Bölüm 4.5: Soyut Sınıflar ve Arayüzler

Java ile Nesne Yönelimli Programlama

Soyutlama | Soyut sınıflar | Soyut metotlar | Arayüzler

implements ve çoklu arayüz | default, static, private metotlar

Lambda ve fonksiyonel arayüz | sealed, non-sealed, permits | Arayüz mü, soyut sınıf mı?

Tasarım desenleri

Sercan KÜLCÜ · Giresun Üniversitesi · Elektrik-Elektronik Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Soyut sınıf

`abstract` sınıf ve metot tanımlamak; soyut sınıftan neden nesne oluşturulamadığını ve alt sınıfın neyi uygulamak zorunda olduğunu açıklamak.

Şablon metot

Soyut sınıfın somut bir metodunun, alt sınıfta uygulanan soyut metodu çağırdığını kodu izleyerek göstermek.

Arayüz

`interface` tanımlamak, `implements` ile bir ya da birden çok arayüzü uygulamak; `default`, `static` ve `private` metotları kullanmak.

Lambda

Fonksiyonel arayüzü tanımak; tek metotluk davranışı lambda ifadesiyle yazmak ve lambdanın ne zaman çalıştığını izlemek.

Tasarım desenleri

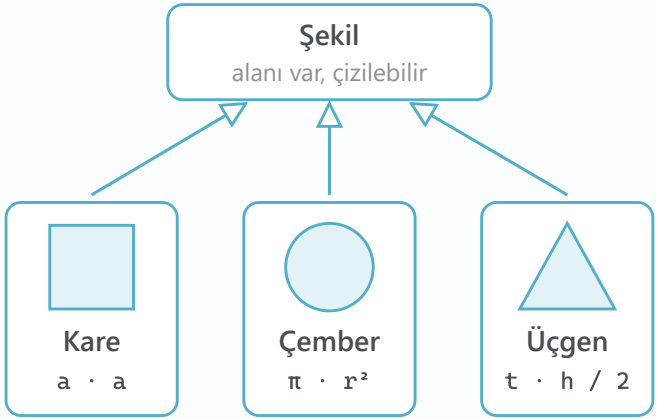
Singleton, Fabrika, Gözlemci, Strateji ve MVC'nin hangi sorunu çözdüğünü ve arayüze nasıl dayandığını açıklamak.

Tasarım kararı

Mühürlü (`sealed`) tipleri tanımak; bir problem için arayüz mü soyut sınıf mı (ya da ikisi birden) gerektiğine karar vermek; jenerik tipleri ve etiketleri (annotation) kullanmak.

Soyutlama (Abstraction)

- **Soyutlama**: karmaşık ayrıntıları gizleyip yalnızca **temel özellikleri** göstermek.
- Araba kullanan kişi direksiyonu ve pedalları bilir; motorun içini bilmek zorunda değildir.
- "Şekil" bir kavramdır: her şeklin alanı vardır, ama **genel bir şeklin** alanı hesaplanamaz. Formülü yalnızca Kare, Çember, Üçgen bilir.
- Java'da soyutlamanın iki aracı vardır: **soyut sınıflar** (`abstract class`) ve **arayüzler** (`interface`).
- İkisi de **ne** yapılacağını söyler; **nasıl** yapılacağını alt sınıflara bırakır.



Ortak kavram yukarıda, ayrıntı aşağıda.

SOYUT SINIFLAR

Soyut Sınıf (Abstract Class)

- `abstract` anahtar sözcüğüyle tanımlanır: `public abstract class Sekil`.
- Hem **soyut** (gövdesiz) hem de **somut** (concrete, gövdeli) metotlar içerebilir.

```
public abstract class Sekil {
    // Soyut olmayan (somut) metot
    void tasi(int x, int y) {
        System.out.println("(" + x + ", " + y + ") noktasına taşındı");
    }
    // Soyut metot (alt sınıflarca uygulanmalı)
    abstract void ciz();
}
```

Somut metot: `tasi`

Gövdesi hazırdır. Alt sınıflar onu olduğu gibi **kalıtımla alır**; isterse geçersiz kılar (override).

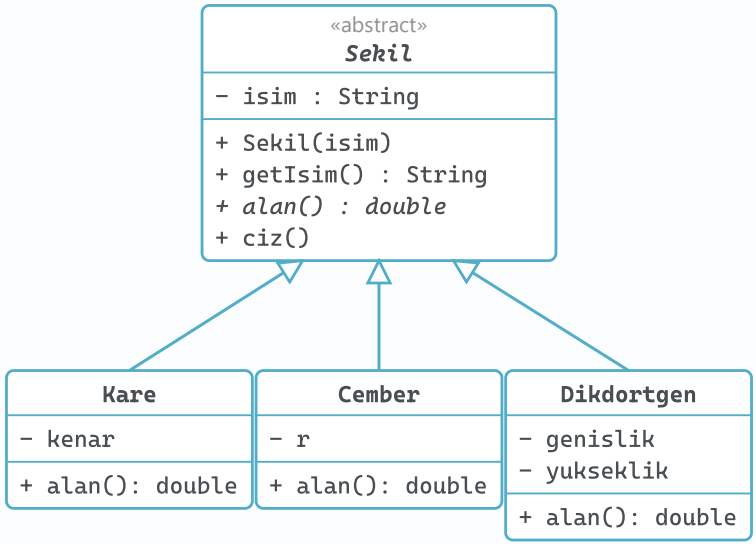
Soyut metot: `ciz`

Yalnızca imzası vardır. Her somut alt sınıf kendi `ciz()` gövdesini **yazmak zorundadır**.

Soyut sınıf yarım kalmış bir plan gibidir: bazı odalar bitmiş, bazılarının yalnızca yeri belli.

Soyut Sınıfın Özellikleri

- Soyut sınıftan **nesne oluşturulamaz** (örneği, instance'ı olamaz).
- Başka sınıflar onu `extends` ile genişletir (kalıtım).
- **Somut** bir alt sınıf, tüm soyut metotları **uygulamak** (implement) zorundadır.
- Ortak durumu (alanlar) ve davranışları tek yerde toplar → kod **yeniden kullanılır**.
- Alt sınıflar soyut metotları kendi ihtiyacına göre doldurur.
- **Yapıcı metotları** (constructor) olabilir; alt sınıf onları `super(...)` ile çağırır.



UML'de soyut sınıf ve soyut metot *eğik* yazılır.

Soyut Sınıftan Nesne Oluşturulamaz

```
Sekil s1 = new Sekil("?");      // HATA
Sekil s2 = new Kare(3);         // Doğru: değişkenin türü soyut olabilir
Sekil[] dizi = new Sekil[3];    // Doğru: dizi yalnızca referans tutar
```

Derleyici mesajı (1. satır)

```
error: Sekil is abstract; cannot be instantiated
```

Neden yasak?

`Sekil`'in `alan()` metodunun gövdesi yok. `new Sekil()` olsaydı `s1.alan()` çağrısında çalıştırılacak kod bulunmazdı.

Referans tipi olarak serbest

Değişken, parametre, dizi öğesi ve dönüş tipi `Sekil` olabilir. `new Sekil[3]` nesne değil, 3 boş (`null`) referans oluşturur.

`abstract final class` yazılamaz: `final` genişletmeyi yasaklar, `abstract` genişletmeyi gerektirir → `illegal combination of modifiers: abstract and final`.

SOYUT SINIFLAR

Soyut Metotlar (Abstract Methods)

- **Gerçekleme (implementation) olmadan** tanımlanır.
- Süslü parantez **içermez**, **noktalı virgülle** biter:

```
public abstract void ciz();
```

- Soyut metodu olan sınıf **soyut olmak zorundadır**.
- Soyut sınıflar genellikle kalıtım hiyerarşisinin **tepesinde** yer alır.

```
public abstract class Sekil {
    // Soyut metot
    public abstract void ciz();
}

// Ayrı bir dosyada:
public class Cember extends Sekil {
    // ciz metodunu uygulamak zorunlu
    public void ciz() {
        System.out.println("Çember çiziliyor");
    }
}
```

`new Cember().ciz();` çıktısı

```
Çember çiziliyor
```

SOYUT SINIFLAR

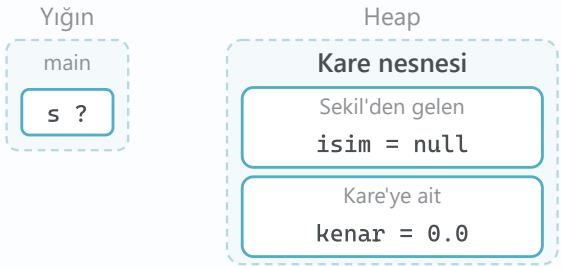
Soyut Metot Kuralları

Yazılan	Sonuç
`abstract void ciz();` (soyut sınıfta)	✓ Doğru
`abstract void ciz() { }`	`abstract methods cannot have a body`
`class Sekil { abstract void ciz(); }`	`Sekil is not abstract and does not override abstract method ciz() in Sekil`
`class Cember extends Sekil { }` (ciz yok)	`Cember is not abstract and does not override abstract method ciz() in Sekil`
`private abstract`, `static abstract`, `final abstract`	`illegal combination of modifiers: abstract and private` (static, final için de aynı)
`abstract class Cember extends Sekil { }`	✓ Doğru: soyut alt sınıf, gerçeklemeyi kendi alt sınıflarına bırakabilir
`abstract class A { void f() { } }`	✓ Doğru: hiç soyut metodu olmayan soyut sınıf da yazılabilir (yine de `new A()` yapılamaz)

Neden `private` / `static` / `final` olmaz? Üçü de metodun alt sınıfta geçersiz kılınmasını engeller; soyut metot ise geçersiz kılınmak için vardır.

`new Kare(3)`: Yapıcı Zinciri

```
1  abstract class Sekil {
2      private String isim;
3      Sekil(String isim) {
4          this.isim = isim;
5          System.out.println(isim + " kuruldu");
6      }
7      abstract double alan();
8  }
9  class Kare extends Sekil {
10     private double kenar;
11     Kare(double kenar) {
12         super("Kare");
13         this.kenar = kenar;
14     }
15     double alan() { return kenar * kenar; }
16 }
17 // main içinde:
▶  Sekil s = new Kare(3);
19 System.out.println(s.alan());
```



`new Kare(3)` : önce heap'te bir **Kare** nesnesi ayrılır. Alanlar varsayılan değer alır: `null` ve `0.0` .

Adım 1 / 10

ÇAĞRI YIĞINI

`main()`

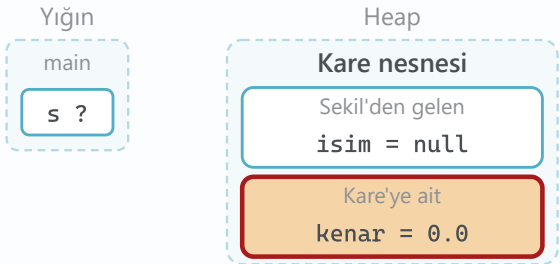
DEĞİŞKENLER

`s = –`

ÇIKTI

`new Kare(3)`: Yapıcı Zinciri

```
1  abstract class Sekil {
2      private String isim;
3      Sekil(String isim) {
4          this.isim = isim;
5          System.out.println(isim + " kuruldu");
6      }
7      abstract double alan();
8  }
9  class Kare extends Sekil {
10     private double kenar;
▶      Kare(double kenar) {
12         super("Kare");
13         this.kenar = kenar;
14     }
15     double alan() { return kenar * kenar; }
16 }
17 // main içinde:
18 Sekil s = new Kare(3);
19 System.out.println(s.alan());
```



Kare'nin yapıcısı çalışır. Parametre `kenar = 3.0` (`int` 3, `double` 'a genişletildi).

Adım 2 / 10

ÇAĞRI YIĞINI

`Kare(3)`

`main()`

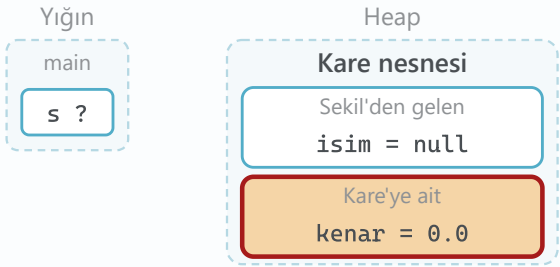
DEĞİŞKENLER

`kenar = 3.0`

ÇIKTI

`new Kare(3)`: Yapıcı Zinciri

```
1  abstract class Sekil {
2      private String isim;
3      Sekil(String isim) {
4          this.isim = isim;
5          System.out.println(isim + " kuruldu");
6      }
7      abstract double alan();
8  }
9  class Kare extends Sekil {
10     private double kenar;
11     Kare(double kenar) {
▶          super("Kare");
13         this.kenar = kenar;
14     }
15     double alan() { return kenar * kenar; }
16 }
17 // main içinde:
18 Sekil s = new Kare(3);
19 System.out.println(s.alan());
```



İlk iş `super("Kare")`: üst sınıfın yapıcısı çağrılır. Soyut sınıfın da yapıcısı olabilir.

Adım 3 / 10

ÇAĞRI YIĞINI

- `Kare(3)`
- `main()`

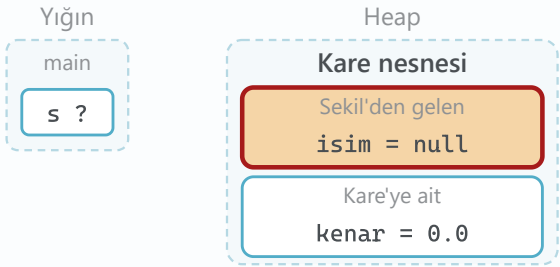
DEĞİŞKENLER

`kenar = 3.0`

ÇIKTI

`new Kare(3)`: Yapıcı Zinciri

```
1  abstract class Sekil {
2      private String isim;
▶      Sekil(String isim) {
4          this.isim = isim;
5          System.out.println(isim + " kuruldu");
6      }
7      abstract double alan();
8  }
9  class Kare extends Sekil {
10     private double kenar;
11     Kare(double kenar) {
12         super("Kare");
13         this.kenar = kenar;
14     }
15     double alan() { return kenar * kenar; }
16 }
17 // main içinde:
18 Sekil s = new Kare(3);
19 System.out.println(s.alan());
```



Sekil yapıcısı çalışıyor. Ayrı bir Sekil nesnesi **oluşmaz**: aynı nesnenin Sekil'den gelen kısmı kuruluyor.

Adım 4 / 10

ÇAĞRI YIĞINI

- `Sekil("Kare")`
- `Kare(3)`
- `main()`

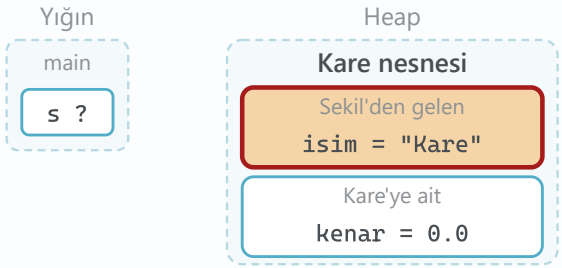
DEĞİŞKENLER

`isim = "Kare"`

ÇIKTI

`new Kare(3)`: Yapıcı Zinciri

```
1  abstract class Sekil {
2      private String isim;
3      Sekil(String isim) {
▶          this.isim = isim;
5          System.out.println(isim + " kuruldu");
6      }
7      abstract double alan();
8  }
9  class Kare extends Sekil {
10     private double kenar;
11     Kare(double kenar) {
12         super("Kare");
13         this.kenar = kenar;
14     }
15     double alan() { return kenar * kenar; }
16 }
17 // main içinde:
18 Sekil s = new Kare(3);
19 System.out.println(s.alan());
```



`this.isim = isim`: nesnenin `isim` alanı "Kare" olur.

Adım 5 / 10

ÇAĞRI YIĞINI

- **`Sekil("Kare")`**
- `Kare(3)`
- `main()`

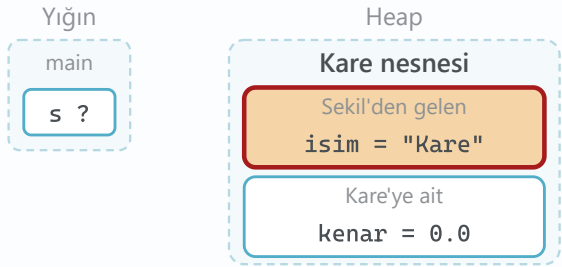
DEĞİŞKENLER

`isim = "Kare"`

ÇIKTI

`new Kare(3)`: Yapıcı Zinciri

```
1  abstract class Sekil {
2      private String isim;
3      Sekil(String isim) {
4          this.isim = isim;
▶          System.out.println(isim + " kuruldu");
6      }
7      abstract double alan();
8  }
9  class Kare extends Sekil {
10     private double kenar;
11     Kare(double kenar) {
12         super("Kare");
13         this.kenar = kenar;
14     }
15     double alan() { return kenar * kenar; }
16 }
17 // main içinde:
18 Sekil s = new Kare(3);
19 System.out.println(s.alan());
```



Mesaj yazdırılır. Sekil yapıcısı biter, çerçevesi yığından çıkar.

Adım 6 / 10

ÇAĞRI YIĞINI

- `Sekil("Kare")`
- `Kare(3)`
- `main()`

DEĞİŞKENLER

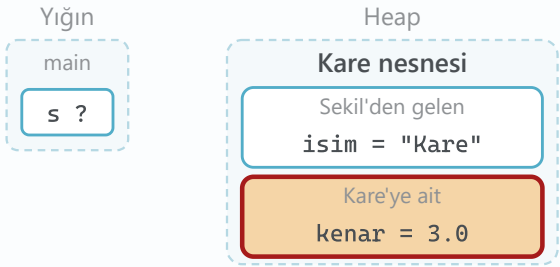
`isim = "Kare"`

ÇIKTI

```
Kare kuruldu
```

`new Kare(3)`: Yapıcı Zinciri

```
1  abstract class Sekil {
2      private String isim;
3      Sekil(String isim) {
4          this.isim = isim;
5          System.out.println(isim + " kuruldu");
6      }
7      abstract double alan();
8  }
9  class Kare extends Sekil {
10     private double kenar;
11     Kare(double kenar) {
12         super("Kare");
▶          this.kenar = kenar;
14     }
15     double alan() { return kenar * kenar; }
16 }
17 // main içinde:
18 Sekil s = new Kare(3);
19 System.out.println(s.alan());
```



Kare yapıcısına dönülür: `this.kenar = 3.0`. Nesne artık tamamen kurulmuştur.

Adım 7 / 10

ÇAĞRI YIĞINI

- `Kare(3)`
- `main()`

DEĞİŞKENLER

`kenar = 3.0`

ÇIKTI

```
Kare kuruldu
```

`new Kare(3)`: Yapıcı Zinciri

```
1  abstract class Sekil {
2      private String isim;
3      Sekil(String isim) {
4          this.isim = isim;
5          System.out.println(isim + " kuruldu");
6      }
7      abstract double alan();
8  }
9  class Kare extends Sekil {
10     private double kenar;
11     Kare(double kenar) {
12         super("Kare");
13         this.kenar = kenar;
14     }
15     double alan() { return kenar * kenar; }
16 }
17 // main içinde:
▶  Sekil s = new Kare(3);
19 System.out.println(s.alan());
```



Nesnenin referansı `s`'ye atanır. Değişkenin tipi **Sekil** (soyut), gösterdiği nesne **Kare**.

Adım 8 / 10

ÇAĞRI YIĞINI

`main()`

DEĞİŞKENLER

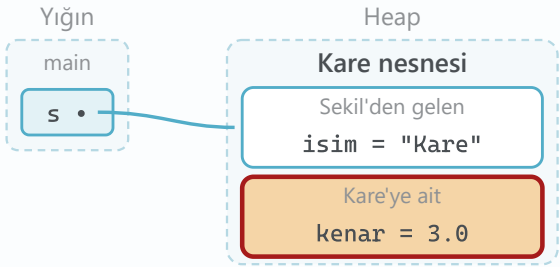
`s = → Kare`

ÇIKTI

```
Kare kuruldu
```

`new Kare(3)`: Yapıcı Zinciri

```
1  abstract class Sekil {
2      private String isim;
3      Sekil(String isim) {
4          this.isim = isim;
5          System.out.println(isim + " kuruldu");
6      }
7      abstract double alan();
8  }
9  class Kare extends Sekil {
10     private double kenar;
11     Kare(double kenar) {
12         super("Kare");
13         this.kenar = kenar;
14     }
▶      double alan() { return kenar * kenar; }
16 }
17 // main içinde:
18 Sekil s = new Kare(3);
19 System.out.println(s.alan());
```



`s.alan()` : Sekil'de `alan()` soyut. Çalışma zamanında nesnenin tipine bakılır → **Kare.alan()**: 3.0 · 3.0.

Adım 9 / 10

ÇAĞRI YIĞINI

- `Kare.alan()`
- `main()`

DEĞİŞKENLER

`kenar = 3.0`

ÇIKTI

```
Kare kuruldu
```

`new Kare(3)`: Yapıcı Zinciri

```
1  abstract class Sekil {
2      private String isim;
3      Sekil(String isim) {
4          this.isim = isim;
5          System.out.println(isim + " kuruldu");
6      }
7      abstract double alan();
8  }
9  class Kare extends Sekil {
10     private double kenar;
11     Kare(double kenar) {
12         super("Kare");
13         this.kenar = kenar;
14     }
15     double alan() { return kenar * kenar; }
16 }
17 // main içinde:
18 Sekil s = new Kare(3);
▶  System.out.println(s.alan());
```



Dönen `9.0` yazdırılır.

Adım 10 / 10

ÇAĞRI YIĞINI

`main()`

DEĞİŞKENLER

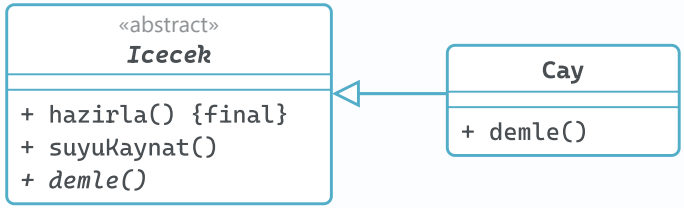
`s = → Kare`

ÇIKTI

```
Kare kuruldu
9.0
```

Somut Metot Soyut Metodu Çağırır

```
1  abstract class Icecek {
2      final void hazirla() {
3          suyuKaynat();
4          demle();
5          System.out.println("Servis edildi");
6      }
7      void suyuKaynat() {
8          System.out.println("Su kaynadı");
9      }
10     abstract void demle();
11 }
12 class Cay extends Icecek {
13     void demle() {
14         System.out.println("Çay demlendi");
15     }
16 }
17 // main içinde:
▶  Icecek i = new Cay();
19 i.hazirla();
```



Değişken tipi `Icecek` (soyut), nesne **Cay**. Cay tek soyut metodu `demle()` 'yi uyguladığı için somuttur.

Adım 1 / 8

ÇAĞRI YIĞINI

`main()`

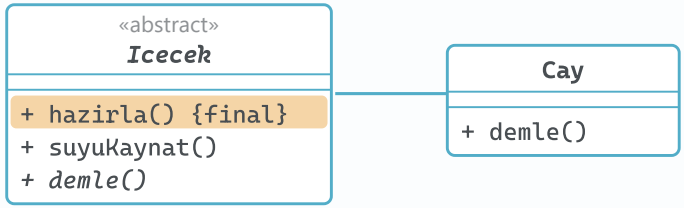
DEĞİŞKENLER

`i = → Cay`

ÇIKTI

Somut Metot Soyut Metodu Çağırır

```
1  abstract class Icecek {
▶       final void hazirla() {
3           suyuKaynat();
4           demle();
5           System.out.println("Servis edildi");
6       }
7       void suyuKaynat() {
8           System.out.println("Su kaynadı");
9       }
10      abstract void demle();
11 }
12 class Cay extends Icecek {
13      void demle() {
14          System.out.println("Çay demlendi");
15      }
16 }
17 // main içinde:
18 Icecek i = new Cay();
19 i.hazirla();
```



Cay'da `hazirla()` yok → Icecek'ten kalıtımla alınan somut metot çalışır. `final` : alt sınıflar bu sırayı değiştiremez.

Adım 2 / 8

ÇAĞRI YIĞINI

- **hazirla()**
- main()

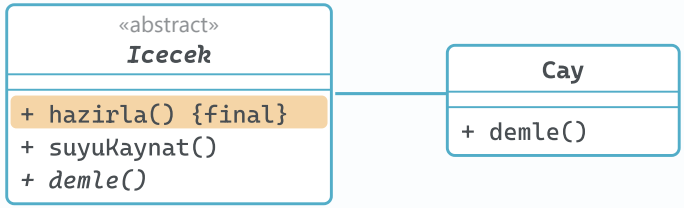
DEĞİŞKENLER

this = → Cay

ÇIKTI

Somut Metot Soyut Metodu Çağırır

```
1  abstract class Icecek {
2      final void hazirla() {
▶          suyuKaynat();
4          demle();
5          System.out.println("Servis edildi");
6      }
7      void suyuKaynat() {
8          System.out.println("Su kaynadı");
9      }
10     abstract void demle();
11 }
12 class Cay extends Icecek {
13     void demle() {
14         System.out.println("Çay demlendi");
15     }
16 }
17 // main içinde:
18 Icecek i = new Cay();
19 i.hazirla();
```



1. adım: `suyuKaynat()`, yani `this.suyuKaynat()` çağrılır.

Adım 3 / 8

ÇAĞRI YIĞINI

- **hazirla()**
- main()

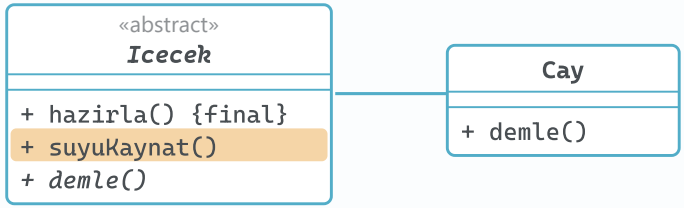
DEĞİŞKENLER

this = → **Cay**

ÇIKTI

Somut Metot Soyut Metodu Çağırır

```
1  abstract class Icecek {
2      final void hazirla() {
3          suyuKaynat();
4          demle();
5          System.out.println("Servis edildi");
6      }
7      void suyuKaynat() {
▶          System.out.println("Su kaynadı");
9      }
10     abstract void demle();
11 }
12 class Cay extends Icecek {
13     void demle() {
14         System.out.println("Çay demlendi");
15     }
16 }
17 // main içinde:
18 Icecek i = new Cay();
19 i.hazirla();
```



Icecek'teki somut `suyuKaynat()` çalışır.

Adım 4 / 8

ÇAĞRI YIĞINI

- **suyuKaynat()**
- hazirla()
- main()

DEĞİŞKENLER

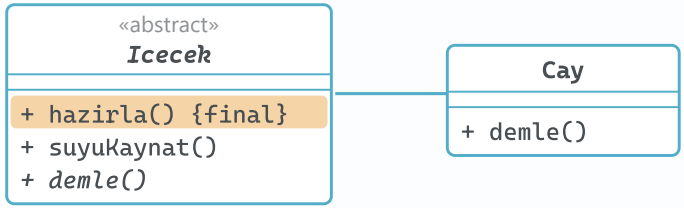
this = → **Cay**

ÇIKTI

```
Su kaynadı
```

Somut Metot Soyut Metodu Çağırır

```
1  abstract class Icecek {
2      final void hazirla() {
3          suyuKaynat();
▶          demle();
5          System.out.println("Servis edildi");
6      }
7      void suyuKaynat() {
8          System.out.println("Su kaynadı");
9      }
10     abstract void demle();
11 }
12 class Cay extends Icecek {
13     void demle() {
14         System.out.println("Çay demlendi");
15     }
16 }
17 // main içinde:
18 Icecek i = new Cay();
19 i.hazirla();
```



2. adım: `demle()`. Icecek'te gövdesi yok, ama `this` bir Cay nesnesi...

Adım 5 / 8

ÇAĞRI YIĞINI

- **hazirla()**
- main()

DEĞİŞKENLER

this = → **Cay**

ÇIKTI

```
Su kaynadı
```

Somut Metot Soyut Metodu Çağırır

```
1  abstract class Icecek {
2      final void hazirla() {
3          suyuKaynat();
4          demle();
5          System.out.println("Servis edildi");
6      }
7      void suyuKaynat() {
8          System.out.println("Su kaynadı");
9      }
10     abstract void demle();
11 }
12 class Cay extends Icecek {
13     void demle() {
▶            System.out.println("Çay demlendi");
15     }
16 }
17 // main içinde:
18 Icecek i = new Cay();
19 i.hazirla();
```



...bu yüzden **Cay.demle()** çalışır. Soyut sınıftaki kod, henüz yazılmamış alt sınıf kodunu çağırabilir.

Adım 6 / 8

ÇAĞRI YIĞINI

- **Cay.demle()**
- hazirla()
- main()

DEĞİŞKENLER

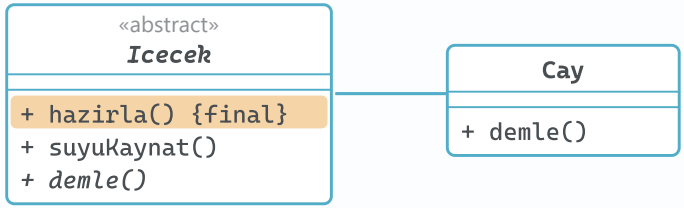
this = → **Cay**

ÇIKTI

```
Su kaynadı
Çay demlendi
```

Somut Metot Soyut Metodu Çağırır

```
1  abstract class Icecek {
2      final void hazirla() {
3          suyuKaynat();
4          demle();
▶          System.out.println("Servis edildi");
6      }
7      void suyuKaynat() {
8          System.out.println("Su kaynadı");
9      }
10     abstract void demle();
11 }
12 class Cay extends Icecek {
13     void demle() {
14         System.out.println("Çay demlendi");
15     }
16 }
17 // main içinde:
18 Icecek i = new Cay();
19 i.hazirla();
```



3. adım: servis mesajı yazdırılır.

Adım 7 / 8

ÇAĞRI YIĞINI

- **hazirla()**
- main()

DEĞİŞKENLER

this = → **Cay**

ÇIKTI

```
Su kaynadı
Çay demlendi
Servis edildi
```

Somut Metot Soyut Metodu Çağırır

```
1  abstract class Icecek {
2      final void hazirla() {
3          suyuKaynat();
4          demle();
5          System.out.println("Servis edildi");
6      }
7      void suyuKaynat() {
8          System.out.println("Su kaynadı");
9      }
10     abstract void demle();
11 }
12 class Cay extends Icecek {
13     void demle() {
14         System.out.println("Çay demlendi");
15     }
16 }
17 // main içinde:
18 Icecek i = new Cay();
▶  i.hazirla();
```



İskelet soyut sınıfta, değişen adım alt sınıfta: **şablon metot** (template method). Bir `Kahve` sınıfı yalnızca `demle()`'yi yazar.

Adım 8 / 8

ÇAĞRI YIĞINI

`main()`

DEĞİŞKENLER

`i = → Cay`

ÇIKTI

```
Su kaynadı
Çay demlendi
Servis edildi
```

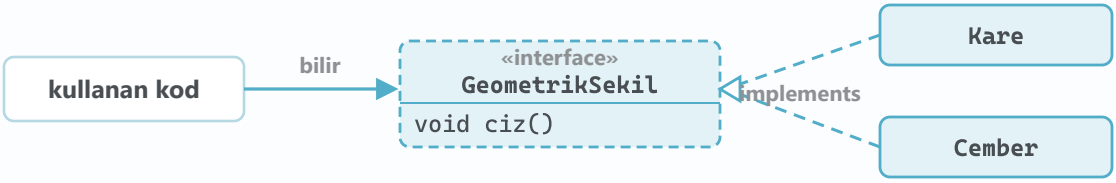
Arayüz (Interface) Nedir?

- Bir sistemin belirli işlevlere sahip olacağını **garanti eden sözleşmedir** (contract).
- İki sistem arasındaki **entegrasyon noktasını** temsil eder: kullanan taraf yalnızca arayüzü bilir.
- `interface` anahtar sözcüğüyle tanımlanır.
- Arayüzdeki metotlar, `public` yazılsın ya da yazılmasın **public**'tir.
- `static`, `default` ya da `private` değilse metot otomatik olarak **abstract**'tır.
- Alan olarak yalnızca `public static final` **sabitler** içerebilir.

Benzetme: priz ve fiş

Priz bir sözleşmedir: hangi cihaz takılırsa takılsın aynı gerilimi verir. Ütü de bilgisayar da prizin standardına uyar; şebeke cihazın içini, cihaz şebekenin içini bilmez.

Soyut sınıf "**bir türüdür**" (is-a) ilişkisi kurar; arayüz "**yapabilir**" (can-do) yeteneği tanımlar.



Arayüz Tanımlama

Yazdığımız

```java
public interface GeometrikSekil {
    // Soyut metot (public abstract)
    void ciz();
    // Sabit (public static final)
    int KENAR_SAYISI = 4;
}
```

Derleyicinin gördüğü

```java
public interface GeometrikSekil {
    public abstract void ciz();
    public static final int KENAR_SAYISI = 4;
}
```

- Sabitlere arayüz adıyla erişilir: `GeometrikSekil.KENAR_SAYISI` → `4`. Adları **BÜYÜK_HARF** ile yazılır.
- Sabite değer vermek zorunludur: `int deger;` → `error: = expected`.
- Sabit değiştirilemez: `GeometrikSekil.KENAR_SAYISI = 5;` → `error: cannot assign a value to static final variable KENAR_SAYISI`.
- Arayüzden de nesne oluşturulamaz: `new GeometrikSekil()` → `error: GeometrikSekil is abstract; cannot be instantiated`.

Arayüzü Uygulamak (implements)

- Bir sınıf, bir arayüzü `implements` anahtar sözcüğüyle **uygular**.
- Arayüzdeki **tüm** soyut metotları uygulamalı ya da kendisi `abstract` olmalıdır.
- Eksik metot: `Kare is not abstract and does not override abstract method ciz() in GeometrikSekil`.
- Uygulayan metot `public` olmalıdır. `void ciz()` yazılırsa: `attempting to assign weaker access privileges; was public`.
- Arayüz tipinde değişken, onu uygulayan her sınıfın nesnesini gösterebilir.

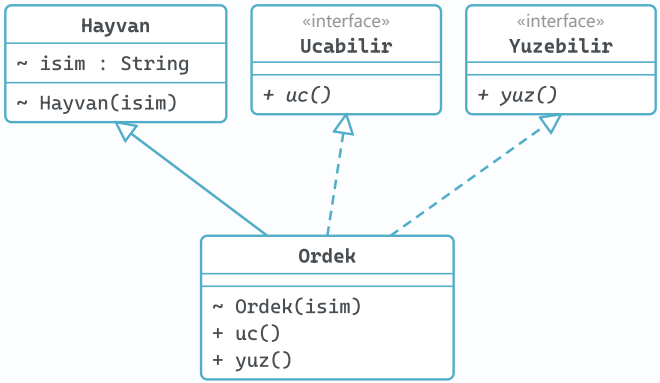
```
public class Kare implements GeometrikSekil {
    // ciz metodunu uygulamak zorunlu
    public void ciz() {
        System.out.println("Kare çiziliyor");
    }
}
// main içinde:
GeometrikSekil g = new Kare();
g.ciz();
System.out.println(GeometrikSekil.KENAR_SAYISI);
```

Çıktı

```
Kare çiziliyor
4
```

Bir Sınıf, Birden Fazla Arayüz

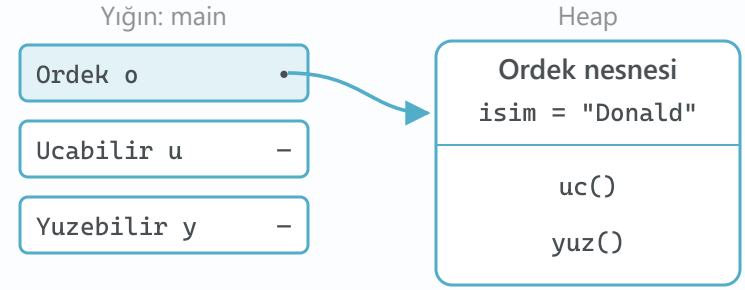
- Bir sınıf **en fazla bir** üst sınıfı genişletebilir (`extends`)...
- ...ama **sıfır ya da daha çok** arayüzü uygulayabilir (`implements A, B, …`).
- Önce `extends` , sonra `implements` yazılır.
- `class C extends A, B` → `error: '{' expected` .




```
class Hayvan {
    String isim;
    Hayvan(String isim) { this.isim = isim; }
}
interface Ucabilir { void uc(); }
interface Yuzebilir { void yuz(); }
class Ordek extends Hayvan implements Ucabilir, Yuzebilir {
    Ordek(String isim) { super(isim); }
    public void uc() { System.out.println(isim + " uçuyor"); }
    public void yuz() { System.out.println(isim + " yüzüyor"); }
}
```

Aynı Nesne, Farklı Arayüz Referansları

```
▶ Ordek o = new Ordek("Donald");
2 Ucabilir u = o;
3 Yuzebilir y = o;
4 u.uc();
5 y.yuz();
6 // u.yuz();   derleme hatası!
7 System.out.println(o.isim);
8 System.out.println(u == o);
```



Heap'te **tek** bir Ordek nesnesi oluşur; `o` onu gösterir.

Adım 1 / 8

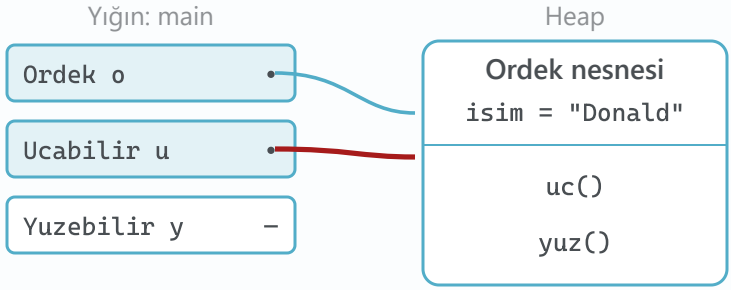
ÇAĞRI YIĞINI

`main()`

ÇIKTI

Aynı Nesne, Farklı Arayüz Referansları

```
1 Ordek o = new Ordek("Donald");
▶ Ucabilir u = o;
3 Yuzebilir y = o;
4 u.uc();
5 y.yuz();
6 // u.yuz();   derleme hatası!
7 System.out.println(o.isim);
8 System.out.println(u == o);
```



Atama nesneyi **kopyalamaz**: `u` aynı nesneyi gösterir. Ordek bir Ucabilir'dir → dönüşüm otomatik.

Adım 2 / 8

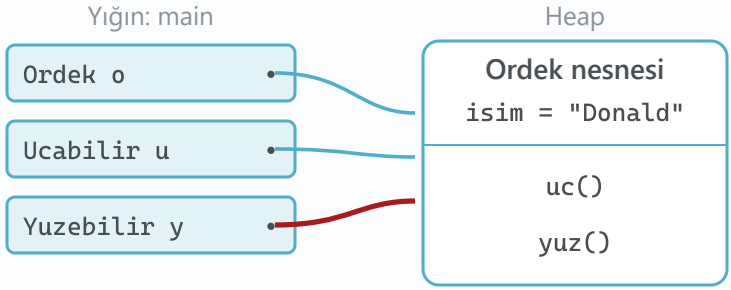
ÇAĞRI YIĞINI

`main()`

ÇIKTI

Aynı Nesne, Farklı Arayüz Referansları

```
1 Ordek o = new Ordek("Donald");
2 Ucabilir u = o;
▶ Yuzebilir y = o;
4 u.uc();
5 y.yuz();
6 // u.yuz();   derleme hatası!
7 System.out.println(o.isim);
8 System.out.println(u == o);
```



`y` da aynı nesneyi gösterir. Bir nesne, üç farklı tipten "bakış".

Adım 3 / 8

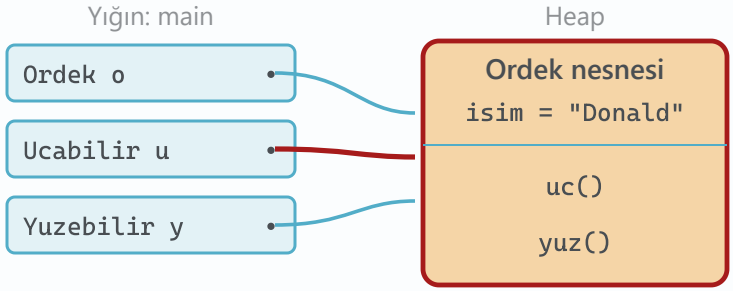
ÇAĞRI YIĞINI

`main()`

ÇIKTI

Aynı Nesne, Farklı Arayüz Referansları

```
1 Ordek o = new Ordek("Donald");
2 Ucabilir u = o;
3 Yuzebilir y = o;
▶ u.uc();
5 y.yuz();
6 // u.yuz();   derleme hatası!
7 System.out.println(o.isim);
8 System.out.println(u == o);
```



`u.uc()` : `u` 'nun tipi Ucabilir; ondan yalnızca `uc()` çağrılabilir. Çalışan, nesnenin kendi `uc()` 'su.

Adım 4 / 8

ÇAĞRI YIĞINI

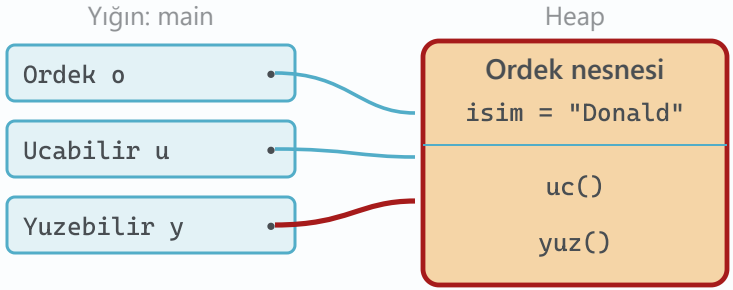
- `Ordek.uc()`
- `main()`

ÇIKTI

```
Donald uçuyor
```

Aynı Nesne, Farklı Arayüz Referansları

```
1 Ordek o = new Ordek("Donald");
2 Ucabilir u = o;
3 Yuzebilir y = o;
4 u.uc();
▶ y.yuz();
6 // u.yuz();   derleme hatası!
7 System.out.println(o.isim);
8 System.out.println(u == o);
```



`y.yuz()` : Yuzebilir tipi `yuz()` 'ü bilir.

Adım 5 / 8

ÇAĞRI YIĞINI

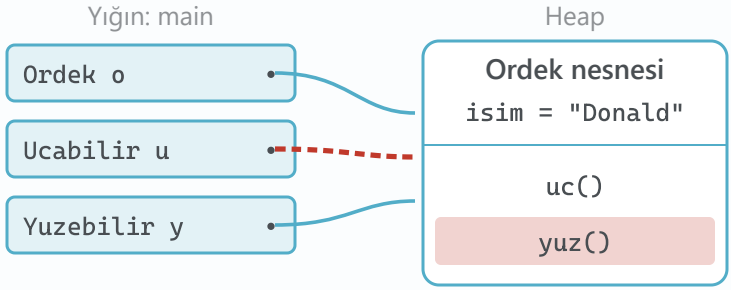
- `Ordek.yuz()`
- `main()`

ÇIKTI

```
Donald uçuyor
Donald yüzüyor
```

Aynı Nesne, Farklı Arayüz Referansları

```
1 Ordek o = new Ordek("Donald");
2 Ucabilir u = o;
3 Yuzebilir y = o;
4 u.uc();
5 y.yuz();
▶ // u.yuz();   derleme hatası!
7 System.out.println(o.isim);
8 System.out.println(u == o);
```



`u.yuz()` yazılsaydı: `error: cannot find symbol`. Nesnede `yuz()` var, ama derleyici **referansın tipine** bakar.

Adım 6 / 8

ÇAĞRI YIĞINI

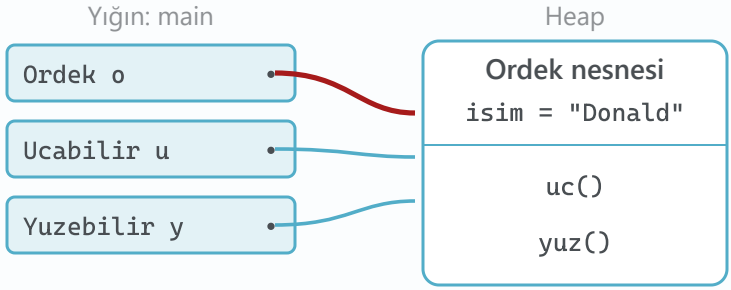
`main()`

ÇIKTI

```
Donald uçuyor
Donald yüzüyor
```

Aynı Nesne, Farklı Arayüz Referansları

```
1 Ordek o = new Ordek("Donald");
2 Ucabilir u = o;
3 Yuzebilir y = o;
4 u.uc();
5 y.yuz();
6 // u.yuz();   derleme hatası!
▶ System.out.println(o.isim);
8 System.out.println(u == o);
```



`o` 'nun tipi Ordek: alan `isim` dahil her şey erişilebilir.

Adım 7 / 8

ÇAĞRI YIĞINI

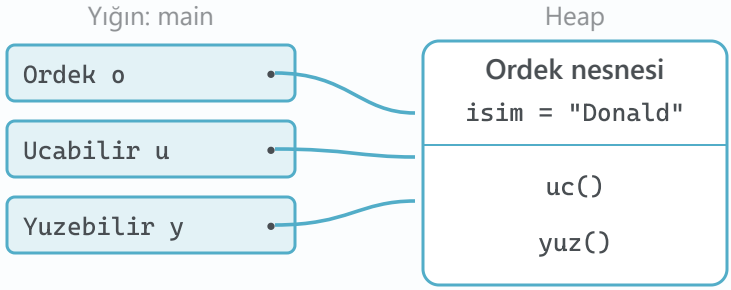
`main()`

ÇIKTI

```
Donald uçuyor
Donald yüzüyor
Donald
```

Aynı Nesne, Farklı Arayüz Referansları

```
1 Ordek o = new Ordek("Donald");
2 Ucabilir u = o;
3 Yuzebilir y = o;
4 u.uc();
5 y.yuz();
6 // u.yuz();   derleme hatası!
7 System.out.println(o.isim);
▶ System.out.println(u == o);
```



`==` referansları karşılaştırır: hepsi aynı nesneyi gösterir → `true`.

Adım 8 / 8

ÇAĞRI YIĞINI

`main()`

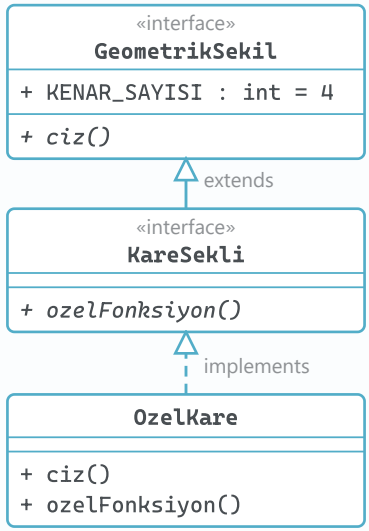
ÇIKTI

```
Donald uçuyor
Donald yüzüyor
Donald
true
```

Arayüzden Kalıtım

Bir arayüz, başka bir arayüzü `extends` ile genişletebilir; virgülle **birden çok** arayüzü de genişletebilir: `interface C extends A, B`.

```
public interface KareSekli extends GeometrikSekil {
    // Ek metotlar veya sabitler eklenebilir
    void ozelFonksiyon();
}
class OzelKare implements KareSekli {
    public void ciz() {
        System.out.println("Özel kare çiziliyor");
    }
    public void ozelFonksiyon() {
        System.out.println("Köşegen: " + Math.sqrt(2));
    }
}
// main içinde:
KareSekli k = new OzelKare();
k.ciz();
k.ozelFonksiyon();
GeometrikSekil g = k;   // üst arayüze atanabilir
```

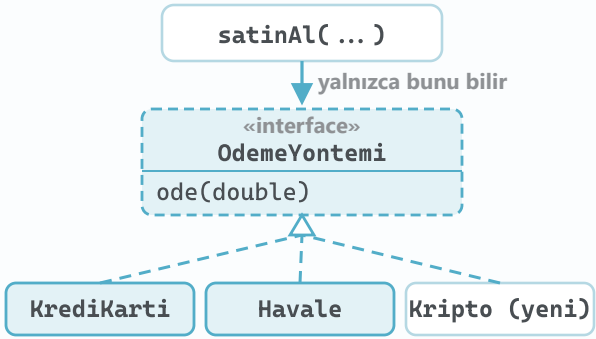


Çıktı

```
Özel kare çiziliyor
Köşegen: 1.4142135623730951
```

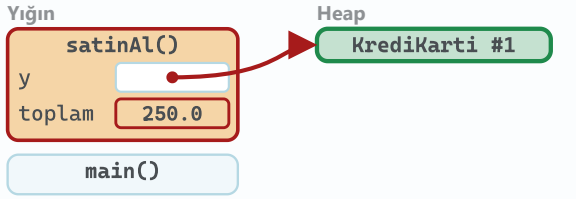
Arayüz Tipiyle Programlama

- Kullanan kod somut sınıfa değil, **arayüze** bağlanır: değişken, parametre ve dönüş tipi arayüz olur.
- `satinAl(OdemeYontemi y, double toplam)` kartla mı havaleyle mi ödendiğini bilmez; `y.ode(toplam)` der, gerisini nesne yapar.
- Yeni bir ödeme türü (`Kripto`) yalnızca arayüzü uygular; `satinAl` hiç değişmez (gevşek bağlılık).
- Test kolaylaşır: gerçek banka yerine arayüzü uygulayan **sahte** bir sınıf verilir.
- Java kütüphanesi de böyledir: `List<String> l = new ArrayList<>();`



Ödeme Arayüzü: Kod İzleme

```
 1 interface OdemeYontemi { void ode(double tutar); }
 2 class KrediKarti implements OdemeYontemi {
 3     public void ode(double tutar) {
 4         System.out.println("Kart: " + tutar + " TL");
 5     }
 6 }
 7 class Havale implements OdemeYontemi {
 8     public void ode(double tutar) {
 9         System.out.println("Havale: " + (tutar + 5) + " TL");
10     }
11 }
▶ static void satinAl(OdemeYontemi y, double toplam) {
13     y.ode(toplam);
14 }
▶ satinAl(new KrediKarti(), 250.0);   // main
16 satinAl(new Havale(), 120.0);
```



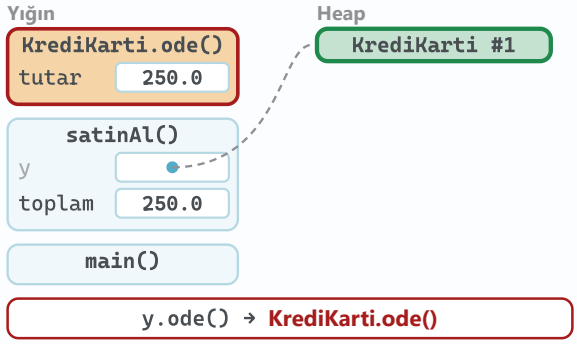
Parametre `y` 'nin tipi arayüzdür; gelen nesne bir `KrediKarti`.

Adım 1 / 6

ÇIKTI

Ödeme Arayüzü: Kod İzleme

```
 1 interface OdemeYontemi { void ode(double tutar); }
 2 class KrediKarti implements OdemeYontemi {
 3     public void ode(double tutar) {
 4         System.out.println("Kart: " + tutar + " TL");
 5     }
 6 }
 7 class Havale implements OdemeYontemi {
 8     public void ode(double tutar) {
 9         System.out.println("Havale: " + (tutar + 5) + " TL");
10     }
11 }
12 static void satinAl(OdemeYontemi y, double toplam) {
13     y.ode(toplam);
14 }
15 satinAl(new KrediKarti(), 250.0);   // main
16 satinAl(new Havale(), 120.0);
```



`y.ode(toplam)` dinamik bağlanır: nesne kart olduğu için `KrediKarti.ode()` çalışır.

Adım 2 / 6

ÇIKTI

```
Kart: 250.0 TL
```

Ödeme Arayüzü: Kod İzleme

```
 1 interface OdemeYontemi { void ode(double tutar); }
 2 class KrediKarti implements OdemeYontemi {
 3     public void ode(double tutar) {
 4         System.out.println("Kart: " + tutar + " TL");
 5     }
 6 }
 7 class Havale implements OdemeYontemi {
 8     public void ode(double tutar) {
 9         System.out.println("Havale: " + (tutar + 5) + " TL");
10     }
11 }
12 static void satinAl(OdemeYontemi y, double toplam) {
13     y.ode(toplam);
14 }
▶ satinAl(new KrediKarti(), 250.0);   // main
16 satinAl(new Havale(), 120.0);
```

Yığın

main()

Heap

KrediKarti #1

Çağrı biter; kart nesnesini artık gösteren referans yoktur, çöp toplayıcı onu silebilir.

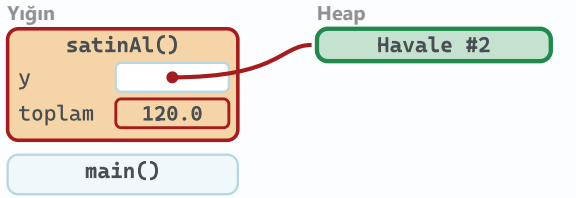
Adım 3 / 6

ÇIKTI

```
Kart: 250.0 TL
```

Ödeme Arayüzü: Kod İzleme

```
 1 interface OdemeYontemi { void ode(double tutar); }
 2 class KrediKarti implements OdemeYontemi {
 3     public void ode(double tutar) {
 4         System.out.println("Kart: " + tutar + " TL");
 5     }
 6 }
 7 class Havale implements OdemeYontemi {
 8     public void ode(double tutar) {
 9         System.out.println("Havale: " + (tutar + 5) + " TL");
10     }
11 }
▶ static void satinAl(OdemeYontemi y, double toplam) {
13     y.ode(toplam);
14 }
15 satinAl(new KrediKarti(), 250.0);   // main
▶ satinAl(new Havale(), 120.0);
```



Aynı `satinAl`, bu kez bir `Havale` nesnesiyle.

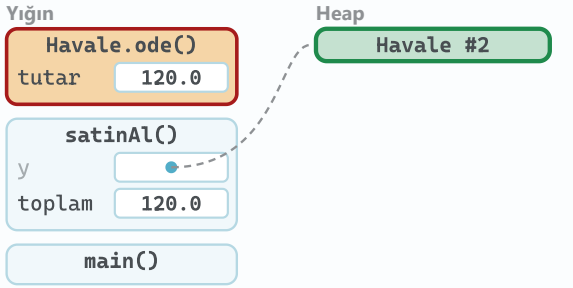
Adım 4 / 6

ÇIKTI

```
Kart: 250.0 TL
```

Ödeme Arayüzü: Kod İzleme

```
 1 interface OdemeYontemi { void ode(double tutar); }
 2 class KrediKarti implements OdemeYontemi {
 3     public void ode(double tutar) {
 4         System.out.println("Kart: " + tutar + " TL");
 5     }
 6 }
 7 class Havale implements OdemeYontemi {
 ▶     public void ode(double tutar) {
 ▶         System.out.println("Havale: " + (tutar + 5) + " TL");
10     }
11 }
12 static void satinAl(OdemeYontemi y, double toplam) {
 ▶     y.ode(toplam);
14 }
15 satinAl(new KrediKarti(), 250.0);   // main
16 satinAl(new Havale(), 120.0);
```



Aynı satır bu kez `Havale.ode()` 'yi çalıştırır; 5 TL masraf ekler: **125.0**.

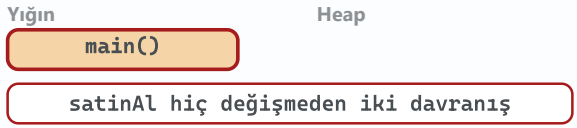
Adım 5 / 6

ÇIKTI

```
Kart: 250.0 TL
Havale: 125.0 TL
```

Ödeme Arayüzü: Kod İzleme

```
1  interface OdemeYontemi { void ode(double tutar); }
2  class KrediKarti implements OdemeYontemi {
3      public void ode(double tutar) {
4          System.out.println("Kart: " + tutar + " TL");
5      }
6  }
7  class Havale implements OdemeYontemi {
8      public void ode(double tutar) {
9          System.out.println("Havale: " + (tutar + 5) + " TL");
10     }
11 }
▶  static void satinAl(OdemeYontemi y, double toplam) {
▶      y.ode(toplam);
14 }
15 satinAl(new KrediKarti(), 250.0);   // main
16 satinAl(new Havale(), 120.0);
```



`satinAl` iki farklı sınıfla çalıştı ama hiçbirinin adını bilmiyor; yeni bir sınıf eklense de değişmez.

Adım 6 / 6

ÇIKTI

```
Kart: 250.0 TL
Havale: 125.0 TL
```

Anonim (Lambda) Fonksiyonlar

- **Anonim fonksiyonlar**, isimsiz fonksiyonlardır: isim yerine doğrudan kod parçasını temsil ederler.
- Java'da **lambda ifadeleri** (lambda expressions) olarak bilinir; **Java 8+**.
- Söz dizimi: `(ParametreListesi) -> İfade`
- Çok satırlı gövde: `(a, b) -> { ...; return ...; }`
- Parametre tipleri çoğu zaman yazılmaz, derleyici çıkarır: `(x, y) -> x + y`. Tek parametrede parantez de gerekmez: `s -> s * 2`.



Toplama lambda ifadesi: `(int x, int y) -> x + y` iki tam sayı alır ve toplar. Aynı işi yapan metot:

```
static int toplama(int x, int y) {
    return x + y;
}
```

Fonksiyonel Arayüz (Functional Interface)

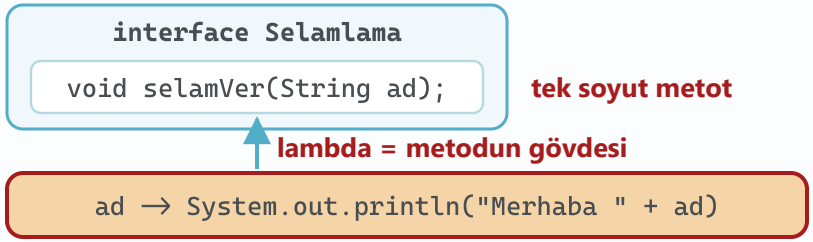
- **Fonksiyonel arayüz**: **tek bir soyut metodu** olan arayüz. Lambda ancak böyle bir tipe atanabilir.
- Lambda, o tek metodun gövdesini yazar; parametre ve dönüş tipleri metottan çıkarılır.
- `@FunctionalInterface` isteğe bağlıdır; yazılırsa derleyici ikinci bir soyut metot eklenmesine izin vermez.
- Lambda tanımlandığında **çalışmaz**; metot çağrıldığında (`s.selamVer(...)`) çalışır.

```
@FunctionalInterface
interface Selamlama {
    void selamVer(String ad);
}

Selamlama s = ad -> System.out.println("Merhaba " + ad);
s.selamVer("Ayşe");
```

ÇIKTI

```
Merhaba Ayşe
```



Hazır Fonksiyonel Arayüzler

Her seferinde arayüz yazmak gerekmez: `java.util.function` paketinde sık kullanılan biçimler hazırdır.

Arayüz	Girdi	Çıktı	Metot	Örnek kullanım
`Consumer<T>`	bir değer	yok	`accept`	yazdırma, kaydetme
`Supplier<T>`	yok	bir değer	`get`	değer üretme
`Function<T, R>`	bir değer	bir değer	`apply`	dönüştürme
`Predicate<T>`	bir değer	`boolean`	`test`	süzme, koşul

```
Consumer<String> yaz = m -> System.out.println(m);
yaz.accept("Consumer çalıştı");
Supplier<Integer> zar = () -> 4;
System.out.println(zar.get());
Function<String, Integer> uzunluk = m -> m.length();
System.out.println(uzunluk.apply("Giresun"));
Predicate<Integer> pozitifMi = x -> x > 0;
System.out.println(pozitifMi.test(-7));
```

ÇIKTI

```
Consumer çalıştı
4
7
false
```

Lambda Ne Zaman Çalışır? Adım Adım

```
▶ Predicate<Integer> ciftMi = x -> x % 2 == 0;
2 List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> sonuc = new ArrayList<>();
4 for (int s : sayilar) {
5     if (ciftMi.test(s)) {
6         sonuc.add(s);
7     }
8 }
9 System.out.println(sonuc);
```

`ciftMi: x → x % 2 == 0`

Lambda bir `Predicate<Integer>` nesnesi olarak saklanır; gövdesi henüz **çalışmadı**. `ciftMi` bu davranışı gösteren bir referanstır.

Adım 1 / 14

ÇIKTI

DEĞİŞKENLER

`sonuc = –`

Lambda Ne Zaman Çalışır? Adım Adım

```
1 Predicate<Integer> ciftMi = x -> x % 2 == 0;
▶ List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> sonuc = new ArrayList<>();
4 for (int s : sayilar) {
5     if (ciftMi.test(s)) {
6         sonuc.add(s);
7     }
8 }
9 System.out.println(sonuc);
```

ciftMi: x → x % 2 == 0

sayilar

3	8	5	12

Liste: 3, 8, 5, 12.

Adım 2 / 14

ÇIKTI

DEĞİŞKENLER

sonuc = –

Lambda Ne Zaman Çalışır? Adım Adım

```
1 Predicate<Integer> ciftMi = x -> x % 2 == 0;
2 List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
▶ List<Integer> sonuc = new ArrayList<>();
4 for (int s : sayilar) {
5     if (ciftMi.test(s)) {
6         sonuc.add(s);
7     }
8 }
9 System.out.println(sonuc);
```

`ciftMi: x → x % 2 == 0`

sayilar: `3` `8` `5` `12`

sonuc: boş

Sonuç listesi boş başlar.

Adım 3 / 14

ÇIKTI

DEĞİŞKENLER

`sonuc = []`

Lambda Ne Zaman Çalışır? Adım Adım

```
1 Predicate<Integer> ciftMi = x -> x % 2 == 0;
2 List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> sonuc = new ArrayList<>();
▶ for (int s : sayilar) {
5     if (ciftMi.test(s)) {
6         sonuc.add(s);
7     }
8 }
9 System.out.println(sonuc);
```

`ciftMi: x → x % 2 == 0`

sayilar: **3** | 8 | 5 | 12

sonuc: boş

Sıradaki öğe: `s = 3`.

Adım 4 / 14

ÇIKTI

DEĞİŞKENLER

`s = 3` `sonuc = []`

Lambda Ne Zaman Çalışır? Adım Adım

```
1 Predicate<Integer> ciftMi = x -> x % 2 == 0;
2 List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> sonuc = new ArrayList<>();
4 for (int s : sayilar) {
▶     if (ciftMi.test(s)) {
6         sonuc.add(s);
7     }
8 }
9 System.out.println(sonuc);
```

`ciftMi: x → x % 2 == 0` **test(3) → false**

sayilar: **3** | 8 | 5 | 12

sonuc: boş

`ciftMi.test(3)` lambdanın gövdesini çalıştırır: `3 % 2 == 0` → **false**.

Adım 5 / 14

ÇIKTI

DEĞİŞKENLER

`s = 3` `test = false` `sonuc = []`

Lambda Ne Zaman Çalışır? Adım Adım

```
1 Predicate<Integer> ciftMi = x -> x % 2 == 0;
2 List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> sonuc = new ArrayList<>();
▶ for (int s : sayilar) {
5     if (ciftMi.test(s)) {
6         sonuc.add(s);
7     }
8 }
9 System.out.println(sonuc);
```

`ciftMi: x → x % 2 == 0`

sayilar: `3` `8` `5` `12`

sonuc: boş

Sıradaki öğe: `s = 8`.

Adım 6 / 14

ÇIKTI

DEĞİŞKENLER

`s = 8` `sonuc = []`

Lambda Ne Zaman Çalışır? Adım Adım

```
1 Predicate<Integer> ciftMi = x -> x % 2 == 0;
2 List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> sonuc = new ArrayList<>();
4 for (int s : sayilar) {
▶     if (ciftMi.test(s)) {
6         sonuc.add(s);
7     }
8 }
9 System.out.println(sonuc);
```

`ciftMi: x → x % 2 == 0` **test(8) → true**

sayilar: `3` `8` `5` `12`

sonuc: boş

`ciftMi.test(8)` lambdanın gövdesini çalıştırır: `8 % 2 == 0` → **true**.

Adım 7 / 14

ÇIKTI

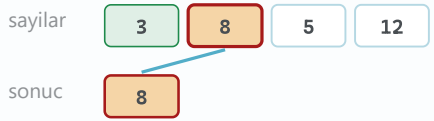
DEĞİŞKENLER

`s = 8` `test = true` `sonuc = []`

Lambda Ne Zaman Çalışır? Adım Adım

```
1 Predicate<Integer> ciftMi = x -> x % 2 == 0;
2 List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> sonuc = new ArrayList<>();
4 for (int s : sayilar) {
5     if (ciftMi.test(s)) {
          sonuc.add(s);
7     }
8 }
9 System.out.println(sonuc);
```

ciftMi: x → x % 2 == 0



Koşul doğru: 8 sonuç listesine eklenir.

Adım 8 / 14

ÇIKTI

DEĞİŞKENLER

`s = 8` `test = true` `sonuc = [8]`

Lambda Ne Zaman Çalışır? Adım Adım

```
1 Predicate<Integer> ciftMi = x -> x % 2 == 0;
2 List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> sonuc = new ArrayList<>();
▶ for (int s : sayilar) {
5     if (ciftMi.test(s)) {
6         sonuc.add(s);
7     }
8 }
9 System.out.println(sonuc);
```

`ciftMi: x → x % 2 == 0`

sayilar: 3 | 8 | 5 | 12

sonuc: 8

Sıradaki öğe: `s = 5`.

Adım 9 / 14

ÇIKTI

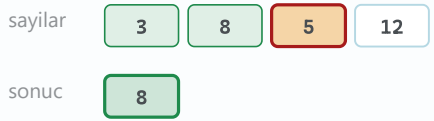
DEĞİŞKENLER

`s = 5` `sonuc = [8]`

Lambda Ne Zaman Çalışır? Adım Adım

```
1 Predicate<Integer> ciftMi = x -> x % 2 == 0;
2 List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> sonuc = new ArrayList<>();
4 for (int s : sayilar) {
▶     if (ciftMi.test(s)) {
6         sonuc.add(s);
7     }
8 }
9 System.out.println(sonuc);
```

ciftMi: x → x % 2 == 0 — test(5) → false



`ciftMi.test(5)` lambdanın gövdesini çalıştırır: `5 % 2 == 0` → **false**.

Adım 10 / 14

ÇIKTI

DEĞİŞKENLER

`s = 5` `test = false` `sonuc = [8]`

Lambda Ne Zaman Çalışır? Adım Adım

```
1 Predicate<Integer> ciftMi = x -> x % 2 == 0;
2 List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> sonuc = new ArrayList<>();
▶ for (int s : sayilar) {
5     if (ciftMi.test(s)) {
6         sonuc.add(s);
7     }
8 }
9 System.out.println(sonuc);
```

ciftMi: x → x % 2 == 0

sayilar: 3 | 8 | 5 | 12

sonuc: 8

Sıradaki öğe: `s = 12`.

Adım 11 / 14

ÇIKTI

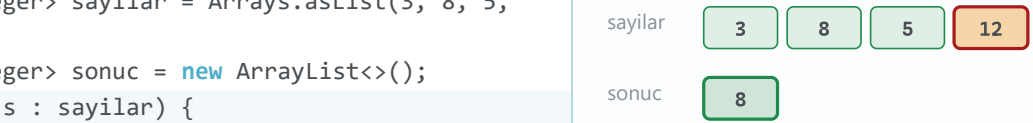
DEĞİŞKENLER

`s = 12` `sonuc = [8]`

Lambda Ne Zaman Çalışır? Adım Adım

```
1 Predicate<Integer> ciftMi = x -> x % 2 == 0;
2 List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> sonuc = new ArrayList<>();
4 for (int s : sayilar) {
▶     if (ciftMi.test(s)) {
6         sonuc.add(s);
7     }
8 }
9 System.out.println(sonuc);
```

ciftMi: x → x % 2 == 0 — test(12) → true



`ciftMi.test(12)` lambdanın gövdesini çalıştırır: `12 % 2 == 0` → **true**.

Adım 12 / 14

ÇIKTI

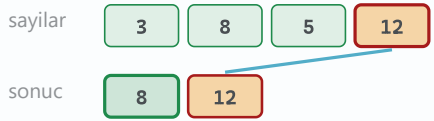
DEĞİŞKENLER

`s = 12` `test = true` `sonuc = [8]`

Lambda Ne Zaman Çalışır? Adım Adım

```
1 Predicate<Integer> ciftMi = x -> x % 2 == 0;
2 List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> sonuc = new ArrayList<>();
4 for (int s : sayilar) {
5     if (ciftMi.test(s)) {
          sonuc.add(s);
7     }
8 }
9 System.out.println(sonuc);
```

ciftMi: x → x % 2 == 0



Koşul doğru: 12 sonuç listesine eklenir.

Adım 13 / 14

ÇIKTI

DEĞİŞKENLER

s = 12

test = true

sonuc = [8, 12]

Lambda Ne Zaman Çalışır? Adım Adım

```
1 Predicate<Integer> ciftMi = x -> x % 2 == 0;
2 List<Integer> sayilar = Arrays.asList(3, 8, 5,
  12);
3 List<Integer> sonuc = new ArrayList<>();
4 for (int s : sayilar) {
5     if (ciftMi.test(s)) {
6         sonuc.add(s);
7     }
8 }
▶ System.out.println(sonuc);
```

`ciftMi: x → x % 2 == 0`



Döngü bitti; `[8, 12]` yazdırılır. Lambda dört kez çalıştı: her `test` çağrısında bir kez.

Adım 14 / 14

ÇIKTI

```
[8, 12]
```

DEĞİŞKENLER

`sonuc = [8, 12]`

Lambda Örneği: Liste Filtreleme

`import java.util.*;` , `java.util.function.IntBinaryOperator` ve `java.util.stream.Collectors` içe aktarılmış `main` içinde:

```
IntBinaryOperator toplama = (int x, int y) -> x + y;
System.out.println(toplama.applyAsInt(5, 3));

List<Integer> sayilar = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
List<Integer> ciftSayilar = sayilar.stream().
                    filter(s -> s % 2 == 0).
                    collect(Collectors.toList());
System.out.println(ciftSayilar);
System.out.println(sayilar.stream().map(s -> s * s).toList());
```

ÇIKTI

```
8
[2, 4, 6]
[1, 4, 9, 16, 25, 36, 49]
```

- Lambda bir **fonksiyonel arayüz** (functional interface) tipine atanır: `IntBinaryOperator` iki `int` alıp `int` döndürür; çağrı `applyAsInt` .
- `filter(s -> s % 2 == 0)` çift sayıları tutar; `map` dönüştürür.
- `.toList()` Java 16+; öncesinde `collect(Collectors.toList())` .

Lambda ve Anonim Sınıf

```
Runnable r1 = new Runnable() {
    @Override
    public void run() {
        System.out.println("Anonim sınıf");
    }
};
Runnable r2 =
        () -> System.out.println("Lambda");
r1.run();
r2.run();
```

ÇIKTI

```
Anonim sınıf
Lambda
```

Lambdadan önce tek metotluk bir davranış **anonim sınıfla** yazılırdı. Lambda aynı işi tek satırda yapar.

Sınırlar

- Lambda yalnızca bir **fonksiyonel arayüz** tipine atanabilir.
- Kullandığı yerel değişkenler `final` ya da fiilen final (effectively final) olmalıdır: sonradan değer atanmamalıdır.
- Lambda içindeki `this`, çevreleyen nesneyi gösterir; anonim sınıfta ise anonim nesnenin kendisini.

```
int kdv = 20;
Function<Integer, Integer> vergili =
        fiyat -> fiyat + fiyat * kdv / 100;
System.out.println(vergili.apply(150)); // 180
kdv = 18;  // eklenirse: derleme hatası
```

```
error: local variables referenced from
  a lambda expression must be final
  or effectively final
```

Kullanım Alanları

- **Koleksiyon işlemleri:** liste filtreleme, sıralama, eşleştirme (`filter` , `sort` , `map` , `forEach`).
- **İş parçacığı yönetimi:** paralel yürütülen işlerin gövdesini kısaca vermek.
- **Daha okunabilir kod:** tek yerde kullanılacak küçük işlemler için ayrı fonksiyon tanımlamak yerine.

```
List<String> isimler = new ArrayList<>(List.of("Zeynep", "Ali", "Mehmet"));
isimler.sort((a, b) -> a.length() - b.length());
System.out.println(isimler);
isimler.forEach(isim -> System.out.println("Merhaba " + isim));
Thread is = new Thread(() -> System.out.println("İş parçacığı çalıştı"));
is.start();
```

ÇIKTI

```
[Ali, Zeynep, Mehmet]
Merhaba Ali
Merhaba Zeynep
Merhaba Mehmet
İş parçacığı çalıştı
```

Sıralama ölçütü bir lambdadır: fark negatifse `a` önce gelir. Eşit uzunluktakiler (Zeynep, Mehmet) ilk sıralarını korur. `() -> ...` parametresiz bir lambdadır.

Varsayılan Metotlar (default, Java 8+)

- Arayüz, **gövdesi olan** varsayılan (`default`) metotlara sahip olabilir.
- Uygulayan sınıf onu **kalıtımla alır**; isterse geçersiz kılar. Amaç: arayüze yeni metot eklerken onu uygulayan eski sınıfları **bozmamak**.

```
interface Collection {
    void add(Object o);
    void remove(Object o);
    int size();
    default boolean isEmpty() {
        return size() == 0;
    }
}
```

```
class Sepet implements Collection {
    private int adet = 0;
    public void add(Object o) { adet++; }
    public void remove(Object o) { adet--; }
    public int size() { return adet; }
    // isEmpty() yazılmadı: default sürüm kullanılır
}
// main içinde:
Sepet sepet = new Sepet();
System.out.println(sepet.isEmpty());
sepet.add("elma");
sepet.add("armut");
System.out.println(sepet.size() + " " + sepet.isEmpty());
```

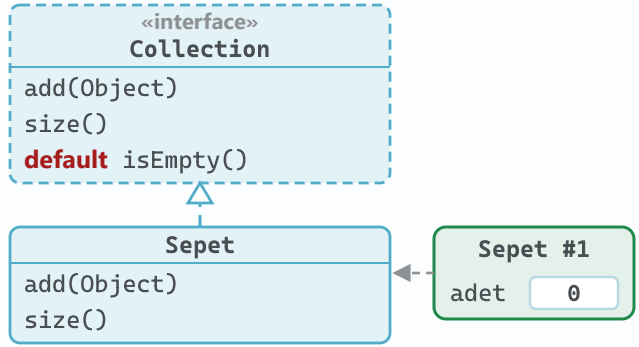
Çıktı

```
true
2 false
```

`isEmpty()` , soyut `size()` 'ı çağırır; hangi `size()` 'ın çalışacağı nesneye göre belirlenir. (Bu `Collection` bizim arayüzümüzdür; `java.util.Collection` ile karıştırılmamalı.)

Varsayılan Metot: Kod İzleme

```
 1 interface Collection {
 2     void add(Object o);
 3     int size();
 4     default boolean isEmpty() {
 5         return size() == 0;
 6     }
 7 }
 8 class Sepet implements Collection {
 9     private int adet = 0;
10     public void add(Object o) { adet++; }
11     public int size() { return adet; }
12 }
▶ Collection c = new Sepet();      // main
14 System.out.println(c.isEmpty());
15 c.add("elma");
16 System.out.println(c.isEmpty());
```



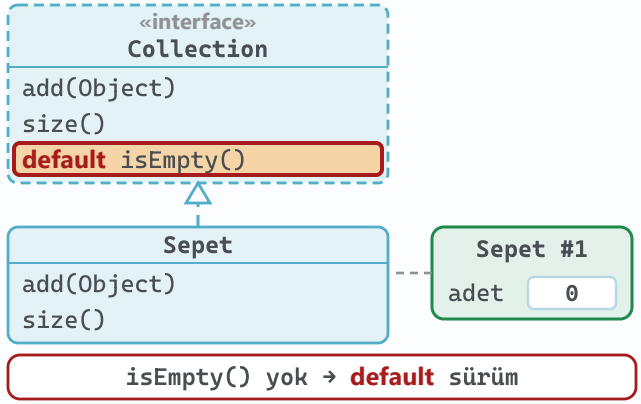
`Sepet` yalnızca iki soyut metodu yazdı; `isEmpty()` 'i yazmadı.

Adım 1 / 6

ÇIKTI

Varsayılan Metot: Kod İzleme

```
 1 interface Collection {
 2     void add(Object o);
 3     int size();
 ▶     default boolean isEmpty() {
 5         return size() == 0;
 6     }
 7 }
 8 class Sepet implements Collection {
 9     private int adet = 0;
10     public void add(Object o) { adet++; }
11     public int size() { return adet; }
12 }
13 Collection c = new Sepet();      // main
 ▶ System.out.println(c.isEmpty());
15 c.add("elma");
16 System.out.println(c.isEmpty());
```



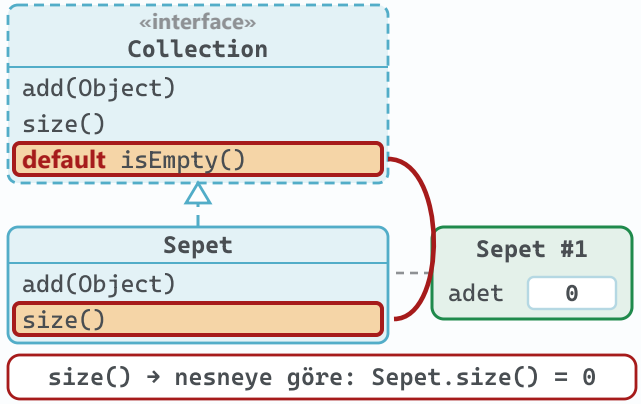
Arama `Sepet`'te `isEmpty()` bulamaz; arayüzdeki **default** gövde çalışır.

Adım 2 / 6

ÇIKTI

Varsayılan Metot: Kod İzleme

```
1  interface Collection {
2      void add(Object o);
3      int size();
4      default boolean isEmpty() {
▶          return size() == 0;
6      }
7  }
8  class Sepet implements Collection {
9      private int adet = 0;
10     public void add(Object o) { adet++; }
▶      public int size() { return adet; }
12 }
13 Collection c = new Sepet();      // main
14 System.out.println(c.isEmpty());
15 c.add("elma");
16 System.out.println(c.isEmpty());
```



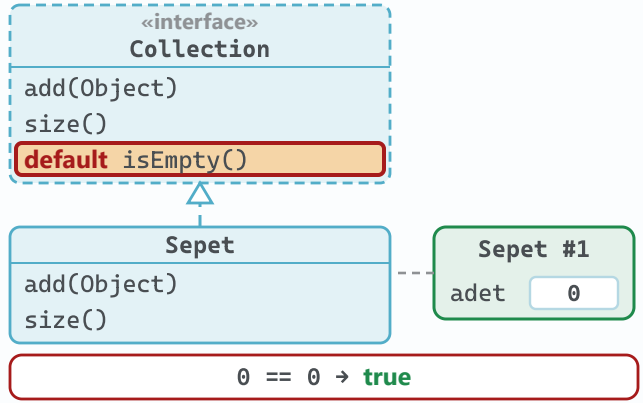
Default metot soyut `size()` 'ı çağırır; hangi gövdenin çalışacağını **nesne** belirler: `Sepet.size()` → 0.

Adım 3 / 6

ÇIKTI

Varsayılan Metot: Kod İzleme

```
 1 interface Collection {
 2     void add(Object o);
 3     int size();
 4     default boolean isEmpty() {
 5         return size() == 0;
 6     }
 7 }
 8 class Sepet implements Collection {
 9     private int adet = 0;
10     public void add(Object o) { adet++; }
11     public int size() { return adet; }
12 }
13 Collection c = new Sepet();      // main
14 System.out.println(c.isEmpty());
15 c.add("elma");
16 System.out.println(c.isEmpty());
```



`0 == 0` : **true**.

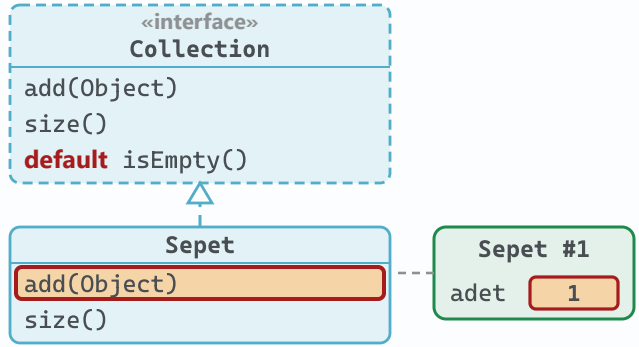
Adım 4 / 6

ÇIKTI

```
true
```

Varsayılan Metot: Kod İzleme

```
 1 interface Collection {
 2     void add(Object o);
 3     int size();
 4     default boolean isEmpty() {
 5         return size() == 0;
 6     }
 7 }
 8 class Sepet implements Collection {
 9     private int adet = 0;
▶      public void add(Object o) { adet++; }
11     public int size() { return adet; }
12 }
13 Collection c = new Sepet();      // main
14 System.out.println(c.isEmpty());
▶  c.add("elma");
16 System.out.println(c.isEmpty());
```



`add` Sepet'in sürümüdür: `adet` 1 olur.

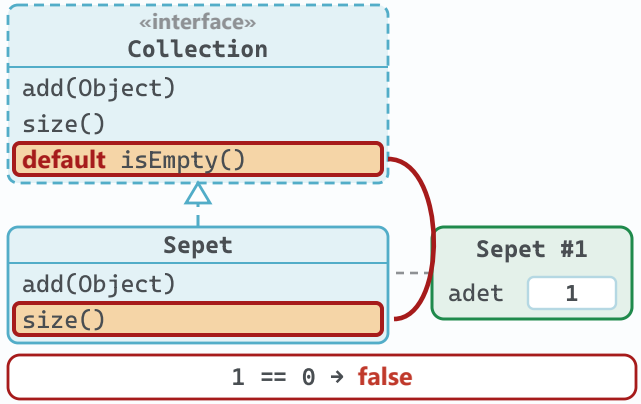
Adım 5 / 6

ÇIKTI

```
true
```

Varsayılan Metot: Kod İzleme

```
 1 interface Collection {
 2     void add(Object o);
 3     int size();
 4     default boolean isEmpty() {
 ▶           return size() == 0;
 6     }
 7 }
 8 class Sepet implements Collection {
 9     private int adet = 0;
10     public void add(Object o) { adet++; }
 ▶     public int size() { return adet; }
12 }
13 Collection c = new Sepet();      // main
14 System.out.println(c.isEmpty());
15 c.add("elma");
 ▶ System.out.println(c.isEmpty());
```



Aynı default gövde bu kez 1 görür: **false**. Arayüz yazıldıktan sonra eklenen bir default metot, eski sınıfları bozmadan onlara da gelir.

Adım 6 / 6

ÇIKTI

```
true
false
```

static Metotlar (Java 8+)

```
interface Geometry {
    static double circleArea(double r) { return Math.PI * r * r; }
    static double squareArea(double s) { return s * s; }
    static double perimeter(double s) { return s * 4.0; }
    static double boxVolume(double w, double h, double d) {
        return w * h * d;
    }
}
// main içinde:
System.out.println(Geometry.circleArea(2));
System.out.println(Geometry.squareArea(3));
System.out.println(Geometry.perimeter(2.5));
System.out.println(Geometry.boxVolume(2, 3, 4));
```

Çıktı

```
12.566370614359172
9.0
10.0
24.0
```

- Nesneye bağlı olmayan **yardımcı** işlevler; yalnızca **arayüz adıyla** çağrılır: `Geometry.circleArea(2)`.
- Uygulayan sınıfa **kalıtımla geçmez**: `Daire implements Geometry` olsa da `Daire.squareArea(2)` → `cannot find symbol`.
- Yalnız `PI` yazılırsa `cannot find symbol`; sabit `Math` sınıfındadır: `Math.PI`.

Default Metot Çakışması

- İki arayüz **aynı imzalı** default metot getirirse sınıf hangisini alacağını bilemez; metodu **kendisi yazmak** (geçersiz kılmak) zorundadır.
- İstenen arayüzün sürümü `ArayuzAdi.super.metot()` ile çağrılabilir.

```
interface Selamlayan {
    default void selam() { System.out.println("Merhaba"); }
}
interface Karsilayan {
    default void selam() { System.out.println("Hoş geldiniz"); }
}
class Gorevli implements Selamlayan, Karsilayan {
    public void selam() {       // geçersiz kılmak zorunlu
        Selamlayan.super.selam();
        Karsilayan.super.selam();
    }
}
```

`new Gorevli().selam();`

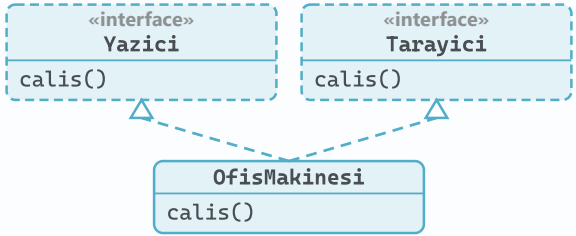
```
Merhaba
Hoş geldiniz
```

`selam()` yazılmasaydı

```
error: types Selamlayan and Karsilayan are incompatible;
  class Gorevli inherits unrelated defaults for selam()
  from types Selamlayan and Karsilayan
```

Default Çakışması: Kod İzleme

```
1  interface Yazici {
2      default void calis() { System.out.println("Yazdırılıyor"); }
3  }
4  interface Tarayici {
5      default void calis() { System.out.println("Taranıyor"); }
6  }
7  class OfisMakinesi implements Yazici, Tarayici {
8      public void calis() {
9          Tarayici.super.calis();
10         Yazici.super.calis();
11         System.out.println("Kopya hazır");
12     }
13 }
▶  Yazici y = new OfisMakinesi();   // main
15 y.calis();
```



çakışma → sınıf **kendisi** yazar

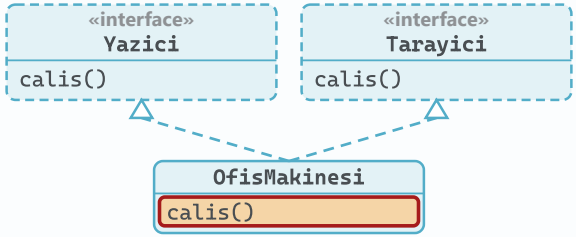
İki arayüz de `calis()` için default gövde getirir. Belirsizliği çözmek için `OfisMakinesi` metodu kendisi yazmıştır.

Adım 1 / 6

ÇIKTI

Default Çakışması: Kod İzleme

```
 1 interface Yazici {
 2     default void calis() { System.out.println("Yazdırılıyor"); }
 3 }
 4 interface Tarayici {
 5     default void calis() { System.out.println("Taranıyor"); }
 6 }
 7 class OfisMakinesi implements Yazici, Tarayici {
 8     public void calis() {
 9         Tarayici.super.calis();
10         Yazici.super.calis();
11         System.out.println("Kopya hazır");
12     }
13 }
14 Yazici y = new OfisMakinesi();   // main
15 y.calis();
```



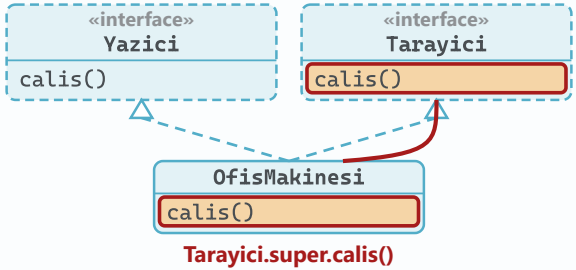
Referans tipi `Yazici` olsa da nesne `OfisMakinesi` : sınıfın kendi `calis()` 'ı çalışır.

Adım 2 / 6

ÇIKTI

Default Çakışması: Kod İzleme

```
 1 interface Yazici {
 2     default void calis() { System.out.println("Yazdırılıyor"); }
 3 }
 4 interface Tarayici {
 ▶     default void calis() { System.out.println("Taranıyor"); }
 6 }
 7 class OfisMakinesi implements Yazici, Tarayici {
 8     public void calis() {
 ▶         Tarayici.super.calis();
10         Yazici.super.calis();
11         System.out.println("Kopya hazır");
12     }
13 }
14 Yazici y = new OfisMakinesi();   // main
15 y.calis();
```



`Tarayici.super.calis()` : adı verilen arayüzün default gövdesi çağrılır.

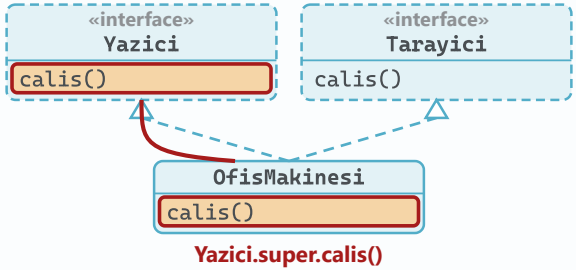
Adım 3 / 6

ÇIKTI

```
Taranıyor
```

Default Çakışması: Kod İzleme

```
1  interface Yazici {
▶      default void calis() { System.out.println("Yazdırılıyor"); }
3  }
4  interface Tarayici {
5      default void calis() { System.out.println("Taranıyor"); }
6  }
7  class OfisMakinesi implements Yazici, Tarayici {
8      public void calis() {
9          Tarayici.super.calis();
▶          Yazici.super.calis();
11         System.out.println("Kopya hazır");
12     }
13 }
14 Yazici y = new OfisMakinesi();   // main
15 y.calis();
```



Sonra `Yazici` 'nınki. Hangi sürümlerin hangi sırayla kullanılacağını sınıf seçer.

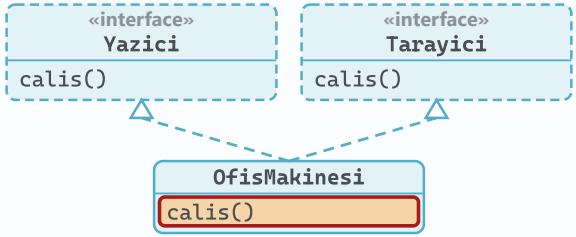
Adım 4 / 6

ÇIKTI

```
Taranıyor
Yazdırılıyor
```

Default Çakışması: Kod İzleme

```
1  interface Yazici {
2      default void calis() { System.out.println("Yazdırılıyor"); }
3  }
4  interface Tarayici {
5      default void calis() { System.out.println("Taranıyor"); }
6  }
7  class OfisMakinesi implements Yazici, Tarayici {
8      public void calis() {
9          Tarayici.super.calis();
10         Yazici.super.calis();
▶          System.out.println("Kopya hazır");
12     }
13 }
14 Yazici y = new OfisMakinesi();   // main
15 y.calis();
```



Son olarak sınıf kendi eklemesini yapar: iki yeteneği birleştiren bir davranış.

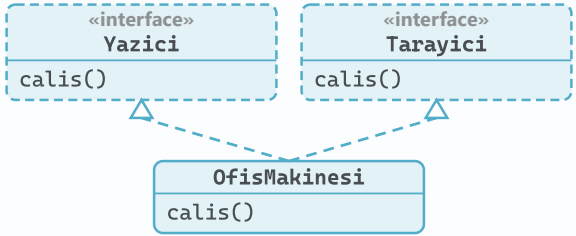
Adım 5 / 6

ÇIKTI

```
Taranıyor
Yazdırılıyor
Kopya hazır
```

Default Çakışması: Kod İzleme

```
1  interface Yazici {
2      default void calis() { System.out.println("Yazdırılıyor"); }
3  }
4  interface Tarayici {
5      default void calis() { System.out.println("Taranıyor"); }
6  }
7  class OfisMakinesi implements Yazici, Tarayici {
8      public void calis() {
9          Tarayici.super.calis();
10         Yazici.super.calis();
11         System.out.println("Kopya hazır");
12     }
13 }
14 Yazici y = new OfisMakinesi();   // main
▶  y.calis();
```



calis() yazılmasaydı: derleme hatası

`OfisMakinesi` `calis()` 'ı yazmasaydı sınıf derlenmezdi: `inherits unrelated defaults`.

Adım 6 / 6

ÇIKTI

```
Taranıyor
Yazdırılıyor
Kopya hazır
```

Özel (private) Arayüz Metotları (Java 9+)

```
interface Yazdirilabilir {
    String icerik();
    default void yazdir() {
        System.out.println(baslik() + ": " + icerik());
    }
    private String baslik() {   // Java 9+
        return "Belge";
    }
}
class Rapor implements Yazdirilabilir {
    public String icerik() { return "Vize sonuçları"; }
}
```

```
new Rapor().yazdir();
```

```
Belge: Vize sonuçları
```

- `private` metotlar, default (ve static) metotların **ortak yardımcı kodunu** dışarı açmadan paylaşır.
- Gövdeleri vardır ve yalnızca arayüzün **içinden** çağrılabilir.
- Uygulayan sınıf onları **göremez**: `Rapor` içinde `baslik()` çağrılamaz.
- Akış: `yazdir()` (default) → `baslik()` (private) + `icerik()` (Rapor'un uyguladığı soyut metot).



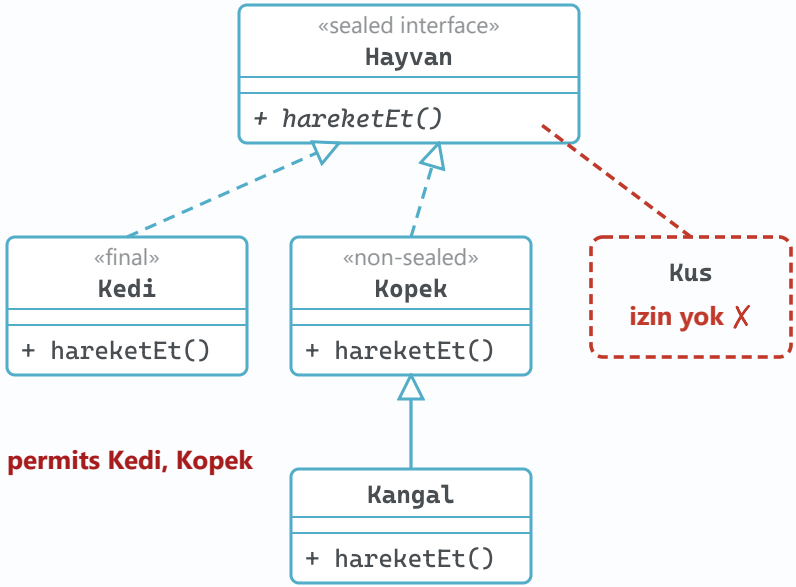
Arayüz Üyeleri: Özet Tablosu

Üye	Örnek	Örtük belirteçler	Gövde	Java
Soyut metot	`void ciz();`	`public abstract`	yok	1.0
Sabit	`int KENAR_SAYISI = 4;`	`public static final`	—	1.0
Varsayılan metot	`default boolean isEmpty() {…}`	`public`	var	8
static metot	`static double squareArea(…) {…}`	`public`	var	8
Özel metot	`private String baslik() {…}`	yalnızca arayüz içinde	var	9
Örnek alanı, yapıcı	—	**İzin verilmez**: arayüzün nesnesi yoktur, durum (state) tutamaz		

Uygulayan sınıf için kural: arayüzdeki tüm **soyut** metotları `public` olarak yazmak. Default metotlar isteğe bağlı, static ve private metotlar zaten kalıtımla geçmez.

Mühürleme: `sealed`, `non-sealed`, `permits`

- Sınıflar ve arayüzler **mühürlenebilir** (Java 17+).
- `sealed` : tipi **yalnızca izin verilen** alt tipler genişletebilir ya da uygulayabilir.
- `permits` : izin verilen alt tiplerin listesi.
- İzin verilen her alt tip birini seçmek **zorundadır**:
  `final` : hiyerarşi burada biter;
  `sealed` : yeniden kısıtlar;
  `non-sealed` : kısıtı kaldırır, herkes genişletebilir.
- Yarar: derleyici tüm alt tipleri **bilir** (kapalı hiyerarşi).



Mühürlü Arayüz Örneği

```java
public sealed interface Hayvan permits Kedi, Kopek {
    void hareketEt();
}
final class Kedi implements Hayvan {
    public void hareketEt() {
        System.out.println("Kedi sessizce yürüyor");
    }
}
non-sealed class Kopek implements Hayvan {
    public void hareketEt() {
        System.out.println("Köpek koşuyor");
    }
}
class Kangal extends Kopek {   // serbest
    public void hareketEt() {
        System.out.println("Kangal sürüyü koruyor");
    }
}
```

```java
// Java 21: switch tüm alt türleri kapsar,
// default dalı gerekmez
static String tur(Hayvan h) {
    return switch (h) {
        case Kedi k -> "kedi";
        case Kopek k -> "köpek";
    };
}
// main içinde:
Hayvan[] hayvanlar =
    { new Kedi(), new Kopek(), new Kangal() };
for (Hayvan h : hayvanlar) {
    h.hareketEt();
    System.out.println(tur(h));
}
```

Çıktı

```
Kedi sessizce yürüyor
kedi
Köpek koşuyor
köpek
Kangal sürüyü koruyor
köpek
```

KARŞILAŞTIRMA

Arayüz ve Soyut Sınıf Karşılaştırması

Özellik	Arayüz	Soyut sınıf
Tanım / kullanım	`interface` · `implements`	`abstract class` · `extends`
Metotlar	Soyut; Java 8+ `default` ve `static`, Java 9+ `private`	Hem soyut hem somut metotlar
Alanlar	Yalnızca `public static final` sabitler; örnek alanı (instance field) **olamaz**	Her türlü alan (örnek alanları dahil)
Yapıcı	Yok	Olabilir; alt sınıf `super(…)` ile çağırır
Erişim	Üyeler örtük `public` (private yardımcılar hariç)	`public`, `protected`, `private`, paket
Kalıtım	Bir sınıf **çok sayıda** arayüz uygular; arayüz birden çok arayüzü genişletir → çoklu kalıtım özelliği	Bir sınıf yalnızca **tek** bir sınıftan türetilir
Amaç	İlgisiz sınıflara ortak yetenek (“yapabilir”)	Ortak alanlara sahip sınıfları gruplamak (“bir türüdür”)
Nesne	İkisinden de `new` ile nesne oluşturulamaz; ikisi de referans tipi olarak kullanılır	

KARŞILAŞTIRMA

Hangisini Seçmeli?

Soyut sınıf uygundur

+ Alt sınıflar **yakın akraba**: Kare, Çember birer Sekil'dir.
+ Ortak **alanlar** ve **yapıcı** gerekiyor (`isim`).
+ Ortak kodun bir kısmı hazır (şablon metot).

Arayüz uygundur

+ **İlgisiz** sınıflara aynı yetenek: Kus, Ucak, Drone hepsi Ucabilir.
+ Bir sınıfa **birden çok** yetenek kazandırmak.
+ Kullanan kodla uygulayan kod arasında **sözleşme** (API).

Çoğu zaman **ikisi birlikte** kullanılır:

```
abstract class Hayvan implements Sesli { … }
class Ordek extends Hayvan implements Ucabilir, Yuzebilir { … }
```

Java, sınıflarda çoklu kalıtıma izin vermez: iki üst sınıf aynı alanı ya da metodu farklı tanımlarsa hangisinin geçerli olduğu belirsizleşir (elmas problemi). Arayüzler durum taşımadığı için bu sorun büyük ölçüde ortadan kalkar; kalan default çakışmasını ise derleyici sınıfa çözdürür.

Neden Jenerikler?

Object ile

```
class KutuObj {
    private Object icerik;
    void koy(Object o) { icerik = o; }
    Object al() { return icerik; }
}
KutuObj k = new KutuObj();
k.koy("Merhaba");
String s = (String) k.al(); // dönüşüm şart
k.koy(42);                  // itiraz yok
String t = (String) k.al(); // çalışırken hata
```

```
Exception in thread "main"
java.lang.ClassCastException
```

Jenerik sınıf ile

```
class Kutu<T> {
    private T icerik;
    void koy(T x) { icerik = x; }
    T al() { return icerik; }
}
Kutu<String> k = new Kutu<>();
k.koy("Merhaba");
String s = k.al();          // dönüşüm yok
k.koy(42);                  // derlenmez
```

```
error: incompatible types:
  int cannot be converted to String
```

- **Tip güvenliği**: yanlış tipteki değer çalışırken değil, **derlerken** yakalanır.
- Okurken açık dönüşüm (cast) gerekmez; tek gerçekleme her tip için çalışır.

Jenerik Sınıf ve Jenerik Metot

```java
class Kutu<T> {                    // T: tip parametresi
    private T icerik;
    void koy(T x) { icerik = x; }
    T al() { return icerik; }
}
static <T> void yazdir(T oge) {    // jenerik metot
    System.out.println("Öğe: " + oge);
}
// main içinde:
Kutu<String> k1 = new Kutu<>();
k1.koy("Merhaba");
Kutu<Integer> k2 = new Kutu<>();
k2.koy(42);
int n = k2.al() + 1;               // kutudan çıkarma
System.out.println(k1.al() + " " + n);
yazdir(3.14);
yazdir('A');
```

Çıktı

```
Merhaba 43
Öğe: 3.14
Öğe: A
```

- `Kutu<String>` : tip argümanı `T` 'nin yerine geçer.
- `new Kutu<>()` : elmas (`<>`), tipi soldan çıkarır.
- Jenerik metot tip parametresini dönüş tipinden önce bildirir: `<T>` ; tipi argümandan çıkarılır.
- Tip argümanı referans tipi olmalı: `Kutu<int>` derlenmez, `Kutu<Integer>` yazılır.
- Yaygın adlar: `T` tip, `E` öğe, `K` anahtar, `V` değer, `R` sonuç.

Sınırlı Tip Parametresi

`<T extends Ust>` : `T` yalnızca `Ust` ya da onun alt tipi olabilir (üst sınır). Karşılığında metot gövdesi `Ust` 'un metotlarını kullanabilir.

```
static <T extends Number> double ikiKati(T x) {
    return x.doubleValue() * 2;   // Number'ın metodu
}
// main içinde:
System.out.println(ikiKati(21));    // T = Integer
System.out.println(ikiKati(1.5));   // T = Double
System.out.println(ikiKati("on")); // derlenmez
```

İlk iki satır

```
42.0
3.0
```

Üçüncü satır

```
error: method ikiKati in class Main cannot be
  applied to given types; ... inference variable T
  has incompatible bounds (upper: Number, lower: String)
```

Arayüz de sınır olabilir (`extends` yazılır). Aşağıdaki metot her **karşılaştırılabilir** tipte (`Integer` , `String` ...) en büyük öğeyi bulur, çünkü `compareTo` çağırabileceği garantidir:

```
static <T extends Comparable<T>> T enBuyuk(T[] dizi)
```

enBuyuk: Kod İzleme

```
1  static <T extends Comparable<T>> T enBuyuk(T[] dizi) {
2      T eb = dizi[0];
3      for (int i = 1; i < dizi.length; i++) {
4          if (dizi[i].compareTo(eb) > 0) {
5              eb = dizi[i];
6          }
7      }
8      return eb;
9  }
10 // main içinde:
▶  Integer[] sayilar = {7, 12, 5};
▶  String[] adlar = {"elma", "kiraz", "armut"};
13 System.out.println(enBuyuk(sayilar));
14 System.out.println(enBuyuk(adlar));
```

iki dizi

7	12	5

eb ?

İki dizi de uygun: `Integer` ve `String`, `Comparable`'ı kendi tipleriyle uygular. Sayılar otomatik kutulanır.

Adım 1 / 9

ÇIKTI

enBuyuk: Kod İzleme

```
▶ static <T extends Comparable<T>> T enBuyuk(T[] dizi) {
▶     T eb = dizi[0];
3     for (int i = 1; i < dizi.length; i++) {
4         if (dizi[i].compareTo(eb) > 0) {
5             eb = dizi[i];
6         }
7     }
8     return eb;
9 }
10 // main içinde:
11 Integer[] sayilar = {7, 12, 5};
12 String[] adlar = {"elma", "kiraz", "armut"};
▶ System.out.println(enBuyuk(sayilar));
14 System.out.println(enBuyuk(adlar));
```

T = Integer



eb



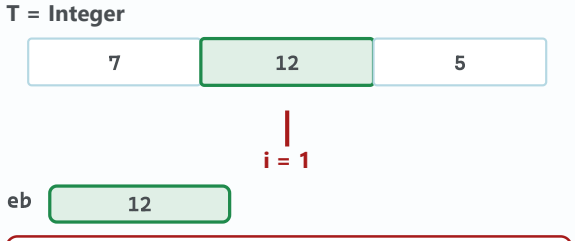
Derleyici `T`'yi `Integer` olarak çıkarır. İlk öğe en büyük aday: `eb = 7`.

Adım 2 / 9

ÇIKTI

enBuyuk: Kod İzleme

```
1  static <T extends Comparable<T>> T enBuyuk(T[] dizi) {
2      T eb = dizi[0];
▶      for (int i = 1; i < dizi.length; i++) {
▶          if (dizi[i].compareTo(eb) > 0) {
▶              eb = dizi[i];
6          }
7      }
8      return eb;
9  }
10 // main içinde:
11 Integer[] sayilar = {7, 12, 5};
12 String[] adlar = {"elma", "kiraz", "armut"};
13 System.out.println(enBuyuk(sayilar));
14 System.out.println(enBuyuk(adlar));
```



`compareTo = 1 > 0 → eb = 12`

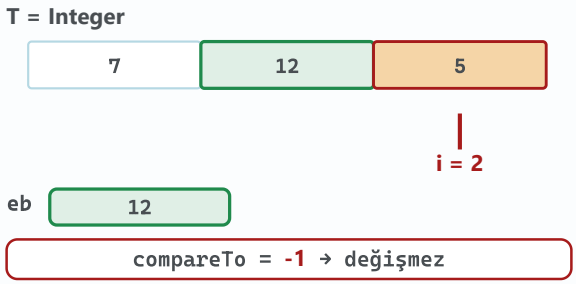
`compareTo` pozitifse öğe daha büyüktür: `eb = 12`.

Adım 3 / 9

ÇIKTI

enBuyuk: Kod İzleme

```
1  static <T extends Comparable<T>> T enBuyuk(T[] dizi) {
2      T eb = dizi[0];
▶      for (int i = 1; i < dizi.length; i++) {
▶          if (dizi[i].compareTo(eb) > 0) {
5              eb = dizi[i];
6          }
7      }
8      return eb;
9  }
10 // main içinde:
11 Integer[] sayilar = {7, 12, 5};
12 String[] adlar = {"elma", "kiraz", "armut"};
13 System.out.println(enBuyuk(sayilar));
14 System.out.println(enBuyuk(adlar));
```



Negatif sonuç: 5 daha küçük, `eb` değişmez.

Adım 4 / 9

ÇIKTI

enBuyuk: Kod İzleme

```
1  static <T extends Comparable<T>> T enBuyuk(T[] dizi) {
2      T eb = dizi[0];
3      for (int i = 1; i < dizi.length; i++) {
4          if (dizi[i].compareTo(eb) > 0) {
5              eb = dizi[i];
6          }
7      }
▶      return eb;
9  }
10 // main içinde:
11 Integer[] sayilar = {7, 12, 5};
12 String[] adlar = {"elma", "kiraz", "armut"};
▶  System.out.println(enBuyuk(sayilar));
14 System.out.println(enBuyuk(adlar));
```

T = Integer

7	12	5

eb 12

Döngü biter, **12** döner. Dönüş tipi `T` = `Integer`: dönüşüm gerekmez.

Adım 5 / 9

ÇIKTI

```
12
```

enBuyuk: Kod İzleme

```
1  static <T extends Comparable<T>> T enBuyuk(T[] dizi) {
▶      T eb = dizi[0];
3      for (int i = 1; i < dizi.length; i++) {
4          if (dizi[i].compareTo(eb) > 0) {
5              eb = dizi[i];
6          }
7      }
8      return eb;
9  }
10 // main içinde:
11 Integer[] sayilar = {7, 12, 5};
12 String[] adlar = {"elma", "kiraz", "armut"};
13 System.out.println(enBuyuk(sayilar));
▶  System.out.println(enBuyuk(adlar));
```

T = String



eb `"elma"`

Aynı metot, bu kez `T = String`. `compareTo` artık sözlük sırasıyla karşılaştırır.

Adım 6 / 9

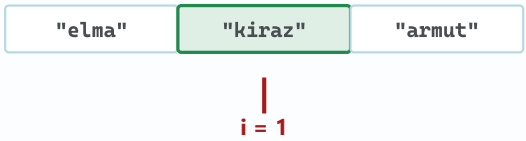
ÇIKTI

```
12
```

enBuyuk: Kod İzleme

```
1  static <T extends Comparable<T>> T enBuyuk(T[] dizi) {
2      T eb = dizi[0];
▶      for (int i = 1; i < dizi.length; i++) {
▶          if (dizi[i].compareTo(eb) > 0) {
▶              eb = dizi[i];
6          }
7      }
8      return eb;
9  }
10 // main içinde:
11 Integer[] sayilar = {7, 12, 5};
12 String[] adlar = {"elma", "kiraz", "armut"};
13 System.out.println(enBuyuk(sayilar));
14 System.out.println(enBuyuk(adlar));
```

T = String



`compareTo = 6 > 0 → eb = "kiraz"`

İlk farklı harfler `'k' - 'e'` = 6: pozitif, `eb = "kiraz"`.

Adım 7 / 9

ÇIKTI

```
12
```

enBuyuk: Kod İzleme

```
1  static <T extends Comparable<T>> T enBuyuk(T[] dizi) {
2      T eb = dizi[0];
▶      for (int i = 1; i < dizi.length; i++) {
▶          if (dizi[i].compareTo(eb) > 0) {
5              eb = dizi[i];
6          }
7      }
8      return eb;
9  }
10 // main içinde:
11 Integer[] sayilar = {7, 12, 5};
12 String[] adlar = {"elma", "kiraz", "armut"};
13 System.out.println(enBuyuk(sayilar));
14 System.out.println(enBuyuk(adlar));
```

T = String



eb `"kiraz"`

`compareTo = -10 → değişmez`

`'a' - 'k'` = −10: negatif, `eb` değişmez.

Adım 8 / 9

ÇIKTI

```
12
```

enBuyuk: Kod İzleme

```
1  static <T extends Comparable<T>> T enBuyuk(T[] dizi) {
2      T eb = dizi[0];
3      for (int i = 1; i < dizi.length; i++) {
4          if (dizi[i].compareTo(eb) > 0) {
5              eb = dizi[i];
6          }
7      }
▶      return eb;
9  }
10 // main içinde:
11 Integer[] sayilar = {7, 12, 5};
12 String[] adlar = {"elma", "kiraz", "armut"};
13 System.out.println(enBuyuk(sayilar));
▶  System.out.println(enBuyuk(adlar));
```

T = String

"elma"	"kiraz"	"armut"

eb "kiraz"

kiraz döner. Tek gerçekleme iki tipte çalıştı; sınır sayesinde `compareTo` çağrısı derleme zamanında güvence altında.

Adım 9 / 9

ÇIKTI

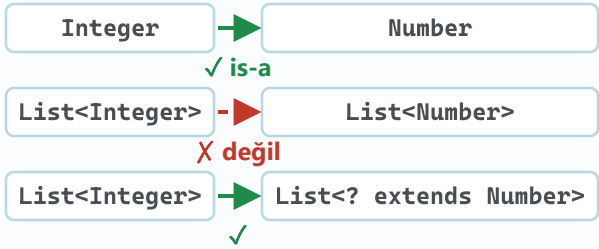
```
12
kiraz
```

Joker Tipler (`?`) ve Değişmezlik

```
static double topla(List<? extends Number> liste) {
    double t = 0;
    for (Number n : liste) t += n.doubleValue();  // okuma
    return t;
}
static void doldur(List<? super Integer> liste) {
    liste.add(1);                                 // yazma
    liste.add(2);
}
// main içinde:
List<Integer> tamlar = Arrays.asList(1, 2, 3);
List<Number> sayilar = new ArrayList<>();
doldur(sayilar);
sayilar.add(2.5);
System.out.println(topla(tamlar) + " " + sayilar);
// List<Number> x = tamlar;   derlenmez
```

Çıktı

```
6.0 [1, 2, 2.5]
```



- **Değişmezlik**: `List<Integer>`, `List<Number>`'un alt tipi değildir; olsaydı listeye `2.5` eklenebilirdi.
- `?` : bilinmeyen herhangi bir tip.
- `? extends T` : liste değer **üretir**, okunur.
- `? super T` : liste değer **tüketir**, yazılır.
- Kural (PECS): *Producer Extends, Consumer Super*.

Tip Silme (Type Erasure)

```
class Kutu<T> {
    private T icerik;
    // T yeni() { return new T(); }   derlenmez
}
// main içinde:
Kutu<String> a = new Kutu<>();
Kutu<Integer> b = new Kutu<>();
System.out.println(a.getClass() == b.getClass());
System.out.println(a.getClass().getName());
```

Çıktı

```
true
Kutu
```



- Jenerik tip bilgisi çoğunlukla **derleyici** içindir; denetimden sonra silinir (*type erasure*). `T` yerine `Object` (ya da sınırı) kalır.
- Çalışırken `Kutu<String>` ile `Kutu<Integer>` **aynı sınıftır**.
- Bu yüzden `new T()` (`unexpected type`) ve `new T[10]` (`generic array creation`) yazılamaz.

Etiketler (Annotations)

Etiket (*annotation*), sınıf, metot, alan ya da parametreye eklenen **üst veridir**. Kendi başına davranış değiştirmez; onu derleyici, araçlar ya da çalışan program (yansıma ile) okur.

Etiket	Kim okur?	Ne yapar?
`@Override`	derleyici	Üst tipte böyle bir metot yoksa hata verir
`@Deprecated`	derleyici	Eskimiş öğe; kullanan kod uyarı alır
`@SuppressWarnings("unchecked")`	derleyici	Belirtilen uyarıyı susturur
`@FunctionalInterface`	derleyici	Arayüzde yalnızca bir soyut metot olduğunu denetler
`@Test` (JUnit)	test aracı	Test metotlarını bulur ve çalıştırır

```java
@FunctionalInterface
interface Islem {
    int uygula(int a, int b);
    int ters(int a);   // 2. soyut metot
}
```

```
error: Unexpected @FunctionalInterface annotation
  Islem is not a functional interface
    multiple non-overriding abstract methods
    found in interface Islem
```

ETİKETLER

Özel Etiket ve Yansıma

```
@Retention(RetentionPolicy.RUNTIME)  // çalışırken okunur
@Target(ElementType.METHOD)          // yalnızca metotlara
@interface Gorev {
    String yazar();
    int oncelik() default 1;
}
class Isler {
    @Gorev(yazar = "Ayşe", oncelik = 3)
    public void yedekle() { System.out.println("Yedek alındı"); }
    public void temizle() { System.out.println("Temizlendi"); }
}
// main içinde (yansıma, java.lang.reflect.Method):
Isler isler = new Isler();
for (Method m : Isler.class.getDeclaredMethods()) {
    Gorev g = m.getAnnotation(Gorev.class);
    if (g != null) {
        System.out.println(m.getName() + ", "
            + g.yazar() + ", " + g.oncelik());
        m.invoke(isler);
    }
}
```

Çıktı

```
yedekle, Ayşe, 3
Yedek alındı
```

`@Retention`	Etiket nerede kalır?
`SOURCE`	Yalnızca kaynakta (`@Override`)
`CLASS`	.class dosyasında; yansıma görmez
`RUNTIME`	Çalışırken yansımayla okunur

- **Yansıma** (*reflection*): program kendi sınıflarını, metotlarını ve etiketlerini çalışırken inceler.
- Derleyici denetimi zayıflar, kod yavaşlar; yalnızca çerçeveler gibi gerçekten gerektiğinde kullanılır.

Tasarım Desenleri (Design Patterns)

- **Tasarım deseni**: sık karşılaşılan bir tasarım sorununa denenmiş, yeniden kullanılabilir bir **çözüm kalıbı**. Kopyalanacak kod değil, sorunu ve ödünleşimleri adlandıran ortak bir **sözlüktür**.
- Sorumlulukları ayırır, sınıflar arasındaki **bağımlılığı azaltır**: bir parça, ötekilere dokunmadan **değiştirilebilir** ve tek başına denenebilir.

Oluşturucu	Yapısal	Davranışsal
nesne nasıl oluşturulur?	nesneler nasıl birleştirilir?	nesneler nasıl iletişim kurar?
`Singleton, Fabrika, Builder`	`Adapter, Decorator, Facade`	`Observer, Strategy, Command`

Desen ancak **gerçek bir sorunu** çözüyorsa yararlıdır. Gereksiz eklenen desen programı karmaşıklaştırır. Desenler, "az bağımlılık", "kalıtım yerine birleşim" gibi daha genel tasarım **ilkelerinin** somut uygulamalarıdır.

Singleton: Tek Nesne

```
public final class Ayarlar {
  private static final Ayarlar TEK = new Ayarlar();
  private Ayarlar() {
    System.out.println("Ayarlar oluşturuldu");
  }
  public static Ayarlar getInstance() { return TEK; }
}
// main:
System.out.println("main başladı");
Ayarlar a = Ayarlar.getInstance();
Ayarlar b = Ayarlar.getInstance();
System.out.println(a == b);
```

Çıktı

```
main başladı
Ayarlar oluşturuldu
true
```

- Sınıfın **yalnızca bir** nesnesi olur ve ona tek bir yerden erişilir.
- `private` yapıcı: dışarıdan `new Ayarlar()` derlenmez.
- Nesne, sınıf ilk kullanıldığında JVM tarafından bir kez oluşturulur: iş parçacığı güvenlidir.
- Gizli bir küresel değişken gibidir: yerine başka bir nesne koymak zorlaşır. Çoğu zaman nesneyi yapıcıya parametre olarak vermek (bağımlılık enjeksiyonu) daha iyidir.

Fabrika (Factory)

```
interface Sekil { double alan(); }
class Kare implements Sekil { … }   // alan: a * a
class Daire implements Sekil { … }  // alan: π r²

class SekilFabrikasi {
  static Sekil olustur(String tur, double olcu) {
    switch (tur.toLowerCase(Locale.ROOT)) {
      case "kare":  return new Kare(olcu);
      case "daire": return new Daire(olcu);
      default:      return null;  // bilinmeyen tür
    }
  }
}
Sekil s = SekilFabrikasi.olustur("Kare", 3);
System.out.println(s.alan());   // 9.0
```

- Nesne oluşturma **tek bir yerde** toplanır; çağıran kod somut sınıfı (`Kare`) değil, **arayüzü** (`Sekil`) bilir.
- Yeni bir şekil eklemek yalnızca fabrikayı değiştirir. Bilinmeyen tür için `null` döner; çağıran kod sonucu denetler.
- `Locale.ROOT` : Türkçe yerel ayarda `"DAIRE".toLowerCase()` → `"daıre"` olur ve eşleşmez.
- Zaten basit oluşturulan nesneler için fabrika yazmak gereksiz karmaşıklıktır.

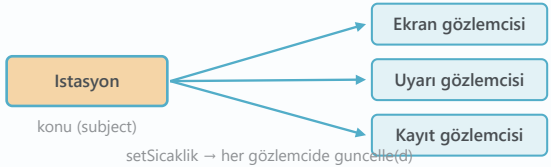
Bilinmeyen tür

```
Sekil u = SekilFabrikasi.olustur("üçgen", 2);
System.out.println(u == null);   // true
```

Gözlemci (Observer)

```
interface Gozlemci { void guncelle(int derece); }

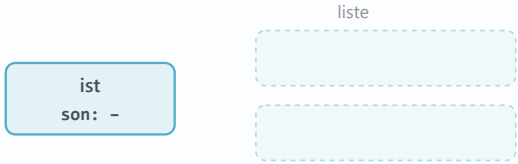
class Istasyon {
  private final List<Gozlemci> liste = new ArrayList<>();
  void ekle(Gozlemci g) { liste.add(g); }
  void setSicaklik(int d) {
    for (Gozlemci g : liste) g.guncelle(d);
  }
}
```



- **Bire çok** ilişki: konu değişince kayıtlı bütün gözlemciler haberdar edilir.
- Konu, gözlemcilerin somut sınıfını bilmez; yalnızca arayüzü çağırır.
- Gözlemciler konsola yazan küçük nesneler olabilir: biri ölçümü gösterir, biri uyarır, biri kaydeder.

Gözlemci Deseni Adım Adım

```
1  interface Gozlemci { void guncelle(int derece); }
2  class Istasyon {
3    private final List<Gozlemci> liste = new ArrayList<>();
4    void ekle(Gozlemci g) { liste.add(g); }
5    void setSicaklik(int d) {
6      for (Gozlemci g : liste) g.guncelle(d);
7    }
8  }
9  // main:
▶ Istasyon ist = new Istasyon();
11 ist.ekle(d -> System.out.println("Ekran: " + d + " °C"));
12 ist.ekle(d -> { if (d > 30) System.out.println("Uyarı:
   sıcak!"); });
13 ist.setSicaklik(25);
14 ist.setSicaklik(34);
```



Konu nesnesi oluşur; gözlemci listesi boş.

Adım 1 / 9

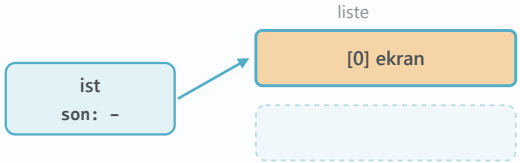
DEĞİŞKENLER

`ist = Istasyon` `liste = []`

ÇIKTI

Gözlemci Deseni Adım Adım

```
1  interface Gozlemci { void guncelle(int derece); }
2  class Istasyon {
3    private final List<Gozlemci> liste = new ArrayList<>();
▶    void ekle(Gozlemci g) { liste.add(g); }
5    void setSicaklik(int d) {
6      for (Gozlemci g : liste) g.guncelle(d);
7    }
8  }
9  // main:
10 Istasyon ist = new Istasyon();
▶ ist.ekle(d -> System.out.println("Ekran: " + d + " °C"));
12 ist.ekle(d -> { if (d > 30) System.out.println("Uyarı:
   sıcak!"); });
13 ist.setSicaklik(25);
14 ist.setSicaklik(34);
```



İlk gözlemci bir lambda: `Gozlemci` arayüzünün tek metodunu uygular. Listeye eklenir, **çalışmaz**.

Adım 2 / 9

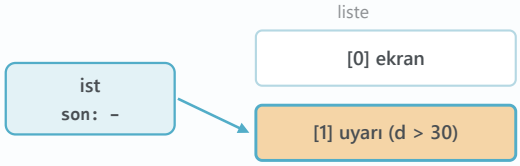
DEĞİŞKENLER

`liste = [ekran]`

ÇIKTI

Gözlemci Deseni Adım Adım

```
1  interface Gozlemci { void guncelle(int derece); }
2  class Istasyon {
3    private final List<Gozlemci> liste = new ArrayList<>();
▶    void ekle(Gozlemci g) { liste.add(g); }
5    void setSicaklik(int d) {
6      for (Gozlemci g : liste) g.guncelle(d);
7    }
8  }
9  // main:
10 Istasyon ist = new Istasyon();
11 ist.ekle(d -> System.out.println("Ekran: " + d + " °C"));
▶  ist.ekle(d -> { if (d > 30) System.out.println("Uyarı:
   sıcak!"); });
13 ist.setSicaklik(25);
14 ist.setSicaklik(34);
```



İkinci gözlemci eklenir: yalnızca 30 derecenin üstünde konuşur.

Adım 3 / 9

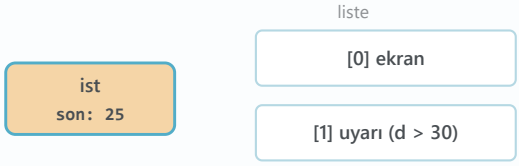
DEĞİŞKENLER

`liste = [ekran, uyarı]`

ÇIKTI

Gözlemci Deseni Adım Adım

```
1  interface Gozlemci { void guncelle(int derece); }
2  class Istasyon {
3    private final List<Gozlemci> liste = new ArrayList<>();
4    void ekle(Gozlemci g) { liste.add(g); }
5    void setSicaklik(int d) {
▶      for (Gozlemci g : liste) g.guncelle(d);
7    }
8  }
9  // main:
10 Istasyon ist = new Istasyon();
11 ist.ekle(d -> System.out.println("Ekran: " + d + " °C"));
12 ist.ekle(d -> { if (d > 30) System.out.println("Uyarı:
   sıcak!"); });
▶  ist.setSicaklik(25);
14 ist.setSicaklik(34);
```



`setSicaklik(25)` : döngü listede baştan gezinir.

Adım 4 / 9

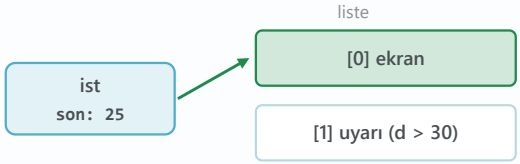
DEĞİŞKENLER

d = 25

ÇIKTI

Gözlemci Deseni Adım Adım

```
1  interface Gozlemci { void guncelle(int derece); }
2  class Istasyon {
3    private final List<Gozlemci> liste = new ArrayList<>();
4    void ekle(Gozlemci g) { liste.add(g); }
5    void setSicaklik(int d) {
▶      for (Gozlemci g : liste) g.guncelle(d);
7    }
8  }
9  // main:
10 Istasyon ist = new Istasyon();
▶ ist.ekle(d -> System.out.println("Ekran: " + d + " °C"));
12 ist.ekle(d -> { if (d > 30) System.out.println("Uyarı:
   sıcak!"); });
13 ist.setSicaklik(25);
14 ist.setSicaklik(34);
```



Ekran gözlemcisinin `guncelle(25)` 'i çağrılır.

Adım 5 / 9

DEĞİŞKENLER

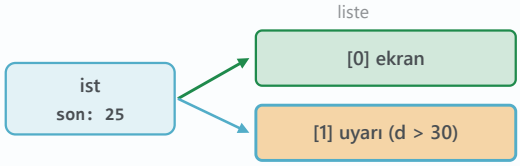
`d = 25` `g = ekran`

ÇIKTI

```
Ekran: 25 °C
```

Gözlemci Deseni Adım Adım

```
1  interface Gozlemci { void guncelle(int derece); }
2  class Istasyon {
3    private final List<Gozlemci> liste = new ArrayList<>();
4    void ekle(Gozlemci g) { liste.add(g); }
5    void setSicaklik(int d) {
▶      for (Gozlemci g : liste) g.guncelle(d);
7    }
8  }
9  // main:
10 Istasyon ist = new Istasyon();
11 ist.ekle(d -> System.out.println("Ekran: " + d + " °C"));
▶  ist.ekle(d -> { if (d > 30) System.out.println("Uyarı:
   sıcak!"); });
13 ist.setSicaklik(25);
14 ist.setSicaklik(34);
```



Uyarı gözlemcisi de haber alır ama 25 > 30 yanlış: bir şey yazmaz.

Adım 6 / 9

DEĞİŞKENLER

`d = 25` `g = uyarı`

ÇIKTI

```
Ekran: 25 °C
```

Gözlemci Deseni Adım Adım

```
1  interface Gozlemci { void guncelle(int derece); }
2  class Istasyon {
3    private final List<Gozlemci> liste = new ArrayList<>();
4    void ekle(Gozlemci g) { liste.add(g); }
5    void setSicaklik(int d) {
▶      for (Gozlemci g : liste) g.guncelle(d);
7    }
8  }
9  // main:
10 Istasyon ist = new Istasyon();
11 ist.ekle(d -> System.out.println("Ekran: " + d + " °C"));
12 ist.ekle(d -> { if (d > 30) System.out.println("Uyarı:
   sıcak!"); });
13 ist.setSicaklik(25);
▶  ist.setSicaklik(34);
```

ist
son: 34

liste

[0] ekran

[1] uyarı (d > 30)

Yeni ölçüm: 34.

Adım 7 / 9

DEĞİŞKENLER

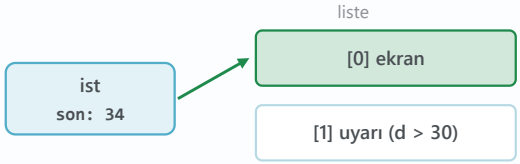
d = 34

ÇIKTI

```
Ekran: 25 °C
```

Gözlemci Deseni Adım Adım

```
1  interface Gozlemci { void guncelle(int derece); }
2  class Istasyon {
3    private final List<Gozlemci> liste = new ArrayList<>();
4    void ekle(Gozlemci g) { liste.add(g); }
5    void setSicaklik(int d) {
▶      for (Gozlemci g : liste) g.guncelle(d);
7    }
8  }
9  // main:
10 Istasyon ist = new Istasyon();
▶ ist.ekle(d -> System.out.println("Ekran: " + d + " °C"));
12 ist.ekle(d -> { if (d > 30) System.out.println("Uyarı:
   sıcak!"); });
13 ist.setSicaklik(25);
14 ist.setSicaklik(34);
```



Ekran yeni değeri yazar.

Adım 8 / 9

DEĞİŞKENLER

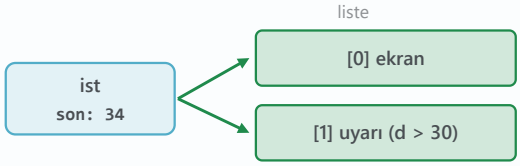
`d = 34` `g = ekran`

ÇIKTI

```
Ekran: 25 °C
Ekran: 34 °C
```

Gözlemci Deseni Adım Adım

```
1  interface Gozlemci { void guncelle(int derece); }
2  class Istasyon {
3    private final List<Gozlemci> liste = new ArrayList<>();
4    void ekle(Gozlemci g) { liste.add(g); }
5    void setSicaklik(int d) {
▶      for (Gozlemci g : liste) g.guncelle(d);
7    }
8  }
9  // main:
10 Istasyon ist = new Istasyon();
11 ist.ekle(d -> System.out.println("Ekran: " + d + " °C"));
▶  ist.ekle(d -> { if (d > 30) System.out.println("Uyarı:
   sıcak!"); });
13 ist.setSicaklik(25);
14 ist.setSicaklik(34);
```



Bu kez koşul doğru: uyarı yazılır. `Istasyon` hiçbir gözlemcinin ne yaptığını bilmedi; yeni gözlemci eklemek onun kodunu değiştirmez.

Adım 9 / 9

DEĞİŞKENLER

`d = 34` `g = uyarı`

ÇIKTI

```
Ekran: 25 °C
Ekran: 34 °C
Uyarı: sıcak!
```

Strateji (Strategy)

```
interface Indirim { double uygula(double tutar); }

class OgrenciIndirimi implements Indirim {
  public double uygula(double tutar) { return tutar * 0.90; }
}
class Kasa {
  private final Indirim strateji;
  Kasa(Indirim strateji) { this.strateji = strateji; }
  double odenecek(double tutar) {
    return strateji.uygula(tutar);
  }
}
Kasa k = new Kasa(new OgrenciIndirimi());
System.out.println(k.odenecek(250));   // 225.0
```

- Birbirinin yerine geçebilen algoritmalar ortak bir **arayüzün** arkasına konur.
- `Kasa`'nın kodu değişmeden başka bir strateji nesnesiyle davranış değişir.
- Arayüzün tek metodu olduğundan strateji bir **lambda** da olabilir: `new Kasa(t -> t * 0.90)`.
- `Comparator` ile sıralama da bir strateji örneğidir.

Strateji Deseni Adım Adım

```
1  interface Indirim { double uygula(double tutar); }
2  class Kasa {
3    private final Indirim strateji;
4    Kasa(Indirim strateji) { this.strateji = strateji; }
5    double odenecek(double tutar) { return
   strateji.uygula(tutar); }
6  }
7  // main:
▶  Kasa ogrenci = new Kasa(t -> t * 0.90);
9  Kasa kampanya = new Kasa(t -> t >= 200 ? t - 50 : t);
10 System.out.println(ogrenci.odenecek(250));
11 System.out.println(kampanya.odenecek(250));
12 System.out.println(kampanya.odenecek(120));
13 // sabit strateji: Kasa tek başına denenir
14 System.out.println(new Kasa(t -> 7).odenecek(100));
```



`Kasa` indirimi kendisi hesaplamaz; yapıcıda verilen **strateji** nesnesine sorar. İlk kasaya %10 indirim takılır.

Adım 1 / 6

DEĞİŞKENLER

`ogrenci = Kasa(%10)`

ÇIKTI

Strateji Deseni Adım Adım

```
1  interface Indirim { double uygula(double tutar); }
2  class Kasa {
3    private final Indirim strateji;
4    Kasa(Indirim strateji) { this.strateji = strateji; }
5    double odenecek(double tutar) { return
   strateji.uygula(tutar); }
6  }
7  // main:
8  Kasa ogrenci = new Kasa(t -> t * 0.90);
▶  Kasa kampanya = new Kasa(t -> t >= 200 ? t - 50 : t);
10 System.out.println(ogrenci.odenecek(250));
11 System.out.println(kampanya.odenecek(250));
12 System.out.println(kampanya.odenecek(120));
13 // sabit strateji: Kasa tek başına denenir
14 System.out.println(new Kasa(t -> 7).odenecek(100));
```



Aynı `Kasa` sınıfı, başka bir stratejiyle: 200 TL ve üstüne 50 TL indirim.

Adım 2 / 6

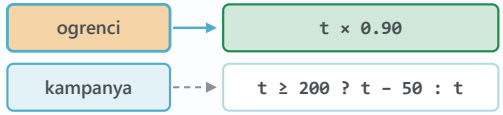
DEĞİŞKENLER

kampanya	Kasa(200 ve üstü −50)

ÇIKTI

Strateji Deseni Adım Adım

```
1  interface Indirim { double uygula(double tutar); }
2  class Kasa {
3    private final Indirim strateji;
4    Kasa(Indirim strateji) { this.strateji = strateji; }
▶    double odenecek(double tutar) { return
   strateji.uygula(tutar); }
6  }
7  // main:
8  Kasa ogrenci = new Kasa(t -> t * 0.90);
9  Kasa kampanya = new Kasa(t -> t >= 200 ? t - 50 : t);
▶  System.out.println(ogrenci.odenecek(250));
11 System.out.println(kampanya.odenecek(250));
12 System.out.println(kampanya.odenecek(120));
13 // sabit strateji: Kasa tek başına denenir
14 System.out.println(new Kasa(t -> 7).odenecek(100));
```



odenecek(250) → 225.0

`odenecek` işi stratejiye devreder: 250 × 0.90 = 225.0.

Adım 3 / 6

DEĞİŞKENLER

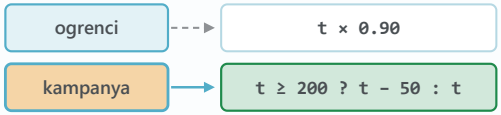
`tutar = 250.0`

ÇIKTI

```
225.0
```

Strateji Deseni Adım Adım

```
1  interface Indirim { double uygula(double tutar); }
2  class Kasa {
3    private final Indirim strateji;
4    Kasa(Indirim strateji) { this.strateji = strateji; }
▶    double odenecek(double tutar) { return
   strateji.uygula(tutar); }
6  }
7  // main:
8  Kasa ogrenci = new Kasa(t -> t * 0.90);
9  Kasa kampanya = new Kasa(t -> t >= 200 ? t - 50 : t);
10 System.out.println(ogrenci.odenecek(250));
▶  System.out.println(kampanya.odenecek(250));
12 System.out.println(kampanya.odenecek(120));
13 // sabit strateji: Kasa tek başına denenir
14 System.out.println(new Kasa(t -> 7).odenecek(100));
```



odenecek(250) → 200.0

Kampanya stratejisi: 250 ≥ 200 → 250 − 50 = 200.0.

Adım 4 / 6

DEĞİŞKENLER

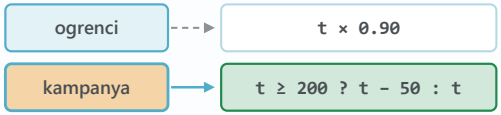
`tutar = 250.0`

ÇIKTI

```
225.0
200.0
```

Strateji Deseni Adım Adım

```
1  interface Indirim { double uygula(double tutar); }
2  class Kasa {
3    private final Indirim strateji;
4    Kasa(Indirim strateji) { this.strateji = strateji; }
▶    double odenecek(double tutar) { return
   strateji.uygula(tutar); }
6  }
7  // main:
8  Kasa ogrenci = new Kasa(t -> t * 0.90);
9  Kasa kampanya = new Kasa(t -> t >= 200 ? t - 50 : t);
10 System.out.println(ogrenci.odenecek(250));
11 System.out.println(kampanya.odenecek(250));
▶  System.out.println(kampanya.odenecek(120));
13 // sabit strateji: Kasa tek başına denenir
14 System.out.println(new Kasa(t -> 7).odenecek(100));
```



odenecek(120) → 120.0

120 < 200: indirim yok. `Kasa`'nın kodu hiç değişmeden davranış değişti.

Adım 5 / 6

DEĞİŞKENLER

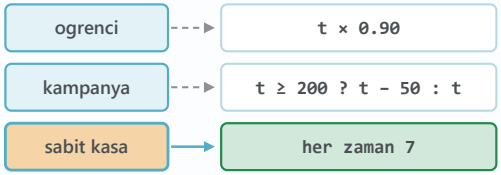
`tutar = 120.0`

ÇIKTI

```
225.0
200.0
120.0
```

Strateji Deseni Adım Adım

```
1  interface Indirim { double uygula(double tutar); }
2  class Kasa {
3    private final Indirim strateji;
4    Kasa(Indirim strateji) { this.strateji = strateji; }
5    double odenecek(double tutar) { return
   strateji.uygula(tutar); }
6  }
7  // main:
8  Kasa ogrenci = new Kasa(t -> t * 0.90);
9  Kasa kampanya = new Kasa(t -> t >= 200 ? t - 50 : t);
10 System.out.println(ogrenci.odenecek(250));
11 System.out.println(kampanya.odenecek(250));
12 System.out.println(kampanya.odenecek(120));
13 // sabit strateji: Kasa tek başına denenir
▶  System.out.println(new Kasa(t -> 7).odenecek(100));
```



odenecek(100) → 7.0

Gerçek indirim kuralı yerine **sabit** bir strateji takılır: tutar ne olursa olsun sonuç 7.0. Böylece yalnızca `Kasa`'nın işi stratejiye doğru devrettiği görülür.

Adım 6 / 6

DEĞİŞKENLER

`tutar = 100.0`

ÇIKTI

```
225.0
200.0
120.0
7.0
```

MVC: Model, Görünüm, Denetleyici



- **Model**: veri ve kurallar (`Ogrenci` , `getAd()`); ekrandan habersizdir, ekran olmadan denenebilir.
- **Görünüm**: yalnızca gösterir (`OgrenciGorunum.goster(ad)` konsola yazar).
- **Denetleyici**: girdiyi alır, modeli ve görünümü yönetir; `guncelle()` modeli okur, sonucu görünüme verir.
- Çerçeveler MVC'yi farklı biçimlerde uygular; tek bir “doğru” sınıf yapısı yoktur.

```
Ogrenci m = new Ogrenci("Ayşe");
OgrenciGorunum g = new OgrenciGorunum();
OgrenciDenetleyici d =
    new OgrenciDenetleyici(m, g);
d.guncelle();  // g.goster(m.getAd())
```

Çıktı

```
Öğrenci: Ayşe
```

Desenler: Ne Zaman Kullanılmaz?

Singleton'ı küresel değişken yapmak

Her yerden erişilen değişebilir durum, programın parçalarını birbirine bağımlı kılar.

Gereksiz fabrika

`new Nokta(3, 4)` kadar basit bir oluşturma için ayrı sınıf yazmak.

Gereksiz gözlemci

Tek bir doğrudan metot çağrısı yeterken olay sistemi kurmak; akışı izlemek zorlaşır.

Deseni ad olsun diye kullanmak

Önce sorun, sonra desen: "Hangi değişikliği kolaylaştırıyor? Hangi bağımlılığı azaltıyor?"

Strateji ve bağımlılığın dışarıdan verilmesi, gerçek parçanın yerine basit bir **sahte nesne** koymayı sağlar (`Kasa` 'ya sabit strateji takmak gibi). Tek başına denemesi zor bir sınıf, çoğu zaman tasarımının gözden geçirilmesi gerektiğini gösterir.

Özet

Kavram	Özü
Soyutlama	Ayrıntıyı gizleyip "ne yapılır"ı tanımlamak; "nasıl"ı alt sınıflara bırakmak
Soyut sınıf	`abstract class`; nesnesi olmaz; alan, yapıcı, soyut ve somut metot içerebilir
Soyut metot	Gövdesiz, `;` ile biter; somut alt sınıf uygulamak zorunda
Arayüz	Sözleşme; metotlar örtük `public abstract`, alanlar `public static final`
implements	Tek `extends`, çok sayıda `implements`; uygulayan metot `public`
Lambda	`(parametreler) -> ifade`; fonksiyonel arayüze atanır (Java 8+)
default / static / private	Gövdeli arayüz metotları (Java 8 / 8 / 9)
sealed	`permits` ile izinli alt tipler; her biri `final`, `sealed` ya da `non-sealed`
Jenerikler	`Kutu<T>`, `<T extends Ust>`, joker `?`; denetim derlemede, tip bilgisi sonra silinir
Etiketler	`@Override` gibi üst veri; `RUNTIME` etiketleri yansımayla okunur
Tasarım desenleri	Singleton, Fabrika, Gözlemci, Strateji, MVC: denenmiş çözüm kalıpları

ÖZET

İlgili Sayfalar

Dinamik Bağlama

Soyut tipten referansla çağrılan metodun hangi sınıftan seçildiğini adım adım gösteren simülatör.

Çok Biçimlilik

Arayüz ve soyut sınıf tipindeki referanslarla çalışan dinamik bağlamanın temelleri.

Soyut metot ya da arayüz metodu uygulanırken yazılan `@Override`, ad ya da parametre yanlış yazıldığında derleme hatasıyla hemen fark edilir.

ÖZET

Sık Yapılan Hatalar (1): Soyut Sınıflar

Soyut sınıftan nesne

`new Sekil("?")` → `Sekil is abstract; cannot be instantiated`. Somut bir alt sınıf kullanılır.

Soyut metodu unutmak

`Cember is not abstract and does not override abstract method ciz() in Sekil`. Metot yazılır ya da sınıf `abstract` yapılır.

Soyut metoda gövde

`abstract void ciz() { }` → `abstract methods cannot have a body`. Ya `abstract` ya da gövde.

Sınıfı abstract yapmamak

`class Sekil { abstract void ciz(); }` → `Sekil is not abstract and does not override abstract method ciz() in Sekil`.

Çelişen belirteçler

`abstract final`, `private abstract`, `static abstract` → `illegal combination of modifiers`.

İmzayı yanlış yazmak

`double Alan()` ya da `double alan(int x)` yeni bir metottur, soyut `alan()` hâlâ eksiktir. `@Override` bunu yakalar.

ÖZET

Sık Yapılan Hatalar (2): Arayüzler

`public`'i unutmak

`void ciz() { … }` → `attempting to assign weaker access privileges; was public`. Arayüz metodu `public` yazılır.

Sabiti değiştirmek

`GeometrikSekil.KENAR_SAYISI = 5;` → `cannot assign a value to static final variable KENAR_SAYISI`.

İki sınıftan türetmek

`class C extends A, B` → `'{' expected`. Tek `extends`, çok sayıda `implements`.

Referans tipinde olmayan metot

`Ucabilir u = new Ordek(); u.yuz();` → `cannot find symbol`. Metot, onu bilen tipte bir referansla çağrılır.

Çakışan default metotlar

`class Gorevli inherits unrelated defaults for selam()`. Metot sınıfta geçersiz kılınır.

Mühürleme hataları

İzinsiz alt tip: `class is not allowed to extend sealed class: Hayvan`. İzinli alt tipte belirteç yok: `sealed, non-sealed or final modifiers expected`.

ÖZET

Örnek Problemler (1): Derlenir mi?

```
// (a)
abstract class A {
    void f() { System.out.println("f"); }
}
// (b)
class B { abstract void g(); }
// (c)
interface I { int X = 1; }
...  I.X++;
// (d)
interface J { void h(); }
class C implements J { void h() { } }
// (e)
abstract class D { abstract void k(); }
abstract class E extends D { }
```

1. Her parça (a)–(e) için: derlenir mi? Derlenmiyorsa hata nedir?
2. Derlenen parçalardan hangisinin nesnesi `new` ile oluşturulabilir?

▶ **Cevapları göster**

ÖZET

Örnek Problemler (2): Çıktı Nedir?

```
interface Sesli {
    default String ses() { return "..."; }
}
abstract class Hayvan implements Sesli {
    Hayvan() { System.out.print("H "); }
    abstract String ad();
}
class Kedi extends Hayvan {
    Kedi() { System.out.print("K "); }
    String ad() { return "kedi"; }
    public String ses() { return "miyav"; }
}
class Balik extends Hayvan {
    String ad() { return "balık"; }
}
// main içinde:
Hayvan[] d = { new Kedi(), new Balik() };
System.out.println();
for (Hayvan h : d)
    System.out.println(h.ad() + ": " + h.ses());
```

1. Program ne yazdırır?
2. `Balik`, `ses()`'i yazmadığı hâlde neden derlenir?
3. `Kedi`'deki `ses()`'ten `public` silinirse ne olur?
4. `Sesli s = new Balik(); s.ad();` derlenir mi?

▶ **Cevapları göster**

ÖZET

Alıştırmalar (1)

1. Soyut `Calisan` sınıfı yazın: `isim` alanı, yapıcı, soyut `double maas()` ve adı ile maaşı yazdıran `bilgi()`. Alt sınıflar: `TamZamanli` (aylık 30000), `SaatUcretli` (120 saat × 200). Bir `Calisan[]` dizisinde `bilgi()` çağırın.
2. `Sekil2B` arayüzü: soyut `double alan()` ve `"Alan: " + alan()` döndüren default `ozet()`. `Ucgen(6, 4)` için `ozet()` ne döndürür?
3. `interface Hesap { static int topla(int a, int b) … }` ve `class Basit implements Hesap {}` : `Hesap.topla(3, 4)` ve `Basit.topla(3, 4)` sonuçları?

▶ **Cevaplar / ipuçları**

ÖZET

Alıştırmalar (2)

4. `sealed abstract class Sekil permits Kare, Cember` yazın; Kare `final`, Cember `non-sealed`. `Elips extends Cember` ve `Ucgen extends Sekil` derlenir mi?
5. Soyut sınıf mı, arayüz mü? (a) plakası ve yapıcısı olan `Tasit`; (b) `Ogrenci` ve `Fatura`'nın ortak `kaydet()` yeteneği; (c) hem Hayvan hem yüzebilen `Balik`.
6. `class Kare implements GeometrikSekil` içinde `public void Ciz() { … }` yazıldı. Ne olur? Metodun üstüne `@Override` eklenirse?

▶ **Cevaplar / ipuçları**

ÖZET

Kaynaklar

- Oracle, *The Java Tutorials*: Abstract Methods and Classes; Interfaces; Default Methods; Lambda Expressions.
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21 Edition (8.1.1.1 abstract Classes, 8.1.1.2 sealed Classes, Bölüm 9: Interfaces, 15.27 Lambda Expressions).
- Oracle, *JEP 409: Sealed Classes* (Java 17) ve *JEP 441: Pattern Matching for switch* (Java 21).
- P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (Bölüm 10: Object-Oriented Programming: Polymorphism and Interfaces).
- K. Sierra ve B. Bates, *Head First Java*, 3. baskı, O'Reilly (Bölüm 8: Interfaces and Abstract Classes).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson (Bölüm 5–6).
- E. Gamma, R. Helm, R. Johnson ve J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

Son

Bölüm 4.5: Soyut Sınıflar ve Arayüzler

Sonraki bölüm: Bölüm 5: Koleksiyonlar · Terim sözlüğü