a=8, r=9. İlk 'a' 6 indisinde, son 'a' 8 indisindedir. "say" metni 5 indisindeki 's' ile başlar; sonuç 5'tir. `substring(2, 5)` 2, 3 ve 4 indislerindeki karakterleri alır: `"lgi"`. Uzunluğu *5 − 2 = 3*'tür.

11.4.2 Sık kullanılan metotlar

Tablo 11.3 en sık kullanılan metotları özetler. Ardından gelen program, hepsini kısa bir dizgi üzerinde dener.

Metot	Sonuç	Dönüş tipi
`length()`	Karakter (kod birimi) sayısı	`int`
`charAt(i)`	i konumundaki karakter	`char`
`substring(a, b)`	a dahil, b hariç alt dizgi	`String`
`indexOf(x)`	x'in ilk konumu; yoksa −1	`int`
`toLowerCase()`, `toUpperCase()`	Küçük ya da büyük harfli yeni dizgi	`String`
`trim()`, `strip()`	Baştaki ve sondaki boşlukları atılmış yeni dizgi	`String`
`isEmpty()`, `isBlank()`	Uzunluk 0 mı; yalnızca boşluktan mı oluşuyor?	`boolean`

Tablo 11.3: Sık kullanılan String metotları. Hepsi asıl dizgiyi değiştirmeden bir sonuç döndürür.



Şekil 11.7: `s = "Merhaba"` üzerinde beş metot. `substring(1, 4)` 1, 2 ve 3 indislerini alır: “erh”. `indexOf('a')` ilk 'a'nın indisini, 4'ü verir; 'z' hiç geçmediği için `indexOf('z')` −1'dir.

```
public class SikMetotlar {
    public static void main(String[] args) {
        String s = "Merhaba";
        System.out.println(s.length());
        System.out.println(s.charAt(1));
        System.out.println(s.substring(1, 4));
        System.out.println(s.indexOf('a'));
        System.out.println(s.indexOf('z'));
        System.out.println(s.toLowerCase());
        System.out.println(s.toUpperCase());
        String k = "  Ada Can \t";
        System.out.println("[" + k.trim() + "] [" + k.strip() + "]");
        String g = " Ada ";
        System.out.println(g.trim().length() + " " + g.strip().length());
        System.out.println("".isEmpty() + " " + "   ".isEmpty() + " " + "   ".isBlank());
        System.out.println("ab".repeat(3));
    }
}
```

```
7
e
erh
4
-1
merhaba
MERHABA
[Ada Can] [Ada Can]
5 3
true false true
ababab
```

İlk yedi satır Şekil 11.7'deki değerleri doğrular. Sonraki satırlar boşluk temizleme metotlarını dener. `k` dizgisinin başında iki boşluk, sonunda bir boşluk ve bir sekme karakteri (`\t`) vardır. `trim()` ve `strip()` ikisi de bunları siler ve aradaki boşluğa dokunmaz: `[Ada Can]`.

İki metodun farkı daha seyrek karakterlerde ortaya çıkar. `trim()` yalnızca kodu 32 ve altında olan karakterleri, yani ASCII boşluk ve kontrol karakterlerini siler. `strip()` ise Unicode'un bütün boşluk karakterlerini tanır. `g` dizgisinin iki ucunda “em boşluğu” (U+2003) adlı geniş bir boşluk vardır. `trim()` bu karakteri tanımaz ve uzunluk 5 kalır; `strip()` siler ve uzunluk 3'e iner. Kullanıcı girdisinde ve kopyalanıp yapıştırılmış metinde `strip()` daha güvenlidir.

`isEmpty()` yalnızca uzunluğu 0 olan dizgide `true` verir. `isBlank()` ise boş dizgide ve yalnızca boşluktan oluşan dizgide `true` verir. Bu yüzden üç boşluklu `" "` için ilki `false`, ikincisi `true`'dur. Son satırdaki `repeat(3)` dizgiyi üç kez yan yana yazar: `ababab`.

11.4.3 split ile bölme

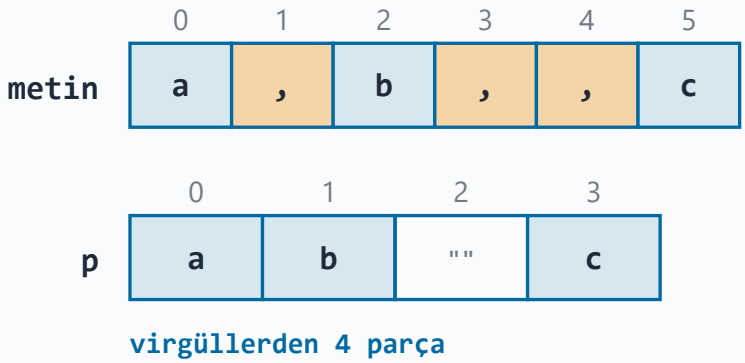
Virgülle ayrılmış bir liste, boşluklarla ayrılmış bir cümle ya da noktalı bir tarih, parçalara ayrılarak işlenir. Parçaları ayıran karaktere **ayırıcı** (*delimiter*) denir. `split` metodu dizgiyi ayırıcıların bulunduğu yerlerden keser ve parçaları bir `String` dizisinde döndürür.

```java
import java.util.Arrays;

public class SplitReplace {
    public static void main(String[] args) {
        String[] p = "a,b,,c".split(",");
        System.out.println(Arrays.toString(p) + " " + p.length); // [a, b, , c] 4
        String s = "ali veli";
        String t = s.replace('l', 'L');
        System.out.println(t); // aLi veLi
        System.out.println(s); // ali veli
        String[] q = "a,b,,".split(",");
        System.out.println(Arrays.toString(q) + " " + q.length);
        String[] r = ",a,b".split(",");
        System.out.println(Arrays.toString(r) + " " + r.length);
        String[] u = "3.14".split(".");
        System.out.println(u.length);
        String[] v = "3.14".split("\\.");
        System.out.println(Arrays.toString(v));
    }
}
```

```
[a, b, , c] 4
aLi veLi
ali veli
[a, b] 2
[, a, b] 3
0
[3, 14]
```

Programın ilk altı satırı sunumdaki örnektir; sonraki satırlar `split`'in öteki kurallarını dener. `"a,b,,c".split(",")` metni üç virgülden keser ve dört parça üretir: `"a"`, `"b"`, `""` ve `"c"`. Ardışık iki virgülün arası boştur; bu boş dizgi de diziye girer. `Arrays.toString(p)` diziyi okunabilir biçimde yazar: `[a, b, , c]`. İki virgül arasındaki boşluk, boş parçanın yeridir. Şekil 11.8 bu bölmeyi gösterir.



Şekil 11.8: `"a,b,,c".split(",")` . Üstteki altı karakterlik metin 1, 3 ve 4 indislerindeki virgüllerden kesilir. Alttaki `p` dizisinin dört hücresi vardır; 2 indisindeki hücre boş dizgidir.

Ek satırlar `split`'in üç kuralını gösterir:

- **Sondaki boş parçalar atılır.** `"a,b,,".split(",")` dört parça değil iki parça verir: `[a, b]`. Sondaki iki boş parça diziye girmez.
- **Baştaki boş parça kalır.** `",a,b".split(",")` üç parça verir: `[, a, b]`. İlk virgülden önceki boş metin bir parça sayılır.
- **Ayırıcı bir düzenli ifadedir.** `split`'e verilen metin düz bir karakter değil, bir kalıptır. Kalıp dilinde nokta "herhangi bir karakter" anlamına gelir. Bu yüzden `"3.14".split(".")` her karakterden keser, bütün parçalar boş kalır ve sondaki boş parçalar atılınca dizi uzunluğu 0 olur. Noktanın kendisiyle bölmek için `split("\\.")` yazılır ve sonuç `[3, 14]` olur.

Tanım 11.5

Düzenli ifade (*regular expression*), bir metin kalıbını tarif eden küçük bir dildir. Bazı karakterlerin bu dilde özel anlamı vardır: `.` herhangi bir karakter, `+` "bir ya da daha fazla", `[0-9]` herhangi bir rakam, `\s` herhangi bir boşluk karakteri anlamına gelir. Özel bir karakteri kendi anlamıyla kullanmak için önüne ters bölü konur. Java dizgi sabitinde ters bölünün kendisi `\\` diye yazıldığı için kalıp `"\\."` ya da `"\\s+"` biçimini alır.

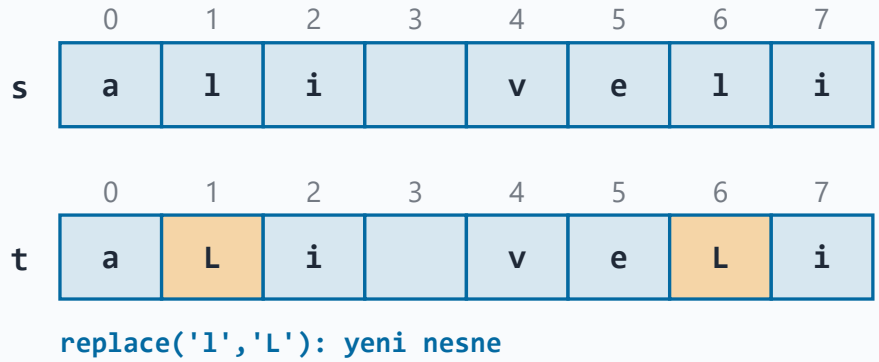
Örnek 11.3

`"23.10.2026".split("\\.")` ve `"elma, armut,kiraz".split(",")` kaç parça verir, parçalar nelerdir?

Çözüm. İlk dizgide iki nokta vardır; üç parça çıkar: `"23"`, `"10"`, `"2026"`. İkinci dizgide iki virgül vardır; yine üç parça çıkar: `"elma"`, `" armut"`, `"kiraz"`. İkinci parçanın başındaki boşluk korunur, çünkü ayırıcı yalnızca virgüldür. Boşluğun da atılması için ya her parçaya `strip()` uygulanır ya da ayırıcı `",\\s*"` (virgül ve ardından sıfır ya da daha fazla boşluk) biçiminde verilir.

11.4.4 replace ve düzenli ifadeler

`replace` bir dizgideki bütün eşleşmeleri başka bir metinle değiştirilmiş yeni bir dizgi döndürür. Aynı programın 4–7. satırları bunu gösterir: `s.replace('l', 'L')`, `"ali veli"` içindeki iki 'l' harfini (1 ve 6 indisleri) büyük harfe çevirir ve `"aLi veLi"` döndürür. `s` aynı kalır; son satır yine `ali veli` yazar. Şekil 11.9 iki nesneyi yan yana gösterir.



Şekil 11.9: `t = s.replace('l', 'L')`. Üstte `s`'nin gösterdiği "ali veli" nesnesi değişmeden durur. Altta `t`, 1 ve 6 indislerinde 'L' bulunan yeni nesneyi gösterir.

Değiştirme ailesinin dört üyesi vardır ve aralarındaki fark aranan şeyin türündedir. Tablo 11.4 sunumdaki örnekleri, JDK 21 çıktılarıyla birlikte verir.

Çağrı	Sonuç	Açıklama
`"banana".replace('a', 'o')`	`bonono`	her 'a' karakteri 'o' olur
`"banana".replace("an", "AN")`	`bANANa`	düz metin araması; soldan sağa, çakışmadan
`"a1b22c333".replaceAll("[0-9]+", "#")`	`a#b#c#`	her rakam grubu tek bir # olur
`"a1b22c333".replaceFirst("[0-9]+", "#")`	`a#b22c333`	yalnızca ilk eşleşme değişir
`"Ada Can Ece".replaceAll(" +", " ")`	`Ada Can Ece`	art arda boşluklar teke iner

Tablo 11.4: Değiştirme metotları. Sonuç sütunu programın gerçek çıktısıdır.

`replace` düz metin arar; `replaceAll` ve `replaceFirst` ise düzenli ifade arar. `[0-9]+` kalıbı "bir ya da daha fazla rakam" demektir. `"a1b22c333"` içinde üç rakam grubu vardır: `1`, `22` ve `333`. Her grup bütünüyle eşleşir ve tek bir `#` ile değişir. `" +"` kalıbı da "bir ya da daha fazla boşluk" demektir; iki ve üç boşluklu aralıklar tek boşluğa iner.

Sık yapılan hata

Düz bir metni değiştirmek için `replaceAll` kullanmak. Aranan metinde nokta, artı, yıldız gibi özel karakterler varsa sonuç beklenmedik olur. `"1.5".replace(".", ",")` beklenen `1,5`'i verir. `"1.5".replaceAll(".", ",")` ise her karakteri değiştirir ve `,,,` üretir; nokta "herhangi bir karakter" anlamına gelir. Düz metin için `replace` yeterlidir ve o da bütün eşleşmeleri değiştirir.

11.4.5 join, valueOf ve format

Üç `static` metot, bir nesne üzerinden değil sınıf adıyla çağrılır: `String.join`, `String.valueOf` ve `String.format`.

- `String.join(", ", "Ada", "Can", "Ece")`, parçaları aralarına ayırıcı koyarak birleştirir: `Ada, Can, Ece`. `split`'in tersi gibi düşünülebilir.
- `String.valueOf(3.5)` bir değeri dizgiye çevirir: `"3.5"`. Karakter dizisi verilirse karakterleri birleştirir: `String.valueOf(new char[]{'h', 'i'})` `"hi"` döndürür.
- `String.format`, `printf` ile aynı biçim dizgisini kullanır ama sonucu ekrana yazmaz, dizgi biçiminde döndürür.

Tablo 11.5 `format`'ın üç örneğini iki farklı yerel ayarla verir. Programın tamamı `ReplaceTablosu` sınıfıdır; bir kez İngilizce (ABD), bir kez Türkçe yerel ayarlı bir Java sanal makinesinde çalıştırılmıştır.

Çağrı	en_US makinede	tr_TR makinede
`String.format(Locale.US, "%.2f", 3.14159)`	`3.14`	`3.14`
`String.format(Locale.US, "%5d|%-5d|%05d", 42, 42, 42)`	`␣␣42|42␣␣␣|00042`	`␣␣42|42␣␣␣|00042`
`String.format(Locale.US, "%,d", 1234567)`	`1,234,567`	`1,234,567`
`String.format("%.2f", 3.14159)`	`3.14`	`3,14`
`String.format("%,d", 1234567)`	`1,234,567`	`1.234.567`

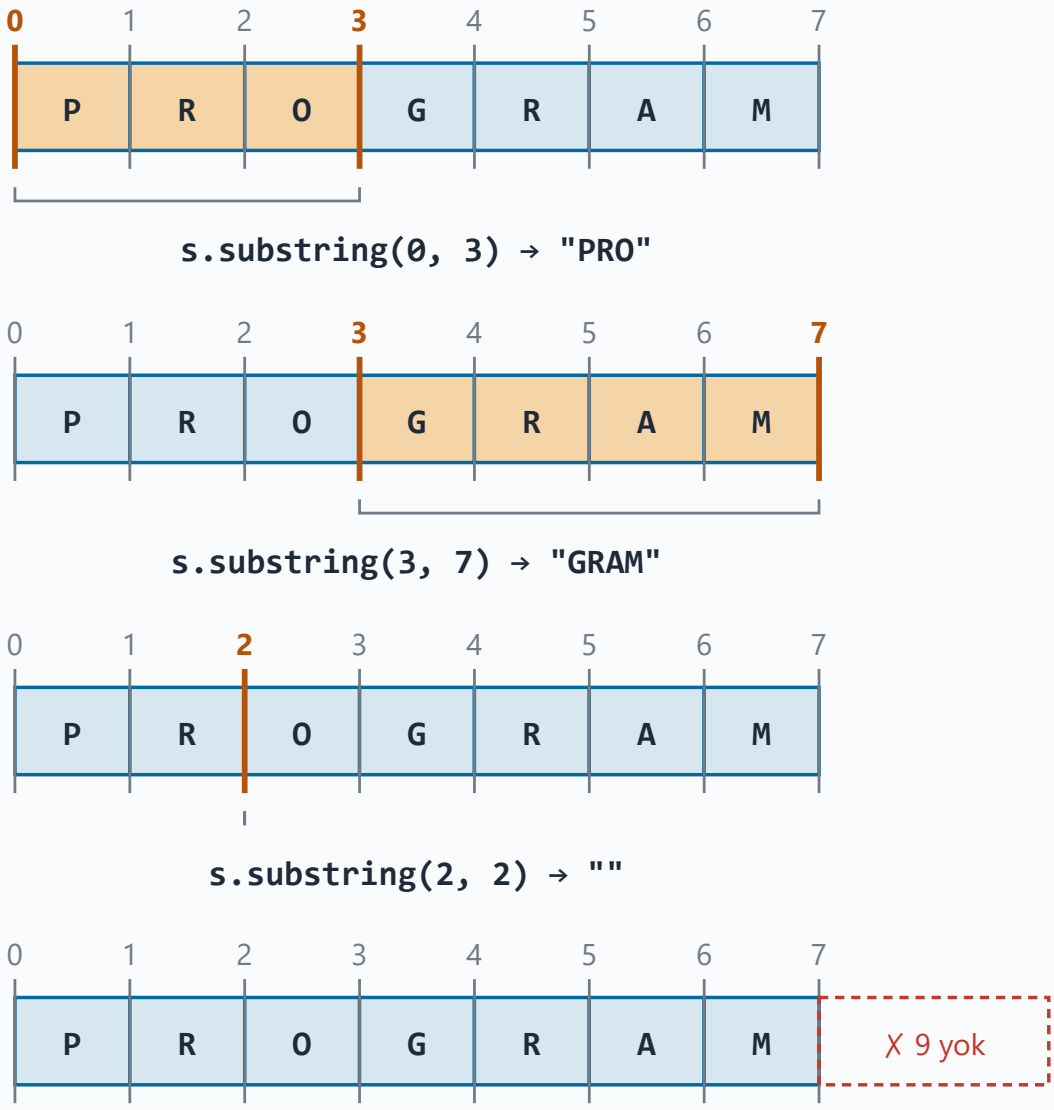
Tablo 11.5: String.format örnekleri. Son iki satırda yerel ayar verilmemiştir; sonuç makinenin ayarına bağlıdır.

Tablodaki ␣ işareti, sonuçtaki boşlukları görünür kılmak içindir. `%5d` sayıyı 5 karakterlik alana sağa yaslar; 42 iki karakter olduğu için önüne üç boşluk gelir. `%-5d` sola yaslar ve boşlukları sona koyar. `%05d` boşluk yerine sıfırla doldurur. `%,d` binlik ayırıcı ekler. `%.2f` sayıyı virgülden sonra iki basamağa yuvarlar.

`Locale.US` verilen satırlar her makinede aynı sonucu verir. Yerel ayar verilmeyen son iki satır ise makinenin diline uyar: Türkçe ayarlı bilgisayarda ondalık ayırıcı virgül, binlik ayırıcı noktadır. Bir dosyaya ya da başka bir programa gidecek sayılarda yerel ayar açıkça verilir; ekranda kullanıcıya gösterilecek sayılarda makinenin ayarı uygun olabilir.

11.4.6 Alt dizgi sınırları

`substring` en sık hata yapılan metottur. Kuralı akılda tutmanın kolay bir yolu, indisleri karakterlerin üzerine değil *aralarındaki çizgilere* yazmaktır. Şekil 11.10'da `"PROGRAM"` dizgisinin yedi karakteri arasında 0'dan 7'ye sekiz çizgi vardır. `substring(a, b)`, a çizgisiyle b çizgisi arasında kalan karakterleri alır.



Şekil 11.10: `s = "PROGRAM"` üzerinde dört çağrı. 0 ile 3 çizgileri arası "PRO"dur. Bitiş `length()` = 7 olabilir: 3 ile 7 arası "GRAM"dır. Başlangıç bitişe eşitse arada karakter yoktur: "". 9 numaralı çizgi olmadığı için `substring(5, 9)` hata verir.

Tanım 11.6

Alt dizgi (*substring*), bir dizginin ardışık karakterlerinden oluşan parçasıdır. `s.substring(a, b)`, a indisindeki karakterden başlayıp b indisindeki karakterden önce biten alt dizgiyi döndürür; uzunluğu b − a'dır. Geçerli aralık $0 \le a \le b \le s.length()$'tir. `s.substring(a)` ise a indisinden dizginin sonuna kadar olan parçayı verir ve `s.substring(a, s.length())` ile aynıdır.

```
public class AltDizgi {
    public static void main(String[] args) {
        String s = "PROGRAM";
        System.out.println(s.substring(0, 3));
        System.out.println(s.substring(3, 7));
        System.out.println(s.substring(3));
        System.out.println("[" + s.substring(2, 2) + "]");
        System.out.println("[" + s.substring(7) + "]");
        System.out.println(s.substring(0, 3).length() + " " + s.substring(2, 5));
        System.out.println(s.substring(5, 9));
    }
}
```

```
PRO
GRAM
GRAM
[]
[]
3 OGR
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: Range [5, 9) out of
bounds for length 7
```

Çıktı her kuralı doğrular. `substring(0, 3)` üç karakter alır: $3 - 0 = 3$. `substring(3, 7)` ve `substring(3)` aynı sonucu verir, çünkü dizginin uzunluğu 7'dir. `substring(2, 2)` boş dizgidir; bu bir hata değildir, “sıfır karakter al” demektir. `substring(7)` da boş dizgidir: 7 son çizgidir ve ondan sonra karakter yoktur. Son satırdaki `substring(5, 9)` ise bitiş uzunluğu aştığı için istisna fırlatır. İleti, [5, 9) aralığının 7 uzunluklu dizgi için geçersiz olduğunu söyler. Köşeli parantez başlangıcın dahil, yuvarlak parantez bitişin hariç olduğunu gösterir; matematikteki yarı açık aralık gösterimidir.

Başlangıç bitişten büyükse de istisna fırlatılır:

```
public class AltDizgiTers {
    public static void main(String[] args) {
        String s = "PROGRAM";
        System.out.println(s.substring(5, 3));
    }
}
```

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: Range [5, 3) out of
bounds for length 7
```

Çağrı	Koşul 0 ≤ a ≤ b ≤ 7	Sonuç	Uzunluk
`s.substring(0, 3)`	0 ≤ 0 ≤ 3 ≤ 7 sağlanır	"PRO"	3
`s.substring(2, 5)`	sağlanır	"OGR"	3
`s.substring(3, 7)`	sağlanır	"GRAM"	4
`s.substring(2, 2)`	sağlanır (a = b)	""	0
`s.substring(7)`	sağlanır (a = 7)	""	0
`s.substring(5, 9)`	9 ≤ 7 yanlış	istisna	–
`s.substring(5, 3)`	5 ≤ 3 yanlış	istisna	–

Tablo 11.6: s = "PROGRAM" (uzunluk 7) için substring çağrıları.

Sık yapılan hata

`substring`'in ikinci parametresini "kaç karakter alınacağı" sanmak. `"PROGRAM".substring(2, 3)`, 2 indisinden başlayan üç karakteri değil, yalnızca bir karakteri ("O") verir. İkinci parametre uzunluk değil bitiş konumudur. k karakter almak için `substring(a, a + k)` yazılır.

11.4.7 Uygulama: e-postayı parçalamak

`indexOf` ile `substring` birlikte kullanıldığında bir metin bir ayırıcının iki yanına ayrılabilir. Bir e-posta adresi `@` işaretiyle kullanıcı adına ve alan adına ayrılır:

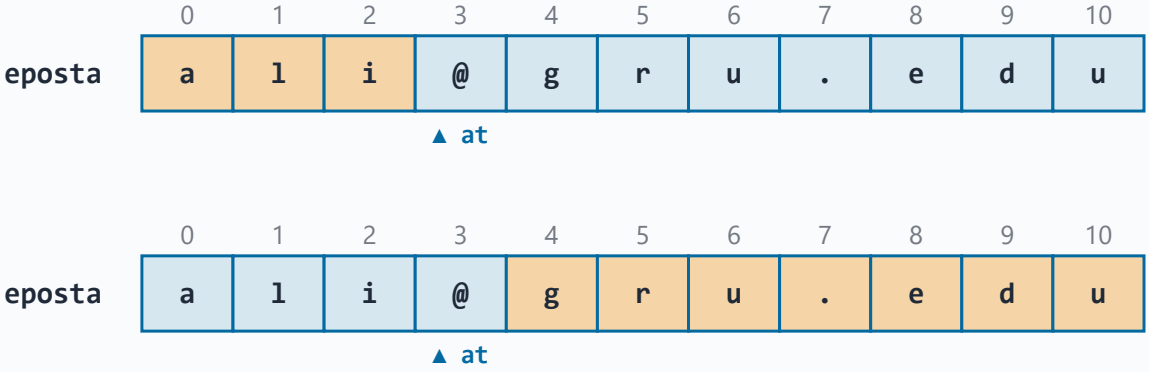
```
String eposta = "ali@gru.edu";
int at = eposta.indexOf('@');
String kullanici = eposta.substring(0, at);
String alan = eposta.substring(at + 1);
char ilk = eposta.charAt(0);
System.out.println(kullanici + " / " + alan);
System.out.println(Character.toUpperCase(ilk));
```

```
ali / gru.edu
A
```

Program yedi satırda şunu yapar: `@`'in konumunu bulur, ondan öncesini ve sonrasını ayrı ayrı alır, ilk karakteri okur ve büyük harfe çevirir. Tablo 11.7 izleme tablosudur; Şekil 11.11 iki kesme noktasını gösterir.

Satır	İşlem	Değişken	Çıktı
2	`indexOf('@')` : @ 3 indisinde	at = 3	
3	`substring(0, 3)` : 0, 1, 2 indisleri	kullanici = "ali"	
4	`substring(4)` : 4'ten sona kadar	alan = "gru.edu"	
5	`charAt(0)`	ilk = 'a'	
6	birleştirme		ali / gru.edu
7	`Character.toUpperCase('a')`		A

Tablo 11.7: E-posta programının izleme tablosu. eposta = "ali@gru.edu" (11 karakter, indisler 0–10).



Şekil 11.11: "ali@gru.edu" dizgisinin parçalanması. Üstte `substring(0, at)` : @'in solundaki 0–2 indisleri, "ali". Altta `substring(at + 1)` : @'den sonraki 4 indisinden sona kadar, "gru.edu". @ karakterinin kendisi iki parçaya da girmez.

İki ayrıntı dikkat ister. Birincisi, `at + 1` yazılmasının nedenidir: `substring(at)` yazılsaydı alan adı `"@gru.edu"` olurdu. İkincisi, `charAt` 'in dönüş tipidir. `ilk` bir `char` 'dır ve büyük harfe çevirmek için `String` 'in değil `Character` sınıfının `toUpperCase` metodu kullanılır. Bu metot `char` alır ve `char` döndürür.

Programın zayıf noktası, adreste `@` bulunmadığı durumdur. O zaman `indexOf` −1 döndürür ve `substring(0, -1)` çağrısı istisna fırlatır:

```
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: Range [0, -1) out of
bounds for length 10
```

Sağlam bir program önce aramanın sonucunu denetler:

```
public class EpostaGuvenli {
    public static void main(String[] args) {
        String[] adresler = {"ali@gru.edu", "aligru.edu"};
        for (String eposta : adresler) {
            int at = eposta.indexOf('@');
            if (at < 0) {
                System.out.println(eposta + ": @ yok");
            } else {
                System.out.println(eposta.substring(0, at) + " / " + eposta.substring(at +
1));
            }
        }
    }
}
```

```
ali / gru.edu
aligru.edu: @ yok
```

Örnek 11.4

`s = "ece.kaya@uni.tr"` adresinden "KAYA, ece" çıktısını üretmek için hangi indisler kullanılır?

Çözüm. Adres 15 karakterdir; indisler 0–14'tür. `s.indexOf('@')` 8'dir; `s.substring(0, 8)` `"ece.kaya"`yı verir. Bu yeni dizgide nokta 3 indisindedir. `substring(0, 3)` `"ece"`, `substring(4)` `"kaya"` verir. `"kaya".toUpperCase()` yeni bir `"KAYA"` dizgisi döndürür. Program çalıştırıldığında çıktı şudur:

```
KAYA, ece
```

11.5 Karşılaştırma

İki dizginin aynı olup olmadığını sormak, programlamanın en sık işlerinden biridir: girilen parola doğru mu, kullanıcı "evet" mi yazdı, iki öğrencinin adı aynı mı? Sayılarda bu soru `==` ile sorulur. Dizgilerde ise `==` beklenen soruyu sormaz.

11.5.1 == ile equals

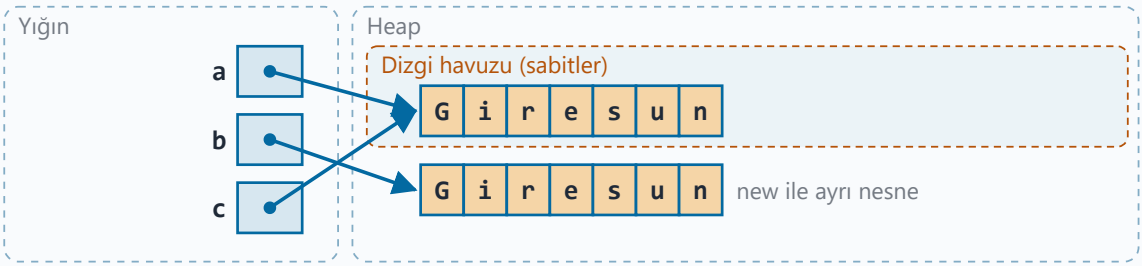
Bir `String` değişkeni bir referans tuttuğu için `a == b` iki referansı karşılaştırır. Soru "iki değişken aynı nesneyi mi gösteriyor?" olur. İki ayrı nesne aynı karakterleri taşısa bile `==` `false` verir. İçeriği karşılaştırmak için `equals` metodu kullanılır.

> **Tanım 11.7**
>
> **Referans karşılaştırması** (`a == b`), iki değişkenin aynı nesneyi gösterip göstermediğini sınar. **İçerik karşılaştırması** (`a.equals(b)`), iki dizginin aynı uzunlukta olup aynı konumlarda aynı karakterleri taşıyıp taşımadığını sınar. Dizgilerin eşitliği her zaman `equals` ile sorulur.

```
String a = "Giresun";
String b = new String("Giresun");
String c = "Giresun";
System.out.println(a == b);      // false
System.out.println(a == c);      // true
System.out.println(a.equals(b)); // true
```

```
false
true
true
```

Program üç değişken oluşturur ve üç karşılaştırma yapar. Şekil 11.12 belleğin son durumunu, Tablo 11.8 satır satır izlemeyi verir.



Şekil 11.12: Üç satırdan sonra bellek. `a` ve `c` , dizgi havuzundaki tek "Giresun" nesnesini gösterir. `b` , `new` ile heap'te açılan ayrı bir "Giresun" nesnesini gösterir. İki nesnenin içeriği aynı, yeri farklıdır.

Satır	İşlem	Sonuç	Gerekçe
1	`a = "Giresun"`	a → havuz	sabit havuza konur
2	`b = new String("Giresun")`	b → yeni nesne	`new` her zaman ayrı nesne açar

Satır	İşlem	Sonuç	Gerekçe
3	`c = "Giresun"`	c → havuz	aynı sabit; havuzdaki nesne paylaşılır
4	`a == b`	false	iki ok farklı nesneye gider
5	`a == c`	true	iki ok aynı nesnededir
6	`a.equals(b)`	true	karakterler tek tek eşittir: G=G, i=i, ..., n=n

Tablo 11.8: == ile equals programının izleme tablosu.

5. satırdaki `true` yanıltıcıdır. `a == c` 'nin doğru çıkması, içeriklerin eşit olmasından değil, iki sabitin havuzda aynı nesneyi paylaşmasındandır. Dizgilerden biri klavyeden okunmuş, bir dosyadan gelmiş ya da bir işlemle üretilmiş olsaydı ayrı bir nesne olurdu ve `==` `false` verirdi. Bir dizginin nereden geldiği koda bakınca her zaman anlaşılmaz. Bu yüzden kural basittir: içerik için `equals`.

Sık yapılan hata

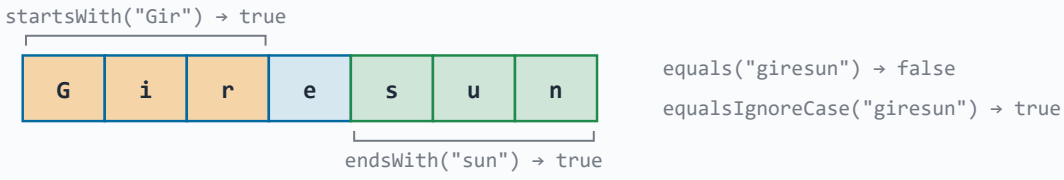
Kullanıcının yazdığı cevabı `if (cevap == "evet")` ile denetlemek. `Scanner` 'ın okuduğu dizgi her zaman yeni bir nesnedir; kullanıcı "evet" yazsa bile koşul yanlış çıkar. Program derlenir ve hata vermez, yalnızca yanlış çalışır. Doğrusu `if (cevap.equals("evet"))` 'tir.

11.5.2 Karşılaştırma metotları

`equals` dışında dört karşılaştırma metodu daha sık kullanılır. Tablo 11.9 bunları örnekleriyle verir; sonuçlar programın gerçek çıktısıdır.

Metot	Amaç	Örnek	Sonuç
`equals`	içerik eşitliği	`"abc".equals("abc")`	true
`equalsIgnoreCase`	harf büyüklüğünü yok sayan eşitlik	`"Java".equalsIgnoreCase("JAVA")`	true
`startsWith`	belirli metinle başlama	`"Giresun".startsWith("Gir")`	true
`endsWith`	belirli metinle bitme	`"Giresun".endsWith("sun")`	true
`compareTo`	sözlüksel sıra: negatif, 0 ya da pozitif	`"elma".compareTo("armut")`	4

Tablo 11.9: Karşılaştırma metotları.



Şekil 11.13: "Giresun" dizgisi "Gir" ile başlar ve "sun" ile biter. `equals("giresun")` `false` verir, çünkü 'G' ile 'g' farklı karakterlerdir. `equalsIgnoreCase("giresun")` harf büyüklüğünü yok saydığı için `true` verir.

```
public class KarsilastirmaMetotlari {
    public static void main(String[] args) {
        System.out.println("abc".equals("abc"));
        System.out.println("Java".equalsIgnoreCase("JAVA"));
        System.out.println("Giresun".startsWith("Gir"));
        System.out.println("Giresun".endsWith("sun"));
        System.out.println("elma".compareTo("armut"));
        System.out.println("Giresun".equals("giresun"));
        System.out.println("Giresun".equalsIgnoreCase("giresun"));
        System.out.println("Giresun".startsWith("gir"));
    }
}
```

```
true
true
true
true
4
false
true
false
```

Son üç satır büyük-küçük harf duyarlılığını gösterir. `equals`, `startsWith` ve `endsWith` karakterleri olduğu gibi karşılaştırır: `"Giresun".startsWith("gir")` `false`'tur. Harf büyüklüğünün önemsiz olduğu yerlerde (bir komutun "EVET", "Evet" ya da "evet" yazılması gibi) `equalsIgnoreCase` kullanılır.

11.5.3 compareTo ve sözlüksel sıra

Eşitlik dışında iki dizginin sırası da sorulabilir: hangisi sözlükte önce gelir? Bu sıraya **sözlüksel sıra** (*lexicographic order*) denir. `a.compareTo(b)` bir tam sayı döndürür. Sonuç negatifse `a` önce, sıfırsa ikisi eşit, pozitifse `a` sonra gelir.

Tanım 11.8

`a.compareTo(b)` iki dizgiyi baştan başlayarak karakter karakter karşılaştırır. İki durum vardır:

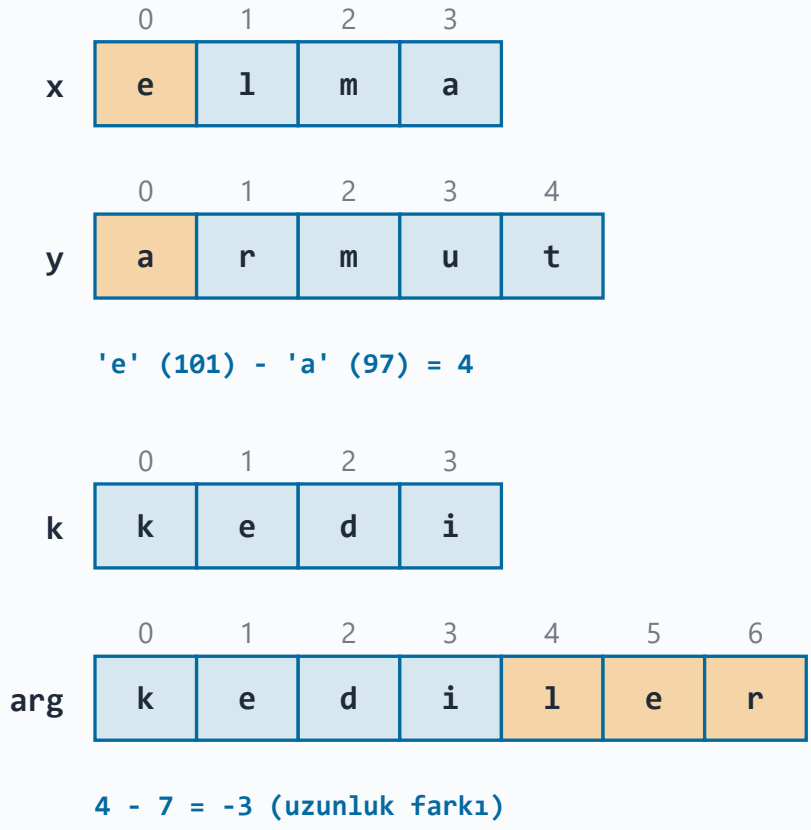
1. İlk farklı karakter k konumundaysa sonuç *a.charAt(k) − b.charAt(k)*'dır; yani iki karakterin kod farkıdır.
2. Kısa olan dizginin bütün karakterleri uzun olanın başıyla aynıysa sonuç *a.length() − b.length()*'dir; yani uzunluk farkıdır.

İki dizgi tamamen aynıysa ikinci kural 0 verir.

```
String x = "elma";
String y = "armut";
System.out.println(x.compareTo(y)); // 4
String k = "kedi";
System.out.println(k.compareTo("kediler")); // -3
```

Sunumdaki iki örnek, iki kuralı ayrı ayrı işletir. Şekil 11.14 her ikisinin kritik adımını gösterir.

- `"elma".compareTo("armut")`: 0 konumunda 'e' ile 'a' karşılaşır ve farklıdır. Sonuç *'e' − 'a' = 101 − 97 = 4*'tür. Pozitif olduğu için "elma", "armut"tan sonra gelir. Dizgilerin uzunlukları (4 ve 5) sonuca hiç girmez; ilk farklı karakter kararı verir.
- `"kedi".compareTo("kediler")`: ilk dört konumda k=k, e=e, d=d, i=i'dir. Ardından "kedi" biter; farklı karakter bulunamamıştır. Sonuç uzunluk farkıdır: *4 − 7 = −3*. Negatif olduğu için kısa olan "kedi" önce gelir.



Şekil 11.14: İki compareTo kuralı. Üstte "elma" ile "armut": ilk karakterler farklıdır ve sonuç kod farkıdır, *101 − 97 = 4*. Altta "kedi" ile "kediler": ilk dört karakter eşittir ve sonuç uzunluk farkıdır, *4 − 7 = −3*.

Aşağıdaki program sunumdaki iki satırın yanında sekiz karşılaştırma daha yapar:

```
public class CompareTo {
    public static void main(String[] args) {
        String x = "elma";
        String y = "armut";
        System.out.println(x.compareTo(y)); // 4
        String k = "kedi";
        System.out.println(k.compareTo("kediler")); // -3
        System.out.println(y.compareTo(x));
        System.out.println("Ali".compareTo("ali"));
        System.out.println("abc".compareTo("abc"));
        System.out.println("Zeytin".compareTo("armut"));
        System.out.println("çilek".compareTo("dut"));
        System.out.println("ılgın".compareTo("incir"));
        System.out.println("Ali".compareToIgnoreCase("ali"));
        System.out.println("10".compareTo("9"));
    }
}
```

```
4
-3
-4
-32
0
-7
131
200
0
-8
```

Tablo 11.10 her sonucu kurala göre açıklar.

Çağrı	İlk fark	Hesap	Sonuç
`"elma".compareTo("armut")`	konum 0: e / a	101 − 97	4
`"kedi".compareTo("kediler")`	yok	4 − 7	−3
`"armut".compareTo("elma")`	konum 0: a / e	97 − 101	−4
`"Ali".compareTo("ali")`	konum 0: A / a	65 − 97	−32
`"abc".compareTo("abc")`	yok	3 − 3	0
`"Zeytin".compareTo("armut")`	konum 0: Z / a	90 − 97	−7
`"çilek".compareTo("dut")`	konum 0: ç / d	231 − 100	131
`"ılgın".compareTo("incir")`	konum 0: ı / i	305 − 105	200
`"Ali".compareToIgnoreCase("ali")`	harf büyüklüğü yok sayılır	–	0
`"10".compareTo("9")`	konum 0: 1 / 9	49 − 57	−8

Tablo 11.10: compareTo sonuçlarının hesabı. Kod sütunu ilk farklı karakterlerin Unicode kodlarıdır.

Tablonun alt yarısı `compareTo`'nun gerçek sözlük sırasından nerede ayrıldığını gösterir:

- **Büyük harfler küçüklerden önce gelir.** Büyük harflerin kodları 65–90, küçük harflerinki 97–122 aralığındadır. Bu yüzden "Zeytin", "armut"tan önce gelir.
- **Türkçe harfler sona düşer.** 'ç' (231), 'ı' (305), 'ş' (351) gibi harflerin kodları bütün İngilizce harflerden büyüktür. "çilek", "dut"tan sonra; "ılgın", "incir"den sonra gelir. Türk alfabesinde ise ç, d'den önce; ı, i'den önce gelir.
- **Sayılar metin gibi karşılaştırılır.** "10", "9"dan önce gelir, çünkü ilk karakterler '1' ve '9'dur. Sayıların değerini karşılaştırmak için önce `Integer.parseInt` ile sayıya çevirmek gerekir.

Türkçe alfabeye uygun sıralama için Java'nın `java.text.Collator` sınıfı kullanılır. `Collator`, bir dilin sıralama kurallarını bilir. Aşağıdaki program aynı meyve listesini önce `compareTo`'nun sırasıyla, sonra Türkçe `Collator` ile sıralar:

```
import java.text.Collator;
import java.util.Arrays;
import java.util.Locale;

public class Siralama {
    public static void main(String[] args) {
        String[] meyveler = {"zeytin", "çilek", "armut", "Elma", "ıhlamur", "incir",
"üzüm", "dut"};
        String[] a = meyveler.clone();
        Arrays.sort(a);
        System.out.println(Arrays.toString(a));
        String[] b = meyveler.clone();
        Arrays.sort(b, Collator.getInstance(Locale.forLanguageTag("tr")));
        System.out.println(Arrays.toString(b));
    }
}
```

```
[Elma, armut, dut, incir, zeytin, çilek, üzüm, ıhlamur]
[armut, çilek, dut, Elma, ıhlamur, incir, üzüm, zeytin]
```

İlk satırda büyük harfle başlayan "Elma" en başa geçmiş, Türkçe harfle başlayan "çilek", "üzüm" ve "ıhlamur" en sona düşmüştür. İkinci satır Türk alfabesinin sırasıdır: a, ç, d, e, ı, i, ü, z. `Collator` ayrıca büyük-küçük harf farkını ikinci planda tutar; "Elma", e harfinin yerine yerleşir.

Örnek 11.5

`"kalem".compareTo("kale")`, `"kale".compareTo("kalp")` ve `"Kale".compareTo("kale")` değerleri nedir?

Çözüm. İlk çağrıda “kale”, “kalem”in başıdır; farklı karakter yoktur. Sonuç uzunluk farkıdır: *5 − 4 = 1*. İkinci çağrıda ilk üç karakter eşittir; 3 konumunda 'e' ile 'p' karşılaşır. Sonuç *101 − 112 = −11*'dir. Üçüncü çağrıda ilk karakterler 'K' (75) ve 'k'dir (107). Sonuç *75 − 107 = −32*'dir. Aynı harfin büyüğü ile küçüğü arasındaki fark İngilizce harflerin hepsinde 32'dir.

11.5.4 Türkçe karakterler ve yerel ayar

Büyük-küçük harf dönüşümü çoğu dilde basittir: a ↔ A, b ↔ B. Türkçede ise noktalı ve noktasız iki ayrı i harfi vardır: i ↔ İ ve ı ↔ I. İngilizcede bu çiftler i ↔ I biçimindedir. Aynı “I” harfinin küçüğü İngilizcede “i”, Türkçede “ı”dır. Java bu farkı **yerel ayar** (*locale*) ile çözer. Yerel ayar, bir dilin ve bölgenin yazım kurallarını taşıyan `java.util.Locale` nesnesidir.

Tanım 11.9

`toLowerCase(Locale)` ve `toUpperCase(Locale)`, harf dönüşümünü verilen dilin kurallarına göre yapar. Türkçe için yerel ayar `Locale.forLanguageTag("tr")`, İngilizce için `Locale.ENGLISH` ile elde edilir. Parametresiz `toLowerCase()` ve `toUpperCase()`, programın çalıştığı makinenin varsayılan yerel ayarını kullanır.

```
import java.util.Locale;

public class TurkceLocale {
    public static void main(String[] args) {
        Locale tr = Locale.forLanguageTag("tr");
        System.out.println("TITLE".toLowerCase(Locale.ENGLISH));
        System.out.println("TITLE".toLowerCase(Locale.forLanguageTag("tr")));
        System.out.println("i".toUpperCase(tr));
        System.out.println("i".toUpperCase(Locale.ENGLISH));
        System.out.println("I".toLowerCase(tr));
        System.out.println("ISPARTA".toLowerCase(tr));
        System.out.println("ısparta".toUpperCase(tr));
        System.out.println("TITLE".toLowerCase(tr).equals("title"));
        System.out.println("title".equalsIgnoreCase("TITLE"));
        System.out.println("istanbul".toUpperCase(tr) + " " +
"istanbul".toUpperCase(Locale.ENGLISH));
        System.out.println("İZMİR".toLowerCase(tr) + " " +
"İZMİR".toLowerCase(tr).length());
        System.out.println("İZMİR".toLowerCase(Locale.ENGLISH).length() + " " +
"İZMİR".length());
        System.out.println("ISPARTA".equalsIgnoreCase("ısparta") + " " +
"i".equalsIgnoreCase("İ"));
    }
}
```

```
title
tıtle
İ
I
ı
ısparta
ISPARTA
false
true
İSTANBUL ISTANBUL
izmir 5
7 5
true true
```

Tablo 11.11 ilk yedi çağrının sonuçlarını iki dilin kurallarıyla açıklar.

Çağrı	Sonuç	Kural
`"TITLE".toLowerCase(Locale.ENGLISH)`	title	İngilizce: I → i
`"TITLE".toLowerCase(Locale.forLanguageTag("tr"))`	tıtle	Türkçe: I → ı
`"i".toUpperCase(tr)`	İ	Türkçe: i → İ
`"i".toUpperCase(Locale.ENGLISH)`	I	İngilizce: i → I
`"I".toLowerCase(tr)`	ı	Türkçe: I → ı
`"ISPARTA".toLowerCase(tr)`	ısparta	Türkçe: I → ı
`"ısparta".toUpperCase(tr)`	ISPARTA	Türkçe: ı → I

Tablo 11.11: Türkçe ve İngilizce harf dönüşümü. tr = Locale.forLanguageTag("tr").

Sonraki satırlar dönüşümün eşitlik sınamalarına etkisini gösterir. `"TITLE".toLowerCase(tr)` “tıtle” verdiği için `"title"` 'a eşit değildir. `equalsIgnoreCase` ise karakterleri tek tek, dilden bağımsız bir kuralla karşılaştırır ve `true` verir. “istanbul” Türkçe kurallarla “İSTANBUL”, İngilizce kurallarla “ISTANBUL” olur. “İZMİR”in Türkçe küçüğü beş harfli “izmir”dir. İngilizce kurallarla küçültüldüğünde ise her “İ”, bir “i” ve onun üstüne eklenen ayrı bir nokta işaretine dönüşür. Sonuç ekranda “izmir”e benzer ama uzunluğu 7'dir. Bu metin “izmir” ile `equals` sınamasından geçmez.

En sinsi sorun parametresiz çağrılardadır. Aşağıdaki program aynı kodu iki farklı makinede çalıştırır: biri İngilizce (ABD), öteki Türkçe yerel ayarlıdır.

```
import java.util.Locale;

public class VarsayilanDil {
    public static void main(String[] args) {
        System.out.println(Locale.getDefault());
        String dosya = "ARSIV.ZIP".toLowerCase();
        System.out.println(dosya);
        System.out.println(dosya.endsWith(".zip"));
        System.out.println("title".toUpperCase());
        System.out.println("ARSIV.ZIP".toLowerCase(Locale.ENGLISH).endsWith(".zip"));
        System.out.printf("%.2f%n", 17.5);
    }
}
```

İngilizce ayarlı makinede çıktı şudur:

```
en_US
arsiv.zip
true
TITLE
true
17.50
```

Türkçe ayarlı makinede aynı program şunu yazar:

```
tr_TR
arsıv.zıp
false
TİTLE
true
17,50
```

Program bir dosya adının “.zip” ile bitip bitmediğini denetliyor. İngilizce makinede “ARSIV.ZIP” küçültülünce “arsiv.zip” olur ve denetim geçer. Türkçe makinede aynı satır “arsıv.zıp” üretir ve denetim başarısız olur. Kodda hiçbir değişiklik yoktur; yalnızca bilgisayarın dili farklıdır. `"title".toUpperCase()` da Türkçe makinede “TİTLE” verir. Yerel ayarı açıkça veren beşinci satır ise iki makinede de `true` yazar. Son satırdaki `printf` da yerel ayardan etkilenir: ondalık ayırıcı bir makinede nokta, ötekinde virgüldür.

Sık yapılan hata

Anahtar sözcük, komut adı, dosya uzantısı, e-posta adresi gibi teknik metinleri parametresiz `toLowerCase()` ya da `toUpperCase()` ile dönüştürmek. Program geliştiricinin bilgisayarında doğru çalışır, Türkçe ayarlı bir bilgisayarda bozulur. Bu hataya yazılım dünyasında “Türk i'si

sorunu" adı verilir. Teknik metinde `Locale.ENGLISH` (ya da dilden bağımsız `Locale.ROOT`), Türkçe metinde `Locale.forLanguageTag("tr")` açıkça verilir.

Eski kaynaklarda Türkçe yerel ayar `new Locale("tr")` biçiminde de oluşturulur. Bu yapıcı JDK 19'dan beri kullanımdan kaldırılmıştır (*deprecated*); JDK 21 derleyicisi `-Xlint` seçeneğiyle bu satır için uyarı verir. Yerine `Locale.forLanguageTag("tr")` ya da `Locale.of("tr")` kullanılır.

Örnek 11.6

Bir program kullanıcının yazdığı il adını büyük harfe çevirip `"IĞDIR"` ile karşılaştırıyor. Kullanıcı "ığdır" yazıyor. `toUpperCase(Locale.ENGLISH)` ve `toUpperCase(tr)` ile sonuç ne olur?

Çözüm. Noktasız "ı"nın büyüğü iki dilde de "I"dır; İngilizce kurallar da bu Türkçe harfi tanır. Her iki çağrı "IĞDIR" verir ve karşılaştırma geçer. Kullanıcı "iğdir" yazsaydı durum değişirdi: Türkçe kurallarla "İĞDİR", İngilizce kurallarla "IĞDIR" çıkardı. Türkçe bir metinde doğru olan Türkçe kurallardır; bu durumda "iğdir"in yanlış yazıldığı ortaya çıkar.

11.6 Bellek: dizgi havuzu

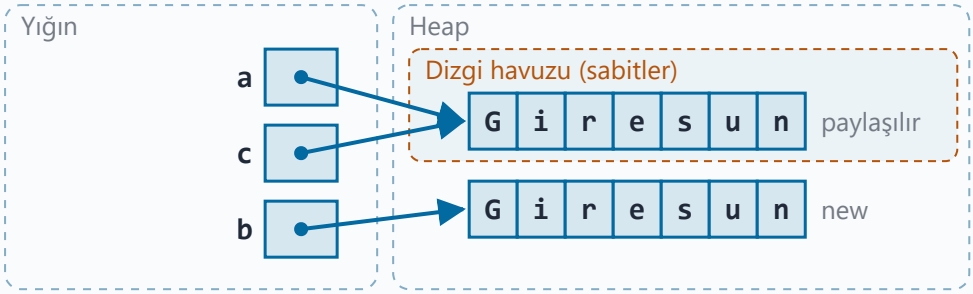
Bir programda aynı dizgi sabiti pek çok yerde geçebilir: "Giresun", "evet", boş dizgi. Her biri için ayrı nesne açmak belleği boşa harcar. Java bu yüzden sabitleri ortak bir havuzda toplar.

11.6.1 Havuz ve new

Tanım 11.10

Dizgi havuzu (*string pool*), Java sanal makinesinin dizgi sabitlerini sakladığı bellek bölgesidir. Aynı içerikli bütün sabitler havuzdaki tek bir nesneyi paylaşır. `new String(...)` ise havuza bakmaz ve her çağrıda heap'te yeni bir nesne oluşturur.

Paylaşım ancak dizgiler değiştirilemez olduğu için güvenlidir. Değiştirilebilseydi, bir değişken üzerinden yapılan değişiklik aynı sabiti kullanan bütün değişkenleri etkilerdi. Şekil 11.15 üç değişkenin durumunu gösterir.



Şekil 11.15: Dizgi havuzu. `a` ve `c` aynı sabitle oluşturulduğu için havuzdaki tek "Giresun" nesnesini paylaşır. `b`, `new` ile oluşturulduğu için heap'te ayrı bir nesneyi gösterir.

```
public class Havuz {
    public static void main(String[] args) {
        String a = "Giresun";
        String c = "Giresun";
        String b = new String("Giresun");
        String d = new String("Giresun");
        System.out.println((a == c) + " " + (a == b) + " " + (b == d));
        System.out.println(a.equals(b) + " " + b.equals(d));
        System.out.println((b.intern() == a) + " " + (b.intern() == b));
    }
}
```

```
true false false
true true
true false
```

İlk satır üç referans karşılaştırması yapar. `a == c` doğrudur: ikisi de havuzdaki nesneyi gösterir. `a == b` yanlıştır: `b` heap'te ayrı bir nesnedir. `b == d` de yanlıştır: iki `new` çağrısı iki ayrı nesne üretir. İkinci satırdaki `equals` sınamaları ise hepsi doğrudur, çünkü içerikler aynıdır.

11.6.2 Sabit birleştirme ve intern

Bir dizginin havuzdan gelip gelmediği her zaman koddan kolayca okunmaz. Aşağıdaki program bunun iki ince durumunu gösterir:

```
String a = "a", b = "b";
String r = a + b;
final String f = "a";
System.out.println(("a" + "b") == "ab");  // true
System.out.println((f + "b") == "ab");    // true
System.out.println(r == "ab");            // false
System.out.println(r.equals("ab"));       // true
System.out.println(r.intern() == "ab");   // true
```

```
true
true
false
true
true
```

Beş sonuç şöyle açıklanır:

1. `("a" + "b") == "ab"` doğrudur. İki işlenen de sabit olduğu için birleştirmeyi program çalışmadan önce derleyici yapar. Derlenmiş kodda `"a" + "b"` yerine doğrudan `"ab"` sabiti bulunur ve bu sabit havuzdan gelir.
2. `(f + "b") == "ab"` de doğrudur. `f`, `final` ile bildirilmiş ve bir sabitle başlatılmıştır. Değeri hiç değişemeyeceği için derleyici onu da bir sabit gibi işler.
3. `r == "ab"` yanlıştır. `r = a + b` satırındaki `a` ve `b` sıradan değişkenlerdir. Birleştirme çalışma zamanında yapılır ve sonuç heap'te yeni bir nesnedir.
4. `r.equals("ab")` doğrudur; içerikler aynıdır.
5. `r.intern() == "ab"` doğrudur. `intern()` metodu havuza bakar: aynı içerikte bir dizgi varsa onu, yoksa dizgiyi havuza ekleyip onu döndürür. Dönen referans havuzdaki `"ab"`'dir.

Derleme sırasında değeri belli olan ifadelere **sabit ifade** (*constant expression*) denir: dizgi sabitleri, `final` sabitler ve bunların birleşimleri. Sabit ifadelerin sonucu havuza konur. Değişken içeren her birleştirme ise yeni bir nesne üretir.

`intern`'ün etkisi `Havuz` programının son satırında da görülür. `b.intern() == a` doğrudur: `b`'nin içeriği havuzda vardır ve `intern` havuzdaki nesneyi döndürür. `b.intern() == b` ise yanlıştır: `b`'nin kendisi havuz dışındaki nesnedir. Uygulamada `intern`'e nadiren gerek duyulur; burada havuzun işleyişini görünür kılmak için kullanılmıştır.

Sık yapılan hata

Birkaç denemede `==`'nin doğru sonuç verdiğini görüp dizgileri `==` ile karşılaştırmaya devam etmek. Denemelerde kullanılan sabitler havuzda paylaşıldığı için `==` tesadüfen doğru çıkar. Aynı kod, metin klavyeden ya da bir dosyadan geldiğinde yanlış sonuç verir. `==`'nin doğru çık-

ması içeriklerin eşit olduğunu değil, nesnenin aynı olduğunu gösterir.

11.6.3 null güvenli karşılaştırma

`equals` bir metottur ve bir nesne üzerinden çağrılır. Değişken `null` ise çağrı yapılamaz ve `NullPointerException` oluşur. Aynı karşılaştırmayı ters yönde yazmak bu sorunu ortadan kaldırır:

```
public class NullGuvenli {
    public static void main(String[] args) {
        String x = null;
        System.out.println("sabit".equals(x));
        System.out.println(x.equals("sabit"));
    }
}
```

```
false
Exception in thread "main" java.lang.NullPointerException: Cannot invoke
"String.equals(Object)" because "x" is null
```

`"sabit".equals(x)` satırında metot gerçek bir nesne olan `"sabit"` üzerinden çağrılır. Parametre `null` olduğunda `equals` hata vermez, yalnızca `false` döndürür. `x.equals("sabit")` satırında ise metot `null` üzerinden çağrılmaya çalışılır ve program durur. Bu yüzden sabit değer sola yazılır: `"evet".equals(cevap)`, `cevap` `null` olsa bile güvenlidir.

11.7 Scanner ile metin girdisi

Klavyeden metin okumak için `java.util.Scanner` sınıfı kullanılır. Sınıfın iki metin okuma metodu vardır: `next()` bir kelime, `nextLine()` bir satır okur. İki metodun davranışını anlamanın anahtarı, klavyeden gelen karakterlerin beklediği **tampondur** (*buffer*).

11.7.1 next ve nextLine

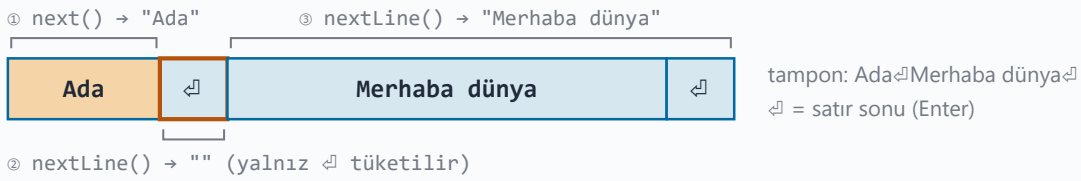
Kullanıcı bir şey yazıp Enter tuşuna bastığında yazdığı karakterler ve satır sonu karakteri tampona girer. Bu notta satır sonu karakteri ⏎ ile gösterilir. `Scanner`'ın her okuma metodu tamponun başından karakter alır ve okuma konumunu öne kaydırır. Okunan karakterler bir daha okunmaz.

Tanım 11.11

`next()`, baştaki boşlukları ve satır sonlarını atlar, ardından boşluk olmayan karakterleri ilk boşluğa ya da satır sonuna kadar okur. Bu parçaya **kelime** (*token*) denir. Kelimeden sonraki boşluk ya da ⏎ tamponda kalır. `nextLine()` ise bulunduğu konumdan satır sonuna kadar her şeyi, boşluklar dahil okur. ⏎'ı tüketir ama sonuca eklemez.

```
import java.util.Scanner;
// ...
Scanner klavye = new Scanner(System.in);
String kelime = klavye.next();
klavye.nextLine();
String satir = klavye.nextLine();
```

Şekil 11.16 bu kodun, tamponda "Ada⏎Merhaba dünya⏎" varken nasıl çalıştığını özetler. `next()` "Ada"yı okur ve ⏎'ı bırakır. Aradaki `nextLine()` yalnızca o ⏎'ı tüketir ve boş dizgi döndürür. Sonraki `nextLine()` ikinci satırın tamamını okur.



Şekil 11.16: Tampon "Ada⏎Merhaba dünya⏎" iken üç okuma. ① `next()` "Ada"yı okur. ② `nextLine()` yalnızca ⏎'ı tüketir ve "" döndürür. ③ `nextLine()` "Merhaba dünya"yı okur.

`next()`'in boşlukta durması bir satırdaki kelimeleri tek tek okumayı kolaylaştırır. Aşağıdaki programa tek satırda "␣␣Ada␣␣␣Can␣Ece␣Su" girilir (␣ boşluğu gösterir):

```
import java.util.Scanner;

public class NextKelimeler {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        String a = klavye.next();
        String b = klavye.next();
        String c = klavye.nextLine();
        System.out.println("[" + a + "] [" + b + "] [" + c + "]");
    }
}
```

```
[Ada] [Can] [ Ece Su]
```

İlk `next()` baştaki iki boşluğu atlar ve "Ada"yı okur. İkinci `next()` aradaki üç boşluğu atlar ve "Can"ı okur. `nextLine()` ise satırın geri kalanını olduğu gibi alır: başındaki boşluk da sonuca girer ve "␣Ece␣Su" okunur. `nextLine()` boşluk atlamaz.

11.7.2 Tamponu adım adım izlemek

Sunumdaki izleme örneği beş satırlıktır:

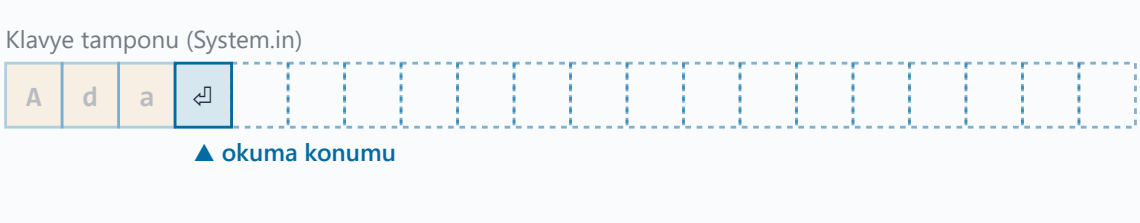
```
Scanner klavye = new Scanner(System.in);
String kelime = klavye.next();
klavye.nextLine();
String satir = klavye.nextLine();
System.out.println(kelime + " | " + satir);
```

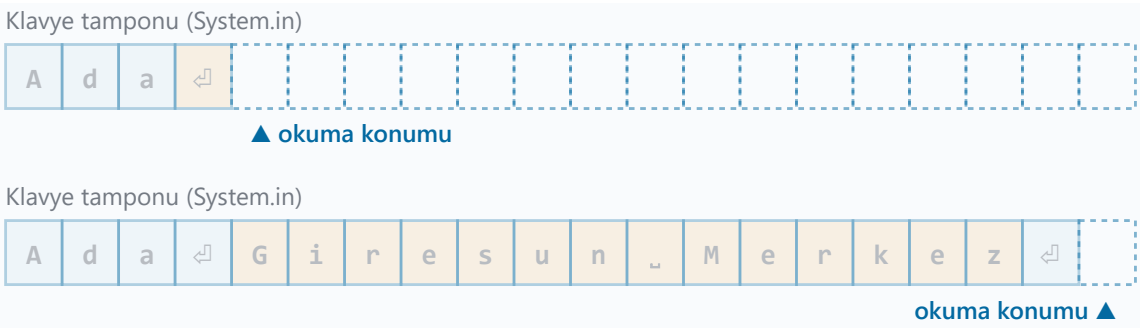
Program bir kez çalıştırılır. Kullanıcı önce "Ada" yazıp Enter'a, sonra "Giresun Merkez" yazıp Enter'a basar. Çıktı şudur:

```
Ada | Giresun Merkez
```

Adım	Satır	Olay	Okunmamış tampon	Sonuç
1	1	`Scanner` klavyeye bağlanır	(boş)	
2	2	`next()` okuyacak bir şey bulamaz ve bekler; kullanıcı "Ada⏎" girer	Ada⏎	
3	2	`next()` boşluk ya da ⏎'a kadar okur	⏎	kelime = "Ada"
4	3	`nextLine()` ⏎'a kadar okur, ⏎'ı tüketir	(boş)	"" (atılır)
5	4	`nextLine()` bekler; kullanıcı "Giresun Merkez⏎" girer	Giresun␣Merkez⏎	
6	4	`nextLine()` satırın tamamını okur, ⏎'ı tüketir	(boş)	satir = "Giresun Merkez"
7	5	yazdırma	(boş)	Ada | Giresun Merkez

Tablo 11.12: Scanner tamponunun izlenmesi. Tampon sütununda okuma konumundan sonraki, henüz okunmamış karakterler gösterilir.





Şekil 11.17: Tablodaki 3., 4. ve 6. adımlarda klavye tamponu. Okunmuş karakterler soluk, okuma konumu ▲ ile gösterilir. 3. adımda `next()` "Ada"yı okumuş, ⏎ tamponda kalmıştır. 4. adımda `nextLine()` ⏎'ı tüketmiştir. 6. adımda ikinci satır da tamamen okunmuştur.

3. satırdaki `klavye.nextLine();` çağrısının sonucu hiçbir değişkene atanmaz. Bu satırın tek görevi tamponda kalan ⏎'ı temizlemektir. Satır silinirse ne olur? 4. satırdaki `nextLine()` kalan ⏎'ı bulur, hemen boş dizgi döndürür ve kullanıcıya ikinci satırı yazma fırsatı vermez. Aşağıdaki program bu durumu gösterir; girdi yine aynıdır:

```
import java.util.Scanner;

public class TamponHatasi {
    public static void main(String[] args) {
        Scanner k = new Scanner(System.in);
        String kelime = k.next();
        String satir = k.nextLine();
        System.out.println(kelime + "|" + satir + "|");
    }
}
```

```
Ada||
```

`satir` boş dizgidir; iki dikey çizginin arasında hiçbir şey yoktur. Kullanıcının yazdığı "Giresun Merkez" hiç okunmamış, tamponda kalmıştır.

11.7.3 nextInt sonrası nextLine

Aynı sorun `nextInt()`, `nextDouble()` gibi sayı okuyan metotlardan sonra da ortaya çıkar. Bu metotlar da `next()` gibi davranır: sayıyı okur, satır sonunu tamponda bırakır. Aşağıdaki program önce yaşı, sonra ad soyadı okumak ister. Kullanıcı "19⏎" ve "Ada Yılmaz⏎" girer:

```
import java.util.Scanner;

public class YasAd {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        System.out.print("Yaş: ");
        int yas = klavye.nextInt();
        System.out.print("Ad soyad: ");
        String ad = klavye.nextLine();
        System.out.println();
        System.out.println("[" + ad + "] " + yas);
    }
}
```

```
Yaş: Ad soyad:
[] 19
```

"Ad soyad:" sorusu ekrana gelir ama program cevabı beklemez. `nextLine()`, 19'dan sonra kalan ⏎'ı okur ve `ad` boş kalır. Çözüm, sayıdan sonra artığı temizleyen bir `nextLine()` eklemektir:

```
import java.util.Scanner;

public class YasAdDuzeltme {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        System.out.print("Yaş: ");
        int yas = klavye.nextInt();
        klavye.nextLine();
        System.out.print("Ad soyad: ");
        String ad = klavye.nextLine();
        System.out.println();
        System.out.println("[" + ad + "] " + yas);
    }
}
```

```
Yaş: Ad soyad:
[Ada Yılmaz] 19
```

Çıktıda "Ad soyad:" yazısından sonra satırın boş göründüğü dikkat çekebilir. Bunun nedeni girdinin bir dosyadan verilmiş olmasıdır: klavyeden çalıştırıldığında kullanıcının yazdığı metin o satırda görünür. `System.out.println()` bu yüzden programa eklenmiştir.

Sık yapılan hata

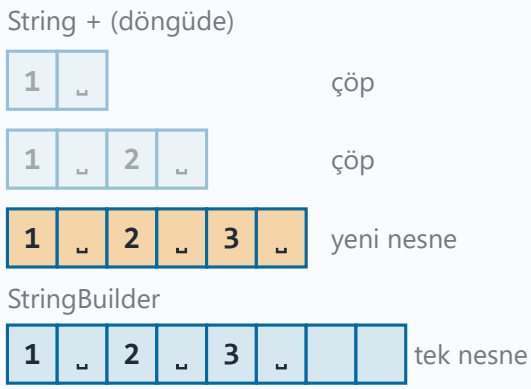
`nextInt()` ile `nextLine()`'ı art arda kullanıp ikinci okumanın neden "atlandığını" anlayamamak. Okuma atlanmamıştır: `nextLine()` tamponda kalan boş satırı okumuştur. Kural şudur: `next()`, `nextInt()` ya da `nextDouble()`'dan sonra bir satır okunacaksa araya ⏎'ı tüketen bir `nextLine()` konur. Başka bir yol, bütün girdiyi `nextLine()` ile satır satır okuyup sayıları `Integer.parseInt` ile dönüştürmektir.

11.8 Birleştirme ve StringBuilder

`+` işleci birkaç dizgiyi birleştirmek için okunaklı ve yeterlidir. Ancak birleştirme bir döngünün içinde binlerce kez yapılırsa değiştirilemezliğin bedeli belirginleşir.

11.8.1 Döngüde + kullanmanın maliyeti

`s = s + x` satırı `s`'yi uzatmaz. Her çalıştığında `s`'nin bütün karakterlerini ve `x`'i taşıyan yeni bir nesne üretir. Eski nesne çöpe gider. Şekil 11.18 bunu üç turluk bir döngü için gösterir.



Şekil 11.18: Üstte `+` ile birleştirme: her tur "1␣", "1␣2␣", "1␣2␣3␣" adlı yeni bir nesne üretir; öncekiler çöp olur. Altta `StringBuilder`: tek bir nesne büyür.

Kopyalanan karakter sayısı hesaplanabilir. Her turda tek karakter eklenen bir döngü düşünelim. 1. turda 1 karakterlik, 2. turda 2 karakterlik, n. turda n karakterlik yeni bir nesne kurulur. Toplam kopyalanan karakter sayısı şudur:

$$1 + 2 + 3 + \ldots + n = n(n + 1) / 2$$

Bu toplam kabaca $n^2/2$'dir. n = 1.000 için 500.500, n = 100.000 için yaklaşık 5 milyar karakter eder. Aşağıdaki program bu sayıları döngünün içinde sayarak doğrular:

```
public class Maliyet {
    public static void main(String[] args) {
        for (int n : new int[]{1000, 10000, 100000}) {
            long kopya = 0;
            String s = "";
            for (int i = 0; i < n; i++) {
                kopya += s.length() + 1;
                s = s + "x";
            }
            StringBuilder sb = new StringBuilder();
            for (int i = 0; i < n; i++) {
                sb.append('x');
            }
            System.out.println(n + " " + kopya + " " + (s.length() == sb.length()));
        }
    }
}
```

```
1000 500500 true
10000 50005000 true
100000 5000050000 true
```

n on katına çıktığında kopyalanan karakter sayısı yaklaşık yüz katına çıkar: 500.500'den 50.005.000'e, oradan 5.000.050.000'e. Bu büyüme hızına **karesel** büyüme denir ve büyük n değerlerinde program belirgin biçimde yavaşlar. `StringBuilder` ile yapılan ikinci döngüde ise her karakter tampona bir kez yazılır; toplam iş n ile orantılıdır. Satırın sonundaki `true`, iki yöntemin aynı uzunlukta dizgi ürettiğini gösterir.

11.8.2 StringBuilder ile birleştirme

Tanım 11.12

`StringBuilder`, karakterleri **değiştirilebilir** (*mutable*) bir tamponda tutan sınıftır. `append` metodu tamponun sonuna ekleme yapar ve yeni nesne üretmez. Tamponun içeriği bittiğinde `toString()` ile bir `String`'e dönüştürülür. `StringBuilder` de `java.lang` paketindedir.

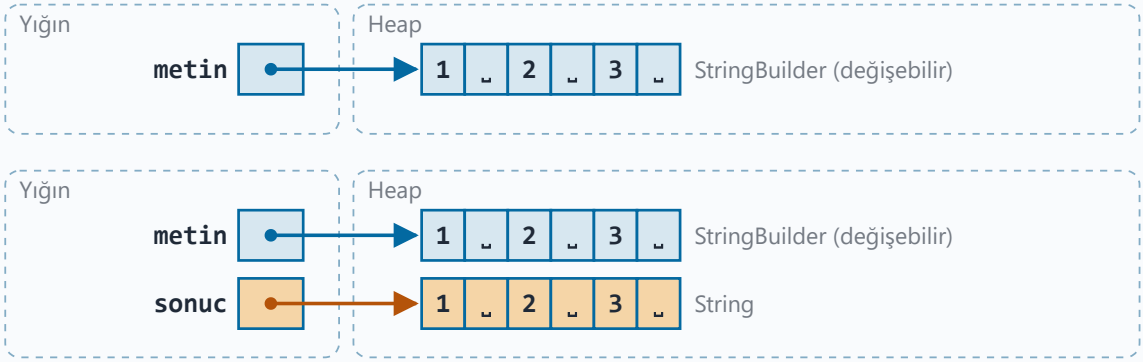
```
StringBuilder metin = new StringBuilder();
for (int i = 1; i <= 3; i++) {
    metin.append(i).append(' ');
}
String sonuc = metin.toString();
System.out.println(sonuc + "|");
```

```
1 2 3 |
```

Tablo 11.13 programın on adımını verir. Döngü değişkeni `i` 1'den 3'e kadar döner ve her turda iki ekleme yapılır: sayı ve ardından bir boşluk.

Adım	Satır	i	Koşul / işlem	metin (tampon)	sonuc
1	1		boş tampon oluşturulur	""	
2	2	1	1 <= 3 → true	""	
3	3	1	`append(1).append(' ')`	"1␣"	
4	2	2	i++; 2 <= 3 → true	"1␣"	
5	3	2	`append(2).append(' ')`	"1␣2␣"	
6	2	3	i++; 3 <= 3 → true	"1␣2␣"	
7	3	3	`append(3).append(' ')`	"1␣2␣3␣"	
8	2	4	i++; 4 <= 3 → false, döngü biter	"1␣2␣3␣"	
9	5		`toString()`	"1␣2␣3␣"	"1␣2␣3␣"
10	6		yazdırma: `sonuc + "|"`	"1␣2␣3␣"	"1␣2␣3␣"

Tablo 11.13: StringBuilder programının izleme tablosu. ␣ boşluk karakteridir.



Şekil 11.19: Üstte 7. adımdan sonra: tek `StringBuilder` nesnesi "1␣2␣3␣" karakterlerini taşır; döngü boyunca yeni nesne açılmamıştır. Altta 9. adımdan sonra: `toString()` tamponun o anki içeriğinden ayrı bir `String` nesnesi üretmiş, `sonuc` onu gösterir.

3. satırdaki `metin.append(i).append(' ')` yazımı dikkat çekicidir. `append`, ekleme yaptığı nesnenin kendisini döndürür. Bu yüzden ikinci `append` aynı nesne üzerinde çağrılabilir. Bu tür ardışık çağrılara **zincirleme** denir. Sondaki `"|"` yalnızca çıktının sonundaki boşluğu görünür kılmak içindir.

`append` her tipten değer alır: `int`, `char`, `double`, `boolean`, `String`. Sayılar metin karşılıklarıyla eklenir. `append(1)` tampona 1 karakterini ekler; bir toplama yapmaz.

11.8.3 StringBuilder'ın öteki metotları

`StringBuilder` değiştirilebilir olduğu için `String`'de bulunmayan metotlar da taşır. Bunlar tamponu yerinde değiştirir:

```
public class SbMetotlar {
    public static void main(String[] args) {
        StringBuilder sb = new StringBuilder("Java");
        sb.append(21);
        System.out.println(sb + " " + sb.length());
        sb.insert(0, "JDK ");
        System.out.println(sb);
        sb.setCharAt(4, 'j');
        System.out.println(sb);
        sb.deleteCharAt(sb.length() - 1);
        System.out.println(sb);
        sb.reverse();
        System.out.println(sb);
        StringBuilder t = sb;
        t.append('!');
        System.out.println(sb + " " + (t == sb));
        System.out.println(new StringBuilder("ab").equals(new StringBuilder("ab")) + " "
                + new StringBuilder("ab").toString().equals("ab"));
    }
}
```

```
Java21 6
JDK Java21
JDK java21
JDK java2
2avaj KDJ
2avaj KDJ! true
false true
```

Çağrı	Etki	Tampon
`new StringBuilder("Java")`	başlangıç içeriği	Java
`append(21)`	sona ekler	Java21 (uzunluk 6)
`insert(0, "JDK ")`	0 konumuna ekler	JDK Java21
`setCharAt(4, 'j')`	4 konumundaki karakteri değiştirir	JDK java21
`deleteCharAt(length() - 1)`	son karakteri siler	JDK java2
`reverse()`	tamponu ters çevirir	2avaj KDJ

Tablo 11.14: StringBuilder metotları ve programdaki etkileri.

Programın son iki satırı iki önemli farkı gösterir. `StringBuilder t = sb;` yeni bir tampon açmaz; `t` aynı nesneyi gösterir. `t.append('!')` bu yüzden `sb`'yi de değiştirir ve `t == sb` doğrudur. Değiştirilebilir nesnelerde paylaşım bu tür yan etkilere yol açar; `String`'de böyle bir durum olamaz. İkinci fark karşılaştırmadadır: `StringBuilder`'ın `equals` metodu içeriğe değil referansa bakar. Aynı "ab" içerikli iki tampon için `equals` `false` verir. İçerik karşılaştırması için önce `toString()` çağrılır.

Örnek 11.7

`StringBuilder sb = new StringBuilder("ab");` ile başlayan ve 1'den 3'e kadar her turda `sb.append(i).append('-')` yapan bir döngüden sonra `String s = sb.toString(); sb.reverse();` satırları çalışıyor. `s + " " + sb` ne yazar?

Çözüm. Döngü tamponu "ab" → "ab1-" → "ab1-2-" → "ab1-2-3-" yapar. `toString()` o anki içerikten ayrı bir `String` üretir: `s` = "ab1-2-3-". `reverse()` yalnızca tamponu ters çevirir: "-3-2-1ba". `s` etkilenmez, çünkü ayrı ve değiştirilemez bir nesnedir. Birleştirmede `sb` için `toString()` kendiliğinden çağrılır. Program çıktısı:

```
ab1-2-3- -3-2-1ba
```

Hangi durumda hangisi seçilir? Birkaç parçayı bir kez birleştirmek için `+` okunaklıdır ve yeterlidir; Java derleyicisi bu tür tek satırlık birleştirmeleri zaten verimli koda çevirir. Bir döngü içinde ya da sayısı önceden bilinmeyen çok sayıda parçayla metin kurulurken `StringBuilder` kullanılır.

11.9 Uygulamalar

Bu kesimdeki programların hepsi aynı iskeleti paylaşır. Bir döngü dizginin karakterlerini sırayla ya da iki uçtan gezer. Her karakter `charAt` ile okunur ve bir karar verilir. Programları birbirinden ayıran şey, her adımda yapılan iştir.

11.9.1 Dizgiyi ters çevirme

Bir dizgiyi ters çevirmek için karakterler sondan başa doğru okunur ve yeni bir dizgiye eklenir. Sunumdaki program bunu iki yolla yapar: önce elle yazılmış bir döngüyle, sonra `StringBuilder`'ın hazır metoduyla.

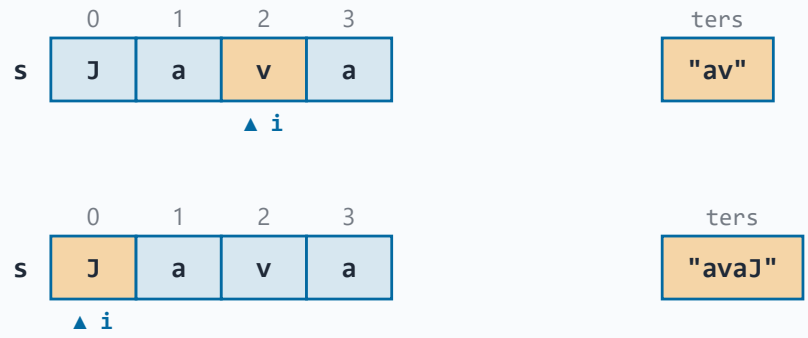
```
String s = "Java";
String ters = "";
for (int i = s.length() - 1; i >= 0; i--)
  ters += s.charAt(i);
System.out.println(ters);
String t2 = new StringBuilder(s).reverse()
                                .toString();
System.out.println(t2);
```

```
avaJ
avaJ
```

Döngü `i = s.length() - 1`, yani son indisle başlar ve `i >= 0` olduğu sürece `i`'yi bir azaltır. Her turda `s.charAt(i)`, `ters`'in sonuna eklenir. Tablo 11.15 dört turu verir.

i	i >= 0	s.charAt(i)	ters (turdan sonra)
3	true	'a'	"a"
2	true	'v'	"av"
1	true	'a'	"ava"
0	true	'J'	"avaJ"
−1	false	–	döngü biter

Tablo 11.15: Ters çevirme döngüsünün izleme tablosu. s = "Java".



Şekil 11.20: Ters çevirme döngüsü. Üstte i = 2 turu: 'v' eklenmiş, `ters` "av" olmuştur. Altta i = 0 turu: 'J' de eklenmiş, sonuç "avaJ"dır. `s` hiç değişmez.

Döngüdeki her `ters += s.charAt(i)` yeni bir `String` üretir; dört karakterlik bir dizgide bu önemsizdir. İkinci yol olan `new StringBuilder(s).reverse().toString()` ise tek satırdır. `s`'nin içeriğiyle bir tampon açar, tamponu ters çevirir ve sonucu `String`'e dönüştürür. İki yol aynı çıktıyı verir ve ikisinde de `s` değişmez.

Bu döngünün iki sınır durumu da doğru çalışır. Boş dizgide `i` −1 ile başlar, koşul ilk denetimde yanlıştır ve `ters` boş kalır. Tek karakterli dizgide döngü bir kez döner ve dizginin kendisi elde edilir.

11.9.2 Palindrom cümle

Tersten okunuşu düzden okunuşuyla aynı olan kelime ya da cümleye **palindrom** denir: “kazak”, “ana”, “Ey Edip Adanada pide ye”. Cümlede büyük harfler ve boşluklar bulunduğu için doğrudan karşılaştırma yanlış sonuç verir: 'E' ile 'e' farklı karakterlerdir. Önce metin temizlenir, sonra iki uçtan içeriye doğru karşılaştırılır.

```
String cumle = "Ey Edip Adanada pide ye";
String t = cumle.toLowerCase().replace(" ", "");
int sol = 0, sag = t.length() - 1;
boolean palindrom = true;
while (sol < sag && palindrom) {
  if (t.charAt(sol) != t.charAt(sag))
    palindrom = false;
  sol++; sag--;
}
System.out.println(palindrom);
```

```
true
eyedipadanadapideye 19 9 9
```

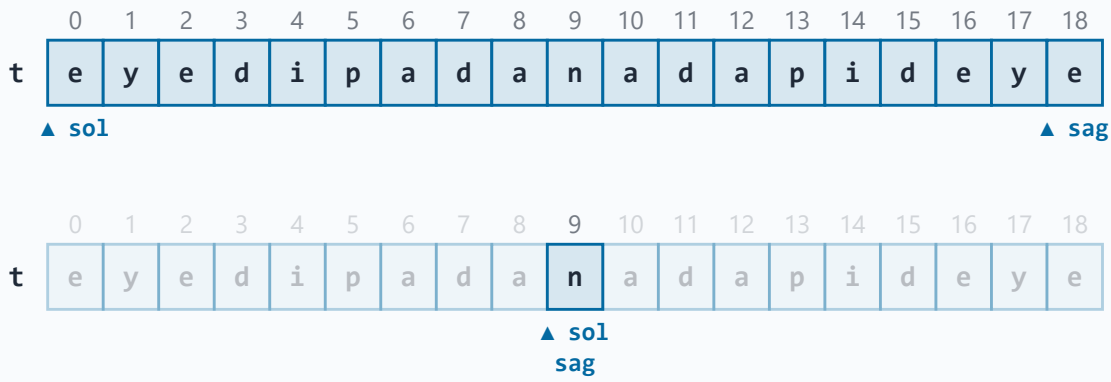
Program şu adımlarla çalışır:

1. `toLowerCase()` bütün harfleri küçültür, `replace(" ", "")` boşlukları siler. 23 karakterlik cümle 19 karakterlik `"eyedipadanadapideye"` olur.
2. İki **işaretçi** değişken kurulur: `sol` 0'dan, `sag` son indis olan 18'den başlar. Aksi kanıtlanana kadar `palindrom = true` kabul edilir.
3. Her turda iki uçtaki karakter karşılaştırılır. Eşit değillerse `palindrom` `false` yapılır ve döngü koşulu bir sonraki denetimde döngüyü bitirir. Eşitlerse iki işaretçi birer adım içeri kayar.
4. `sol < sag` koşulu yanlış olunca bütün çiftler denetlenmiş demektir.

Tur	sol	sag	t.charAt(sol)	t.charAt(sag)	Eşit mi?
1	0	18	e	e	evet
2	1	17	y	y	evet
3	2	16	e	e	evet
4	3	15	d	d	evet
5	4	14	i	i	evet
6	5	13	p	p	evet

Tur	sol	sag	t.charAt(sol)	t.charAt(sag)	Eşit mi?
7	6	12	a	a	evet
8	7	11	d	d	evet
9	8	10	a	a	evet
–	9	9	sol < sag yanlış: döngü biter, palindrom = true		

Tablo 11.16: Palindrom döngüsünün izleme tablosu. t = "eyedipadanadapideye" (19 karakter).



Şekil 11.21: Palindrom denetiminde iki işaretçi. Üstte başlangıç: `sol` 0'da, `sag` 18'de. Altta son durum: işaretçiler ortadaki 9 indisinde buluşmuştur. Ortadaki 'n' kendisiyle karşılaştırılmaz.

Programın çıktısındaki ikinci satır, döngüden sonraki değişkenleri gösteren bir denetim satırıdır: temizlenmiş metin, uzunluğu 19 ve işaretçilerin son değerleri 9 ve 9. Uzunluk tek olduğu için ortada eşi olmayan bir karakter kalır. Dokuz tur yeter: *[19 / 2] = 9*. Genel kural şudur: n karakterlik bir metin için en çok n/2 tur gerekir.

Aynı mantık bir metot içine alınıp birkaç cümleyle denenebilir. Bu sürümde küçültme Türkçe kurallarla yapılır; böylece “I” ve “İ” içeren cümleler de doğru işlenir:

```
import java.util.Locale;

public class PalindromDeneme {
    static boolean palindromMu(String cumle) {
        String t = cumle.toLowerCase(Locale.forLanguageTag("tr")).replace(" ", "");
        int sol = 0, sag = t.length() - 1;
        boolean palindrom = true;
        while (sol < sag && palindrom) {
            if (t.charAt(sol) != t.charAt(sag))
                palindrom = false;
            sol++; sag--;
        }
        return palindrom;
    }

    public static void main(String[] args) {
        String[] cumleler = {"Ey Edip Adanada pide ye", "Kaç kişi", "Ana", "Kazak", "Ali"};
        for (String c : cumleler) {
            System.out.println(c + " -> " + palindromMu(c));
        }
    }
}
```

```
Ey Edip Adanada pide ye -> true
Kaç kişi -> false
Ana -> true
Kazak -> true
Ali -> false
```

“Kaç kişi” palindrom değildir: temizlenmiş hâli “kaçkişi”, tersi “işikçak”tır. İlk turda 'k' ile 'i' karşılaşır ve döngü orada biter. “Ali” de ilk turda elenir: 'a' ≠ 'i'.

11.9.3 Sesli harf sayma

Bir karakterin belirli bir kümeye ait olup olmadığını anlamanın kısa bir yolu, kümeyi bir dizgide toplayıp `indexOf` ile aramaktır. Karakter kümede varsa `indexOf` 0 ya da daha büyük bir değer, yoksa −1 döndürür.

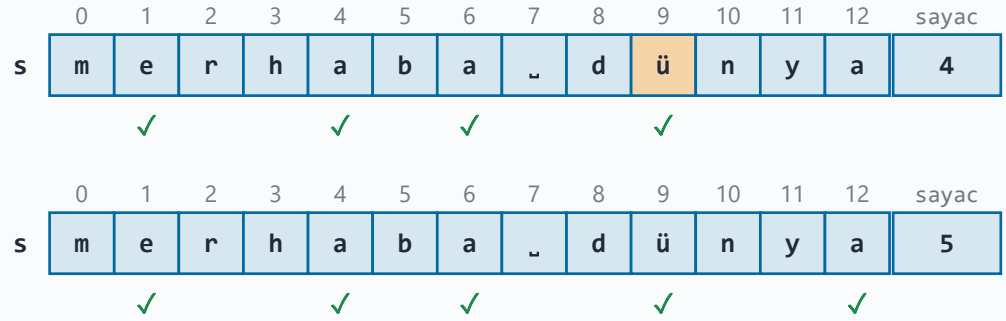
```
String s = "Merhaba Dünya".toLowerCase();
String sesliler = "aeıioöuü";
int sayac = 0;
for (int i = 0; i < s.length(); i++) {
  char c = s.charAt(i);
  if (sesliler.indexOf(c) >= 0) sayac++;
}
System.out.println(sayac);
```

```
5
```

“Merhaba Dünya” önce küçültülür: “merhaba dünya”. `sesliler` dizgisi Türkçenin sekiz sesli harfini tutar: a, e, ı, i, o, ö, u, ü. Döngü on üç karakteri sırayla dener. Tablo 11.17 her karakter için `indexOf`'un sonucunu verir.

i	c	sesliler.indexOf(c)	Sesli mi?	sayac
0	m	−1	hayır	0
1	e	1	evet	1
2	r	−1	hayır	1
3	h	−1	hayır	1
4	a	0	evet	2
5	b	−1	hayır	2
6	a	0	evet	3
7	␣	−1	hayır	3
8	d	−1	hayır	3
9	ü	7	evet	4
10	n	−1	hayır	4
11	y	−1	hayır	4
12	a	0	evet	5

Tablo 11.17: Sesli harf sayımının izleme tablosu. sesliler = "aeıioöuü"; ␣ boşluktur.



Şekil 11.22: Sesli harf taraması. Üstte i = 9: 'ü', `sesliler` dizgisinin 7 indisinde bulunur ve sayaç 4 olur. Altta tarama bitmiştir: işaretli beş hücre (e, a, a, ü, a) sesli harftir.

Programın başındaki `toLowerCase()` çağrısı burada önemlidir. Küçültme yapılmasaydı büyük “A”, “E” gibi harfler `sesliler` dizgisinde bulunamazdı. Ancak parametresiz küçültme makinenin diline bağlıdır. Metinde büyük “I” ya da “İ” varsa Türkçe olmayan bir makinede yanlış küçük harfe dönüşebilir. Türkçe metin için `toLowerCase(Locale.forLanguageTag("tr"))` güvenli yoldur.

Örnek 11.8

Aynı program "Giresun Üniversitesi" için kaç sayar? Sayılan harfler hangileridir?

Çözüm. Küçültülmüş metin "giresun üniversitesi"dir. Sesliler sırayla i, e, u (giresun: üç), ü, i, e, i, e, i (üniversitesi: altı) olur. Toplam *3 + 6 = 9*'dur. Boşluk ve ünsüzler için `indexOf` −1 döndürür ve sayaç artmaz.

11.9.4 Kelime sayma

Bir cümledeki kelime sayısı, boşlukları sayıp bir eklemekle bulunamaz. Cümlenin başında, sonunda ya da kelimeler arasında birden çok boşluk olabilir. Daha sağlam bir yol, "kelime dışından kelime içine" geçişleri saymaktır.

Tanım 11.13

Bu programda **kelime**, boşluk olmayan karakterlerin kesintisiz dizisidir. Bir metindeki kelime sayısı, boşluk olmayan bir karakterden hemen önce boşluk ya da metnin başı bulunan konumların sayısına eşittir.

```
String metin = "Ali  ata bak";
int kelime = 0;
boolean icinde = false;
for (int i = 0; i < metin.length(); i++) {
  char c = metin.charAt(i);
  if (c != ' ' && !icinde) {
    kelime++;
    icinde = true;
  } else if (c == ' ') icinde = false;
}
System.out.println(kelime);
```

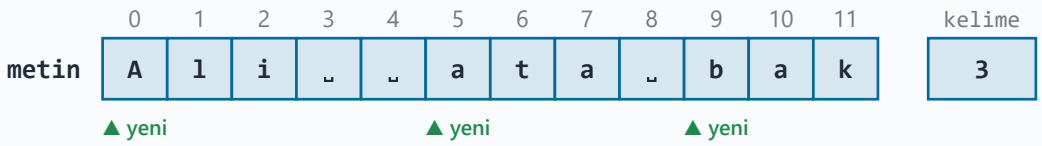
```
3
```

Program bir `boolean` bayrak tutar: `icinde`, o anda bir kelimenin içinde olunup olunmadığını gösterir. Boşluk olmayan bir karakter gelir ve bayrak `false` ise yeni bir kelime başlamıştır: sayaç artar ve bayrak `true` yapılır. Boşluk gelirse bayrak `false` yapılır. Boşluk olmayan karakter geldiğinde bayrak zaten `true` ise hiçbir şey yapılmaz; aynı kelime sürer.

i	c	icinde (önce)	Olay	kelime	icinde (sonra)
0	A	false	yeni kelime	1	true

i	c	icinde (önce)	Olay	kelime	icinde (sonra)
1	l	true	devam	1	true
2	i	true	devam	1	true
3	␣	true	kelime biter	1	false
4	␣	false	boşluk, sayılmaz	1	false
5	a	false	yeni kelime	2	true
6	t	true	devam	2	true
7	a	true	devam	2	true
8	␣	true	kelime biter	2	false
9	b	false	yeni kelime	3	true
10	a	true	devam	3	true
11	k	true	devam	3	true

Tablo 11.18: Kelime sayma döngüsünün izleme tablosu. metin = "Ali␣␣ata␣bak" (12 karakter).



Şekil 11.23: "Ali␣␣ata␣bak" metninde üç kelime başlangıcı. ▲ işaretleri, boşluktan harfe geçilen 0, 5 ve 9 indislerini gösterir. Çift boşluk (3 ve 4 indisleri) ikinci bir kelime başlatmaz.

Aynı sonuç `split` ile tek satırda da bulunabilir: `metin.trim().split("\\s+").length`. `trim()` baştaki ve sondaki boşlukları atar; `"\\s+"` kalıbı "bir ya da daha fazla boşluk karakteri" demektir ve art arda boşlukları tek ayırıcı sayar. Bu kısa yolun bir tuzağı vardır: boş ya da yalnızca boşluktan oluşan metinde sonuç 0 değil 1'dir. `"".split(...)` bir parçalı dizi döndürür ve o parça boş dizgidir. Aşağıdaki program beş metin için üç yöntemi karşılaştırır:

```
public class KelimeKisaYol {
    public static void main(String[] args) {
        String[] metinler = {"Ali  ata bak", "  Ali ata  bak ", "", "   ", "tek"};
        for (String metin : metinler) {
            int k1 = metin.trim().split("\\s+").length;
            int k2 = metin.isBlank() ? 0 : metin.trim().split("\\s+").length;
            System.out.println("[" + metin + "] " + k1 + " " + k2 + " " + metin.split("
").length);
        }
    }
}
```

```
[Ali  ata bak] 3 3 4
[  Ali ata  bak ] 3 3 6
[] 1 0 1
[   ] 1 0 0
[tek] 1 1 1
```

Her satırda sırasıyla kısa yolun sonucu, `isBlank()` denetimli sonucu ve `split(" ")`'nin sonucu yazılır. İlk iki sütun boş metinler dışında aynıdır. `split(" ")` ise her tek boşluğu ayırıcı saydığı için çift boşluklarda boş parçalar üretir: "Ali␣␣ata␣bak" için 3 yerine 4 verir. Bu yüzden kısa yol, önce `isBlank()` ile denetlenerek kullanılır.

11.9.5 Karakter frekansı

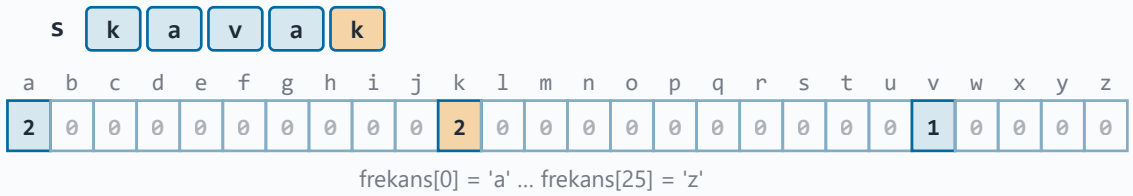
Bir metinde her harfin kaç kez geçtiğini saymak için 26 hücreli bir `int` dizisi kullanılır. Hücre 0 'a' harfinin, hücre 1 'b' harfinin, hücre 25 'z' harfinin sayacıdır. Bir harfin hücresi, `char`'ların sayı olmasından yararlanılarak `c - 'a'` ile bulunur.

```
String s = "kavak";
int[] frekans = new int[26];
for (int i = 0; i < s.length(); i++) {
  char c = s.charAt(i);
  frekans[c - 'a']++;
}
for (int k = 0; k < 26; k++)
  if (frekans[k] > 0)
    System.out.println((char) ('a' + k)
                       + ": " + frekans[k]);
```

```
a: 2
k: 2
v: 1
```

i	c	c – 'a'	Güncellenen hücre	frekans[0]	frekans[10]	frekans[21]
0	k	107 – 97 = 10	frekans[10]	0	1	0
1	a	97 – 97 = 0	frekans[0]	1	1	0
2	v	118 – 97 = 21	frekans[21]	1	1	1
3	a	97 – 97 = 0	frekans[0]	2	1	1
4	k	107 – 97 = 10	frekans[10]	2	2	1

Tablo 11.19: "kavak" için frekans dizisinin güncellenmesi. Yalnızca değişen hücreler gösterilmiştir.



Şekil 11.24: "kavak" tarandıktan sonra 26 hücreli frekans dizisi. Yalnızca 'a' (2), 'k' (2) ve 'v' (1) hücreleri sıfırdan farklıdır.

İkinci döngü 26 hücreyi sırayla gezer ve yalnızca sıfırdan büyük olanları yazar. Hücre numarasından harfe geri dönmek için `(char) ('a' + k)` yazılır: *'a' + 10 = 97 + 10 = 107*, yani 'k'. Dönüşüm (`(char)`) gereklidir; yoksa `'a' + k` bir `int` olur ve ekrana 107 yazılır. Çıktının alfabe sırasında olması da bu ikinci döngüden gelir: hücreler 0'dan 25'e gezilir.

Programın iki maliyeti ayrı ayrı hesaplanabilir. İlk döngü n karakterlik metni bir kez tarar; ikinci döngü her zaman 26 tur döner. Toplam iş $n + 26$ ile orantılıdır. Her harf için metni baştan tarayan bir çözüm ise $26 \cdot n$ karşılaştırma yapardı.

Sık yapılan hata

Programı büyük harf, boşluk ya da Türkçe harf içeren bir metinle çalıştırmak. 26 hücreli tablo yalnızca küçük İngilizce harfler içindir. 'ç' için *'ç' − 'a' = 231 − 97 = 134* çıkar ve dizinin sınırı aşılır:

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: Index 134 out
of bounds for length 26
```

Büyük harfler önce küçültülür, harf olmayan karakterler `if (c >= 'a' && c <= 'z')` ile atlanır. Türkçe harfler için ya 29 harflik bir alfabe dizgisi ve `indexOf` kullanılır ya da daha geniş bir tablo kurulur.

11.9.6 Sezar şifresi

Sezar şifresi (*Caesar cipher*), her harfi alfabede sabit sayıda konum ileri kaydıran çok eski bir şifreleme yöntemidir. Kaydırma 3 ise 'a' → 'd', 'b' → 'e' olur. Alfabenin sonundaki harfler başa sarar: 'x' → 'a', 'y' → 'b', 'z' → 'c'.

Başa sarmanın aritmetiği **mod** işlemidir. Harfin alfabedeki yeri `c - 'a'` ile 0–25 arasında bir sayıya çevrilir. Kaydırma eklenir ve sonucun 26'ya bölümünden kalan alınır:

$$\text{yer} = (c - \text{'a'} + \text{kaydir}) \bmod 26$$

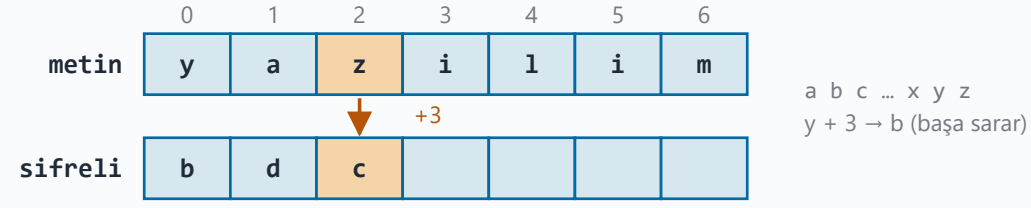
Kalan her zaman 0 ile 25 arasındadır; bu yüzden sonuç alfabenin dışına taşmaz. Ardından `'a' + yer` ile yeniden harfe dönülür.

```
String metin = "yazilim";
int kaydir = 3;
StringBuilder sifreli = new StringBuilder();
for (int i = 0; i < metin.length(); i++) {
  char c = metin.charAt(i);
  int yer = (c - 'a' + kaydir) % 26;
  char yeni = (char) ('a' + yer);
  sifreli.append(yeni);
}
System.out.println(sifreli);
```

```
bdclolp
```

i	c	c – 'a'	(c – 'a' + 3) % 26	yeni	sifreli
0	y	24	27 % 26 = 1	b	b
1	a	0	3 % 26 = 3	d	bd
2	z	25	28 % 26 = 2	c	bdc
3	i	8	11 % 26 = 11	l	bdcl
4	l	11	14 % 26 = 14	o	bdclo
5	i	8	11 % 26 = 11	l	bdclol
6	m	12	15 % 26 = 15	p	bdclolp

Tablo 11.20: “yazilim” kelimesinin 3 kaydırmayla şifrelenmesi.



Şekil 11.25: 'z' harfinin şifrelenmesi. *25 + 3 = 28* alfabenin dışındadır; *28 % 26 = 2* başa sarar ve 'c' elde edilir. `sifreli` tamponunda o ana kadar “bdc” birikmiştir.

Şifreyi çözmek için harfler aynı miktarda geri kaydırılır. Ancak `(c - 'a' - 3) % 26` yazmak tehlikelidir: Java'da negatif bir sayının kalanı negatiftir ve *(0 – 3) % 26 = –3* çıkar. Geri kaydırma yerine *26 – 3 = 23* ileri kaydırma yapılır; alfabede 23 ileri gitmek 3 geri gitmekle aynıdır ve sonuç hep pozitif kalır. Aşağıdaki program şifreleme işini bir metoda alır ve aynı metotla şifreyi çözer:

```
public class SezarCoz {
    static String kaydir(String metin, int kaydir) {
        StringBuilder sonuc = new StringBuilder();
        for (int i = 0; i < metin.length(); i++) {
            char c = metin.charAt(i);
            int yer = (c - 'a' + kaydir) % 26;
            sonuc.append((char) ('a' + yer));
        }
        return sonuc.toString();
    }

    public static void main(String[] args) {
        String sifreli = kaydir("yazilim", 3);
        System.out.println(sifreli);
        System.out.println(kaydir(sifreli, 26 - 3));
        System.out.println(kaydir("kaz", 2));
        System.out.println((-3) % 26);
    }
}
```

```
bdclolp
yazilim
mcb
-3
```

İkinci satır çözülmüş metindir: “yazilim”. Üçüncü satır “kaz” kelimesinin 2 kaydırmayla şifresidir: 'k' (10) → 12 → 'm', 'a' → 'c', 'z' (25) → *27 % 26 = 1* → 'b'. Son satır negatif kalanın Java'daki değerini doğrular. Sezar şifresinin yalnızca 25 anlamlı anahtarı vardır; bilgisayar hepsini bir anda dener. Bu yüzden yöntem güvenlik için değil, mod aritmetiğini ve karakter işlemeyi öğretmek için kullanılır.

11.9.7 Alıştırmalardan çözümlü örnekler

Sunumun alıştırmalarından altısı aşağıda çözülmüştür. Her çözüm, notta anlatılan araçlardan birini ya da birkaçını kullanır.

Örnek 11.9

Bir cümledeki büyük harf, küçük harf, rakam ve boşluk sayılarını ayrı ayrı bulan program.

Çözüm. Her karakter bir else-if zinciriyle sınıflandırılır. `Character` sınıfının `isUpperCase`, `isLowerCase` ve `isDigit` metotları Türkçe harfleri de tanır; bu yüzden “Ö” büyük, “ı” küçük harf sayılır.

```
public class KarakterTurleri {
    public static void main(String[] args) {
        String cumle = "Giresun 2024 Yılı Öğrenci";
        int buyuk = 0, kucuk = 0, rakam = 0, bosluk = 0, diger = 0;
        for (int i = 0; i < cumle.length(); i++) {
            char c = cumle.charAt(i);
            if (Character.isUpperCase(c)) buyuk++;
            else if (Character.isLowerCase(c)) kucuk++;
            else if (Character.isDigit(c)) rakam++;
            else if (c == ' ') bosluk++;
            else diger++;
        }
        System.out.println("Büyük: " + buyuk + ", küçük: " + kucuk + ", rakam: " +
rakam
                + ", boşluk: " + bosluk + ", diğer: " + diger);
    }
}
```

```
Büyük: 3, küçük: 15, rakam: 4, boşluk: 3, diğer: 0
```

“Giresun 2024 Yılı Öğrenci” cümlesinde G, Y ve Ö büyük harftir. Küçük harfler “iresun” (6), “ılı” (3) ve “ğrenci”dir (6): toplam 15. Dört rakam ve üç boşluk vardır. Toplam *3 + 15 + 4 + 3 = 25*, cümlenin uzunluğuna eşittir.

Örnek 11.10

İki kelimenin anagram olup olmadığını (aynı harflerin yeniden dizilişi, ör. “elma” – “alem”) karakter frekanslarıyla denetleyen program.

Çözüm. Uzunluklar farklıysa kelimeler anagram olamaz. Uzunluklar aynıysa tek bir frekans dizisi yeter: ilk kelimenin harfleri sayacı artırır, ikincininkiler azaltır. Sonunda bütün hücreler sıfırsa iki kelime aynı harfleri aynı sayıda taşır.

```java
public class Anagram {
    static boolean anagramMi(String a, String b) {
        if (a.length() != b.length()) return false;
        int[] frekans = new int[26];
        for (int i = 0; i < a.length(); i++) {
            frekans[a.charAt(i) - 'a']++;
            frekans[b.charAt(i) - 'a']--;
        }
        for (int k = 0; k < 26; k++)
            if (frekans[k] != 0) return false;
        return true;
    }

    public static void main(String[] args) {
        System.out.println(anagramMi("elma", "alem"));
        System.out.println(anagramMi("kavak", "kavka"));
        System.out.println(anagramMi("kalem", "kelam"));
        System.out.println(anagramMi("kitap", "katip"));
        System.out.println(anagramMi("ders", "dere"));
    }
}
```

```
true
true
true
true
false
```

“ders” ile “dere” aynı uzunluktadır ama 's' hücresi +1, 'e' hücresi −1 kalır; sonuç `false` 'tur. Program yalnızca küçük İngilizce harfler için yazılmıştır; Türkçe harfler için frekans kesimindeki uyarı burada da geçerlidir.

Örnek 11.11

Kullanıcının girdiği ad-soyadı “SOYAD, Ad” biçiminde, soyadı büyük harfle yazdıran program.

Çözüm. Satır `nextLine()` ile okunur, çünkü ad soyad boşluk içerir. Birden çok adı olan kişiler için soyadı son kelime kabul edilir ve `lastIndexOf(' ')` ile ayrılır. Büyük harfe çevirme Türkçe kurallarla yapılır.

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

public class AdSoyad {
    public static void main(String[] args) {
        Scanner klavye = new Scanner(System.in);
        String satir = klavye.nextLine().strip();
        int bosluk = satir.lastIndexOf(' ');
        String ad = satir.substring(0, bosluk).strip();
        String soyad = satir.substring(bosluk + 1);
        System.out.println(soyad.toUpperCase(Locale.forLanguageTag("tr")) + ", " +
ad);
    }
}
```

“Ada Nur Yılmaz” girildiğinde çıktı:

```
YILMAZ, Ada Nur
```

“␣␣Mehmet Işık␣” girildiğinde `strip()` kenar boşluklarını atar ve çıktı şudur:

```
IŞIK, Mehmet
```

“Işık”ın büyük harfli hâli Türkçe kurallarla “IŞIK”tır. Program tek kelimelik bir girdide `lastIndexOf` 'un −1 döndürmesi yüzünden hata verir; bu durum ayrıca denetlenebilir.

Örnek 11.12

Bir cümledeki en uzun kelimeyi `split` ile bulan program.

Çözüm. Cümle `"\\s+"` kalıbıyla kelimelere bölünür. İlk aday ilk kelimedir; her kelime adayla karşılaştırılır ve daha uzunsa yeni aday olur.

```
public class EnUzunKelime {
    public static void main(String[] args) {
        String cumle = "Bilgisayar mühendisliği  öğrencileri dizgileri seviyor";
        String[] kelimeler = cumle.trim().split("\\s+");
        String enUzun = kelimeler[0];
        for (String k : kelimeler) {
            if (k.length() > enUzun.length()) enUzun = k;
        }
        System.out.println(enUzun + " " + enUzun.length() + " " + kelimeler.length);
    }
}
```

```
mühendisliği 12 5
```

Cümlede çift boşluk olmasına karşın beş kelime bulunur. “mühendisliği” 12 karakterle en uzunudur. “öğrencileri” 11 karakterdir. Koşul `>` olduğu için eşit uzunlukta iki kelime varsa önce geleni kalır.

Örnek 11.13

Sezar şifresini büyük harfleri de (A–Z) destekleyecek biçimde genişleten, öteki karakterleri olduğu gibi bırakan program.

Çözüm. Her karakter üç gruba ayrılır. Küçük harfler 'a'ya, büyük harfler 'A'ya göre kaydırılır. Rakam, boşluk ve noktalama değişmeden eklenir.

```java
public class SezarBuyuk {
    static String sifrele(String metin, int kaydir) {
        StringBuilder sonuc = new StringBuilder();
        for (int i = 0; i < metin.length(); i++) {
            char c = metin.charAt(i);
            if (c >= 'a' && c <= 'z') {
                sonuc.append((char) ('a' + (c - 'a' + kaydir) % 26));
            } else if (c >= 'A' && c <= 'Z') {
                sonuc.append((char) ('A' + (c - 'A' + kaydir) % 26));
            } else {
                sonuc.append(c);
            }
        }
        return sonuc.toString();
    }

    public static void main(String[] args) {
        String s = sifrele("Merhaba, Dunya XYZ!", 3);
        System.out.println(s);
        System.out.println(sifrele(s, 26 - 3));
    }
}
```

```
Phukded, Gxqbd ABC!
Merhaba, Dunya XYZ!
```

'M' → 'P', 'X' → 'A', 'Y' → 'B', 'Z' → 'C' olur; virgül, boşluk ve ünlem aynen kalır. İkinci satır, 23 kaydırmanın metni geri çevirdiğini gösterir. Türkçe harfler bu programda “öteki karakterler” grubuna girer ve şifrelenmez; bu yüzden örnek metinde “Dünya” yerine “Dunya” kullanılmıştır.

Örnek 11.14

`replace` kullanmadan bir dizgideki bütün boşlukları `StringBuilder` ile silen program.

Çözüm. Dizgi taranır ve boşluk olmayan her karakter tampona eklenir. Boşluklar atlanır.

```
public class BoslukSil {
    public static void main(String[] args) {
        String s = " Ey Edip  Adanada pide ye ";
        StringBuilder sb = new StringBuilder();
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (c != ' ') sb.append(c);
        }
        System.out.println("[" + sb + "] " + sb.length() + " " + s.length());
    }
}
```

```
[EyEdipAdanadapideye] 19 26
```

26 karakterlik metinde yedi boşluk vardır: baştaki, sondaki, “Edip”ten sonraki çift boşluk ve üç tek boşluk. Geriye 19 karakter kalır. Bu sayı palindrom örneğindeki temizlenmiş metnin uzunluğuyla aynıdır; harf büyüklüğü dışında sonuç da aynıdır.

11.10 printf ile biçimleme

Bir rapor ya da tablo yazdırırken değerlerin hizalı durması, ondalıklı sayıların belirli basamakta kesilmesi istenir. `+` ile birleştirme bunu kolayca sağlamaz. `System.out.printf` metodu ise metni bir kalıba göre **biçimler** (*formatting*).

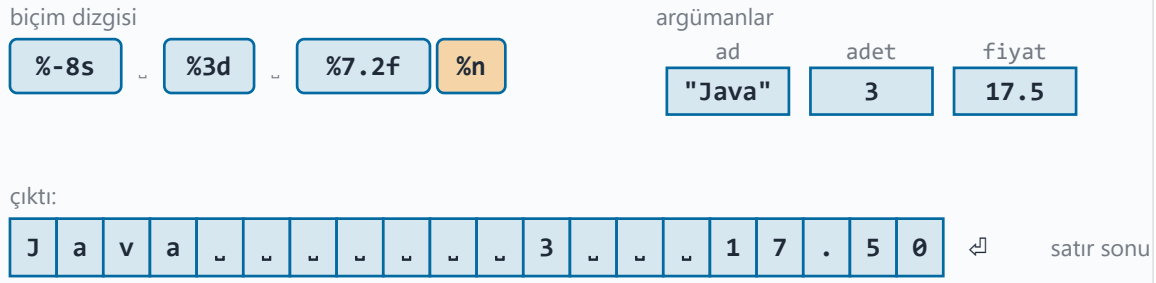
Tanım 11.14

`printf`'in ilk parametresi **biçim dizgisidir** (*format string*). Biçim dizgisindeki düz metin olduğu gibi yazılır. `%` ile başlayan her **biçim belirteci** (*format specifier*) ise sıradaki argümanın yerini tutar ve onun nasıl yazılacağını söyler. Belirtecin genel yapısı `%[-][genişlik][.duyarlık]tür` biçimindedir. Başlıca türler `s` (metin), `d` (tam sayı), `f` (kayan noktalı sayı) ve `n`'dir (satır sonu).

```
String ad = "Java"; int adet = 3; double fiyat = 17.5;
System.out.printf("%-8s %3d %7.2f%n", ad, adet, fiyat);
```

```
Java       3   17.50
[Java    ][  3][  17.50]
[    Java][3  ][17.5][1.750000e+01]
[Giresun][12345][3][4]
true c 50%
```

Programın ilk `printf` satırı sunumdaki örnektir. Üç belirteç üç argümanla sırayla eşleşir. Şekil 11.26 çıktının hücre hücre nasıl oluştuğunu gösterir.



Şekil 11.26: `"%-8s %3d %7.2f%n"` biçim dizgisi. "Java" 8 hücreye sola yaslanır ve dört boşlukla tamamlanır. Ardından biçim dizgisindeki düz boşluk gelir. 3 sayısı 3 hücreye sağa yaslanır. 17.5, iki ondalıkla "17.50" olur ve 7 hücreye sağa yaslanır. `%n` satırı bitirir.

- `%-8s` : metin en az 8 karakterlik alana yazılır. Eksi işareti sola yaslar; "Java" dört karakter olduğu için sağına dört boşluk eklenir.
- `%3d` : tam sayı 3 karakterlik alana sağa yaslanır; 3'ün önüne iki boşluk gelir.
- `%7.2f` : sayı virgülden sonra iki basamakla yazılır ("17.50", beş karakter) ve 7 karakterlik alana sağa yaslanır; önüne iki boşluk gelir.
- `%n` : işletim sistemine uygun satır sonu ekler.

Programın sonraki satırları alanların sınırını köşeli parantezle görünür kılar ve başka biçimleri dener. `%8s` eksi işareti olmadığı için sağa yaslar. `%-3d` sola yaslar. `%.1f` genişlik vermeden bir ondalık yazar. `%e` bilimsel gösterimdir: $1{,}75 \times 10^{1}$. Dördüncü satır iki önemli kuralı gösterir. Birincisi, genişlik bir alt sınırdır: "Giresun" 7 karakter olduğu için `%3s` onu kesmez, tamamını yazar. İkincisi, `%.0f` yarımları yukarı yuvarlar: 2,5 → 3 ve 3,5 → 4. Son satırda `%s` 'nin `boolean` ve `char` değerlerini de yazdığı görülür. `%%` ise yüzde işaretinin kendisidir.

Biçim belirteçleri tablolar için idealdir. Aynı genişlikteki alanlar alt alta yazıldığında sütunlar kendiliğinden hizalanır:

```
public class Tablo {
    public static void main(String[] args) {
        String[] urun = {"Kalem", "Defter", "Hesap makinesi"};
        int[] adet = {12, 3, 1};
        double[] fiyat = {7.5, 42.0, 349.9};
        System.out.printf("%-15s %5s %9s%n", "Ürün", "Adet", "Tutar");
        for (int i = 0; i < urun.length; i++) {
            System.out.printf("%-15s %5d %9.2f%n", urun[i], adet[i], adet[i] * fiyat[i]);
        }
    }
}
```

```
Ürün             Adet     Tutar
Kalem              12     90.00
Defter              3    126.00
Hesap makinesi      1    349.90
```

Başlık satırı ile veri satırları aynı genişlikleri (15, 5, 9) kullandığı için sütunlar düzgün durur. Metin sütunu sola, sayı sütunları sağa yaslanmıştır; tablolarda alışılmış düzen budur. Tutar sütununda *12 × 7,5 = 90*, *3 × 42 = 126* ve *1 × 349,9 = 349,9* iki ondalıkla yazılmıştır.

Ondalık ayırıcı yerel ayara bağlıdır. Aynı program Türkçe ayarlı bir makinede şunu yazar:

```
Ürün             Adet     Tutar
Kalem              12     90,00
Defter              3    126,00
Hesap makinesi      1    349,90
```

Sayının kendisi değişmez, yalnızca yazılışı değişir. Sunumdaki örnek de Türkçe makinede “17,50” yazar. Her makinede aynı çıktı isteniyorsa `printf(Locale.US, …)` biçiminde yerel ayar ilk parametrede verilir.

Biçim dizgisini derleyici denetlemez. Belirteç ile argümanın tipi uyuşmazsa ya da argüman eksikse hata program çalışırken ortaya çıkar:

```
public class BicimHata {
    public static void main(String[] args) {
        System.out.printf("%d%n", 3.5);
    }
}
```

```
Exception in thread "main" java.util.IllegalFormatConversionException: d !=
java.lang.Double
```

`%d` bir tam sayı bekler, `3.5` ise `double`'dır. İleti, `d` dönüşümünün `Double` ile kullanılamayacağını söyler. Java sayıyı yuvarlamaz ya da kesmez, istisna fırlatır. Eksik argümanda da benzer bir istisna oluşur; ilk belirteç yazıldıktan sonra program durur:

```
public class BicimEksik {
    public static void main(String[] args) {
        System.out.printf("%s ve %s%n", "Ada");
    }
}
```

```
Ada ve
Exception in thread "main" java.util.MissingFormatArgumentException: Format specifier '%s'
```

Sık yapılan hata

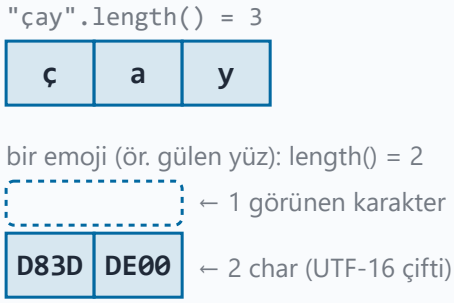
Ondalıklı bir değeri `%d` ile ya da tam sayıyı `%f` ile yazdırmak. İkisi de `IllegalFormatConversionException` fırlatır. `int` için `%d`, `double` için `%f` kullanılır; her tip için güvenli seçenek `%s`'dir ama o da biçim denetimi sağlamaz. Satır sonunu unutmak da sık görülür: `printf`, `println` gibi kendiliğinden satır atlamaz; `%n` gerekir.

11.11 Unicode ve metin uzunluğu

Java'nın `char` tipi 16 bittir ve 65.536 farklı değer alabilir. Türkçe harfler de dahil gündelik metinlerin hemen bütün karakterleri bu aralıktadır. Ancak Unicode standardı bugün 150 binden fazla karakter tanımlar: eski yazı sistemleri, matematik simgeleri, emojiler. Bunların bir kısmı tek bir `char`'a sığmaz.

Tanım 11.15

Unicode'da her karakterin bir numarası vardır; bu numaraya **kod noktası** (*code point*) denir. Java dizgileri bu numaraları UTF-16 kodlamasıyla, 16 bitlik **kod birimleri** (*code unit*) biçiminde saklar. Bir `char` bir kod birimidir. U+FFFF'ten büyük kod noktaları iki kod biriminden oluşan bir çiftle saklanır. `length()` kod birimi sayısını verir, karakter sayısını değil.



Şekil 11.27: "çay" üç kod biriminden oluşur ve uzunluğu 3'tür; ç tek bir `char`'a sığar. Gülen yüz emojisi tek bir görünür karakterdir ama D83D ve DE00 adlı iki kod biriminden oluşan bir UTF-16 çiftiyle saklanır; uzunluğu 2'dir.

```
public class UnicodeOrnek {
    public static void main(String[] args) {
        String cay = "çay";
        String emoji = "😜";
        System.out.println(cay.length() + " " + emoji.length() + " " +
emoji.codePointCount(0, emoji.length()));
        System.out.printf("%X %X %X%n", (int) emoji.charAt(0), (int) emoji.charAt(1),
emoji.codePointAt(0));
        String birlesik = "çay";
        System.out.println(birlesik.length() + " " + birlesik.equals(cay));
        System.out.println(java.text.Normalizer.normalize(birlesik,
java.text.Normalizer.Form.NFC).equals(cay));
        System.out.println("straße".toUpperCase(java.util.Locale.ROOT) + " " +
"straße".length() + " " + "straße".toUpperCase(java.util.Locale.ROOT).length());
    }
}
```

```
3 2 1
D83D DE00 1F600
4 false
true
STRASSE 6 7
```

İlk satırda "çay"ın uzunluğu 3, emojinin uzunluğu 2'dir. `codePointCount` ise kod noktalarını sayar ve emoji için 1 verir. İkinci satır emojinin iki kod birimini ve gerçek kod noktasını onaltılık tabanda yazar: U+1F600. Bu sayı 65.535'ten büyük olduğu için tek bir `char`'a sığmaz.

Üçüncü satır daha ince bir durumu gösterir. "ç" harfi Unicode'da iki yolla yazılabilir: tek bir hazır karakter (U+00E7) ya da bir "c" ve onun altına eklenen ayrı bir çengel işareti (U+0327). İkinci yazım ekranda aynı görünür ama uzunluğu 4'tür ve `equals` ile ilk yazıma eşit değildir.

`java.text.Normalizer` sınıfı iki yazımı aynı biçime getirir; dördüncü satır normalleştirmeden sonra eşitliğin sağlandığını gösterir. Son satırda Almanca "ß" harfi büyük harfe çevrilince iki harfe, "SS"ye dönüşür ve dizgi uzar: 6 karakterlik "straße", 7 karakterlik "STRASSE" olur.

Bu ayrıntıların günlük programlardaki sonucu şudur: Türkçe metinlerde `length()` ve `charAt` beklendiği gibi çalışır. Emoji, birleşik işaretler ya da başka yazı sistemleri içeren metinlerde ise "kullanıcının gördüğü karakter sayısı" `length()` 'ten farklı olabilir. Bu tür metinlerde ters çevirme gibi işlemler de bir çifti ortasından bölüp bozuk karakterler üretebilir.

11.12 Özet

- `char` tek bir karakteri tek tırnakla (`'T'`), `String` karakter dizisini çift tırnakla (`"Tarkan"`) tutar. `String` ilkel tip değil, `java.lang` paketindeki bir sınıftır; değişken nesnenin **referansını** tutar. `""` uzunluğu 0 olan gerçek bir nesnedir; `null` hiçbir nesneyi göstermez ve üzerinden metot çağrılırsa `NullPointerException` oluşur.
- `+`, işlenenlerden biri dizgiyse **birleştirme**, ikisi de sayıysa toplamadır ve soldan sağa işlenir: `"12" + 3 + 4` = "1234", `3 + 4 + "12"` = "712". `char` bir sayıdır: *'a' + 1 = 98*, *'k' − 'a' = 10*.
- `String` **değiştirilemezdir**. `toUpperCase`, `replace`, `substring`, `+` yeni bir nesne döndürür; sonuç atanmazsa kaybolur. Değişen nesne değil, değişkenin gösterdiği yerdir. Kimsenin göstermediği nesneleri çöp toplayıcı temizler.
- İndisler 0'dan `length() - 1` 'e kadardır; sınır dışı indis `StringIndexOutOfBoundsException` verir. `length()` dizgilerde metot, dizilerde `length` alandır.
- `substring(a, b)` : a dahil, b hariç; uzunluk b − a; geçerli aralık *0 ≤ a ≤ b ≤ length()*. `indexOf` ilk konumu ya da −1 verir. `split` düzenli ifade alır; ardışık ayırıcılar boş parça üretir, sondaki boş parçalar atılır.
- `replace` düz metin, `replaceAll` düzenli ifade arar. `trim` yalnızca ASCII boşlukları, `strip` bütün Unicode boşluklarını siler. `format` ve `printf` yerel ayara bağlıdır.
- `==` referansı, `equals` içeriği karşılaştırır; dizgiler her zaman `equals` ile karşılaştırılır. `compareTo` ilk farklı karakterin kod farkını, fark yoksa uzunluk farkını verir. Büyük harfler küçüklerden, İngilizce harfler Türkçe harflerden önce gelir; Türkçe sıralama için `Collator` kullanılır.
- Türkçede i ↔ İ ve ı ↔ I eşleşir. Parametresiz `toLowerCase()` / `toUpperCase()` makinenin diline bağlıdır; teknik metinde `Locale.ENGLISH`, Türkçe metinde `Locale.forLanguageTag("tr")` verilir.
- Aynı sabitler **dizgi havuzunda** tek nesneyi paylaşır; `new` her zaman yeni nesne açar. Sabit ifadeler derleme sırasında birleştirilir ve havuza girer; değişkenli birleştirme yeni nesne üretir. `intern()` havuzdaki nesneyi verir. `"sabit".equals(x)` null'a karşı güvenlidir.
- `next()` bir kelime okur ve ⏎'ı tamponda bırakır; `nextLine()` satırın kalanını okur ve ⏎'ı tüketir. `next()` ya da `nextInt()` 'ten sonra satır okunacaksa araya bir `nextLine()` konur.

- Döngüde `s = s + x` her turda yeni nesne üretir; n turda yaklaşık $n^2/2$ karakter kopyalanır. `StringBuilder` değiştirilebilir bir tampondur: `append`, `insert`, `reverse` aynı nesneyi değiştirir; sonuç `toString()` ile alınır.
- Uygulamaların ortak kalıbı karakter taramasıdır. Ters çevirme sondan başa, palindrom iki uçtan içeri doğru tarar. Kelime sayma bir bayrak kullanır. Frekans tablosu hücreyi `c - 'a'` ile seçer; Sezar şifresi `(c - 'a' + k) % 26` ile kaydırır.
- `printf`'te `%-8s` sola, `%3d` sağa yaslar; `%7.2f` iki ondalıkla 7 hücreye yazar; `%n` satır sonudur. Tip uyuşmazlığı çalışma zamanında istisna verir.
- Dizgiler UTF-16 **kod birimlerinden** oluşur; Türkçe harfler tek `char`'a sığar, emoji gibi karakterler iki `char` tutar. Bu yüzden `length()` görünen karakter sayısıyla her zaman aynı değildir.

11.13 Kavram listesi

Terimlerin İngilizce karşılıkları ve öteki sunumlardaki kullanımları terim sözlüğünde bulunur.

dizgi *string* · karakter *character* · dizgi sabiti *string literal* · ilkel tip *primitive type* · nesne *object* · referans *reference* · heap *heap* · dizgi birleştirme *string concatenation* · değiştirilemez *immutable* · değiştirilemezlik *immutability* · çöp toplayıcı *garbage collector* · indis *index* · indisleme *indexing* · sınır dışı *out of bounds* · alt dizgi *substring* · ayırıcı *delimiter* · düzenli ifade *regular expression* · boşluk *whitespace* · içerik karşılaştırması / referans karşılaştırması *content vs reference comparison* · sözlüksel sıra *lexicographic order* · büyük harf / küçük harf *uppercase / lowercase* · yerel ayar *locale* · dizgi havuzu *string pool* · tampon *buffer* · değiştirilebilir *mutable* · ters çevirme *reversing* · Sezar şifresi *Caesar cipher* · biçimleme *formatting* · biçim dizgisi *format string* · biçim belirteci *format specifier* · kod noktası *code point* · kod birimi *code unit* · istisna *exception*

11.14 Kendini Dene

1. `int p = 2, q = 5;` iken `System.out.println(p + q + "x" + p + q);` satırı ne yazar?

   Cevap

   `+` soldan sağa işlenir. İlk iki işlenen sayıdır: *2 + 5 = 7*. Sonra *7 + "x"* = "7x" olur ve ifade birleştirmeye döner: "7x2", ardından "7x25". Çıktı `7x25`'tir.

2. `String s = "dizgi"; s.toUpperCase(); s = s.replace('i', 'ı');` satırlarından sonra `s`

neyi gösterir?

Cevap

`toUpperCase()`'in sonucu atanmadığı için kaybolur; `s` hâlâ "dizgi"dir. `replace`'in sonucu ise atanır ve iki 'i' 'ı' olur: `s` "dızgı"yı gösterir. "dizgi" nesnesi değişmemiştir; yalnızca `s`'nin oku yeni nesneye taşınmıştır.

3. `String t = "Karadeniz";` için `t.length()`, `t.charAt(4)`, `t.indexOf("de")`, `t.substring(4, 8)` ve `t.lastIndexOf('a')` değerleri nedir?

Cevap

İndisler: K=0, a=1, r=2, a=3, d=4, e=5, n=6, i=7, z=8. Uzunluk 9'dur. `charAt(4)` 'd'dir. "de" 4 indisinde başlar. `substring(4, 8)` 4–7 indislerini alır: "deni" (uzunluk $8 - 4 = 4$). Son 'a' 3 indisindedir.

4. `"x;;y;z;;".split(";").length` kaçtır? Parçalar nelerdir?

Cevap

Metin her noktalı virgülden kesilir: "x", "", "y", "z", "", "". Ardışık iki noktalı virgülün arasındaki boş parça kalır, sondaki iki boş parça atılır. Sonuç `[x, , y, z]`, uzunluk 4'tür.

5. `"Masa".compareTo("masa")`, `"kitap".compareTo("kitaplık")` ve `"ödev".compareTo("sınav")` değerleri nedir? Hangileri gerçek sözlük sırasıyla çelişir?

Cevap

İlkinde ilk karakterler 'M' (77) ve 'm'dir (109): $77 - 109 = -32$. İkincisinde "kitap", "kitaplık"ın başıdır: $5 - 8 = -3$. Üçüncüsünde 'ö' (246) ile 's' (115) karşılaşır: $246 - 115 = 131$, yani "ödev" "sınav"dan sonra gelir. Türk alfabesinde ö, s'den önce geldiği için üçüncü sonuç sözlük sırasıyla çelişir.

6. Türkçe ayarlı bir bilgisayarda `"MAIL".toLowerCase().equals("mail")` ifadesinin değeri nedir? İfade nasıl düzeltilir?

Cevap

Parametresiz `toLowerCase()` Türkçe kuralları kullanır ve "I"yı "ı"ya çevirir: "maıl". "mail"e eşit olmadığı için değer `false`'tur. "MAIL" teknik bir sözcük olduğu için `toLowerCase(Locale.ENGLISH)` yazılır ya da doğrudan `"MAIL".equalsIgnoreCase("mail")` kullanılır; ikisi de `true` verir.

7. `String a = "ab"; String b = "a"; String c = b + "b"; final String d = "a"; String e = d + "b";` satırlarından sonra `a == c`, `a == e` ve `a == c.intern()` değerleri nedir?

   Cevap

   `b` `final` olmadığı için `b + "b"` çalışma zamanında yeni bir nesne üretir: `a == c` `false`'tur. `d` `final` bir sabit olduğu için `d + "b"` derleme sırasında "ab" sabitine dönüşür ve havuzdan gelir: `a == e` `true`'dur. `c.intern()` havuzdaki "ab"yi döndürür: `true`.

8. Kullanıcı önce "42 ⏎ ", sonra "Ali Can ⏎ " yazıyor. `int n = k.nextInt(); String s = k.nextLine(); String t = k.next();` satırlarından sonra `s` ve `t` nedir?

   Cevap

   `nextInt()` 42'yi okur ve ⏎'ı bırakır. `nextLine()` o ⏎'a kadar okur: `s` boş dizgidir. `next()` ikinci satırdaki ilk kelimeyi okur: `t` "Ali"dir. " Can⏎" tamponda kalır.

9. "ankara" kelimesi 4 kaydırmalı Sezar şifresiyle şifreleniyor. Sonuç nedir? Şifreyi çözmek için kaç kaydırma kullanılır?

   Cevap

   Harflerin yerleri: a=0, n=13, k=10, r=17. 4 eklenince 4, 17, 14, 21 olur; hiçbiri 26'yı aşmaz. Altı harf sırayla e, r, o, e, v, e'ye dönüşür: "eroeve". Çözmek için *26 − 4 = 22* kaydırma kullanılır.

10. `System.out.printf("[%6.1f][%-4s][%03d]%n", 2.75, "ab", 7);` satırı ne yazar?

    Cevap

    `%6.1f`, 2,75'i bir ondalığa yuvarlar: "2.8" (üç karakter) ve 6 hücreye sağa yaslar; önüne üç boşluk gelir. `%-4s`, "ab"yi 4 hücreye sola yaslar; sonuna iki boşluk gelir. `%03d`, 7'yi 3 hücreye sıfırla doldurur: "007". Çıktı `[ 2.8][ab  ][007]`'dir (İngilizce yerel ayarda; Türkçede "2,8").

11.15 İleri okuma

- Oracle, *The Java Tutorials*, "Strings" ve "The StringBuilder Class" sayfaları: dizgi oluşturma, birleştirme, metotlar ve `StringBuilder` resmî örneklerle anlatılır.
- Oracle, *Java SE 21 API Belgeleri*, `java.lang.String`, `java.util.Formatter` ve `java.util.Locale` sınıfları: her metodun kesin davranışı, `split`'in boş parça kuralları ve biçim belirteçlerinin tam listesi.
- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), Kesim 3.1 "Using Data Types": `String` bir veri tipi gözüyle ele alınır; genom dizgileri üzerinde arama örnekleri verilir.

- Unicode Consortium, *The Unicode Standard*, “Case Mappings” bölümü ve `SpecialCasing.txt` dosyası: Türkçe ve Azerice i/İ/ı/I kurallarının ve “ß → SS” gibi uzunluk değiştiren dönüşümlerin kaynağı.

11.16 Kaynaklar

1. P. Deitel, H. Deitel, *Java: How to Program, Early Objects*, 11. baskı, Pearson, 2018.
2. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13. baskı, Pearson, 2024.
3. Oracle, *The Java Tutorials: Strings; The StringBuilder Class*.
4. Oracle, *Java SE 21 API: String, StringBuilder*.
5. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 3.1 Using Data Types.
6. Unicode Consortium, *The Unicode Standard*.

KISIM III · BÖLÜM 12

Özyineleme

Bazı problemler kendi içinde aynı problemin daha küçük bir kopyasını taşır. 4!, 4 ile 3!'in çarpımıdır; 3! de 3 ile 2!'in çarpımıdır. Böyle bir problemi çözen metot, işin küçük kısmını yine kendisine bırakabilir. Bir metodun kendini çağırmasına özyineleme denir. Not, önce her özyinelemeli metotta bulunması gereken iki parçayı tanıtır: taban durum ve özyinelemeli durum. Ardından çağrıların bellekte nasıl üst üste biriktiğini, yani çağrı yığınını adım adım izler. Toplam, rakamlar toplamı, palindrom, EBOB, üs alma, ikili arama, birleştirme sıralaması, Hanoi kuleleri ve alt küme üretimi gibi örneklerden sonra maliyet konusu ele alınır. Özyinelemenin döngüye göre ne zaman pahalı olduğu, Fibonacci örneğinde aynı işin neden binlerce kez tekrarlandığı ve belleklemenin bu tekrarı nasıl kaldırdığı sayılarla gösterilir. Bütün Java programları JDK 21 ile derlenip çalıştırılmıştır. Notta verilen çıktılar, çağrı sayıları ve yığın derinlikleri bu çalıştırmalardan alınmıştır.

Bu bölümün kodu: `github.com/sercankulcu/computer-programming-java/Bolum12/src`

ÖĞRENME ÇIKTILARI

Yapı
Özyinelemeli bir metotta taban durumu ve özyinelemeli durumu göstermek; problemin her çağrıda hangi ölçüye göre küçüldüğünü ve taban duruma neden ulaşıldığını açıklamak.

İzleme
Bir özyinelemeli çağrıyı çağrı yığını üzerinde adım adım izlemek; hangi işin çağrı sırasında, hangisinin dönüş sırasında yapıldığını ayırt etmek; en derin anda yığındaki çerçeve sayısını bulmak.

Karşılaştırma
Özyinelemeli çözümü döngülü çözümle zaman, ek bellek ve okunabilirlik açısından karşılaştırmak; yığın taşmasının ve tam sayı taşmasının nedenlerini açıklamak.

İyileştirme
Özyineleme ağacında tekrarlanan alt problemleri bulmak; belleklemeyle (*memoization*) çağrı sayısını üstelden doğrusala indirmek; kuyruk özyinelemesinin Java'da neden yığını küçültmediğini açıklamak.

12.1 Özyineleme nedir?

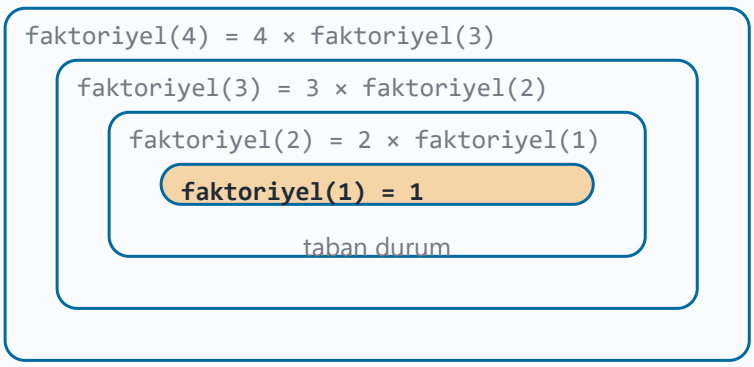
12.1.1 Kendini çağıran metot

Metotlar birbirini çağırabilir. `main` bir metodu çağırır, o metot da başka bir metodu çağırabilir. Çağrılan metot işini bitirince denetim çağırana döner ve program kaldığı yerden devam eder. Java'da bir metodun çağırabileceği metotlar arasında kendisi de vardır. Bir metot kendi gövdesinde kendi adını kullanabilir.

Bu ilk bakışta tuhaf görünür: kendini çağıran metot hiç bitmeyecekmiş gibi gelir. Oysa her çağrı problemin biraz daha küçük bir hâliyle yapılırsa durum değişir. Problem küçüldükçe bir gün cevabı doğrudan bilinen kadar küçük hâle gelir. O noktada metot kendini çağırmaz, cevabı hemen döndürür. Bekleyen çağrılar da bu cevabı sırayla kullanarak kendi işlerini tamamlar.

> **Tanım 12.1**
>
> **Özyineleme** (*recursion*), bir metodun bir problemi aynı problemin daha küçük bir örneğine indirip kendisini çağırmasıdır. Kendini çağıran metoda **özyinelemeli metot** denir.

Faktöriyel bu fikri iyi gösterir. Matematikte *4! = 4 × 3 × 2 × 1 = 24*'tür. Aynı çarpım *4! = 4 × 3!* biçiminde de yazılabilir. Yani 4!'i bulmak için önce 3!'i bulmak yeter. 3! için 2!, 2! için de 1! gerekir. 1! ise doğrudan bilinir: 1'dir. Şekil 12.1 bu iç içe yapıyı gösterir. Her kutunun içinde bir öncekinin daha küçük bir kopyası vardır. En içteki kutu kendi içinde yeni bir kutu açmaz.



Şekil 12.1: İç içe çağrılar. faktoriyel(4) kendi içinde faktoriyel(3)'ü, o da faktoriyel(2)'yi çağırır. En içteki faktoriyel(1) taban durumdur; yeni çağrı yapmadan 1 döndürür.

Her çağrı kendi parametreleri ve kendi yerel değişkenleriyle çalışır. faktoriyel(4) çağrısındaki `n` ile faktoriyel(3) çağrısındaki `n` aynı adı taşır ama ayrı değişkenlerdir. Biri 4, öteki 3 değerini tutar. Java her çağrı için bellekte ayrı bir alan açar. Bu alana **yığın çerçevesi** denir. Çerçeveler, açılış sırasıyla üst

üste dizilir. Çerçevelerin oluşturduğu bu yapı da **çağrı yığını**dır. Çağrı yığını özyinelemeyi anlamanın anahtarıdır ve notun büyük bölümünde kullanılır.

Tanım 12.2

Yığın çerçevesi (*stack frame*), bir metot çağrısı için bellekte açılan ve o çağrının parametrelerini, yerel değişkenlerini ve işi bitince dönülecek yeri tutan alandır. Çalışmakta olan bütün çağrıların çerçeveleri **çağrı yığını**nda (*call stack*) üst üste durur. Yeni çağrının çerçevesi en üste eklenir; çağrı bitince en üstteki çerçeve silinir.

12.1.2 İki zorunlu parça

Doğru çalışan her özyinelemeli metotta iki parça bulunur. Biri, cevabın doğrudan bilindiği durumdur. Öteki, problemi küçültüp metodun kendisini yeniden çağırdığı durumdur.

Tanım 12.3

Taban durum (*base case*), sonucun doğrudan bilindiği ve yeni çağrı yapılmayan durumdur. Faktöriyelde $n \le 1$ ise sonuç 1'dir. **Özyinelemeli durum** (*recursive case*), problemi daha küçük bir örneğe dönüştürüp metodun kendisini çağırdığı durumdur. Faktöriyelde bu, $n \times faktoriyel(n-1)$ hesabıdır.

Şekil 12.2 bu iki parçanın bir çağrıdaki sırasını gösterir. Çağrı başlayınca önce taban durum sorulur. Cevap "evet" ise sonuç hemen döndürülür. Cevap "hayır" ise problem küçültülür ve yeni bir çağrı yapılır. Yeni çağrı da aynı soruyla başlar.



Şekil 12.2: Bir özyinelemeli çağrının akışı. Taban durumdaysa sonuç doğrudan döner. Değilse problem küçültülür ve f(n − 1) çağrılır; bu yeni çağrı akışa baştan girer.

İki parçadan biri eksik olursa metot doğru çalışmaz. Taban durum yoksa çağrılar hiç durmaz. Her çağrı yeni bir çerçeve açar ve bir süre sonra bellek dolar. Taban durum varsa ama problem her çağrıda ona yaklaşmıyorsa sonuç yine aynıdır. Örneğin `return n * faktoriyel(n);` satırında n hiç küçülmez. Taban duruma hiç ulaşılmaz.

Bu yüzden bir özyinelemeli metot yazılırken üç soru sorulur:

1. **Taban durum nedir?** Cevabı hesaplamadan bilinen en küçük girdi hangisidir? Faktöriyelde $n \leq 1$'dir.
2. **Problem nasıl küçülür?** Hangi ölçü her çağrıda azalır? Faktöriyelde n her çağrıda 1 azalır.
3. **Küçük cevap büyük cevaba nasıl dönüşür?** Alt çağrının sonucuyla ne yapılır? Faktöriyelde alt sonuç n ile çarpılır.

İkinci sorunun cevabı aynı zamanda metodun biteceğinin gerekçesidir. n bir tam sayıdır ve her çağrıda 1 azalır. 4'ten başlayan bir n en çok üç adımda 1'e iner. Azalan bir tam sayı, alttan sınırlı bir değere sonlu sayıda adımda ulaşır. Taban durumun ulaşılabilir olduğu böyle gösterilir.

> **Sık yapılan hata**
>
> Taban durumu bazı girdileri atlayacak biçimde yazmak. `if (n == 0) return 0; return n + f(n - 2);` metodu çift n için doğru çalışır. f(5) çağrısında ise n sırasıyla 5, 3, 1, −1, −3 … olur. Tek sayıların hiçbiri 0'a eşit değildir; taban durum atlanır ve çağrılar sürer. Koşulu `n <= 0` yapmak bu hatayı giderir. Eşitlik yerine "küçük ya da eşit" denetimi, atlamalı küçülen girdilerde daha güvenlidir.

12.1.3 Özyinelemeli düşünme

Özyinelemeli bir metodu yazarken bütün çağrıları akılda canlandırmak gerekmez. Bunun yerine şöyle düşünülür: "Metot, kendisinden daha küçük her girdi için zaten doğru çalışıyor. Bu varsayımla, verilen girdinin cevabını nasıl kurarım?" Faktöriyel için cevap kısadır. faktoriyel(n − 1) doğru sonucu veriyorsa n! bu sonucun n katıdır.

Bu düşünme biçimi matematikteki tümevarıma benzer. Tümevarımda önce en küçük durum için iddia gösterilir. Sonra iddia k için doğruysa k + 1 için de doğru olduğu gösterilir. Taban durum, tümevarımın başlangıç adımına; özyinelemeli durum da tümevarım adımına karşılık gelir. İki adım birlikte doğruysa metot her geçerli girdi için doğru sonuç verir.

Yine de yeni başlayan biri için en güvenli yol, küçük bir girdiyle metodu elle izlemektir. Hangi çağrının ne zaman açıldığı, neyi beklediği ve ne döndürdüğü bir tabloya yazılır. Notun geri kalanında hemen her örnek böyle bir tabloyla izlenir. Tabloların hepsi, metotlara çağrı ve dönüş anlarını kaydeden satırlar eklenerek JDK 21'de üretilmiştir.

12.2 Faktöriyel ve çağrı yığını

12.2.1 Özyinelemeli faktöriyel

Faktöriyelin matematiksel tanımı iki satırdır: *0! = 1! = 1* ve *n! = n × (n − 1)!*. Negatif sayıların faktöriyeli tanımlı değildir. Java metodu bu tanımı neredeyse aynen izler:

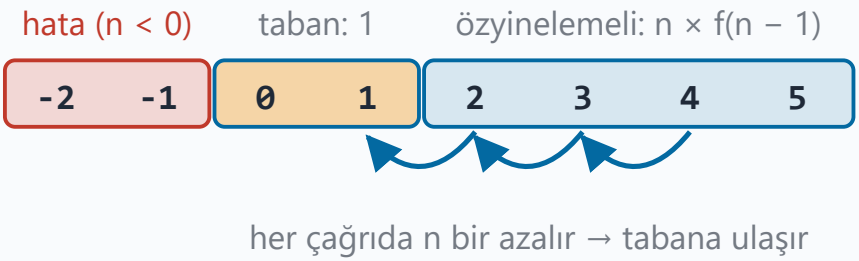
```
static long faktoriyel(int n) {
    if (n < 0)
        throw new IllegalArgumentException();
    if (n <= 1) return 1;
    return n * faktoriyel(n - 1);
}
```

Metot satır satır şöyle çalışır:

1. `static long faktoriyel(int n)` : metot bir `int` alır ve `long` döndürür. Faktöriyel çok hızlı büyüdüğü için sonuç türü `long` seçilmiştir. int en çok 12!'i tutabilir; long 20!'e kadar yeter.
2. `if (n < 0) throw new IllegalArgumentException();` : negatif girdi geçersizdir. Metot bu durumda bir hata nesnesi fırlatır ve çalışmayı keser. Fırlatma (*throw*), metodun "bu girdiyle iş yapamam" demesinin Java'daki yoludur.
3. `if (n <= 1) return 1;` : taban durum. 0 ve 1 için sonuç 1'dir; yeni çağrı yapılmaz.
4. `return n * faktoriyel(n - 1);` : özyinelemeli durum. Önce faktoriyel(n − 1) çağrılır. Bu çağrı bitip bir değer döndürünce o değer n ile çarpılır ve sonuç döndürülür.

Son satırdaki sıra önemlidir. Çarpma, alt çağrının sonucu gelmeden yapılamaz. Bu yüzden faktoriyel(4) çarpmayı yarıda bırakıp bekler. faktoriyel(3) çağrısı da aynı biçimde bekler. Çarpmaların hepsi dönüş yolunda yapılır.

Şekil 12.3 girdinin hangi bölgeye düştüğüne göre metodun ne yaptığını sayı doğrusu üzerinde gösterir. Negatif sayılar hata bölgesindedir. 0 ve 1 taban durumdur. 2 ve üstü özyinelemeli bölgededir; her çağrı bir adım sola kayar ve sonunda taban duruma girer.



Şekil 12.3: faktoriyel metodunun girdi bölgeleri. n < 0 hata, n = 0 ve n = 1 taban durum, n ≥ 2 özyinelemeli durumdur. faktoriyel(4) çağrısında n 4, 3, 2, 1 diye iner.

Metot birkaç girdiyle çalıştırıldığında beklenen sonuçlar alınır. Aşağıdaki çıktının ilk satırı faktoriyel(4), faktoriyel(0), faktoriyel(1) ve faktoriyel(20) değerleridir. İkinci satır, faktoriyel(−3) çağrısının fırlattığı hatanın yakalanıp yazdırılmasıdır:

```
24 1 1 2432902008176640000
faktoriyel(-3): java.lang.IllegalArgumentException
```

12.2.2 faktoriyel(4)'ü adım adım izlemek

Sunumdaki izleme slaydı aşağıdaki programı kullanır. `main` metodu faktoriyel(4)'ü çağırır ve sonucu yazdırır:

```
static long faktoriyel(int n) {
    if (n < 0)
        throw new IllegalArgumentException();
    if (n <= 1) return 1;
    return n * faktoriyel(n - 1);
}
System.out.println(faktoriyel(4));
```

```
24
```

Çalışma sırasında çağrı yığınında neler olduğu Tablo 12.1'de adım adım verilmiştir. “Çerçeve sayısı” sütunu, o anda yığında bulunan çerçevelerin sayısıdır; `main`'in çerçevesi de bu sayıya dahildir.

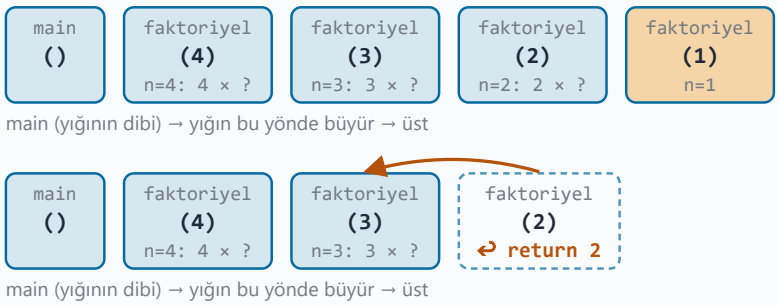
Adım	Olay	Çerçeve sayısı (main dahil)	Yığın (alttan üste)	Açıklama
1	faktoriyel(4) çağrıldı	2	main, f(4)	4 × faktoriyel(3) bekler
2	faktoriyel(3) çağrıldı	3	main, f(4), f(3)	3 × faktoriyel(2) bekler

Adım	Olay	Çerçeve sayısı (main dahil)	Yığın (alttan üste)	Açıklama
3	faktoriyel(2) çağrıldı	4	main, f(4), f(3), f(2)	2 × faktoriyel(1) bekler
4	faktoriyel(1) çağrıldı	5	main, f(4), f(3), f(2), f(1)	taban durum
5	faktoriyel(1) döndü	5	main, f(4), f(3), f(2), f(1)	return 1
6	faktoriyel(2) döndü	4	main, f(4), f(3), f(2)	2 × 1 = 2
7	faktoriyel(3) döndü	3	main, f(4), f(3)	3 × 2 = 6
8	faktoriyel(4) döndü	2	main, f(4)	4 × 6 = 24
9	main sonucu yazdırır	1	main	24

Tablo 12.1: faktoriyel(4) çağrısında yığının durumu. f(k), faktoriyel(k) çağrısının çerçevesidir.

Tablonun ilk dört satırı açılış evresidir. faktoriyel(4) çağrılır ve bir çerçeve açılır. Bu çerçevede n = 4'tür. *4 < 0* ve *4 ≤ 1* yanlış olduğundan son satıra gelinir. Çarpma için faktoriyel(3)'ün sonucu gerekir; bu çerçeve bekleyişe geçer ve yeni bir çağrı yapılır. Aynı şey n = 3 ve n = 2 için tekrarlanır. 4. adımda faktoriyel(1) çağrılır. Bu çerçevede *1 ≤ 1* doğrudur: taban duruma ulaşılmıştır.

Yığın bu anda en derin hâlindedir. main ve dört faktoriyel çerçevesiyle toplam 5 çerçeve vardır. Hiçbir çarpma henüz yapılmamıştır. Her çerçeve kendi n değerini saklayarak üstündeki çağrının sonucunu bekler. Şekil 12.4'ün üst kısmı bu anı gösterir.



Şekil 12.4: faktoriyel(4) çağrısında yığının iki anı. Üstte en derin an: dört faktoriyel çerçevesi açık, her biri "n × ?" hesabını bekliyor. Altta faktoriyel(1) ve faktoriyel(2) dönmüş, faktoriyel(2)'nin çerçevesi 2 değerini döndürmek üzere.

5. adımdan sonra dönüş evresi başlar. faktoriyel(1) 1 döndürür ve çerçevesi silinir. Denetim faktoriyel(2)'nin çerçevesine geçer. Bu çerçeve beklediği değeri almıştır: *2 × 1 = 2* hesaplanır ve döndürülür. faktoriyel(3) bu değerle *3 × 2 = 6*, faktoriyel(4) de *4 × 6 = 24* hesaplar. En sonunda main 24'ü yazdırır.

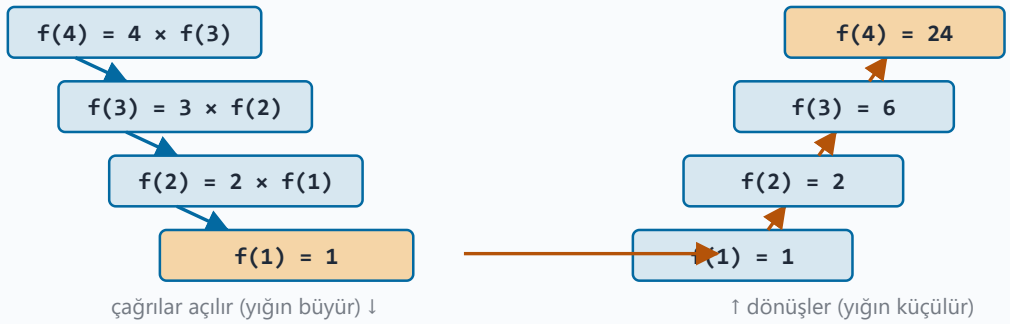
Bu izlemeden iki sonuç çıkar. Birincisi, çerçeveler açıldıkları sıranın tersine kapanır. En son açılan faktoriyel(1) ilk kapanır; ilk açılan faktoriyel(4) en son kapanır. Bu davranış "son giren ilk çıkar" (*last in, first out*, LIFO) kuralıdır. Yığın veri yapısının adı da buradan gelir. İkincisi, faktoriyel(n) çağrısında yığında aynı anda n tane faktoriyel çerçevesi bulunur. Ek bellek n ile doğru orantılı büyür; yani *O(n)*'dir. faktoriyel(20) için sayaçlı bir sürüm 20 çağrı ve 20 çerçevelik en büyük derinlik ölçmüştür.

Sık yapılan hata

Çarpmanın çağrı sırasında yapıldığını sanmak. `return n * faktoriyel(n - 1);` satırı soldan sağa okunduğu için 4 × 3 × 2 × 1 çarpımının bu sırayla yapıldığı düşünülebilir. Oysa çarpmanın ikinci çarpanı henüz yoktur. İlk tamamlanan çarpma *2 × 1*'dir, sonra *3 × 2*, en son *4 × 6* gelir. Çağrı yığını izlenirken her çerçeve için "bu çerçeve neyi bekliyor?" sorusu sorulursa sıra karışmaz.

12.2.3 Açılım ve dönüş

Özyinelemeli bir hesap iki evrede düşünülebilir. Şekil 12.5'te sol sütun açılım evresidir: çağrılar aşağı doğru açılır ve yığın büyür. Sağ sütun dönüş evresidir: değerler yukarı doğru taşınır ve yığın küçülür.



Şekil 12.5: faktoriyel(4) çağrısının açılımı ve dönüşü. Çağrılar taban duruma ulaşıncaya kadar açılır; bu sırada hiçbir çarpma yapılmaz. f(1) = 1 döndükten sonra bekleyen çarpımlar ters sırada tamamlanır: 2, 6, 24.

Açılım evresinde her çağrı yalnızca bir soru sorar ve bir sonraki çağrıyı başlatır. Asıl hesap dönüş evresinde yapılır. Bu ayrım, özyinelemeli bir metodun çıktısını tahmin ederken çok işe yarar. Bir yazdırma deyimi özyinelemeli çağrıdan önceyse açılım sırasında çalışır. Çağrıdan sonraysa dönüş sırasında, yani ters sırada çalışır. Aynı metot, yazdırma satırının yeri değiştirilerek bir sayının rakamlarını düz ya da ters sırada yazdırabilir.

Örnek 12.1

Aşağıdaki metot n > 0 iken n'yi yazar, kendini *n − 2* ile çağırır ve dönüşte n'yi yine yazar:

```
static void g(int n) {
    if (n <= 0) return;
    System.out.print(n + " ");
    g(n - 2);
    System.out.print(n + " ");
}
```

g(5) çağrısında açılım sırasında 5, 3 ve 1 yazılır. g(−1) taban durumdur ve hiçbir şey yazmadan döner. Dönüş evresinde çerçeveler ters sırada kapanır ve 1, 3, 5 yazılır. JDK 21 çıktısı `5 3 1 1 3 5` 'tir. Çıktı bir aynaya benzer: ilk yarı açılımı, ikinci yarı dönüşü gösterir.

12.3 Tek çağrılı örnekler

Faktöriyelde her çağrı en çok bir yeni çağrı yapar. Böyle metotların çağrıları tek bir zincir oluşturur: f(4) → f(3) → f(2) → f(1). Bu biçime **doğrusal özyineleme** denir. Bu kesimdeki beş örnek de doğrusaldır. Her birinde aynı üç soru sorulur: taban durum nedir, problem nasıl küçülür, alt sonuç nasıl kullanılır?

12.3.1 1'den n'ye toplam

1'den n'ye kadar sayıların toplamı da faktöriyel gibi tanımlanabilir. toplam(n), n ile toplam(n − 1)'in toplamıdır. Hiç sayı yoksa toplam 0'dır: *toplam(0) = 0*. Böylece *toplam(3) = 3 + toplam(2) = 3 + 2 + toplam(1) = 3 + 2 + 1 + toplam(0) = 6* olur.

```
static int toplam(int n) {
    if (n <= 0) return 0;
    return n + toplam(n - 1);
}
System.out.println(toplam(3));
```

```
6
```

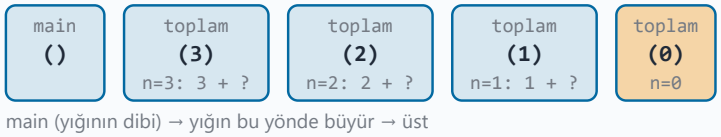
Taban durum `n <= 0` biçiminde yazılmıştır. Bu seçim negatif girdilerde de metodu güvenli kılar. toplam(−5) çağrısı ilk denetimde taban duruma girer ve 0 döndürür. Koşul `n == 0` olsaydı toplam(−5) çağrısında n −6, −7 … diye iner ve hiç durmazdı.

Adım	Olay	Çerçeve sayısı (main dahil)	Açıklama
1	toplam(3) çağrıldı	2	3 <= 0 → false; 3 + toplam(2) bekler
2	toplam(2) çağrıldı	3	2 <= 0 → false; 2 + toplam(1) bekler
3	toplam(1) çağrıldı	4	1 <= 0 → false; 1 + toplam(0) bekler
4	toplam(0) çağrıldı	5	0 <= 0 → true
5	toplam(0) döndü	5	return 0
6	toplam(1) döndü	4	1 + 0 = 1
7	toplam(2) döndü	3	2 + 1 = 3
8	toplam(3) döndü	2	3 + 3 = 6

Adım	Olay	Çerçeve sayısı (main dahil)	Açıklama
9	main 6 yazdırır	1	

Tablo 12.2: toplam(3) çağrısının izlemesi. Her toplama, alt çağrı döndükten sonra yapılır.

Tablo 12.2 faktöriyel izlemesine çok benzer. Açılımda dört toplam çerçevesi açılır: toplam(3), toplam(2), toplam(1) ve toplam(0). En derin anda main ile birlikte 5 çerçeve vardır. toplam(0) 0 döndürür. Dönüşte sırasıyla *1 + 0 = 1*, *2 + 1 = 3* ve *3 + 3 = 6* hesaplanır. Sonuç 6 = 3 + 2 + 1 + 0'dır.



Şekil 12.6: toplam(3) çağrısında en derin an. toplam(0) çerçevesi taban durumdadır; altındaki üç çerçeve "3 + ?", "2 + ?" ve "1 + ?" hesaplarını bekler.

toplam(n) metodu n + 1 çağrı yapar ve yığın n + 1 çerçeve derinleşir. Zaman da ek bellek de *O(n)*'dir. Aynı toplam *n(n + 1)/2* formülüyle tek adımda bulunabilir. Bu örnek özyinelemenin mekanizmasını göstermek için seçilmiştir; toplamı hesaplamanın en iyi yolu değildir.

12.3.2 Rakamlar toplamı ve yardımcı metot

Bir sayının rakamları toplamı da özyinelemeyle yazılabilir. Fikir rakam kalıbına dayanır: `n % 10` son rakamı verir, `n / 10` son rakamı atar. 472'nin rakamları toplamı, son rakam 2 ile 47'nin rakamları toplamının toplamıdır. Tek basamaklı bir sayının rakamları toplamı kendisidir. Bu cümleler metodun taban durumunu ve özyinelemeli durumunu birlikte verir.

```
static int rakamToplami(int n) {
    long m = Math.abs((long) n);
    return rakamToplamiPozitif(m);
}
static int rakamToplamiPozitif(long n) {
    if (n < 10) return (int) n;
    return (int) (n % 10)
        + rakamToplamiPozitif(n / 10);
}
System.out.println(rakamToplami(472));
```

Program 13 yazar. Kodda iki metot vardır. İşi asıl yapan `rakamToplamiPozitif`'tir ve yalnızca negatif olmayan sayılarla doğru çalışır. `rakamToplami` ise dışarıdan çağrılan metottur. Girdiyi hazırlar ve asıl işi yardımcı metoda bırakır. Böyle metotlara **yardımcı metot** (*helper method*) denir.

Hazırlık adımı neden gereklidir? Negatif bir sayıda `n % 10` negatif çıkar. Örneğin Java'da *−472 % 10 = −2*'dir. Ayrıca *−472 < 10* doğru olduğu için metot hemen taban duruma girer ve −472 döndürür. Bunu önlemek için önce mutlak değer alınır. Mutlak değer neden long'a çevrildikten sonra alınır? int'in en küçük değeri −2.147.483.648'dir; int'in en büyük değeri ise 2.147.483.647'dir. `Math.abs(Integer.MIN_VALUE)` bu yüzden int'e sığmaz ve yine −2.147.483.648 döner. `Math.abs((long) n)` ise 2.147.483.648 değerini doğru verir. Yardımcı metot bu nedenle long parametre alır.

Program birkaç uç girdiyle de çalıştırılmıştır. Çıktının ikinci satırı rakamToplami(−472), rakamToplami(0) ve rakamToplami(Integer.MIN_VALUE) sonuçlarıdır:

```
13
13 0 47
```

Son değer 47'dir: *2 + 1 + 4 + 7 + 4 + 8 + 3 + 6 + 4 + 8 = 47*. Mutlak değer int ile alınsaydı metot negatif bir sayıyla taban duruma girer ve yanlış sonuç döndürürdü.

Çerçeve	Koşul	n % 10	n / 10	Sonuç
rakamToplamiPozitif(472)	472 < 10 → false	2	47	bekler
rakamToplamiPozitif(47)	47 < 10 → false	7	4	bekler
rakamToplamiPozitif(4)	4 < 10 → true	—	—	return 4
rakamToplamiPozitif(47)	dönüş	7	4	7 + 4 = 11
rakamToplamiPozitif(472)	dönüş	2	47	2 + 11 = 13
rakamToplami(472)	—	—	—	return 13

Tablo 12.3: rakamToplami(472) izlemesi. İlk üç satır açılım, sonraki üç satır dönüştür.

Tablo 12.3 her çerçevenin hangi rakamı beklettiğini gösterir. 472'nin çerçevesi 2'yi, 47'nin çerçevesi 7'yi bekletir. 4 tek basamaklıdır ve taban durumdur. Dönüşte *7 + 4 = 11*, ardından *2 + 11 = 13* hesaplanır. Yardımcı metot 13 döndürür ve rakamToplami bu değeri aynen iletir.



Şekil 12.7: rakamToplami(472) çağrısında en derin an. main'in üstünde rakamToplami ve üç rakamToplamiPozitif çerçevesi vardır. Alttaki iki yardımcı çerçeve 2 ve 7 rakamlarını bekletir.

Yığın derinliği sayının basamak sayısı kadardır. Basamak sayısı, sayının büyüklüğüyle logaritmik büyür: 10 basamaklı bir int en çok 10 yardımcı çerçeve açar. Bu yüzden rakam problemlerinde yığın taşması kaygısı yoktur.

12.3.3 Palindrom kontrolü

Palindrom, tersten okunuşu düz okunuşuyla aynı olan dizgidir: “kayak”, “ana”, “abba” gibi. Bir dizginin palindrom olup olmadığı iki uçtan içe doğru ilerlenerek anlaşılır. İlk ve son karakter farklıysa dizgi palindrom değildir. Aynıysa bu iki karakter atılır ve kalan iç dizgi aynı soruyla denetlenir. Ortada tek karakter ya da hiç karakter kalmadıysa dizgi palindromdur.

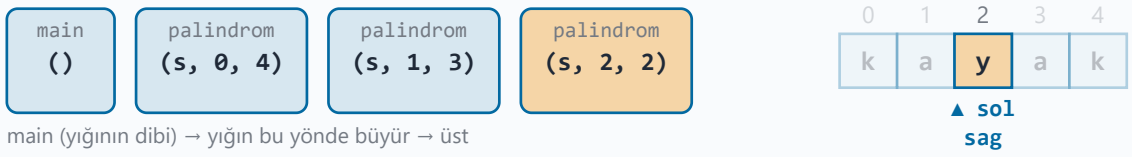
```
static boolean palindrom(String s,
                         int sol, int sag) {
    if (sol >= sag) return true;
    if (s.charAt(sol) != s.charAt(sag))
        return false;
    return palindrom(s, sol + 1, sag - 1);
}
System.out.println(palindrom("kayak", 0, 4));
```

Metot dizginin kopyasını oluşturmaz. Bunun yerine bakılacak bölgeyi iki indisle tanımlar: `sol` ve `sag`. Her çağrıda sol 1 artar, sag 1 azalır. Problemin ölçüsü bölgenin uzunluğudur: *sag − sol + 1*. Bu uzunluk her çağrıda 2 azalır ve sonunda 1 ya da 0 olur. Taban durum `sol >= sag` bu iki durumu birlikte kapsar.

Metodun iki ayrı çıkışı vardır. Uçlar farklıysa false döner; bu da bir taban durum sayılır, çünkü yeni çağrı yapılmaz. Uçlar eşitse son satırdaki çağrının sonucu aynen döndürülür. Dönüş yolunda yapılan bir hesap yoktur.

Çerçeve	sol >= sag	Uçlar	Sonraki adım
palindrom(s, 0, 4)	0 >= 4 → false	'k' ile 'k' eşit	palindrom(s, 1, 3)
palindrom(s, 1, 3)	1 >= 3 → false	'a' ile 'a' eşit	palindrom(s, 2, 2)
palindrom(s, 2, 2)	2 >= 2 → true	—	return true

Tablo 12.4: palindrom("kayak", 0, 4) izlemesi.



Şekil 12.8: palindrom("kayak", 0, 4) çağrısında en derin an. Üç palindrom çerçevesi açıktır; en üstteki çerçevede sol ve sag ikisi de ortadaki 'y' karakterini gösterir.

Tablo 12.4'e göre “kayak” için üç çağrı yapılır. 0. ve 4. karakterler 'k', 1. ve 3. karakterler 'a'dır. Üçüncü çağrıda *2 ≥ 2* doğrudur: ortada yalnız 'y' kalmıştır ve true döner. Bu değer iki çerçeveden geçerek main'e ulaşır. “kitap” için tek bir çağrı yeter. İlk karakter 'k', son karakter 'p'dir; metot hemen false

döndürür. Program ek girdilerle de çalıştırılmıştır. Aşağıdaki ikinci satır sırasıyla "kitap", "abba" ve "a" sonuçlarıdır:

```
true
false true true
```

Çift uzunluklu "abba" dizgisinde iç bölge sonunda boşalır: palindrom(s, 2, 1) çağrısında *2 ≥ 1* doğrudur. Taban durumun eşitlik yerine `>=` ile yazılması bu durumu da yakalar. n karakterlik bir dizgi için en çok *⌊n/2⌋ + 1* çağrı yapılır; zaman ve yığın derinliği *O(n)*'dir.

12.3.4 Öklid algoritmasıyla EBOB

İki sayının en büyük ortak böleni (EBOB), ikisini de kalansız bölen en büyük sayıdır. EBOB(48, 18) = 6'dır. Öklid algoritması bu sayıyı bölme kalanlarıyla bulur. Dayandığı kural şudur: b sıfırdan farklıysa EBOB(a, b) = EBOB(b, a mod b)'dir. b = 0 ise EBOB(a, 0) = |a|'dır. Bu iki satır özyinelemeli bir metodu doğrudan tarif eder.

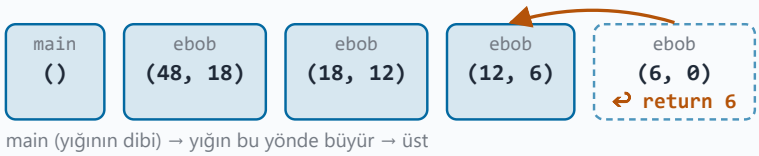
```
static int ebob(int a, int b) {
    if (b == 0) return Math.abs(a);
    return ebob(b, a % b);
}
System.out.println(ebob(48, 18));
```

Kural neden doğrudur? a ile b'yi bölen her sayı *a − k·b* farkını da böler. a mod b de böyle bir farktır. Tersine, b ile a mod b'yi bölen her sayı a'yı da böler. Yani (a, b) ve (b, a mod b) çiftlerinin ortak bölenleri aynıdır. Ortak bölenleri aynı olan iki çiftin en büyük ortak böleni de aynıdır.

Çerçeve	a	b	a % b	Sonraki adım
ebob(48, 18)	48	18	12	ebob(18, 12)
ebob(18, 12)	18	12	6	ebob(12, 6)
ebob(12, 6)	12	6	0	ebob(6, 0)
ebob(6, 0)	6	0	—	return 6

Tablo 12.5: ebob(48, 18) izlemesi. Her satırda a % b hesaplanır ve sonraki çağrı (b, a % b) ile yapılır.

Tablo 12.5 dört çağrıyı gösterir. *48 % 18 = 12*, *18 % 12 = 6* ve *12 % 6 = 0*'dır. ebob(6, 0) taban durumdur ve 6 döndürür. Bu değer üç çerçeveden geçerek aynen geri gelir. Son sıfırdan farklı kalan, yani 6, aranan EBOB'dur.



Şekil 12.9: ebob(48, 18) çağrısında taban duruma ulaşıldığı an. ebob(6, 0) 6 döndürmek üzeredir; alttaki çerçeveler bu değeri değiştirmeden iletir.

Metot farklı girdilerle de denenmiştir. Aşağıdaki ikinci satır ebob(18, 48), ebob(17, 5), ebob(−12, 8) ve ebob(7, 0) sonuçlarıdır:

```
6
6 1 4 7
```

ebob(18, 48) çağrısında ilk adım sayıların yerini değiştirir: *18 % 48 = 18* olduğundan sonraki çağrı ebob(48, 18)'dir. Bu yüzden sayaçlı sürüm bu girdide 5 çağrı sayar, ebob(48, 18)'de ise 4. Taban durumdaki `Math.abs` negatif girdilerde sonucu pozitif yapar: ebob(−12, 8) = 4'tür.

Öklid algoritması çok hızlıdır. Kalanlar her iki adımda en az yarıya iner. Bu yüzden çağrı sayısı sayıların basamak sayısıyla orantılıdır. En kötü durum ardışık iki Fibonacci sayısıyla ortaya çıkar. int'e sığan büyük bir ardışık Fibonacci çifti olan 1.134.903.170 ve 701.408.733 için bile sayaçlı sürüm yalnızca 44 çağrı saymıştır.

12.3.5 Dizgiyi ters çevirme

Bir dizginin tersi de özyinelemeyle tanımlanabilir. “abc” dizgisinin tersi, “bc”nin tersinin sonuna 'a' eklenerek bulunur. Uzunluğu 0 ya da 1 olan dizginin tersi kendisidir.

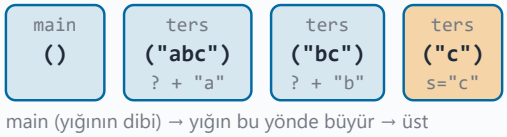
```
static String ters(String s) {
  if (s.length() <= 1) return s;
  return ters(s.substring(1)) + s.charAt(0);
}
System.out.println(ters("abc"));
```

`s.substring(1)` ilk karakter hariç kalan dizgiyi verir. `s.charAt(0)` ilk karakterdir. Alt çağrı kalanın tersini döndürür; ilk karakter bu sonucun sonuna eklenir. Toplama bir String ile bir char arasında yapıldığı için sonuç yine String'dir.

Çerçeve	İş	Dönüş değeri
ters("c")	uzunluk 1 ≤ 1 → taban durum	return "c"
ters("bc")	ters("c") + 'b'	"c" + 'b' = "cb"

Çerçeve	İş	Dönüş değeri
ters("abc")	ters("bc") + 'a'	"cb" + 'a' = "cba"

Tablo 12.6: ters("abc") çağrısında dönüş evresi. Satırlar, çerçevelerin kapanma sırasındadır.



Şekil 12.10: ters("abc") çağrısında en derin an. Alttaki iki çerçeve "? + "a"" ve "? + "b"" birleştirmelerini bekler; en üstteki ters("c") taban durumdadır.

Program boş dizgi, "kayak" ve "Java" ile de çalıştırılmıştır:

```
cba
[] kayak avaJ
```

Boş dizginin tersi boş dizgidir; köşeli ayraçlar bunu görünür kılmak için yazdırılmıştır. "kayak" palindrom olduğu için tersi kendisiyle aynıdır.

Bu metot kısa ve okunaklıdır ama verimli değildir. Her çağrı `substring` ile yeni bir alt dizgi üretir. n karakterlik bir dizgi için üretilen alt dizgilerin toplam uzunluğu yaklaşık $n(n + 1)/2$'dir. Sayaçlı sürüm 1000 karakterlik bir dizgi için 1000 çağrı ve toplam 500.500 karakterlik alt dizgi saymıştır. Zaman $O(n^2)$'dir. Döngüyle ve `StringBuilder` ile ters çevirme $O(n)$'de yapılır. Bir çözümün kısa olması, ucuz olduğu anlamına gelmez.

12.4 Üs alma

12.4.1 Düz özyinelemeli üs

Bir tabanın n'inci kuvveti, tabanın n kez kendisiyle çarpımıdır. Bu tanım da özyinelemeli yazılabilir: $taban^n = taban \times taban^{n-1}$ ve $taban^0 = 1$. Örneğin $2^4 = 2 \times 2^3$'tür.

```
static long us(long taban, int n) {
  if (n == 0) return 1;
  return taban * us(taban, n - 1);
}
System.out.println(us(2, 4));
```

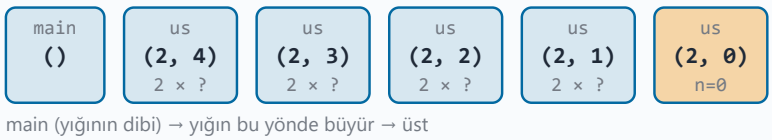
```
16
```

Yapı faktöriyelle aynıdır. Taban durum `n == 0`'dır. Özyinelemeli durumda n 1 azalır ve alt sonuç taban ile çarpılır. Taban her çağrıda aynı kalır; yalnızca üs küçülür.

Çerçeve	Koşul	Dönüş değeri
us(2, 0)	0 == 0 → true	return 1
us(2, 1)	1 == 0 → false	2 × 1 = 2
us(2, 2)	2 == 0 → false	2 × 2 = 4
us(2, 3)	3 == 0 → false	2 × 4 = 8
us(2, 4)	4 == 0 → false	2 × 8 = 16

Tablo 12.7: us(2, 4) çağrısının dönüş evresi. Satırlar çerçevelerin kapanma sırasındadır.

us(2, 4) çağrısında us(2, 3), us(2, 2), us(2, 1) ve us(2, 0) çağrıları açılır. us(2, 0) taban durumdur ve 1 döndürür. Tablo 12.7 dönüş yolundaki çarpmaları verir: 2, 4, 8 ve 16. Toplam *n + 1 = 5* çağrı yapılır. us(2, 10) için bu sayı 11'dir. Zaman ve yığın derinliği *O(n)*'dir.



Şekil 12.11: us(2, 4) çağrısında en derin an. Dört çerçeve "2 × ?" hesabını bekler; en üstteki us(2, 0) taban durumdadır.

Metot negatif üsle çağrılırsa ne olur? us(2, −1) çağrısında n −1'dir ve *−1 == 0* yanlıştır. Sonraki çağrıda n −2, sonra −3 olur. n sıfırdan uzaklaşır ve taban duruma hiç ulaşılmaz. Çağrılar yığın dolana kadar sürer. Bu durum 12.11.2'de ölçülerek ele alınır.

12.4.2 Hızlı üs

Düz yöntem 2^{10} için 10 çarpma yapar. Oysa $2^{10} = (2^5)^2$'dir. 2^5 bir kez hesaplanıp kendisiyle çarpılırsa iş yarıya iner. Aynı fikir 2^5 için de kullanılabilir. Üs tek olduğundan bir taban fazladan çarpılır: $2^5 = (2^2)^2 \times 2$. Her adımda üs yarıya indiği için çağrı sayısı çok azalır. Bu yönteme **hızlı üs** ya da karesini alarak üs alma denir.

```
static long hizliUs(long taban, int us) {
  if (us == 0) return 1;
  long yarim = hizliUs(taban, us / 2);
  if (us % 2 == 0) return yarim * yarim;
  return yarim * yarim * taban;
}
System.out.println(hizliUs(2, 10));
```

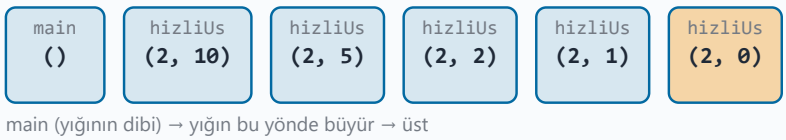
Metot önce `us / 2` ile kendini çağırır ve sonucu `yarim` değişkenine koyar. Java'da iki int'in bölümü tam sayıdır; *5 / 2 = 2*'dir. Üs çiftse sonuç *yarim × yarim*'dır. Üs tekse bölmede 1 kaybolmuştur ve sonuç bir kez daha tabanla çarpılır.

`yarim` değişkeni bu metodun en önemli ayrıntısıdır. Kod `return hizliUs(taban, us / 2) * hizliUs(taban, us / 2);` biçiminde yazılsaydı aynı alt problem iki kez hesaplanırdı. Her çağrı iki çağrı yapar ve kazanç tamamen kaybolurdu. Alt sonucu bir kez hesaplayıp yerel değişkende saklamak, yöntemi hızlı yapan şeydir.

Çerçeve	us / 2	yarim	us	Dönüş değeri
hizliUs(2, 0)	—	—	—	return 1
hizliUs(2, 1)	0	1	tek	1 × 1 × 2 = 2
hizliUs(2, 2)	1	2	çift	2 × 2 = 4
hizliUs(2, 5)	2	4	tek	4 × 4 × 2 = 32
hizliUs(2, 10)	5	32	çift	32 × 32 = 1024

Tablo 12.8: hizliUs(2, 10) çağrısının dönüş evresi. “us” sütunu üssün tek ya da çift olduğunu gösterir.

Açılımda üs 10, 5, 2, 1 ve 0 olur. hizliUs(2, 0) 1 döndürür. Tablo 12.8 dönüş yolunu gösterir. Üs 1 tektir: *1 × 1 × 2 = 2*. Üs 2 çifttir: *2 × 2 = 4*. Üs 5 tektir: *4 × 4 × 2 = 32*. Üs 10 çifttir: *32 × 32 = 1024*. Toplam 5 çağrı yapılmıştır; düz yöntem 11 çağrı yapardı.



Şekil 12.12: hizliUs(2, 10) çağrısında en derin an. Üs her çerçevede yarıya inmiştir: 10, 5, 2, 1, 0.

Çağrı sayısı üssün ikili gösterimindeki basamak sayısına bağlıdır. 1 ve üstü bir üs için çağrı sayısı *⌊log₂ us⌋ + 2*'dir. Sayaçlı sürümün ölçümleri bu formülle uyuşur:

Üs	Düz us: çağrı	hizliUs: çağrı	⌊log₂ us⌋ + 2
10	11	5	3 + 2 = 5

Üs	Düz us: çağrı	hizliUs: çağrı	$\lfloor \log_2 us \rfloor + 2$
16	17	6	4 + 2 = 6
62	63	7	5 + 2 = 7
1000	1001	11	9 + 2 = 11
1.000.000	1.000.001	21	19 + 2 = 21

Tablo 12.9: Düz ve hızlı üs yöntemlerinin çağrı sayıları (sayaçlı sürümlerle ölçülmüştür).

Milyonuncu kuvvet için düz yöntem bir milyon çağrı yapar; varsayılan yığın boyutunda bu, yığın taşmasına yol açar. Hızlı üs yalnızca 21 çağrı yapar. Zaman ve yığın derinliği *O(log n)*'dir. Program hizliUs(3, 16) = 43.046.721 ve hizliUs(2, 62) = 4.611.686.018.427.387.904 değerlerini de doğru hesaplamıştır:

```
1024
43046721 4611686018427387904
```

Sonuçların long'a sığması gerekir. 2^{63} long'un sınırını aşar; hizliUs(2, 63) çağrısı hata vermeden –9.223.372.036.854.775.808 döndürür. Taşma, metodun hızından bağımsız bir sorundur.

12.5 Böl ve fethet

Hızlı üs bir problemi yarı büyüklükte bir probleme indirir. Bu fikrin genel adı **böl ve fethet**tir (*divide and conquer*). Problem iki ya da daha çok küçük parçaya bölünür. Parçalar aynı yöntemle, yani özyinelemeyle çözülür. Sonra parça çözümleri birleştirilerek asıl çözüm elde edilir. Parçalar yeterince küçülünce doğrudan çözülür; bu da taban durumdur.

Tanım 12.4

Böl ve fethet, bir problemi aynı türden daha küçük alt problemlere bölen, alt problemleri özyinelemeyle çözen ve çözümlerini birleştiren algoritma tasarım yöntemidir.

12.5.1 Özyinelemeli ikili arama

İkili arama sıralı bir dizide bir değeri arar. Aralığın ortasındaki öğeye bakılır. Aranan değer ortadakine eşitse arama biter. Ortadaki değer daha küçükse aranan değer sağ yarıdadır; daha büyükse sol yarıdadır. Bakılmayan yarı bir daha hiç incelenmez. Döngüyle yazılan ikili arama, özyinelemeyle de aynı adımlarla yazılabilir. Her çağrı aralığın yarısıyla kendini çağırır.

```
static int ara(int[] a, int x,
               int lo, int hi) {
  if (lo > hi) return -1;
  int mid = (lo + hi) / 2;
  if (a[mid] == x) return mid;
  if (a[mid] < x) return ara(a, x, mid + 1, hi);
  return ara(a, x, lo, mid - 1);
}
// a = {2, 5, 8, 12, 16, 23, 38, 56}
System.out.println(ara(a, 38, 0, 7));
```

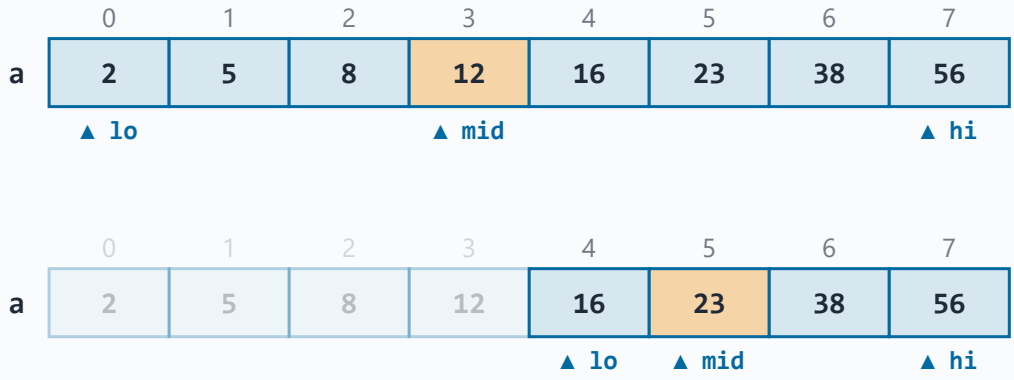
Aralık iki indisle tanımlanır: `lo` alt sınır, `hi` üst sınırdır. İki taban durum vardır. `lo > hi` ise aralık boştur ve aranan değer dizide yoktur; metot −1 döndürür. `a[mid] == x` ise değer bulunmuştur ve indisi döner. Öteki iki satır aralığı yarıya indirir. Dizinin kendisi kopyalanmaz; bütün çağrılar aynı diziye bakar.

Programda dizi bir yorum satırıyla verilmiştir. Çalıştırılan sürümde dizi `int[] a = {2, 5, 8, 12, 16, 23, 38, 56};` satırıyla tanımlanmıştır. Çıktının ikinci satırı 2, 56 ve 20 değerlerinin aranmasıdır:

```
6
0 7 -1
```

Çerçeve	lo	hi	mid	a[mid]	Karar
ara(a, 38, 0, 7)	0	7	3	12	12 < 38: sağ yarı ara(a, 38, 4, 7)
ara(a, 38, 4, 7)	4	7	5	23	23 < 38: sağ yarı ara(a, 38, 6, 7)
ara(a, 38, 6, 7)	6	7	6	38	38 == 38: return 6

Tablo 12.10: ara(a, 38, 0, 7) izlemesi.





Şekil 12.13: 38'in aranması. Üstte ilk çağrı: lo = 0, hi = 7, mid = 3. Ortada ikinci çağrı: lo = 4, hi = 7, mid = 5. Altta üçüncü çağrı: lo = 6, hi = 7, mid = 6 ve a[6] = 38.

Tablo 12.10'a göre 38 üç çağrıda bulunur. İlk çağrıda *mid = (0 + 7) / 2 = 3* ve a[3] = 12'dir. 12 < 38 olduğundan arama sağ yarıda sürer. İkinci çağrıda mid = 5 ve a[5] = 23'tür; arama yine sağa kayar. Üçüncü çağrıda mid = 6 ve a[6] = 38'dir: değer bulunmuştur. 6 iki çerçeveden geçerek main'e ulaşır.

Dizide olmayan bir değer arandığında aralık sonunda boşalır. Tablo 12.11 20'nin aranmasını gösterir. Dördüncü çağrıda lo = 5, hi = 4 olur; *5 > 4* doğrudur ve −1 döner.

Çerçeve	lo	hi	mid	a[mid]	Karar
ara(a, 20, 0, 7)	0	7	3	12	12 < 20: sağ yarı ara(a, 20, 4, 7)
ara(a, 20, 4, 7)	4	7	5	23	23 > 20: sol yarı ara(a, 20, 4, 4)
ara(a, 20, 4, 4)	4	4	4	16	16 < 20: sağ yarı ara(a, 20, 5, 4)
ara(a, 20, 5, 4)	5	4	—	—	5 > 4: return −1

Tablo 12.11: ara(a, 20, 0, 7) izlemesi. 20 dizide yoktur.

Her çağrıda aralık yarıya indiği için n öğeli bir dizide en çok yaklaşık $log_2\ n + 1$ çağrı yapılır. 8 öğeli dizide en kötü durum 4 çağrıdır: 56 ve 20 aramaları 4 çağrı sürmüştür. Zaman *O(log n)*'dir. Özyinelemeli sürümün ek yığın belleği de *O(log n)*'dir; döngülü sürüm aynı işi *O(1)* ek bellekle yapar.

Sık yapılan hata

Özyinelemeli çağrının önüne `return` yazmayı unutmak. `if (a[mid] < x) ara(a, x, mid + 1, hi);` satırında alt çağrı değeri bulur ama bu değer kullanılmaz. Metot alt satırlara devam eder ve yanlış bir sonuç döndürür. Değer döndüren bir özyinelemeli metotta alt çağrının sonucu ya döndürülmeli ya da bir hesapta kullanılmalıdır.

12.5.2 Birleştirme sıralaması

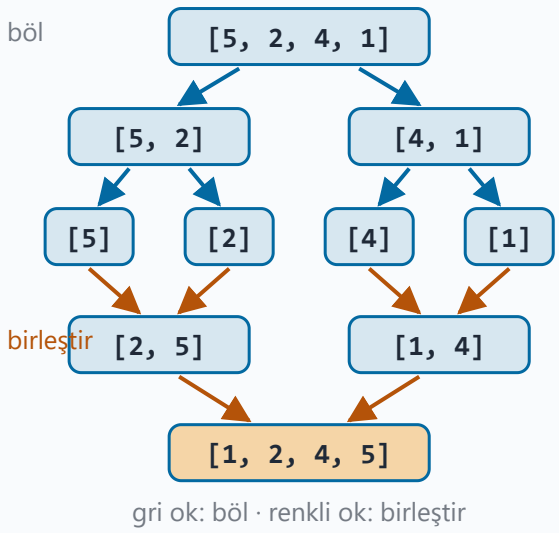
Birleştirme sıralaması (*merge sort*) böl ve fethet yönteminin tipik bir örneğidir. Dizi ortadan ikiye bölünür. Her yarı özyinelemeyle sıralanır. Sonra sıralı iki yarı tek bir sıralı dizide birleştirilir. Tek öğeli bir dizi zaten sıralıdır; bu da taban durumdur.

```
static void sirala(int[] a, int lo, int hi) {
    if (lo >= hi) return;
    int mid = (lo + hi) / 2;
    sirala(a, lo, mid);
    sirala(a, mid + 1, hi);
    int[] t = new int[hi - lo + 1];
    int i = lo, j = mid + 1, k = 0;
    while (i <= mid && j <= hi)
        t[k++] = a[i] <= a[j] ? a[i++] : a[j++];
    while (i <= mid) t[k++] = a[i++];
    while (j <= hi) t[k++] = a[j++];
    System.arraycopy(t, 0, a, lo, t.length);
}
```

Metot üç bölümden oluşur:

1. **Taban durum:** `lo >= hi` ise aralıkta en çok bir öğe vardır ve metot hemen döner.
2. **Bölme:** `mid` hesaplanır; `sirala(a, lo, mid)` sol yarıyı, `sirala(a, mid + 1, hi)` sağ yarıyı sıralar.
3. **Birleştirme:** `t` adlı geçici bir dizi açılır. `i` sol yarıda, `j` sağ yarıda ilerler. Her adımda iki yarının baştaki öğelerinden küçüğü t'ye yazılır. Yarılardan biri bitince ötekinin kalanı olduğu gibi eklenir. En sonda `System.arraycopy` t'yi dizinin ilgili bölgesine geri kopyalar.

`a[i] <= a[j]` karşılaştırmasındaki "küçük ya da eşit" bilinçli bir seçimdir. Eşit öğelerde soldaki önce yazılır. Böylece eşit değerlerin özgün sırası korunur. Bu özelliğe kararlılık denir. Birleştirme sıralaması kararlı bir sıralama yöntemidir.



Şekil 12.14: {5, 2, 4, 1} dizisinin birleştirme sıralaması. Gri oklar bölmeyi, renkli oklar birleştirmeyi gösterir. [5, 2] → [2, 5] ve [4, 1] → [1, 4] olur; sonunda [1, 2, 4, 5] elde edilir.

Çağrı no	Çerçeve	Yapılan iş	Dizi
1	sirala(a, 0, 3)	mid = 1: [0..1] ve [2..3] sıralanacak	[5, 2, 4, 1]
2	sirala(a, 0, 1)	mid = 0: [0..0] ve [1..1] sıralanacak	[5, 2, 4, 1]
3	sirala(a, 0, 0)	tek öğe: hemen döner	[5, 2, 4, 1]
4	sirala(a, 1, 1)	tek öğe: hemen döner	[5, 2, 4, 1]
2	sirala(a, 0, 1)	birleştirme: [2, 5]	[2, 5, 4, 1]
5	sirala(a, 2, 3)	mid = 2: [2..2] ve [3..3] sıralanacak	[2, 5, 4, 1]
6	sirala(a, 2, 2)	tek öğe: hemen döner	[2, 5, 4, 1]
7	sirala(a, 3, 3)	tek öğe: hemen döner	[2, 5, 4, 1]
5	sirala(a, 2, 3)	birleştirme: [1, 4]	[2, 5, 1, 4]
1	sirala(a, 0, 3)	birleştirme: [1, 2, 4, 5]	[1, 2, 4, 5]

Tablo 12.12: sirala(a, 0, 3) çağrılarının sırası, a = {5, 2, 4, 1}. Aynı çağrı numarası, bir çerçevenin önce bölüp sonra birleştirdiği iki anı gösterir.

Tablo 12.12 çağrıların gerçek sırasını verir. 1 numaralı çağrı diziyi [0..1] ve [2..3] diye böler. Sol yarı tamamen bitmeden sağ yarıya geçilmez. 2 numaralı çağrı [5, 2]'yi iki tek öğeye böler, ikisi de hemen döner, sonra birleştirme [2, 5] üretir. Dizi bu anda [2, 5, 4, 1] olur. Sağ yarı aynı biçimde [1, 4]'e dönüşür. En sonda 1 numaralı çağrı [2, 5] ile [1, 4]'ü birleştirir. Karşılaştırmalar 2 ile 1, 2 ile 4, 5 ile 4 arasındadır; sağ yarı bitince 5 eklenir. Sonuç [1, 2, 4, 5]'tir.

Program ayrıca 7 öğeli bir diziyi de sıralamıştır:

```
[1, 2, 4, 5]
[3, 9, 10, 27, 38, 43, 82]
```

Maliyet şöyle hesaplanır. Bölme sırasında dizinin boyu her düzeyde yarıya iner. n öğeli bir dizi yaklaşık $log_2\ n$ düzeyde tek öğelere ayrılır. Her düzeyde bütün parçaların birleştirilmesi toplam n öğe yazmayı gerektirir. Toplam iş düzey sayısı ile düzey başına işin çarpımıdır: $O(n \log n)$. Birleştirme için geçici dizi kullanıldığı için ek bellek $O(n)$'dir. Yığın derinliği ise yalnızca $O(\log n)$'dir. Sayaçlı sürüm ters sıralı 1024 öğelik dizi için 2047 çağrı, en çok 11 çerçeve derinlik ve 5120 karşılaştırma saymıştır. $5120 = 512 \times 10$, yani $(n/2) \cdot log_2\ n$'dir.

Karşılaştırma için: seçme ya da kabarcık sıralaması gibi iç içe iki döngüyle çalışan yöntemler yaklaşık $n^2/2$ karşılaştırma yapar. 1024 öğe için bu yaklaşık 520 bin karşılaştırma demektir. Birleştirme sıralaması aynı diziyi yüz kat daha az karşılaştırmayla sıralar.

12.6 İki çağrılı özyineleme

Şimdiye kadarki örneklerin çoğunda her çağrı en çok bir yeni çağrı yaptı. Birleştirme sıralaması ise iki çağrı yapar: biri sol yarı, biri sağ yarı için. Her çağrının iki çağrı yaptığı metotlarda çağrılar bir zincir değil, bir ağaç oluşturur. Bu ağaca **özyineleme ağacı** (*recursion tree*) denir. Ağacın kökü ilk çağrıdır; her düğümün çocukları o çağrının yaptığı alt çağrılardır. Hanoi kuleleri ve alt küme üretimi bu yapının iki klasik örneğidir.

12.6.1 Hanoi kuleleri

Hanoi kuleleri bir bulmacadır. Üç çubuk vardır: A, B ve C. A çubuğunda büyükten küçüğe dizilmiş n disk bulunur. Amaç bütün diskleri C çubuğuna taşımaktır. İki kural vardır: bir hamlede yalnızca bir disk, o da bir çubuğun en üstündeki disk taşınabilir. Büyük bir disk küçük bir diskin üstüne konamaz.

Bulmaca ilk bakışta zordur ama özyinelemeli çözümü üç satırdır. n diski A'dan C'ye taşımak için önce üstteki n − 1 disk B'ye taşınır. Bu sırada C yardımcı çubuk olarak kullanılır. Sonra A'da yalnız kalan en büyük disk C'ye taşınır. En son B'deki n − 1 disk, A yardımcı olarak kullanılarak C'ye taşınır. n − 1 diski taşımak aynı problemin küçük bir örneğidir. Hiç disk yoksa yapılacak bir şey yoktur.

```
static void hanoi(int n, char kaynak,
                  char hedef, char ara) {
  if (n == 0) return;
  hanoi(n - 1, kaynak, ara, hedef);
  System.out.print(kaynak + "->" + hedef + " ");
  hanoi(n - 1, ara, hedef, kaynak);
}
hanoi(3, 'A', 'C', 'B');
```

```
A->C A->B C->B A->C B->A B->C A->C
```

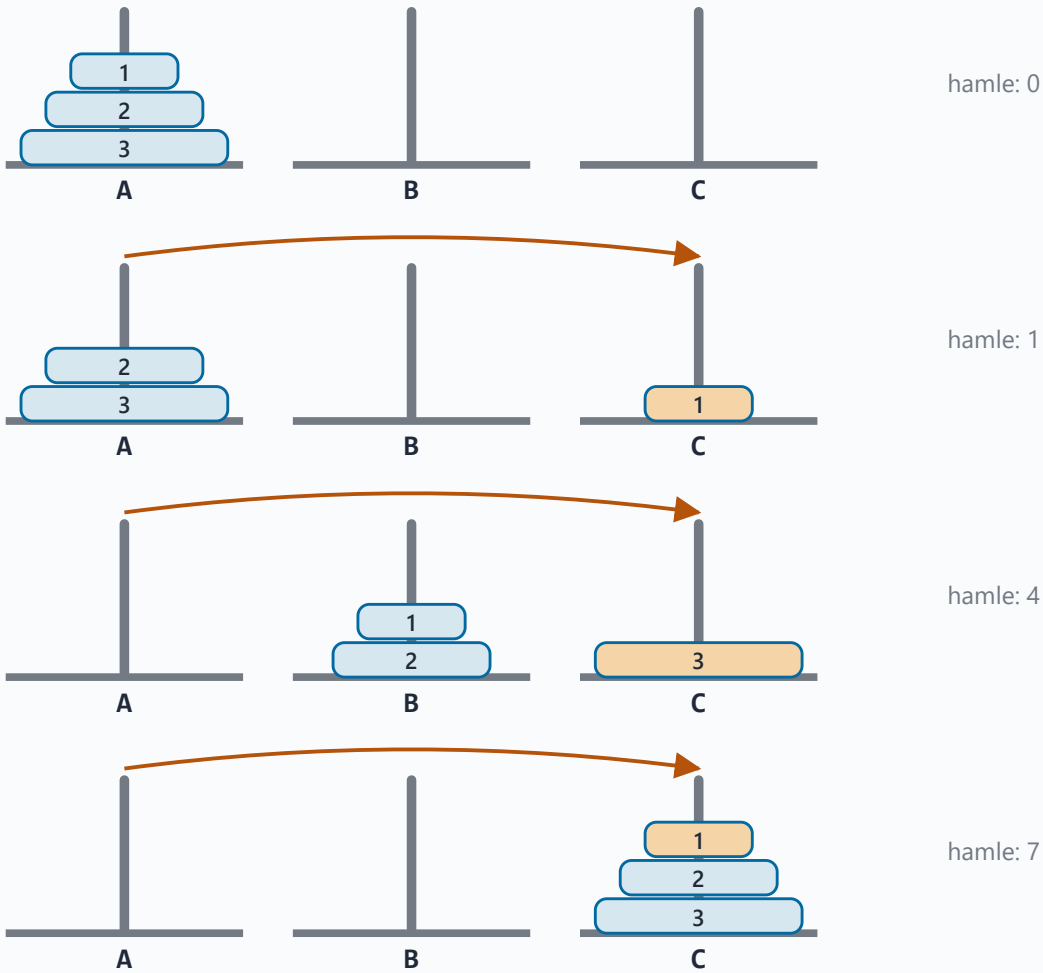
Metodun dört parametresi vardır: disk sayısı, kaynak çubuk, hedef çubuk ve yardımcı (ara) çubuk. Birinci özyinelemeli çağrıda hedef ile ara yer değiştirir: n − 1 disk ara çubuğa gider. İkinci çağrıda kaynak ile ara yer değiştirir: n − 1 disk ara çubuktan hedefe gider. Yazdırma deyimi iki çağrının arasındadır. Bu yüzden bir hamle, sol alt ağaç bütünüyle bittikten sonra yazılır.

Hamle	Disk	Taşıma	Hamleyi yapan çerçeve
1	1	A → C	hanoi(1, A, C, B)
2	2	A → B	hanoi(2, A, B, C)
3	1	C → B	hanoi(1, C, B, A)
4	3	A → C	hanoi(3, A, C, B)
5	1	B → A	hanoi(1, B, A, C)

Hamle	Disk	Taşıma	Hamleyi yapan çerçeve
6	2	B → C	hanoi(2, B, C, A)
7	1	A → C	hanoi(1, A, C, B)

Tablo 12.13: hanoi(3, 'A', 'C', 'B') hamleleri. Son sütun, hamleyi yazdıran çerçevedir.

Tablo 12.13 yedi hamleyi ve her hamleyi hangi çağrının yaptığını gösterir. Disk 1 en küçük, disk 3 en büyük disktir. İlk üç hamle üstteki iki diski B'ye taşır. 4. hamle en büyük diski C'ye götürür; bu hamleyi en dıştaki hanoi(3, A, C, B) çağrısı yapar. Son üç hamle B'deki iki diski C'ye taşır. Disk 1 her iki hamlede bir hareket eder: 1., 3., 5. ve 7. hamleler onundur. Şekil 12.15 dört anı gösterir.



Şekil 12.15: Üç diskli Hanoi kulesinin dört anı. Yukarıdan aşağıya: başlangıç, 1. hamleden sonra (disk 1 C'de), 4. hamleden sonra (disk 3 C'de, disk 1 ve 2 B'de), 7. hamleden sonra (bütün diskler C'de).

Hamle sayısı da özyinelemeyle bulunur. h(n), n diski taşımak için gereken hamle sayısı olsun. Çözüm iki kez n − 1 disk taşır ve bir kez de en büyük diski taşır: $h(n) = 2 \cdot h(n-1) + 1$, $h(0) = 0$. Buradan h(1) = 1, h(2) = 3, h(3) = 7, h(4) = 15 ve h(5) = 31 bulunur. Genel formül $h(n) = 2^n - 1$'dir. Formül tümevarım-

la doğrulanır: $2 \cdot (2^{n-1} - 1) + 1 = 2^n - 1$.

Disk sayısı n	Hamle	$2^n - 1$	Çağrı	En büyük derinlik
1	1	1	3	2
2	3	3	7	3
3	7	7	15	4
4	15	15	31	5
5	31	31	63	6
10	1023	1023	2047	11
20	1.048.575	1.048.575	2.097.151	21

Tablo 12.14: Hanoi çözümünün hamle ve çağrı sayıları (sayaçlı sürümle ölçülmüştür; hanoi(0) çağrıları dahil).

Tablo 12.14 iki farklı büyüme gösterir. Hamle ve çağrı sayısı üsteldir: her disk eklendiğinde yaklaşık iki katına çıkar. Yığın derinliği ise doğrusaldır: n + 1. Hanoi'nin bu çözümü en iyisidir. n diski $2^n - 1$'den az hamleyle taşımak mümkün değildir. Çünkü en büyük disk bir kez hareket edecekse üstündeki n − 1 disk iki kez bütünüyle taşınmak zorundadır. Üstel büyüme algoritmanın değil, problemin kendisinin özelliğidir. 64 disk için gereken hamle sayısı $2^{64} - 1 \approx 1{,}8 \times 10^{19}$'dur.

12.6.2 Alt küme üretimi ve geri izleme

{1, 2} kümesinin dört alt kümesi vardır: {1, 2}, {1}, {2} ve boş küme. n öğeli bir kümenin 2^n alt kümesi vardır. Bunun nedeni basittir: her öğe için iki seçenek vardır, ya alt kümeye dahil edilir ya da edilmez. n öğe için seçenekler çarpılır: $2 \times 2 \times \ldots \times 2 = 2^n$.

Bu "her öğe için iki seçenek" fikri özyinelemeli bir metoda dönüşür. Metot i'nci öğe için karar verir. Önce öğeyi listeye ekler ve kalan öğeler için kendini çağırır. Sonra öğeyi listeden çıkarır ve kalan öğeler için kendini yeniden çağırır. Bütün öğeler için karar verildiyse liste bir alt kümedir ve yazdırılır.

```
static void altKume(int[] a, int i,
                    List<Integer> sec) {
    if (i == a.length) {
        System.out.println(sec);
        return;
    }
    sec.add(a[i]);
    altKume(a, i + 1, sec);
    sec.remove(sec.size() - 1);
    altKume(a, i + 1, sec);
}
altKume(new int[]{1, 2}, 0, new ArrayList<>());
```

`sec`, o ana kadar seçilen öğeleri tutan bir listedir (`List<Integer>`). Bütün çerçeveler aynı liste nesnesini paylaşır; her çağrıya listenin kopyası değil, kendisi gider. `sec.add(a[i])` öğeyi sona ekler. `sec.remove(sec.size() - 1)` son öğeyi çıkarır ve listeyi eklemeden önceki hâline döndürür. Bu çıkarma adımı olmasaydı ikinci çağrı, birinci çağrının eklediği öğeyi listede bulurdu.

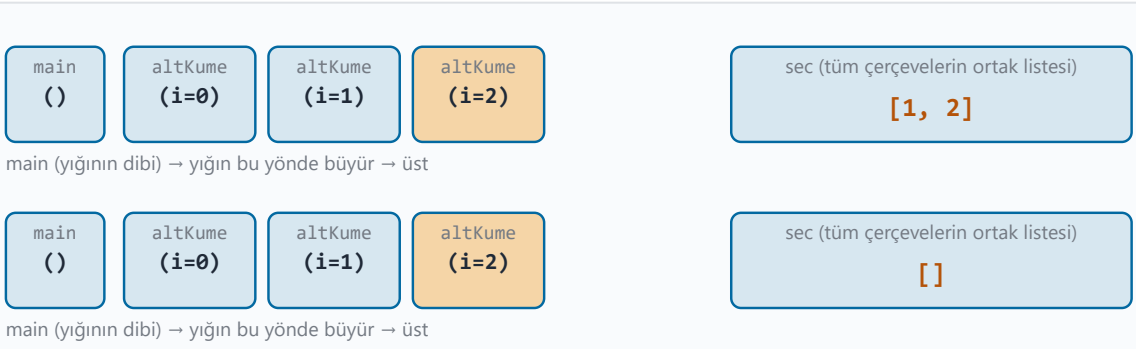
Tanım 12.5

Geri izleme (*backtracking*), seçimleri adım adım yapan, bir seçimin sonuçlarını özyinelemeyle inceledikten sonra o seçimi geri alıp sıradaki seçeneği deneyen arama yöntemidir.

Adım	Çerçeve	İş	sec	Çıktı
1	altKume(i=0)	1 dahil: sec.add(1)	[1]	
2	altKume(i=1)	2 dahil: sec.add(2)	[1, 2]	
3	altKume(i=2)	taban durum: yazdır	[1, 2]	[1, 2]
4	altKume(i=1)	2 hariç: sec.remove(...)	[1]	[1, 2]
5	altKume(i=2)	taban durum: yazdır	[1]	[1, 2] [1]
6	altKume(i=0)	1 hariç: sec.remove(...)	[]	[1, 2] [1]
7	altKume(i=1)	2 dahil: sec.add(2)	[2]	[1, 2] [1]
8	altKume(i=2)	taban durum: yazdır	[2]	[1, 2] [1] [2]
9	altKume(i=1)	2 hariç: sec.remove(...)	[]	[1, 2] [1] [2]
10	altKume(i=2)	taban durum: yazdır	[]	[1, 2] [1] [2] []

Tablo 12.15: altKume(new int[]{1, 2}, 0, ...) izlemesi. “sec” sütunu ortak listenin o adımdan sonraki hâlidir.

Tablo 12.15 on adımda dört alt kümenin nasıl üretildiğini gösterir. 1. ve 2. adımlarda iki öğe de eklenir ve liste [1, 2] olur. 3. adımda i = 2'dir, yani dizinin sonuna gelinmiştir: [1, 2] yazdırılır. 4. adımda geri dönülür ve 2 çıkarılır. Liste [1] olur ve “2 hariç” dalı [1]'i yazdırır. 6. adımda i = 0 çerçevesine dönülür ve 1 de çıkarılır. Bundan sonra “1 hariç” dalları [2] ve [] alt kümelerini üretir.



Şekil 12.16: Alt küme üretiminin iki yaprağı. Üstte ilk yaprak: üç altKume çerçevesi açık ve ortak liste [1, 2]. Altta son yaprak: liste boştur ve boş küme yazdırılır.

Programın çıktısında önce iki öğeli kümenin, "--" satırından sonra da {1, 2, 3} kümesinin alt kümeleri vardır:

```
[1, 2]
[1]
[2]
[]
--
[1, 2, 3]
[1, 2]
[1, 3]
[1]
[2, 3]
[2]
[3]
[]
```

Çıktı sırası özyineleme ağacının soldan sağa gezilme sırasıdır. Her düzeyde önce "dahil" dalı, sonra "hariç" dalı çalışır. Bu yüzden ilk satır bütün öğeleri içeren küme, son satır boş kümedir.

Özyineleme ağacının her düzeyi bir öğeye karşılık gelir. n öğeli bir dizide ağacın 2^n yaprağı ve toplam $2^{n+1} - 1$ düğümü vardır. Sayaçlı sürüm n = 2 için 7, n = 3 için 15, n = 10 için 2047 çağrı saymıştır. Yığın derinliği ise yalnızca n + 1'dir. Zaman $O(2^n)$'dir; ama bu, 2^n alt kümenin hepsini üretmek isteyen her yöntemin ödemek zorunda olduğu bedeldir.

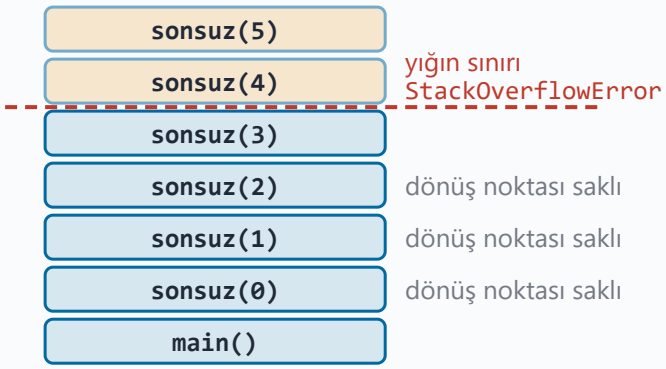
12.7 Çağrı yığını ve yığın taşması

12.7.1 Yığın çerçevesinde neler saklanır?

Önceki örneklerin hepsinde çağrı yığını kullanıldı. Bu kesim, yığında tam olarak neyin saklandığını ve yığının neden sınırlı olduğunu açıklar. Bir metot başka bir metodu çağırdığında çağıran metodun işi yarıda kalır. İş daha sonra kaldığı yerden sürecektir. Bunun için üç bilgi saklanmalıdır:

1. **Parametreler:** çağrıya verilen değerler. faktoriyel(3) çerçevesinde n = 3'tür.
2. **Yerel değişkenler:** metodun içinde tanımlanan değişkenler. hizliUs'taki `yarim` ya da sirala'daki `mid`, `i`, `j` böyle değişkenlerdir.
3. **Dönüş noktası:** çağrılan metot bitince çağıran metotta hangi komuttan devam edileceği. faktoriyel(3) bittiğinde faktoriyel(4)'ün çerçevesinde çarpma işlemine dönülür.

Java sanal makinesi (*Java Virtual Machine*, JVM) bu bilgileri her çağrı için ayrı bir yığın çerçevesinde tutar. Çerçeveler, iş parçacığına ayrılmış yığın belleğinde üst üste durur. Yeni bir çağrı yapıldığında en üste bir çerçeve eklenir. Metot `return` ile bitince en üstteki çerçeve silinir ve denetim bir alttaki çerçeveye geçer. Çerçeveler bu yüzden ters sırada kapanır: son açılan ilk kapanır.



Şekil 12.17: Taban durumu olmayan özyinelemede çerçeveler yığın sınırını aşar. Her çerçeve dönüş noktasını saklar. Sınırın üstüne çıkacak çerçeveye yer kalmadığında StackOverflowError oluşur.

Yığın belleği sınırlıdır. Java'da her iş parçacığının yığını, program başlarken belirlenen sabit bir boyuttadır. 64 bitlik sistemlerde bu varsayılan boyut genellikle 1 MB'tır. Çerçeveler sınırı aşacak kadar çoğalırsa JVM yeni çerçeveye yer bulamaz ve bir hata fırlatır. Bu duruma **yığın taşması** denir.

Tanım 12.6

Yığın taşması (*stack overflow*), çağrı yığınındaki çerçevelerin yığın için ayrılan belleği doldurmasıdır. Java'da yığın taşması `java.lang.StackOverflowError` hatasıyla bildirilir. Taban duruma hiç ulaşmayan özyinelemeye **sonsuz özyineleme** (*infinite recursion*) denir; sonsuz özyineleme her zaman yığın taşmasıyla biter.

Yığın taşmasının iki nedeni vardır. Birincisi sonsuz özyinelemedir: taban durum yoktur, yanlış yazılmıştır ya da problem küçülmez. İkincisi, doğru yazılmış ama çok derine inen özyinelemedir. Taban duruma ulaşılacaktır ama o ana kadar biriken çerçeveler yığına sığmaz. Doğrusal bir özyineleme, girdinin boyu kadar çerçeve açar. Bu yüzden büyük girdilerde doğru bir metot bile taşabilir.

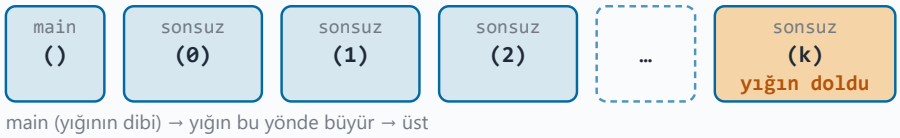
12.7.2 StackOverflowError

Sunumdaki en kısa örnek, hiçbir koşul içermeyen bir metottur:

```
static void sonsuz(int n) {
    sonsuz(n + 1);
}
sonsuz(0);
```

sonsuz(0) çağrısı hemen sonsuz(1)'i çağırır, o da sonsuz(2)'yi. Hiçbir çağrı dönmediği için hiçbir çerçeve silinmez. Şekil 12.18 bu birikmeyi gösterir. Yığın dolunca JVM bir StackOverflowError fırlatır. Hata yakalanmadığı için program biter ve hata metni standart hata akışına (stderr) yazılır. JDK 21'de

çalıştırılan programın hata metninin ilk satırları şöyledir:



Şekil 12.18: sonsuz(0) çağrısında yığın doluyor. Çerçeveler sonsuz(0), sonsuz(1), sonsuz(2) … diye birikir; k'inci çağrıda yığın dolar.

```
Exception in thread "main" java.lang.StackOverflowError
    at S19.sonsuz(S19.java:5)
    at S19.sonsuz(S19.java:5)
    at S19.sonsuz(S19.java:5)
    ...
```

Her “at” satırı yığındaki bir çerçeveyi gösterir: hangi sınıfın hangi metodunda ve kaynak dosyanın hangi satırında olduğu yazar. Satır numarası 5'tir, çünkü programda sonsuz metodu sınıfın 5. satırındaki çağrıyı yapar. Bu satırlar binlerce kez tekrarlanır. JVM hata metnine varsayılan olarak en çok 1024 çerçeve yazar; çalıştırılan programın hata metni de 1 başlık satırı ve 1024 “at” satırından oluşmuştur. Böyle bir hata metninde aynı satırın art arda tekrarlanması, sonsuz ya da çok derin özyinelemenin açık işaretidir.

Taşmanın kaç çağrıdan sonra olduğu sabit değildir. Çerçevenin boyu metodun parametre ve yerel değişken sayısına bağlıdır. JVM'in kodu yorumlayarak mı yoksa makine koduna derleyerek mi çalıştırdığı da çerçeve boyunu değiştirir. Aşağıdaki program taşma anına kadar yapılan çağrıları sayar. Hata `catch` ile yakalandığı için program bitmez ve sayıyı yazdırabilir:

```
// Varsayılan yığın boyutunda kaç çağrıda StackOverflowError oluştuğunu sayar (JVM'e ve
koşullara göre değişir).
public class Derinlik {
    static int sayac;

    static void sonsuz(int n) {
        sayac++;
        sonsuz(n + 1);
    }

    static long us(long taban, int n) {
        sayac++;
        if (n == 0) return 1;
        return taban * us(taban, n - 1);
    }

    static int toplam(int n) {
        if (n <= 0) return 0;
        return n + toplam(n - 1);
    }

    public static void main(String[] args) {
        for (int k = 0; k < 3; k++) {
            sayac = 0;
            try {
                sonsuz(0);
            } catch (StackOverflowError e) {
                System.out.println("sonsuz: " + sayac + " cagri sonra " +
e.getClass().getName());
            }
        }
        sayac = 0;
        try {
            us(2, -1);
        } catch (StackOverflowError e) {
            System.out.println("us(2, -1): " + sayac + " cagri sonra " +
e.getClass().getSimpleName());
        }
        for (int n : new int[]{1000, 5000, 10000, 100000, 1000000}) {
            try {
                System.out.println("toplam(" + n + ") = " + toplam(n));
            } catch (StackOverflowError e) {
                System.out.println("toplam(" + n + "): StackOverflowError");
            }
        }
    }
}
```

Programın bir çalıştırmasındaki çıktı şöyledir:

```
sonsuz: 23539 cagri sonra java.lang.StackOverflowError
sonsuz: 62252 cagri sonra java.lang.StackOverflowError
sonsuz: 62252 cagri sonra java.lang.StackOverflowError
us(2, -1): 8079 cagri sonra StackOverflowError
toplam(1000) = 500500
toplam(5000) = 12502500
toplam(10000) = 50005000
toplam(100000): StackOverflowError
toplam(1000000): StackOverflowError
```

Aynı metot aynı programda üç kez denendiğinde ilk denemede yaklaşık 23 bin, sonrakilerde yaklaşık 62 bin çağrıdan sonra taşmıştır. Program birkaç kez daha çalıştırıldığında ilk denemenin sayısı 22 bin ile 27 bin arasında değişmiştir. Sayılar her çalıştırmada biraz farklı çıkabilir. Ama büyüklük sırası bellidir: varsayılan yığın boyutunda birkaç on bin çağrı. toplam(10.000) çalışır, toplam(100.000) taşar. Yığın boyutu `java -Xss4m Program` gibi bir seçenekle büyütülebilir. Bu sınırı yalnızca uzaklaştırır; derin özyineleme sorununu çözmez.

Sık yapılan hata

StackOverflowError'ı `catch` ile yakalayıp programı sürdürmeye çalışmak. Bu hata, bir programlama yanlışının ya da yanlış seçilmiş bir algoritmanın belirtisidir. Yukarıdaki program hatayı yalnızca ölçüm yapmak için yakalar. Gerçek bir programda çözüm taban durumu düzeltmek, girdiyi denetlemek ya da derin özyinelemeyi döngüye çevirmektir.

12.8 Özyineleme mi döngü mü?

12.8.1 Seçim ölçütleri

Özyinelemeyle yazılan her metot bir döngüyle de yazılabilir. Döngüyle yazılan her metot da özyinelemeyle yazılabilir. İkisi arasındaki seçim, hangisinin daha “güçlü” olduğuna değil, problemin yapısına ve maliyete bağlıdır.

Özyineleme, kendine benzeyen problemlerde doğal bir seçimdir. Ağaç biçimindeki veriler, dizin ve klasör yapıları, böl ve fethet algoritmaları ve geri izleme bu türdendir. Bu problemlerde özyinelemeli kod matematiksel tanıma çok yakındır ve kolay okunur. Hanoi ve alt küme üretimini döngüyle yazmak mümkündür. Ancak bunun için çağrı yığınının işini yapan bir yığın veri yapısını elle yönetmek gerekir.

Basit doğrusal tekrarlarda ise döngü üstündür. Faktöriyel, 1'den n'ye toplam ve rakamlar toplamı bir döngüyle aynı açıklıkta yazılır. Döngü tek bir çerçevede çalışır; her tur için yeni çerçeve açmaz. Ek yığın belleği *O(1)*'dir ve girdi ne kadar büyük olursa olsun yığın taşmaz. Çağrı yapmanın da küçük bir zaman maliyeti vardır: çerçeve açmak ve kapatmak döngünün bir turundan pahalıdır.

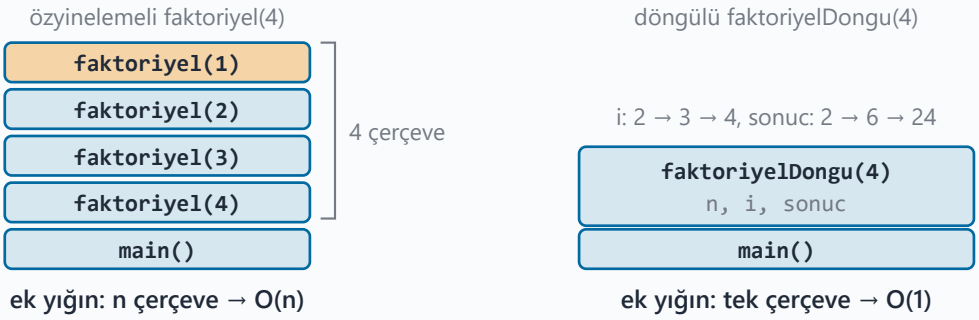


Şekil 12.19: faktoriyel(4) için yığın kullanımı. Özyinelemeli sürüm main ile birlikte 5 çerçeve kullanır. Döngülü sürüm 2 çerçeve kullanır; i ve sonuc aynı çerçevede her turda güncellenir.

Seçim yalnızca kodun kısalığına bakılarak yapılmamalıdır. Üç ölçüt birlikte düşünülür: doğruluk, okunabilirlik ve kaynak kullanımı. Taban durumu ve küçülmesi açık bir özyinelemeli çözüm hata yapma olasılığını azaltabilir. Ama girdinin boyu on binleri aşabiliyorsa doğrusal özyineleme yığını tehlikeye sokar. Aynı alt problemi defalarca çözen bir özyineleme ise zamanı tehlikeye sokar; bu durum Fibonacci örneğinde ayrıntılı görülür.

Özellik	Özyinelemeli	Döngülü
Zaman karmaşıklığı	O(n)	O(n)
Ek yığın belleği	O(n): n çerçeve	O(1): tek çerçeve
Akış	Matematiksel tanıma yakın	Açık sayaç ve biriktirici
Derinlik riski	Var: büyük n'de yığın taşabilir	Yok

Tablo 12.16: Özyinelemeli ve döngülü faktöriyelin karşılaştırılması.



Şekil 12.20: n = 4 için yığın. Solda özyinelemeli sürümün 4 faktoriyel çerçevesi (ek yığın O(n)), sağda döngülü sürümün tek çerçevesi (ek yığın O(1)); döngüde i 2'den 4'e, sonuc 2, 6, 24 diye değişir.

12.8.2 Döngülü faktöriyel

Faktöriyelin döngülü sürümü bir biriktiriciyle çalışır. `sonuc` 1'den başlar ve her turda sayaç i ile çarpılır:

```
static long faktoriyelDongu(int n) {
    long sonuc = 1;
    for (int i = 2; i <= n; i++)
        sonuc *= i;
    return sonuc;
}
System.out.println(faktoriyelDongu(4));
```

```
24
```

Tur	i	i <= n	sonuc (gövdeden sonra)
1	2	`true`	2
2	3	`true`	6
3	4	`true`	24
—	5	`false`	24 (döngü biter)

Tablo 12.17: faktoriyelDongu(4) izlemesi.
Bütün turlar aynı çerçevede çalışır.

Tablo 12.17 üç turu gösterir. i = 2'de sonuc 2, i = 3'te 6, i = 4'te 24 olur. i = 5'te *5 ≤ 4* yanlıştır ve döngü biter. Döngünün i = 2'den başlaması bilinçlidir: 1 ile çarpmak sonucu değiştirmez. n = 0 ve n = 1 için döngü hiç dönmez ve 1 döndürülür; bu da 0! = 1! = 1 tanımıyla uyuşur.

Özyinelemeli sürümle karşılaştırıldığında hesap aynıdır: *2 × 3 × 4 = 24*. Fark, çarpmaların sırası ve yığın kullanımındadır. Döngülü sürümde çarpmalar küçükten büyüğe, gidiş yolunda yapılır. Özyinelemeli sürümde büyükten küçüğe çağrılar açılır ve çarpmalar dönüş yolunda yapılır. Döngülü sürümde yığın hiçbir anda 2 çerçeveyi aşmaz. Çalışma sorularındaki `while` sürümü de aynı işi aynı maliyetle yapar; n = 0'dan 5'e kadar sonuçları 1, 1, 2, 6, 24 ve 120'dir.

12.9 Fibonacci ve bellekleme

12.9.1 Doğrudan Fibonacci

Fibonacci dizisi 0 ve 1 ile başlar. Sonraki her terim, kendinden önceki iki terimin toplamıdır: 0, 1, 1, 2, 3, 5, 8, 13, 21 … Tanım özyinelemelidir: *fib(0) = 0*, *fib(1) = 1* ve *fib(n) = fib(n − 1) + fib(n − 2)*. Java'ya doğrudan çevrilen metot şöyledir:

```
static long fib(int n) {
    if (n <= 1) return n;
    return fib(n - 1) + fib(n - 2);
}
System.out.println(fib(4));
```

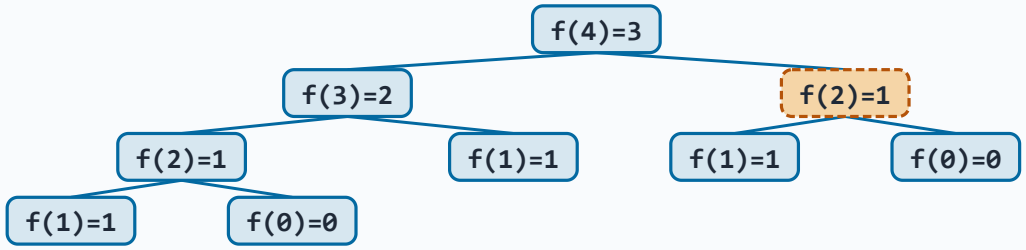
```
3
```

Metot tanımın kelimesi kelimesine karşılığıdır ve doğru sonuç verir. Ama faktöriyelden önemli bir farkı vardır: özyinelemeli durumda iki çağrı yapılır. Önce fib(n − 1) bütünüyle hesaplanır, sonra fib(n − 2). Çağrılar bir zincir değil, bir ağaç oluşturur.

Çağrı no	Çerçeve	Derinlik	Açıklama
1	fib(4)	1	fib(3) + fib(2) bekler
2	fib(3)	2	fib(2) + fib(1) bekler
3	fib(2)	3	fib(1) + fib(0) bekler
4	fib(1)	4	taban durum: return 1
5	fib(0)	4	taban durum: return 0
	fib(2) dönüş	3	1 + 0 = 1
6	fib(1)	3	taban durum: return 1
	fib(3) dönüş	2	1 + 1 = 2
7	fib(2)	2	fib(1) + fib(0) bekler
8	fib(1)	3	taban durum: return 1
9	fib(0)	3	taban durum: return 0
	fib(2) dönüş	2	1 + 0 = 1
	fib(4) dönüş	1	2 + 1 = 3

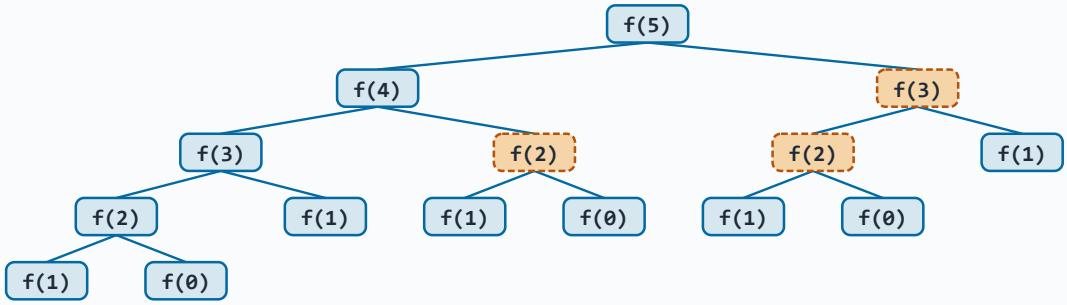
Tablo 12.18: fib(4) çağrılarının sırası. “Derinlik” sütunu main hariç çerçeve sayısıdır; numarasız satırlar bir çerçevenin iki alt sonucu toplayıp döndüğü anlardır.

Tablo 12.18 dokuz çağrının sırasını verir. fib(4), fib(3)'ü çağırır; fib(3), fib(2)'yi; fib(2) de fib(1)'i. fib(1) ve fib(0) taban durumdur. fib(2) = 1 + 0 = 1 hesaplanır. fib(3) sağ terimi için fib(1)'i çağırır ve 1 + 1 = 2 bulur. Sonra fib(4) sağ terimi için fib(2)'yi çağırır. 7 numaralı bu çağrı, 3 numaralı çağrının yaptığı işin aynısını baştan yapar. fib(2) iki kez, fib(1) üç kez, fib(0) iki kez hesaplanır. Sonuç 3 olsa da 9 çağrı yapılmıştır. Yığın derinliği ise en çok 4'tür: aynı anda yalnızca kökten yaprağa giden bir yolun çerçeveleri açıktır.



Şekil 12.21: fib(4) çağrısının tamamlanmış özyineleme ağacı. Her düğümün içinde döndürdüğü değer yazar. Sağ alt ağaçtaki f(2), sol alt ağaçta zaten hesaplanmış bir alt problemin tekrarıdır.

n büyüdükçe tekrar hızla artar. Şekil 12.22 fib(5) ağacını gösterir. 15 çağrı yapılır: f(3) 2 kez, f(2) 3 kez, f(1) 5 kez, f(0) 3 kez hesaplanır. Ağacın sağ yarısındaki f(3) alt ağacı, sol yarıdaki f(4) alt ağacının içinde zaten vardır. fib(n − 1) ve fib(n − 2) alt ağaçları büyük ölçüde örtüşür.



Şekil 12.22: fib(5) özyineleme ağacı: 15 çağrı. Kesikli turuncu düğümler daha önce hesaplanmış bir alt problemin tekrarıdır.

12.9.2 Çağrı sayısının büyümesi

Çağrı sayısı en güvenilir biçimde sayılarak bulunur. Aşağıdaki programda metoda bir sayaç eklenmiştir. Her çağrı sayacı 1 artırır:

```
public class FibSayac {
    static long cagri = 0;

    static long fib(int n) {
        cagri++;
        if (n <= 1) return n;
        return fib(n - 1) + fib(n - 2);
    }

    public static void main(String[] args) {
        for (int n : new int[]{5, 10, 20, 30}) {
            cagri = 0;
            long sonuc = fib(n);
            System.out.println("fib(" + n + ") = " + sonuc + ", çağrı sayısı = " + cagri);
        }
    }
}
```

```
fib(5) = 5, çağrı sayısı = 15
fib(10) = 55, çağrı sayısı = 177
fib(20) = 6765, çağrı sayısı = 21891
fib(30) = 832040, çağrı sayısı = 2692537
```

Sayılar bir formülle de bulunabilir. ç(n), fib(n) için yapılan çağrı sayısı olsun. İlk çağrı 1 sayılır; ona iki alt ağacın çağrıları eklenir: *ç(n) = 1 + ç(n − 1) + ç(n − 2), ç(0) = ç(1) = 1*. Buradan ç(2) = 3, ç(3) = 5, ç(4) = 9, ç(5) = 15 ve ç(6) = 25 bulunur. Bu yineleme bağıntısının çözümü *ç(n) = 2·fib(n + 1) − 1*'dir. Örneğin fib(11) = 89 olduğundan ç(10) = 177'dir.

n	fib(n)	Özyinelemeli: çağrı sayısı	Döngülü: tur sayısı
5	5	15	5
10	55	177	10
20	6765	21.891	20
30	832.040	2.692.537	30
40	102.334.155	331.160.281	40

Tablo 12.19: Özyinelemeli ve döngülü Fibonacci'nin iş miktarı. Sayılar, her çağrıda ve her turda bir sayaç artırılarak elde edilmiştir; süre ölçümü değildir.

Tablo 12.19 iki büyüme biçimini yan yana koyar. Döngülü sürüm n tur döner; iş n ile doğru orantılıdır. Özyinelemeli sürümün çağrı sayısı ise n her 10 arttığında yaklaşık 123 katına çıkar. Bunun nedeni Fibonacci sayılarının kendisidir. Ardışık iki Fibonacci sayısının oranı yaklaşık 1,618'e yaklaşır. Bu sayıya altın oran denir ve φ ile gösterilir. $\varphi^{10} \approx 123$'tür. Çağrı sayısı fib(n + 1) ile orantılı olduğu için o da

her adımda yaklaşık 1,618 katına çıkar. Zaman $O(\varphi^n)$'dir. Sunumdaki $O(2^n)$ yazımı bunun kaba bir üst sınırıdır: her çağrı en çok iki çağrı yapar ve ağacın derinliği n'dir. Hangisiyle ifade edilirse edilsin büyüme üsteldir.

Üstel büyümenin pratikteki anlamı şudur: fib(40) için 331 milyon çağrı yapılır. fib(50) için bu sayı 40 milyarı aşar. Döngülü sürüm ise fib(50)'yi 50 turda bulur. Yığın derinliği iki sürümde de sorun değildir: özyinelemeli sürüm en çok n çerçeve açar. Sorun zamandadır.

Sık yapılan hata

Tanıma birebir uyan her özyinelemeli metodun verimli olduğunu sanmak. Doğrudan Fibonacci metodu doğrudur ama aynı alt problemleri tekrar tekrar çözer. Özyineleme ağacında aynı argümanla birden çok düğüm görülüyorsa metot gereksiz iş yapıyor demektir. Bu tekrar, sonuçları saklayarak ya da hesabı döngüyle küçükten büyüğe yaparak giderilir.

12.9.3 Bellekleme

Tekrarın çaresi basittir: bir alt problemin sonucu ilk kez hesaplandığında bir yere yazılır. Aynı alt problem yeniden istendiğinde hesap yapılmaz; kayıtlı değer döndürülür. Bu yönteme **bellekleme** denir.

Tanım 12.7

Bellekleme (*memoization*), özyinelemeli bir metodun hesapladığı alt problem sonuçlarını bir dizide ya da tabloda saklaması ve aynı girdi yeniden geldiğinde özyineleme ağacını yeniden açmadan kayıtlı değeri döndürmesidir.

Fibonacci için kayıt yeri bir `long` dizisidir. bellek[k], fib(k)'nin sonucunu tutar. Java'da yeni bir sayı dizisinin bütün hücreleri 0 ile başlar. fib(k) ≥ 1 olduğu için (k ≥ 1) 0 değeri “henüz hesaplanmadı” anlamında kullanılabilir.

```
static long fib(int n, long[] bellek) {
    if (n <= 1) return n;
    if (bellek[n] != 0) return bellek[n];
    bellek[n] = fib(n - 1, bellek)
              + fib(n - 2, bellek);
    return bellek[n];
}
long[] bellek = new long[6];
System.out.println(fib(5, bellek));
```

```
5
```

Metot düz sürümden yalnızca üç noktada ayrılır:

1. Diziyi bir parametre olarak alır. Bütün çağrılar aynı dizi nesnesini paylaşır; bir çağrının yazdığı değeri öteki çağrılar görür.
2. Taban durumdan sonra kayda bakar: `bellek[n] != 0` ise sonuç zaten bilinir ve hemen döndürülür.
3. Sonucu döndürmeden önce kaydeder: `bellek[n] = fib(n - 1, bellek) + fib(n - 2, bellek);`.

Bellekli sürümün çağrı sayısı da sayaçla ölçülmüştür:

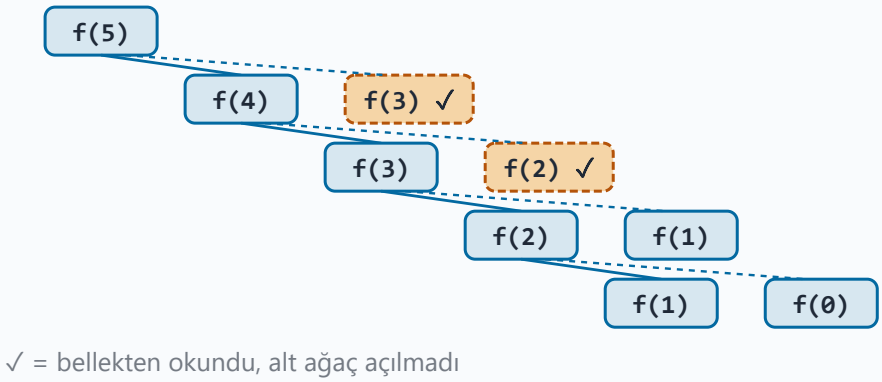
```
public class BellekSayac {
    static long cagri = 0;

    static long fib(int n, long[] bellek) {
        cagri++;
        if (n <= 1) return n;
        if (bellek[n] != 0) return bellek[n];
        bellek[n] = fib(n - 1, bellek)
                  + fib(n - 2, bellek);
        return bellek[n];
    }

    public static void main(String[] args) {
        for (int n : new int[]{5, 10, 20, 30, 50}) {
            cagri = 0;
            long sonuc = fib(n, new long[n + 1]);
            System.out.println("fib(" + n + ") = " + sonuc + ", çağrı sayısı = " + cagri);
        }
    }
}
```

```
fib(5) = 5, çağrı sayısı = 9
fib(10) = 55, çağrı sayısı = 19
fib(20) = 6765, çağrı sayısı = 39
fib(30) = 832040, çağrı sayısı = 59
fib(50) = 12586269025, çağrı sayısı = 99
```

1 ve üstü n için çağrı sayısı *2n − 1*'dir. Nedeni şudur: fib(n), fib(n − 1), …, fib(2) çağrılarının her biri bir kez hesaplanır ve iki çağrı yapar. Bunlardan n − 1 tane vardır; ilk çağrı da eklenince *1 + 2(n − 1) = 2n − 1* çıkar. fib(30) için çağrı sayısı 2.692.537'den 59'a iner. Zaman *O(n)* olur. Bunun karşılığında n + 1 hücrelik bir dizi kullanılır: ek bellek *O(n)*'dir. Zamandan kazanmak için bellek harcanmıştır.



Şekil 12.23: Belleklemeli fib(5) çağrıları. Ağaç bir zincire dönüşmüştür: 9 çağrı yapılır. ✓ işaretli f(3) ve f(2) bellekten okunur; altlarında yeni alt ağaç açılmaz.

12.9.4 Belleklemeli fib(5)'i izlemek

Çağrı no	Çerçeve	İş	bellek
1	fib(5)	bellek[5] == 0: hesaplanacak	[0, 0, 0, 0, 0, 0]
2	fib(4)	bellek[4] == 0: hesaplanacak	[0, 0, 0, 0, 0, 0]
3	fib(3)	bellek[3] == 0: hesaplanacak	[0, 0, 0, 0, 0, 0]
4	fib(2)	bellek[2] == 0: hesaplanacak	[0, 0, 0, 0, 0, 0]
5	fib(1)	taban durum: return 1	[0, 0, 0, 0, 0, 0]
6	fib(0)	taban durum: return 0	[0, 0, 0, 0, 0, 0]
	fib(2) dönüş	bellek[2] = 1 kaydedildi	[0, 0, 1, 0, 0, 0]
7	fib(1)	taban durum: return 1	[0, 0, 1, 0, 0, 0]
	fib(3) dönüş	bellek[3] = 2 kaydedildi	[0, 0, 1, 2, 0, 0]
8	fib(2)	bellek[2] = 1: bellekten return 1	[0, 0, 1, 2, 0, 0]
	fib(4) dönüş	bellek[4] = 3 kaydedildi	[0, 0, 1, 2, 3, 0]
9	fib(3)	bellek[3] = 2: bellekten return 2	[0, 0, 1, 2, 3, 0]
	fib(5) dönüş	bellek[5] = 5 kaydedildi	[0, 0, 1, 2, 3, 5]

Tablo 12.20: Belleklemeli fib(5) izlemesi. Son sütun, satırdaki işten sonraki bellek dizisidir.

Tablo 12.20 dokuz çağrıyı ve belleğin dolmasını gösterir. İlk dört çağrı fib(5), fib(4), fib(3) ve fib(2)'dir. Bunların hiçbiri bellekte yoktur ve hepsi hesaplanmak üzere sol terimlerini çağırır. fib(1) ve fib(0) taban durumdur. fib(2) = 1 + 0 = 1 hesaplanır ve bellek[2]'ye yazılır. fib(3) sağ terimi için fib(1)'i çağırır ve 2 bulur; bellek[3] = 2 olur.

Belleklemenin etkisi 8. çağrıda görülür. fib(4) sağ terimi için fib(2)'yi çağırır. Düz sürümde bu çağrı yeni bir alt ağaç açardı. Burada bellek[2] = 1'dir ve değer hemen döner. 9. çağrıda fib(5)'in istediği fib(3) de bellekten gelir. Sonunda bellek dizisi [0, 0, 1, 2, 3, 5] olur ve 5 yazdırılır. bellek[0] ve bel-

lek[1] hiç kullanılmaz, çünkü 0 ve 1 taban durumdur.



Şekil 12.24: Bellek dizisinin iki anı. Üstte 8. çağrı: fib(2) bellekten okunur, bellek[2] = 1 ve bellek[3] = 2 doludur. Altta son durum: [0, 0, 1, 2, 3, 5], toplam 9 çağrı.

Çalışma sorularındaki sürüm diziyi parametre yerine bir `static` alan olarak tutar: `static long[] bellek = new long[50];`. Bu biçimde dizi bütün programda bir kez oluşturulur ve sonraki fib çağrıları önceki hesapları da kullanır. 50 hücreli dizi fib(49)'a kadar yeter; fib(49) = 7.778.742.049 int'e sığmaz ama long'a sığar.

12.9.5 Belleklemeyle kombinasyon

Bellekleme yalnızca Fibonacci'ye özgü değildir. Alt problemleri örtüşen her özyinelemede işe yarar. Sunumun alıştırmalarından biri kombinasyon sayısıdır. n öğeden k tanesini seçmenin yollarının sayısı C(n, k) ile gösterilir ve Pascal üçgeni kuralını sağlar: *C(n, k) = C(n − 1, k − 1) + C(n − 1, k)*. Kural şöyle açıklanır: belirli bir öğe ya seçilir (kalan n − 1 öğeden k − 1 seçilir) ya da seçilmez (kalan n − 1 öğeden k seçilir). Taban durumlar *C(n, 0) = 1* ve *C(n, n) = 1*'dir: hiç öğe seçmenin ve bütün öğeleri seçmenin birer yolu vardır.

Örnek 12.2

Aşağıdaki program kombinasyonu düz ve belleklemeli olarak hesaplar ve iki sürümün çağrı sayılarını karşılaştırır. Belleklemeli sürüm iki boyutlu bir `long` dizisi kullanır; bellek[n][k], C(n, k)'nin sonucunu tutar.

```
// Sunumdaki alıştırma: kombinasyon(n, k) = kombinasyon(n - 1, k - 1) +
kombinasyon(n - 1, k)
public class Kombinasyon {
    static long cagri;

    static long kombinasyon(int n, int k) {
        cagri++;
        if (k == 0 || k == n) return 1;
        return kombinasyon(n - 1, k - 1) + kombinasyon(n - 1, k);
    }

    static long kombinasyonB(int n, int k, long[][] bellek) {
        cagri++;
        if (k == 0 || k == n) return 1;
        if (bellek[n][k] != 0) return bellek[n][k];
        bellek[n][k] = kombinasyonB(n - 1, k - 1, bellek) + kombinasyonB(n - 1, k,
bellek);
        return bellek[n][k];
    }

    public static void main(String[] args) {
        int[][] g = {{4, 2}, {5, 2}, {10, 5}, {20, 10}, {30, 15}};
        for (int[] x : g) {
            cagri = 0;
            long v = kombinasyon(x[0], x[1]);
            long c1 = cagri;
            cagri = 0;
            long w = kombinasyonB(x[0], x[1], new long[x[0] + 1][x[1] + 1]);
            System.out.println("C(" + x[0] + ", " + x[1] + ") = " + v + " / " + w +
": belleksiz " + c1 + " cagri, bellekli " + cagri + " cagri");
        }
    }
}
```

```
C(4, 2) = 6 / 6: belleksiz 11 cagri, bellekli 9 cagri
C(5, 2) = 10 / 10: belleksiz 19 cagri, bellekli 13 cagri
C(10, 5) = 252 / 252: belleksiz 503 cagri, bellekli 51 cagri
C(20, 10) = 184756 / 184756: belleksiz 369511 cagri, bellekli 201 cagri
C(30, 15) = 155117520 / 155117520: belleksiz 310235039 cagri, bellekli 451 cagri
```

İki sürüm aynı sonucu verir: C(5, 2) = 10, C(30, 15) = 155.117.520. Düz sürümün çağrı sayısı *2·C(n, k) − 1*'dir, çünkü sonuç ağacın yapraklarındaki 1'lerin toplanmasıyla kurulur. C(30, 15) için bu, 310 milyondan fazla çağrı demektir. Belleklemeli sürüm her (n, k) çiftini bir kez hesaplar ve aynı sonucu 451 çağrıda bulur.

12.10 Kuyruk özyinelemesi

12.10.1 Birikim parametresi

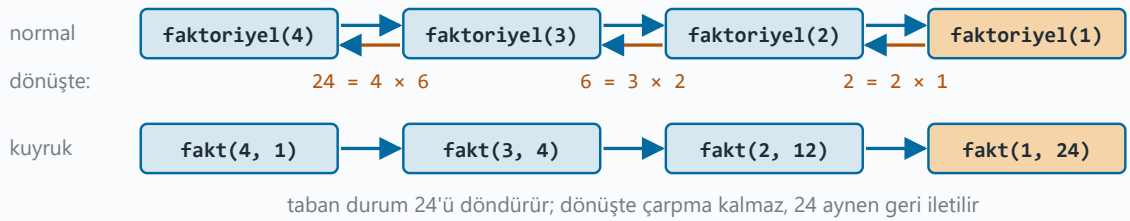
Özyinelemeli faktöriyelde her çerçeve bir işi dönüş yoluna bırakır: alt sonucu n ile çarpmak. Çerçevenin bekleyişi bu yüzdendir. Aynı hesap, çarpma çağrıdan önce yapılacak biçimde de yazılabilir. Bunun için ara sonuç ikinci bir parametrede taşınır. Bu parametreye birikim denir.

```
static long fakt(int n, long birikim) {
    if (n <= 1) return birikim;
    return fakt(n - 1, n * birikim);
}
// fakt(4, 1) → fakt(3, 4)
//   → fakt(2, 12) → fakt(1, 24) → 24
```

fakt(n, birikim), *n! × birikim* değerini hesaplar. İlk çağrıda birikim 1 verilir: fakt(4, 1) = 4!. Özyinelemeli durumda çarpma çağrının içinde, argüman hesaplanırken yapılır: `n * birikim`. Çağrı döndüğünde yapılacak iş kalmaz; alt çağrının sonucu olduğu gibi döndürülür. Taban durumda birikim, aranan sonucun tamamını taşır.

Tanım 12.8

Kuyruk özyinelemesi (*tail recursion*), özyinelemeli çağrının metodun son işlemi olduğu özyinelemedir. Çağrı döndükten sonra çerçevede yapılacak hesap kalmaz; alt çağrının sonucu aynen döndürülür.



Şekil 12.25: Normal ve kuyruk özyinelemesi. Solda normal özyinelemede çarpmalar dönüşte yapılır: 2 = 2 × 1, 6 = 3 × 2, 24 = 4 × 6. Sağda kuyruk özyinelemesinde birikim ileri taşınır; taban durum 24'ü döndürür ve dönüşte çarpma kalmaz.

Dönüşte iş kalmadığına göre eski çerçeveyi saklamaya gerek yoktur. Bazı diller ve derleyiciler bu durumu tanır. Yeni çerçeve açmak yerine eski çerçeveyi yeni argümanlarla yeniden kullanırlar. Bu dönüşüme kuyruk çağrısı eniyilemesi (*tail call optimization*) denir. Eniyileme yapılırsa kuyruk özyinelemesi bir döngü kadar az bellek kullanır: *O(1)*. Scheme gibi dillerin standardı bunu zorunlu tutar.

Java ise bu eniyilemeyi garanti etmez. Java derleyicisi ve JDK 21'in sanal makinesi kuyruk çağrılarını döngüye çevirmez. Her çağrı yine yeni bir çerçeve açar. Bu yüzden Java'da kuyruk özyinelemesi yığın güvenliği sağlamaz: fakt(100.000, 1) da derin bir doğrusal özyineleme gibi taşar. Java'da derin bir kuyruk özyinelemesi gerekiyorsa onu döngüye çevirmek programcının işidir. Çeviri de kolaydır: parametreler döngü değişkeni olur, özyinelemeli çağrı da bu değişkenlerin güncellenmesine dönüşür.

12.10.2 fakt(4, 1)'i izlemek

```
static long fakt(int n, long birikim) {
    if (n <= 1) return birikim;
    return fakt(n - 1, n * birikim);
}
System.out.println(fakt(4, 1)); // 24
```

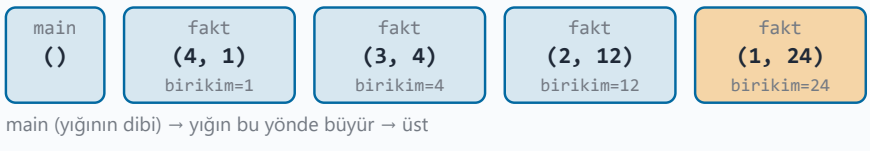
Program 24 yazar. Çalıştırılan sürüm fakt(20, 1), fakt(1, 1) ve fakt(0, 1) sonuçlarını da yazdırmıştır:

```
24
2432902008176640000 1 1
```

Çerçeve	n	birikim	Çağrıdan önce yapılan iş	Sonraki adım
fakt(4, 1)	4	1	4 × 1 = 4	fakt(3, 4)
fakt(3, 4)	3	4	3 × 4 = 12	fakt(2, 12)
fakt(2, 12)	2	12	2 × 12 = 24	fakt(1, 24)
fakt(1, 24)	1	24	1 <= 1 → true	return 24

Tablo 12.21: fakt(4, 1) izlemesi. Çarpmalar her çağrıdan önce yapılır.

Tablo 12.21 zinciri verir: fakt(4, 1) → fakt(3, 4) → fakt(2, 12) → fakt(1, 24) → 24. Çarpmalar *4 × 1 = 4*, *3 × 4 = 12* ve *2 × 12 = 24*'tür ve hepsi gidiş yolunda yapılır. Normal faktöriyelde çarpmalar 2, 6, 24 sırasıyla dönüşte yapılıyordu. Burada çarpma sırası 4, 12, 24'tür ve dönüşte yalnızca 24 taşınır.



Şekil 12.26: fakt(4, 1) çağrısında taban duruma ulaşıldığı an. Dört fakt çerçevesi yine üst üste durur: Java eski çerçeveleri silmez. main ile birlikte 5 çerçeve vardır.

Şekil 12.26 Java'daki asıl durumu gösterir. Eski çerçevelerin bekleyecek bir işi yoktur ama yığında durmaya devam ederler. En derin anda main ile birlikte 5 çerçeve vardır; bu, normal faktoriyel(4) ile aynıdır. Kuyruk özyinelemesi Java'da belleği azaltmaz; ek bellek yine *O(n)*'dir. Yöntemin yararı başka dillerde ortaya çıkar. Java'da ise bir döngüye çevirmenin ara adımı olarak düşünülebilir.

> **Sık yapılan hata**
>
> İlk çağrıda birikimi yanlış başlatmak. Çarpma biriktiren bir metotta birikim 1 ile, toplama biriktiren bir metotta 0 ile başlar. fakt(4, 0) çağrısı her adımda 0 ile çarptığı için 0 döndürür. Bu hatayı önlemek için çoğu zaman tek parametreli bir sarmalayıcı metot yazılır: `static long faktoriyel(int n) { return fakt(n, 1); }`. Dışarıdan yalnızca bu metot çağrılır.

12.11 Taşma, uç durumlar ve hata ayıklama

12.11.1 long taşması

Yığın taşması ile tam sayı taşması farklı şeylerdir. Yığın taşmasında çağrılar belleğe sığmaz. Tam sayı taşmasında ise hesaplanan sayı, türün tutabileceği en büyük değeri aşar. Faktöriyel bu ikinci türe çok çabuk ulaşır. long en çok 9.223.372.036.854.775.807 tutar. 20! = 2.432.902.008.176.640.000 bu sınırın altındadır. 21! = 51.090.942.171.709.440.000 ise sınırın yaklaşık beş buçuk katıdır.

Java tam sayı taşmasında hata fırlatmaz. Sonuç sessizce sarar: sınırı aşan kısım atılır ve sayı negatif bölgeden geri gelir. Aşağıdaki program faktoriyel metodunu 19'dan 22'ye kadar çalıştırmıştır:

```
public class Tasma {
    static long faktoriyel(int n) {
        if (n < 0)
            throw new IllegalArgumentException();
        if (n <= 1) return 1;
        return n * faktoriyel(n - 1);
    }

    public static void main(String[] args) {
        for (int n = 19; n <= 22; n++)
            System.out.println("faktoriyel(" + n + ") = " + faktoriyel(n));
        System.out.println("Long.MAX_VALUE = " + Long.MAX_VALUE);
        System.out.println("21! (BigInteger) = " +
java.math.BigInteger.valueOf(21).multiply(new
java.math.BigInteger("2432902008176640000")));
        System.out.println("fib(92), fib(93) long: " + fib(92) + " " + fib(93));
        System.out.println("hizliUs(2, 63) = " + hizliUs(2, 63) + ", hizliUs(2, 64) = " +
hizliUs(2, 64));
        System.out.println("Integer.MIN_VALUE: Math.abs = " + Math.abs(Integer.MIN_VALUE) +
", Math.abs((long) ...) = " + Math.abs((long) Integer.MIN_VALUE));
    }

    static long fib(int n) {
        long a = 0, b = 1;
        for (int i = 0; i < n; i++) { long c = a + b; a = b; b = c; }
        return a;
    }

    static long hizliUs(long taban, int us) {
        if (us == 0) return 1;
        long yarim = hizliUs(taban, us / 2);
        if (us % 2 == 0) return yarim * yarim;
        return yarim * yarim * taban;
    }
}
```

```
faktoriyel(19) = 121645100408832000
faktoriyel(20) = 2432902008176640000
faktoriyel(21) = -4249290049419214848
faktoriyel(22) = -1250660718674968576
Long.MAX_VALUE = 9223372036854775807
21! (BigInteger) = 51090942171709440000
fib(92), fib(93) long: 7540113804746346429 -6246583658587674878
hizliUs(2, 63) = -9223372036854775808, hizliUs(2, 64) = 0
Integer.MIN_VALUE: Math.abs = -2147483648, Math.abs((long) ...) = 2147483648
```

faktoriyel(21) negatif bir sayı döndürür. Program çökmediği için hata gözden kaçabilir. Aynı sorun öteki örneklerde de vardır. fib(92) long'a sığar, fib(93) sığmaz ve negatif çıkar. hizliUs(2, 63) long'un en küçük değerini, hizliUs(2, 64) ise 0'ı verir. Daha büyük sonuçlar gerekiyorsa `java.math.BigInteger` sınıfı kullanılır; bu sınıfın sayıları bellek yettiği sürece büyüyebilir. Programdaki 21! değeri de BigInteger ile hesaplanmıştır.

12.11.2 Negatif üs ve öteki uç girdiler

us(2, −1) çağrısı 12.4.1'de tartışıldı. Taban durum `n == 0` hiç sağlanmaz; n −1, −2, −3 … diye iner ve çerçeveler birikir. Derinlik programında bu çağrı yaklaşık 8 bin çağrıdan sonra StackOverflowError ile bitmiştir. Bu sayı sonsuz metodunun sayısından küçüktür. Çerçeve boyu metodun parametrelerine ve JVM'in o anki çalışma biçimine bağlı olduğu için taşma noktası metottan metoda değişir.

İki önlem vardır. Birincisi girdiyi denetlemektir. n < 0 ise `IllegalArgumentException` fırlatılır; faktoriyel metodu bunu negatif girdi için zaten yapar. İkincisi negatif üssü tanımlı hâle getirmektir. Sonuç türü `double` yapılır ve negatif üs pozitif üsse çevrilir: $2^{-2} = 1 / 2^2 = 0{,}25$. Çalışma sorularındaki sürüm bunu bir satırla yapar: `if (n < 0) return 1.0 / us(t, -n);`. Negatif dal yalnızca ilk çağrıda çalışır; sonraki çağrılarda üs pozitiftir.

Özyinelemeli bir metot test edilirken şu girdiler denenir:

1. **Taban durumun kendisi:** faktoriyel(0), faktoriyel(1), toplam(0), boş dizgi, tek karakterli dizgi.
2. **Taban durumun bir üstü:** faktoriyel(2), iki karakterli dizgi. Özyinelemeli durumun en basit kullanımı burada sınanır.
3. **Geçersiz girdiler:** negatif n, negatif üs, null dizgi. Metot ya anlamlı bir hata fırlatmalı ya da tanımlı bir değer döndürmelidir.
4. **Büyük girdiler:** sonucun türe sığıp sığmadığı (faktoriyel(21)) ve derinliğin yığına sığıp sığmadığı (toplam(100.000)).
5. **Özel değerler:** rakam problemlerinde 0 ve Integer.MIN_VALUE, aramada dizinin ilk ve son öğesi ile dizide olmayan değer.

12.11.3 Özyinelemeli kodda hata ayıklama

Özyinelemeli bir metot yanlış sonuç veriyorsa ya da hiç bitmiyorsa sorun çoğu zaman iki parçadan birindedir. Hata ayıklama dört adımda yapılır:

1. **Taban duruma ulaşılabildiği gösterilir.** Her geçerli girdi için çağrı zinciri sonunda taban duruma girmelidir. Atlamalı küçülmelerde (n − 2 gibi) ve negatif girdilerde bu ayrıca denetlenir.
2. **Küçülme ölçüsü belirlenir.** Her çağrıda azalan bir büyüklük olmalıdır: n, dizginin uzunluğu, aralığın boyu (hi − lo + 1) ya da kalan öğe sayısı (a.length − i).
3. **Küçük bir girdi tabloya dökülür.** Çağrı ve dönüş anları, derinlik ve dönen değerlerle birlikte yazılır.

4. **Derinlik ve tekrar değerlendirilir.** En büyük derinlik girdinin boyuyla mı, logaritmasıyla mı büyüyor? Özyineleme ağacında aynı argüman birden çok kez görülüyor mu?

Üçüncü adım için metoda geçici yazdırma satırları eklenebilir. Derinlik bir sayaçla tutulur ve her satır derinlik kadar girintiyle yazılır. Böylece çağrıların iç içe yapısı çıktıda görünür. Tablo 12.22 toplam(2) için böyle bir izlemedir.

Derinlik	Olay	Değer
0	toplam(2) çağrıldı	
1	toplam(1) çağrıldı	
2	toplam(0) çağrıldı	
2	toplam(0) döndü	0
1	toplam(1) döndü	1
0	toplam(2) döndü	3

Tablo 12.22: toplam(2) için girintili çağrı ve dönüş izlemesi. Derinlik 0, ilk çağrıdır.

Tablo bir çağrının açılışını ve kapanışını aynı girinti düzeyinde gösterir. toplam(2) derinlik 0'da açılır ve en son kapanır. Arada toplam(1) ve toplam(0) açılıp kapanır. Dönen değerler 0, 1 ve 3'tür. Beklenen değerle karşılaştırma yapılırken hangi çağrının ilk kez yanlış değer döndürdüğüne bakılır. Hata o çağrının kendi satırlarındadır, çünkü onun kullandığı bütün alt sonuçlar doğrudur.

Bir metodun ne kadar derine indiği ve kaç çağrı yaptığı, sayaçlı bir sürümle de ölçülebilir. Bu notta verilen bütün çağrı sayıları ve derinlikler böyle ölçülmüştür. Sayaç, metodun girişinde artırılan bir `static` alandır. Derinlik için girişte artırılıp çıkışta azaltılan ikinci bir alan ve görülen en büyük değeri tutan üçüncü bir alan yeter.

Özyineleme Görselleştirici simülatörü bu izlemeleri etkileşimli olarak gösterir. Simülatörde faktoriyel, fib, us ve hanoi metotlarının çağrıları adım adım açılır ve kapanır. Çağrı yığını ve özyineleme ağacı aynı anda izlenir. Fibonacci'de bellekleme açılıp kapatılarak iki sürümün çağrı sayıları karşılaştırılabilir.

12.12 Özet

- Özyineleme, bir metodun problemi aynı problemin daha küçük bir örneğine indirip kendini çağırmasıdır. Her özyinelemeli metotta iki parça bulunur: sonucun doğrudan bilindiği taban durum ve problemi küçültüp kendini çağıran özyinelemeli durum.
- Problem her çağrıda ölçülebilir biçimde küçülmeli ve taban duruma ulaşmalıdır. Atlamalı küçülmelerde ve negatif girdilerde taban durum `==` yerine `<=` ile yazılırsa daha güvenlidir.

- Her çağrı, parametreleri, yerel değişkenleri ve dönüş noktasını tutan bir yığın çerçevesi açar. Çerçeveler açıldıkları sıranın tersine kapanır. Özyinelemeli çağrıdan önceki işler açılımda, sonraki işler dönüşte ve ters sırada yapılır.
- Doğrusal özyineleme (faktöriyel, toplam, düz üs) n çerçeve açar: zaman ve ek yığın *O(n)*'dir. Rakamlar toplamı, hızlı üs ve ikili arama gibi her adımda büyük ölçüde küçülen problemlerde derinlik logaritmiktir.
- Böl ve fethet, problemi parçalara bölüp parçaları özyinelemeyle çözer ve birleştirir. Birleştirme sıralaması *O(n log n)* zamanda, *O(n)* ek bellekle ve *O(log n)* yığın derinliğiyle çalışır.
- Her çağrının iki çağrı yaptığı metotlar bir özyineleme ağacı oluşturur. Hanoi kuleleri $2^n - 1$ hamle, alt küme üretimi 2^n alt küme üretir. Bu problemlerde üstel iş problemin kendisinden gelir; yığın derinliği ise n + 1'dir.
- Taban durumu olmayan ya da çok derine inen özyineleme yığını doldurur ve StackOverflowError ile biter. JDK 21'de varsayılan yığın birkaç on bin çağrı taşıyabilmiştir.
- Basit doğrusal tekrarlarda döngü tek çerçevede, *O(1)* ek yığınla çalışır ve taşmaz. Kendine benzeyen problemlerde özyineleme tanıma yakın ve okunaklıdır. Seçim doğruluk, okunabilirlik ve kaynak kullanımına göre yapılır.
- Doğrudan Fibonacci aynı alt problemleri tekrar tekrar çözer. Çağrı sayısı $2 \cdot fib(n + 1) - 1$'dir ve üstel büyür: fib(30) için 2.692.537 çağrı.
- Bellekleme, alt problem sonuçlarını saklayarak tekrarı kaldırır. Belleklemeli fib(n) $2n - 1$ çağrı yapar; zaman *O(n)*'e iner, karşılığında *O(n)* bellek kullanılır.
- Kuyruk özyinelemesinde çağrı metodun son işidir ve ara sonuç bir birikim parametresinde taşınır. Java kuyruk çağrılarını eniyilemez; çerçeveler yine birikir ve ek bellek *O(n)* kalır.
- Tam sayı taşması yığın taşmasından farklıdır ve sessizdir: long faktoriyel(21) negatif bir sayı döndürür. Özyinelemeli metotlar taban durum, taban durumun bir üstü, geçersiz girdi ve büyük girdilerle test edilir.

12.13 Kavram listesi

özyineleme *recursion* · taban durum *base case* · özyinelemeli durum *recursive case* · çağrı yığını *call stack* · yığın çerçevesi *stack frame* · özyineleme derinliği *recursion depth* · özyineleme ağacı *recursion tree* · yardımcı metot *helper method* · fırlatma *throw (exception)* · palindrom *palindrome* · en büyük ortak bölen *greatest common divisor* · Öklid algoritması *Euclid's algorithm* · alt dizgi *substring* · böl ve fethet *divide and conquer* · ikili arama *binary search* · Hanoi kuleleri *Towers of Hanoi* · alt küme *subset* · geri izleme *backtracking* · yığın taşması *stack overflow* · sonsuz özyineleme *infinite recursion* · döngülü *iterative (version)* · zaman karmaşıklığı *time complexity* · üstel *exponential (growth)* · Fibonacci dizisi *Fibonacci sequence* · bellekleme *memoization*

kuyruk özyinelemesi *tail recursion* | taşma *overflow (integer)*

12.14 Kendini Dene

1. toplam(5) çağrısında (toplam(n) = n + toplam(n − 1), toplam(0) = 0) toplam metodu kaç kez çağrılır? Yığın en derin anında main dahil kaç çerçeve içerir? Sonuç kaçtır?

   Cevap

   toplam(5), toplam(4), toplam(3), toplam(2), toplam(1) ve toplam(0) çağrılır: 6 çağrı. toplam(0) çağrıldığında hiçbir çağrı henüz dönmemiştir; yığında main ile birlikte 7 çerçeve vardır. Sonuç *5 + 4 + 3 + 2 + 1 + 0 = 15*'tir.

2. ebob(21, 13) çağrısının zinciri nasıldır? Kaç çağrı yapılır ve sonuç kaçtır?

   Cevap

   ebob(21, 13) → ebob(13, 8) → ebob(8, 5) → ebob(5, 3) → ebob(3, 2) → ebob(2, 1) → ebob(1, 0). Toplam 7 çağrı yapılır ve sonuç 1'dir. 21 ve 13 ardışık Fibonacci sayılarıdır. Bu yüzden kalanlar da Fibonacci sayıları olarak birer birer küçülür; Öklid algoritmasının en yavaş olduğu durum budur.

3. hizliUs(2, 13) çağrısında üs hangi değerlerden geçer? Kaç çağrı yapılır? Dönüş yolunda hangi çerçeveler tabanla bir kez daha çarpar? Düz us(2, 13) kaç çağrı yapardı?

   Cevap

   Üs 13, 6, 3, 1 ve 0 olur: 5 çağrı. Dönüşte üssü tek olan çerçeveler (1, 3 ve 13) sonucu bir kez daha 2 ile çarpar. Değerler sırasıyla 1, 2, 8, 64 ve 8192'dir: *1·1·2 = 2*, *2·2·2 = 8*, *8·8 = 64*, *64·64·2 = 8192*. Düz yöntem *n + 1 = 14* çağrı yapardı.

4. Hanoi kulelerinde 6 disk için kaç hamle gerekir? Notta verilen hanoi metodu (hanoi(0) çağrıları dahil) kaç kez çağrılır?

   Cevap

   Hamle sayısı $2^6 - 1 = 63$'tür. Çağrı sayısı $2^7 - 1 = 127$'dir: özyineleme ağacının her düzeyinde çağrı sayısı ikiye katlanır ve 7 düzey vardır (n = 6'dan n = 0'a). Yaprakların hepsi hamle yapmayan hanoi(0) çağrılarıdır.

5. fib(10) hesaplanırken düz özyinelemeli sürüm ve belleklemeli sürüm kaçar çağrı yapar? Hangi

formüllerle bulunur?

Cevap

Düz sürüm $2 \cdot fib(11) - 1 = 2 \cdot 89 - 1 = 177$ çağrı yapar. Belleklemeli sürüm $2 \cdot 10 - 1 = 19$ çağrı yapar. Sonuç ikisinde de fib(10) = 55'tir.

6. Notta verilen altKume metodu {1, 2, 3} dizisiyle çağrılıyor. Kaç satır yazdırılır? 4. satır nedir? Metot kaç kez çağrılır?

Cevap

$2^3 = 8$ satır yazdırılır: [1, 2, 3], [1, 2], [1, 3], [1], [2, 3], [2], [3], []. 4. satır [1]'dir: 1 dahil edilmiş, 2 ve 3 hariç tutulmuştur. Çağrı sayısı $2^4 - 1 = 15$'tir.

7. a = {2, 5, 8, 12, 16, 23, 38, 56} dizisinde ara(a, 5, 0, 7) çağrısı kaç çağrı yapar ve ne döndürür?

Cevap

İlk çağrıda mid = 3 ve a[3] = 12'dir; $12 > 5$ olduğundan ara(a, 5, 0, 2) çağrılır. Bu çağrıda mid = 1 ve a[1] = 5'tir: değer bulunmuştur. 2 çağrı yapılır ve 1 döner.

8. 1'den n'ye toplamın kuyruk özyinelemeli sürümü şöyledir: `static int toplamK(int n, int birikim) { if (n == 0) return birikim; return toplamK(n - 1, birikim + n); }`. toplamK(4, 0) çağrısının zinciri nasıldır? Java'da en derin anda yığında main dahil kaç çerçeve vardır?

Cevap

toplamK(4, 0) → toplamK(3, 4) → toplamK(2, 7) → toplamK(1, 9) → toplamK(0, 10) → 10. Toplamalar çağrıdan önce yapılır ve dönüşte iş kalmaz. Java kuyruk çağrılarını eniyilemediği için beş toplamK çerçevesinin hepsi yığındadır; main ile birlikte 6 çerçeve vardır.

9. Bir dizgide bir karakterin kaç kez geçtiğini sayan say(s, c, i) metodu i'nci indisten başlayarak sayıyor. Taban durum ve özyinelemeli durum ne olmalıdır? say("kayak", 'a', 0) kaç çağrı yapar ve ne döndürür?

Cevap

Taban durum `i == s.length()` 'tir: bakılacak karakter kalmamıştır ve 0 döner. Özyinelemeli durumda s.charAt(i) c'ye eşitse 1, değilse 0 alınır ve say(s, c, i + 1) ile toplanır. “kayak” için i = 0'dan 5'e 6 çağrı yapılır. 'a' 1. ve 3. indislerdedir; sonuç 2'dir.

12.15 İleri okuma

- R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach* (Addison-Wesley, 2016), Kesim 2.3: özyineleme Öklid algoritması, Hanoi kuleleri, Gray kodları ve özyinelemeli grafiklerle anlatılır; doğrudan Fibonacci'nin neden yavaş olduğu ve belleklemeyle hızlandırılması da bu kesimdedir. Çevrim içi sürümü introcs.cs.princeton.edu adresindedir.
- E. Roberts, *Thinking Recursively with Java* (Wiley, 2006): bütünüyle özyinelemeli düşünmeye ayrılmış bir kitaptır. Küçük problemin çözüldüğünü varsayıp büyük problemi kurma alışkanlığını adım adım kazandırır.
- A. B. Downey, C. Mayfield, *Think Java* (2. baskı, O'Reilly, 2019): özyinelemeli metotlar ve yığın diyagramları programlamaya yeni başlayanlar için sade bir dille anlatılır; kitabın çevrim içi sürümü ücretsizdir.
- Oracle, *Java SE 21 API*: StackOverflowError sınıfının resmî açıklaması; hatanın bir uygulama çok derine özyinelendiğinde fırlatıldığı burada belirtilir.

12.16 Kaynaklar

1. E. Roberts, *Thinking Recursively with Java*, Wiley, 2006.
2. R. Sedgewick, K. Wayne, *Computer Science: An Interdisciplinary Approach*, Addison-Wesley, 2016: 2.3 Recursion.
3. T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms*, 4. baskı, MIT Press, 2022.
4. Y. D. Liang, *Introduction to Java Programming and Data Structures*, 13. baskı, Pearson, 2024.
5. Oracle, *Java SE 21 API: StackOverflowError*.
6. A. B. Downey, C. Mayfield, *Think Java*, 2. baskı, O'Reilly, 2019.

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

absolute path → mutlak yol 7

absolute reference → mutlak başvuru 9

absolute value → mutlak değer 16

abstraction → soyutlama 4

accept / reject → kabul et / reddet 8

access (public) → erişim 14

access point → erişim noktası 3

accessibility → erişilebilirlik 5

accessibility checker → erişilebilirlik denetleyicisi 8

accumulator → biriktirici 13

accumulator (ACC) → akümülatör (ACC) 4

acknowledgment (ACK) → onay (ACK) 2

action button → eylem düğmesi 10

active cell → etkin hücre 9

actual argument → gerçek argüman 18

actuator → eyleyici 1

ADC / DAC → ADC (analogdan dijitale dönüştürücü) / DAC (dijitalden analoğa dönüştürücü) 3

address bar → adres çubuğu 7

address space → adres alanı 6

adjacent → komşu 12

administrator → yönetici 7

administrator account → yönetici hesabı 6

agile → çevik 11

algorithm → algoritma 5

algorithm analysis → algoritma analizi 12

algorithm design → algoritma tasarımı 5

align / distribute → hizala / dağıt 10

alignment → hizalama 8

alt text → alternatif metin 8

alternating / direct current → alternatif akım (AC) / doğru akım (DC) 3

analog → analog 1

AND / OR / NOT / XOR → AND / OR / NOT / XOR 4

animation → animasyon 10

Animation Painter → Animasyon Boyacısı 10

animation pane → animasyon bölmesi 10

annotation (flowchart) → açıklama 13

application → uygulama 1

application / app → uygulama 7

application / transport / internet / link layer → uygulama / taşıma / internet / bağlantı katmanı 2

application programming interface (API) → API (uygulama programlama arayüzü) 11

application software → uygulama yazılımı 6

approximate match → yaklaşık eşleşme 9

architecture → mimari 4

archive / compress → arşiv / sıkıştırma 7

argument → argüman 18

argument (Excel) → bağımsız değişken 9

arithmetic logic unit (ALU) → aritmetik-mantık birimi (ALU) 1

arithmetic operation → aritmetik işlem 1

Armstrong number → Armstrong sayısı 17

array → dizi 19

array initializer → ilklendirici 19

arrow form (->) → ok biçimi 16

artificial intelligence → yapay zekâ 1

artificial intelligence (AI) → yapay zekâ 2

ASCII → ASCII 4

aspect ratio → en-boy oranı 3

assembler → assembler 11

assembly language → assembly 4

assignment → atama 15

assignment statement → atama deyimi 13

asymmetric encryption → asimetrik şifreleme 2

attachment → ek 2

attributes → nitelikler 7

audience → izleyici 10

authentication → kimlik doğrulama 6

authentication / verification → doğrulama 7

authoritative server → yetkili sunucu 2

authorization → yetkilendirme 6

AutoCorrect → otomatik düzeltme 8

AutoFill → otomatik doldurma 9

automation → otomasyon 1
AutoRecover → Otomatik Kurtarma 8
AutoSave → Otomatik Kaydet 8
availability → erişilebilirlik 2
axis → eksen 10

B

backlight → arka ışık 3
backtracking → geri izleme 21
backup → yedekleme 3
backward compatible → geriye uyumlu 3
bandwidth → bant genişliği 3
bar / column chart → çubuk / sütun grafik 10
base (radix) → taban 4
base and limit registers → taban ve limit yazmaçları 6
base case → taban durum 21
best case → en iyi durum 12
bias (IEEE 754 exponent) → sapma 4
bibliography → kaynakça 8
big data → büyük veri 1
Big O notation → büyük O gösterimi 12
binary → ikili 11
binary (number system) → ikili 1
binary digit → ikili basamak (binary digit) 1
binary search → ikili arama 12
biometric → biyometrik 7
BIOS / UEFI → BIOS / UEFI 3
bit → bit 1
bit depth → bit derinliği 1
bit mask → bit maskesi 4
bit shift (<<, >>) → bit kaydırma 4
bitwise operation → bit işlemi 4
block → blok 7
bold / italic / underline → kalın / italik / altı çizili 8
bookmark → yer işareti 8
boolean → boolean (mantıksal tip) 16
boot → önyükleme 6
boot / booting → önyükleme 6
boot loader → önyükleyici 6
boot order → önyükleme sırası 6
bootloader → önyükleyici 6
border → kenarlık 9
border and shading → kenarlık ve gölgelendirme 8
boundary (edge) case → sınır durumu 14
boundary value → sınır değeri 13
branch → dal 13
branch (if/else) → dal 18
branch (instruction) → dallanma 4
branching → dallanma 13
break (statement) → break 17
breakpoint → kesme noktası 11
brightness / contrast → parlaklık / kontrast 3
bubble sort → kabarcık sıralaması 12
buffer → tampon 6
buffer (input buffer) → tampon 20
bulleted list → madde işaretli liste 8
bus (data / address / control) → veri yolu / adres yolu / kontrol yolu 4
byte → bayt 1
byte (unit) → bayt 15
bytecode → bytecode (ara kod) 14
bytecode verifier → bytecode doğrulayıcı 14

C

cache → önbellek 2
cache (memory) → önbellek 3
cache hit / miss → isabet / ıska 3
cache level (L1 / L2 / L3) → önbellek düzeyi 3
cache line (block) → önbellek satırı (blok) 3
Caesar cipher → Sezar şifresi 20
call → çağrı 12
call stack → çağrı yığını 21
call stack / stack frame → çağrı yığını / yığın çerçevesi 21
capacity → kapasite 3
caption → resim yazısı 8
captions / subtitles → altyazı 10
carry → elde 4
case label → case etiketi 16
case-sensitive → büyük-küçük harfe duyarlı 14
cast (explicit conversion) → açık dönüşüm 15
cell → hücre 8
cell (array slot) → hücre 19
cell reference → hücre başvurusu 9
central processing unit (CPU) → merkezî işlem birimi (CPU) 3
certificate → sertifika 2
channel → kanal 2
channel width (Wi-Fi) → kanal genişliği 3
character → karakter 4
character encoding → karakter kodlaması 14
character set → karakter kümesi 1
chart → grafik 10
chip → yonga 3
chipset → yonga seti 3
circular wait → döngüsel bekleme 6
citation → alıntı (atıf) 8
citation style → atıf stili 8
class → sınıf 14
class declaration → sınıf bildirimi 14
class library → sınıf kütüphanesi 14

class loader → sınıf yükleyici 14
client → istemci 1
clipboard → pano 8
clock cycle → saat turu (tur) 3
clock speed → saat hızı 3
closed source / proprietary → kapalı kaynak 6
cloud → bulut 8
cloud computing → bulut bilişim 1
co-authoring → ortak yazarlık 10
code → kod 12
code editor → kod düzenleyici 11
code generation → kod üretimi 11
code optimization → kod iyileştirme 11
code point → kod noktası 4
code tracing → kod izleme 18
code unit → kod birimi 4
code unit (UTF-16) → kod birimi 20
coding → kodlama 11
color accuracy → renk doğruluğu 3
color depth → renk derinliği 1
color scales → renk ölçekleri 9
column → sütun 8
column / bar chart → sütun / çubuk grafik 9
command → komut 8
command bar → komut çubuğu 7
command line → komut satırı 14
command prompt → Komut İstemi 6
command queue → komut kuyruğu 3
command-line argument → komut satırı argümanı 14
command-line interface (CLI) → komut satırı arayüzü (CLI) 6
comment → açıklama 8
commit (version control) → kayıt 11
comparison → karşılaştırma 1
comparison operator → karşılaştırma işleci 15
compatibility → uyumluluk 3
Compatibility Checker → Uyumluluğu Denetle 10
compile → derleme 11
compile / compilation → derleme 11
compile error → derleme hatası 14
compile-time error → derleme hatası 15
compiled language → derlenen dil 11
compiler → derleyici 11
compound assignment → bileşik atama 15
compound condition → bileşik koşul 13
compress (media) → sıkıştırma 10
computability → hesaplanabilirlik 5
computation → hesaplama 5
computational thinking → hesaplamalı düşünme 5
computer → bilgisayar 1
computer case → kasa 3
computer network → bilgisayar ağı 2
computer science → bilgisayar bilimi 5
concatenation (Excel &) → birleştirme 9
concurrency → eş zamanlılık 6
condition → koşul 1
conditional (ternary) operator → koşul işleci 16
conditional flow (selection) → koşullu akış 13
conditional formatting → koşullu biçimlendirme 9
confidentiality → gizlilik 2
connector (flowchart, shape) → bağlayıcı 13
connector (hardware) → konektör 3
console → konsol 14
constant → sabit 9
constant / literal → sabit 15
constant expression → sabit ifade 20
constant folding → sabit katlama 11
constant pool → sabit havuzu 14
constraint → kısıt 12
container → konteyner 6
container engine → konteyner motoru 6
content delivery network (CDN) → içerik dağıtım ağı (CDN) 2
content vs reference comparison → içerik karşılaştırması / referans karşılaştırması 20
context menu → bağlam menüsü 7
context switch → bağlam değiştirme 6
contextual tab → bağlamsal sekme 8
continue → atla 13
contract (method contract) → sözleşme 18
contrast → kontrast 10
control character → kontrol karakteri 4
control flow → akış 16
control structure → kontrol yapısı 16
control unit → kontrol birimi 4
controller → denetleyici 7
controller (memory / keyboard / SSD) → denetleyici 3
cookie → çerez 2
cooling → soğutma 3
Copilot → Copilot 10
copy → kopya 19
copy / cut / paste → kopyala / kes / yapıştır 7
copy-on-write → yazınca kopyalama 6
copyleft → copyleft 6
core (CPU) → çekirdek 3
correctness → doğruluk 12
cost → maliyet 12
counter → sayaç 13
counter-controlled loop → sayaçlı döngü 13

CPI (cycles per instruction) → CPI (komut başına saat turu) 4
CPU → CPU 7
criteria → ölçüt 9
crop → kırpma 8
cross-reference → çapraz başvuru 8
curly braces → süslü parantez 14
current (working) directory → geçerli klasör 6
current folder → geçerli klasör 7
cursor → imleç 3
cursor (text) → imleç 8
custom show → özel gösteri 10

D

dangling else → sarkan else 16
data → veri 1
data bars → veri çubukları 9
data entry → veri girişi 9
data life cycle → veri yaşam döngüsü 1
data series → seri 9
data source → veri kaynağı 8
data store (flowchart symbol) → veri deposu 13
data type → veri tipi 4
data type (Excel) → veri tipi 9
data validation → veri doğrulama 9
database → veri tabanı 1
date serial number → seri değer 9
De Morgan's laws → De Morgan kuralları 16
deadline → son tarih 6
deadlock → kilitlenme 6
debugger → hata ayıklayıcı 11
debugging → hata ayıklama 6
decimal → onluk 4
decimal (base 10) → onluk 17
decimal (floating-point) → kayan noktalı sayı 20
decimal separator → ondalık ayırıcı 15
decision → karar 13
decision structure → karar yapısı 16
decision symbol → karar simgesi 13
declaration → bildirim 14
declaration statement → bildirim deyimi 14
declarative knowledge → bildirimsel bilgi 5
decomposition → ayrıştırma 5
decryption → çözme 2
default app → varsayılan uygulama 7
default gateway → varsayılan ağ geçidi 2
default port → varsayılan port 2
default route → varsayılan yol 2
default value → varsayılan değer 19
definiteness → kesinlik 12
defragmentation → birleştirme 7
delay → gecikme 10
delimiter / separator → ayırıcı 20
dependency → bağımlılık 6
deployment → dağıtım 11
deprecated → kullanımdan kaldırılmış 20
design → tasarım 11
Designer → Tasarım Önerileri 10
desktop → masaüstü 7
desktop / laptop → masaüstü / dizüstü 1
device → aygıt 3
device driver → aygıt sürücüsü 6
digit → basamak 12
digit (symbol) → rakam 17
digit / place value → basamak / basamak değeri 1
digital → dijital 1
digital footprint → dijital ayak izi 2
digital signature → dijital imza 2
DIMM / PCIe / M.2 / SATA / NVMe → DIMM / PCIe / M.2 / SATA / NVMe 3
direct formatting → doğrudan biçimlendirme 8
disk → disk 7
display panel (IPS / VA / TN / OLED / LCD) → panel (IPS / VA / TN / OLED / LCD) 3
divide and conquer → böl ve fethet 21
dividend → bölünen 15
divisor → bölen 12
DNS (Domain Name System) → DNS 2
DNS resolution → DNS çözümlemesi 2
DNS resolver → DNS çözümleyicisi 2
document → belge 8
Document Inspector → Belgeyi İncele 10
domain name → alan adı 2
domain-specific language → alana özgü dil 11
dotted decimal notation → noktalı onluk gösterim 2
DPI → DPI 3
drag and drop → sürükle-bırak 7
drive → sürücü 7
drive letter → sürücü harfi 6
drop-down list → açılır liste 9
duplex printing → çift taraflı yazdırma 8
duration → süre 10
dynamic typing → dinamik tipleme 11

E

e-commerce → elektronik ticaret 2
e-mail → e-posta 2
early exit (loop) → erken çıkış 13
early return → erken dönüş 18
edge case / boundary case → sınır durumu 11

Editor → Düzenleyici 8
effect → efekt 10
efficiency → verimlilik 3
EFI system partition → EFI sistem bölümü 6
element → öğe 19
element (item) → öğe 17
else-if chain → else-if zinciri 16
embed fonts → yazı tipi gömme 10
embedded software → gömülü yazılım 11
embedded system → gömülü sistem 1
empty array → boş dizi 19
empty statement → boş deyim 15
encoding → kodlama 4
encryption → şifreleme 2
end of file (EOF) → dosya sonu 6
end task → görevi sonlandır 7
end-user license agreement (EULA) → son kullanıcı lisans sözleşmesi (EULA) 6
endianness (big / little endian) → bayt sırası (büyük / küçük uçlu) 4
endnote → sonnot 8
enhanced for loop (for-each) → gelişmiş for 19
entrance / emphasis / exit → giriş / vurgu / çıkış 10
entry point → giriş noktası 14
environment variable (PATH, JAVA_HOME) → ortam değişkeni 14
epsilon (tolerance) → eşik 15
error alert → hata uyarısı 9
error message → hata iletisi 14
error message (Excel) → hata iletisi 9
error value → hata değeri 9
error-correcting code (ECC) → hata düzelten kod (ECC) 1
escape sequence → kaçış dizisi 14
Euclid's algorithm → Öklid algoritması 5
exact match → tam eşleşme 9
exception → istisna 14
executable → çalıştırılabilir 7
executable file → çalıştırılabilir dosya 11
execute / run → yürütmek / çalıştırmak 11
execution → yürütme 13
execution model → çalışma modeli 14
expansion card → genişleme kartı 3
exponent → üs 9
exponent / base → üs / taban 17
exponential (growth) → üstel 21
export → dışa aktarma 8
expression → ifade 1
expression tree → ifade ağacı 15
extract all → tümünü ayıkla 7

F

factorial → faktöriyel 12
fairness → adalet 5
fall-through → aşağı düşme 16
FCFS (first come, first served) → FCFS (ilk gelen ilk hizmet alır) 6
feedback → geri bildirim 11
fetch / decode / execute → getir / çöz / yürüt 4
fetch-decode-execute cycle → getir–çöz–yürüt döngüsü 4
Fibonacci sequence → Fibonacci dizisi 17
field → alan 4
FIFO (first in, first out) → FIFO (ilk giren ilk çıkar) 6
file → dosya 3
file association → ilişkilendirme 7
file descriptor → dosya tanımlayıcısı 6
File Explorer → Dosya Gezgini 7
file extension → uzantı 7
file format → dosya biçimi 4
file name extension → uzantı 7
file system → dosya sistemi 6
file type → dosya türü 7
fill (color) → dolgu 9
fill handle → doldurma tutamacı 9
filter → filtre 9
filtering → filtreleme 9
Find and Replace → Bul ve Değiştir 8
finiteness → sonluluk 12
firewall → güvenlik duvarı 2
firmware → ürün yazılımı 3
first-line indent → ilk satır girintisi 8
fixed program → sabit program 5
flag (variable) → bayrak 13
Flash Fill → Hızlı Doldurma 9
flash memory → flash bellek 3
floating point → kayan nokta 4
floating-point / decimal number → kayan noktalı sayı 15
floating-point number → kayan noktalı sayı 15
flow → akış 13
flow line / arrow → ok 13
flowchart → akış diyagramı 13
folder → klasör 7
folder (directory) → klasör 6
font → yazı tipi 8
font size / point → punto (pt) 8
footnote → dipnot 8
form factor → form faktörü 3
formal parameter → biçimsel parametre 18
format (a disk) → biçimlendirme 6
Format Page Numbers → Sayfa Numaralarını Biçimlendir 8

Format Painter → Biçim Boyacısı 8
format specifier → biçim belirteci 15
format string → biçim dizgisi 15
formatted output → biçimli çıktı 15
formatting → biçimlendirme 8
formatting (printf) → biçimleme 20
formatting marks → biçimlendirme işaretleri 8
formula → formül 9
formula bar → formül çubuğu 9
fragment → parça 2
fragmented → parçalanmış 7
frame → kare 1
frame (page frame) → çerçeve 6
frames per second (fps) → fps (saniyedeki kare sayısı) 1
freeze panes → dondurma (Bölmeleri Dondur) 9
frequency → frekans 19
frequency band → bant 2
full adder → tam toplayıcı 4
function → fonksiyon 11
function (Excel) → işlev 9
function call → çağrı 18
function keys → işlev tuşları 3
functional programming → fonksiyonel programlama 11
functionality → işlevsellik 6

G

game controller → oyun denetleyicisi 3
Gantt chart → Gantt çizelgesi 6
garbage collection → çöp toplama 14
garbage collector → çöp toplayıcı 20
general purpose → genel amaçlı 5
general-purpose language → genel amaçlı dil 11
GPL (GNU General Public License) → GPL 6
GPT (GUID Partition Table) → GPT (genel benzersiz kimlikli bölüm tablosu) 6
graphical user interface (GUI) → grafik kullanıcı arayüzü (GUI) 6
graphics card → ekran kartı 3
graphics processing unit (GPU) → grafik işlem birimi (GPU) 3
greatest common divisor (GCD) → en büyük ortak bölen (EBOB) 18
grid / cell → ızgara / hücre 17
gridlines / guides → ızgara / kılavuz 10
group → grup 10
guest operating system → misafir işletim sistemi 6

H

half adder → yarım toplayıcı 4
halting problem → durma problemi 5
hand tracing (desk check) → elle izleme 12
handle (Windows) → tanıtıcı 6
handouts → dinleyici notları 10
hanging indent → asılı girinti 8
haptic feedback → dokunsal geri bildirim 3
hard disk drive (HDD) → sabit disk (HDD) 3
hardware → donanım 3
hardware security key → donanım anahtarı 2
Harvard architecture → Harvard mimarisi 4
hash (cryptographic) → özet (hash) 2
HDMI / DisplayPort / VGA → HDMI / DisplayPort / VGA 3
header → başlık 2
header / footer → üstbilgi / altbilgi 8
header row → başlık satırı 8
heading → başlık 8
heap → heap 16
heap (memory) → heap 19
helper method → yardımcı metot 21
hexadecimal → onaltılık 4
hidden → gizli 7
hidden slide → gizli slayt 10
high-level / low-level language → yüksek / düşük düzeyli dil 11
high-level language → yüksek düzeyli dil 4
histogram → histogram 9
hold and wait → tut ve bekle 6
hop → atlama 2
host operating system → ana işletim sistemi 6
HTML / CSS / JavaScript → HTML / CSS / JavaScript 2
HTTP / HTTPS → HTTP / HTTPS 2
hybrid → hibrit 11
hybrid kernel → hibrit çekirdek 6
hyperlink → köprü (bağlantı) 8
hypervisor → hipervizör 6
hyphenation → heceleme 8

I

I/O (input/output) → G/Ç 6
IaaS / PaaS / SaaS → IaaS / PaaS / SaaS 2
icon → simge 7
icon sets → simge kümeleri 9
IDE (integrated development environment) → IDE 11
identifier → tanımlayıcı 14
identity element → etkisiz eleman 17
IEEE 754 → IEEE 754 4
if-else statement → if-else yapısı 16
immutability → değiştirilemezlik 20
immutable → değiştirilemez 20
imperative knowledge → buyurgan bilgi 5
imperative programming → buyurgan yaklaşım 11
implementation → gerçekleme 11

implicit conversion → örtük dönüşüm 15
in line with text → satır içi 8
in-place → yerinde 19
increment / decrement → artırma / azaltma 15
indent → girinti 8
indentation → girinti 12
index (array) → indis 19
index (search engine) → dizin 2
index (Word) → dizin 8
indexing → indisleme 20
infinite loop → sonsuz döngü 17
infinite recursion → sonsuz özyineleme 21
information → bilgi 1
information security (CIA) → bilgi güvenliği 2
information technology (IT) → bilgi teknolojisi (BT) 1
initial value → başlangıç değeri 13
initialization (initial value) → ilk değer verme 15
initialization (loop) → ilk değer verme 17
inkjet / laser printer → mürekkep püskürtmeli / lazer yazıcı 3
inner loop → iç döngü 13
inner loop / outer loop → iç döngü / dış döngü 17
input (data) → girdi 11
input device / input unit → giriş birimi 3
input message → girdi iletisi 9
input size → girdi boyutu 12
input unit → giriş birimi 3
input validation → girdi doğrulama 17
input/output symbol → giriş/çıkış simgesi 13
insertion sort → eklemeli sıralama 12
install / uninstall → kurulum / kaldırma 7
instant messaging → anlık mesajlaşma 2
instruction → komut 4
instruction / command → komut 14
instruction format → komut biçimi 4
instruction register (IR) → komut yazmacı (IR) 4
instruction set → komut kümesi 3
integer → tam sayı 4
integer division → tam sayı bölmesi 15
integrated / discrete GPU → entegre / bağımsız GPU 3
integrated development environment (IDE) → tümleşik geliştirme ortamı (IDE) 6
integrity → bütünlük 2
interface → arayüz 3
interference → girişim 2
intermediate code → ara kod 11
Internet → internet 2
Internet of Things (IoT) → nesnelerin interneti 1
Internet service provider (ISP) → internet servis sağlayıcısı (İSS) 2
interpret → yorumlama 11
interpreted → yorumlanan 11
interpreter → yorumlayıcı 11
interrupt → kesme 6
invalid input → geçersiz girdi 12
inverted index → ters dizin 2
IP (Internet Protocol) → IP (internet protokolü) 2
IP address → IP adresi 2
IPv4 / IPv6 → IPv4 / IPv6 2
iteration → tur 17
iteration (one pass of a loop) → tur 12
iteration (one pass) → tur 17
iterative → yinelemeli 11
iterative (version) → döngülü 21

J

jagged array → düzensiz dizi 19
Java Development Kit (JDK) → JDK (geliştirme kiti) 14
Java Runtime Environment (JRE) → JRE (çalışma ortamı) 14
Java Virtual Machine (JVM) → Java Sanal Makinesi (JVM) 14
Javadoc / API documentation → Javadoc (API belgesi) 14
JIT (just-in-time) compiler → JIT (tam zamanında) derleyici 11
journaling → günlükleme 6
just-in-time compilation (JIT) → JIT 14
justify → iki yana yaslama 8

K

kernel → çekirdek 6
kernel (OS) → işletim sistemi çekirdeği 6
kernel mode → çekirdek modu 6
key (insertion sort) → anahtar 12
key agreement (key exchange) → anahtar anlaşması 2
keyboard → klavye 3
keyboard layout → klavye düzeni 7
keyboard shortcut → klavye kısayolu 7
keyword → anahtar sözcük 14
kibibyte / mebibyte / gibibyte → KiB / MiB / GiB 1
kilobyte / megabyte / gigabyte / terabyte → KB / MB / GB / TB 1
kiosk (browsed at a kiosk) → bilgi noktası (kiosk) 10

L

labeled break → etiketli break 17
lane (PCIe) → şerit 3
laser pointer → lazer işaretçi 10
latency → gecikme 3
layer → katman 2
least common multiple (LCM) → EKOK 18
length → uzunluk 19
level → düzey 8

lexical analysis → sözcük çözümleme 11
lexicographic order → sözlüksel sıra 20
library → kütüphane 6
license → lisans 6
LIFO (last in, first out) → LIFO (son giren ilk çıkar) 21
line (of code/output) → satır 13
line chart → çizgi grafik 10
line number → satır numarası 14
line spacing → satır aralığı 8
linear recursion → doğrusal özyineleme 21
linear search → doğrusal arama 12
linked / embedded object → bağlı / gömülü nesne 10
linker → bağlayıcı 11
linking error → bağlama hatası 11
Linux distribution → Linux dağıtımı 6
liquid crystal → sıvı kristal 3
list → liste 19
literal → sabit değer 14
literal / constant → sabit 20
loader → yükleyici 11
local account → yerel hesap 7
local area network (LAN) → yerel ağ (LAN) 2
local variable → yerel değişken 15
locale → yerel ayar 15
lock (screen) → kilit 7
log (journal) → günlük 6
logging → kayıt tutma 6
logic error → mantık hatası 14
logic gate → mantık kapısı 4
logical address → mantıksal adres 6
logical operation → mantıksal işlem 1
logical operator → mantıksal işleç 16
logical test → mantıksal sınama 9
logical value (TRUE/FALSE) → mantıksal değer (DOĞRU/YANLIŞ) 9
long-term support (LTS) → uzun süreli destek (LTS) 14
longest prefix match → en uzun önek eşleşmesi 2
loop → döngü 13
loop / cycle → döngü 1
loop body → döngü gövdesi 13
loop invariant → döngü değişmezi 12
loop variable → döngü değişkeni 17
lossless compression → kayıpsız sıkıştırma 1
lossy compression → kayıplı sıkıştırma 1
low-level → düşük düzeyli 3
LRU (least recently used) → LRU (en uzun süredir kullanılmayan) 6

M

MAC address → MAC adresi 2
machine code → makine kodu 11
machine instruction → makine komutu 4
machine language → makine dili 11
macro → makro 8
magic number → sihirli sayı 11
magnifier → Büyüteç 7
mail merge → adres mektup birleştirme 8
mail server → posta sunucusu 2
mailbox → posta kutusu 2
main document → ana belge 8
main memory → ana bellek 3
main method → main metodu 14
mainframe → ana bilgisayar 1
maintainability → bakım kolaylığı 6
maintenance → bakım 11
malware → kötü amaçlı yazılım 2
margin → kenar boşluğu 8
markup → işaretleme 8
markup language → işaretleme dili 11
matrix → matris 19
MBR (Master Boot Record) → MBR (ana önyükleme kaydı) 6
median → ortanca 19
memoization → bellekleme 21
memory → bellek 4
memory address → bellek adresi 4
memory allocation → ayırma 6
memory cell → bellek hücresi 4
memory hierarchy → bellek hiyerarşisi 3
memory management → bellek yönetimi 6
memory management unit (MMU) → bellek yönetim birimi (MMU) 6
memory protection → koruma 6
menu → menü 17
merge (version control) → birleştirme 11
merge cells → hücre birleştirme 8
merge conflict → çakışma 11
merge field → birleştirme alanı 8
merge sort → birleştirme sıralaması 12
message → ileti 2
metadata → meta veri 6
method → metot 18
method call → çağrı 18
method overloading → aşırı yükleme 18
method signature → imza 18
metropolitan area network (MAN) → metropol alan ağı (MAN) 2
microcontroller → mikrodenetleyici 1
microkernel → mikro çekirdek 6
middleware → ara yazılım (middleware) 6

minimize / maximize / restore down / close → küçült / büyüt / geri yükle / kapat 7
mixed reference → karma başvuru 9
mobile device → mobil cihaz 1
modem → modem 2
modified Harvard architecture → değiştirilmiş Harvard mimarisi 4
modulo (mod) → mod 5
monitor / screen → monitör / ekran 3
monolithic kernel → monolitik çekirdek 6
Morph → Dönüşüm 10
motherboard → anakart 3
motion path → hareket yolu 10
mouse → fare 3
move → taşıma 7
multi-factor authentication → çok faktörlü doğrulama 7
multi-factor authentication (MFA) → çok faktörlü doğrulama 2
multi-level sort → çok düzeyli sıralama 9
multi-way decision → çok yollu karar 13
multidimensional array → çok boyutlu dizi 19
multifunction device → çok işlevli aygıt 3
multilevel list → çok düzeyli liste 8
multiprogramming → çoklu programlama 6
mutable → değiştirilebilir 20
mutual exclusion → karşılıklı dışlama 6

N

name box → ad kutusu 9
named range → adlandırılmış aralık 9
naming convention → adlandırma kuralı 15
NaN (Not a Number) → NaN (sayı değil) 15
narrowing conversion → daraltma 15
NAT (network address translation) → NAT 2
navigation pane → gezinti bölmesi 8
negation → olumsuzlama 9
nested → iç içe 13
nested decision → iç içe karar 16
nested loop → iç içe döngü 13
network → ağ 3
network / cloud storage → ağ depolama / bulut depolama 3
network byte order → ağ bayt sırası 4
network interface card → ağ kartı 3
network interface card (NIC) → ağ kartı (NIC) 3
neural processing unit (NPU) → sinirsel işlem birimi (NPU) 3
newline → satır sonu 15
newline / line terminator → satır sonu (yeni satır) 14
no preemption → ön alınamazlık 6
non-breaking space → bölünmez boşluk 8
non-volatile / persistent → kalıcı 3
not responding → yanıt vermiyor 7
notification center → bildirim merkezi 7
null reference → null 19
number format → sayı biçimi 9
numbered list → numaralı liste 8

O

object → nesne 14
object file → nesne dosyası 11
object-oriented → nesne yönelimli 11
octal → sekizli 4
octet → sekizli 2
off-by-one error → sınır hatası 17
off-page connector → sayfa dışı bağlayıcı 13
offset → bağıl konum 6
on-premises → yerinde 2
one's complement → birler tümleyeni 4
online → çevrim içi 2
opcode → opcode (işlem kodu) 4
open source → açık kaynak 6
operand → işlenen 15
operand stack → işlenen yığını 14
operating system → işletim sistemi 1
operating system (OS) → işletim sistemi 6
operator → işleç 15
operator (Excel) → işleç 9
operator precedence → işleç önceliği 9
optical disc / external disk → optik disk / harici disk 3
optimize (drives) → iyileştirme 7
orientation → yönlendirme (sayfa yönü) 8
out of bounds (index) → sınır dışı 19
outer loop → dış döngü 13
outline → anahat 10
output (data) → çıktı 11
output device / output unit → çıkış birimi 3
output stream → çıktı akışı 14
output unit → çıkış birimi 3
overflow → taşma 4
overflow (integer) → taşma 21

P

package (Java) → paket 14
package for CD (PowerPoint) → paketleme 10
package manager → paket yöneticisi 11
packet → paket 2
packet switching → paket anahtarlama 2
page → sayfa 6
page break → sayfa sonu 8
page fault → sayfa hatası 6
page file → sayfa dosyası 6

page number → sayfa numarası 8
page replacement → sayfa yer değiştirme 6
page table → sayfa tablosu 6
paging → sayfalama 6
palindrome → palindrom 12
paragraph → paragraf 8
paragraph spacing → paragraf aralığı 8
parallelism → paralellik 4
parameter → parametre 18
parent directory → üst klasör 7
parity bit → parite biti 1
parsing / syntax analysis → söz dizimi çözümleme 11
partition → bölüm 6
partition / volume → bölüm (birim) 7
partitioning scheme → bölümleme düzeni 6
pass (sorting) → geçiş 12
pass by value → değer ile aktarım 18
passkey → geçiş anahtarı 2
password → parola 2
password manager → parola yöneticisi 2
path → yol 6
pattern (coding) → kalıp 17
pattern (output) → desen 17
pattern recognition → örüntü tanıma 5
perfect number → mükemmel sayı 17
performance → performans 3
peripheral → çevre birimi 6
permanent delete → kalıcı silme 7
permission → izin 6
personal area network (PAN) → kişisel alan ağı (PAN) 2
personal computer → kişisel bilgisayar 1
personal data → kişisel veri 2
phishing → oltalama 2
physical address → fiziksel adres 6
picture / image → resim 8
pie chart → pasta grafik 10
PIN → PIN 7
PivotTable → Özet Tablo 9
pixel → piksel 1
pixel density (PPI) → piksel yoğunluğu (PPI) 3
placeholder → yer tutucu 10
plain text → düz metin 14
plaintext → düz metin 2
platform → platform 14
platter → plaka 3
plug and play → tak ve çalıştır 6
pointer (mouse / laser) → işaretçi 10
pointer (two-pointer technique) → işaretçi 19
pointing device → işaretleme aygıtı 3
polling rate → yoklama hızı 3
port → port 3
portable → taşınabilir 11
portrait / landscape → dikey / yatay 8
positional number system → konumsal sayı sistemi 4
POST (power-on self-test) → açılışta öz sınama (POST) 6
post-test loop (do-while) → son koşullu döngü 13
postfix / suffix → sonek 15
power supply unit (PSU) → güç kaynağı (PSU) 3
power-on self-test (POST) → açılışta öz sınama (POST) 6
pre-test loop (while) → ön koşullu döngü 13
precision → hassasiyet 4
predefined process → önceden tanımlı işlem 13
prefix → önek 15
preparation (flowchart symbol) → hazırlık 13
presentation (act of presenting) → sunum 10
presentation (file) → sunu 10
presenter → sunucu 10
Presenter View → Sunucu Görünümü 10
preview → önizleme 8
prime number → asal sayı 12
primitive type → ilkel tip 15
principle of least privilege → en az ayrıcalık 6
print area → yazdırma alanı 9
print preview → baskı önizleme 8
printer → yazıcı 3
privacy → mahremiyet 1
private (local) IP address → yerel IP adresi 2
private key → özel anahtar 2
privileged → ayrıcalıklı 6
problem definition → problem tanımı 12
problem solving → problem çözme 12
process → süreç 6
process ID (PID) → süreç kimliği (PID) 6
process state → süreç durumu 6
process symbol → işlem simgesi 13
processing (IPOS stage) → işleme 1
processor → işlemci 3
processor (CPU) → işlemci 6
program → program 6
program counter (PC) → program sayacı (PC) 4
programming → programlama 11
programming language → programlama dili 11
programming paradigm → programlama paradigması 11
projector → projektör 3
prompt → istem 15
properties → özellikler 7
protocol → protokol 3
prototyping → prototipleme 11

pseudocode → sözde kod 12
public IP address → genel IP adresi 2
public key → açık anahtar 2
pure method/function → saf metot 18

Q

quantization → nicemleme 1
quantum computer → kuantum bilgisayar 1
qubit → qubit (kuantum biti) 1
query → sorgu 1
quick settings → hızlı ayarlar 7
quotient → bölüm 15

R

RAM → RAM 6
random → rastgele 12
random access → rastgele erişim 12
random access memory (RAM) → RAM (rastgele erişimli bellek) 3
random number → rastgele sayı 17
range (Excel) → aralık 9
range (wireless) → menzil 2
ransomware → fidye yazılımı 2
read-only → salt okunur 7
read-only memory (ROM) → ROM 3
read-write head → okuma-yazma kafası 3
read/write head → okuma/yazma başlığı 5
readability → okunabilirlik 13
reading order → okuma sırası 10
ready queue → hazır kuyruğu 6
real-time protection → gerçek zamanlı koruma 7
real-time system → gerçek zamanlı sistem 6
recipient → alıcı 8
record → kayıt 4
record (DNS record) → kayıt 2
recovery → kurtarma 7
recovery key → kurtarma anahtarı 7
recursion → özyineleme 21
recursion depth → özyineleme derinliği 21
recursion tree / call tree → özyineleme ağacı 21
recursive case (step) → özyinelemeli durum 21
Recycle Bin → Geri Dönüşüm Kutusu 7
reference → referans 15
refresh rate → yenileme hızı 3
register → yazmaç 4
regular expression → düzenli ifade 20
rehearse → prova 10
Rehearse Timings → Zamanlamaları Prova Et 10
relative path → göreli yol 7
relative reference → göreli başvuru 9
reliability → güvenilirlik 6
remainder → kalan 5
remainder (modulo) → kalan 12
rename → yeniden adlandırma 7
repetition → tekrar 13
repetition (iteration) structure → döngü yapısı 16
repetitive flow (iteration) → tekrarlı akış 13
Replace Fonts → Yazı Tiplerini Değiştir 10
request → istek 2
request / response → istek / yanıt 1
requirement → gereksinim 11
reserved name → ayrılmış ad 7
reset → sıfırlama 7
resolution → çözünürlük 3
resource → kaynak 6
response → yanıt 2
response time → tepki süresi 3
restore → geri yükleme 7
restore point → geri yükleme noktası 7
return → döndürmek 18
return (a value) → döndürmek 18
return (value) → döndürmek 18
return type → dönüş tipi 18
return value → dönüş değeri 18
reuse → yeniden kullanım 18
reversing (an array/string) → ters çevirme 19
RGB (color model) → RGB 1
ribbon → şerit 8
ripple-carry adder → dalgalı elde toplayıcı 4
RISC / CISC → RISC / CISC 4
ROM / flash memory → ROM / flash bellek 6
root (directory) → kök 6
root (folder) → kök 7
root server → kök sunucu 2
rotational latency → dönme gecikmesi 3
round robin → döner sıralı çizelgeleme (round robin) 6
rounding → yuvarlama 4
router → yönlendirici 3
routing table → yönlendirme tablosu 2
row → satır 8
ruler → cetvel 8
run → çalıştırma 13
run (run-length encoding) → koşu 1
run-length encoding (RLE) → çalışma uzunluğu kodlaması (RLE) 1
runtime → çalışma zamanı 6
runtime error → çalışma zamanı hatası 14

S

sampling / sample → örnekleme / örnek 1
sampling rate → örnekleme hızı 1

sandbox → korumalı alan 6
save / save as → kaydet / farklı kaydet 8
scale → ölçek 7
scanner → tarayıcı 3
scatter chart → dağılım (XY) grafiği 9
scheduler → çizelgeleyici 6
scheduling → çizelgeleme 6
scope → kapsam 15
screen → ekran 13
screen reader → ekran okuyucu 7
screen resolution → çözünürlük 7
screen size → ekran boyutu 3
screenshot → ekran görüntüsü 7
scripting language → betik dili 11
scroll bar → kaydırma çubuğu 7
search → arama 12
search engine → arama motoru 2
search operator → operatör 2
search range (interval) → arama aralığı 12
section → bölüm 8
section break → bölüm sonu 8
sector → sektör 3
Secure Boot → Güvenli Önyükleme 6
security → güvenlik 1
seek time → arama süresi 3
segment (TCP) → bölüt 2
segmentation fault (access violation) → erişim ihlali 6
selection → seçim 8
selection (conditional) → koşullu yapı 16
selection (decision) structure → karar yapısı 16
selection pane → seçim bölmesi 10
selection sort → seçimli sıralama 12
semantic analysis → anlam çözümleme 11
semantics → anlam 11
semicolon → noktalı virgül 14
sensor → sensör 1
sentinel value → bekçi değer 17
sentinel-controlled loop → bekçi değerli döngü 13
separator → ayırıcı 14
sequence (sequential structure) → sıralı yapı 16
sequence number → sıra numarası 2
sequence structure → sıralı yapı 16
sequential flow (sequence) → sıralı akış 13
sequential read → sıralı okuma 3
server → sunucu 1
session key → oturum anahtarı 2
settings → ayarlar 7
shadowing → gölgeleme 18
shallow copy → sığ kopya 19
shape → şekil 10
sheet protection → sayfa koruması 9
sheet tab → sayfa sekmesi 9
short-circuit evaluation → kısa devre değerlendirme 15
shortcut (file, .lnk) → kısayol 7
Show Formulas → Formülleri Göster 9
side effect → yan etki 18
Sieve of Eratosthenes → Eratosthenes eleği 19
sign / exponent / fraction (IEEE 754) → işaret / üs / kesir 4
sign bit → işaret biti 15
sign in / sign out → oturum açma / oturumu kapatma 7
signed → işaretli 4
signed overflow → işaretli taşma 4
simultaneous → eş zamanlı 8
SJF (shortest job first) → SJF (en kısa iş önce) 6
slide → slayt 10
slide layout → slayt düzeni 10
slide master → Asıl Slayt 10
slide pane / notes pane → slayt bölmesi / notlar bölmesi 10
slide show → slayt gösterisi 10
slide size → slayt boyutu 10
slide sorter → Slayt Sıralayıcısı 10
sliding window → kayan pencere 12
slot → yuva 3
SmartArt → SmartArt 10
SMTP / IMAP → SMTP / IMAP 2
snap → yaslama 7
Snipping Tool → Ekran Alıntısı Aracı 7
social engineering → sosyal mühendislik 2
socket (CPU) → soket 3
software → yazılım 1
software development life cycle → yazılım geliştirme süreci 11
solid-state drive (SSD) → SSD 3
sort → sıralama 9
sorted → sıralı 12
sorting → sıralama 12
sound card → ses kartı 3
source code → kaynak kod 11
source file → kaynak dosya 14
space complexity → alan karmaşıklığı 12
spaghetti code → spagetti akış 13
spatial locality → uzamsal yerellik 3
speaker / headphones → hoparlör / kulaklık 3
speaker notes → konuşmacı notları 10
spelling and grammar → yazım ve dilbilgisi 8
spill → taşma 9
spreadsheet → elektronik tablo 9
spyware → casus yazılım 2

square root → karekök 12
SSD / HDD → SSD / HDD 7
stack (memory) → yığın 18
stack frame → yığın çerçevesi 18
stack machine → yığın makinesi 14
stack overflow → yığın taşması 21
stack trace → yığın izi 14
standard account → standart hesap 6
standard user → standart kullanıcı 7
Start menu → Başlat menüsü 7
startup apps → başlangıç uygulamaları 7
state → durum 5
statement → deyim 14
static method → static metot 18
static typing → statik tipleme 11
status bar → durum çubuğu 7
status code → durum kodu 2
step → adım 12
storage → depolama 3
Storage Sense → Depolama Algılaması 7
stored program → depolanan program 4
string → dizgi 4
string concatenation → dizgi birleştirme 20
string literal → dizgi sabiti 20
string pool → dizgi havuzu 20
structured programming → yapısal programlama 13
structured reference → yapılandırılmış başvuru 9
style → stil 8
subdomain → alt alan adı 2
subfolder → alt klasör 6
subprocess → alt süreç 13
subscription → abonelik 6
subset → alt küme 21
substring → alt dizgi 20
SUBTOTAL → ALTTOPLAM 9
Summary / Section / Slide Zoom → Özet / Bölüm / Slayt Yakınlaştırma 10
supercomputer → süper bilgisayar 1
superscript / subscript → üst simge / alt simge 8
sustainability → sürdürülebilirlik 5
swap → takas 12
swap space → takas alanı 6
swapping → takas 6
swapping / swap space → takas / takas alanı 6
switch → switch 2
switch (network) → switch (anahtar) 3
switch expression → switch ifadesi 16
switch statement → switch deyimi 16
symbol → simge 5
symmetric encryption → simetrik şifreleme 2
syntax → söz dizimi 11
syntax error → söz dizimi hatası 14
syntax tree → söz dizimi ağacı 11
system call → sistem çağrısı 6
system software → sistem yazılımı 6
system tray / notification area → sistem tepsisi 7
System-on-Chip (SoC) → SoC (yonga üstü sistem) 3

T

tab (character) → sekme 14
tab (ribbon / Explorer / tab stops) → sekme 8
tab stop → sekme durağı 14
table → tablo 8
table (Excel table) → tablo 9
table of contents → içindekiler (tablosu) 8
table of figures → şekil tablosu 8
tail call optimization → kuyruk çağrısı eniyilemesi 21
tail recursion → kuyruk özyinelemesi 21
tape → bant 5
target → hedef 12
target audience → hedef kitle 10
Task Manager → Görev Yöneticisi 7
task view → görev görünümü 7
taskbar → görev çubuğu 7
TCP / UDP → TCP / UDP 2
template → şablon 8
temporal locality → zamansal yerellik 3
temporary file → geçici dosya 7
temporary variable → geçici değişken 12
terminal (start/end) symbol → başla/bitir simgesi 13
test → test 12
test case → test durumu 12
testing → test 5
text → metin 9
text (string) → metin 12
text box → metin kutusu 10
text editor → metin editörü 6
text file → metin dosyası 11
text wrapping → metin kaydırma 8
theme → tema 8
thermal throttling → ısıl kısma 3
thesaurus → eş anlamlılar 8
thin client → ince istemci 1
thread → iş parçacığı 3
three-way handshake → üçlü el sıkışma 2
throw → fırlatma 15
throw (an exception) → fırlatma 14
throw (exception) → fırlatma 21
thumbnail → küçük resim 10

time complexity → zaman karmaşıklığı 12
time complexity / Big O → zaman karmaşıklığı 21
time sharing → zaman paylaşımı 6
time slice (quantum) → zaman dilimi 6
timer → zamanlayıcı 10
timing → zamanlama 10
title bar → başlık çubuğu 7
TLS → TLS 2
token → belirteç 14
toolbar → araç çubuğu 7
top-level domain (TLD) → üst düzey alan 2
total row → toplam satırı 9
touch interface → dokunmatik arayüz 6
touchpad → dokunmatik yüzey 3
touchscreen → dokunmatik ekran 3
Towers of Hanoi → Hanoi kuleleri 21
TPM → TPM 6
trace (code tracing) → kod izleme 17
trace / tracing → izleme 12
trace table → izleme tablosu 12
traceroute → tracert / traceroute 2
track → iz 3
track changes → değişiklik izleme 8
transfer (data) → aktarım 3
transistor → transistör 1
transition → geçiş 10
transition rule → geçiş kuralı 5
translation → çeviri 11
translator (compiler/interpreter) → çeviri aracı 11
traversal (iterate over array) → gezinme 19
trigger → tetikleyici 10
TRIM → TRIM 7
Trojan horse → Truva atı 2
truncation → kesme 15
truth table → doğruluk tablosu 4
TTL (time to live) → TTL 2
tunnel → tünel 2
Turing machine → Turing makinesi 5
turnaround time → tamamlanma süresi 6
two's complement → ikiye tümleyen 4
two-dimensional array → iki boyutlu dizi 19
type → tip 15
type conversion → tip dönüşümü 15
type error → tip hatası 14
type promotion → tip yükseltme 15
type system → tip sistemi 14

U

UEFI / BIOS → UEFI / BIOS 6
unauthorized access → yetkisiz erişim 1
Unicode → Unicode 4
Unicode escape → Unicode kaçışı 14
universal Turing machine → evrensel Turing makinesi 5
unsigned → işaretsiz 4
update → güncelleme 6
uppercase / lowercase → büyük harf / küçük harf 20
URL → URL 2
usability → kullanılabilirlik 6
USB → USB 3
USB flash drive → USB bellek 3
user account → kullanıcı hesabı 7
User Account Control (UAC) → Kullanıcı Hesabı Denetimi (UAC) 7
user interface → arayüz 6
user mode → kullanıcı modu 6
UTF-8 → UTF-8 4
utility → yardımcı araç 6

V

varargs (variable-length arguments) → değişken sayıda argüman 18
variable → değişken 4
VBA → VBA 8
verification / validation → doğrulama 12
version → sürüm 14
version control → sürüm kontrolü 11
version history → sürüm geçmişi 8
video memory (VRAM) → video belleği (VRAM) 3
view → görünüm 8
viewing angle → görüş açısı 3
virtual address → sanal adres 6
virtual desktop → sanal masaüstü 7
virtual machine → sanal makine 14
virtual machine (VM) → sanal makine 6
virtual memory → sanal bellek 6
virtualization → sanallaştırma 6
virus → virüs 2
visual hierarchy → görsel hiyerarşi 10
volatile / temporary → geçici 3
von Neumann architecture → von Neumann mimarisi 4
von Neumann bottleneck → von Neumann darboğazı 4
VPN → VPN 2

W

waiting time → bekleme süresi 6
waterfall model → şelale modeli 11
watermark → filigran 8
web browser → web tarayıcısı 2
web crawler → web robotu 2
whitespace → boşluk 20
Wi-Fi → Wi-Fi 2

wide area network (WAN) → geniş alan ağı (WAN) 2
widening conversion → genişletme 15
widescreen → geniş ekran 10
wildcard → joker karakter 7
window → pencere 7
word → sözcük 8
word count → sözcük sayısı 8
word processor → kelime işlemci 6
workbook → çalışma kitabı 9
workload → iş yükü 3
worksheet → çalışma sayfası 9
World Wide Web → Web 2
worm → solucan 2
worst case → en kötü durum 12
wrap text → metni kaydır 9
write once, run anywhere → bir kez yaz, her yerde çalıştır 14
WYSIWYG (what you see is what you get) → WYSIWYG (ne görürsen onu alırsın) 8

Z

zoom → yakınlaştırma 8

EK B

Bilgisayar ve Programlamaya Giriş İçin Ön Bilgiler

Bilgisayar ve Programlamaya Giriş · Ön Bilgiler

Ders, programlama deneyimi beklemez. Bu sayfa 12 bölümdeki 21 sunumun dayandığı günlük bilgisayar becerilerini ve okul matematiğini toplar. Her konunun kartında ne bilinmesi gerektiği, hangi sunumlarda kullanıldığı ve iki kısa sınama sorusu yer alır. Ayrıntılar **Ayrıntılar ve sınama** düğmesiyle açılır, sorunun üstüne tıklanınca cevabı görünür. Her sunumun bir de çalışma soruları sayfası vardır. Bu sayfalara en alttaki sunum haritasından ulaşılır.

En gerekli çekirdek. Doğru yazmak ve dosyaları bulabilmek, dört işlem ile tam sayı bölümü ve kalanı (`/` ve `%`), yüzde ve ortalama, sayıların yerine harf kullanmak, ve/veya/değil, adımları sırasıyla izlemek. Java'nın kendisi Bölüm 5'te sıfırdan anlatılır. Buradaki bir konu eksikse ilk haftalarda kısa bir tekrar yeterlidir.

Terimler. TERİM satırı slaytlardaki Türkçe terimleri ve İngilizce karşılıklarını verir. Tam liste ders sözlüğündedir.

Bilgisayar kullanımının temelleri

Klavye ve fare, dikkatli yazma, dosyalar ve klasörler, web ve e-posta, yazılım kurma. Bölüm 1–3 bunların üzerine kurulur, Java laboratuvarları da bunları bilinen sayar.

Klavye, fare ve kısayollar

Tıklama · seçme · Ctrl kısayolları

TERİM ***klavye kısayolu*** (*keyboard shortcut*); ***kopyala / kes / yapıştır*** (*copy / cut / paste*); ***sürükle-bırak*** (*drag and drop*)

Seçme, kopyalama, taşıma ve geri alma işleri klavyeyle hızlıca yapılabilmelidir. Windows ve Office sunumları bunlara ek olarak daha birçok kısayol tanıtır.

BÖLÜMLER **Bölüm 1.2 · Bilgisayar Donanımı** **Bölüm 2.2 · Windows İşletim Sistemi** **Bölüm 3.1 · Microsoft Word** **Bölüm 3.2 · Microsoft Excel** **Bölüm 3.3 · Microsoft PowerPoint**

- Fare: tek tıklama seçer, çift tıklama açar, sağ tıklama bağlam menüsünü (*context menu*) açar. Sürükleme öğeyi taşır, Ctrl basılı tutulursa kopyalar.
- Metin seçme: Shift ile ok tuşları seçimi genişletir. Ctrl + A her şeyi, çift tıklama tek bir sözcüğü seçer.
- Temel kısayollar: Ctrl + C (kopyala), Ctrl + X (kes), Ctrl + V (yapıştır), Ctrl + Z (geri al), Ctrl + Y (yinele), Ctrl + S (kaydet), Ctrl + F (bul).
- Bölüm 2'de kullanılan pencere ve sistem kısayolları: Win + E (Dosya Gezgini), Win + D (masaüstünü göster), Alt + Tab (pencereler arasında geçiş), Win + Shift + S (ekranın bir bölgesinin görüntüsünü alma).

KENDİNİ SINAMA

▶ **Seçili bir paragrafta Ctrl + X'e, ardından başka bir yerde Ctrl + V'ye basılıyor. Paragrafa ne olur?**

▶ **Bir cümle yanlışlıkla silindi. Hangi kısayol onu geri getirir?**

Aa

Doğru yazma ve özel karakterler

Büyük-küçük harf · noktalama · parantezler · Türkçe Q klavye

TERİM ***büyük-küçük harfe duyarlı*** (*case-sensitive*); ***noktalı virgül*** (*semicolon*); ***söz dizimi hatası*** (*syntax error*)

Java hata affetmez: tek bir yanlış büyük harf ya da eksik bir noktalı virgül derlemeyi durdurur. Kod, Türkçe klavyede yazması zahmetli simgeler de dahil, harfi harfine yazılmalıdır.

BÖLÜMLER **Bölüm 2.2 · Windows İşletim Sistemi** **Bölüm 5 · Java'ya Giriş ve İlk Program** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler** **Bölüm 7 · Koşullu İfadeler** **Bölüm 11 · Dizgiler**

- Java büyük-küçük harfe duyarlıdır: `System`, `system` ve `SYSTEM` üç ayrı addır ve yalnızca ilki tanımlıdır.
- Her gün yazılan simgeler: `; { } ( ) [ ] " ' + - * / % = < > ! & |`.
- Türkçe Q klavyede parantezler AltGr ile yazılır: AltGr + 7 `{`, AltGr + 8 `[`, AltGr + 9 `]`, AltGr + 0 `}` verir. Klavye düzenleri arasında geçiş (Win + Boşluk) Windows sunumunda anlatılır.
- Birbirine benzeyen karakterler karıştırılmamalıdır: 0 rakamı ile O harfi, 1 rakamı ile l harfi, düz tırnak `"` ile Word'ün koyduğu kıvrık tırnaklar “ ”. Java kıvrık tırnakları kabul etmez.

- Türkçedeki noktalı ve noktasız i (i, ı, İ, I) tanımlayıcılarda ve dizgi karşılaştırmalarında sorun çıkarabilir (Bölüm 11, "Türkçe Karakter ve Locale").

KENDİNİ SINAMA

▶ **Java `system.out.println("Hi");` satırını kabul eder mi?**

▶ **Türkçe Q klavyede `{` ve `}` nasıl yazılır?**



Dosyalar, klasörler, yollar ve uzantılar

Klasör ağacı · mutlak ve göreli yol · .java/.class

TERİM ***klasör*** (*folder, directory*); ***uzantı*** (*file extension*); ***mutlak yol*** (*absolute path*); ***göreli yol*** (*relative path*)

Programlar klasörlerde duran dosyalardır. Bir dosyanın nereye kaydedildiğini, uzantısıyla birlikte tam adını ve `C:\Ders\Hafta1\Merhaba.java` gibi bir yolun nasıl okunduğunu bilmek gerekir.

BÖLÜMLER **Bölüm 1.2 · Bilgisayar Donanımı** **Bölüm 2.1 · Yazılım ve İşletim Sistemleri** **Bölüm 2.2 · Windows İşletim Sistemi** **Bölüm 3.1 · Microsoft Word** **Bölüm 5 · Java'ya Giriş ve İlk Program**

- Yol, sürücüden (kökten) dosyaya kadar inen klasörleri sıralar: sürücü `C:`, klasörler `Ders` ve `Hafta1`, dosya `Merhaba.java`.
- Uzantı, adın son noktadan sonraki kısmıdır ve dosyanın türünü gösterir: `.docx`, `.xlsx`, `.pptx`, `.pdf`, `.txt`, `.java` (kaynak kod), `.class` (derlenmiş bytecode). Uzantıların görünmesi için Dosya Gezgini'nde "Dosya adı uzantıları" seçeneği açılır.
- Göreli yol geçerli klasörden başlar: `.` geçerli klasörü, `..` onun üst klasörünü gösterir.
- Java kuralı (Bölüm 5): `public class Merhaba` içeren dosyanın adı tam olarak `Merhaba.java` olmalıdır.
- Ders çalışmalarını her hafta ya da her bölüm için ayrı bir klasörde toplamak işi kolaylaştırır. Her şeyi Masaüstü'ne ya da İndirilenler'e kaydetmek dağınıklık yaratır.

KENDİNİ SINAMA

▶ **`C:\Users\Ali\Documents\odev\Merhaba.java` yolundaki dosyanın uzantısı nedir, dosya hangi klasördedir?**

▶ **Geçerli klasör `C:\Ders\Hafta1` ise `..\resim.png` göreli yolu nereyi gösterir?**

@ Web tarayıcısı, arama ve e-posta

Web adresleri · arama · ekler · güvenli alışkanlıklar

TERİM ***web tarayıcısı*** (*web browser*); ***arama motoru*** (*search engine*); ***ek*** (*attachment*); ***oltalama*** (*phishing*)

Ders boyunca malzeme indirilir, hata iletileri aranır, ödevler e-postayla ya da bir öğrenme platformu üzerinden gönderilir. Bölüm 1, web adreslerinin (URL), alan adı sisteminin (DNS) ve e-postanın arka planda nasıl çalıştığını da anlatır.

BÖLÜMLER **Bölüm 1.1 · Temel Kavramlar** **Bölüm 1.4 · İnternet** **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 11 · Dizgiler**

- Bir URL soldan sağa okunur: önce protokol (`https`), sonra alan adı (`www.example.edu.tr`), en sonda sunucudaki yol (`/duyurular`).
- Etkili aramada hata iletisi tırnak içinde aynen yazılır, yanına dilin adı ("java") eklenir ve resmî belgeler öne alınır.
- Bir e-posta adresi kullanıcı adı, `@` ve alan adından oluşur. İyi bir e-postanın konusu açıktır, eki gönderilmeden önce kontrol edilir ve doğru kişiye yanıt verilir.
- Güvenli alışkanlıklar: her hesap için ayrı ve güçlü bir parola, çok faktörlü doğrulama, beklenmedik bağlantılara ve eklere karşı temkin (oltalama).
- Bölüm 11, bir e-posta adresini `indexOf('@')` ve `substring` ile kullanıcı adı ve alan adı olarak ikiye ayırır. Adresin yapısını önceden tanımak bu yüzden işe yarar.

KENDİNİ SINAMA

▶ **`https://www.example.edu.tr/tr/duyurular` adresinde alan adı hangi kısımdır?**

▶ **`ali.veli@ogr.example.edu.tr` adresinde `@` işaretinden önceki ve sonraki kısımlar nedir?**

⬇ Yazılım kurma ve güncelleme

Resmî kaynaklar · kurulum dosyaları · sürümler

TERİM ***kurulum / kaldırma*** (*install / uninstall*); ***güncelleme*** (*update*); ***işletim sistemi*** (*operating system*)

Programlama bölümlerinden önce bir JDK (Java geliştirme kiti) ile bir düzenleyicinin kurulması gerekir. Bunun için kurulum dosyası resmî kaynaktan indirilir, çalıştırılır ve kurulan sürüm kontrol edilir.

BÖLÜMLER **Bölüm 2.1 · Yazılım ve İşletim Sistemleri** **Bölüm 2.2 · Windows İşletim Sistemi** **Bölüm 5 · Java'ya Giriş ve İlk Program**

- Programlar yalnızca üreticinin kendi sitesinden ya da resmî mağazadan indirilmelidir. "Ücretsiz indir" sitelerinden ve kırılmış yazılımlardan uzak durmak gerekir.
- Kullanılan sistemi tanımak gerekir: Ayarlar → Sistem → Hakkında ekranı Windows sürümünü ve sistemin 64 bit olup olmadığını gösterir.
- Kurulumdan sonra doğrulama yapılır: program (ya da bir terminal) açılır ve bildirdiği sürüme bakılır.
- İşletim sistemi ve tarayıcı güncel tutulur. Programlar klasörleri silinerek değil, Ayarlar → Uygulamalar üzerinden kaldırılır.

KENDİNİ SINAMA

▶ **Windows 11'de sistemin 64 bit olup olmadığı nerede görülür?**

▶ **Bir programı `C:\Program Files` içindeki klasörünü silerek kaldırmak neden yanlıştır?**

Aritmetik ve sayılar

İşleç önceliği, tam sayılar ve ondalıklı sayılar, bölüm ve kalan, yüzde ve ortalama, üsler ve sayı tabanları, ölçü birimlerinin önekleri ve logaritma.

Aritmetik ve işleç önceliği

Parantez · önce × ÷, sonra + − · soldan sağa

TERİM ***işleç önceliği*** (*operator precedence*); ***ifade*** (*expression*); ***formül*** (*formula*)

Excel formülleri ve Java ifadeleri okulda öğrenilen kurallara uyar: önce parantezler, sonra çarpma ve bölme, en son toplama ve çıkarma. Aynı öncelikteki işlemler soldan sağa yapılır.

BÖLÜMLER **Bölüm 3.2 · Microsoft Excel** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler** **Bölüm 7 · Koşullu İfadeler**

- Öncelik sırası: ( ) → üs alma (Excel'de `^`; Java'da üs işleci yoktur, `Math.pow` kullanılır) → `* / %` → `+ -`.
- Aynı düzeydeki işlemler soldan sağa yapılır: `12 / 3 * 2` ifadesi (12 / 3) · 2 = 8 olur, 12 / 6 değil.
- Emin olunamayan yerde parantez eklemek en iyisidir. Parantezin hiçbir maliyeti yoktur ve niyeti açıkça gösterir.
- Tek terimli eksi (*unary minus*) ve karşılaştırmalar da öncelik tablosunda yer alır. Bölüm 3 (Excel) ve Bölüm 6 (Java) iki tabloyu da verir.

Örnek (Bölüm 6). Java'da `"Sonuç: " + 2 + 3` ifadesi soldan sağa hesaplanır ve `Sonuç: 23` metnini verir. `"Sonuç: " + (2 + 3)` ise `Sonuç: 5` verir.

KENDİNİ SINAMA

▶ `2 + 3 * 4` **ve** `(2 + 3) * 4` **kaçtır?**

▶ **Soldan sağa hesaplanınca** `10 - 4 - 3` **kaçtır?**

2,5 Tam sayılar ve ondalıklı sayılar

Tam sayı · ondalıklı sayı · yuvarlama · ondalık ayırıcı

TERİM ***tam sayı*** (*integer*); ***kayan noktalı sayı*** (*floating-point number*); ***yuvarlama*** (*rounding*); ***kesme*** (*truncation*)

Bilgisayar tam sayıları ve ondalıklı sayıları farklı biçimde saklar. Bir hesabın hangi tür sayıyla yapıldığı önemlidir: `5 / 2` işleminin sonucu Java'da 2, hesap makinesinde ise 2,5'tir.

BÖLÜMLER **Bölüm 1.3 · Bilgisayarlar** **Bölüm 3.2 · Microsoft Excel** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler** **Bölüm 7 · Koşullu İfadeler**

- Tam sayılar (..., −2, −1, 0, 1, 2, ...) tam olarak saklanır. Java'nın `int` tipi 2.147.483.647'ye kadar olan değerleri tutar.
- Ondalıklı sayılar yaklaşık olarak saklanır (Bölüm 1, "0,1 Neden Tam Değil?"). Java'da `0.1 + 0.2` ifadesi `0.30000000000000004` yazdırır.
- Yuvarlama ile kesme farklıdır: 3,99 en yakın tam sayıya yuvarlanınca 4, kesilince (ondalık kısmı atılınca) 3 olur. Java'daki `(int) 3.99` keser ve 3 verir.
- Yazım: Türkçede 3,14 yazılır, Türkçe ayarlı Excel de virgül kullanır. Java kaynak kodunda ise her zaman nokta kullanılır: `3.14`.

KENDİNİ SINAMA

▶ **Java'da `5 / 2` ve `5.0 / 2` kaçtır?**

▶ **7,5 ve 7,49 en yakın tam sayıya yuvarlanınca ve kesilince ne olur?**

%

Bölüm ve kalanla bölme

Tam sayılarda / ve % · tek/çift · basamaklar

TERİM ***tam sayı bölmesi*** (*integer division*); ***bölüm*** (*quotient*); ***kalan*** (*remainder, modulo*); ***bölen*** (*divisor*)

İlkokulda öğrenilen kalanlı bölme (17 ÷ 5 = 3, kalan 2) derste en çok kullanılan matematiktir: `/` bölümü, `%` kalanı verir.

BÖLÜMLER **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 4.1 · Algoritmalar** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler** **Bölüm 7 · Koşullu İfadeler** **Bölüm 8 · Döngüler** **Bölüm 9 · Metotlar** **Bölüm 12 · Özyineleme**

- Pozitif tam sayılar için a = b · q + r ve 0 ≤ r < b olur. Java'da `q = a / b`, `r = a % b` yazılır.
- Tek mi, çift mi: `n % 2 == 0` ise n çifttir (Bölüm 7 "Tek mi Çift mi?", Bölüm 9 `ciftMi()`).
- Basamaklar: `n % 10` son basamağı verir, `n / 10` son basamağı atar. Bu iki adım tekrarlanarak basamaklar sayılır ya da toplanır (Bölüm 8, Bölüm 9, Bölüm 12).
- Birim dönüştürme: saniyeden saate, dakikaya ve saniyeye (Bölüm 6 "Saniyeyi Saate Çevirme").
- Bölünebilme: `a % b == 0` ise b, a'yı tam böler (asal sayılar, en büyük ortak bölen, artık yıllar).
- Java'da kalanın işareti bölünenin işaretiyle aynıdır: `-7 % 3` sonucu −1'dir (Bölüm 6).

KENDİNİ SINAMA

▶ **Tam sayılarla `17 / 5` ve `17 % 5` kaçtır?**

▶ **/ ve % kullanılarak bulunursa 3725 saniye kaç saat, kaç dakika, kaç saniyedir?**

$\bar{x}$ Yüzde ve ortalama

Bir değerin yüzdesi · ağırlıklı not · ortalama

TERİM ***ortalama*** (*average, mean*); ***yüzde*** (*percentage*); ***işlev*** (Excel'de *function*)

Excel'deki not çizelgeleri, ortalama sıcaklık hesaplayan metotlar ve dizi istatistikleri yüzdeye, düz ortalamaya ve ağırlıklı ortalamaya dayanır.

BÖLÜMLER **Bölüm 3.2 · Microsoft Excel** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler** **Bölüm 9 · Metotlar** **Bölüm 10 · Diziler**

- x'in %p'si x · p / 100'dür. %p artış değeri (1 + p/100) ile, %p azalış ise (1 − p/100) ile çarpar.
- Ortalama (aritmetik ortalama) = toplam ÷ adet. Excel'de `=ORTALAMA(B2:B10)` yazılır. Java'da bir döngü önce toplar, sonra böler. Kesirli kısmın kaybolmaması için `double` kullanılır.
- Ağırlıklı ortalama: vizenin %40, finalin %60 ağırlıkta olduğu ders notu 0,4 · vize + 0,6 · final olur.
- Bölüm 10, bir dizide ortalamanın üstündeki elemanları, en küçük ve en büyük değeri bulur.

KENDİNİ SINAMA

▶ **Vize 70 (ağırlık %40), final 85 (ağırlık %60). Ders notu kaçtır?**

▶ **250 TL'lik bir fiyata %20 indirim yapılırsa yeni fiyat ne olur? 4, 8 ve 9'un ortalaması kaçtır?**

10^n Üsler ve basamak değeri

Üsler · basamakların konumu · 2'nin kuvvetleri

TERİM ***üs / taban*** (*exponent / base*); ***basamak / basamak değeri*** (*digit / place value*)

Bir rakamın değeri bulunduğu basamağa bağlıdır (birler, onlar, yüzler = 10^0, 10^1, 10^2). Aynı fikir 2'nin kuvvetleriyle ikili sayıları açıklar. Üs hesaplamak da döngü ve özyineleme için klasik bir örnektir.

BÖLÜMLER **Bölüm 1.1 · Temel Kavramlar** **Bölüm 1.3 · Bilgisayarlar** **Bölüm 3.2 · Microsoft Excel** **Bölüm 8 · Döngüler** **Bölüm 12 · Özyineleme**

- a^n, a'nın n kez kendisiyle çarpımıdır. $a^0 = 1$, $a^1 = a$ ve $a^m \cdot a^n = a^{m+n}$.

- Basamak değeri: $307 = 3 \cdot 10^2 + 0 \cdot 10^1 + 7 \cdot 10^0$.
- Bilinmesi gereken 2'nin kuvvetleri: $2^0 = 1$, $2^1 = 2$, $2^4 = 16$, $2^8 = 256$, $2^{10} = 1024$.
- Excel'de `=2^10` yazılır. Java'da ^ üs alma değil, XOR işlemidir. Üs için `Math.pow(2, 10)` ya da bir döngü kullanılır (Bölüm 8 "Üs Alma", Bölüm 12 "Hızlı Üs").

KENDİNİ SINAMA

▶ **2^{10} ve 5^0 kaçtır?**

▶ **x = 1 ile başlanıp x 4 kez ikiye katlanırsa x kaç olur?**

01 Sayı tabanları, bit ve bayt

İkili sayılar · KB ve KiB · sınırlı aralıklar

TERİM ***ikili*** (*binary*); ***taban*** (*base, radix*); ***bayt*** (*byte*); ***KiB / MiB / GiB*** (*kibibyte / mebibyte / gibibyte*); ***taşma*** (*overflow*)

Bütün veriler bit olarak saklanır. Küçük sayıları onluk ve ikili tabanlar arasında çevirmek, depolama boyutlarını bayt cinsinden hesaplamak ve sabit sayıda bitin değer aralığını neden sınırladığını görmek gerekir.

BÖLÜMLER **Bölüm 1.1 · Temel Kavramlar** **Bölüm 1.2 · Bilgisayar Donanımı** **Bölüm 1.3 · Bilgisayarlar** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler** **Bölüm 8 · Döngüler**

- İkili tabanda basamak değerleri 1, 2, 4, 8, 16, … diye gider: $1011_2 = 8 + 0 + 2 + 1 = 11$.
- Onluktan ikiliye çevirmek için sayı art arda 2'ye bölünür ve kalanlar aşağıdan yukarıya okunur (Bölüm 1, Bölüm 8 "Onluktan İkiliye").
- n bit 2^n farklı değer verir: 8 bit = 1 bayt = 256 değer (işaretsiz olarak 0–255).
- Birimler: disk üreticilerinin kullandığı onluk (SI) gösterimde 1 KB = 1000 B, ikili (IEC) gösterimde ise 1 KiB = 1024 B olur. Bölüm 1'deki tabloda 1 KB = 1.024 B yazılır ve 1024 tabanlı birimlere KiB, MiB, GiB denir.
- Ağ hızları saniyedeki bit sayısıyla verilir: 100 Mbit/sn, 12,5 MB/sn eder.
- Sabit genişlikli bir değer taşar: 8 bitlik 255'e 1 eklemek dokuzuncu bir bit gerektirir. Java'nın `int` tipi 2.147.483.647'den sonra başa sarar (Bölüm 6 "Taşma").

KENDİNİ SINAMA

▶ **1011_2 onluk tabanda, 13 ise ikili tabanda nasıl yazılır?**

▶ **500 GB ($500 \cdot 10^9$ bayt) diye satılan bir diski Windows yaklaşık kaç GiB gösterir?**

GHz

Ölçü birimleri ve önekler

kilo, mega, giga · mili, mikro, nano · Hz ve saniye

TERİM ***saat hızı*** (*clock speed*); ***frekans*** (*frequency*); ***gecikme*** (*latency*); ***bant genişliği*** (*bandwidth*)

Donanım, ağ ve işletim sistemi konularında çok büyük ve çok küçük sayılar önekli birimlerle yazılır: 3 GHz saniyede 3 milyar tur, 5 ms ise saniyenin binde beşidir. Her önekin 10'un hangi kuvvetini gösterdiğini bilmek ve frekanstan süreye geçebilmek gerekir.

BÖLÜMLER **Bölüm 1.2 · Bilgisayar Donanımı** **Bölüm 1.3 · Bilgisayarlar** **Bölüm 1.4 · İnternet** **Bölüm 2.1 · Yazılım ve İşletim Sistemleri**

- Büyüten önekler: kilo (k) = 10^3, mega (M) = 10^6, giga (G) = 10^9, tera (T) = 10^{12}. Küçülten önekler: mili (m) = 10^{-3}, mikro (μ) = 10^{-6}, nano (n) = 10^{-9}.
- Negatif üs bölme demektir: $10^{-3} = 1/1000$. 1 s = 1000 ms = 1.000.000 μs = 10^9 ns.
- Hertz (Hz) saniyedeki tekrar sayısıdır. Bir turun süresi frekansın tersidir: 3 GHz bir işlemcide $1 \div (3 \times 10^9)$ s ≈ 0,33 ns.
- Hesapta birimler aynı tutulur: 100 Mbit/s hat saniyede 12,5 MB taşır, 500 MB'lık dosya en iyi durumda 500 ÷ 12,5 = 40 s'de iner. Bit ile bayt karıştırılırsa sonuç 8 kat yanlış çıkar.
- Bant genişliği saniyede taşınan bit miktarıdır; gecikme (ping) ise ilk bitin karşıya varma süresidir ve ms ile ölçülür. Biri yüksekken öteki de iyi olmak zorunda değildir.

KENDİNİ SINAMA

▶ **2 GHz bir işlemcide bir saat turu kaç ns sürer?**

▶ **0,25 s kaç milisaniyedir? 20 ms kaç saniyedir?**

$\log_2$

Logaritma ($\log_2$)

2'nin kuvvetlerinin tersi · yarıya bölme sayısı · bit sayısı

TERİM ***logaritma*** (*logarithm*); ***logaritmik zaman*** (*logarithmic time*)

$\log_2 n$, 2'nin kaçıncı kuvvetinin n ettiğini söyler: $2^{10} = 1024$ olduğundan $\log_2 1024 = 10$'dur. Aynı sayı, n'nin 1'e inene kadar kaç kez ikiye bölünebileceğini de verir. Bir değer kümesini kodlamak için gereken bit sayısı ve ikili aramanın tur sayısı bu yolla bulunur.

BÖLÜMLER **Bölüm 1.1 · Temel Kavramlar** **Bölüm 4.1 · Algoritmalar** **Bölüm 4.1 · Algoritmalar** **Bölüm 10 · Diziler** **Bölüm 12 · Özyineleme**

- Üssün tersidir: $2^k = n$ ise $\log_2 n = k$. $\log_2 1 = 0$, $\log_2 8 = 3$, $\log_2 1024 = 10$.
- Tam kuvvet olmayan sayılarda sonuç iki tam sayının arasına düşer: $2^5 = 32 < 58 < 64 = 2^6$ olduğundan $\log_2 58 \approx 5{,}86$. N farklı değeri kodlamak için gereken bit sayısı bu değerin yukarı yuvarlanmışıdır: 58 değer için 6 bit.
- Sıralı n öğeli bir dizide ikili arama her turda aralığı yarıya indirir ve en çok $\lfloor \log_2 n \rfloor + 1$ turda biter. Bir milyon öğe için bu sınır 20 turdur.
- Büyük O gösterimindeki O(log n) ve O(n log n) bu yavaş büyümeyi anlatır: n bin kat artınca $\log_2 n$ yalnızca 10 kadar artar, çünkü $2^{10} \approx 1000$.

KENDİNİ SINAMA

▶ **$\log_2 64$ kaçtır? 64 öğeli sıralı bir dizide ikili arama en çok kaç turda biter?**

▶ **40 öğrencinin her birine ayrı bir ikili numara vermek için en az kaç bit gerekir?**

Cebir, mantık ve fonksiyonlar

Sayıların yerine kullanılan harfler, basit formüller, eşitsizlikler, ve/veya/değil, girdiyi çıktıya dönüştüren bir kural olarak fonksiyon.

x

Değişkenler ve basit cebir

Sayı yerine harf · yerine koyma · atama

TERİM ***değişken*** (*variable*); ***atama*** (*assignment*); ***veri tipi*** (*data type*)

Cebirde bir harf bir sayının yerini tutar. Programlamada değişken, içeriği değişebilen adlandırılmış bir kutudur. Bir ifadede harflerin yerine değer koymak, bilgisayarın yaptığı işin ta kendisidir.

BÖLÜMLER **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 4.2 · Akış Diyagramları** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler** **Bölüm 8 · Döngüler**

- Yerine koyma: a = 3 ve b = 4 için $a^2 + b^2 = 9 + 16 = 25$.
- Java'da `=` eşitlik değil, atamadır: `x = x + 1`, "x + 1'i hesapla ve sonucu yeniden x'e yaz" demektir. Yanlış bir denklem değildir. Eşitlik `==` ile sınanır.
- İki değerin yerini değiştirmek için geçici bir değişken gerekir: `t = a; a = b; b = t;` (Bölüm 6 "Takas", Bölüm 9 "swap Tuzağı").
- Sayaçlar ve biriktiriciler (Bölüm 4 akış diyagramları, Bölüm 8): `sayac = sayac + 1`, `toplam = toplam + x`.

KENDİNİ SINAMA

▶ **x = 5 iken önce `x = x + 1;`, sonra `x = x * 2;` çalışırsa x kaç olur?**

▶ **a = 3, b = 5 iken `t = a; a = b; b = t;` çalışırsa a ve b ne olur?**

πr²

Günlük hayattan formüller

Alan · sıcaklık · ikinci dereceden denklem

TERİM ***karekök*** (*square root*); ***formül*** (*formula*); ***mutlak değer*** (*absolute value*)

İlk programların çoğu bir okul formülünü koda döker: dairenin alanı ve çevresi, Celsius'tan Fahrenheit'a çevirme, ikinci dereceden denklemin kökleri, mutlak değer.

BÖLÜMLER **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler** **Bölüm 7 · Koşullu İfadeler** **Bölüm 9 · Metotlar**

- Daire: alan πr^2, çevre $2\pi r$ (Bölüm 6 "Daire Alanı ve Çevresi", `Math.PI`).
- Sıcaklık: $F = C \cdot 9/5 + 32$ ve $C = (F - 32) \cdot 5/9$ (Bölüm 6, Bölüm 9 "Sıcaklık Metotları").
- İkinci dereceden $ax^2 + bx + c = 0$ denklemi: diskriminant $\Delta = b^2 - 4ac$. $\Delta > 0$ ise iki, $\Delta = 0$ ise bir gerçek kök vardır, $\Delta < 0$ ise gerçek kök yoktur. Kökler $(-b \pm \sqrt{\Delta}) / 2a$ formülüyle bulunur (Bölüm 7).
- Mutlak değer $|x|$, $x \geq 0$ ise x'e, değilse $-x$'e eşittir (Bölüm 7 "Mutlak Değer"). Karekökün içi negatif olamaz (Bölüm 1 "Karekök Adım Adım").

KENDİNİ SINAMA

▶ **25 °C kaç Fahrenheit eder?**

▶ **$x^2 - 5x + 6 = 0$ denkleminde Δ ve kökler nedir?**

Eşitsizlikler ve aralıklar

< ≤ > ≥ · aralıklar · sınırlar

TERİM ***karşılaştırma işleci*** (*comparison operator*); ***koşul*** (*condition*); ***sınır değeri*** (*boundary value*); ***sınır hatası*** (*off-by-one error*)

"10 < x < 20", "not ≥ 50" ya da "indis 0'dan n − 1'e kadar" gibi koşullar birer eşitsizliktir. Uç noktanın dahil olup olmadığı birçok sınır hatasının kaynağıdır.

BÖLÜMLER **Bölüm 3.2 · Microsoft Excel** **Bölüm 4.1 · Algoritmalar** **Bölüm 4.2 · Akış Diyagramları** **Bölüm 7 · Koşullu İfadeler** **Bölüm 8 · Döngüler** **Bölüm 10 · Diziler**

- Kesin eşitsizlik (<, >) uç noktayı dışarıda bırakır, kesin olmayan (≤, ≥) ise içine alır. Java'da ≤ `<=`, ≠ ise `!=` diye yazılır.
- Çift taraflı bir eşitsizlik "ve" ile bağlanmış iki koşuldur: 10 < x < 20, `x > 10 && x < 20` olur. Java'da `10 < x < 20` yazılamaz.
- Bir aralıktaki tam sayıları saymak: a'dan b'ye (ikisi de dahil) b − a + 1 sayı vardır.
- Döngüler: `for (int i = 0; i < 5; i++)` 5 kez (i = 0..4), `i <= 5` ile 6 kez döner (Bölüm 8 "Sınır Hataları"). n uzunluklu bir dizinin indisleri 0 … n − 1'dir (Bölüm 10).

KENDİNİ SINAMA

▶ **10 < x < 20 koşulunu kaç tam sayı sağlar? 10 ≤ x ≤ 20 koşulunu kaç tam sayı sağlar?**

▶ `for (int i = 0; i < 5; i++)` **döngüsü kaç kez döner?** `i <= 5` **olursa kaç kez döner?**

Mantık: ve, veya, değil

Doğru/yanlış · doğruluk tabloları · De Morgan

TERİM ***boolean (mantıksal tip)*** (*boolean*); ***mantıksal işleç*** (*logical operator*); ***AND / OR / NOT / XOR*** (*AND / OR / NOT / XOR*); ***De Morgan kuralları*** (*De Morgan's laws*)

Bir programdaki her karar doğru ya da yanlış olan bir önermedir ve çoğu zaman ve, veya, değil ile birleştirilir. Bu mantık, mantık kapılarında (Bölüm 1), Excel'in VE/YADA/DEĞİL işlevlerinde (Bölüm 3) ve Java'nın `&&`, `||`, `!` işleçlerinde karşımıza çıkar.

BÖLÜMLER **Bölüm 1.1 · Temel Kavramlar** **Bölüm 1.3 · Bilgisayarlar** **Bölüm 3.2 · Microsoft Excel** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler**

Bölüm 7 · Koşullu İfadeler **Bölüm 8 · Döngüler**

- A ve B yalnızca ikisi de doğruysa doğrudur. A veya B en az biri doğruysa doğrudur. Değil A, A'nın değerini tersine çevirir.
- Doğruluk tablosu bütün durumları listeler: iki girdi için 4 satır vardır.
- De Morgan: değil (A ve B) = (değil A) veya (değil B); değil (A veya B) = (değil A) ve (değil B) (Bölüm 7).
- Mantıktaki "veya" kapsayıcıdır: doğru veya doğru, doğrudur. "Ya da" anlamındaki dışlayıcı veya (XOR) ayrı bir işlemdir.
- Artık yıl kuralı (Bölüm 7): 4'e bölünür ve (100'e bölünmez veya 400'e bölünür).

KENDİNİ SINAMA

▶ **x = 15 iken `x > 10 && x < 20` ve `!(x > 10)` ifadelerinin değeri nedir?**

▶ **Artık yıl kuralına göre 1900 ve 2000 artık yıl mıdır?**

f(x)

Girdi → çıktı olarak fonksiyonlar

f(x) · argümanlar · dönüş değeri · bileşke

TERİM ***fonksiyon*** (*function*); ***metot*** (*method*); ***parametre*** (*parameter*); ***dönüş değeri*** (*return value*)

Matematikteki bir fonksiyon girdileri alır ve tek bir çıktı verir: f(x) = 2x + 3. Excel işlevleri (`=TOPLA(…)`) ve Java metotları da böyle çalışır: argümanlar girer, bir değer döner.

BÖLÜMLER **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 3.2 · Microsoft Excel** **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 9 · Metotlar** **Bölüm 12 · Özyineleme**

- f(x) = 2x + 3, "x verilince 2x + 3'ü döndür" demektir ve f(4) = 11 olur. Girdinin adı önemli değildir: f(t) = 2t + 3 aynı fonksiyondur.
- Birden çok girdili fonksiyonlar: max(a, b), ebob(a, b), Excel'de `=MAK(B2:B10)`.
- Bileşke (*composition*): f(f(1)) hesaplanırken önce içteki çağrı hesaplanır. Bölüm 9 iç içe çağrıları çağrı yığını üzerinde adım adım izler.
- Bir Java metodunun parametre listesi (girdiler), dönüş tipi (çıktının tipi) ve gövdesi vardır. `void` metotlar değer döndürmez.

KENDİNİ SINAMA

▶ **f(x) = 2x + 3 ise f(4) ve f(f(1)) kaçtır?**

▶ **Excel'de B2:B4 hücrelerinde 3, 9 ve 4 var. `=MAK(B2:B4)` ve `=TOPLA(B2:B4)` ne döndürür?**

Sayı dizileri, bölünebilme ve özyineleme

1 + 2 + … + n gibi toplamlar, asal sayılar ve en büyük ortak bölen, faktöriyel ve Fibonacci, satır-sütun düzenlerini ve konumları saymak. Bunlar döngü, metot, dizi ve özyineleme sunumlarının standart örnekleridir.

Σ Sayı dizileri ve toplamlar

1 + 2 + … + n · basamak toplamı · birikimli toplam

TERİM ***biriktirici*** (*accumulator*); ***sayaç*** (*counter*); ***döngü*** (*loop*)

Bir sayı dizisini terim terim toplamak her döngünün ilk işidir (Bölüm 4 akış diyagramı "1'den n'ye Toplam", Bölüm 8, Bölüm 10, Bölüm 12). Kapalı formülleri bilmek, programın verdiği cevabı kontrol etmeyi sağlar.

BÖLÜMLER **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 4.2 · Akış Diyagramları** **Bölüm 8 · Döngüler** **Bölüm 10 · Diziler** **Bölüm 12 · Özyineleme**

- 1 + 2 + … + n = n(n + 1)/2 (Gauss). Örneğin 1 + 2 + … + 100 = 5050.
- Aritmetik dizi: a ile başlar, d kadar artar (2, 4, 6, …). Geometrik dizi: a ile başlar, her adımda r ile çarpılır (1, 2, 4, 8, …).
- Birikimli toplam 0'dan başlar (toplamanın etkisiz elemanı), birikimli çarpım ise 1'den.
- 4827'nin basamak toplamı 4 + 8 + 2 + 7 = 21'dir (Bölüm 8 "Rakamlar Toplamı", Bölüm 12'de özyinelemeli hâli).

KENDİNİ SINAMA

▶ **1 + 2 + … + 100 ve 2 + 4 + … + 20 toplamları kaçtır?**

▶ **s = 0 iken i = 1, 2, 3, 4 için s = s + i · i yapılırsa sonunda s kaç olur?**

÷ Bölünebilme, asal sayılar ve EBOB

Bölenler · asallık testi · Öklid · EKOK

TERİM ***asal sayı*** (*prime number*); ***en büyük ortak bölen (EBOB)*** (*greatest common divisor, GCD*); ***en küçük ortak kat (EKOK)*** (*least common multiple, LCM*); ***Öklid algoritması*** (*Euclid's algorithm*)

Asallık testi, mükemmel sayılar, EBOB/EKOK ve Eratosthenes eleği algoritma, döngü, metot, dizi ve özyineleme sunumlarının standart örnekleridir.

BÖLÜMLER **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 4.1 · Algoritmalar** **Bölüm 8 · Döngüler** **Bölüm 9 · Metotlar** **Bölüm 10 · Diziler** **Bölüm 12 · Özyineleme**

- n % d == 0 ise d, n'yi böler. Asal sayının tam iki böleni vardır: 1 ve kendisi. 1 asal değildir, 2 de tek çift asaldır.
- n'nin asal olup olmadığını anlamak için √n'ye kadar olan bölenleri denemek yeter (Bölüm 4, Bölüm 8, Bölüm 9 `asalMi()`).
- Öklid algoritması: kalan 0 olana kadar ebob(a, b) = ebob(b, a % b) uygulanır (Bölüm 1, Bölüm 9 `ebob()`, Bölüm 12). Ardından ekok(a, b) = a · b / ebob(a, b) bulunur.
- Mükemmel sayı, kendisi dışındaki bölenlerinin toplamına eşit olan sayıdır: 6 = 1 + 2 + 3, 28 = 1 + 2 + 4 + 7 + 14 (Bölüm 8).

KENDİNİ SINAMA

▶ **91 asal mıdır?**

▶ **Öklid algoritmasıyla ebob(48, 18) ve ardından ekok(48, 18) kaçtır?**

n!

Faktöriyel, Fibonacci ve özyinelemeli tanımlar

n! = n · (n − 1)! · taban durum · hızlı büyüme

TERİM ***faktöriyel*** (*factorial*); ***Fibonacci dizisi*** (*Fibonacci sequence*); ***özyineleme*** (*recursion*); ***taban durum*** (*base case*)

Bazı diziler önceki terimlere başvurularak tanımlanır: 0! = 1 olmak üzere n! = n · (n − 1)!, her Fibonacci sayısı da kendinden önceki iki sayının toplamıdır. Bölüm 12'deki özyinelemeli metotlar tam olarak böyle yazılır.

BÖLÜMLER **Bölüm 4.1 · Algoritmalar** **Bölüm 8 · Döngüler** **Bölüm 12 · Özyineleme**

- n! = 1 · 2 · … · n olduğundan 5! = 120'dir. Özyinelemeli tanım n! = n · (n − 1)!, taban durum ise 0! = 1'dir (1! = 1 de olur).

- Fibonacci: 0, 1, 1, 2, 3, 5, 8, 13, 21, … Her terim kendinden önceki iki terimin toplamıdır.
- Her özyinelemeli tanımın onu durduran bir taban durumu olmalıdır. Taban durum yoksa hesap hiç bitmez (Bölüm 12 `StackOverflowError`).
- Bu sayılar hızla büyür: 13! = 6.227.020.800 bir `int` içine sığmaz (en büyük değer 2.147.483.647). Bölüm 12 bu yüzden `long` kullanır ve taşmayı tartışır.

KENDİNİ SINAMA

▶ **5! = 120 olduğu biliniyor. n! = n · (n − 1)! kuralıyla 6! kaçtır?**

▶ **5, 8, 13'ten sonra gelen Fibonacci sayısı kaçtır?**



Sayma, satır-sütun düzeni ve konumlar

Satır × sütun · desenler · 0'dan saymak

TERİM ***iç içe döngü*** (*nested loop*); ***indis*** (*index*); ***desen*** (*pattern*, çıktıda); ***dizi*** (*array*)

İç içe döngüler, yıldız desenleri, çarpım tabloları, iki boyutlu diziler ve dizgi indisleri hep aynı iki beceriye dayanır: satır ve sütunlardaki hücreleri saymak ve konumları 0'dan numaralamak.

BÖLÜMLER **Bölüm 3.2 · Microsoft Excel** **Bölüm 4.2 · Akış Diyagramları** **Bölüm 8 · Döngüler** **Bölüm 10 · Diziler** **Bölüm 11 · Dizgiler**

- r satır ve c sütunlu bir tabloda r · c hücre vardır. Dış döngüsü r, iç döngüsü c adım süren iç içe bir döngü r · c kez çalışır.
- Üçgen desenler: satırlarında 1, 2, …, n yıldız bulunan bir üçgende toplam 1 + 2 + … + n yıldız vardır (Bölüm 8 “Artan Üçgen”).
- Java konumları 0'dan sayar: n uzunluklu bir dizinin ya da dizginin indisleri 0 … n − 1'dir, sonuncusu n − 1'dir (Bölüm 10, Bölüm 11).
- Excel hücre başvuruları (B3 = B sütunu, 3. satır) ve karma başvurularla kurulan çarpım tablosu (Bölüm 3) da aynı satır-sütun düşüncesine dayanır.

KENDİNİ SINAMA

▶ **Bir yıldız üçgeninin satırlarında 1, 2, 3, 4, 5 yıldız var. Toplam kaç yıldız vardır?**

▶ **“MERHABA” sözcüğü 7 harflidir. Konumlar 0'dan numaralanırsa son konum kaçtır, 2. konumda hangi harf vardır?**

Okuma ve problem çözme becerileri

Yönergeleri harfi harfine izlemek, değerleri adım adım takip etmek, dikkatle seçilmiş girdilerle test etmek, teknik İngilizce ve Türkçe metin okumak.

123 Adım adım yönergeleri izleme

Tarifler · algoritmalar · akış diyagramı simgeleri

TERİM ***algoritma*** (*algorithm*); ***akış diyagramı*** (*flowchart*); ***sözde kod*** (*pseudocode*); ***adım*** (*step*)

Algoritma, "ne kastedildiği tahmin edilmeden", harfi harfine ve sırayla uygulanması gereken bir tariftir. Akış diyagramlarını ve sözde kodu okumak, günlük yönergelerden Java'ya geçişin köprüsüdür.

BÖLÜMLER **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 4.1 · Algoritmalar** **Bölüm 4.2 · Akış Diyagramları**

- Her adım sırayla ve tam olarak söylendiği gibi uygulanır. Döngü "geri dön ve tekrarla", karar ise "dallardan yalnızca birine git" demektir.
- Akış diyagramı simgeleri (Bölüm 4): oval = başla/bitir, paralelkenar = giriş/çıkış, dikdörtgen = işlem, eşkenar dörtgen = karar, oklar = sıra.
- Üç yapı taşı vardır: sıralı yapı, seçim (if) ve tekrar (döngü) (Bölüm 1, Bölüm 7).
- İyi bir yönerge açıktır ve sonlu sayıda adımda biter. Bölüm 4, iyi bir algoritmanın özelliklerini sıralar.

KENDİNİ SINAMA

▶ **Adımlar: (1) x ← 1. (2) 4 kez tekrarla: x ← x · 2. (3) x'i yazdır. Ekrana ne yazılır?**

▶ **Akış diyagramında karar hangi şekille gösterilir ve bu şekilden kaç ok çıkar?**

Değer tablosuyla izleme

İzleme tabloları · her adımdan sonraki durum

TERİM ***izleme tablosu*** (*trace table*); ***tur*** (*iteration*); ***döngü değişkeni*** (*loop variable*)

Her sunumda değişkenlerin her adımdan sonraki değerlerini gösteren "Adım Adım" slaytları vardır. Böyle bir tablo kâğıt üzerinde de tutulabilmelidir: her değişken için bir sütun, her adım için bir satır.

BÖLÜMLER **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 4.1 · Algoritmalar** **Bölüm 4.2 · Akış Diyagramları** **Bölüm 8 · Döngüler** **Bölüm 9 · Metotlar** **Bölüm 12 · Özyineleme**

- Sütun başlıklarına değişken adları yazılır. Bir değer her değiştiğinde yeni bir satır eklenir ve hiçbir adım atlanmaz.
- Döngü koşulu her turdan önce denetlenir. Döngüyü bitiren son denetim de bunlara dahildir.
- Tablo programın çıktısıyla karşılaştırılır. İkisinin ayrıştığı ilk satır hatanın yerini gösterir.
- Bölüm 9 ve Bölüm 12 izlemeyi çağrı yığınına taşır: her metot çağrısı kendi satırını (çerçevesini) alır.

KENDİNİ SINAMA

▶ **a = 10, b = 3, k = 0. a ≥ b olduğu sürece a ← a − b ve k ← k + 1 yapılırsa sonunda a ve k kaç olur?**

▶ **Önceki soruda a ≥ b koşulu kaç kez denetlenir?**

✓

Problemi parçalara ayırma ve test etme

Ayrıştırma · test durumları · sınır durumları

TERİM ***ayrıştırma*** (*decomposition*); ***test durumu*** (*test case*); ***sınır durumu*** (*edge case, boundary case*); ***hesaplamalı düşünme*** (*computational thinking*)

Kod yazmadan önce problem girdilere, adımlara ve çıktılara ayrılır. Kod yazıldıktan sonra hem sıradan girdilerle hem de programların genellikle hata yaptığı sınır değerleriyle test edilir.

BÖLÜMLER **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 4.1 · Algoritmalar** **Bölüm 4.2 · Akış Diyagramları** **Bölüm 8 · Döngüler** **Bölüm 9 · Metotlar**

- Hesaplamalı düşünmenin dört öğesi (Bölüm 1): ayrıştırma, örüntü tanıma, soyutlama ve algoritma tasarımı.

- Kod yazılmadan önce girdi ve çıktı bir örnekle belirlenir (Bölüm 1 “Girdi ve Çıktı Örneği”, Bölüm 9 “Önce Metot Sözleşmesi”).
- Test durumları: sıradan bir değer, her koşulun iki yanındaki sınır değerleri ve 0, 1, negatif sayılar, boş girdi gibi özel değerler.
- Her seferinde tek bir şey değiştirilir ve program yeniden çalıştırılır. Bir hatayı ortaya çıkaran girdiler kalıcı test olarak saklanır.

KENDİNİ SINAMA

▶ **Kural “not ≥ 50 ise geçer”. Sınırı hangi iki girdi test eder?**

▶ **Artık yıl kuralının bütün dallarını birlikte test eden dört yıl hangileri olabilir?**

EN

Teknik İngilizce ve hata iletileri

Java anahtar sözcükleri · javac ve çalışma zamanı iletileri

TERİM ***anahtar sözcük*** (*keyword*); ***hata iletisi*** (*error message*); ***derleme hatası*** (*compile-time error*); ***çalışma zamanı hatası*** (*runtime error*)

Java'nın anahtar sözcükleri, metot adları ve derleyicinin ve çalışma ortamının bütün iletileri İngilizcedir. Bu iletileri dikkatle okumak (ne, nerede, hangi satırda) bir programı düzeltmenin en hızlı yoludur.

BÖLÜMLER **Bölüm 5 · Java'ya Giriş ve İlk Program** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler** **Bölüm 7 · Koşullu İfadeler** **Bölüm 8 · Döngüler** **Bölüm 9 · Metotlar** **Bölüm 10 · Diziler** **Bölüm 11 · Dizgiler** **Bölüm 12 · Özyineleme**

- Anahtar sözcükler anlamı kesin, sıradan İngilizce sözcüklerdir: `if` (eğer), `else` (değilse), `while` (“olduğu sürece”), `for`, `return` (döndür), `break`, `new`, `void`, `static`.
- javac iletileri dosyayı, satırı, kısa bir açıklamayı ve hatanın yerini gösteren bir şapka işaretini (^) verir: `Merhaba.java:5: error: ';' expected`.
- Sık görülen iletiler: `cannot find symbol` (yanlış yazılmış ya da tanımlanmamış ad), `incompatible types` (uyumsuz tipler), `missing return statement` (eksik `return` deyimi).
- Çalışma zamanı iletileri istisnalardır: `ArithmeticException: / by zero`, `ArrayIndexOutOfBoundsException`, `StackOverflowError`.

- Önce ilk hata düzeltilir, çünkü sonraki iletiler çoğu zaman onun sonucudur (Bölüm 5 “Sık Derleme Hataları”).

KENDİNİ SINAMA

▶ **javac `error: ';' expected` iletisi veriyor. Neye bakmak gerekir?**

▶ **`error: cannot find symbol … symbol: variable toplm` iletisi genellikle ne anlama gelir?**

TR

Türkçe teknik terimler

Her kavram için tek terim · ders sözlüğü

TERİM ***dizi*** (*array*); ***dizgi*** (*string*); ***döngü*** (*loop*); ***özyineleme*** (*recursion*)

Slaytlar Türkçedir ve her İngilizce kavram için hep aynı Türkçe terimi kullanır. Bu eşleşmeleri erken öğrenmek işe yarar, çünkü birbirine benzeyen sözcükler çok farklı şeyleri anlatabilir. Tam liste ders sözlüğündedir.

BÖLÜMLER **Bölüm 1.1 · Temel Kavramlar** **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 5 · Java'ya Giriş ve İlk Program** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler** **Bölüm 7 · Koşullu İfadeler** **Bölüm 8 · Döngüler** **Bölüm 9 · Metotlar** **Bölüm 10 · Diziler** **Bölüm 11 · Dizgiler** **Bölüm 12 · Özyineleme**

- Kolayca karışan çiftler: *dizi* ve *dizgi*; *bölüm* (bölmenin sonucu, ayrıca kitabın bölümü) ve *bölen*; *ifade* (bir değer üreten kod parçası) ve *deyim* (tek başına çalışan komut).
- Bazı terimler slaytlarda İngilizce kalır: `boolean`, `IDE`, `JDK`, `break` ve koddaki metot adları.
- Sözlük bölüme göre süzülebilir ve iki dilde de aranabilir. Bilinmeyen bir sözcüğe dersten önce bakmak işi kolaylaştırır.
- Ders kodlarındaki tanımlayıcılar Türkçedir ama Türkçeye özgü harf içermez (`toplam`, `ciftMi`, `asalMi`).

KENDİNİ SINAMA

▶ ***dizi*** **ile** ***dizgi*** **arasındaki fark nedir?**

▶ ***taban durum*** **ve** ***döngü değişkeni*** **terimlerinin İngilizce karşılıkları nedir?**

Araçlar ve kurulum

Bir JDK ile bir düzenleyici ya da IDE, birkaç terminal komutu ve bir ofis paketi.

JDK ve bir düzenleyici ya da IDE

javac · java · .java → .class

TERİM ***JDK (geliştirme kiti)*** (*Java Development Kit*); ***tümleşik geliştirme ortamı (IDE)*** (*integrated development environment*); ***derleyici*** (*compiler*); ***kaynak kod*** (*source code*)

Bölüm 5'ten itibaren Java programları kişinin kendi bilgisayarında yazılır, derlenir ve çalıştırılır. Bunun için güncel bir JDK ile düz bir düzenleyici ya da bir IDE (örneğin Eclipse, IntelliJ IDEA ya da Java eklentili VS Code) kurulur.

BÖLÜMLER **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 5 · Java'ya Giriş ve İlk Program**

- Yalnızca JRE (Java çalışma ortamı) değil, JDK kurulmalıdır. Bölüm 5'teki iletiler JDK 21 ile üretilmiştir. Kurulum `javac -version` ve `java -version` ile kontrol edilir.
- Kaynak kod, `.java` uzantılı düz metin dosyasıdır. Word belgeleri derlenemez.
- Çalışma sırası şöyledir: `Merhaba.java` yazılır → `javac Merhaba.java` komutu `Merhaba.class` dosyasını (bytecode) üretir → `java Merhaba` komutu onu Java Sanal Makinesi (JVM) üzerinde çalıştırır.
- IDE bu adımları tek düğmeyle yapar ve hataların altını yazarken çizer. Yine de bu adımları en az bir kez elle yapmak gerekir.
- Dizgilerdeki Türkçe karakterlerin bozulmaması için dosyalar UTF-8 olarak kaydedilir (`javac -encoding UTF-8` , Bölüm 5).

KENDİNİ SINAMA

▶ **`Merhaba.java` hangi komutla derlenir, program hangi komutla çalıştırılır ve derleme hangi dosyayı oluşturur?**

▶ **Dosyanın adı `Odev1.java` , içinde ise `public class Merhaba` var. Ne olur?**

Terminal kullanımı (en temel düzeyde)

cd · dir/ls · komut çalıştırma

TERİM ***komut satırı*** (*command line*); ***Komut İstemi*** (*Command Prompt*); ***geçerli klasör*** (*current/working directory*)

Birkaç komut yeterlidir: nerede olunduğunu görmek, dosyaları listelemek, klasör değiştirmek ve `javac` / `java` çalıştırmak. Bölüm 2 bu komutları Windows (cmd) ve Linux/macOS için tanıtır.

BÖLÜMLER **Bölüm 2.1 · Yazılım ve İşletim Sistemleri** **Bölüm 2.2 · Windows İşletim Sistemi** **Bölüm 5 · Java'ya Giriş ve İlk Program**

- Terminal açmak: Başlat → "cmd" (Komut İstemi) ya da Windows Terminal. Dosya Gezgini'nin adres çubuğuna `cmd` yazılırsa terminal o klasörde açılır.
- Windows cmd: `cd` (klasörü göster ya da değiştir), `dir` (listele), `mkdir` (yeni klasör). Linux/macOS: `pwd`, `ls`, `cd`, `mkdir`.
- `cd ..` üst klasöre çıkar. Boşluk içeren yollar tırnak içinde yazılır: `cd "C:\Ders Notlari"`.
- Komutlar geçerli klasörde çalışır: yanlış klasörde bulunulursa `javac Merhaba.java` "dosya bulunamadı" hatası verir.
- `del` ve `rm` dosyaları kalıcı olarak siler (Geri Dönüşüm Kutusu'na göndermez). Enter'a basmadan önce komutu kontrol etmek gerekir.

KENDİNİ SINAMA

▶ **`C:\Ders\Hafta1` klasöründeyken `cd ..` yazılırsa hangi klasöre geçilir?**

▶ **Geçerli klasördeki dosyaları listeleyen Windows komutu nedir, Linux/macOS'taki karşılığı hangisidir?**

Ofis yazılımları

Word · Excel · PowerPoint ya da ücretsiz bir seçenek

TERİM ***kelime işlemci*** (*word processor*); ***hücre başvurusu*** (*cell reference*); ***sunu*** (*presentation*, dosya)

Bölüm 3, Türkçe arayüzlü Microsoft Word, Excel ve PowerPoint ile anlatılır. Üniversitenin Microsoft 365 hesabı ya da LibreOffice gibi ücretsiz bir seçenek dersi izlemeye yeter.

BÖLÜMLER **Bölüm 1.1 · Temel Kavramlar** **Bölüm 3.1 · Microsoft Word** **Bölüm 3.2 · Microsoft Excel** **Bölüm 3.3 · Microsoft PowerPoint**

- Dosya açmak, kaydetmek ("Farklı Kaydet" ile klasör ve biçim seçerek), yazdırmak ya da PDF olarak dışa aktarmak bilinmelidir.

- Biçimler: `.docx` (Word), `.xlsx` (Excel), `.pptx` (PowerPoint). Ücretsiz paketler de bu dosyaları açıp kaydedebilir.
- Türkçe Excel, işlev adlarını Türkçe yazar ve argümanları noktalı virgülle ayırır: `=EĞER(B2>=50;"Geçti";"Kaldı")`.
- Temel Excel sözcükleri: hücre (B3), aralık (B2:B10), formül (= ile başlar), göreli ve mutlak başvuru (`$A$1`).

KENDİNİ SINAMA

▶ **A1'den A10'a kadar olan hücreleri toplayan Türkçe Excel formülü nedir?**

▶ **Word, Excel ve PowerPoint varsayılan olarak hangi uzantıları kullanır?**

Aramaya ya da süzgeçlere uyan bir ön bilgi bulunamadı.

▶ Sunum haritası

© Sercan Külcü · Bilgisayar ve Programlamaya Giriş

EK C

Bilgisayar ve Programlamada Önemli İsimler

Bilgisayar ve Programlamaya Giriş · İsimler

Derste kullanılan fikirlerin arkasındaki 74 matematikçi, mühendis ve programcının 58 kartta toplanan kısa profilleri. Liste Öklid algoritmasından ve Boole mantığından Turing makinelerine, transistöre, internete, Unix'e, elektronik tablolara ve Java'ya uzanır. Her kart, kişinin çalışmalarının geçtiği ders bölümlerine bağlanır; ayrıntılar kartın altındaki düğmeyle açılır.

EU

Öklid (Euclid of Alexandria)

MÖ 300 dolayları · Yunan · İskenderiye

Öklid algoritması (EBOB) · *Elements*, VII. kitap

Elements (Öğeler) adlı eserinde en büyük ortak bölenin (EBOB, *greatest common divisor*) nasıl bulunacağını anlattı. Bu yöntem bilinen en eski algoritmalardan biridir ve bugün de öğretilir.

en büyük ortak bölen · kalan · sonlanma

DERS BÖLÜMLERİ: **Bölüm 9 · Metotlar** · **Bölüm 12 · Özyineleme**

Önemi: ebob(a, b) = ebob(b, a mod b) kuralı b = 0 olunca durur. Derste hem döngülü hem özyinelemeli metodun örnek modeli bu kuraldır.

Öklid'in özgün yöntemi art arda çıkarma kullanır; kalanlı biçim bunun bugünkü karşılığıdır. Her adımda sayılar küçülür. Algoritmanın her zaman durduğunu gösteren standart gerekçe budur.

ER

Kireneli Eratosthenes (Eratosthenes of Cyrene)

MÖ yak. 276 – yak. 194 · Yunan · İskenderiye Kütüphanesi

Eratosthenes eleği · asal sayılar

İskenderiye Kütüphanesinin başkütüphanecisiydi. Asal sayıları bulmak için her asalın katlarını art arda işaretleyip eleyen yöntem onun adıyla anılır.

asal sayılar · Eratosthenes eleği · boolean dizi

DERS BÖLÜMLERİ: **Bölüm 8 · Döngüler** · **Bölüm 10 · Diziler**

Önemi: Elek, her sayının elenip elenmediğini bir `boolean` dizisinde tutar. Bir asal bulunduğunda onun bütün katları tek geçişte işaretlenir. Dış döngü $i \cdot i > n$ olunca durabilir, çünkü n'ye kadar her bileşik sayının $\sqrt{n}$'den büyük olmayan bir asal böleni vardır. Her sayıyı ayrı ayrı bölerek denemekten çok daha hızlıdır.

Yaz gündönümünde İskenderiye ile Syene'deki (bugünkü Asvan) öğle gölgelerini karşılaştırarak Dünya'nın çevresini hesapladı. Bazı tarihçiler yöntemi Eratosthenes'in bulmadığını, yalnızca "elek" adını verdiğini düşünür.

AK

Muhammad ibn Musa al-Khwarizmi (Harezmî)

yak. 780 – yak. 850 · İranlı · Beytülhikme (House of Wisdom), Bağdat

"Algoritma" ve "cebir" sözcüklerinin kaynağı

Hint-Arap rakamlarıyla aritmetik ve denklem çözümü üzerine etkili kitaplar yazdı. Adının Latince biçimi Algoritmi, "algoritma" sözcüğüne dönüştü.

adım adım yöntemler · onluk rakamlar · cebir

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 4.1 · Algoritmalar**

Önemi: Dersin merkezindeki fikir onun adını taşır: bir problemi çözen, sonlu ve kesin adımlar dizisi.

"Cebir" anlamındaki *algebra* sözcüğü, denklemler üzerine yazdığı kitabın (yak. 820) adındaki *al-jabr*'dan gelir. Aritmetik kitabının Latince çevirileri, basamak değerli onluk gösterimi Orta Çağ Avrupası'na yaydı.

LF

Pisalı Leonardo (Leonardo of Pisa, Fibonacci)

yak. 1170 – yak. 1250 · İtalyan · Pisa

***Liber Abaci* · 1202**

1202'de yazdığı *Liber Abaci* ile Hint-Arap rakamlarını Avrupa'da yaygınlaştırdı. Kitaptaki tavşan problemi 0, 1, 1, 2, 3, 5, 8, … dizisini verir.

Fibonacci dizisi · yineleme bağıntısı · döngü ve özyineleme

DERS BÖLÜMLERİ **Bölüm 4.1 · Algoritmalar** **Bölüm 8 · Döngüler** **Bölüm 12 · Özyineleme**

Önemi: $F(n) = F(n-1) + F(n-2)$ döngü alıştırmalarının klasiğidir. Saf özyinelemenin aynı işi neden tekrar tekrar yaptığını da bu dizi gösterir.

Dizi Hint matematiğinde daha önce de biliniyordu. “Fibonacci dizisi” adını 19. yüzyılda Édouard Lucas yaygınlaştırdı. Terimler o kadar hızlı büyür ki Java'nın `long` tipi bile 92. terimden sonra taşar.

GL

Gottfried Wilhelm Leibniz

1646–1716 · Alman · çok yönlü bilgin, Hannover

İkili aritmetik · 1703 · kademeli hesap makinesi

Yalnız 0 ve 1 kullanan ikili (*binary*) aritmetiği 1703'te yayımladı. Kademeli hesap makinesi (*stepped reckoner*) adlı mekanik bir hesap makinesi de yaptı.

ikili sayılar · mekanik hesaplama · simgesel akıl yürütme

DERS BÖLÜMLERİ **Bölüm 1.1 · Temel Kavramlar** **Bölüm 1.3 · Bilgisayarlar**

Önemi: Bilgisayardaki her sayı, karakter ve komut, Leibniz'in tanımladığı ikili sistemde saklanır.

“Explication de l'arithmétique binaire” adlı yazısı Paris Bilimler Akademisinin yıllığında çıktı. Leibniz, akıl yürütmenin hesapla yapılabileceği biçimsel bir dil de hayal ediyordu.

JJ

Joseph Marie Jacquard

1752–1834 · Fransız · dokumacı ve mucit, Lyon

Delikli kartlı dokuma tezgâhı · 1804

Desenleri kendiliğinden dokuyan bir tezgâh düzeneği yaptı (ilk gösterimi 1804 dolayları). Düzeneği bir delikli kart (*punched card*) zinciri yönetiyordu.

delikli kartlar · saklanan komutlar · otomasyon

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme**

Önemi: Kartlar değişince makine aynı kalır, desen değişir. Bu, programı donanımdan ayrı tutmanın erken bir biçimidir.

Babbage, Analitik Makine'ye Jacquard tipi kartlarla girdi vermeyi planladı. Delikli kartlar daha sonra 1970'lere kadar veri ve program taşıdı.

CG

Carl Friedrich Gauss

1777–1855 · Alman · Göttingen Üniversitesi

1 + 2 + … + n = n(n + 1)/2 toplamı

Bilinen (belki de abartılmış) bir öyküye göre okul çağındayken 1'den 100'e kadar olan sayıların toplamını hemen buldu. Bunun için terimleri baştan ve sondan eşleştirdi.

kapalı biçimli formül | verimlilik | döngü ve formül

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme**

Önemi: Derste toplama döngüsü (n toplama) Gauss formülüyle (bir çarpma ve bir bölme) karşılaştırılır. Böylece algoritmaların maliyetinin farklı olabileceği görülür.

Gauss tarihin en önemli matematikçilerinden biridir. Sayılar kuramı, astronomi, jeodezi ve manyetizma üzerine çalıştı. Gauss eleme yöntemi ve Gauss (normal) dağılımı onun adını taşır.

CB

Charles Babbage

1791–1871 · İngiliz · Cambridge Üniversitesi (Lucasian Professor)

Fark Makinesi · Analitik Makine (1837)

Tablo hesaplamak için Fark Makinesi'ni (*Difference Engine*) tasarladı. Ardından 1837'de genel amaçlı mekanik bir bilgisayar olan Analitik Makine'yi (*Analytical Engine*) tasarladı; bu makine hiç tamamlanmadı.

depo ve değirmen | delikli kartla girdi | genel amaçlı makine

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme**

Önemi: Analitik Makine belleği (“depo”, *store*) işlemeden (“değirmen”, *mill*) ayırıyordu. Bugünkü bilgisayarlar da bu düzeni kullanır.

Tasarımda delikli kartlarla yönetilen koşullu dallanma ve döngüler vardı. Londra'daki Science Museum, Fark Makinesi No. 2'nin çalışan bir örneğini onun çizimlerinden 1991'de tamamladı.

AM

Augustus De Morgan

1806–1871 · İngiliz · University College London

De Morgan kuralları · *Formal Logic*, 1847

Bir VE ifadesinin değilinin, terimlerin değillerinin VEYA'sına eşit olduğunu ve bunun tersini kural olarak yazdı. Kurallar 1846'daki bir makalesinde çıktı, 1847'de *Formal Logic* kitabında yeniden yayımlandı.

De Morgan kuralları | mantıksal işleçler | koşulun tersi

DERS BÖLÜMLERİ **Bölüm 7 · Koşullu İfadeler**

Önemi: Java'da `!(a && b)` ile `!a || !b`, `!(a || b)` ile de `!a && !b` her zaman aynı değeri verir. Bir koşulun tersi yazılırken `!` parantezin içine dağıtılır ve `&&` ile `||` yer değiştirir: `!(s >= 15 && s <= 30)` koşulu `s < 15 || s > 30` olur.

Hindistan'ın Madurai kentinde doğdu; University College London'da uzun yıllar matematik profesörlüğü yaptı. 1840–1842 yıllarında Ada Lovelace'a mektupla matematik dersi verdi. 1865'te kurulan London Mathematical Society'nin ilk başkanı oldu.

GB

George Boole

1815–1864 · İngiliz · Queen's College Cork, İrlanda

Boole cebiri · 1847, 1854

Mantığı doğru/yanlış değerleri üzerinde bir cebir olarak ele aldı. Bu fikri *The Mathematical Analysis of Logic* (1847) ve *The Laws of Thought* (1854) adlı kitaplarında geliştirdi.

AND / OR / NOT · doğruluk tabloları · boolean tipi

DERS BÖLÜMLERİ **Bölüm 1.3 · Bilgisayarlar** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler** **Bölüm 7 · Koşullu İfadeler**

Önemi: İşlemcideki mantık kapıları da `if` deyimlerindeki ve döngülerdeki koşullar da Boole ifadeleridir. Java'nın `boolean` tipi onun adını taşır.

Boole büyük ölçüde kendi kendini yetiştirdi ve Queen's College Cork'un ilk matematik profesörü oldu. Claude Shannon daha sonra Boole cebirinin anahtarlama devrelerini betimlediğini gösterdi.

AL

Ada Lovelace

1815–1852 · İngiliz · matematikçi, Londra

Analitik Makine notları · 1843

Babbage'ın Analitik Makine'si üzerine bir makaleyi çevirdi ve 1843'te ona uzun notlar ekledi. Notlarda Bernoulli sayılarını adım adım hesaplayan bir yöntem de vardı.

bir makine için yayımlanan ilk algoritma · Note G · simge işleme

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 4.3 · Programlamaya Giriş**

Önemi: G notu (*Note G*) çoğu zaman ilk bilgisayar programı sayılır. Lovelace, böyle bir makinenin yalnız sayıları değil simgeleri de işleyebileceğini fark etti.

Şair Lord Byron'ın kızıydı. Ada programlama dili (1980) onun adını taşır. Ada Lovelace Günü de bilim ve teknolojideki kadınları kutlar.

ÉL

Édouard Lucas

1842–1891 · Fransız · matematikçi, Paris

Hanoi kuleleri bulmacası · 1883

Hanoi kuleleri (*Tower of Hanoi*) bulmacasını icat etti ve 1883'te "N. Claus de Siam" takma adıyla piyasaya sürdü.

Hanoi kuleleri | $2^n - 1$ hamle | özyinelemeli düşünme

DERS BÖLÜMLERİ **Bölüm 12 · Özyineleme**

Önemi: n diski taşımak, n – 1 diski iki kez taşımaya indirgenir. Özyinelemeli çözümü döngülü çözümden çok daha basit olan problemlerin ders kitabı örneği budur.

Lucas, Fibonacci sayılarını da inceledi ve onlara bu adı verdi. Ona bağlı Lucas sayıları ve asal sayı testleri üzerine de çalıştı.

HH

Herman Hollerith

1860–1929 · Amerikalı · Tabulating Machine Company

Delikli kartlı sayım makinesi · 1890 ABD nüfus sayımı

Delikli kartlara kaydedilen verileri sayan elektromekanik makineler yaptı. Bu makineler 1890 ABD nüfus sayımının verilerini işledi.

veri işleme | delikli kartlar | sayım makineleri

DERS BÖLÜMLERİ **Bölüm 1.1 · Temel Kavramlar**

Önemi: Sayım makineleri veri işlemeyi bir sektöre dönüştürdü: girdi, işleme ve çıktı artık makinelerle yapılıyordu.

Kurduğu Tabulating Machine Company (1896), 1911'de Computing-Tabulating-Recording Company çatısında birleşti. Bu şirketin adı 1924'te International Business Machines (IBM) oldu.

AC

Alonzo Church

1903–1995 · Amerikalı · Princeton University, UCLA

Lambda hesabı · Church–Turing tezi

Lambda hesabını (*lambda calculus*) geliştirdi. 1936'da bu araçla mantığın karar probleminin (*Entscheidungsproblem*) genel bir çözümü olmadığını gösterdi.

lambda hesabı · hesaplanabilirlik · karar verilemezlik

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme**

Önemi: Church–Turing tezine göre etkili bir yöntemle hesaplanabilen her şey bir Turing makinesiyle de hesaplanabilir. Lambda hesabının hesaplama gücü de Turing makinesininkiyle aynıdır.

Church sonuca Turing'den kısa süre önce ve farklı bir yoldan ulaştı. Turing'in Princeton'daki doktorasını (1938) o yönetti. Lambda hesabı, fonksiyonel programlamanın ve bugünkü dillerdeki lambda ifadelerinin temelidir.

JvN

John von Neumann

1903–1957 · Macar asıllı Amerikalı · Institute for Advanced Study, Princeton

Depolanan program mimarisi · 1945 · birleştirme sıralaması

Program ile verinin aynı bellekte durduğu depolanan program (*stored program*) bilgisayarını tanımladı. Tanım, 1945 tarihli *First Draft of a Report on the EDVAC* raporundadır.

von Neumann mimarisi · getir–çöz–yürüt · birleştirme sıralaması

DERS BÖLÜMLERİ **Bölüm 1.3 · Bilgisayarlar**

Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme **Bölüm 4.1 · Algoritmalar**

Bölüm 4.2 · Akış Diyagramları **Bölüm 12 · Özyineleme**

Önemi: Bilgisayar donanımı anlatılırken ele alınan işlemci–bellek düzeni ve “von Neumann darboğazı” onun adını taşır.

Birleştirme sıralaması (*merge sort*, 1945) ona atfedilir. Herman Goldstine ile programları planlamak için akış diyagramlarını tanıttı (1947). Oyun kuramına, kuantum mekaniğine ve sayısal analize de büyük katkılar yaptı.

TF

Tommy Flowers

1905–1998 · İngiliz · Post Office Research Station, Dollis Hill

Colossus · 1944

Vakum tüpleriyle (*vacuum tube*) çalışan elektronik bir makine olan Colossus'u tasarlayıp yaptı. Colossus, 1944 başından itibaren Bletchley Park'ta Alman Lorenz şifrelerinin kırılmasına yardım etti.

vakum tüpleri | elektronik hesaplama | şifre kırma

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme**

Önemi: Colossus, çok sayıda tüpün dijital hesaplamada güvenilir biçimde çalışabileceğini gösterdi. Anahtar ve fişlerle programlanıyordu ama programını bellekte saklamıyordu.

Colossus Mark 1'de yaklaşık 1.600, Mark 2'de yaklaşık 2.400 tüp vardı. Makineler on yıllarca gizli tutuldu. Bu yüzden Flowers'ın emeği 1970'lere kadar pek bilinmedi.

GH

Grace Hopper

1906–1992 · Amerikalı · ABD Donanması (tuğamiral), Remington Rand

A-0 derleyicisi · FLOW-MATIC · COBOL

İlk derleyicilerden (*compiler*) biri olan A-0 sistemini 1952'de geliştirdi. İngilizceye benzeyen FLOW-MATIC dilini de o yarattı; bu dil COBOL'u (1959) güçlü biçimde etkiledi.

derleyiciler | COBOL | İngilizceye benzeyen söz dizimi

DERS BÖLÜMLERİ **Bölüm 4.3 · Programlamaya Giriş**

Önemi: Programların insanların okuyabildiği dillerde yazılıp otomatik olarak çevrilmesini savundu. Dersteki her derleyicinin arkasında bu fikir vardır.

İkinci Dünya Savaşı sırasında Harvard Mark I'i programladı. Ekibinin 1947'de kayıt defterine yapıştırdığı güve, bilgisayar "böcekleri" (*bug*) üzerine ünlü bir öyküdür; sözcük aslında daha önce de kullanılıyordu. 2016'da ölümünden sonra Presidential Medal of Freedom (ABD Başkanlık Özgürlük Madalyası) ile onurlandırıldı.

ME

John Mauchly & J. Presper Eckert

1907–1980 · 1919–1995 · Amerikalı · Moore School, University of Pennsylvania

ENIAC · 1945–46 · UNIVAC I

Genel amaçlı elektronik bilgisayar ENIAC'ın tasarımını yönettiler. ENIAC 1945'te tamamlandı ve Şubat 1946'da tanıtıldı.

ENIAC · vakum tüpleri · UNIVAC

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme**

Önemi: ENIAC büyük ölçekli elektronik hesaplamanın mümkün olduğunu gösterdi. Kablolarla yeniden programlamanın zorluğu da doğrudan depolanan program tasarımına yol açtı.

ENIAC'ta yaklaşık 18.000 vakum tüpü vardı. Makine anahtarlar ve kablolar ayarlanarak programlanıyordu. Kurdukları şirket UNIVAC I'i yaptı; ilk makine 1951'de ABD Nüfus Sayım Bürosuna teslim edildi.

BBS

John Bardeen, Walter Brattain & William Shockley

1908–1991 · 1902–1987 · 1910–1989 · Amerikalı · Bell Labs

Transistör · 1947

Bardeen ve Brattain, ilk nokta temaslı transistörü (*transistor*) Aralık 1947'de Bell Labs'te yaptı. Kısa süre sonra Shockley eklem transistörünü tasarladı.

Ödüller: 1956 Nobel Fizik Ödülü (üçü birlikte); Bardeen 1972'de ikinci kez Nobel Fizik Ödülü aldı

transistör · yarı iletkenler · anahtarlar

DERS BÖLÜMLERİ **Bölüm 1.1 · Temel Kavramlar** **Bölüm 1.3 · Bilgisayarlar**

Önemi: Transistör küçücük bir elektronik anahtardır. Mantık kapıları ve işlemciler milyonlarca, hatta milyarlarca transistörden kurulur.

Transistörler daha küçük, daha hızlı, daha az ısınan ve daha güvenilir oldukları için vakum tüplerinin yerini aldı. Shockley'nin sonradan kurduğu şirket Silikon Vadisi'nin doğuşuna katkı verdi. Bu şirketten ayrılan bazı mühendisler Fairchild Semiconductor'ı kurdu.

KZ

Konrad Zuse

1910–1995 · Alman · inşaat mühendisi, Berlin

Z3 · 1941 · Plankalkül

Rölelerle kurulan, programlanabilir ve tam otomatik dijital bilgisayar Z3'ü 1941'de tamamladı. İlk yüksek düzeyli programlama dillerinden Plankalkül'ü de tasarladı.

ikili kayan nokta · röleler · Plankalkül

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme**

Bölüm 4.3 · Programlamaya Giriş

Önemi: Z3 ikili aritmetik ve kayan noktalı (*floating-point*) sayılar kullanıyordu. Bu fikirler veri gösteriminin ve Java'daki `double` tipinin merkezindedir.

Z3 programını delikli filmden okuyordu ve 1943'teki bir hava saldırısında yok oldu. Plankalkül 1942–1945 arasında tasarlandı ama ancak 1972'de yayımlandı.

AT

Alan Turing

1912–1954 · İngiliz · University of Cambridge, Bletchley Park, University of Manchester

Turing makinesi · 1936

Turing makinesini (*Turing machine*) "On Computable Numbers" (1936) makalesinde tanımladı. Bazı problemlerin hiçbir algoritmayla çözülemeyeceğini kanıtladı.

Turing makinesi · evrensel makine · hesaplanabilirlik

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme**

Önemi: Evrensel Turing makinesi, başka bir makinenin kurallarını kendi bandından okur. Depolanan program fikrinin kuramsal biçimi budur.

İkinci Dünya Savaşı'nda Bletchley Park'ta Enigma şifresini kırma çalışmalarına öncülük etti. ACE bilgisayarını tasarladı (1945) ve Turing testini önerdi (1950). İlk kez 1966'da verilen ACM Turing Ödülü (Turing Award) onun adını taşır.

HG

Herman Goldstine

1913–2004 · Amerikalı · ENIAC projesi, Institute for Advanced Study

Akış diyagramları · 1947

Von Neumann ile birlikte "Planning and Coding of Problems for an Electronic Computing Instrument" (1947–48) raporlarını yazdı. Bu raporlar program tasarımı için akış diyagramlarını (*flowchart*) tanıttı.

akış diyagramları · program planlama · IAS makinesi

DERS BÖLÜMLERİ **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 4.2 · Akış Diyagramları**

Önemi: Algoritmalar konusunda kullanılan akış diyagramlarının kutuları ve okları bu raporlardan gelir.

Ordu subayı olarak ENIAC projesinin irtibat görevlisiydi ve 1944'te von Neumann'ı projeye o kattı. Daha sonra IAS bilgisayar projesini yönetti. *The Computer from Pascal to von Neumann* (1972) adlı bir bilişim tarihi de yazdı.

CS

Claude Shannon

1916–2001 · Amerikalı · MIT, Bell Labs

Anahtarlama devreleri · 1937 · bilgi kuramı · 1948

1937 tarihli yüksek lisans tezinde Boole cebirinin röleli anahtarlama devrelerini betimleyip sadeleştirebileceğini gösterdi. 1948'de bilgi kuramını (*information theory*) kurdu.

mantık devreleri · bit · bilgi kuramı

DERS BÖLÜMLERİ **Bölüm 1.1 · Temel Kavramlar** **Bölüm 1.3 · Bilgisayarlar**

Önemi: Tezi, Boole mantığını işlemcilerdeki mantık kapılarına bağlar. 1948 makalesi de biti bilginin birimi yaptı.

"Bit" (ikili basamak, *binary digit*) sözcüğü 1948 makalesinde geçer; Shannon sözcüğü orada John Tukey'ye atfeder. Veri sıkıştırma ve ağlar üzerinden güvenilir iletişim bilgi kuramına dayanır.

BB

Bob Bemer

1920–2004 · Amerikalı · IBM ve başka şirketler

ASCII · kaçış dizisi · ters eğik çizgi

İlk kez 1963'te yayımlanan ASCII'nin oluşturulmasında öncü rol oynadı. Kaçış dizisi (*escape sequence*) ve ters eğik çizgi (\) karakteri de onun katkılarıdır.

ASCII · karakter kodları · kaçış dizileri

DERS BÖLÜMLERİ **Bölüm 1.1 · Temel Kavramlar** **Bölüm 1.3 · Bilgisayarlar**

Önemi: ASCII'nin 7 bitlik kodları (A = 65), Unicode'un ilk 128 kod noktasıdır. Derste kullanılan karakter aritmetiği bu kodlara dayanır.

ASCII bir komite çalışmasıydı ama Bemer çoğu zaman onun babası sayılır. Ters eğik çizgiyi 1961'de önerdi. 2000 yılı tarih sorununa ilk dikkat çekenlerden biri de oydu.

BJ

Corrado Böhm & Giuseppe Jacopini

1923–2017 (Böhm) · İtalyan · Roma (IAC-CNR); Jacopini, Böhm'ün öğrencisiydi

Yapısal program teoremi · 1966

Akış diyagramıyla yazılan her programın yalnız sıralı yapı, seçim ve tekrarla yeniden yazılabileceğini kanıtladılar (Communications of the ACM, 1966).

sıralı yapı · seçim · tekrar

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 4.2 · Akış Diyagramları** **Bölüm 7 · Koşullu İfadeler** **Bölüm 8 · Döngüler**

Önemi: Sıralı, seçimli ve döngülü üç kontrol yapısı her algoritma için yeterlidir; `goto` türü atlamalara gerek yoktur.

Bu sonuç, Dijkstra'nın 1968'den itibaren savunduğu yapısal programlamanın (*structured programming*) kuramsal temelini oluşturdu. Böhm daha sonra lambda hesabı üzerine de önemli çalışmalar yaptı.

KN

Jack Kilby & Robert Noyce

1923–2005 · 1927–1990 · Amerikalı · Texas Instruments (Kilby); Fairchild Semiconductor, Intel (Noyce)

Tümleşik devre · 1958–59

Tümleşik devreyi (*integrated circuit*) birbirinden bağımsız olarak buldular. Kilby ilk örneği 1958'de Texas Instruments'ta gösterdi. Noyce 1959'da Fairchild'da uygulamaya elverişli düzlemsel silikon biçimini geliştirdi.

Ödüller: Kilby: 2000 Nobel Fizik Ödülü (Noyce 1990'da ölmüştü)

tümleşik devre · yongalar · küçültme

DERS BÖLÜMLERİ **Bölüm 1.2 · Bilgisayar Donanımı** **Bölüm 1.3 · Bilgisayarlar**

Önemi: Çok sayıda transistörü tek bir yongaya sığdırmak işlemcileri, belleği ve bugünkü bilgisayarları mümkün kıldı.

Noyce, Intel'i 1968'de Gordon Moore ile birlikte kurdu. Kilby'nin ekibi ilk elde taşınan elektronik hesap makinesini de geliştirdi (1967).

JB

John Backus

1924–2007 · Amerikalı · IBM

FORTRAN · 1957 · Backus–Naur biçimi

IBM'de FORTRAN'ı (1957) geliştiren ekibi yönetti. FORTRAN, kod iyileştiren derleyicisiyle geniş kitlelerce kullanılan ilk yüksek düzeyli dildir.

Ödüller: 1977 Turing Ödülü

FORTRAN · derleyiciler · BNF dilbilgisi

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme**

Bölüm 4.3 · Programlamaya Giriş

Önemi: FORTRAN, derlenmiş kodun elle yazılmış makine koduna yakın hızda çalışabileceğini gösterdi. Yüksek düzeyli diller bu sayede pratikte kullanılır hâle geldi.

Dillerin söz dizimini yazmak için bir gösterim önerdi. Peter Naur bu gösterimi ALGOL 60 raporunda kullandı. Backus–Naur biçimi (*Backus–Naur Form*, BNF) adıyla bilinen gösterim, dil dilbilgilerini tanımlamak için bugün de kullanılır.

DE

Douglas Engelbart

1925–2013 · Amerikalı · Stanford Research Institute (SRI)

Bilgisayar faresi · "Mother of All Demos", 1968

Bilgisayar faresini icat etti; ilk fareyi 1960'ların ortasında Bill English ile birlikte yaptı. 1968'de pencereleri, hiper metni (*hypertext*) ve aynı anda ortak belge düzenlemeyi canlı olarak gösterdi.

Ödüller: 1997 Turing Ödülü

fare · hiper metin · etkileşimli bilişim

DERS BÖLÜMLERİ **Bölüm 1.2 · Bilgisayar Donanımı** **Bölüm 2.1 · Yazılım ve İşletim Sistemleri** **Bölüm 2.2 · Windows İşletim Sistemi**

Önemi: Windows gibi grafik masaüstlerindeki işaretleme ve pencere fikirleri onun çalışmalarına dayanır.

Aralık 1968'de San Francisco'da yaptığı gösteri sonradan "Mother of All Demos" (Tüm Gösterilerin Anası) diye anıldı. Amacı, bilgisayarları insan zekâsını güçlendirmek için kullanmaktı.

BD

Paul Baran & Donald Davies

1926–2011 · 1924–2000 · Polonya doğumlu Amerikalı · RAND (Baran); Galli · National Physical Laboratory (Davies)

Paket anahtarlama · 1960'lar

Paket anahtarlamayı (*packet switching*) birbirinden bağımsız olarak geliştirdiler. Bu yöntemde ileti küçük bloklara bölünür. Bloklar ağda ayrı ayrı yol alır ve hedefte yeniden birleştirilir.

paket anahtarlama · dağıtık ağlar · yönlendirme

DERS BÖLÜMLERİ **Bölüm 1.4 · İnternet**

Önemi: İnternetteki her ileti birbirinden bağımsız paketler hâlinde taşınır.

Baran dağıtık ağları 1964'te bir dizi RAND raporunda anlattı. Davies "paket" (*packet*) terimini ortaya attı ve İngiltere'deki NPL'de paket anahtarlamalı bir ağ kurdu. İkisinin çalışmaları ARPANET'in (1969) tasarımını etkiledi.

KK

John Kemeny & Thomas Kurtz

1926–1992 · 1928–2024 · Macar doğumlu Amerikalı; Amerikalı · Dartmouth College

BASIC · 1964

BASIC dilini ve Dartmouth Zaman Paylaşımlı Sistemini (*Dartmouth Time-Sharing System*) geliştirdiler. İlk BASIC programı bu sistemde 1 Mayıs 1964'te çalıştı.

BASIC | zaman paylaşımı | herkes için programlama

DERS BÖLÜMLERİ **Bölüm 4.3 · Programlamaya Giriş**

Önemi: BASIC, fen dışı alanlardaki öğrenciler de programlama öğrenebilsin diye tasarlandı. Ev bilgisayarı kullanan pek çok kişinin ilk dili oldu.

Microsoft'un ilk ürünü bir BASIC yorumlayıcısıydı (1975). Kemeny 1970–1981 arasında Dartmouth'un rektörlüğünü yaptı. Kurtz ile birlikte daha sonra True BASIC şirketini kurdular.

JM

John McCarthy

1927–2011 · Amerikalı · MIT, Stanford University

Lisp · 1958 · "yapay zekâ"

Özyinelemeli fonksiyonlar üzerine kurulu Lisp dilini tasarladı (1958). "Yapay zekâ" (*artificial intelligence*) terimini 1955 tarihli Dartmouth önerisinde ortaya attı.

Ödüller: 1971 Turing Ödülü

özyineleme | koşullu ifadeler | Lisp

DERS BÖLÜMLERİ **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 7 · Koşullu İfadeler** **Bölüm 12 · Özyineleme**

Önemi: Lisp, özyinelemeyi (*recursion*) sıradan bir programlama aracı yaptı. `if-else` yapısının atası olan koşullu ifadeyi de bu dil getirdi.

Lisp'i "Recursive Functions of Symbolic Expressions and Their Computation by Machine" (1960) makalesinde tanımladı. Çöp toplamayı (*garbage collection*) da o buldu; zaman paylaşımını da o önerdi.

SP

Seymour Papert

1928–2016 · Güney Afrika doğumlu · MIT

Logo · *Mindstorms* (1980)

Çocuklar için Logo dilini birlikte geliştirdi (1967). "Hesaplamalı düşünme" (*computational thinking*) ifadesini *Mindstorms* (1980) kitabında kullandı.

Logo · programlayarak öğrenme · hesaplamalı düşünme

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme**

Önemi: Programlamanın problemler üzerine bir düşünme biçimi olduğunu savundu. Jeannette Wing bu fikri daha sonra geliştirdi.

Logo'nun kaplumbağa grafiklerinde öğrenci adım adım komutlar verir ve sonucu hemen görür. LEGO Mindstorms robot setleri adını onun kitabından alır.

GM

Gordon Moore

1929–2023 · Amerikalı · Fairchild Semiconductor, Intel

Moore yasası · 1965

1965'te bir yongadaki bileşen sayısının her yıl ikiye katlandığını gözlemledi. 1975'te bu hızı yaklaşık iki yılda bir olarak düzeltti.

Moore yasası · transistör sayısı · yarı iletkenler

DERS BÖLÜMLERİ **Bölüm 1.1 · Temel Kavramlar** **Bölüm 1.2 · Bilgisayar Donanımı**

Önemi: Donanım sunumunda anlatılan işlemci gücü ve bellek kapasitesindeki düzenli artışı Moore yasası açıklar.

Intel'i 1968'de Robert Noyce ile birlikte kurdu. Eğilim on yıllarca sürdü. Transistörler fiziksel sınırlara yaklaştıkça artış yavaşladı.

ED

Edsger W. Dijkstra

1930–2002 · Hollandalı · Mathematisch Centrum, Eindhoven University of Technology, University of Texas at Austin

Yapısal programlama · "Go To Statement Considered Harmful", 1968

1968'de sınırsız `goto` atlamalarının programları anlaşılmaz kıldığını savundu. Bu yazı yapısal programlama akımını başlattı.

Ödüller: 1972 Turing Ödülü

yapısal programlama · en kısa yollar · semaforlar

DERS BÖLÜMLERİ **Bölüm 2.1 · Yazılım ve İşletim Sistemleri** **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 4.2 · Akış Diyagramları** **Bölüm 8 · Döngüler**

Önemi: Java'da olduğu gibi programları açık bir sıralı yapı, seçim ve döngülerle yazmak onun fikirlerini izler.

En kısa yol algoritmasını 1959'da yayımladı. Süreçleri eşgüdümlemek için semaforları (*semaphore*) tanıttı. "Yemek yiyen filozoflar" (*dining philosophers*) problemi, işletim sistemleri sunumunun konularından kilitlenmeyi (*deadlock*) örnekler.

WK

William Kahan

d. 1933 · Kanadalı · University of California, Berkeley

IEEE 754 kayan nokta · 1985

Kayan nokta aritmetiği için IEEE 754 standardının (1985) baş mimarıydı.

Ödüller: 1989 Turing Ödülü

kayan nokta · yuvarlama · double

DERS BÖLÜMLERİ **Bölüm 1.3 · Bilgisayarlar** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler**

Önemi: Java'nın `float` ve `double` tipleri IEEE 754'e uyar. `0.1 + 0.2` işleminin `0.30000000000000004` yazdırmasının nedeni budur.

Standart; işaret, üs ve kesir bitlerinden oluşan biçimleri, yuvarlama kurallarını ve sonsuz, NaN gibi özel değerleri tanımlar. Böylece aynı program farklı makinelerde aynı sonucu verir.

TH

Tony Hoare

1934–2026 · İngiliz · Elliott Brothers, Queen's University Belfast, University of Oxford, Microsoft Research

Quicksort · 1959–61

Hızlı sıralamayı (*quicksort*) 1959–60 dolaylarında buldu ve 1961'de yayımladı. Algoritma diziyi bir pivotun çevresinde iki parçaya ayırır, sonra parçaları özyinelemeli olarak sıralar.

Ödüller: 1980 Turing Ödülü

hızlı sıralama · böl ve fethet · Hoare mantığı

DERS BÖLÜMLERİ **Bölüm 4.1 · Algoritmalar** **Bölüm 12 · Özyineleme**

Önemi: Hızlı sıralama da birleştirme sıralaması gibi bir böl ve fethet (*divide and conquer*) algoritmasıdır. Ortalama çalışma süresi O(n log n)'dir; büyük dizilerde kabarcık sıralamasından çok daha hızlıdır.

Programların doğruluğunu kanıtlamak için Hoare mantığını (1969), eş zamanlılık için CSP'yi geliştirdi. ALGOL W'ya null referansını o ekledi (1965); bunu sonradan "milyar dolarlık hatam" diye niteledi. 5 Mart 2026'da öldü.

NW

Niklaus Wirth

1934–2024 · İsviçreli · ETH Zürih

Pascal · 1970 · *Algorithms + Data Structures = Programs*

Pascal'ı (1970) yapısal programlama öğretmek için tasarladı. Ardından Modula-2 ve Oberon dillerini geliştirdi.

Ödüller: 1984 Turing Ödülü

Pascal | yapısal programlama | adım adım iyileştirme

DERS BÖLÜMLERİ **Bölüm 4.3 · Programlamaya Giriş**

Önemi: Pascal, yapısal ve tip denetimi güçlü programlamayı yirmi yıl boyunca giriş derslerinin standardı yaptı.

Algorithms + Data Structures = Programs (Algoritmalar + Veri Yapıları = Programlar, 1976) kitabının adı bu dersin ana fikirlerinden birini özetler. Adım adım iyileştirmeyi (*stepwise refinement*) de savundu: problem daha küçük adımlara bölünerek çözülür.

EC

Edward G. Coffman Jr.

d. 1934 · Amerikalı · Pennsylvania State University, Bell Laboratories, Columbia University

Kilitlenmenin dört koşulu · 1971

M. J. Elphick ve Arie Shoshani ile yazdığı "System Deadlocks" (*Computing Surveys*, 1971) makalesinde kilitlenmenin (*deadlock*) oluşması için aynı anda sağlanması gereken dört koşulu tanımladı. Bu koşullar bugün Coffman koşulları diye anılır.

kilitlenme | Coffman koşulları | kaynak yönetimi

DERS BÖLÜMLERİ **Bölüm 2.1 · Yazılım ve İşletim Sistemleri**

Önemi: Karşılıklı dışlama, tut ve bekle, ön alınamazlık ve döngüsel bekleme birlikte sağlanmadıkça kilitlenme oluşmaz; birini kırmak yeter. Yazıcıyı tutup dosyayı bekleyen bir süreçle dosyayı tutup yazıcıyı bekleyen bir süreç döngüsel beklemenin en sade örneğidir. Bütün süreçlerin kaynakları aynı sırayla istemesi bu halkanın kurulmasını engeller.

1973'te Peter J. Denning ile *Operating Systems Theory* kitabını yazdı. Çizelgeleme, kutu yerleştirme (*bin packing*) ve kuyruk kuramı üzerine çalıştı; ACM'nin Symposium on Operating Systems Principles konferansının kurucuları arasındadır.

HFS

Ted Hoff, Federico Faggin & Masatoshi Shima

d. 1937 · d. 1941 · d. 1943 · Amerikalı; İtalyan asıllı Amerikalı; Japon · Intel ve Busicom

Intel 4004 mikroişlemcisi · 1971

Stanley Mazor ile birlikte, tek yongalı ilk ticari mikroişlemci (*microprocessor*) olan Intel 4004'ü (1971) geliştirdiler.

mikroişlemci · tek yongada CPU · 4 bit

DERS BÖLÜMLERİ **Bölüm 1.2 · Bilgisayar Donanımı** **Bölüm 1.3 · Bilgisayarlar**

Önemi: Merkezî işlem biriminin (CPU) tamamını tek yongaya sığdırmak kişisel bilgisayarlara, telefonlara ve bugünkü işlemcilere giden yolu açtı.

Mimariyi Hoff önerdi; Busicom'dan Shima mantık tasarımında çalıştı; silikon tasarımını Faggin yönetti. 4004'te yaklaşık 2.300 transistör vardı. Faggin ve Shima daha sonra Zilog Z80'i tasarladı.

DK

Donald Knuth

d. 1938 · Amerikalı · Stanford University

***The Art of Computer Programming* (1968'den beri)**

Algoritma analizini *The Art of Computer Programming* dizisinde sistemli bir alana dönüştürdü. Dizinin ilk cildi 1968'de çıktı.

Ödüller: 1974 Turing Ödülü

algoritma analizi · büyük O · sıralama ve arama

DERS BÖLÜMLERİ **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 4.1 · Algoritmalar**

Önemi: Doğrusal ve ikili aramayı ya da kabarcık sıralaması ile birleştirme sıralamasını adım sayarak karşılaştırmak, onun standart hâle getirdiği yöntemdir.

Dizinin 3. cildi sıralama ve aramayı ayrıntılı işler. TeX dizgi sistemini (1978) de o yarattı ve "okuryazar programlama" (*literate programming*) yaklaşımını savundu.

AK

Alan Kay

d. 1940 · Amerikalı · Xerox PARC

Smalltalk · üst üste binen pencereler · Dynabook

1970'lerde Xerox PARC'ta Smalltalk dilini ve üst üste binen pencereli grafik arayüzü geliştiren ekibe önderlik etti.

Ödüller: 2003 Turing Ödülü

grafik kullanıcı arayüzü | nesne yönelimli programlama | Smalltalk

DERS BÖLÜMLERİ **Bölüm 2.1 · Yazılım ve İşletim Sistemleri** **Bölüm 2.2 · Windows İşletim Sistemi**

Önemi: Apple ve Microsoft'un sonradan yaygınlaştırdığı pencereli, simgeli ve menülü masaüstü bu çalışmalardan doğdu.

Smalltalk'taki yaklaşım için "nesne yönelimli" (*object-oriented*) terimi ona atfedilir; Smalltalk Java'yı da etkiledi. Dynabook tasarısı (1972), çocuklar için tablete benzeyen taşınabilir bir bilgisayar öngörüyordu.

RT

Dennis Ritchie & Ken Thompson

1941–2011 · d. 1943 · Amerikalı · Bell Labs

Unix · 1969 · C dili · 1972

Unix işletim sistemini 1969'dan başlayarak Bell Labs'te geliştirdiler. Ritchie 1972 dolaylarında C dilini geliştirdi; Unix bu dille yeniden yazıldı.

Ödüller: 1983 Turing Ödülü (ortak)

Unix | C dili | taşınabilirlik

DERS BÖLÜMLERİ **Bölüm 1.3 · Bilgisayarlar** **Bölüm 2.1 · Yazılım ve İşletim Sistemleri** **Bölüm 4.3 · Programlamaya Giriş**

Önemi: Linux, macOS ve Android Unix fikirlerini izler. Java'nın söz dizimi (süslü parantezler, `if`, `for`, `while`) de C'den gelir.

Thompson B dilini de yarattı ve Rob Pike ile UTF-8 kodlamasını tasarladı (1992). Google'da Go dilinin tasarımcılarından biri oldu (2009).

RT

Ray Tomlinson

1941–2016 · Amerikalı · BBN (Bolt Beranek and Newman)

Ağ üzerinden e-posta · 1971 · @ işareti

1971'de ARPANET'te bilgisayarlar arasındaki ilk e-postayı gönderdi. Kullanıcı adını bilgisayar adından ayırmak için @ işaretini seçti.

e-posta | @ işareti | ARPANET

DERS BÖLÜMLERİ **Bölüm 1.4 · İnternet**

Önemi: kullanıcı@bilgisayar biçimi bugün her e-posta adresinde kullanılır.

Daha önceki sistemlerde yalnız aynı bilgisayarın kullanıcıları birbirine ileti bırakabiliyordu. Tomlinson bunu ağ üzerine taşıdı. 2012'de Internet Hall of Fame'e (İnternet Onur Listesi) alındı.

BK

Brian Kernighan

d. 1942 · Kanadalı · Bell Labs, Princeton University

"hello, world" · *The C Programming Language* (1978)

Bilinen ilk "hello, world" programlarını Bell Labs'in B (1972) ve C (1974) eğitim belgelerinde yazdı. Bu programı Dennis Ritchie ile yazdığı kitap (1978) ünlendirdi.

hello, world | ilk program | programlama üslubu

DERS BÖLÜMLERİ **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 5 · Java'ya Giriş ve İlk Program**

Önemi: Yeni bir dile kısa bir selam yazdırarak başlamak onun başlattığı bir gelenektir. Java'daki ilk "Merhaba" programı da bu geleneği sürdürür.

AWK dilinin adındaki "K" odur. Programlama üslubu üzerine başkalarıyla birlikte yazdığı kitaplar çok etkili oldu. *The C Programming Language* kitabı çoğu zaman "K&R" diye anılır.

CK

Vint Cerf & Bob Kahn

d. 1943 · d. 1938 · Amerikalı · Stanford, DARPA

TCP/IP · 1974

Farklı ağları birbirine bağlayan TCP/IP protokollerini tasarladılar (makale 1974'te yayımlandı). ARPANET 1 Ocak 1983'te TCP/IP'ye geçti.

Ödüller: 2004 Turing Ödülü (ortak); 2005 ABD Presidential Medal of Freedom

TCP/IP · ağlar arası bağlantı · protokoller

DERS BÖLÜMLERİ **Bölüm 1.4 · İnternet**

Önemi: Katmanlı protokol slaytlarında görüldüğü gibi TCP/IP, ağları internette birleştiren "ortak kurallar kümesidir".

IP, paketleri adresleri kullanarak ağlar arasında taşır. TCP ise veriyi güvenilir biçimde ve sırasıyla teslim eder. İkisi de çoğu zaman "internetin babaları" diye anılır.

GA

Robert Gaskins & Dennis Austin

d. 1943 · 1947–2023 · Amerikalı · Forethought, Inc., sonra Microsoft

PowerPoint · 1987

Presenter adlı sunu programını Gaskins tasarladı, programlamasını Austin yönetti. Program 1987'de Macintosh için PowerPoint 1.0 adıyla çıktı.

sunu yazılımı · slaytlar · Forethought

DERS BÖLÜMLERİ **Bölüm 3.3 · Microsoft PowerPoint**

Önemi: Slayt tabanlı sunu yazılımı, PowerPoint ile standart bir ofis aracı oldu.

Microsoft, Forethought'u 1987'de yaklaşık 14 milyon dolara satın aldı. Bu, şirketin ilk satın almalarından biriydi. PowerPoint daha sonra Microsoft Office'in parçası oldu.

CS

Charles Simonyi

d. 1948 · Macar asıllı Amerikalı · Xerox PARC, Microsoft

Bravo düzenleyicisi · Microsoft Word

Xerox PARC'ta Butler Lampson ile birlikte Bravo'yu geliştirdi. Bravo, ekranda görülenin aynen basıldığı (WYSIWYG, *what you see is what you get*) ilk düzenleyicilerdendir. Simonyi daha sonra Microsoft Word'ün (1983) geliştirilmesini yönetti.

kelime işlemci · WYSIWYG · Microsoft Office

DERS BÖLÜMLERİ **Bölüm 3.1 · Microsoft Word**

Önemi: Belgeyi ekranda, basılacağı biçimiyle düzenlemek Word'ün temel fikridir. Bu fikir Bravo'dan gelir.

1981'de Microsoft'a katıldı ve uygulamalar grubunu yönetti. Multiplan ve Excel de bu grubun ürünleridir. Değişken adlandırmadaki "Macar gösterimi" (*Hungarian notation*) onun adına atıfla anılır. Uluslararası Uzay İstasyonu'na iki kez gitti (2007, 2009).

PM

Paul Mockapetris

d. 1948 · Amerikalı · USC Information Sciences Institute

Alan Adı Sistemi (DNS) · 1983

Alan Adı Sistemini (*Domain Name System*, DNS) tasarladı; sistem RFC 882 ve 883'te (Kasım 1983) tanımlandı. DNS, www.example.edu gibi adları IP adreslerine çevirir.

DNS · alan adları · hiyerarşi

DERS BÖLÜMLERİ **Bölüm 1.4 · İnternet**

Önemi: Adım adım DNS çözümlemesi (kök sunucu, üst düzey alan, yetkili sunucu) onun tasarımını izler.

DNS, ağdaki her bilgisayarı listeleyen tek bir HOSTS.TXT dosyasının yerini aldı. Önbellek kullanan, dağıtık ve hiyerarşik bir veri tabanıdır. RFC'ler 1987'de RFC 1034 ve 1035 ile güncellendi.

BS

Bjarne Stroustrup

d. 1950 · Danimarkalı · Bell Labs, Texas A&M, Columbia University

C++ · 1983

C++ dilini yarattı. Dil 1979'da "C with Classes" (sınıflı C) olarak başladı, 1983'te C++ adını aldı ve 1985'te ticari olarak yayımlandı.

C++ · nesne yönelimli programlama · sınıflar

DERS BÖLÜMLERİ **Bölüm 4.3 · Programlamaya Giriş**

Önemi: C++, sınıfları ve nesneleri C türü söz dizimine taşıdı. Java bu yaklaşımı daha sade ve güvenli bir biçimde izledi.

Ad, C'nin artırma işleci `++`'yı kullanır; Java döngülerinde öğretilen işleç de budur. C++ sistem yazılımlarında, oyunlarda ve performansın önemli olduğu yazılımlarda yaygındır.

BF

Dan Bricklin & Bob Frankston

d. 1951 · d. 1949 · Amerikalı · Software Arts

VisiCalc · 1979

Kişisel bilgisayarlar için ilk elektronik tablo (*spreadsheet*) programı VisiCalc'ı 1979'da Apple II için geliştirdiler.

elektronik tablolar · hücreler ve formüller · çığır açan uygulama

DERS BÖLÜMLERİ **Bölüm 3.2 · Microsoft Excel**

Önemi: Excel'in temeli olan satırlar, sütunlar, hücre başvuruları ve otomatik yeniden hesaplama VisiCalc ile başladı.

Fikir, Bricklin'e Harvard Business School öğrencisiyken geldi; kodun çoğunu Frankston yazdı. Pek çok kişi yalnız VisiCalc'ı çalıştırmak için kişisel bilgisayar aldı. Bu yüzden VisiCalc ilk "çığır açan uygulama" (*killer app*) sayılır.

RS

Richard Stallman

d. 1953 · Amerikalı · MIT AI Lab, Free Software Foundation

GNU projesi · 1983 · GPL

Özgür ve Unix benzeri bir işletim sistemi yapmak için GNU projesini 1983'te duyurdu. 1985'te Free Software Foundation'ı (Özgür Yazılım Vakfı) kurdu.

özgür yazılım · GNU · GPL lisansı

DERS BÖLÜMLERİ **Bölüm 1.1 · Temel Kavramlar** **Bölüm 2.1 · Yazılım ve İşletim Sistemleri**

Önemi: Yazılım lisansları ve açık kaynak (*open source*) konuları, onun başlattığı özgür yazılım (*free software*) hareketine dayanır.

GNU Genel Kamu Lisansı (*GNU General Public License*, GPL, 1989) yazılımı kullanma, inceleme, değiştirme ve paylaşma hakkı verir. Tek koşul, değiştirilmiş sürümlerin de aynı koşullarla dağıtılmasıdır. Linux sistemleri Linux çekirdeğini GNU araçlarıyla birleştirir.

GA

Bill Gates & Paul Allen

d. 1955 · 1953–2018 · Amerikalı · Microsoft

Microsoft · 1975 · MS-DOS · Windows

Microsoft'u 1975'te kurdular. IBM PC için MS-DOS'u sağladılar (1981) ve 1985'te Windows 1.0'ı çıkardılar.

MS-DOS · Windows · Microsoft Office

DERS BÖLÜMLERİ **Bölüm 2.1 · Yazılım ve İşletim Sistemleri** **Bölüm 2.2 · Windows İşletim Sistemi**

Önemi: Derste kullanılan Windows ve Microsoft Office, kurdukları şirketin ürünleridir.

İlk ürünleri Altair 8800 için bir BASIC yorumlayıcısıydı. MS-DOS, Seattle Computer Products'tan satın alınan 86-DOS'a dayanıyordu. Office paketi ilk kez 1990 dolaylarında bir arada sunuldu.

TBL

Tim Berners-Lee

d. 1955 · İngiliz · CERN, MIT, University of Oxford

World Wide Web · 1989

World Wide Web'i (Web) 1989'da CERN'de önerdi. İlk web tarayıcısını, web sunucusunu, HTML'yi ve HTTP'yi o geliştirdi. İlk web sitesi 1991'de yayına girdi.

Ödüller: 2016 Turing Ödülü

WWW · HTML · HTTP ve URL'ler

DERS BÖLÜMLERİ **Bölüm 1.4 · İnternet**

Önemi: Web, internetin üzerinde çalışan bir hizmettir. URL'ler, HTTP ve tarayıcılar onun tasarımıdır.

CERN, web teknolojisini 1993'te herkesin ücretsiz kullanımına açtı; bu da Web'in hızla yayılmasını sağladı. Berners-Lee 1994'te World Wide Web Consortium'u (W3C) kurdu.

JG

James Gosling

d. 1955 · Kanadalı · Sun Microsystems

Java · 1995

Sun Microsystems'ta Java dilinin tasarımını yönetti. Java 1995'te “write once, run anywhere” (bir kez yazılan kod her yerde çalışır) sloganıyla yayımlandı.

Java · bytecode ve JVM · platform bağımsızlığı

DERS BÖLÜMLERİ **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 5 · Java'ya Giriş ve İlk Program** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler**

Önemi: Dersin programlama kısmında Java kullanılır. Kaynak kod `javac` ile bytecode'a (ara kod) derlenir; Java Sanal Makinesi (JVM) bu kodu her platformda çalıştırır.

Dil, 1991'de Sun'ın tüketici cihazlarına yönelik Green projesinde Oak adıyla doğdu. Oracle, Sun'ı 2010'da satın aldı ve Java'nın bakımını bugün o yürütür.

GvR

Guido van Rossum

d. 1956 · Hollandalı · CWI Amsterdam, sonra Google, Dropbox, Microsoft

Python · 1991

Okunabilir koda önem veren Python dilini yarattı. İlk sürüm Şubat 1991'de çıktı.

Python | yorumlayıcı | okunabilirlik

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme** **Bölüm 4.3 · Programlamaya Giriş** **Bölüm 4.1 · Algoritmalar**

Önemi: Derste Python, yorumlanan (bytecode ve sanal makine kullanan) dil örneğidir. Derlenen C ve hibrit Java ile karşılaştırılır.

Python blokları süslü parantezle değil girintiyle belirtir. Van Rossum dili 2018'e kadar "Benevolent Dictator For Life" (ömür boyu iyiliksever diktatör) unvanıyla yönetti.

JW

Jeannette Wing

d. 1956 · Amerikalı · Carnegie Mellon University, Columbia University

"Computational Thinking" · 2006

2006'da Communications of the ACM'de yayımlanan denemesinde hesaplamalı düşünmenin yalnız bilgisayar bilimcileri için değil, herkes için temel bir beceri olduğunu savundu.

ayrıştırma | soyutlama | algoritmik düşünme

DERS BÖLÜMLERİ **Bölüm 1.5 · Hesaplama ve Hesaplamalı Düşünme**

Önemi: Denemesi, hesaplamalı düşünme sunumunun kaynaklarından biridir. Sunumdaki dört bileşen ayrıştırma (*decomposition*), örüntü tanıma (*pattern recognition*), soyutlama (*abstraction*) ve algoritma tasarımıdır.

Biçimsel yöntemler üzerine de çalıştı. Liskov–Wing yerine koyma ilkesi (1994), nesne yönelimli tasarımın bir kuralıdır. Daha sonra ABD Ulusal Bilim Vakfında (NSF) bilgisayar bilimleri biriminin başına geçti; Microsoft Research'te başkan yardımcılığı yaptı.

LT

Linus Torvalds

d. 1969 · Fin asıllı Amerikalı · University of Helsinki, Linux Foundation

Linux çekirdeği · 1991 · Git

Linux çekirdeğini 1991'de Helsinki'de öğrenciyken başlattı ve açık kaynaklı yazılım olarak yayımladı.

Linux çekirdeği | açık kaynak | Git

DERS BÖLÜMLERİ **Bölüm 1.1 · Temel Kavramlar** **Bölüm 2.1 · Yazılım ve İşletim Sistemleri**

Önemi: Android'in, çoğu web sunucusunun ve süper bilgisayarların altında Linux çekirdeği (*kernel*) çalışır. İşletim sistemleri anlatılırken çekirdek ile tam bir dağıtım arasındaki fark da bu örnek üzerinden ele alınır.

Projeyi 25 Ağustos 1991'de "just a hobby" (sadece bir hobi) diyerek duyurdu. 2005'te bugün yazılım geliştirmede en yaygın sürüm kontrol sistemi olan Git'i yarattı.

UC

Joe Becker, Lee Collins & Mark Davis

Amerikalı · Xerox ve Apple · Unicode Consortium

Unicode · 1987–91

Her yazı sistemindeki her karaktere tek bir kod vermek için Unicode projesini 1987'de başlattılar. Tasarımı Becker'ın 1988 tarihli "Unicode 88" yazısı ortaya koydu.

Unicode · kod noktaları · UTF-8

DERS BÖLÜMLERİ **Bölüm 1.1 · Temel Kavramlar** **Bölüm 1.3 · Bilgisayarlar** **Bölüm 6 · Değişkenler, Veri Tipleri ve İşleçler** **Bölüm 11 · Dizgiler**

Önemi: Java'nın `char` ve `String` tipleri Unicode'a dayanır. Bu yüzden 'A' karakterinin kodu 65'tir; ğ (U+011F) gibi Türkçe harfler de saklanıp karşılaştırılabilir.

Becker ve Collins Xerox'ta, Davis Apple'da çalışıyordu. Unicode Consortium 1991'de kuruldu ve Unicode 1.0 aynı yıl çıktı. Ken Thompson ile Rob Pike'ın tasarladığı UTF-8 (1992) en yaygın kodlama oldu.

Ödül yılları, ödülün verildiği yılı gösterir; Turing Ödülü'nü ACM (Association for Computing Machinery) verir. Çalışmaları birlikte anlatılan kişiler ortak kartta toplanmıştır. Doğum yılı gösterilmeyen kartlarda sıralama, ana çalışmanın dönemine göre yapılır. Kişi kartlarında "d." doğum yılını, "yak." yaklaşık tarihi gösterir.