

Bölüm 4: Soyut Sınıflar ve Arayüzler

Java ile Nesne Yönelimli Programlama

Soyutlama

Soyut sınıflar

Soyut metotlar

Arayüzler

implements ve çoklu arayüz

default, static, private metotlar

sealed, non-sealed, permits

Arayüz mü, soyut sınıf mı?

Öğrenme Çıktıları

Soyut sınıf

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

Dinamik bağlama

Soyut tipten bir referansla çağrılan metodun, çalışma zamanında nesnenin gerçek sınıfından seçildiğini kodu izleyerek göstermek.

Arayüz

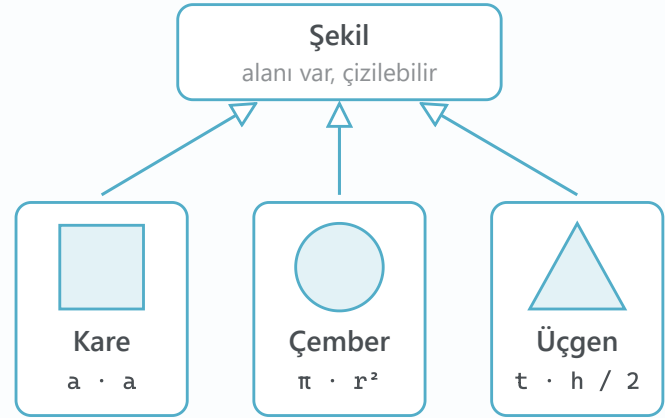
`interface` tanımlamak, `implements` ile bir ya da birden çok arayüzü gerçekleştirmek; `default`, `static` ve `private` metodları kullanmak.

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 kelimesiyle 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 gerçekleştirilmeli)  
    abstract void ciz();  
}
```

Somut metot: `tasi`

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

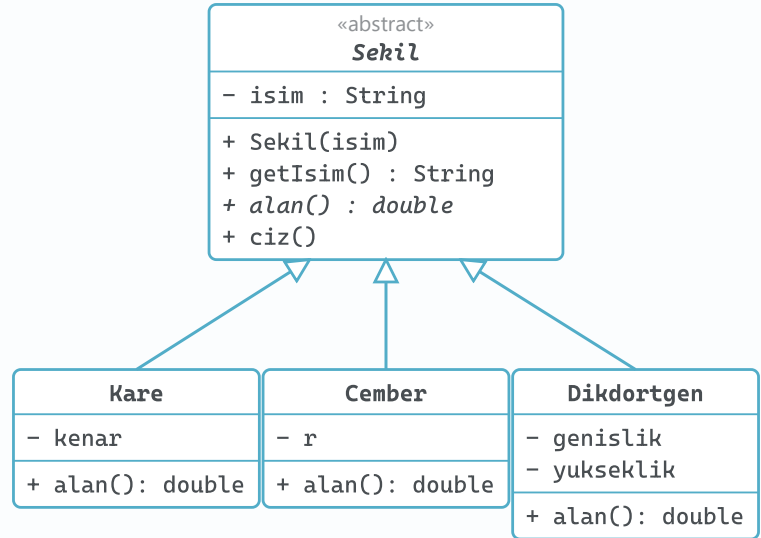
Soyut metot: `ciz`

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 **gerçekleştirmek** (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 SINIFLAR

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()` çağırısında çalıştırılacak kod bulunmazdı.

Referans tipi olarak serbest

Değişken, parametre, dizi ögesi 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 Metotlar (Abstract Methods)

- Gerçekleştirme (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 gerçeklemek zorunlu  
    public void ciz() {  
        System.out.println("Çember çiziliyor");  
    }  
}
```

```
new Cember().ciz();
```

 çıktısı

```
Çember çiziliyor
```

SOYUT SINIFLAR

Soyut Metot Kuralları

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

Yığın

main

s ?

Heap

Kare nesnesi

Sekil'den gelen
isim = nullKare'ye ait
kenar = 0.0

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;  
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());
```

Yığın

main

s ?

Heap

Kare nesnesi

Sekil'den gelen
isim = nullKare'ye ait
kenar = 0.0

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) {  
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());
```

Yığın

main

s ?

Heap

Kare nesnesi

Sekil'den gelen
isim = nullKare'ye ait
kenar = 0.0

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;  
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);  
19 System.out.println(s.alan());
```

Yığın

main

s ?

Heap

Kare nesnesi

Sekil'den gelen
isim = nullKare'ye ait
kenar = 0.0

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) {  
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());
```

Yığın

main

s ?

Heap

Kare nesnesi

Sekil'den gelen

isim = "Kare"

Kare'ye ait

kenar = 0.0

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

Adım 5 / 10

ÇAĞRI YIĞINI

Sekil("Kare")

Kare(3)

main()

DEĞİŞKENLER

isim = "Kare"

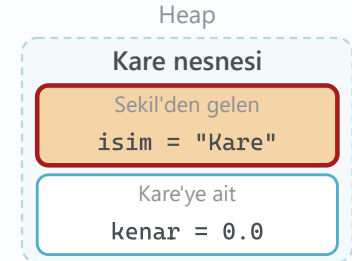
ÇIKTI

new Kare(3): Yapıcı Zinciri

```

1 abstract class Sekil {
2     private String isim;
3     Sekil(String isim) {
4         this.isim = isim;
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());

```



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");  
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());
```

Yığın

main

s ?

Heap

Kare nesnesi

Sekil'den gelen
isim = "Kare"Kare'ye ait
kenar = 3.0

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

Adım 7 / 10

ÇAĞRI YIĞINI

Kare(3)

main()

DEĞİŞKENLER

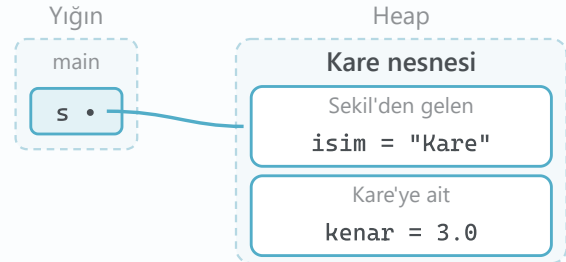
kenar = 3.0

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

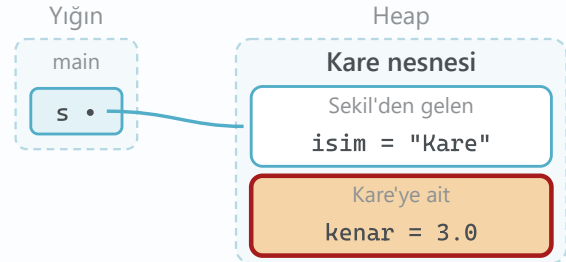
s → Kare

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     }  
15     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 YİĞİNİ

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

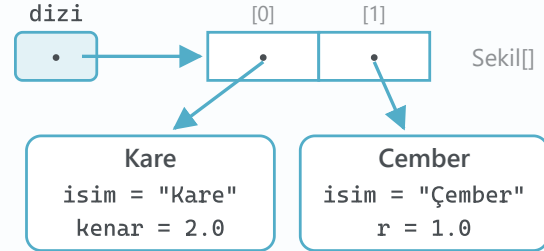
Kare kuruldu
9.0

Soyut Tipte Dizi: Dinamik Bağlama

```

1 abstract class Sekil {
2     // isim alanı ve yapıcısı
3     abstract double alan();
4     void ciz() {
5         System.out.println(isim + " çiziliyor");
6     }
7 }
8 class Kare extends Sekil {
9     // kenar alanı ve yapıcısı
10    double alan() { return kenar * kenar; }
11 }
12 class Cember extends Sekil {
13     // r alanı ve yapıcısı
14    double alan() { return Math.PI * r * r; }
15    void ciz() {
16        System.out.println("Çember, r = " + r);
17    }
18 }
19 // main içinde:
20 ▶ Sekil[] dizi = { new Kare(2), new Cember(1) };
21 for (Sekil t : dizi) {
22     System.out.println(t.alan());
23     t.ciz();
24 }

```



Dizi iki **referans** tutar. Öğelerin tipi **Sekil**, gösterdikleri nesneler **Kare** ve **Cember**.

Adım 1 / 10

ÇAĞRI YİĞİNİ

main()

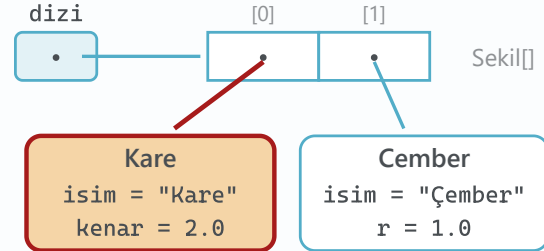
DEĞİŞKENLER

t = -

ÇIKTI

Soyut Tipte Dizi: Dinamik Bağlama

```
1 abstract class Sekil {  
2     // isim alanı ve yapıcısı  
3     abstract double alan();  
4     void ciz() {  
5         System.out.println(isim + " çiziliyor");  
6     }  
7 }  
8 class Kare extends Sekil {  
9     // kenar alanı ve yapıcısı  
10    double alan() { return kenar * kenar; }  
11 }  
12 class Cember extends Sekil {  
13     // r alanı ve yapıcısı  
14    double alan() { return Math.PI * r * r; }  
15    void ciz() {  
16        System.out.println("Çember, r = " + r);  
17    }  
18 }  
19 // main içinde:  
20 Sekil[] dizi = { new Kare(2), new Cember(1) };  
21 for (Sekil t : dizi) {  
22     System.out.println(t.alan());  
23     t.ciz();  
24 }
```



1. tur: `t`, `dizi[0]` 'daki Kare nesnesini gösterir.

Adım 2 / 10

ÇAĞRI YİĞİNİ

main()

DEĞİŞKENLER

t = → Kare

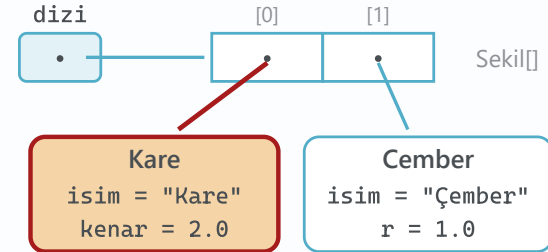
ÇIKTI

Soyut Tipte Dizi: Dinamik Bağlama

```

1 abstract class Sekil {
2     // isim alanı ve yapıcısı
3     abstract double alan();
4     void ciz() {
5         System.out.println(isim + " çiziliyor");
6     }
7 }
8 class Kare extends Sekil {
9     // kenar alanı ve yapıcısı
10    double alan() { return kenar * kenar; }
11 }
12 class Cember extends Sekil {
13     // r alanı ve yapıcısı
14     double alan() { return Math.PI * r * r; }
15     void ciz() {
16         System.out.println("Çember, r = " + r);
17     }
18 }
19 // main içinde:
20 Sekil[] dizi = { new Kare(2), new Cember(1) };
21 for (Sekil t : dizi) {
22     System.out.println(t.alan());
23     t.ciz();
24 }

```



`t.alan()` : derleyici yalnızca `Sekil`'de `alan()` olduğunu bilir; JVM nesneye bakar → `Kare.alan()`: $2.0 \cdot 2.0$.

Adım 3 / 10

ÇAĞRI YİĞİNİ

`Kare.alan()`

`main()`

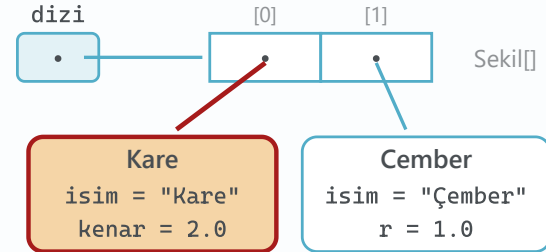
DEĞİŞKENLER

`kenar = 2.0`

ÇIKTI

Soyut Tipte Dizi: Dinamik Bağlama

```
1 abstract class Sekil {  
2     // isim alanı ve yapıcısı  
3     abstract double alan();  
4     void ciz() {  
5         System.out.println(isim + " çiziliyor");  
6     }  
7 }  
8 class Kare extends Sekil {  
9     // kenar alanı ve yapıcısı  
10    double alan() { return kenar * kenar; }  
11 }  
12 class Cember extends Sekil {  
13     // r alanı ve yapıcısı  
14     double alan() { return Math.PI * r * r; }  
15     void ciz() {  
16         System.out.println("Çember, r = " + r);  
17     }  
18 }  
19 // main içinde:  
20 Sekil[] dizi = { new Kare(2), new Cember(1) };  
21 for (Sekil t : dizi) {  
22     ▶ System.out.println(t.alan());  
23     t.ciz();  
24 }
```



Dönen 4.0 yazdırılır.

Adım 4 / 10

ÇAĞRI YİĞİNİ

main()

DEĞİŞKENLER

t = → Kare

ÇIKTI

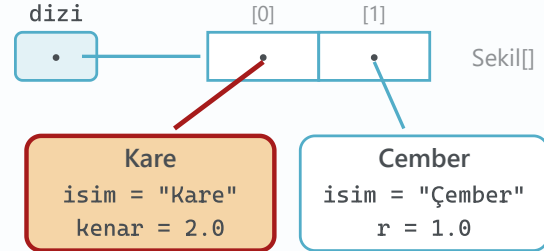
4.0

Soyut Tipte Dizi: Dinamik Bağlama

```

1 abstract class Sekil {
2     // isim alanı ve yapıcısı
3     abstract double alan();
4     void ciz() {
5         System.out.println(isim + " çiziliyor");
6     }
7 }
8 class Kare extends Sekil {
9     // kenar alanı ve yapıcısı
10    double alan() { return kenar * kenar; }
11 }
12 class Cember extends Sekil {
13     // r alanı ve yapıcısı
14    double alan() { return Math.PI * r * r; }
15    void ciz() {
16        System.out.println("Çember, r = " + r);
17    }
18 }
19 // main içinde:
20 Sekil[] dizi = { new Kare(2), new Cember(1) };
21 for (Sekil t : dizi) {
22     System.out.println(t.alan());
23     t.ciz();
24 }

```



Kare `ciz()` 'i geçersiz kılmadı (override etmedi) → Sekil'deki somut `ciz()` kalıtımla alınan metot olarak çalışır.

Adım 5 / 10

ÇAĞRI YİĞİNİ

Sekil.ciz()

main()

DEĞİŞKENLER

isim = "Kare"

ÇIKTI

4.0

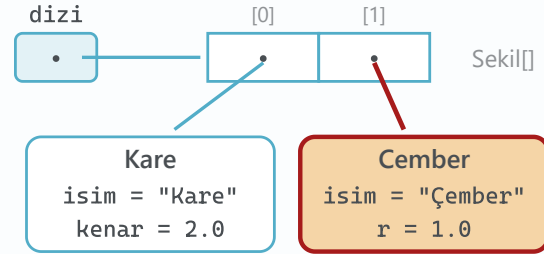
Kare çiziliyor

Soyut Tipte Dizi: Dinamik Bağlama

```

1 abstract class Sekil {
2     // isim alanı ve yapıcısı
3     abstract double alan();
4     void ciz() {
5         System.out.println(isim + " çiziliyor");
6     }
7 }
8 class Kare extends Sekil {
9     // kenar alanı ve yapıcısı
10    double alan() { return kenar * kenar; }
11 }
12 class Cember extends Sekil {
13     // r alanı ve yapıcısı
14    double alan() { return Math.PI * r * r; }
15    void ciz() {
16        System.out.println("Çember, r = " + r);
17    }
18 }
19 // main içinde:
20 Sekil[] dizi = { new Kare(2), new Cember(1) };
21 ▶ for (Sekil t : dizi) {
22     System.out.println(t.alan());
23     t.ciz();
24 }

```



2. tur: **t** artık Cember nesnesini gösterir. Aynı satır, farklı nesne.

Adım 6 / 10

ÇAĞRI YİĞİNİ

main()

DEĞİŞKENLER

t → Cember

ÇIKTI

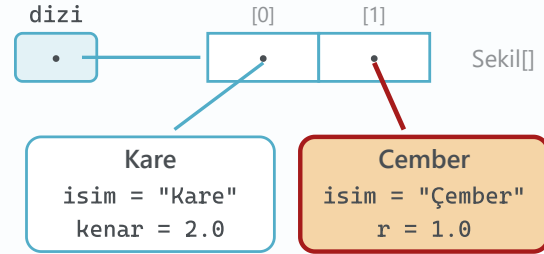
```

4.0
Kare çiziliyor

```

Soyut Tipte Dizi: Dinamik Bağlama

```
1 abstract class Sekil {  
2     // isim alanı ve yapıcısı  
3     abstract double alan();  
4     void ciz() {  
5         System.out.println(isim + " çiziliyor");  
6     }  
7 }  
8 class Kare extends Sekil {  
9     // kenar alanı ve yapıcısı  
10    double alan() { return kenar * kenar; }  
11 }  
12 class Cember extends Sekil {  
13     // r alanı ve yapıcısı  
14     double alan() { return Math.PI * r * r; }  
15     void ciz() {  
16         System.out.println("Çember, r = " + r);  
17     }  
18 }  
19 // main içinde:  
20 Sekil[] dizi = { new Kare(2), new Cember(1) };  
21 for (Sekil t : dizi) {  
22     System.out.println(t.alan());  
23     t.ciz();  
24 }
```



Bu kez **Cember.alan()** seçilir: $\pi \cdot 1.0 \cdot 1.0$.

Adım 7 / 10

ÇAĞRI YİĞİNİ

Cember.alan()

main()

DEĞİŞKENLER

r = 1.0

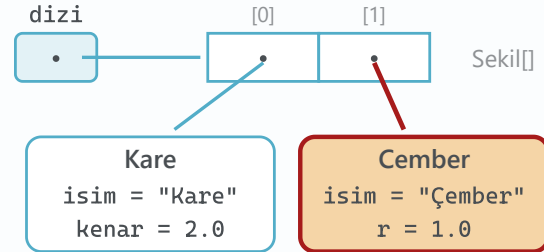
ÇIKTI

4.0

Kare çiziliyor

Soyut Tipte Dizi: Dinamik Bağlama

```
1 abstract class Sekil {  
2     // isim alanı ve yapıcısı  
3     abstract double alan();  
4     void ciz() {  
5         System.out.println(isim + " çiziliyor");  
6     }  
7 }  
8 class Kare extends Sekil {  
9     // kenar alanı ve yapıcısı  
10    double alan() { return kenar * kenar; }  
11 }  
12 class Cember extends Sekil {  
13     // r alanı ve yapıcısı  
14    double alan() { return Math.PI * r * r; }  
15    void ciz() {  
16        System.out.println("Çember, r = " + r);  
17    }  
18 }  
19 // main içinde:  
20 Sekil[] dizi = { new Kare(2), new Cember(1) };  
21 for (Sekil t : dizi) {  
22     ▶ System.out.println(t.alan());  
23     t.ciz();  
24 }
```



Dönen değer yazdırılır: `Math.PI`.

Adım 8 / 10

ÇAĞRI YİĞİNİ

`main()`

DEĞİŞKENLER

`t` → `Cember`

ÇIKTI

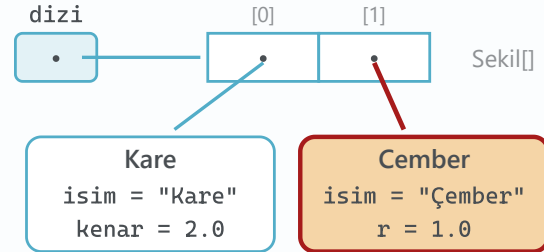
```
4.0  
Kare çiziliyor  
3.141592653589793
```

Soyut Tipte Dizi: Dinamik Bağlama

```

1 abstract class Sekil {
2     // isim alanı ve yapıcısı
3     abstract double alan();
4     void ciz() {
5         System.out.println(isim + " çiziliyor");
6     }
7 }
8 class Kare extends Sekil {
9     // kenar alanı ve yapıcısı
10    double alan() { return kenar * kenar; }
11 }
12 class Cember extends Sekil {
13     // r alanı ve yapıcısı
14    double alan() { return Math.PI * r * r; }
15    void ciz() {
16        System.out.println("Çember, r = " + r);
17    }
18 }
19 // main içinde:
20 Sekil[] dizi = { new Kare(2), new Cember(1) };
21 for (Sekil t : dizi) {
22     System.out.println(t.alan());
23     t.ciz();
24 }

```



Cember `ciz()` 'i geçersiz kılmış → kendi sürümü:
dinamik bağlama (dynamic binding).

Adım 9 / 10

ÇAĞRI YİĞİNİ

Cember.ciz()

main()

DEĞİŞKENLER

r = 1.0

ÇIKTI

```

4.0
Kare çiziliyor
3.141592653589793
Çember, r = 1.0

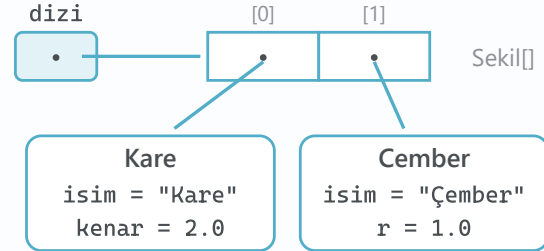
```

Soyut Tipte Dizi: Dinamik Bağlama

```

1 abstract class Sekil {
2     // isim alanı ve yapıcısı
3     abstract double alan();
4     void ciz() {
5         System.out.println(isim + " çiziliyor");
6     }
7 }
8 class Kare extends Sekil {
9     // kenar alanı ve yapıcısı
10    double alan() { return kenar * kenar; }
11 }
12 class Cember extends Sekil {
13     // r alanı ve yapıcısı
14    double alan() { return Math.PI * r * r; }
15    void ciz() {
16        System.out.println("Çember, r = " + r);
17    }
18 }
19 // main içinde:
20 Sekil[] dizi = { new Kare(2), new Cember(1) };
21 for (Sekil t : dizi) {
22     System.out.println(t.alan());
23     t.ciz();
24 }

```



Öğe kalmadı, döngü biter. Yeni bir şekil (ör. Ucgen) eklemek için bu döngüyü **değiştirmek gerekmez**.

Adım 10 / 10

ÇAĞRI YİĞİNİ

main()

DEĞİŞKENLER

t ⇒ Cember

ÇIKTI

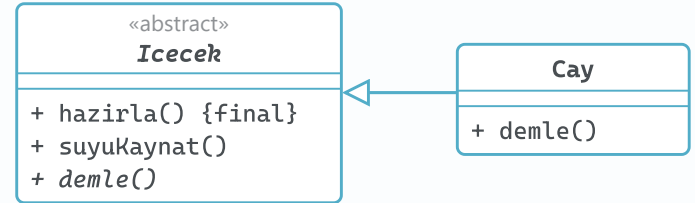
```

4.0
Kare çiziliyor
3.141592653589793
Çember, r = 1.0

```

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:
18 ► Icecek i = new Cay();
19 i.hazirla();
```



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

Adım 1 / 8

ÇAĞRI YİĞİNİ

`main()`

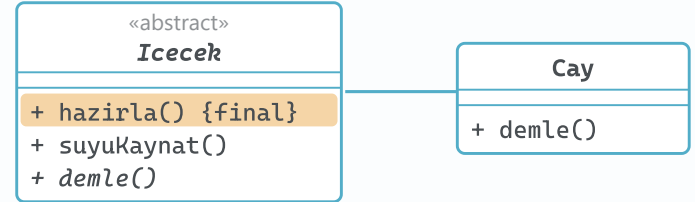
DEĞİŞKENLER

`i = → 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() {  
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 YİĞİNİ

hazirla()

main()

DEĞİŞKENLER

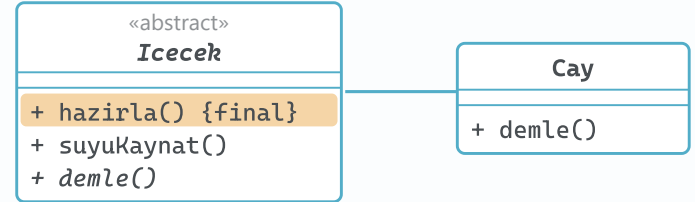
this = → Cay

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() {  
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 YİĞİNİ

hazirla()

main()

DEĞİŞKENLER

this = → Cay

ÇIKTI

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

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();  
19 i.hazirla();
```



Icecek'teki somut `suyuKaynat()` çalışır.

Adım 4 / 8

ÇAĞRI YİĞİNİ

`suyuKaynat()`

`hazirla()`

`main()`

DEĞİŞKENLER

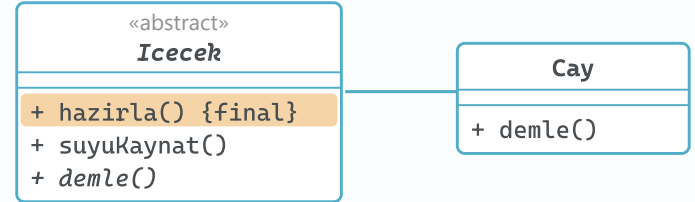
`this = → Cay`

ÇIKTI

Su kaynadı

Somut Metot Soyut Metodu Çağırır

```
1 abstract class Icecek {  
2     final void hazirla() {  
3         suyuKaynat();  
4     }  
5     System.out.println("Servis edildi");  
6 }  
7 void suyuKaynat() {  
8     System.out.println("Su kaynakı");  
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 YİĞİNİ

hazirla()

main()

DEĞİŞKENLER

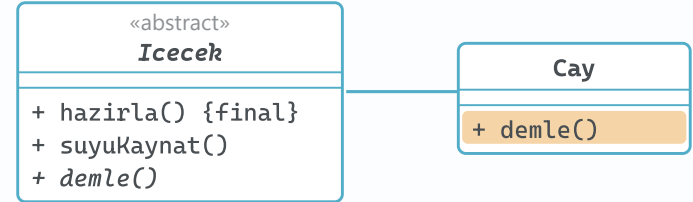
this = → Cay

ÇIKTI

Su kaynakı

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();
19 i.hazirla();
```



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

Adım 6 / 8

ÇAĞRI YİĞİNİ

Cay.demle()

hazirla()

main()

DEĞİŞKENLER

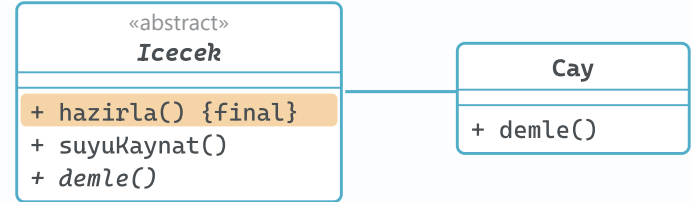
this = → Cay

ÇIKTI

```
Su kaynadı
Çay demlendi
```

Somut Metot Soyut Metodu Çağırır

```
1 abstract class Icecek {  
2     final void hazirla() {  
3         suyuKaynat();  
4         demle();  
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();
```



3. adım: servis mesajı yazdırılır.

Adım 7 / 8

ÇAĞRI YİĞİNİ

hazirla()

main()

DEĞİŞKENLER

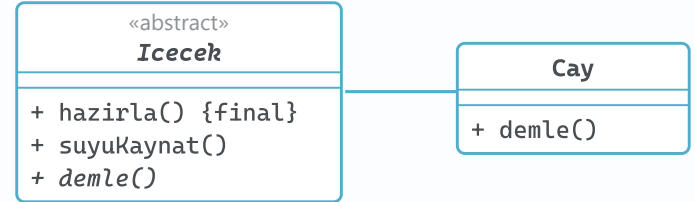
this = → Cay

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 YİĞİNİ

main()

DEĞİŞKENLER

i = → Cay

ÇIKTI

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

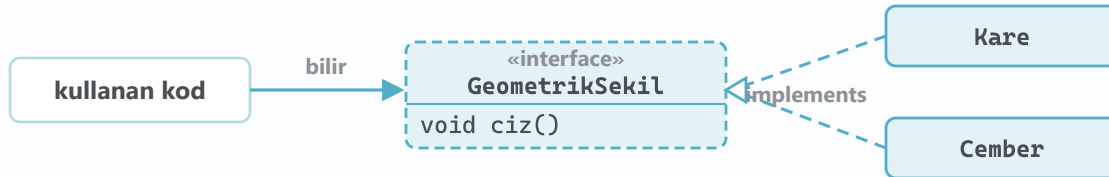
Arayüz (Interface) Nedir?

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

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

Derleyicinin gördüğü

```
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`.

Arayüz Gerçekleştirme (implements)

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

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

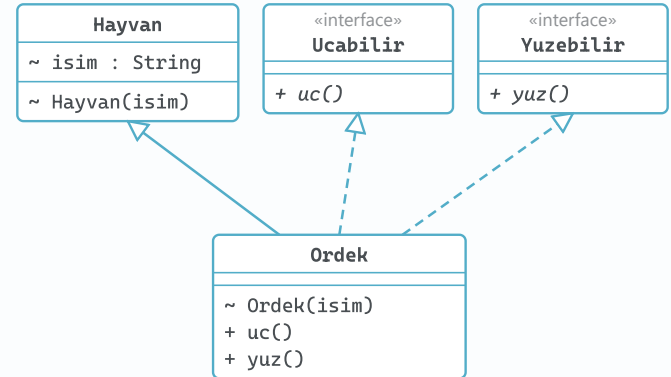
Çıktı

```
Kare çiziliyor  
4
```

ARRAYÜZLER

Bir Sınıf, Birden Fazla Arayüz

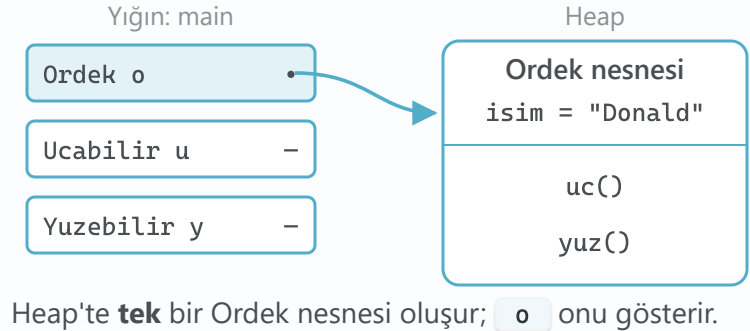
- Bir sınıf **en fazla bir** üst sınıfı genişletebilir (`extends`)...
- ...ama **sıfır ya da daha çok** arayüzü gerçekleştirebilir (`implements A, B, ...`).
- Önce `extends`, sonra `implements` yazılır.
- `class C extends A, B` → `error: '{' expected`.



```
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);
```



Adım 1 / 8

ÇAĞRI YİĞİNİ

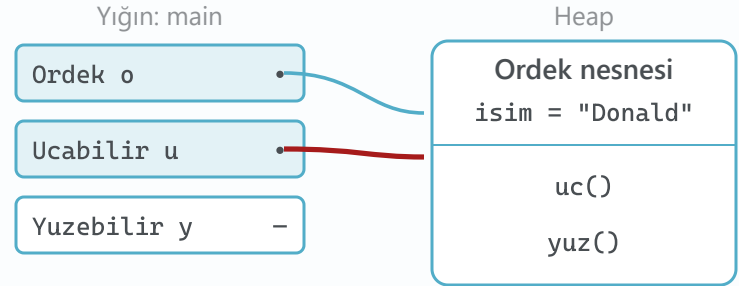
main()

ÇIKTI



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);
8 System.out.println(u == o);
```



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

Adım 2 / 8

ÇAĞRI YİĞİNİ

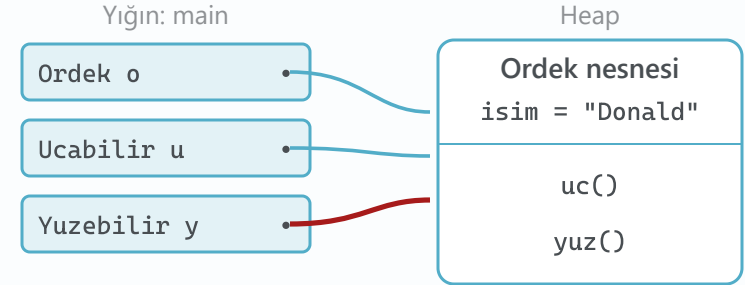
main()

ÇIKTI



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);
8 System.out.println(u == o);
```



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

Adım 3 / 8

ÇAĞRI YIĞINI

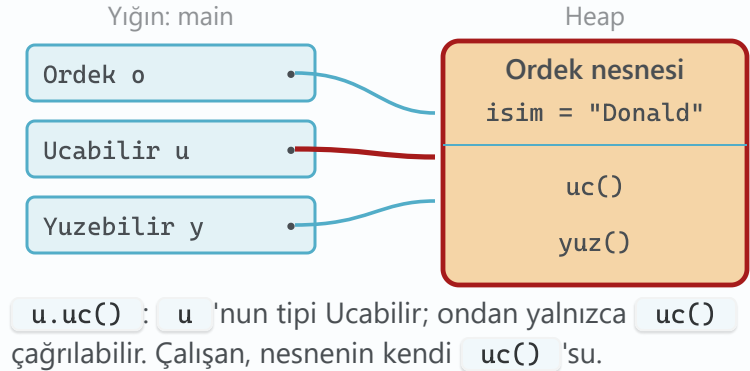
main()

ÇIKTI



Aynı Nesne, Farklı Arayüz Referansları

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



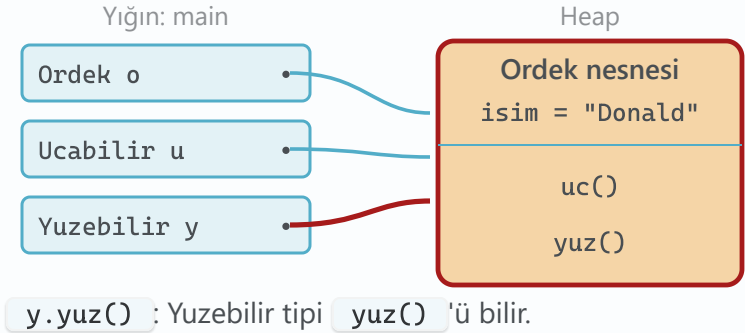
Adım 4 / 8



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);
8 System.out.println(u == o);
    
```

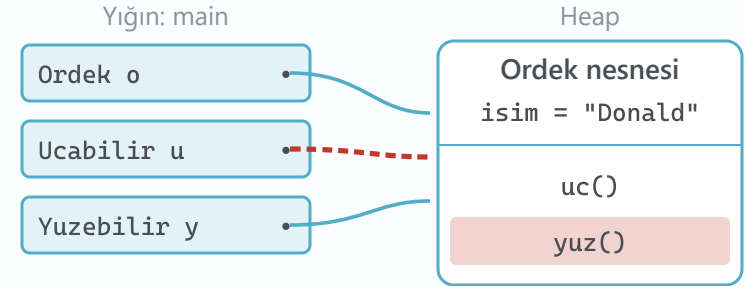


Adım 5 / 8



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);
8 System.out.println(u == o);
```



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

Adım 6 / 8

ÇAĞRI YİĞİNİ

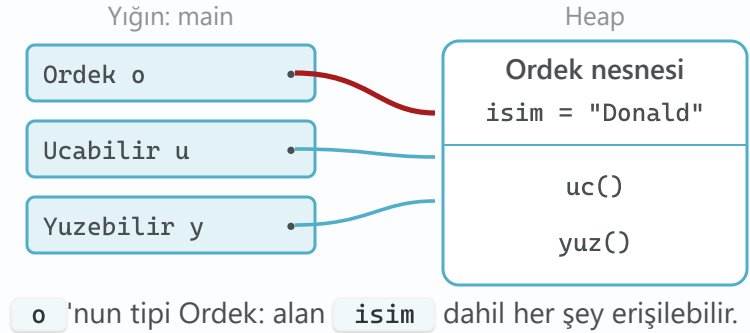
`main()`

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!
7 ► System.out.println(o.isim);
8 System.out.println(u == o);
```



Adım 7 / 8

ÇAĞRI YİĞİNİ

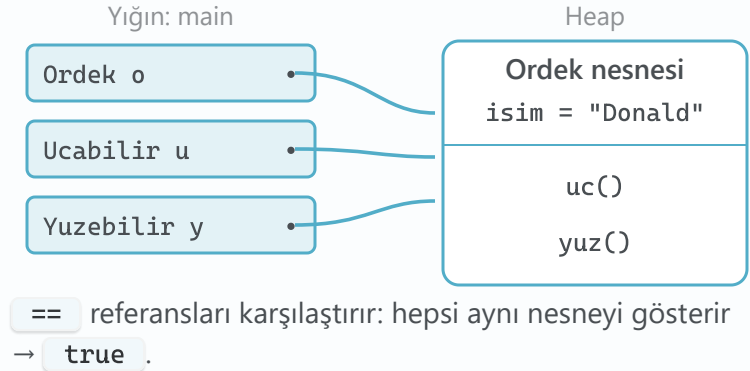
main()

ÇIKTI

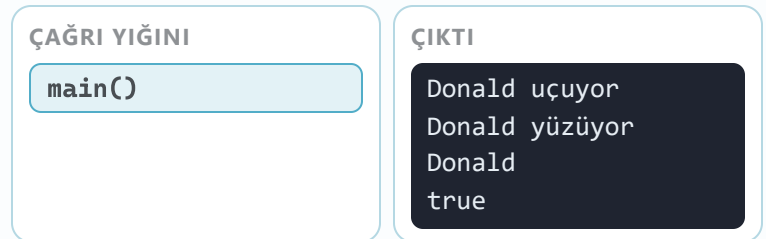
```
Donald uçuyor
Donald yüzüyor
Donald
```

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);
```



Adım 8 / 8

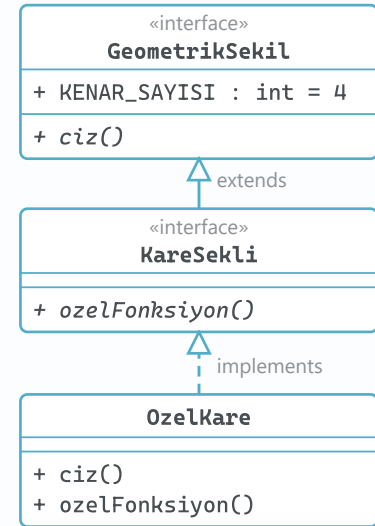


ARAYÜZLER

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
```

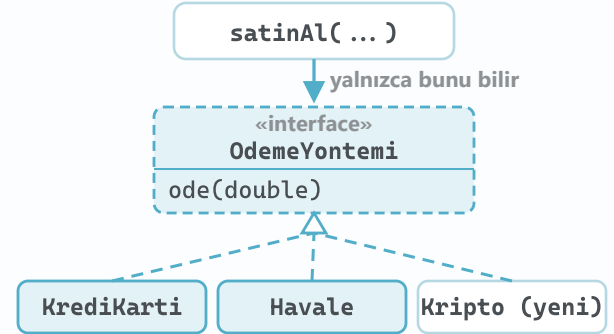


Çıktı

```
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 gerçekleştirir; `satinAl` hiç değişmez (gevşek bağlılık).
- Test kolaylaşır: gerçek banka yerine arayüzü gerçekleştiren **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 }
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);
    
```



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);

```

Yığın

KrediKarti.ode()
tutar 250.0

Heap

KrediKarti #1

satinAl()
y
toplam 250.0

main()

y.ode() → KrediKarti.ode()

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 }
15 ▶ 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

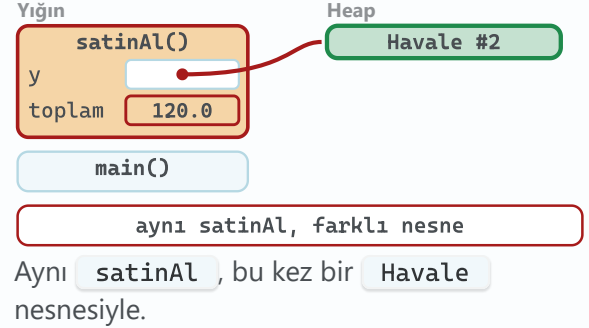
ÇIKTI

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 }
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);
    
```



Adım 4 / 6

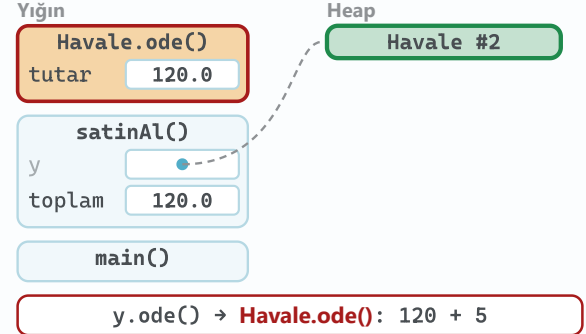
ÇIKTI

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 {
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);
    
```



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 }
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);
    
```

Yığın

Heap

main()

satinAl hiç değişmeden iki davranış

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
    
```

Varsayılan Metotlar (default, Java 8+)

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

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

Çıktı

```
true  
2 false
```

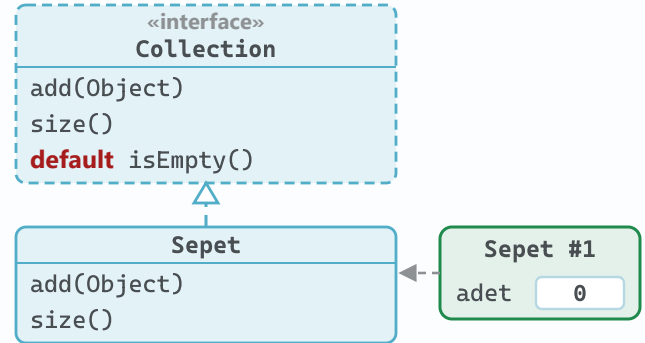
```
class Sepet implements Collection {  
    private int adet = 0;  
    public void add(Object o) { adet++; }  
    public void remove(Object o) { adet--; }  
    public int size() { return adet; }  
    // isEmpty() yazılmadı: default sürüm kullanılır  
}  
  
// main içinde:  
Sepet sepet = new Sepet();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

`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 }
13 ► 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

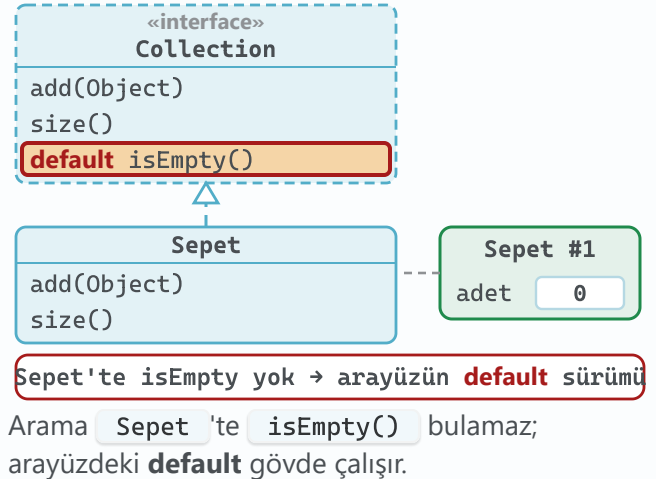
ÇIKTI



Varsayılan Metot: Kod İzleme

```

1 interface Collection {
2     void add(Object o);
3     int size();
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());
    
```



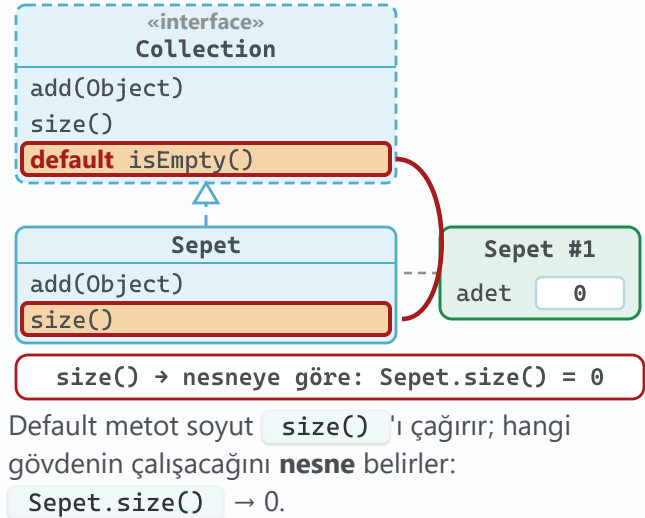
Adım 2 / 6

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());
    
```



Adım 3 / 6

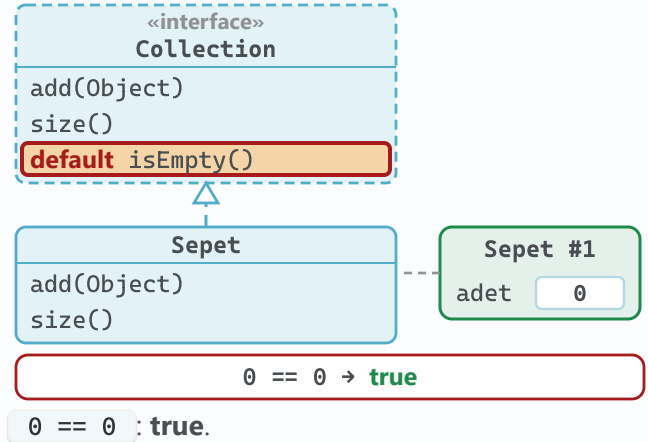
ÇIKTI



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());
    
```



Adım 4 / 6

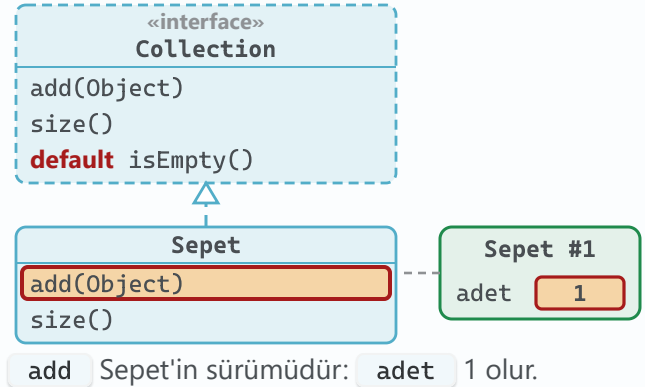
ÇIKTI

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;
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());
    
```



Adım 5 / 6

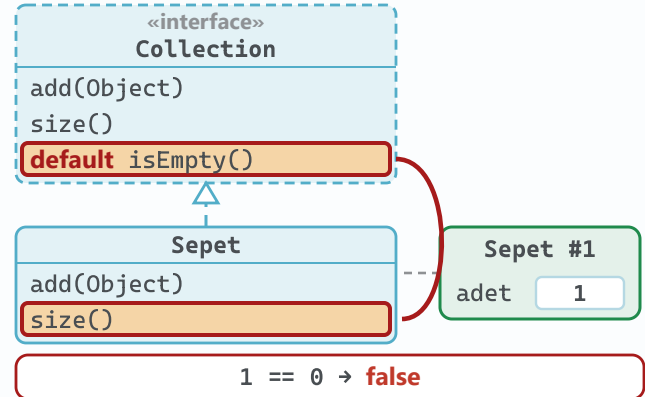
ÇIKTI

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;
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());
    
```



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
    
```

ARRAYÜZ METOTLARI

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)`.
- Gerçekleştiren 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

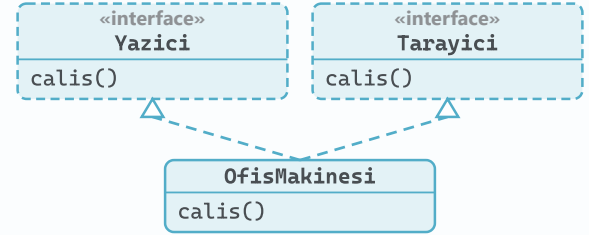
`selam()` yazılmasaydı

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();
    
```



iki default calis() → sınıf **kendisi** yazmak zorunda

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

Adım 1 / 6

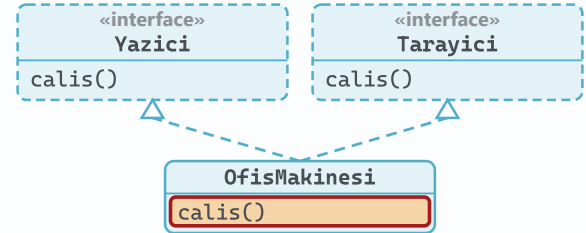
ÇIKTI



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

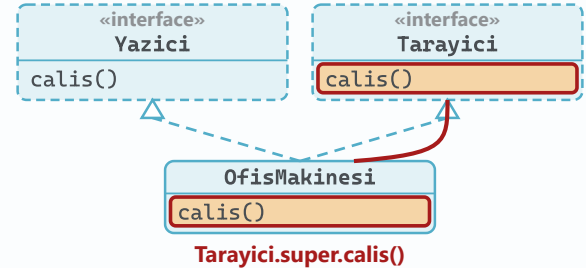
ÇIKTI



Default Çakışması: Kod İzleme

```

1 interface Yazici {
2     default void calis() { System.out.println("Yazdırılıyor"); }
3 }
4 interface Tarayici {
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();
    
```



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

Adım 3 / 6

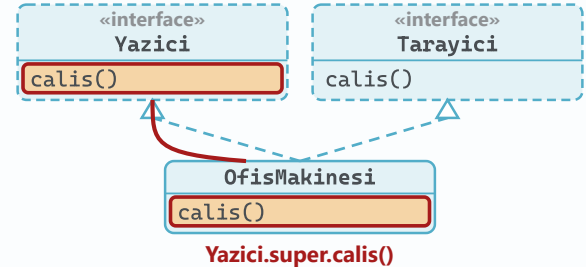
ÇIKTI

Taranıyor

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();
    
```



Sonra **Yazici** 'ninki. Hangi sürümlerin hangi sırayla kullanılacağını sınıf seçer.

Adım 4 / 6

ÇIKTI

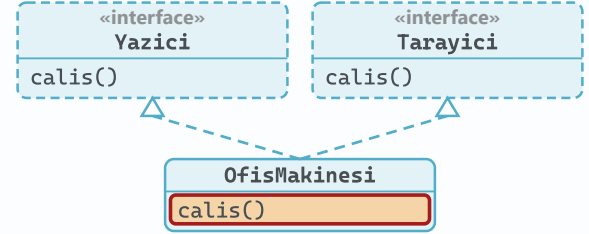
```

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();
11        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

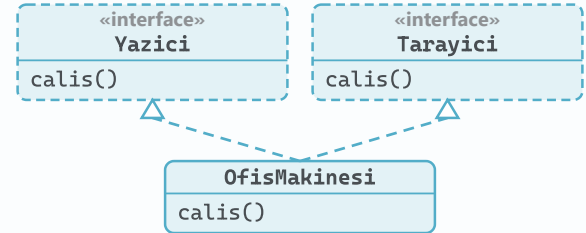
```

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
15 y.calis();
    
```



calis() yazılmasaydı: derleme hatası

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

Adım 6 / 6

ÇIKTI

```

Taranıyor
Yazdırılıyor
Kopya hazır
    
```

ARAYÜZ METOTLARI

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.
- Gerçekleştiren sınıf onları **göremez**: `Rapor` içinde `baslik()` çağrılmaz.
- Akış: `yazdir()` (default) → `baslik()` (private) + `icerik()` (Rapor'un gerçekleştirdiği soyut metot).



Arayüz Üyeleri: Özet Tablosu

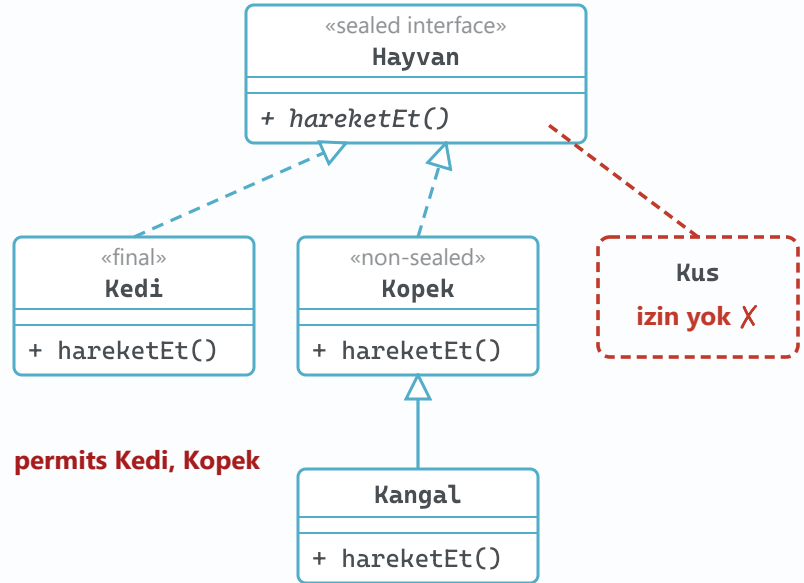
Üye	Örnek	Örtük belirteçler	Gövde	Java
Soyut metot	<code>void ciz();</code>	<code>public abstract</code>	yok	1.0
Sabit	<code>int KENAR_SAYISI = 4;</code>	<code>public static final</code>	—	1.0
Varsayılan metot	<code>default boolean isEmpty() {...}</code>	<code>public</code>	var	8
static metot	<code>static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

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

MÜHÜRLÜ TIPLER

Mühürleme: sealed, non-sealed, permits

- Sınıflar ve arayüzler **mühürlenebilir** (Java 17+).
- sealed** : tipi **yalnızca izin verilen** alt tipler genişletebilir ya da gerçekleştirebilir.
- 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 TIPLER

Mühürlü Arayüz Örneği

```
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 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
```

Arayüz ve Soyut Sınıf Karşılaştırması

Özellik	Arayüz	Soyut sınıf
Tanım / kullanım	<code>interface</code> · <code>implements</code>	<code>abstract class</code> · <code>extends</code>
Metotlar	Soyut; Java 8+ <code>default</code> ve <code>static</code> , Java 9+ <code>private</code>	Hem soyut hem somut metotlar
Alanlar	Yalnızca <code>public static final</code> sabitler; örnek alanı (instance field) olamaz	Her türlü alan (örnek alanları dahil)
Yapıcı	Yok	Olabilir; alt sınıf <code>super(...)</code> ile çağırır
Erişim	Üyeler örtük <code>public</code> (private yardımcılar hariç)	<code>public</code> , <code>protected</code> , <code>private</code> , paket
Kalıtım	Bir sınıf çok sayıda arayüz gerçekleştirir; 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 <code>new</code> ile nesne oluşturulamaz; ikisi de referans tipi olarak kullanılır	

KARŞILAŞTIRMA

Hangisini Seçmeli?

Soyut sınıf seçin

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

Arayüz seçin

- + **İlgisiz** sınıflara aynı yetenek: Kus, Ucak, Drone hepsi Ucabilir.
- + Bir sınıfa **birden çok** yetenek kazandırmak.
- + Kullanan kodla gerçekleştiren 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 derleyici size 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
```

```
Merhaba  
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 gerçekleştirme her tip için çalışır.

Jenerik Sınıf ve Jenerik Metot

```
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("Öge: " + 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
Öge: 3.14
Öge: 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** öge, **K** anahtar, **V** değer, **R** sonuç.

Sınırlı Tip Parametresi

`<T extends Ust>` : `T` yalnızca `Ust` ya da onun alt tipi olabilir (üst sınır). Karşılığında metod 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 metod 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 gerçekleştirir.
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

7	12	5
---	----	---

eb

7

Derleyici T 'yi Integer olarak çıkarır. İlk öge 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];  
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));  
14 System.out.println(enBuyuk(adlar));
```

T = Integer



i = 1

eb 12

compareTo = 1 > 0 → eb = 12

compareTo pozitifse öge 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];  
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));  
14 System.out.println(enBuyuk(adlar));
```

T = Integer



i = 2

eb



compareTo = -1 → değişmez

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     }  
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 = Integer



eb



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) {  
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:  
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

"elma"

"kiraz"

"armut"

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];  
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));  
14 System.out.println(enBuyuk(adlar));
```

T = String

"elma"

"kiraz"

"armut"

↓
i = 1

eb

"kiraz"

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];  
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));  
14 System.out.println(enBuyuk(adlar));
```

T = String

"elma"

"kiraz"

"armut"

↓
i = 2

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     }
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

"elma"

"kiraz"

"armut"

eb

"kiraz"

kiraz döner. Tek gerçekleştirme iki tipte çalıştı; sınır sayesinde `compareTo` çağırısı 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*.

JENERİKLER

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()** ve **new T[10]** yazılamaz (**unexpected type**).

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?
<code>@Override</code>	derleyici	Üst tipte böyle bir metot yoksa hata verir
<code>@Deprecated</code>	derleyici	Eskimiş öge; kullanan kod uyarı alır
<code>@SuppressWarnings("unchecked")</code>	derleyici	Belirtilen uyarıyı susturur
<code>@FunctionalInterface</code>	derleyici	Arayüzde tam bir soyut metot olduğunu denetler
<code>@Test</code> (JUnit)	test aracı	Test metotlarını bulur ve çalıştırır

@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
```

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.

ÖZET

Özet

Kavram	Özü
Soyutlama	Ayrıntıyı gizleyip "ne yapılır"ı tanımlamak; "nasıl"ı alt sınıflara bırakmak
Soyut sınıf	<code>abstract class</code> ; nesnesi olmaz; alan, yapıcı, soyut ve somut metot içerebilir
Soyut metot	Gövdesiz, <code>;</code> ile biter; somut alt sınıf gerçekleştirmek zorunda
Dinamik bağlama	Derleyici referans tipine, JVM nesnenin gerçek tipine bakar
Arayüz	Sözleşme; metotlar örtük <code>public abstract</code> , alanlar <code>public static final</code>
implements	Tek <code>extends</code> , çok sayıda <code>implements</code> ; gerçekleştiren metot <code>public</code>
default / static / private	Gövdeli arayüz metotları (Java 8 / 8 / 9)
sealed	<code>permits</code> ile izinli alt tipler; her biri <code>final</code> , <code>sealed</code> ya da <code>non-sealed</code>
Jenerikler	<code>Kutu<T></code> , <code><T extends Ust></code> , joker <code>?</code> ; denetim derlemede, tip bilgisi sonra silinir
Etiketler	<code>@Override</code> gibi üst veri; <code>RUNTIME</code> etiketleri yansımayla okunur

ÖZET

İlgili Sayfalar

Dinamik Bağlama

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

Yığın ve Heap Gezgini

Referansların, nesnelerin ve metod çerçevelerinin bellekte nasıl durduğunu gösteren simülatör.

Çok Biçimlilik

Arayüz ve soyut sınıf tipindeki referanslarla çalışan dinamik bağlamanın temelleri.

İpucu: soyut metodu ya da arayüz metodunu gerçekleştirirken `@Override` yazın. Adı ya da parametreyi yanlış yazarsanız derleyici hemen uyarır.

Sık Yapılan Hatalar (1): Soyut Sınıflar

Soyut sınıftan nesne

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

Soyut metodu unutmak

`Cember` is not abstract and does not override abstract method `ciz()` in `Sekil`. Metodu yazın ya da sınıfı `abstract` yapın.

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 metodunu `public` yazın.

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`. Uygun tipte referans kullanın.

Çakışan default metotlar

`class Gorevli inherits unrelated defaults for selam()`. Metodu sınıfta geçersiz kılın.

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 Balık extends Hayvan {  
    String ad() { return "balık"; }  
}  
  
// main içinde:  
Hayvan[] d = { new Kedi(), new Balık() };  
System.out.println();  
for (Hayvan h : d)  
    System.out.println(h.ad() + ": " + h.ses());
```

1. Program ne yazdırır?
2. `Balık`, `ses()` 'i yazmadığı hâlde neden derlenir?
3. `Kedi` 'deki `ses()` 'ten `public` silinirse ne olur?
4. `Sesli s = new Balık(); 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 `Balık`.
6. `class Kare implements GeometrikSekil` içinde `public void Ciz() { ... }` yazıldı. Ne olur? Metodun üstüne `@Override` eklenirse?

► Cevaplar / ipuçları

Kaynaklar

- Oracle, *The Java Tutorials*: Abstract Methods and Classes; Interfaces; Default Methods.
- 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).
- 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).

Son

Bölüm 4: Soyut Sınıflar ve Arayüzler

Sonraki bölüm: Bölüm 4: Örnek Sorular · [Terim sözlüğü](#)