Bölüm 4.3: Kalıtım

Java ile Nesne Yönelimli Programlama

Üst ve alt sınıf | extends | super ve yapıcı zinciri | Geçersiz kılma ve @Override | protected | final | is-a ve has-a | Kalıtım mı, oluşum mu? | Çoklu kalıtım | Object sınıfı | toString, equals, hashCode

Sercan KÜLCÜ · Giresun Üniversitesi · Elektrik-Elektronik Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Kavram

Kalıtımı (inheritance), üst sınıf ve alt sınıf kavramlarını tanımlamak; “is-a” ilişkisini tanımak.

Söz dizimi

`extends` ile alt sınıf yazmak; neyin kalıtımla alındığını, neyin alınmadığını açıklamak.

Yapıcılar

`super(...)` ve `super.metot()` kullanmak; yapıcı zincirinin çalışma sırasını izlemek.

Geçersiz kılma

Metodu geçersiz kılmak (overriding), aşırı yüklemeden (overloading) ayırmak, `@Override` kullanmak.

Erişim ve sınırlama

`protected` erişimini ve `final` ile kalıtımın nasıl sınırlandığını açıklamak.

Object sınıfı

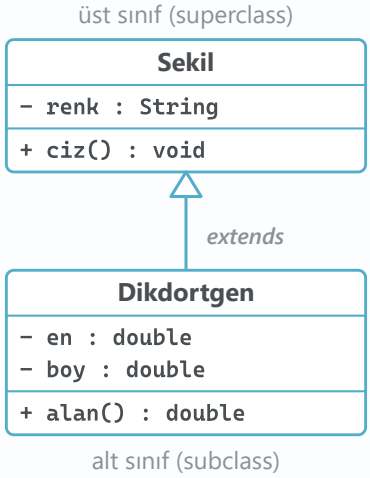
`toString`, `equals` ve `hashCode` metotlarını doğru yazmak; `==` ile `equals` farkını bilmek.

KALITIM KAVRAMI

Kalıtım Nedir?

- **Kalıtım (Inheritance)**: mevcut bir sınıfın alan ve metotlarını kalıtımla alıp yeni veya değiştirilmiş yeteneklerle zenginleştirerek **yeni bir sınıf** oluşturmaktır.
- Mevcut sınıf: **üst sınıf** (superclass).
- Yeni sınıf: **alt sınıf** (subclass).
- Alt sınıfa yeni alan ve metotlar eklenebilir; kalıtımla alınan alanlar **yeniden tanımlanmaz**.

Benzetme: "Her dikdörtgen bir şekildir." Dikdörtgen, şeklin her şeyine sahiptir; üstüne kendi `en`, `boy` ve `alan()` bilgisini ekler.



KALITIM KAVRAMI

extends ile Alt Sınıf

```
// Temel sınıf (üst sınıf)
public class Sekil {
    // Temel sınıfa ait nitelik ve metotlar
    // ...
}

// Yeni sınıf (alt sınıf), Sekil'den türer
public class Dikdortgen extends Sekil {
    // Yeni sınıfa ait nitelik ve metotlar
    // ...
}
```

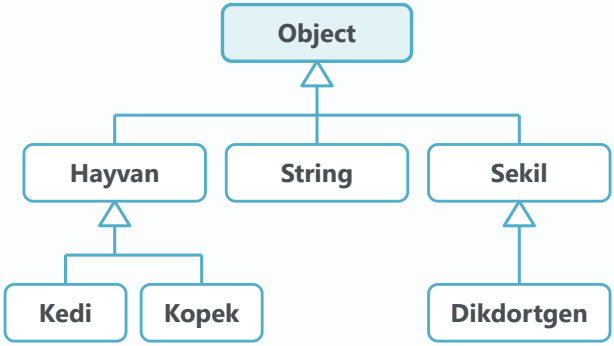
- Söz dizimi: `class AltSinif extends UstSinif` . `extends` : "genişletir".
- `extends` sonrasına **yalnızca bir** sınıf yazılabilir.
- Alt sınıf, üst sınıfın `public` ve `protected` üyelerini kendi üyesiymiş gibi kullanır.
- Üst sınıftaki aynı adlı alanı alt sınıfta yeniden tanımlamak onu **gizler** (hiding); iki ayrı değişken oluşur ve kafa karıştırır. Yapılmamalıdır.
- `extends` yazılmazsa sınıf, gizlice `Object` sınıfından türer.



KALITIM KAVRAMI

Alt ve Üst Sınıf

- Alt sınıf, üst sınıfın davranışlarını sergiler; **ek olarak** kendine özgü davranışlar da sergiler.
- Kalıtım bu yüzden **özelleştirme** (specialization) olarak da adlandırılır: alt sınıf daha **özel** bir grubu temsil eder.
- Her kedi bir hayvandır, ama her hayvan kedi değildir.
- Java sınıf hiyerarşisi, `java.lang` paketindeki `Object` sınıfı ile başlar.
- Her sınıf, **doğrudan** veya **dolaylı** olarak `Object` sınıfından kalıtımla alır.



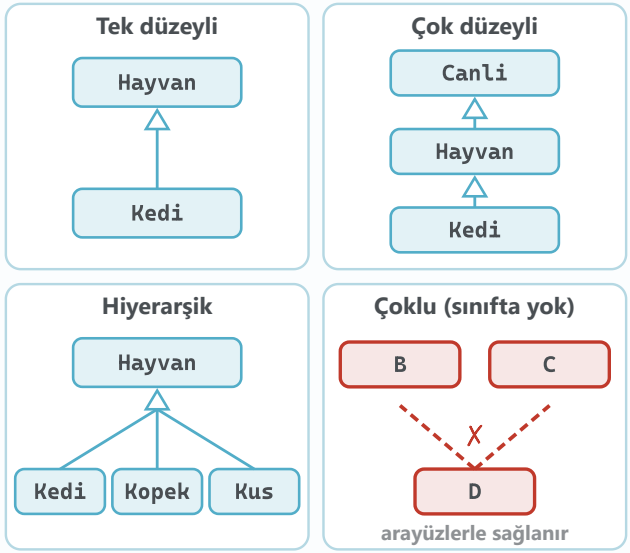
yukarı: genel · aşağı: özel

KALITIM KAVRAMI

Kalıtım Türleri

- **Tek düzeyli** (single): bir alt sınıf, bir üst sınıf. `Kedi extends Hayvan`.
- **Çok düzeyli** (multilevel): zincir biçiminde. `Kedi`, `Canli`'nin üyelerini `Hayvan` üzerinden alır; her düzey bir öncekinin her şeyini taşır.
- **Hiyerarşik** (hierarchical): bir üst sınıftan birçok alt sınıf. Ortak kod bir kez yazılır, her alt sınıf yalnızca farkını ekler.
- **Çoklu** (multiple): bir sınıfın birden çok üst sınıfı. Java sınıflarda buna izin vermez; aynı ihtiyaç arayüzlerle karşılanır.

Java'da sınıf hiyerarşisi bir **ağaçtır**: kökü `Object`'tir ve `Object` dışındaki her sınıfın tam olarak **bir** doğrudan üst sınıfı vardır. Zincirin derinliği sınırsızdır, ama derin zincirler okunmayı ve değiştirmeyi zorlaştırır.



KALITIM KAVRAMI

Neler Kalıtımla Alınır?

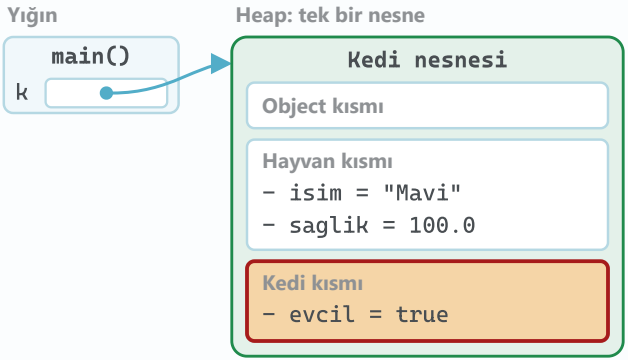
Üye	Alt sınıfa geçer mi?	Not
`public` / `protected` alan ve metotlar	**Evet**	Alt sınıfta doğrudan kullanılır.
Belirteçsiz (paket erişimli) üyeler	Yalnızca aynı paketteyse	Başka paketteki alt sınıf erişemez.
`private` alanlar	Nesnede **yer alır**, ama doğrudan erişilemez	Üst sınıfın `public` / `protected` metotlarıyla (getter, setter) kullanılır.
`private` metotlar	Hayır	Görünmez; geçersiz kılınamaz.
Yapıcılar (constructors)	**Hayır**	Kalıtımla alınmaz; alt sınıf yapıcısı `super(...)` ile çağırır.
`static` üyeler	Erişilebilir	Sınıfa aittir; geçersiz kılınmaz, yalnızca gizlenebilir.

Kısaca: alt sınıf nesnesinin içinde üst sınıfın bütün alanları vardır; hangisine **doğrudan** erişilebileceğini erişim belirteci belirler.

KALITIM KAVRAMI

Alt Sınıf Nesnesinin Yapısı

- `new Kedi(...)` **tek** bir nesne oluşturur; üst sınıf için ayrı bir nesne oluşmaz.
- Nesnede üst sınıftan gelen alanlar için de yer ayrılır. `private` olsalar bile oradadırlar; yalnızca `Kedi` 'nin kodu onlara adıyla erişemez.
- `super` ayrı bir nesne değildir: aynı nesnenin üst sınıf kısmına, üst sınıfın kodu üzerinden ulaşmanın yoludur.



this ve super aynı nesnedir

KANIT: YAPICIDAKİ `getClass()`

```
class Hayvan {
    Hayvan() {
        System.out.println("Hayvan() içinde: "
            + getClass().getSimpleName());
    }
}
class Kedi extends Hayvan { }
// main içinde:
new Kedi();
new Hayvan();
```

```
Hayvan() içinde: Kedi
Hayvan() içinde: Hayvan
```

`Hayvan` yapıcısı çalışırken bile nesnenin gerçek sınıfı `Kedi` 'dir: yapıcı zinciri, **aynı** nesnenin kısımlarını sırayla kurar.

KALITIM KAVRAMI

Ortak Kodu Üst Sınıfa Taşımak

Genelleştirme (generalization): birkaç sınıfta tekrar eden alan ve metotları bulup ortak bir üst sınıfa taşımak. Özelleştirmenin ters yönüdür.

ÖNCE: AYNI KOD İKİ YERDE

```
class Ogrenci {
    private String ad;
    private int yas;
    private double ortalama;
    String kimlik() { return ad + " (" + yas + ")"; }
}
class Ogretmen {
    private String ad;
    private int yas;
    private String brans;
    String kimlik() { return ad + " (" + yas + ")"; }
}
```

Yaş biçimi değişirse iki sınıf da düzeltilmeli; biri unutulursa çıktılar tutarsız olur.

SONRA: ORTAK KISIM `Kisi` 'DE

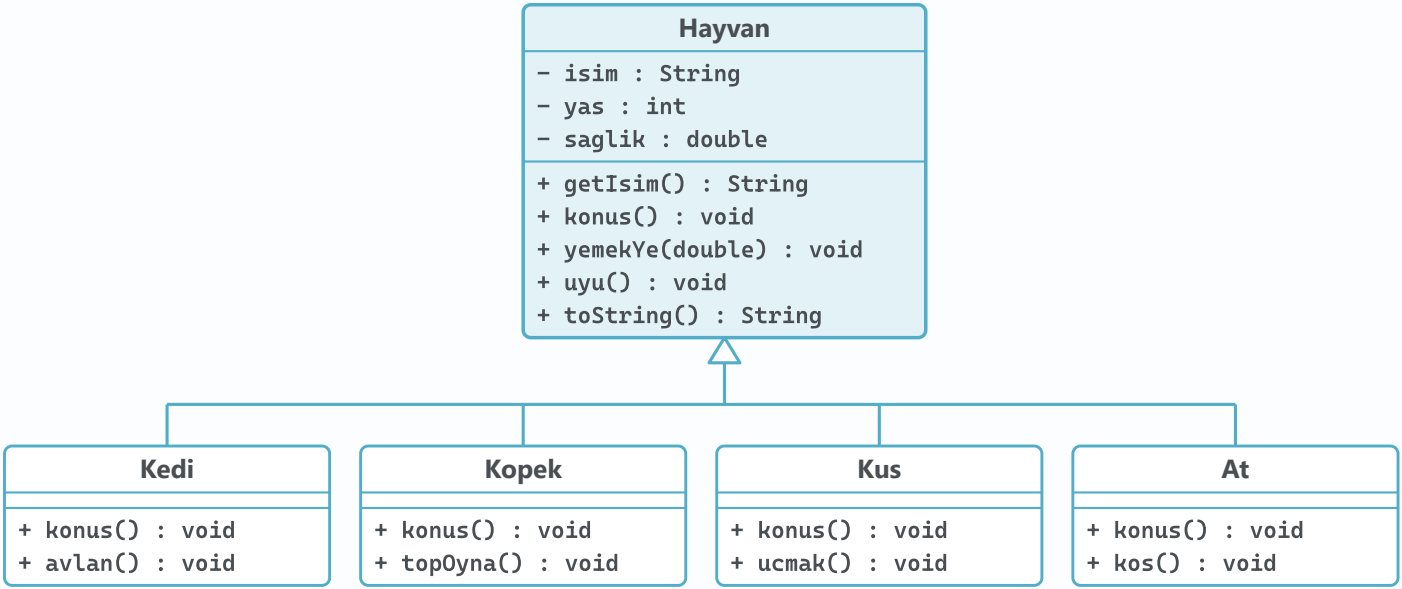
```
class Kisi {
    private String ad;
    private int yas;
    Kisi(String ad, int yas) { this.ad = ad; this.yas = yas; }
    String kimlik() { return ad + " (" + yas + ")"; }
}
class Ogrenci extends Kisi {
    private double ortalama;
    Ogrenci(String ad, int yas, double ortalama) {
        super(ad, yas);
        this.ortalama = ortalama;
    }
    @Override String kimlik() {
        return super.kimlik() + ", ort. " + ortalama;
    }
}   // Ogretmen de aynı biçimde: brans eklenir
```

```
Ece (20), ort. 3.4
Murat (41), Fizik
```

Taşımadan önce sorulacak soru: “Her öğrenci bir kişi midir?” Evet ise genelleştirme doğrudur; yalnızca iki sınıfın birkaç alanı benziyorsa oluşum daha uygundur.

HAYVAN ÖRNEĞİ

Hayvan Sınıf Hiyerarşisi



Ortak alan ve metotlar **bir kez**, `Hayvan` sınıfında yazılır. Her alt sınıf `konus()` metodunu kendine göre **geçersiz kılar** ve kendine özgü bir metot ekler. UML'de kalıtım, ucu üst sınıfa dönük **içi boş üçgen** okla gösterilir.

HAYVAN ÖRNEĞİ

Hayvan ve Kopek Sınıfları

```
public class Hayvan {
    private String isim;     // ortak nitelikler
    private int yas;
    private double saglik;

    public Hayvan(String isim, int yas) {
        this.isim = isim;
        this.yas = yas;
        this.saglik = 100.0;
    }
    public String getIsim() { return isim; }   // private isim'e erişim yolu
    public void konus() {
        System.out.println("Bu hayvan konuşur.");
    }
    public void uyu() {
        saglik += 5;
        System.out.println(isim + " uyudu. Sağlık: " + saglik);
    }
}
```

Üst sınıf `Hayvan` : bütün hayvanlarda ortak olan alanlar ve davranışlar.

HAYVAN ÖRNEĞİ

Hayvan ve Kopek Sınıfları

```
public class Kopek extends Hayvan {
    public Kopek(String isim, int yas) {
        super(isim, yas);   // üst sınıf yapıcısı alanları doldurur
    }
    @Override
    public void konus() {   // miras alınan davranışı değiştirir
        System.out.println(getIsim() + " havlar.");
    }
    public void topOyna() { // yalnızca Kopek'te olan davranış
        System.out.println(getIsim() + " top oynuyor!");
    }
}
```

```
Kopek kopek = new Kopek("Karabaş", 4);
kopek.konus();     // Kopek'in sürümü
kopek.topOyna();   // Kopek'e özgü
kopek.uyu();       // Hayvan'dan miras
```

```
Karabaş havlar.
Karabaş top oynuyor!
Karabaş uyudu. Sağlık: 105.0
```

Alt sınıf `Kopek` yalnızca **farkı** yazar; `uyu()` hiç yazılmadığı hâlde çalışır.

HAYVAN ÖRNEĞİ

Kalıtımın Avantajları

Yeniden kullanım

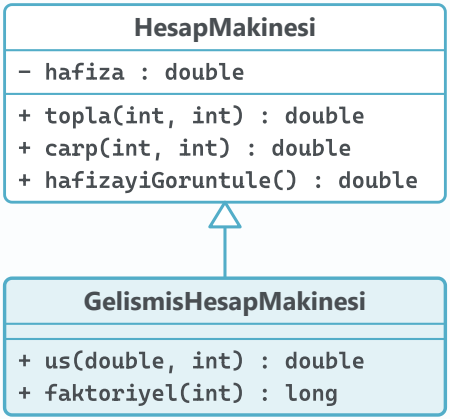
Üst sınıfın metot ve alanları alt sınıfta yeniden yazılmadan kullanılır.

Genişletme

Mevcut sınıf, kodu değiştirilmeden yeni özelliklerle genişletilir.

Değiştirme

Üst sınıfın metotları alt sınıfta **geçersiz kılınarak** (overriding) değiştirilebilir.




```java
public class GelismisHesapMakinesi
        extends HesapMakinesi {
    // Yeni yetenek: üs alma
    public double us(double taban, int us) {
        double sonuc = 1.0;
        for (int i = 0; i < us; i++) {
            sonuc *= taban;
        }
        return sonuc;
    }
    // faktoriyel(int) benzer biçimde eklenir
}
// main içinde
GelismisHesapMakinesi h = new GelismisHesapMakinesi();
System.out.println(h.topla(5, 3));         // 8.0  (miras)
System.out.println(h.hafizayiGoruntule()); // 8.0  (miras)
System.out.println(h.us(2, 5));            // 32.0 (yeni)
```

super Anahtar Sözcüğü

- `super(...)` : alt sınıfın yapıcısı içinde **üst sınıfın yapıcısını** çağırır; üst sınıfın alanlarına ilk değerleri o atar.
- `super(...)` yapıcının **ilk satırı** olmalıdır (Java 25'ten önce; Java 25 önüne nesneye dokunmayan deyimlere izin verir).
- Yazılmazsa derleyici gizlice `super();` (parametresiz) ekler.
- `super.metot()` : üst sınıfın metoduna nokta (.) ile erişir; geçersiz kılınmış bir metodun içinden üst sınıfın sürümünü çağırmak için kullanılır.
- İlke: **her sınıf yalnızca kendi verilerini yönetmelidir**. Kedi, Hayvan'ın `isim` alanına kendisi atama yapmaz; işi `super(isim)` 'e bırakır.

```
class Kedi extends Hayvan {
    private boolean evcil;

    Kedi(String isim, boolean evcil) {
        super(isim);          // Hayvan(String)
        this.evcil = evcil;  // kendi verisi
    }

    @Override
    public String toString() {
        return super.toString()   // Hayvan'ınki
               + ", Evcil: " + evcil;
    }
}
// main içinde
System.out.println(new Kedi("Mavi", true));
```

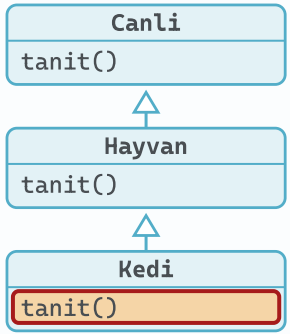
ÇIKTI

```
İsim: Mavi, Sağlık: 100.0, Evcil: true
```

Hayvan'ın `toString()` metodu isim ve sağlık bilgisini döndürür; Kedi yalnızca kendi alanını ekler.

`super.tanit()` Zinciri: Kod İzleme

```
1  class Canli {
2      void tanit() { System.out.println("Canlıyım"); }
3  }
4  class Hayvan extends Canli {
5      void tanit() {
6          super.tanit();
7          System.out.println("Hayvanım");
8      }
9  }
10 class Kedi extends Hayvan {
11     void tanit() {
12         super.tanit();
13         System.out.println("Kediyim");
14     }
15 }
▶  new Kedi().tanit();   // main
```



Çağrı yığını

`Kedi.tanit()`

`main()`

nesne: Kedi #1

`arama gerçek sınıftan başlar: Kedi.tanit()`

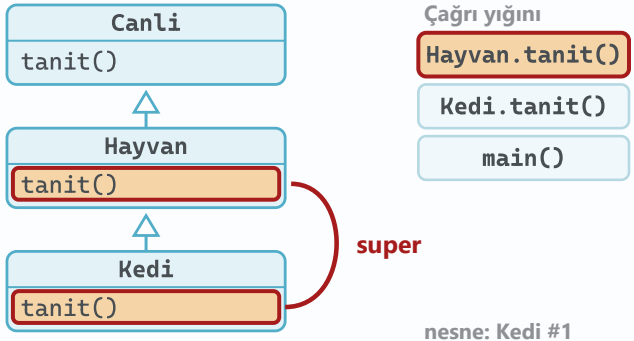
Nesne bir `Kedi` ; arama en alttan başlar ve `Kedi.tanit()` bulunur.

Adım 1 / 7

ÇIKTI

`super.tanit()` Zinciri: Kod İzleme

```
1  class Canli {
2      void tanit() { System.out.println("Canlıyım"); }
3  }
4  class Hayvan extends Canli {
5      void tanit() {
6          super.tanit();
7          System.out.println("Hayvanım");
8      }
9  }
10 class Kedi extends Hayvan {
11     void tanit() {
▶          super.tanit();
13         System.out.println("Kediyim");
14     }
15 }
16 new Kedi().tanit();   // main
```



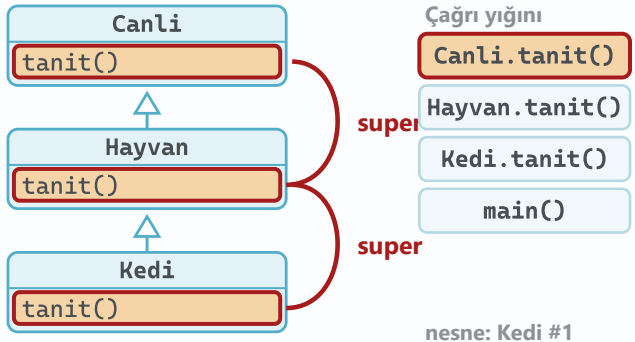
`super.tanit()` : bir **üst** sınıfın sürümü çağrılır.
`Kedi.tanit` bekler, yığında kalır.

Adım 2 / 7

ÇIKTI

`super.tanit()` Zinciri: Kod İzleme

```
1  class Canli {
2      void tanit() { System.out.println("Canlıyım"); }
3  }
4  class Hayvan extends Canli {
▶      void tanit() {
▶          super.tanit();
7          System.out.println("Hayvanım");
8      }
9  }
10 class Kedi extends Hayvan {
11     void tanit() {
12         super.tanit();
13         System.out.println("Kediyim");
14     }
15 }
16 new Kedi().tanit();   // main
```



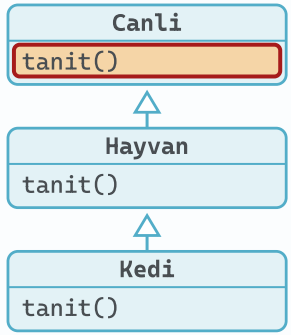
Nesne hâlâ bir `Kedi` olsa da `Hayvan` içindeki `super` sabit olarak `Canli`'yi gösterir: dinamik bağlama **yok**.

Adım 3 / 7

ÇIKTI

`super.tanit()` Zinciri: Kod İzleme

```
1  class Canli {
2      void tanit() { System.out.println("Canlıyım"); }
3  }
4  class Hayvan extends Canli {
5      void tanit() {
6          super.tanit();
7          System.out.println("Hayvanım");
8      }
9  }
10 class Kedi extends Hayvan {
11     void tanit() {
12         super.tanit();
13         System.out.println("Kediyim");
14     }
15 }
16 new Kedi().tanit();   // main
```



Çağrı yığını

- `Canli.tanit()`
- `Hayvan.tanit()`
- `Kedi.tanit()`
- `main()`

nesne: Kedi #1

Canlıyım yazılır, `Canli.tanit()` döner

En üstteki sürüm çalışır ve ilk satırı yazar; sonra döner.

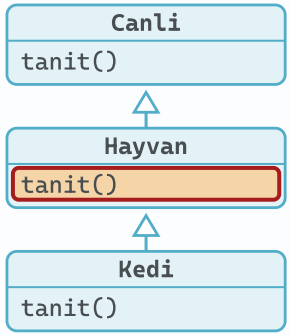
Adım 4 / 7

ÇIKTI

```
Canlıyım
```

`super.tanit()` Zinciri: Kod İzleme

```
1  class Canli {
2      void tanit() { System.out.println("Canlıyım"); }
3  }
4  class Hayvan extends Canli {
5      void tanit() {
6          super.tanit();
▶          System.out.println("Hayvanım");
8      }
9  }
10 class Kedi extends Hayvan {
11     void tanit() {
12         super.tanit();
13         System.out.println("Kediyim");
14     }
15 }
16 new Kedi().tanit();   // main
```



Çağrı yığını

`Hayvan.tanit()`

`Kedi.tanit()`

`main()`

nesne: Kedi #1

`Hayvan.tanit()` kaldığı yerden sürer: **Hayvanım**.

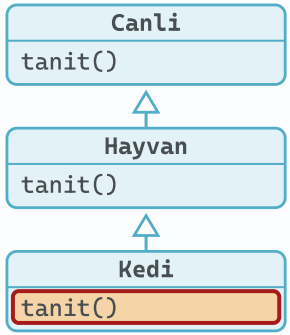
Adım 5 / 7

ÇIKTI

```
Canlıyım
Hayvanım
```

`super.tanit()` Zinciri: Kod İzleme

```
1  class Canli {
2      void tanit() { System.out.println("Canlıyım"); }
3  }
4  class Hayvan extends Canli {
5      void tanit() {
6          super.tanit();
7          System.out.println("Hayvanım");
8      }
9  }
10 class Kedi extends Hayvan {
11     void tanit() {
12         super.tanit();
▶          System.out.println("Kediyim");
14     }
15 }
16 new Kedi().tanit();   // main
```



Çağrı yığını

`Kedi.tanit()`

`main()`

nesne: Kedi #1

Son olarak `Kedi.tanit()` kendi satırını yazar.

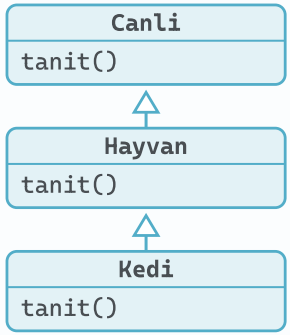
Adım 6 / 7

ÇIKTI

```
Canlıyım
Hayvanım
Kediyim
```

`super.tanit()` Zinciri: Kod İzleme

```
1  class Canli {
2      void tanit() { System.out.println("Canlıyım"); }
3  }
4  class Hayvan extends Canli {
5      void tanit() {
6          super.tanit();
7          System.out.println("Hayvanım");
8      }
9  }
10 class Kedi extends Hayvan {
11     void tanit() {
12         super.tanit();
13         System.out.println("Kediyim");
14     }
15 }
▶  new Kedi().tanit();   // main
```



Çağrı yığını

main()

nesne: Kedi #1

çağrı aşağıdan yukarı, çıktı yukarıdan aşağı

Çağrılar Kedi → Hayvan → Canli yönünde gitti, çıktılar ters sırada geldi: her sınıf önce üst sınıfın işini yaptırıp sonra kendi farkını ekledi.

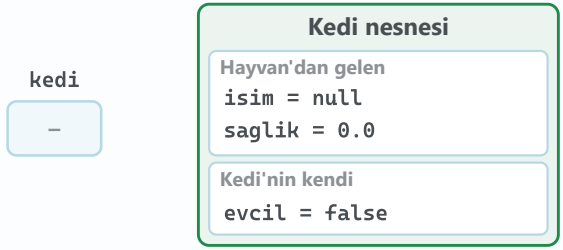
Adım 7 / 7

ÇIKTI

```
Canlıyım
Hayvanım
Kediyim
```

Yapıcı Zinciri: `new Kedi("Mavi", true)`

```
1  class Hayvan {
2      private String isim;
3      private double saglik;
4      Hayvan(String isim) {
5          super();   // Object() çağrılır
6          this.isim = isim;
7          saglik = 100.0;
8          System.out.println("Hayvan yapıcısı");
9      }
10 }
11 class Kedi extends Hayvan {
12     private boolean evcil;
13     Kedi(String isim, boolean evcil) {
14         super(isim);
15         this.evcil = evcil;
16         System.out.println("Kedi yapıcısı");
17     }
18 }
19 // main içinde
▶  Kedi kedi = new Kedi("Mavi", true);
21 System.out.println("Nesne hazır");
```



`new` : önce bellekte nesneye yer ayrılır; **tüm** alanlar (Hayvan'ınkiler dahil) varsayılan değerleri alır.

Adım 1 / 11

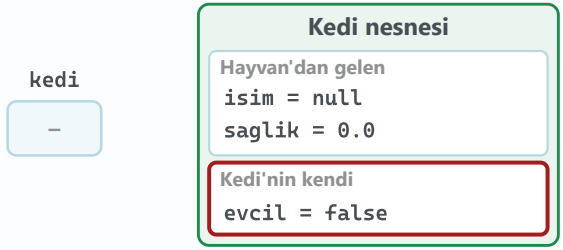
ÇAĞRI YIĞINI

`main()`

ÇIKTI

Yapıcı Zinciri: `new Kedi("Mavi", true)`

```
1  class Hayvan {
2      private String isim;
3      private double saglik;
4      Hayvan(String isim) {
5          super();   // Object() çağrılır
6          this.isim = isim;
7          saglik = 100.0;
8          System.out.println("Hayvan yapıcısı");
9      }
10 }
11 class Kedi extends Hayvan {
12     private boolean evcil;
▶       Kedi(String isim, boolean evcil) {
14         super(isim);
15         this.evcil = evcil;
16         System.out.println("Kedi yapıcısı");
17     }
18 }
19 // main içinde
20 Kedi kedi = new Kedi("Mavi", true);
21 System.out.println("Nesne hazır");
```



Kedi yapıcısı çağrılır. Kendi işine başlamadan önce üst sınıf kısmı hazırlanmalıdır.

Adım 2 / 11

ÇAĞRI YIĞINI

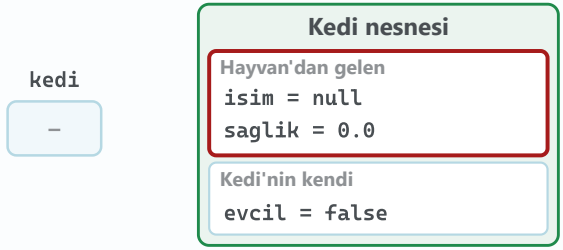
`Kedi("Mavi", true)`

`main()`

ÇIKTI

Yapıcı Zinciri: `new Kedi("Mavi", true)`

```
1  class Hayvan {
2      private String isim;
3      private double saglik;
4      Hayvan(String isim) {
5          super();   // Object() çağrılır
6          this.isim = isim;
7          saglik = 100.0;
8          System.out.println("Hayvan yapıcısı");
9      }
10 }
11 class Kedi extends Hayvan {
12     private boolean evcil;
13     Kedi(String isim, boolean evcil) {
▶            super(isim);
15         this.evcil = evcil;
16         System.out.println("Kedi yapıcısı");
17     }
18 }
19 // main içinde
20 Kedi kedi = new Kedi("Mavi", true);
21 System.out.println("Nesne hazır");
```



`super(isim)` : Hayvan yapıcısı çağrılır; Kedi yapıcısı bekler.

Adım 3 / 11

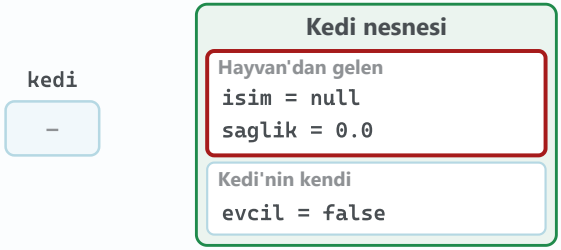
ÇAĞRI YIĞINI

- `Kedi("Mavi", true)`
- `main()`

ÇIKTI

Yapıcı Zinciri: `new Kedi("Mavi", true)`

```
1  class Hayvan {
2      private String isim;
3      private double saglik;
4      Hayvan(String isim) {
▶          super();   // Object() çağrılır
6          this.isim = isim;
7          saglik = 100.0;
8          System.out.println("Hayvan yapıcısı");
9      }
10 }
11 class Kedi extends Hayvan {
12     private boolean evcil;
13     Kedi(String isim, boolean evcil) {
14         super(isim);
15         this.evcil = evcil;
16         System.out.println("Kedi yapıcısı");
17     }
18 }
19 // main içinde
20 Kedi kedi = new Kedi("Mavi", true);
21 System.out.println("Nesne hazır");
```



Hayvan da önce kendi üst sınıfını çağırır: `Object()`. Her zincir Object'te biter.

Adım 4 / 11

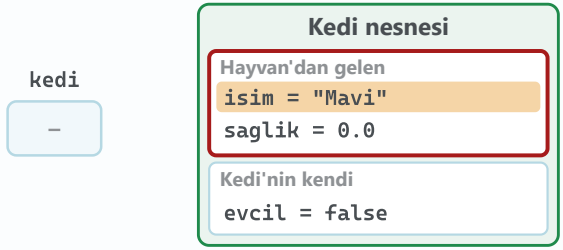
ÇAĞRI YIĞINI

- `Object()`
- `Hayvan("Mavi")`
- `Kedi("Mavi", true)`
- `main()`

ÇIKTI

Yapıcı Zinciri: `new Kedi("Mavi", true)`

```
1  class Hayvan {
2      private String isim;
3      private double saglik;
4      Hayvan(String isim) {
5          super();   // Object() çağrılır
6          this.isim = isim;
7          saglik = 100.0;
8          System.out.println("Hayvan yapıcısı");
9      }
10 }
11 class Kedi extends Hayvan {
12     private boolean evcil;
13     Kedi(String isim, boolean evcil) {
14         super(isim);
15         this.evcil = evcil;
16         System.out.println("Kedi yapıcısı");
17     }
18 }
19 // main içinde
20 Kedi kedi = new Kedi("Mavi", true);
21 System.out.println("Nesne hazır");
```



Object() döndü. Hayvan kendi alanını doldurur: `isim = "Mavi"`.

Adım 5 / 11

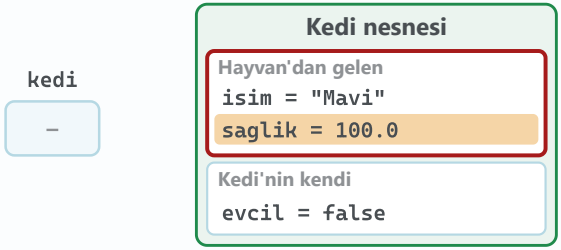
ÇAĞRI YIĞINI

- `Hayvan("Mavi")`
- `Kedi("Mavi", true)`
- `main()`

ÇIKTI

Yapıcı Zinciri: `new Kedi("Mavi", true)`

```
1  class Hayvan {
2      private String isim;
3      private double saglik;
4      Hayvan(String isim) {
5          super();   // Object() çağrılır
6          this.isim = isim;
7          saglik = 100.0;
8          System.out.println("Hayvan yapıcısı");
9      }
10 }
11 class Kedi extends Hayvan {
12     private boolean evcil;
13     Kedi(String isim, boolean evcil) {
14         super(isim);
15         this.evcil = evcil;
16         System.out.println("Kedi yapıcısı");
17     }
18 }
19 // main içinde
20 Kedi kedi = new Kedi("Mavi", true);
21 System.out.println("Nesne hazır");
```



`saglik = 100.0` .

Adım 6 / 11

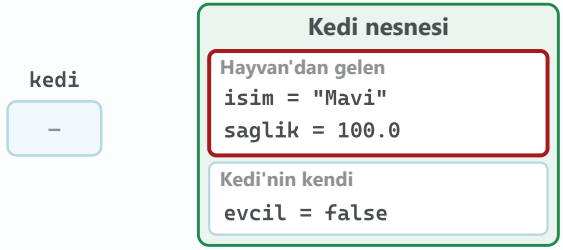
ÇAĞRI YIĞINI

- `Hayvan("Mavi")`
- `Kedi("Mavi", true)`
- `main()`

ÇIKTI

Yapıcı Zinciri: `new Kedi("Mavi", true)`

```
1  class Hayvan {
2      private String isim;
3      private double saglik;
4      Hayvan(String isim) {
5          super();   // Object() çağrılır
6          this.isim = isim;
7          saglik = 100.0;
▶          System.out.println("Hayvan yapıcısı");
9      }
10 }
11 class Kedi extends Hayvan {
12     private boolean evcil;
13     Kedi(String isim, boolean evcil) {
14         super(isim);
15         this.evcil = evcil;
16         System.out.println("Kedi yapıcısı");
17     }
18 }
19 // main içinde
20 Kedi kedi = new Kedi("Mavi", true);
21 System.out.println("Nesne hazır");
```



Hayvan yapıcısı mesajını yazar ve döner: **önce üst sınıf** yazdı.

Adım 7 / 11

ÇAĞRI YIĞINI

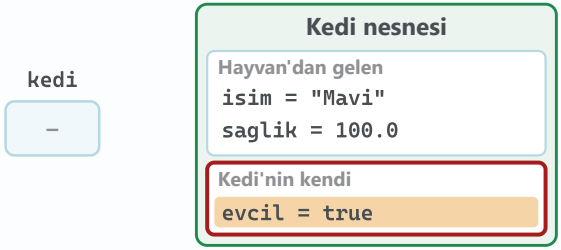
- `Hayvan("Mavi")`
- `Kedi("Mavi", true)`
- `main()`

ÇIKTI

```
Hayvan yapıcısı
```

Yapıcı Zinciri: `new Kedi("Mavi", true)`

```
1  class Hayvan {
2      private String isim;
3      private double saglik;
4      Hayvan(String isim) {
5          super();   // Object() çağrılır
6          this.isim = isim;
7          saglik = 100.0;
8          System.out.println("Hayvan yapıcısı");
9      }
10 }
11 class Kedi extends Hayvan {
12     private boolean evcil;
13     Kedi(String isim, boolean evcil) {
14         super(isim);
15         this.evcil = evcil;
16         System.out.println("Kedi yapıcısı");
17     }
18 }
19 // main içinde
20 Kedi kedi = new Kedi("Mavi", true);
21 System.out.println("Nesne hazır");
```



Kedi yapıcısı kaldığı yerden devam eder: `evcil = true`.

Adım 8 / 11

ÇAĞRI YIĞINI

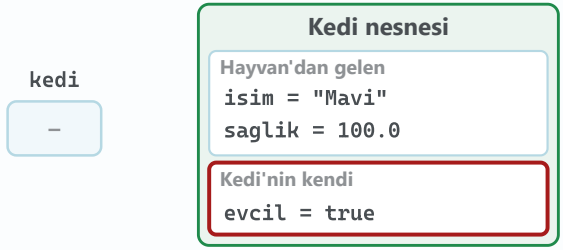
- `Kedi("Mavi", true)`
- `main()`

ÇIKTI

```
Hayvan yapıcısı
```

Yapıcı Zinciri: `new Kedi("Mavi", true)`

```
1  class Hayvan {
2      private String isim;
3      private double saglik;
4      Hayvan(String isim) {
5          super();   // Object() çağrılır
6          this.isim = isim;
7          saglik = 100.0;
8          System.out.println("Hayvan yapıcısı");
9      }
10 }
11 class Kedi extends Hayvan {
12     private boolean evcil;
13     Kedi(String isim, boolean evcil) {
14         super(isim);
15         this.evcil = evcil;
▶            System.out.println("Kedi yapıcısı");
17     }
18 }
19 // main içinde
20 Kedi kedi = new Kedi("Mavi", true);
21 System.out.println("Nesne hazır");
```



Kedi yapıcısı mesajını yazar.

Adım 9 / 11

ÇAĞRI YIĞINI

`Kedi("Mavi", true)`

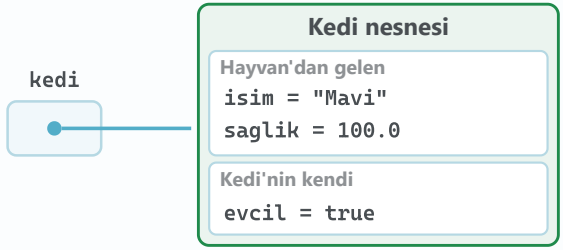
`main()`

ÇIKTI

```
Hayvan yapıcısı
Kedi yapıcısı
```

Yapıcı Zinciri: `new Kedi("Mavi", true)`

```
1  class Hayvan {
2      private String isim;
3      private double saglik;
4      Hayvan(String isim) {
5          super();   // Object() çağrılır
6          this.isim = isim;
7          saglik = 100.0;
8          System.out.println("Hayvan yapıcısı");
9      }
10 }
11 class Kedi extends Hayvan {
12     private boolean evcil;
13     Kedi(String isim, boolean evcil) {
14         super(isim);
15         this.evcil = evcil;
16         System.out.println("Kedi yapıcısı");
17     }
18 }
19 // main içinde
▶ Kedi kedi = new Kedi("Mavi", true);
21 System.out.println("Nesne hazır");
```



Yapıcı bitti; nesnenin adresi `kedi` değişkenine atanır.

Adım 10 / 11

ÇAĞRI YIĞINI

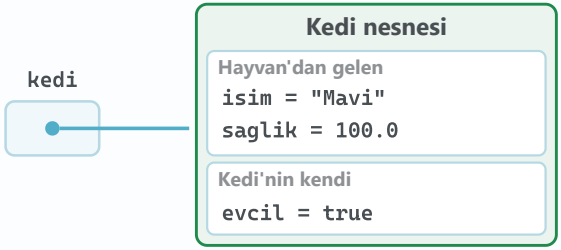
`main()`

ÇIKTI

```
Hayvan yapıcısı
Kedi yapıcısı
```

Yapıcı Zinciri: `new Kedi("Mavi", true)`

```
1  class Hayvan {
2      private String isim;
3      private double saglik;
4      Hayvan(String isim) {
5          super();   // Object() çağrılır
6          this.isim = isim;
7          saglik = 100.0;
8          System.out.println("Hayvan yapıcısı");
9      }
10 }
11 class Kedi extends Hayvan {
12     private boolean evcil;
13     Kedi(String isim, boolean evcil) {
14         super(isim);
15         this.evcil = evcil;
16         System.out.println("Kedi yapıcısı");
17     }
18 }
19 // main içinde
20 Kedi kedi = new Kedi("Mavi", true);
▶  System.out.println("Nesne hazır");
```



Yapıcılar **yukarıdan aşağıya** tamamlanır: Object → Hayvan → Kedi.

Adım 11 / 11

ÇAĞRI YIĞINI

`main()`

ÇIKTI

```
Hayvan yapıcısı
Kedi yapıcısı
Nesne hazır
```

Gizli super() ve Hatalar

1) ÜST SINIFTA PARAMETRESİZ YAPICI YOK

```
class Hayvan {
    private String isim;
    Hayvan(String isim) { this.isim = isim; }
}
class Kedi extends Hayvan {
    Kedi(String isim) {
        // derleyici buraya super(); ekler
        System.out.println("Kedi");
    }
}
```

```
error: constructor Hayvan in class Hayvan
  cannot be applied to given types;
  required: String
  found:    no arguments
```

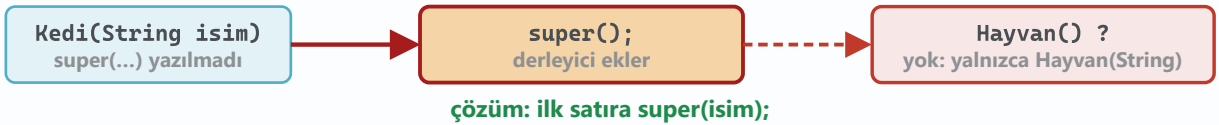
2) SUPER(...) İLK SATIRDA DEĞİL

```
Kedi(String isim) {
    System.out.println("Kedi");
    super(isim);
}
```

```
error: call to super must be first
  statement in constructor
```

Çözüm: alt sınıf yapıcısının ilk satırında uygun üst sınıf yapıcısı açıkça çağrılır: `super(isim);`

Hiç yapıcı yazılmamış sınıfa derleyici parametresiz bir varsayılan yapıcı (default constructor) ekler; bir yapıcı yazıldığı anda bu eklenmez.



`this(...)` ve `super(...)`: Kod İzleme

```
1  class Calisan {
2      private String ad;  private double maas;
3      Calisan(String ad) {
4          this(ad, 20000.0);
5          System.out.println("Calisan(String)");
6      }
7      Calisan(String ad, double maas) {
8          this.ad = ad;  this.maas = maas;
9          System.out.println("Calisan(String, double)");
10     }
11     double getMaas() { return maas; }
12 }
13 class Yonetici extends Calisan {
14     private double prim;
15     Yonetici(String ad) {
16         this(ad, 5000.0);
17         System.out.println("Yonetici(String)");
18     }
19     Yonetici(String ad, double prim) {
20         super(ad);
21         this.prim = prim;
22         System.out.println("Yonetici(String, double)");
23     }
24     double gelir() { return getMaas() + prim; }
25 }
▶ Yonetici y = new Yonetici("Ayşe");   // main
27 System.out.println(y.gelir());
```

Yığın (stack)

`Yonetici(ad)`

`main()`

y	?

Heap

Yonetici #1

ad	null
maas	0.0
prim	0.0

Calisan kısmı: ad, maas

`new`: nesne ayrıldı, alanlar varsayılan

`new` önce nesneye yer ayırır; üç alan da varsayılan değerdedir. Ardından `Yonetici(String)` çağrılır.

Adım 1 / 9

ÇIKTI

DEĞİŞKENLER

`y = ?`

`this(...)` ve `super(...)`: Kod İzleme

```
1  class Calisan {
2      private String ad;  private double maas;
3      Calisan(String ad) {
4          this(ad, 20000.0);
5          System.out.println("Calisan(String)");
6      }
7      Calisan(String ad, double maas) {
8          this.ad = ad;  this.maas = maas;
9          System.out.println("Calisan(String, double)");
10     }
11     double getMaas() { return maas; }
12 }
13 class Yonetici extends Calisan {
14     private double prim;
15     Yonetici(String ad) {
▶          this(ad, 5000.0);
17         System.out.println("Yonetici(String)");
18     }
19     Yonetici(String ad, double prim) {
20         super(ad);
21         this.prim = prim;
22         System.out.println("Yonetici(String, double)");
23     }
24     double gelir() { return getMaas() + prim; }
25 }
26 Yonetici y = new Yonetici("Ayşe");   // main
27 System.out.println(y.gelir());
```

Yığın (stack)

`Yonetici(ad,prim)`

`Yonetici(ad)`

`main()`

y = ?

Heap

Yonetici #1

ad	null
maas	0.0
prim	0.0

Calisan kısmı: ad, maas

`this( ... )` → aynı sınıfın öteki yapıcısı

İlk satır `this(ad, 5000.0)` : aynı sınıfın iki parametreli yapıcısına geçilir. `Yonetici(String)` gövdesini henüz çalıştırmadan bekler.

Adım 2 / 9

ÇIKTI

DEĞİŞKENLER

y = ?

`this(...)` ve `super(...)`: Kod İzleme

```
1  class Calisan {
2      private String ad;  private double maas;
3      Calisan(String ad) {
4          this(ad, 20000.0);
5          System.out.println("Calisan(String)");
6      }
7      Calisan(String ad, double maas) {
8          this.ad = ad;  this.maas = maas;
9          System.out.println("Calisan(String, double)");
10     }
11     double getMaas() { return maas; }
12 }
13 class Yonetici extends Calisan {
14     private double prim;
15     Yonetici(String ad) {
16         this(ad, 5000.0);
17         System.out.println("Yonetici(String)");
18     }
19     Yonetici(String ad, double prim) {
▶        super(ad);
21         this.prim = prim;
22         System.out.println("Yonetici(String, double)");
23     }
24     double gelir() { return getMaas() + prim; }
25 }
26 Yonetici y = new Yonetici("Ayşe");   // main
27 System.out.println(y.gelir());
```

Yığın (stack)

- `Calisan(ad)`
- `Yonetici(ad,prim)`
- `Yonetici(ad)`
- `main()` — y = ?

Heap

Yonetici #1

ad	null
maas	0.0
prim	0.0

Calisan kısmı: ad, maas

`super(ad)` → üst sınıfın yapıcısı

`super(ad)` : üst sınıf `Calisan` 'ın tek parametreli yapıcısı. Yönetici kısmının kurulması, Calisan kısmı bitene kadar bekler.

Adım 3 / 9

ÇIKTI

DEĞİŞKENLER

`y = ?`

`this(...)` ve `super(...)`: Kod İzleme

```
1  class Calisan {
2      private String ad;  private double maas;
3      Calisan(String ad) {
▶          this(ad, 20000.0);
5          System.out.println("Calisan(String)");
6      }
7      Calisan(String ad, double maas) {
8          this.ad = ad;  this.maas = maas;
9          System.out.println("Calisan(String, double)");
10     }
11     double getMaas() { return maas; }
12 }
13 class Yonetici extends Calisan {
14     private double prim;
15     Yonetici(String ad) {
16         this(ad, 5000.0);
17         System.out.println("Yonetici(String)");
18     }
19     Yonetici(String ad, double prim) {
20         super(ad);
21         this.prim = prim;
22         System.out.println("Yonetici(String, double)");
23     }
24     double gelir() { return getMaas() + prim; }
25 }
26 Yonetici y = new Yonetici("Ayşe");   // main
27 System.out.println(y.gelir());
```

Yığın (stack)

- `Calisan(ad,maas)`
- `Calisan(ad)`
- `Yonetici(ad,prim)`
- `Yonetici(ad)`
- `main()` — y = ?

Heap

`Yonetici #1`

ad	null
maas	0.0
prim	0.0

Calisan kısmı: ad, maas

`this( … ) yine; zincirin sonu gizli super()`

`Calisan` da `this(...)` ile devreder. Bu yapıcının ilk satırı `this` ya da `super` olmadığı için derleyici gizli `super()` ekler: `Object()` çalışıp döner.

Adım 4 / 9

ÇIKTI

DEĞİŞKENLER

y = ?

`this(...)` ve `super(...)`: Kod İzleme

```
1  class Calisan {
2      private String ad;  private double maas;
3      Calisan(String ad) {
4          this(ad, 20000.0);
5          System.out.println("Calisan(String)");
6      }
7      Calisan(String ad, double maas) {
▶          this.ad = ad;  this.maas = maas;
▶          System.out.println("Calisan(String, double)");
10     }
11     double getMaas() { return maas; }
12 }
13 class Yonetici extends Calisan {
14     private double prim;
15     Yonetici(String ad) {
16         this(ad, 5000.0);
17         System.out.println("Yonetici(String)");
18     }
19     Yonetici(String ad, double prim) {
20         super(ad);
21         this.prim = prim;
22         System.out.println("Yonetici(String, double)");
23     }
24     double gelir() { return getMaas() + prim; }
25 }
26 Yonetici y = new Yonetici("Ayşe");   // main
27 System.out.println(y.gelir());
```

Yığın (stack)

- `Calisan(ad,maas)`
- `Calisan(ad)`
- `Yonetici(ad,prim)`
- `Yonetici(ad)`
- `main()` — y = ?

Heap

`Yonetici #1`

ad	"Ayşe"
maas	20000.0
prim	0.0

Calisan kısmı: ad, maas

`Calisan kısmı kuruldu`

İlk gövde burada çalışır: `ad` ve `maas` dolar, ilk satır yazılır. Çağrılar aşağı doğru gitmişti, gövdeler en üstteki sınıftan başlayarak tamamlanır.

Adım 5 / 9

ÇIKTI

```
Calisan(String, double)
```

DEĞİŞKENLER

y = ?

`this(...)` ve `super(...)`: Kod İzleme

```
1  class Calisan {
2      private String ad;  private double maas;
3      Calisan(String ad) {
4          this(ad, 20000.0);
▶          System.out.println("Calisan(String)");
6      }
7      Calisan(String ad, double maas) {
8          this.ad = ad;  this.maas = maas;
9          System.out.println("Calisan(String, double)");
10     }
11     double getMaas() { return maas; }
12 }
13 class Yonetici extends Calisan {
14     private double prim;
15     Yonetici(String ad) {
16         this(ad, 5000.0);
17         System.out.println("Yonetici(String)");
18     }
19     Yonetici(String ad, double prim) {
20         super(ad);
21         this.prim = prim;
22         System.out.println("Yonetici(String, double)");
23     }
24     double gelir() { return getMaas() + prim; }
25 }
26 Yonetici y = new Yonetici("Ayşe");   // main
27 System.out.println(y.gelir());
```

Yığın (stack)

- `Calisan(ad)`
- `Yonetici(ad,prim)`
- `Yonetici(ad)`
- `main()` — y = ?

Heap

`Yonetici #1`

ad	"Ayşe"
maas	20000.0
prim	0.0

Calisan kısmı: ad, maas

`Calisan(String)` kaldığı yerden sürer

İki parametreli yapıcı döndü; `Calisan(String)` kendi satırını yazar.

Adım 6 / 9

ÇIKTI

```
Calisan(String, double)
Calisan(String)
```

DEĞİŞKENLER

`y = ?`

`this(...)` ve `super(...)`: Kod İzleme

```
1  class Calisan {
2      private String ad;  private double maas;
3      Calisan(String ad) {
4          this(ad, 20000.0);
5          System.out.println("Calisan(String)");
6      }
7      Calisan(String ad, double maas) {
8          this.ad = ad;  this.maas = maas;
9          System.out.println("Calisan(String, double)");
10     }
11     double getMaas() { return maas; }
12 }
13 class Yonetici extends Calisan {
14     private double prim;
15     Yonetici(String ad) {
16         this(ad, 5000.0);
17         System.out.println("Yonetici(String)");
18     }
19     Yonetici(String ad, double prim) {
20         super(ad);
▶        this.prim = prim;
▶        System.out.println("Yonetici(String, double)");
23     }
24     double gelir() { return getMaas() + prim; }
25 }
26 Yonetici y = new Yonetici("Ayşe");   // main
27 System.out.println(y.gelir());
```

Yığın (stack)

- `Yonetici(ad,prim)`
- `Yonetici(ad)`
- `main()`
  - y: ?

Heap

- `Yonetici #1`
  - ad: "Ayşe"
  - maas: 20000.0
  - prim: 5000.0

Calisan kısmı: ad, maas

`Yonetici kısmı kuruluyor`

`super(ad)` tamamlandı. Ancak şimdi alt sınıfın kendi alanı dolar: `prim = 5000.0` .

Adım 7 / 9

ÇIKTI

```
Calisan(String, double)
Calisan(String)
Yonetici(String, double)
```

DEĞİŞKENLER

`y = ?`

`this(...)` ve `super(...)`: Kod İzleme

```
1  class Calisan {
2      private String ad;  private double maas;
3      Calisan(String ad) {
4          this(ad, 20000.0);
5          System.out.println("Calisan(String)");
6      }
7      Calisan(String ad, double maas) {
8          this.ad = ad;  this.maas = maas;
9          System.out.println("Calisan(String, double)");
10     }
11     double getMaas() { return maas; }
12 }
13 class Yonetici extends Calisan {
14     private double prim;
15     Yonetici(String ad) {
16         this(ad, 5000.0);
▶           System.out.println("Yonetici(String)");
18     }
19     Yonetici(String ad, double prim) {
20         super(ad);
21         this.prim = prim;
22         System.out.println("Yonetici(String, double)");
23     }
24     double gelir() { return getMaas() + prim; }
25 }
26 Yonetici y = new Yonetici("Ayşe");   // main
27 System.out.println(y.gelir());
```

Yığın (stack)

Yonetici(ad)

main()

y	?

Heap

Yonetici #1

ad	"Ayşe"
maas	20000.0
prim	5000.0

Calisan kısmı: ad, maas

ilk çağrılan yapıcı en son biter

En son, ilk çağrılan `Yonetici(String)` biter. Dört yapıcı çalıştı ama tek bir nesne kuruldu.

Adım 8 / 9

ÇIKTI

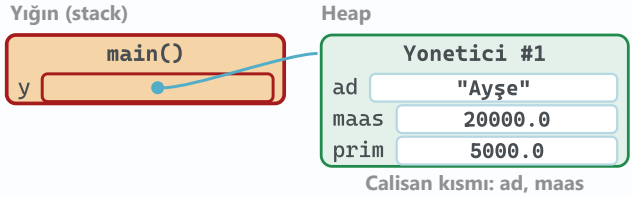
```
Calisan(String, double)
Calisan(String)
Yonetici(String, double)
Yonetici(String)
```

DEĞİŞKENLER

y = ?

`this(...)` ve `super(...)`: Kod İzleme

```
1  class Calisan {
2      private String ad;  private double maas;
3      Calisan(String ad) {
4          this(ad, 20000.0);
5          System.out.println("Calisan(String)");
6      }
7      Calisan(String ad, double maas) {
8          this.ad = ad;  this.maas = maas;
9          System.out.println("Calisan(String, double)");
10     }
11     double getMaas() { return maas; }
12 }
13 class Yonetici extends Calisan {
14     private double prim;
15     Yonetici(String ad) {
16         this(ad, 5000.0);
17         System.out.println("Yonetici(String)");
18     }
19     Yonetici(String ad, double prim) {
20         super(ad);
21         this.prim = prim;
22         System.out.println("Yonetici(String, double)");
23     }
24     double gelir() { return getMaas() + prim; }
25 }
26 Yonetici y = new Yonetici("Ayşe");   // main
▶  System.out.println(y.gelir());
```



`gelir() = 20000.0 + 5000.0`

Nesnenin adresi `y`'ye atanır. `gelir()`, kalıtımla gelen `getMaas()` ile kendi `prim` alanını toplar: `25000.0`.

Adım 9 / 9

ÇIKTI

```
Calisan(String, double)
Calisan(String)
Yonetici(String, double)
Yonetici(String)
25000.0
```

DEĞİŞKENLER

`y = →Yonetici #1`

Kalıtımda İlklendirme Sırası: Kod İzleme

```
1  class Ust {
▶       static { System.out.println("Ust static"); }
3       int a = deger("Ust alanı");
4       Ust() { System.out.print("Ust(), "); }
5       static int deger(String s) {
6           System.out.print(s + ", ");
7           return 1;
8       }
9   }
10  class Alt extends Ust {
11      static { System.out.println("Alt static"); }
12      int b = deger("Alt alanı");
13      Alt() { System.out.println("Alt()"); }
14  }
15  // main içinde:
16  new Alt();
17  new Alt();
```



İlk `new Alt()` için JVM `Alt` sınıfını başlatmalıdır. Önce üst sınıf başlatılır: `Ust` 'un `static` bloğu çalışır.

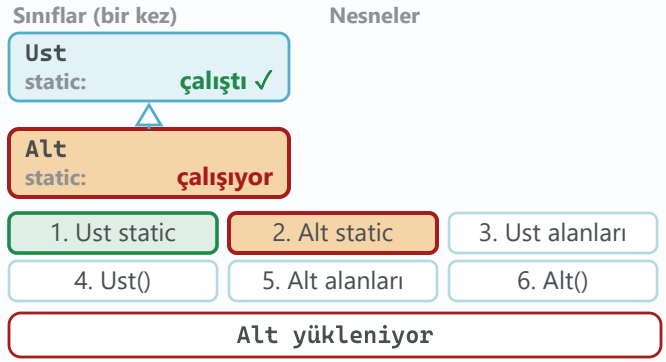
Adım 1 / 7

ÇIKTI

```
Ust static
```

Kalıtımda İlklendirme Sırası: Kod İzleme

```
1  class Ust {
2      static { System.out.println("Ust static"); }
3      int a = deger("Ust alanı");
4      Ust() { System.out.print("Ust(), "); }
5      static int deger(String s) {
6          System.out.print(s + ", ");
7          return 1;
8      }
9  }
10 class Alt extends Ust {
▶      static { System.out.println("Alt static"); }
12     int b = deger("Alt alanı");
13     Alt() { System.out.println("Alt()"); }
14 }
15 // main içinde:
16 new Alt();
17 new Alt();
```



Sonra `Alt` 'ın `static` bloğu. Sınıf başlatma program boyunca **bir kez** yapılır.

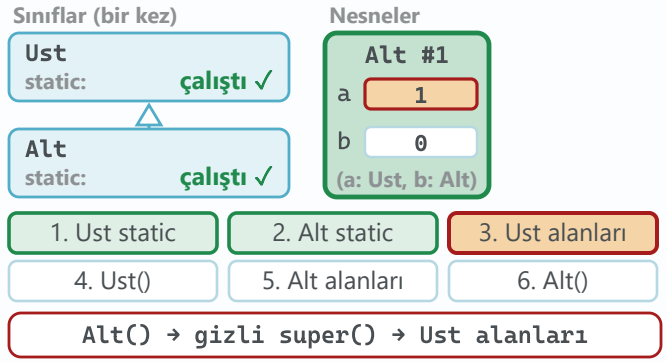
Adım 2 / 7

ÇIKTI

```
Ust static
Alt static
```

Kalıtımda İlklendirme Sırası: Kod İzleme

```
1  class Ust {
2      static { System.out.println("Ust static"); }
3      int a = deger("Ust alanı");
4      Ust() { System.out.print("Ust(), "); }
5      static int deger(String s) {
6          System.out.print(s + ", ");
7          return 1;
8      }
9  }
10 class Alt extends Ust {
11     static { System.out.println("Alt static"); }
12     int b = deger("Alt alanı");
13     Alt() { System.out.println("Alt()"); }
14 }
15 // main içinde:
16 new Alt();
17 new Alt();
```



Nesne ayrıldı (`a` ve `b` 0). `Alt()` önce gizli `super()` der; `Ust()` gövdesinden **önce** Ust'un alan ilk değerleri çalışır: `a = deger(...)` .

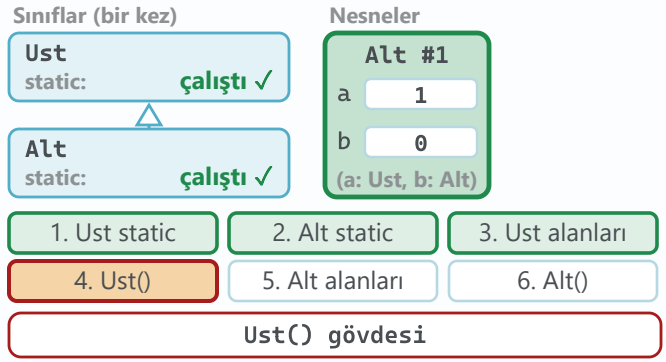
Adım 3 / 7

ÇIKTI

```
Ust static
Alt static
Ust alanı,
```

Kalıtımda İlklendirme Sırası: Kod İzleme

```
1  class Ust {
2      static { System.out.println("Ust static"); }
3      int a = deger("Ust alanı");
▶      Ust() { System.out.print("Ust(), "); }
5      static int deger(String s) {
6          System.out.print(s + ", ");
7          return 1;
8      }
9  }
10 class Alt extends Ust {
11     static { System.out.println("Alt static"); }
12     int b = deger("Alt alanı");
13     Alt() { System.out.println("Alt()"); }
14 }
15 // main içinde:
16 new Alt();
17 new Alt();
```



Ardından `Ust()` gövdesi. Bu anda `b` hâlâ 0: Alt kısmı henüz hiç ilk değer almadı.

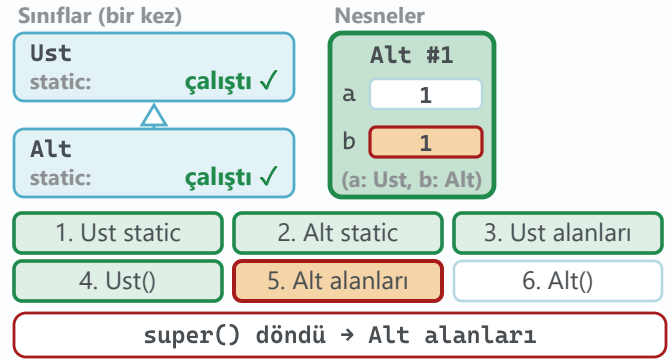
Adım 4 / 7

ÇIKTI

```
Ust static
Alt static
Ust alanı, Ust(),
```

Kalıtımda İlklendirme Sırası: Kod İzleme

```
1  class Ust {
2      static { System.out.println("Ust static"); }
3      int a = deger("Ust alanı");
4      Ust() { System.out.print("Ust(), "); }
5      static int deger(String s) {
6          System.out.print(s + ", ");
7          return 1;
8      }
9  }
10 class Alt extends Ust {
11     static { System.out.println("Alt static"); }
▶      int b = deger("Alt alanı");
13     Alt() { System.out.println("Alt()"); }
14 }
15 // main içinde:
16 new Alt();
17 new Alt();
```



`super()` döndükten sonra alt sınıfın alan ilk değerleri çalışır: `b = deger("Alt alanı")`.

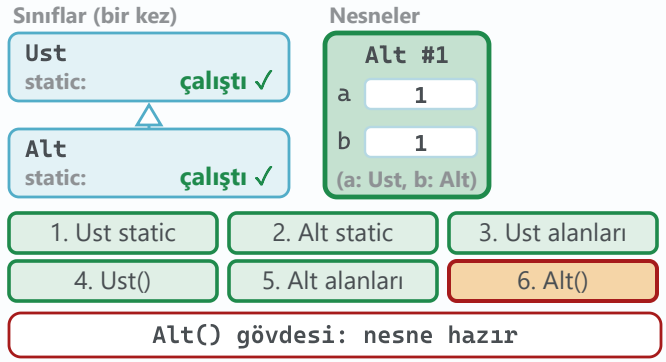
Adım 5 / 7

ÇIKTI

```
Ust static
Alt static
Ust alanı, Ust(), Alt alanı,
```

Kalıtımda İlklendirme Sırası: Kod İzleme

```
1  class Ust {
2      static { System.out.println("Ust static"); }
3      int a = deger("Ust alanı");
4      Ust() { System.out.print("Ust(), "); }
5      static int deger(String s) {
6          System.out.print(s + ", ");
7          return 1;
8      }
9  }
10 class Alt extends Ust {
11     static { System.out.println("Alt static"); }
12     int b = deger("Alt alanı");
▶      Alt() { System.out.println("Alt()"); }
14 }
15 // main içinde:
16 new Alt();
17 new Alt();
```



En son `Alt()` gövdesi. Sıra: üst sınıf alanları → üst yapıcı → alt sınıf alanları → alt yapıcı.

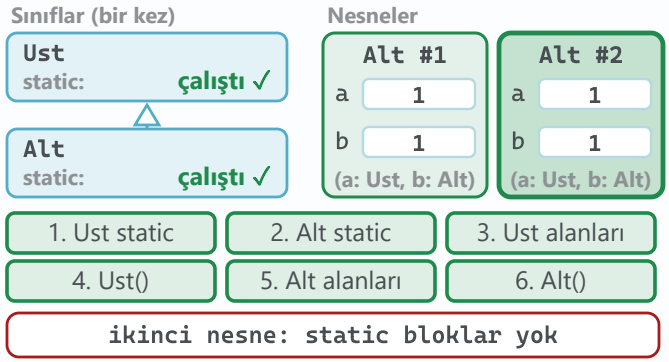
Adım 6 / 7

ÇIKTI

```
Ust static
Alt static
Ust alanı, Ust(), Alt alanı, Alt()
```

Kalıtımda İlklendirme Sırası: Kod İzleme

```
1  class Ust {
2      static { System.out.println("Ust static"); }
3      int a = deger("Ust alanı");
4      Ust() { System.out.print("Ust(), "); }
5      static int deger(String s) {
6          System.out.print(s + ", ");
7          return 1;
8      }
9  }
10 class Alt extends Ust {
11     static { System.out.println("Alt static"); }
12     int b = deger("Alt alanı");
13     Alt() { System.out.println("Alt()"); }
14 }
15 // main içinde:
16 new Alt();
▶  new Alt();
```



İkinci `new Alt()` : sınıflar zaten başlatıldı, `static` bloklar **tekrar çalışmaz**. Nesneye ait dört adım aynı sırayla yinelenir.

Adım 7 / 7

ÇIKTI

```
Ust static
Alt static
Ust alanı, Ust(), Alt alanı, Alt()
Ust alanı, Ust(), Alt alanı, Alt()
```

Metodu Geçersiz Kılma

Geçersiz kılma (overriding): alt sınıfın, üst sınıftan kalıtımla aldığı bir metodu **aynı imzayla** yeniden yazarak kendi davranışını vermesidir.

- Ad ve parametre listesi **aynı** olmalı.
- Dönüş tipi aynı (veya onun alt tipi) olmalı.
- Erişim **daraltılamaz**: `public` metot `public` kalmalı.
- `final`, `static` ve `private` metotlar geçersiz kılınamaz.

```
class Sekil {
    public void ciz() {
        System.out.println("Şekil çiziliyor.");
    }
}
class Dikdortgen extends Sekil {
    @Override
    public void ciz() {
        System.out.println("Dikdörtgen çiziliyor.");
    }
}
// main içinde
new Sekil().ciz();
new Dikdortgen().ciz();
```

```
Şekil çiziliyor.
Dikdörtgen çiziliyor.
```

```
// alt sınıfta: void ciz()
error: ciz() in Dikdortgen cannot override
  ciz() in Sekil
  attempting to assign weaker access
  privileges; was public
```

```
// alt sınıfta: public int ciz()
error: ciz() in Dikdortgen cannot override
  ciz() in Sekil
  return type int is not compatible
  with void
```

Geçersiz Kılma ve Aşırı Yükleme

	Geçersiz kılma (overriding)	Aşırı yükleme (overloading)
Nerede?	Alt sınıfta, kalıtımla alınan metot için	Aynı sınıfta (veya alt sınıfta)
Ad	Aynı	Aynı
Parametreler	**Aynı**	**Farklı** (sayı ya da tip)
Dönüş tipi	Aynı veya alt tipi	Serbest
Amaç	Kalıtımla alınan davranışı **değiştirmek**	Aynı işi farklı girdilerle yapmak
Seçim zamanı	Çalışma zamanı (nesnenin gerçek tipi)	Derleme zamanı (argüman tipleri)

```
class Dikdortgen extends Sekil {
    @Override
    public void ciz() { ... }              // geçersiz kılma
    public void ciz(String renk) { ... }  // aşırı yükleme
}
```

`d.ciz("Mavi")` ÇIKTISI

```
Mavi dikdörtgen çiziliyor.
```

Her iki kavram da **çok biçimliliğin** (polymorphism) temelidir.

GEÇERSİZ KILMA

@Override Etiketi

- `@Override` bir **etikettir** (annotation): metodun, üst sınıftan kalıtımla alınan bir metodu geçersiz kıldığını bildirir.
- Derleyiciye bunu **kontrol etmesini** söyler: üst sınıfta böyle bir metot yoksa derleme hatası verir.
- Kodu daha okunabilir yapar; diğer geliştiricilere metodun üst sınıftan geldiğini gösterir.
- Zorunlu değildir, ama **her zaman yazılması** önerilir: yazım hatalarını hemen yakalar.

```
public class Dikdortgen extends Sekil {
    // miras alınan metodu geçersiz kıl
    @Override
    public void ciz() {
        System.out.println("Dikdörtgen çiziliyor.");
    }
}
```

YAZIM HATASI: `Ciz` (BÜYÜK C)

```
    @Override
    public void Ciz() { ... }
```

```
error: method does not override or implement
  a method from a supertype
```

`@Override` olmasaydı `Ciz()` sessizce yeni bir metot olurdu ve `ciz()` çağrıları hâlâ Sekil'in sürümünü çalıştırırdı.

Geçersiz Kılma Kuralları: Ayrıntılı

Özellik	Alt sınıftaki metotta izin verilen	Kural bozulursa `javac`
Ad ve parametreler	Birebir aynı	Hata yok: yeni bir metot (aşırı yükleme) olur
Dönüş tipi	Aynı ya da bir alt tipi (kovaryant)	`return type String is not compatible with Hayvan`
Erişim	Aynı ya da daha geniş: `protected` → `public`	`attempting to assign weaker access privileges`
Kontrol edilen istisna	Aynısı, daha özeli ya da hiç	`overridden method does not throw Exception`
`static`	Örnek metodu `static` olamaz, `static` metot örnek metodu olamaz	`overriding method is static` / `overridden method is static`
`final` metot	Geçersiz kılınamaz	`overridden method is final`
`private` metot	Görünmez; aynı adlı metot ilgisiz yeni bir metottur	Hata yok (`@Override` varsa hata)

Ortak gerekçe: alt sınıf nesnesi, üst sınıf nesnesinin beklendiği her yerde kullanılabilmelidir. Üst sınıfın verdiği söz (ne döndüreceği, kime açık olduğu, hangi istisnaları fırlatabileceği) alt sınıfta **genişletilebilir** ya da korunabilir, ama **daraltılamaz**.

GEÇERSİZ KILMA

Kovaryant Dönüş Tipi

```
class Hayvan {
    Hayvan yavrula() { return new Hayvan(); }
    public String toString() { return "Hayvan"; }
}
class Kedi extends Hayvan {
    @Override
    Kedi yavrula() { return new Kedi(); }  // alt tip
    void miyavla() { System.out.println("miyav"); }
    public String toString() { return "Kedi"; }
}
// main içinde:
Kedi k = new Kedi();
k.yavrula().miyavla();   // dönüşüm gerekmez
Hayvan h = k;
System.out.println(h.yavrula());
```

```
miyav
Kedi
```

- Geçersiz kılan metot, üst sınıftaki dönüş tipinin bir **alt tipini** döndürebilir: **kovaryant dönüş tipi** (covariant return type).
- `Kedi` referansıyla çağıran, sonucu doğrudan `Kedi` olarak alır; `(Kedi)` dönüşümüne gerek kalmaz.
- `Hayvan` referansıyla çağıran için söz bozulmaz: bir `Kedi` de bir `Hayvan`'dır.
- Yalnızca referans tiplerinde geçerlidir; `int` yerine `long` döndürmek olmaz.

İLGİSİZ TİP: `String yavrula()`

```
error: yavrula() in Kedi cannot override
  yavrula() in Hayvan
  return type String is not compatible
with Hayvan
```

GEÇERSİZ KILMA

`private` Metotlar Geçersiz Kılınmaz

```
class Ust {
    private void selam() { System.out.println("Ust.selam"); }
    void cagir() { selam(); }
}
class Alt extends Ust {
    void selam() { System.out.println("Alt.selam"); } // yeni
}
// main içinde:
Alt a = new Alt();
a.cagir();
a.selam();
```

```
Ust.selam
Alt.selam
```

`Alt` 'TAKİ METODA `@Override` EKLENİRSE

```
error: method does not override or implement
  a method from a supertype
```

- `private` bir metot alt sınıfa görünmez; alt sınıfta aynı adla yazılan metot onunla **ilgisiz**, yeni bir metottur.
- `Ust.cagir()` içindeki `selam()` çağrısı derleme sırasında `Ust` 'un kendi `private` metoduna bağlanır. Nesne bir `Alt` olsa da `Ust.selam` yazılır.
- `@Override` alışkanlığı bu yanılgıyı derleme anında ortaya çıkarır.

Aynı durum `static` metotlarda da görülür: alt sınıftaki aynı imzalı `static` metot üst sınıfınkini geçersiz kılmaz, yalnızca **gizler** (hiding).

Örnek: Ürün Hiyerarşisi

```
class Urun {
    private String ad;
    private double fiyat;
    Urun(String ad, double fiyat) {
        this.ad = ad; this.fiyat = fiyat;
    }
    double kdvOrani() { return 0.20; }
    double satisFiyati() { return fiyat * (1 + kdvOrani()); }
    @Override public String toString() {
        double f = Math.round(satisFiyati() * 100) / 100.0;
        return ad + ": " + f + " TL";
    }
}
class GidaUrunu extends Urun {
    GidaUrunu(String ad, double fiyat) { super(ad, fiyat); }
    @Override double kdvOrani() { return 0.01; }
}
class IndirimliUrun extends Urun {
    private double indirim;
    IndirimliUrun(String ad, double fiyat, double indirim) {
        super(ad, fiyat);
        this.indirim = indirim;
    }
    @Override double satisFiyati() {
        return super.satisFiyati() * (1 - indirim);
    }
    @Override public String toString() {
        return super.toString()
            + " (%" + (int) (indirim * 100) + " indirim)";
    }
}
```

```
Urun[] sepet = {new Urun("Kalem", 50),
    new GidaUrunu("Ekmek", 10),
    new IndirimliUrun("Kulaklık", 1000, 0.25)};
for (Urun u : sepet) System.out.println(u);
```

```
Kalem: 60.0 TL
Ekmek: 10.1 TL
Kulaklık: 900.0 TL (%25 indirim)
```

- `GidaUrunu` yalnızca `kdvOrani()`'nı değiştirir; `satisFiyati()` ve `toString()` yazılmadığı hâlde %1 ile hesaplanır, çünkü `Urun`'un içindeki çağrı nesnenin sınıfına göre seçilir.
- `IndirimliUrun` üst sınıfın hesabını **değiştirmez, genişletir**: `super.satisFiyati()` sonucuna indirimi uygular. 1000 × 1,20 × 0,75 = 900.

ERİŞİM VE SINIRLAMA

`protected` Erişim Düzeyi

Belirteç	Aynı sınıf	Aynı paket	Alt sınıf (başka paket)	Her yer
`private`	✓	✗	✗	✗
(belirteç yok: paket)	✓	✓	✗	✗
`protected`	✓	✓	✓	✗
`public`	✓	✓	✓	✓

- `public` : sınıfın üyelerine her yerden erişilebilir. `private` : yalnızca o sınıfın içinden.
- `protected` : ikisinin arasında bir düzeydir. Sınıfın kendisi, **alt sınıfları** ve **aynı paketteki** diğer sınıflar erişebilir.
- Bir sınıfın `private` alanlarına alt sınıf, yalnızca üst sınıftan kalıtımla alınan `public` veya `protected` metotlar aracılığıyla erişebilir.

İyi uygulama: alanlar yine `private` kalsın; alt sınıfların ihtiyaç duyduğu işlemler `protected` metotlar olarak açılsın.

protected: Kod Örneği

```java
package canli;

public class Hayvan {
    protected String isim;  // alt sınıflar
    private double saglik;  // yalnızca Hayvan

    public Hayvan(String isim) {
        this.isim = isim;
        saglik = 100.0;
    }
    public double getSaglik() { return saglik; }
    protected void saglikEkle(double miktar) {
        saglik += miktar;
    }
}
```

```java
package evcil;     // başka paket
import canli.Hayvan;

public class Kedi extends Hayvan {
    public Kedi(String isim) { super(isim); }
    public void uyu() {
        // protected alan: izin var
        System.out.println(isim + " uyuyor.");
        // saglik += 5;  // private: hata
        saglikEkle(5);   // protected metot
        System.out.println("Sağlık: "
                           + getSaglik());
    }
}
```

`new Kedi("Mavi").uyu();` ÇIKTISI

```
Mavi uyuyor.
Sağlık: 105.0
```

YORUM SATIRI AÇILIRSA

```
error: saglik has private access in Hayvan
```

`protected` Erişimin İnce Kuralı

```java
package canli;
public class Hayvan {
    protected String isim;
    public Hayvan(String isim) {
        this.isim = isim;
    }
}
```

```java
package evcil;
import canli.Hayvan;
public class Kedi extends Hayvan {
    public Kedi(String isim) { super(isim); }
    void karsilastir(Kedi k, Hayvan h) {
        System.out.println(isim);    // kendi alanı
        System.out.println(k.isim);  // Kedi üzerinden
        System.out.println(h.isim);  // Hayvan üzerinden
    }
}
```

```
evcil/Kedi.java:8: error: isim has protected access in Hayvan
        System.out.println(h.isim);
```

- Başka paketteki alt sınıf, `protected` üyeye yalnızca **kendi tipinden** (ya da kendi alt tiplerinden) bir referans üzerinden erişebilir: `isim` ve `k.isim` geçerlidir.
- `h` ise herhangi bir `Hayvan`'ı, örneğin bir `Kopek`'i gösterebilir. `Kedi`'nin başka bir alt sınıfın iç verisine dokunması engellenir.
- Aynı paketteki sınıflar için bu kısıt yoktur: orada `protected`, paket erişimini de kapsar.

`final` ile Kalıtımı Sınırlamak

- `final` anahtar sözcüğü kalıtımı **sınırlamak** için kullanılabilir.
- Bir **metot** `final` tanımlanırsa hiçbir alt sınıfta geçersiz kılınamaz (override).
- Bir **sınıf** `final` tanımlanırsa ondan hiçbir alt sınıf türetilemez. Örnek: `String`, `Math`, `Integer`.

```
class Hayvan {
    public final void nefesAl() { ... }
}
class Kedi extends Hayvan {
    public void nefesAl() { ... }
}
```

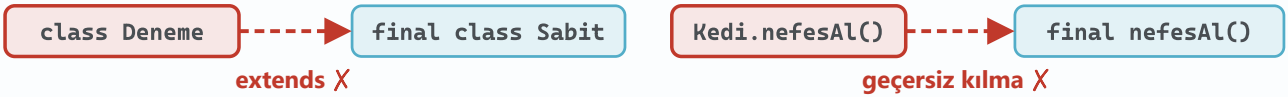
```
error: nefesAl() in Kedi cannot override
  nefesAl() in Hayvan
  overridden method is final
```

```
final class Sabit { ... }
class Deneme extends Sabit { }
```

```
error: cannot inherit from final Sabit
```

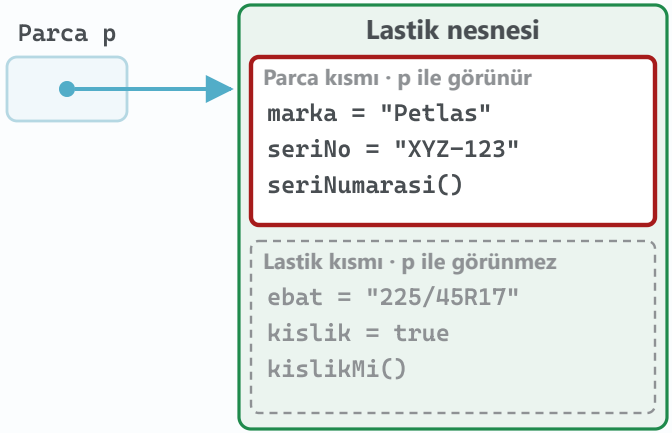
Neden? Güvenlik ve tutarlılık: `String` değiştirilemez olmalıdır; bir alt sınıf bu sözü bozamasın.

Değişkenlerde `final` başka bir anlam taşır: değer bir kez atanır (sabit).



Nesnenin Görünümünü Kısıtlama

- Bir nesne, **üst sınıf tipinde** bir değişkene atanabilir (yukarı dönüşüm, upcasting): her Lastik bir Parça'dır.
- Böylece dışarıya yalnızca **üst sınıfa ait** metotlar açılır.
- Nesnenin gerçek sınıfına bakılmaksızın, değişkenin tipi neyi **çağırabileceğimizi** belirler.




```
Parca p = new Lastik("Petlas", "XYZ-123", "225/45R17", true);
System.out.println(p.seriNumarasi());   // izin verilir → XYZ-123
// p.kislikMi()                         // izin verilmez
```

```
error: cannot find symbol
  symbol:   method kislikMi()
  location: variable p of type Parca
```

Lastik'e özgü metoda ulaşmak için önce tip kontrolü, sonra aşağı dönüşüm (downcasting) gerekir (`instanceof`).

SINIFLAR ARASI İLİŞKİLER

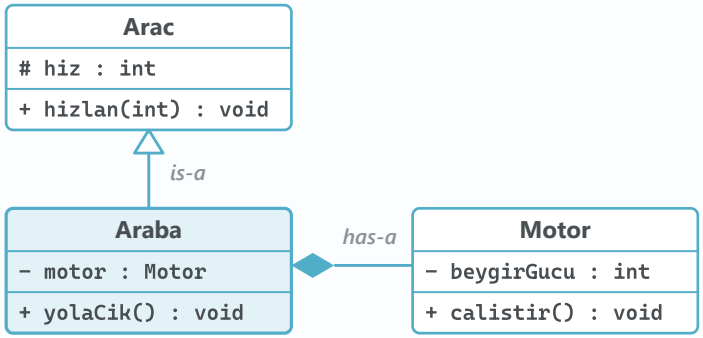
is-a ve has-a İlişkileri

Is-a (kalıtım, inheritance)

- Bir sınıfın başka bir sınıftan türetilmesidir; türetilen sınıf üst sınıfın **bir türüdür**.
- Alt sınıf nesnesi, üst sınıfın nesnesi gibi de kullanılabilir.

Has-a (oluşum, composition)

- Bir sınıfın başka bir sınıf tipinde alan içermesidir.
- Nesne, içinde diğer nesnelere **referanslar** tutar.




```java
// Is-a ilişkisi (kalıtım)
public class Araba extends Arac {
    // Has-a ilişkisi (oluşum)
    private Motor motor;

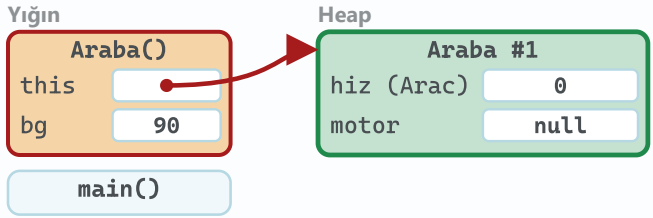
    public Araba(int beygirGucu) {
        motor = new Motor(beygirGucu);
    }
    public void yolaCik() {
        motor.calistir();   // Motor'a devret
        hizlan(50);         // Arac'tan miras
        System.out.println("Hız: " + hiz
                           + " km/sa");
    }
}
// Motor: beygirGucu alanı ve calistir():
//   "Motor çalıştı: " + beygirGucu + " BG"
```

```java
new Araba(110).yolaCik();
```

```
Motor çalıştı: 110 BG
Hız: 50 km/sa
```

is-a ve has-a: Kod İzleme

```
1  class Motor {
2      private int bg;
3      Motor(int bg) { this.bg = bg; }
4      void calistir() {
5          System.out.println("Motor: " + bg + " BG");
6      }
7  }
8  class Araba extends Arac {        // is-a
9      private Motor motor;           // has-a
10     Araba(int bg) { motor = new Motor(bg); }
11     void yolaCik() {
12         motor.calistir();  hizlan(50);
13         System.out.println("Hız: " + hiz);
14     }
15 }
▶  new Araba(90).yolaCik();   // main
```



`Arac kısmı (hiz) + Araba kısmı (motor)`

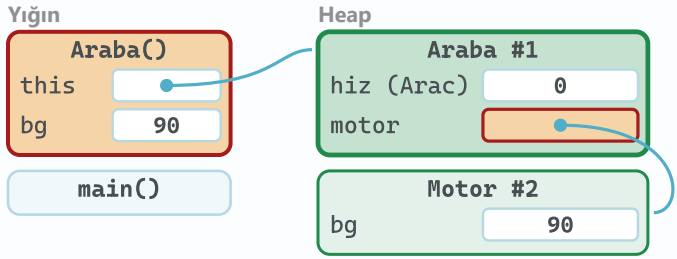
`new Araba(90)` : tek nesne, içinde üst sınıf `Arac` 'tan gelen `hiz` ve kendi `motor` alanı var (**is-a**).

Adım 1 / 6

ÇIKTI

is-a ve has-a: Kod İzleme

```
1  class Motor {
2      private int bg;
▶      Motor(int bg) { this.bg = bg; }
4      void calistir() {
5          System.out.println("Motor: " + bg + " BG");
6      }
7  }
8  class Araba extends Arac {        // is-a
9      private Motor motor;           // has-a
▶      Araba(int bg) { motor = new Motor(bg); }
11     void yolaCik() {
12         motor.calistir();  hizlan(50);
13         System.out.println("Hız: " + hiz);
14     }
15 }
16 new Araba(90).yolaCik();   // main
```



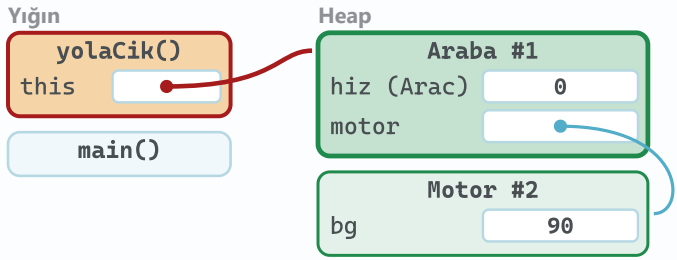
Yapıcı ikinci bir nesne oluşturur: araba bir motor **değildir**, motora **sahiptir** (**has-a**). Bağlantı bir referanstır.

Adım 2 / 6

ÇIKTI

is-a ve has-a: Kod İzleme

```
1  class Motor {
2      private int bg;
3      Motor(int bg) { this.bg = bg; }
4      void calistir() {
5          System.out.println("Motor: " + bg + " BG");
6      }
7  }
8  class Araba extends Arac {        // is-a
9      private Motor motor;           // has-a
10     Araba(int bg) { motor = new Motor(bg); }
▶      void yolaCik() {
12         motor.calistir();  hizlan(50);
13         System.out.println("Hız: " + hiz);
14     }
15 }
▶  new Araba(90).yolaCik();   // main
```



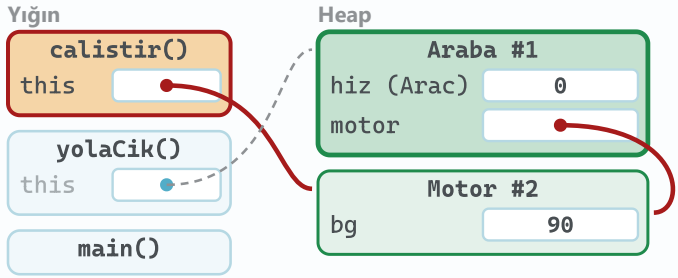
`yolaCik()` Araba nesnesi üzerinde çalışır.

Adım 3 / 6

ÇIKTI

is-a ve has-a: Kod İzleme

```
1  class Motor {
2      private int bg;
3      Motor(int bg) { this.bg = bg; }
▶      void calistir() {
▶          System.out.println("Motor: " + bg + " BG");
6      }
7  }
8  class Araba extends Arac {        // is-a
9      private Motor motor;           // has-a
10     Araba(int bg) { motor = new Motor(bg); }
11     void yolaCik() {
▶          motor.calistir();  hizlan(50);
13         System.out.println("Hız: " + hiz);
14     }
15 }
16 new Araba(90).yolaCik();    // main
```



`motor.calistir()` → işi **Motor** nesnesi yapar

`motor.calistir()` : Araba işi kendi yapmaz, referansı izleyip Motor nesnesine **devreder**.

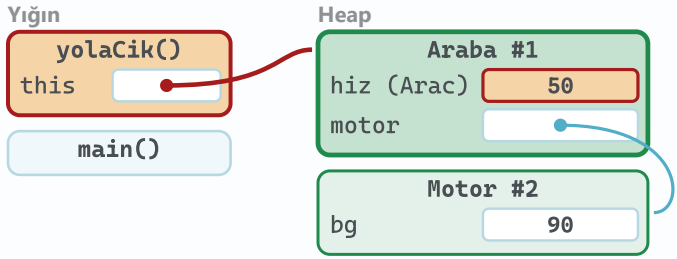
Adım 4 / 6

ÇIKTI

```
Motor: 90 BG
```

is-a ve has-a: Kod İzleme

```
 1 class Motor {
 2     private int bg;
 3     Motor(int bg) { this.bg = bg; }
 4     void calistir() {
 5         System.out.println("Motor: " + bg + " BG");
 6     }
 7 }
 8 class Araba extends Arac {        // is-a
 9     private Motor motor;           // has-a
10     Araba(int bg) { motor = new Motor(bg); }
11     void yolaCik() {
▶           motor.calistir();  hizlan(50);
13         System.out.println("Hız: " + hiz);
14     }
15 }
16 new Araba(90).yolaCik();   // main
```



hizlan(50): Arac'tan kalıtımla gelen metot

`hizlan(50)` Araba'da yazılmadı; `Arac`'tan kalıtımla gelir ve aynı nesnenin `hiz` alanını 50 yapar.

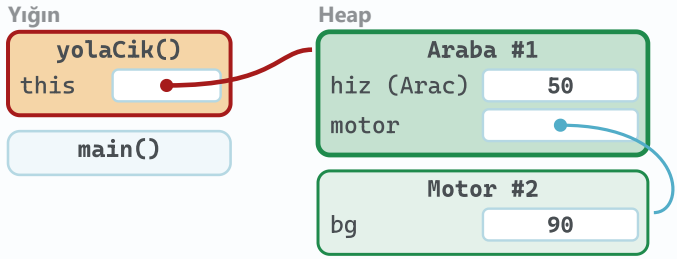
Adım 5 / 6

ÇIKTI

```
Motor: 90 BG
```

is-a ve has-a: Kod İzleme

```
 1 class Motor {
 2     private int bg;
 3     Motor(int bg) { this.bg = bg; }
 4     void calistir() {
 5         System.out.println("Motor: " + bg + " BG");
 6     }
 7 }
 8 class Araba extends Arac {        // is-a
 9     private Motor motor;           // has-a
10     Araba(int bg) { motor = new Motor(bg); }
11     void yolaCik() {
12         motor.calistir();  hizlan(50);
▶           System.out.println("Hız: " + hiz);
14     }
15 }
16 new Araba(90).yolaCik();   // main
```



Kalıtım (`hiz` , `hizlan`) ve oluşum (`motor`) aynı sınıfta birlikte çalıştı.

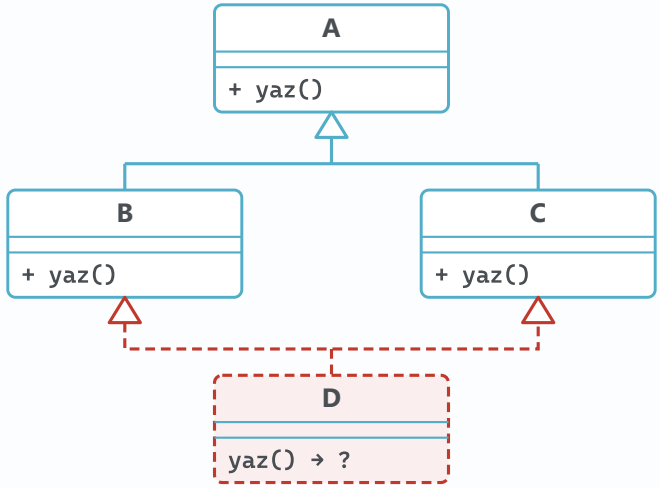
Adım 6 / 6

ÇIKTI

```
Motor: 90 BG
Hız: 50
```

Çoklu Kalıtım

- Java'da **doğrudan çoklu kalıtım desteklenmez**: bir sınıf yalnızca **bir** sınıftan türeyebilir.
- **Elmas problemi** (diamond problem): B ve C `yaz()` 'ı farklı geçersiz kılarsa D hangisini almalı?
- Birden fazla sınıfın özellikleri gerekiyorsa: **oluşum** (composition) ve **arayüzler** (interface; bir sınıf birçok arayüz uygulayabilir).




```
class D extends B, C { }   // geçersiz
```

```
error: '{' expected
```

```
interface Sarj { void sarjEt(); }
class ElektrikliAaraba extends Araba
                       implements Sarj {
    @Override
    public void sarjEt() {
        System.out.println("Şarj ediliyor.");
    }
}
```

SINIFLAR ARASI İLİŞKİLER

Kalıtımla Tasarım İlkeleri

“is-a” testi

“Her X bir Y midir?” cümlesi doğru değilse kalıtım kullanılmaz. Araba bir Motor **değildir**; Motor'a **sahiptir** → oluşum.

Herkes kendi verisini yönetir

Üst sınıfın alanlarına `super(...)` ve üst sınıfın metotlarıyla değer verilir; alanlar `private` tutulur.

Genel metotlar geçersiz kılınır

`toString` ve `equals` (ve onunla birlikte `hashCode`) her anlamlı sınıfta geçersiz kılınmalıdır.

Ortak kavramlar soyut sınıfa

Alt sınıfların ortak kavramları **soyut** (abstract) sınıflarla temsil edilmelidir: “Hayvan” tek başına nesne olmamalıysa.

SINIFLAR ARASI İLİŞKİLER

Kalıtım mı, Oluşum mu?

	Kalıtım (`extends`)	**Oluşum (alan olarak nesne)**
İlişki	is-a: "her X bir Y'dir"	has-a: "X'in bir Y'si vardır"
Bağlılık	Sıkı: alt sınıf, üst sınıfın **gerçeklemesine** de bağlıdır	Gevşek: yalnızca içteki nesnenin `public` metotlarına bağlıdır
Yeniden kullanım	Otomatik: bütün `public` metotlar gelir (istenmeyenler de)	Seçerek: yalnızca iletilmek istenen metotlar yazılır
Değişiklik	Üst sınıf derleme zamanında sabittir	Alan, çalışma zamanında başka bir nesneyle değiştirilebilir
Üst tip yerine kullanım	Doğrudan: `Hayvan h = new Kedi()`	Ortak bir arayüz uygulanırsa

Kalıtım uygundur

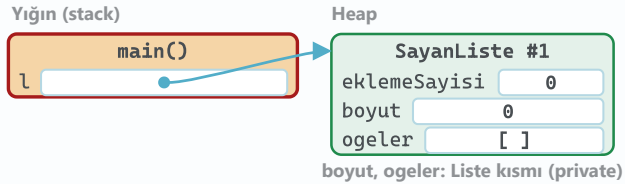
is-a ilişkisi gerçekten doğruysa ve üst sınıf genişletilmek üzere tasarlanıp belgelendiyse: `Kedi` – `Hayvan`.

Oluşum uygundur

Amaç yalnızca başka bir sınıfın kodunu kullanmaksa: `Yigin`, bir `ArrayList` **değildir**; listeyi içinde tutar ve yalnızca `ekle` / `cikar` sunar.

Kırılgan Üst Sınıf: Kod İzleme

```
 1 class Liste {
 2     private String[] ogeler = new String[10];
 3     private int boyut;
 4     void ekle(String s) { ogeler[boyut++] = s; }
 5     void hepsiniEkle(String[] dizi) {
 6         for (String s : dizi) ekle(s);
 7     }
 8     int boyut() { return boyut; }
 9 }
10 class SayanListe extends Liste {
11     int eklemeSayisi = 0;
12     @Override void ekle(String s) {
13         eklemeSayisi++;
14         super.ekle(s);
15     }
16     @Override void hepsiniEkle(String[] dizi) {
17         eklemeSayisi += dizi.length;
18         super.hepsiniEkle(dizi);
19     }
20 }
▶ SayanListe l = new SayanListe();   // main
22 l.hepsiniEkle(new String[] {"a", "b", "c"});
23 System.out.println(l.eklemeSayisi + " " + l.boyut());
```



üç öğe eklenecek: beklenen sayaç 3

`SayanListe`, eklenen öğe sayısını tutmak için `Liste`'den türetildi. İki ekleme metodu da geçersiz kılındı.

Adım 1 / 7

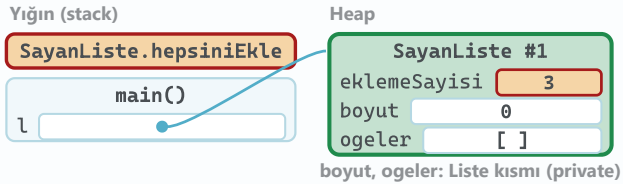
ÇIKTI

DEĞİŞKENLER

`eklemeSayisi = 0` `boyut = 0`

Kırılgan Üst Sınıf: Kod İzleme

```
1  class Liste {
2      private String[] ogeler = new String[10];
3      private int boyut;
4      void ekle(String s) { ogeler[boyut++] = s; }
5      void hepsiniEkle(String[] dizi) {
6          for (String s : dizi) ekle(s);
7      }
8      int boyut() { return boyut; }
9  }
10 class SayanListe extends Liste {
11     int eklemeSayisi = 0;
12     @Override void ekle(String s) {
13         eklemeSayisi++;
14         super.ekle(s);
15     }
16     @Override void hepsiniEkle(String[] dizi) {
▶          eklemeSayisi += dizi.length;
18         super.hepsiniEkle(dizi);
19     }
20 }
21 SayanListe l = new SayanListe();   // main
22 l.hepsiniEkle(new String[] {"a", "b", "c"});
23 System.out.println(l.eklemeSayisi + " " + l.boyut());
```



`dizi.length = 3 eklendi`

Alt sınıfın `hepsiniEkle`'si sayaca dizinin uzunluğunu ekler: 3. Sonra işi üst sınıfa bırakır.

Adım 2 / 7

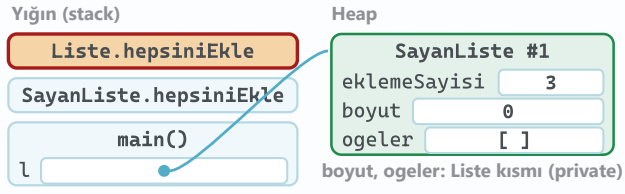
ÇIKTI

DEĞİŞKENLER

`eklemeSayisi = 3` `boyut = 0`

Kırılgan Üst Sınıf: Kod İzleme

```
1  class Liste {
2      private String[] ogeler = new String[10];
3      private int boyut;
4      void ekle(String s) { ogeler[boyut++] = s; }
5      void hepsiniEkle(String[] dizi) {
▶          for (String s : dizi) ekle(s);
7      }
8      int boyut() { return boyut; }
9  }
10 class SayanListe extends Liste {
11     int eklemeSayisi = 0;
12     @Override void ekle(String s) {
13         eklemeSayisi++;
14         super.ekle(s);
15     }
16     @Override void hepsiniEkle(String[] dizi) {
17         eklemeSayisi += dizi.length;
▶          super.hepsiniEkle(dizi);
19     }
20 }
21 SayanListe l = new SayanListe();   // main
22 l.hepsiniEkle(new String[] {"a", "b", "c"});
23 System.out.println(l.eklemeSayisi + " " + l.boyut());
```



`Liste döngüsü kendi ekle()'sini çağırır`

`Liste.hepsiniEkle` her öğe için `ekle(s)`, yani `this.ekle(s)` çağırır. Bu, üst sınıfın bir **gerçekleme ayrıntısıdır**.

Adım 3 / 7

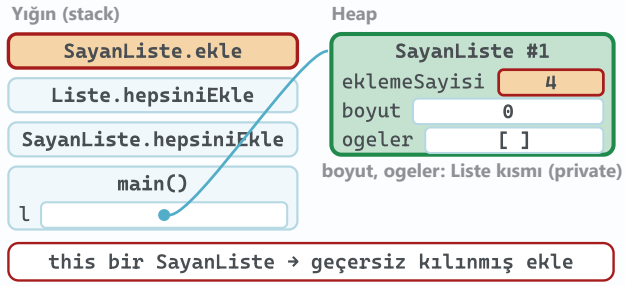
ÇIKTI

DEĞİŞKENLER

`eklemeSayisi = 3` `boyut = 0` `s = "a"`

Kırılgan Üst Sınıf: Kod İzleme

```
1  class Liste {
2      private String[] ogeler = new String[10];
3      private int boyut;
4      void ekle(String s) { ogeler[boyut++] = s; }
5      void hepsiniEkle(String[] dizi) {
6          for (String s : dizi) ekle(s);
7      }
8      int boyut() { return boyut; }
9  }
10 class SayanListe extends Liste {
11     int eklemeSayisi = 0;
12     @Override void ekle(String s) {
▶          eklemeSayisi++;
14         super.ekle(s);
15     }
16     @Override void hepsiniEkle(String[] dizi) {
17         eklemeSayisi += dizi.length;
18         super.hepsiniEkle(dizi);
19     }
20 }
21 SayanListe l = new SayanListe();   // main
22 l.hepsiniEkle(new String[] {"a", "b", "c"});
23 System.out.println(l.eklemeSayisi + " " + l.boyut());
```



Nesne bir `SayanListe` olduğu için çağrı alt sınıfın `ekle`'sine gider. Aynı öğe **ikinci kez** sayılır: 4.

Adım 4 / 7

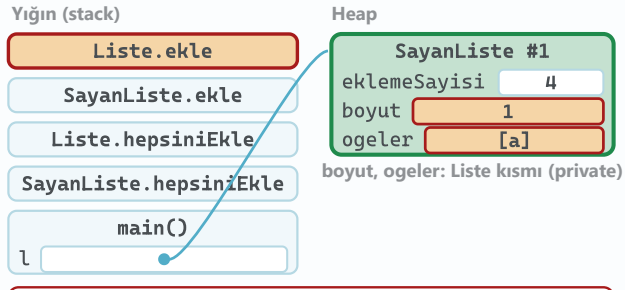
ÇIKTI

DEĞİŞKENLER

`eklemeSayisi = 4` `boyut = 0` `s = "a"`

Kırılgan Üst Sınıf: Kod İzleme

```
1  class Liste {
2      private String[] ogeler = new String[10];
3      private int boyut;
▶      void ekle(String s) { ogeler[boyut++] = s; }
5      void hepsiniEkle(String[] dizi) {
6          for (String s : dizi) ekle(s);
7      }
8      int boyut() { return boyut; }
9  }
10 class SayanListe extends Liste {
11     int eklemeSayisi = 0;
12     @Override void ekle(String s) {
13         eklemeSayisi++;
▶          super.ekle(s);
15     }
16     @Override void hepsiniEkle(String[] dizi) {
17         eklemeSayisi += dizi.length;
18         super.hepsiniEkle(dizi);
19     }
20 }
21 SayanListe l = new SayanListe();   // main
22 l.hepsiniEkle(new String[] {"a", "b", "c"});
23 System.out.println(l.eklemeSayisi + " " + l.boyut());
```



`super.ekle(s)` öğeyi diziye yazar: `boyut = 1`.

Adım 5 / 7

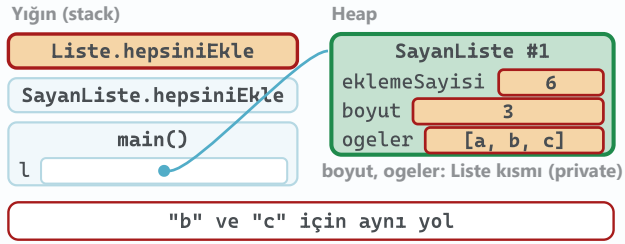
ÇIKTI

DEĞİŞKENLER

`eklemeSayisi = 4` `boyut = 1` `s = "a"`

Kırılgan Üst Sınıf: Kod İzleme

```
1  class Liste {
2      private String[] ogeler = new String[10];
3      private int boyut;
4      void ekle(String s) { ogeler[boyut++] = s; }
5      void hepsiniEkle(String[] dizi) {
6          for (String s : dizi) ekle(s);
7      }
8      int boyut() { return boyut; }
9  }
10 class SayanListe extends Liste {
11     int eklemeSayisi = 0;
12     @Override void ekle(String s) {
13         eklemeSayisi++;
14         super.ekle(s);
15     }
16     @Override void hepsiniEkle(String[] dizi) {
17         eklemeSayisi += dizi.length;
18         super.hepsiniEkle(dizi);
19     }
20 }
21 SayanListe l = new SayanListe();   // main
22 l.hepsiniEkle(new String[] {"a", "b", "c"});
23 System.out.println(l.eklemeSayisi + " " + l.boyut());
```



"b" ve "c" için de aynı yol izlenir: her biri bir kez daha sayılır. Döngü bitince sayaç 6, boyut 3.

Adım 6 / 7

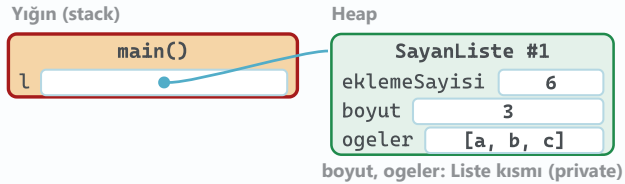
ÇIKTI

DEĞİŞKENLER

`eklemeSayisi = 6` `boyut = 3`

Kırılgan Üst Sınıf: Kod İzleme

```
1  class Liste {
2      private String[] ogeler = new String[10];
3      private int boyut;
4      void ekle(String s) { ogeler[boyut++] = s; }
5      void hepsiniEkle(String[] dizi) {
6          for (String s : dizi) ekle(s);
7      }
8      int boyut() { return boyut; }
9  }
10 class SayanListe extends Liste {
11     int eklemeSayisi = 0;
12     @Override void ekle(String s) {
13         eklemeSayisi++;
14         super.ekle(s);
15     }
16     @Override void hepsiniEkle(String[] dizi) {
17         eklemeSayisi += dizi.length;
18         super.hepsiniEkle(dizi);
19     }
20 }
21 SayanListe l = new SayanListe();   // main
22 l.hepsiniEkle(new String[] {"a", "b", "c"});
▶  System.out.println(l.eklemeSayisi + " " + l.boyut());
```



`üç öğe, sayaç 6: kırılgan üst sınıf`

Yanlış sonuç: `6 3`. Alt sınıf, üst sınıfın kendi metotlarını nasıl kullandığına bağlı kaldı. Üst sınıfın yeni bir sürümü döngüyü değiştirse sonuç yine değişirdi: **kırılgan üst sınıf** (fragile base class).

Adım 7 / 7

ÇIKTI

```
6 3
```

DEĞİŞKENLER

`eklemeSayisi = 6` `boyut = 3`

Oluşumla Çözüm: İletme

```
class SayanListe2 {
    private final Liste liste = new Liste(); // has-a
    private int eklemeSayisi = 0;
    void ekle(String s) {
        eklemeSayisi++;
        liste.ekle(s);            // iletir
    }
    void hepsiniEkle(String[] dizi) {
        eklemeSayisi += dizi.length;
        liste.hepsiniEkle(dizi);  // iletir
    }
    int getEklemeSayisi() { return eklemeSayisi; }
    int boyut() { return liste.boyut(); }
}
```

```
SayanListe2 l2 = new SayanListe2();
l2.hepsiniEkle(new String[] {"a", "b", "c"});
System.out.println(l2.getEklemeSayisi()
    + " " + l2.boyut());
```

```
3 3
```

- Yeni sınıf bir `Liste`'den türemez, bir `Liste`'ye **sahiptir**. Her metot önce kendi işini yapar, sonra çağrıyı içteki nesneye **iletir** (forwarding).
- `liste.hepsiniEkle(...)` içindeki `ekle` çağrıları düz bir `Liste` nesnesinde çalışır; `SayanListe2`'ye geri dönmez. Sayaç doğrudur: 3.
- `Liste`'nin iç ayrıntısı değişse de sonuç değişmez: bağlılık yalnızca `public` metotlaradır.
- Bedeli: `SayanListe2` bir `Liste` beklenen yere verilemez. Gerekirse ikisi ortak bir arayüzü uygular.

Yerine Koyma İlkesi (LSP)

```
class Dikdortgen {
    protected int en, boy;
    void setEn(int en) { this.en = en; }
    void setBoy(int boy) { this.boy = boy; }
    int alan() { return en * boy; }
}
class Kare extends Dikdortgen {
    @Override void setEn(int k) { en = k; boy = k; }
    @Override void setBoy(int k) { en = k; boy = k; }
}
static void genislet(Dikdortgen d) {
    d.setEn(5);
    d.setBoy(4);
    System.out.println("Beklenen 20, bulunan "
        + d.alan());
}
// main içinde:
genislet(new Dikdortgen());
genislet(new Kare());
```

```
Beklenen 20, bulunan 20
Beklenen 20, bulunan 16
```

- **Liskov yerine koyma ilkesi** (Liskov substitution principle): üst sınıf tipiyle yazılmış kod, bir alt sınıfın nesnesiyle de **doğru** çalışmalıdır. Derlenmesi yetmez.
- Matematikte her kare bir dikdörtgendir. Ama bu `Dikdortgen` sınıfı "en ve boy birbirinden bağımsız değişir" sözünü verir; `Kare` bu sözü tutamaz.
- `genislet` hatalı değildir; hata, sözü bozan alt sınıftadır.
- Çözümler: `Kare`'yi `Dikdortgen`'den türetmemek (ikisi de `Sekil`'den türer) ya da sınıfları değiştirilemez yapmak.

is-a testi gerçek dünyaya değil, sınıfın **davranışına** uygulanır: "Kare, Dikdortgen'in her metoduna onun gibi cevap verebilir mi?"

Object Sınıfı

- Her sınıf `java.lang.Object` sınıfından türer; `extends` yazılmamışsa **dolaylı** olarak.
- Bu yüzden her nesnede aşağıdaki metotlar **hazır** bulunur.
- Varsayılan sürümleri genellikle işimize yaramaz; anlamlı sınıflarda geçersiz kılınırlar.

```
class Nokta { }
// derleyicinin gördüğü:
class Nokta extends Object { }
```

Metot	Varsayılan davranış
`String toString()`	Sınıf adı + `@` + hashCode değerinin onaltılık (hex) yazımı
`boolean equals(Object o)`	Referansları karşılaştırır: `this == o`
`int hashCode()`	Nesneye özgü bir tam sayı (genellikle her nesnede farklı)
`Class getClass()`	Nesnenin çalışma zamanındaki sınıfı (final; geçersiz kılınamaz)

OBJECT SINIFI

`toString` Metodu

- `public String toString()` : nesneyi metin olarak gösterir. `println(nesne)` ve `"..." + nesne` onu **kendiliğinden** çağırır.
- Geçersiz kılınmazsa sınıf adı ve nesnenin hashCode değerini içeren bir dizgi döner.

GEÇERSİZ KILINMADAN

```
class Nokta {
    private int x, y;
    Nokta(int x, int y) {
        this.x = x; this.y = y;
    }
}
// main içinde
System.out.println(new Nokta(3, 4));
```

```
Nokta@28a418fc
```

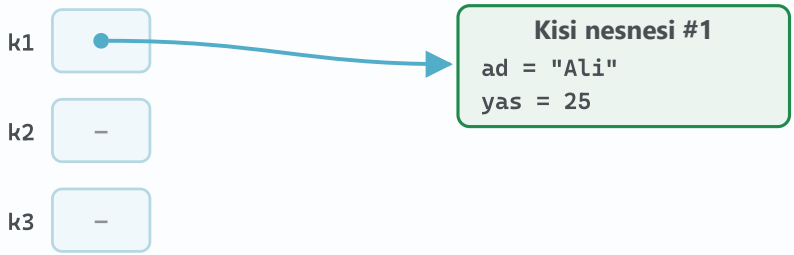
@ sonrası değer her çalıştırmada değişebilir; okunaklı değildir. `null` bir referans yazdırılırsa `toString` çağrılmaz, `null` yazılır.

GEÇERSİZ KILINARAK

```
class Nokta {
    private int x, y;
    Nokta(int x, int y) {
        this.x = x; this.y = y;
    }
    @Override
    public String toString() {
        return "Nokta[x=" + x + ", y=" + y + "]";
    }
}
// main içinde
Nokta n = new Nokta(3, 4);
System.out.println(n);
System.out.println("Konum: " + n);
```

```
Nokta[x=3, y=4]
Konum: Nokta[x=3, y=4]
```

== mi, equals mı?



`new` bir nesne oluşturur; `k1` onun **adresini** (referansını) tutar.

Adım 1 / 6

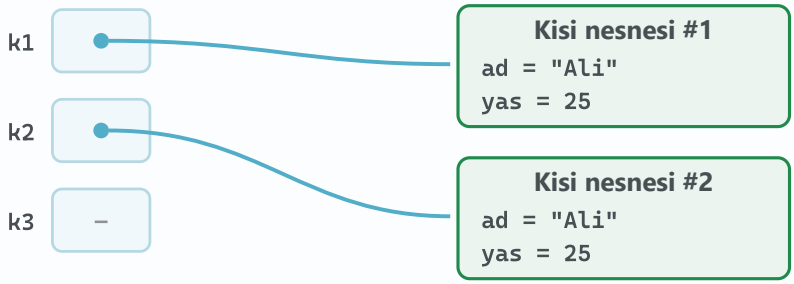
```
▶ Kisi k1 = new Kisi("Ali", 25);
2 Kisi k2 = new Kisi("Ali", 25);
3 Kisi k3 = k1;
4 System.out.println(k1 == k2);
5 System.out.println(k1 == k3);
6 System.out.println(k1.equals(k2));
```

DEĞİŞKENLER

k1 = →#1

ÇIKTI

== mi, equals mı?



İkinci `new` : içeriği aynı, ama **ayrı** bir nesne. `k2` onu gösterir.

Adım 2 / 6

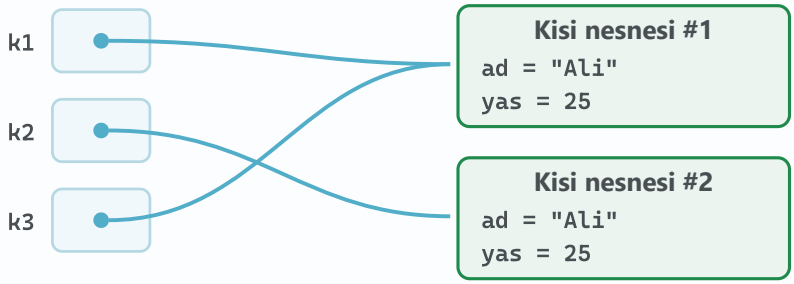
```
1 Kisi k1 = new Kisi("Ali", 25);
▶ Kisi k2 = new Kisi("Ali", 25);
3 Kisi k3 = k1;
4 System.out.println(k1 == k2);
5 System.out.println(k1 == k3);
6 System.out.println(k1.equals(k2));
```

DEĞİŞKENLER

k1 = →#1 | k2 = →#2

ÇIKTI

== mi, equals mı?



`k3 = k1` nesne kopyalamaz; yalnızca adresi kopyalar. k1 ve k3 **aynı** nesneyi gösterir.

Adım 3 / 6

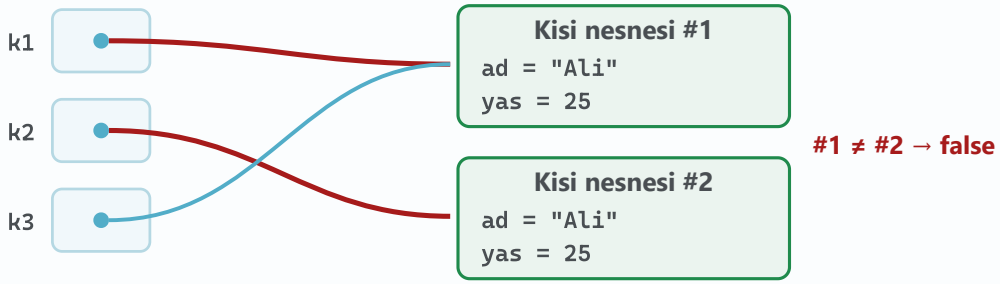
```
1 Kisi k1 = new Kisi("Ali", 25);
2 Kisi k2 = new Kisi("Ali", 25);
▶ Kisi k3 = k1;
4 System.out.println(k1 == k2);
5 System.out.println(k1 == k3);
6 System.out.println(k1.equals(k2));
```

DEĞİŞKENLER

k1 = →#1 | k2 = →#2 | k3 = →#1

ÇIKTI

== mi, equals mı?



`==` referansları karşılaştırır: k1 ve k2 farklı nesneleri gösterir → `false`.

Adım 4 / 6

```
1 Kisi k1 = new Kisi("Ali", 25);
2 Kisi k2 = new Kisi("Ali", 25);
3 Kisi k3 = k1;
▶ System.out.println(k1 == k2);
5 System.out.println(k1 == k3);
6 System.out.println(k1.equals(k2));
```

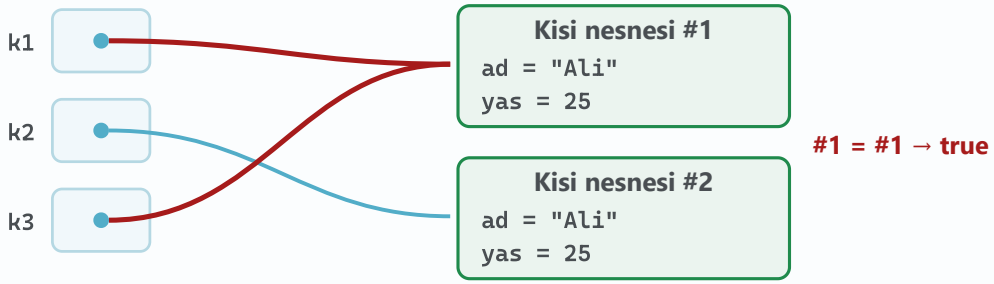
DEĞİŞKENLER

k1 = →#1 | k2 = →#2 | k3 = →#1

ÇIKTI

```
false
```

== mi, equals mı?



k1 ve k3 aynı nesneyi gösterir → `true`.

Adım 5 / 6

```
1 Kisi k1 = new Kisi("Ali", 25);
2 Kisi k2 = new Kisi("Ali", 25);
3 Kisi k3 = k1;
4 System.out.println(k1 == k2);
▶ System.out.println(k1 == k3);
6 System.out.println(k1.equals(k2));
```

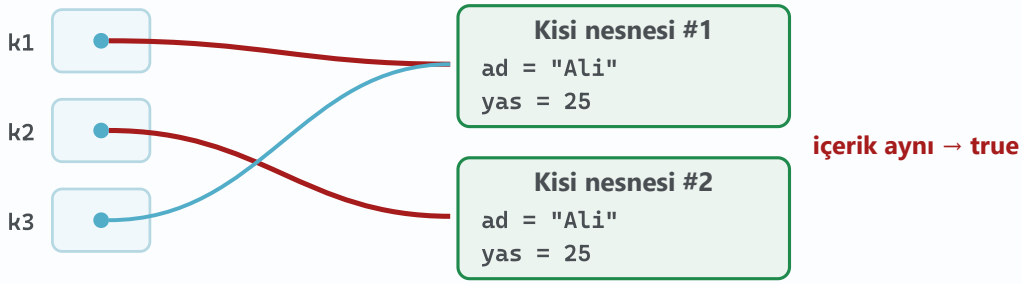
DEĞİŞKENLER

k1 = →#1 | k2 = →#2 | k3 = →#1

ÇIKTI

```
false
true
```

`==` mi, `equals` mı?



`equals`, Kisi'de geçersiz kılındığı için **içeriği** (ad, yas) karşılaştırır → `true`. Geçersiz kılınmasaydı `false` olurdu.

Adım 6 / 6

```
1 Kisi k1 = new Kisi("Ali", 25);
2 Kisi k2 = new Kisi("Ali", 25);
3 Kisi k3 = k1;
4 System.out.println(k1 == k2);
5 System.out.println(k1 == k3);
▶ System.out.println(k1.equals(k2));
```

DEĞİŞKENLER

`k1 = →#1` `k2 = →#2` `k3 = →#1`

ÇIKTI

```
false
true
true
```

OBJECT SINIFI

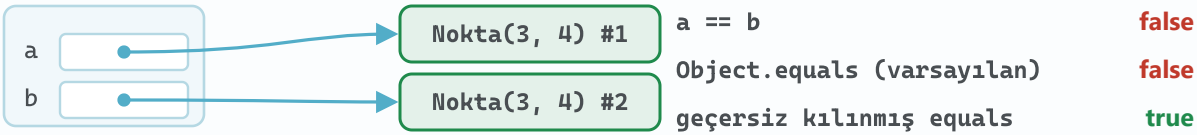
`equals` Metodu

- `public boolean equals(Object other)` : iki nesnenin **eşdeğer** olup olmadığını söyler.
- Varsayılan olarak yalnızca referansları karşılaştırır: geçersiz kılınmazsa `==` işleci gibi davranır.
- Geçersiz kılınarak davranışı değiştirilir: metot iki nesnenin **durumunu** (alanlarını) karşılaştırmalı ve eşitse `true` dönmelidir.
- Parametre `Object` tipindedir; hatalı durumları engellemek için önce **tip kontrolü** yapılmalıdır.

EQUALS SÖZLEŞMESİ

Yansıma	`x.equals(x)` → true
Simetri	`x.equals(y)` ⇔ `y.equals(x)`
Geçişme	x = y ve y = z ise x = z
Tutarlılık	Alanlar değişmedikçe sonuç değişmez
null	`x.equals(null)` → false

`String` karşılaştırmasında da kural aynıdır: `s1 == s2` değil, `s1.equals(s2)` .



`equals` Metodunu Yazmak

`instanceof` İLE TİP KONTROLÜ

```
public boolean equals(Object nesne) {
    if (nesne instanceof Nokta) {
        // nesne bir Nokta nesnesidir;
        // dönüştürüp karşılaştır
        Nokta diger = (Nokta) nesne;
        return x == diger.x && y == diger.y;
    } else {
        // nesne bir Nokta değildir;
        // eşit olamaz
        return false;
    }
}
```

- Parametre tipi `Object` olmalıdır; `equals(Nokta n)` yazmak geçersiz kılma değil, **aşırı yüklemedir**.
- Dönüşüm (`(Nokta) nesne`) yalnızca tip kontrolünden sonra güvenlidir.

DENEME

```
Nokta a = new Nokta(3, 4);
Nokta b = new Nokta(3, 4);
System.out.println(a == b);
System.out.println(a.equals(b));
System.out.println(a.equals("3,4"));
System.out.println(a.equals(null));
```

```
false
true
false
false
```

`"3,4"` bir Nokta değildir → `false` . `null` `instanceof Nokta` da `false` olduğu için ayrı bir null kontrolüne gerek kalmaz.

`getClass()` ile `equals`

```java
public class Kisi {
    private String ad;
    private int yas;
    // ... yapıcı ...

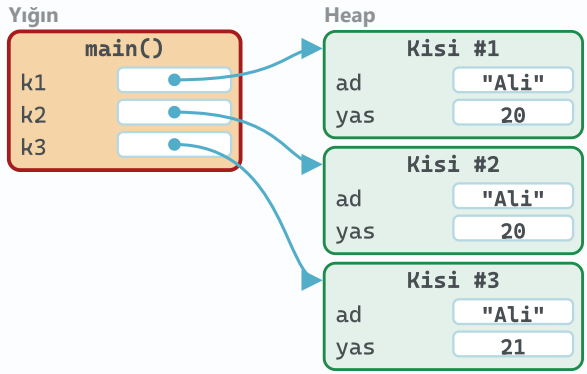
    @Override
    public boolean equals(Object obj) {
        // aynı referanssa eşittir
        if (this == obj) return true;
        // null ya da farklı sınıf: eşit değil
        if (obj == null
                || getClass() != obj.getClass())
            return false;
        // alanları karşılaştır
        Kisi kisi = (Kisi) obj;
        return yas == kisi.yas
               && ad.equals(kisi.ad);
    }
}
```

- `this == obj` : aynı nesneyse hemen `true` (hızlı yol).
- `getClass()` nesnenin **gerçek** sınıfını verir; iki tarafın sınıfı birebir aynı olmalıdır.
- İlkel tipler (`int yas`) `==` ile, nesne tipindeki alanlar (`String ad`) `equals` ile karşılaştırılır.

Tip kontrolü	Alt sınıf nesnesi eşit sayılabilir mi?
`instanceof`	Evet (Kedi nesnesi bir Hayvan'dır)
`getClass()`	Hayır (sınıflar birebir aynı olmalı)

`equals()`: Kod İzleme

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
3      if (obj == null || getClass() != obj.getClass())
4          return false;
5      Kisi kisi = (Kisi) obj;
6      return yas == kisi.yas && ad.equals(kisi.ad);
7  }
8  // main içinde:
▶  Kisi k1 = new Kisi("Ali", 20);
▶  Kisi k2 = new Kisi("Ali", 20);
▶  Kisi k3 = new Kisi("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



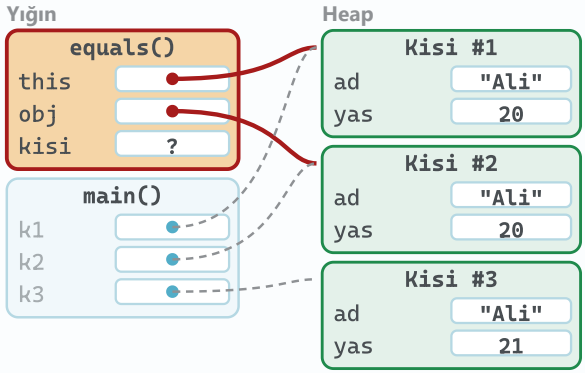
Üç ayrı nesne var. `k1` ile `k2` içerikçe aynı, `k3` yalnızca yaşta farklı.

Adım 1 / 9

ÇIKTI

`equals()`: Kod İzleme

```
▶ public boolean equals(Object obj) {
2     if (this == obj) return true;
3     if (obj == null || getClass() != obj.getClass())
4         return false;
5     Kisi kisi = (Kisi) obj;
6     return yas == kisi.yas && ad.equals(kisi.ad);
7 }
8 // main içinde:
9 Kisi k1 = new Kisi("Ali", 20);
10 Kisi k2 = new Kisi("Ali", 20);
11 Kisi k3 = new Kisi("Ali", 21);
▶ System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



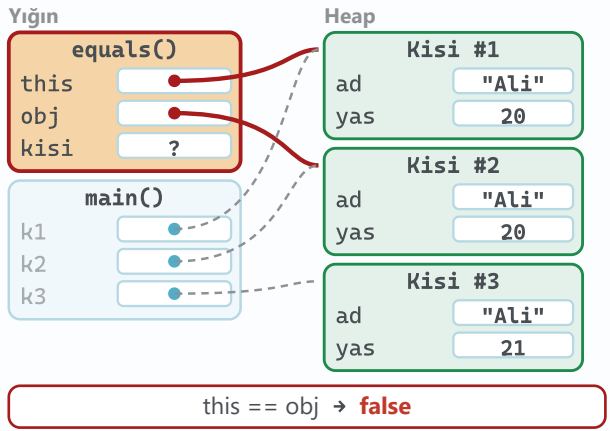
`k1.equals(k2)` : yeni çerçevede `this` = #1, `obj` = #2. `obj` 'nin tipi `Object` .

Adım 2 / 9

ÇIKTI

`equals()`: Kod İzleme

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
3      if (obj == null || getClass() != obj.getClass())
4          return false;
5      Kisi kisi = (Kisi) obj;
6      return yas == kisi.yas && ad.equals(kisi.ad);
7  }
8  // main içinde:
9  Kisi k1 = new Kisi("Ali", 20);
10 Kisi k2 = new Kisi("Ali", 20);
11 Kisi k3 = new Kisi("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



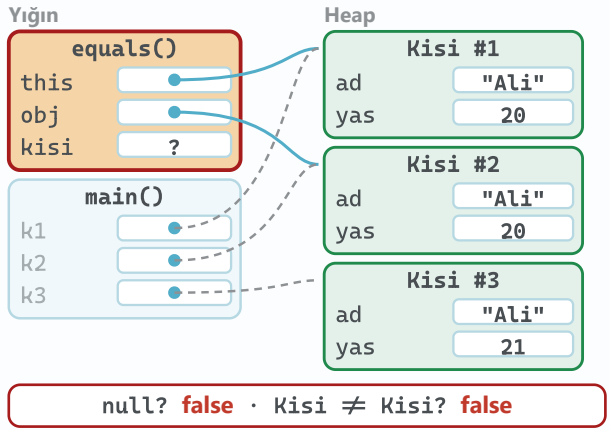
`this == obj` adresleri karşılaştırır: #1 ≠ #2, hızlı yol işlemez.

Adım 3 / 9

ÇIKTI

`equals()`: Kod İzleme

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
▶      if (obj == null || getClass() != obj.getClass())
4          return false;
5      Kisi kisi = (Kisi) obj;
6      return yas == kisi.yas && ad.equals(kisi.ad);
7  }
8  // main içinde:
9  Kisi k1 = new Kisi("Ali", 20);
10 Kisi k2 = new Kisi("Ali", 20);
11 Kisi k3 = new Kisi("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



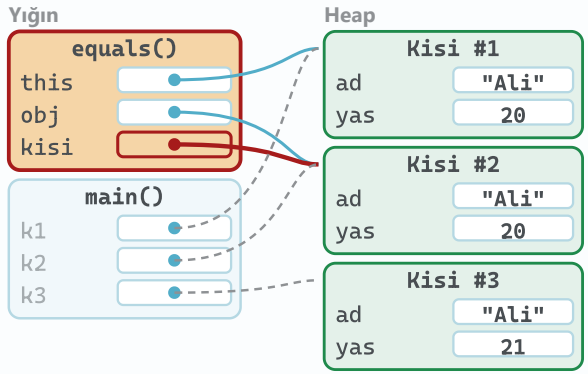
`obj` null değil; `getClass()` iki tarafta da `Kisi`. Koşul yanlış, `return false` atlanır.

Adım 4 / 9

ÇIKTI

`equals()`: Kod İzleme

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
3      if (obj == null || getClass() != obj.getClass())
4          return false;
▶      Kisi kisi = (Kisi) obj;
6      return yas == kisi.yas && ad.equals(kisi.ad);
7  }
8  // main içinde:
9  Kisi k1 = new Kisi("Ali", 20);
10 Kisi k2 = new Kisi("Ali", 20);
11 Kisi k3 = new Kisi("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



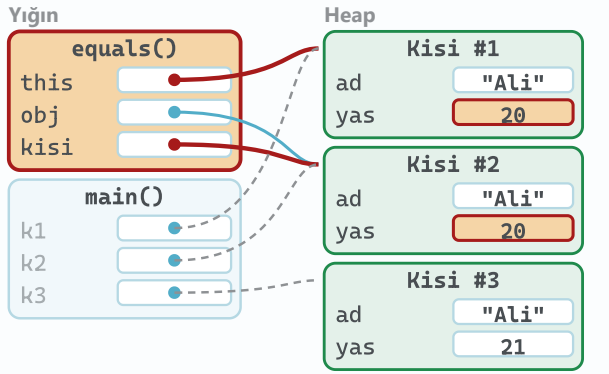
Aşağı dönüşüm yeni nesne üretmez: `kisi` aynı #2'yi, bu kez `Kisi` tipiyle gösterir; artık `kisi.yas` yazılabilir.

Adım 5 / 9

ÇIKTI

`equals()`: Kod İzleme

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
3      if (obj == null || getClass() != obj.getClass())
4          return false;
5      Kisi kisi = (Kisi) obj;
▶      return yas == kisi.yas && ad.equals(kisi.ad);
7  }
8  // main içinde:
9  Kisi k1 = new Kisi("Ali", 20);
10 Kisi k2 = new Kisi("Ali", 20);
11 Kisi k3 = new Kisi("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



`20 == 20` **true** `&&` `"Ali".equals("Ali")` **true**

İlkel `yas` `==` ile, `String` olan `ad` `equals` ile karşılaştırılır. İkisi de doğru: **true**.

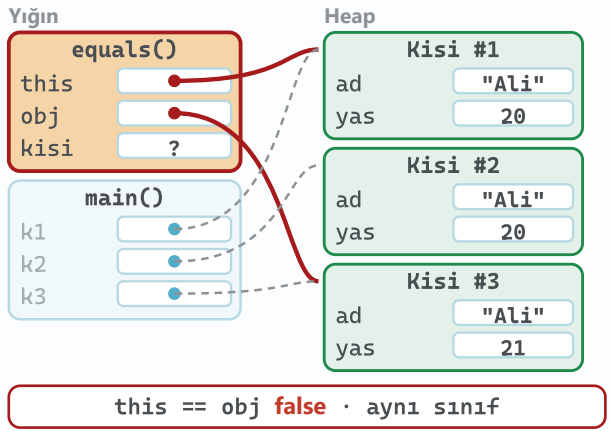
Adım 6 / 9

ÇIKTI

```
true
```

`equals()`: Kod İzleme

```
 1 public boolean equals(Object obj) {
 2     if (this == obj) return true;
 3     if (obj == null || getClass() != obj.getClass())
 4         return false;
 5     Kisi kisi = (Kisi) obj;
 6     return yas == kisi.yas && ad.equals(kisi.ad);
 7 }
 8 // main içinde:
 9 Kisi k1 = new Kisi("Ali", 20);
10 Kisi k2 = new Kisi("Ali", 20);
11 Kisi k3 = new Kisi("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



`k1.equals(k3)`: yeni çağrı, `obj` = #3. Farklı nesne, aynı sınıf: alanlara geçilir.

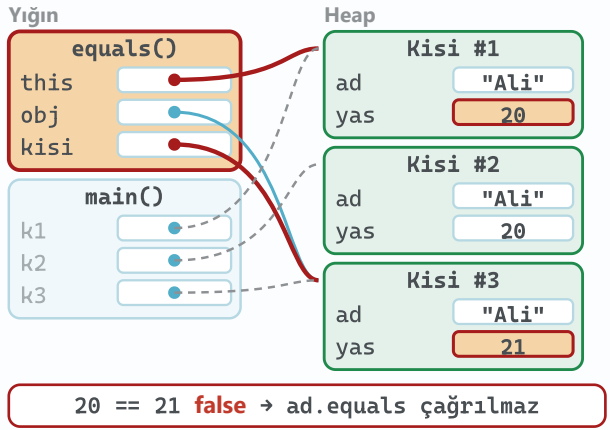
Adım 7 / 9

ÇIKTI

```
true
```

`equals()`: Kod İzleme

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
3      if (obj == null || getClass() != obj.getClass())
4          return false;
5      Kisi kisi = (Kisi) obj;
▶      return yas == kisi.yas && ad.equals(kisi.ad);
7  }
8  // main içinde:
9  Kisi k1 = new Kisi("Ali", 20);
10 Kisi k2 = new Kisi("Ali", 20);
11 Kisi k3 = new Kisi("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
14 System.out.println(k1.equals(null));
```



`20 == 21` **false** → `ad.equals` çağrılmaz

`20 == 21` yanlış; `&&` kısa devre yapar, `ad.equals` hiç çağrılmaz. Sonuç **false**.

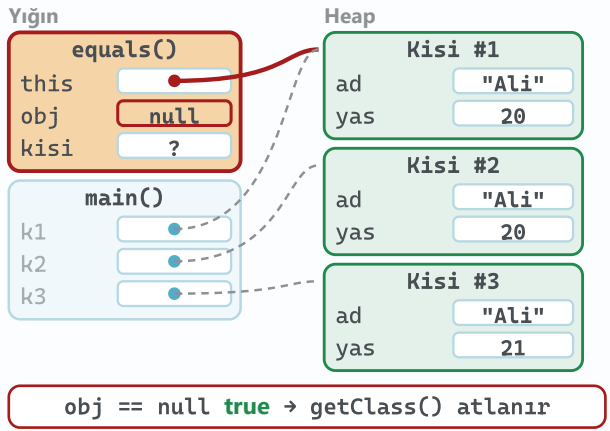
Adım 8 / 9

ÇIKTI

```
true
false
```

`equals()`: Kod İzleme

```
1  public boolean equals(Object obj) {
2      if (this == obj) return true;
▶      if (obj == null || getClass() != obj.getClass())
▶          return false;
5      Kisi kisi = (Kisi) obj;
6      return yas == kisi.yas && ad.equals(kisi.ad);
7  }
8  // main içinde:
9  Kisi k1 = new Kisi("Ali", 20);
10 Kisi k2 = new Kisi("Ali", 20);
11 Kisi k3 = new Kisi("Ali", 21);
12 System.out.println(k1.equals(k2));
13 System.out.println(k1.equals(k3));
▶  System.out.println(k1.equals(null));
```



`obj == null` doğru; `||` sağ tarafı değerlendirmez. `obj.getClass()` çalışsaydı `NullPointerException` olurdu.

Adım 9 / 9

ÇIKTI

```
true
false
false
```

OBJECT SINIFI

`equals` ve `hashCode`

Kural: `equals` geçersiz kılınıyorsa `hashCode` da geçersiz kılınmalıdır. Eşit nesnelerin hashCode değerleri **aynı** olmalıdır.

```
import java.util.Objects;
public class Nokta {
    private int x, y;
    public Nokta(int x, int y) {
        this.x = x; this.y = y;
    }
    @Override public boolean equals(Object o) {
        return (o instanceof Nokta n)
               && x == n.x && y == n.y;
    }
    @Override public int hashCode() {
        return Objects.hash(x, y);
    }
    @Override public String toString() {
        return "Nokta[x=" + x + ", y=" + y + "]";
    }
}
```

Neden? `HashSet` ve `HashMap` önce hashCode ile "kova" seçer, sonra `equals` çağırır. hashCode farklıysa eşit nesneler hiç karşılaştırılmaz.

```
HashSet<Nokta> kume = new HashSet<>();
kume.add(new Nokta(3, 4));
kume.add(new Nokta(3, 4));
System.out.println(kume.size() + " "
    + kume.contains(new Nokta(3, 4)));
```

Nokta sınıfında	Çıktı
yalnızca `equals`	`2 false` ✗
`equals` + `hashCode`	`1 true` ✓

`o instanceof Nokta n` örüntü eşlemesi Java 16+ özelliğidir; klasik biçim: `if (o instanceof Nokta) { Nokta n = (Nokta) o; … }`.

HashSet İçinde equals ve hashCode: Kod İzleme

```
 1 public boolean equals(Object o) {
 2     if (!(o instanceof Nokta)) return false;
 3     Nokta n = (Nokta) o;
 4     return x == n.x && y == n.y;
 5 }
 6 public int hashCode() {
 7     return Objects.hash(x, y);  // 31·(31 + x) + y
 8 }
 9 // main içinde:
▶ HashSet<Nokta> kume = new HashSet<>();
11 kume.add(new Nokta(3, 4));
12 kume.add(new Nokta(4, 3));
13 kume.add(new Nokta(3, 4));
14 System.out.println(kume.size());
15 System.out.println(kume.contains(new Nokta(4, 3)));
```

kume: 16 kova (kova = hash % 16)



Boş bir `HashSet` : öğeler 16 kovadan birine konur. Kovayı `hashCode()` seçer.

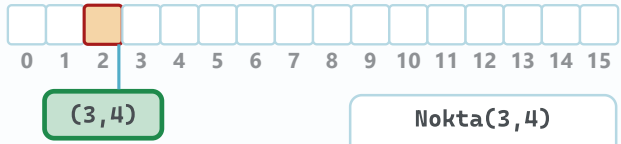
Adım 1 / 7

ÇIKTI

HashSet İçinde equals ve hashCode: Kod İzleme

```
1  public boolean equals(Object o) {
2      if (!(o instanceof Nokta)) return false;
3      Nokta n = (Nokta) o;
4      return x == n.x && y == n.y;
5  }
▶ public int hashCode() {
▶      return Objects.hash(x, y);  // 31·(31 + x) + y
8  }
9  // main içinde:
10 HashSet<Nokta> kume = new HashSet<>();
▶ kume.add(new Nokta(3, 4));
12 kume.add(new Nokta(4, 3));
13 kume.add(new Nokta(3, 4));
14 System.out.println(kume.size());
15 System.out.println(kume.contains(new Nokta(4, 3)));
```

kume: 16 kova (kova = hash % 16)



`1058 % 16 = 2 → kova boş, eklenir`

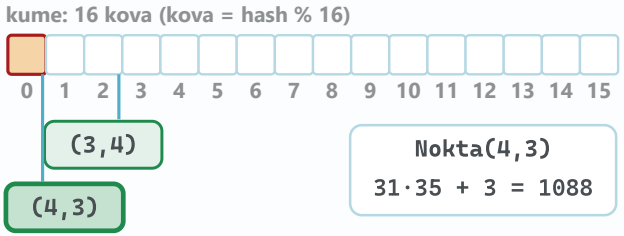
`hashCode()` = 31·(31+3)+4 = **1058**; 1058 % 16 = 2. Kova boş: nesne eklenir.

Adım 2 / 7

ÇIKTI

HashSet İçinde equals ve hashCode: Kod İzleme

```
1  public boolean equals(Object o) {
2      if (!(o instanceof Nokta)) return false;
3      Nokta n = (Nokta) o;
4      return x == n.x && y == n.y;
5  }
6  public int hashCode() {
▶      return Objects.hash(x, y);  // 31·(31 + x) + y
8  }
9  // main içinde:
10 HashSet<Nokta> kume = new HashSet<>();
11 kume.add(new Nokta(3, 4));
▶ kume.add(new Nokta(4, 3));
13 kume.add(new Nokta(3, 4));
14 System.out.println(kume.size());
15 System.out.println(kume.contains(new Nokta(4, 3)));
```



`1088 % 16 = 0 → kova boş, eklenir`

Aynı sayılar, farklı sıra: hash **1088**, kova 0. Yine boş kova, eklenir.

Adım 3 / 7

ÇIKTI

HashSet İçinde equals ve hashCode: Kod İzleme

```
1  public boolean equals(Object o) {
2      if (!(o instanceof Nokta)) return false;
3      Nokta n = (Nokta) o;
4      return x == n.x && y == n.y;
5  }
6  public int hashCode() {
▶      return Objects.hash(x, y);  // 31·(31 + x) + y
8  }
9  // main içinde:
10 HashSet<Nokta> kume = new HashSet<>();
11 kume.add(new Nokta(3, 4));
12 kume.add(new Nokta(4, 3));
▶  kume.add(new Nokta(3, 4));
14 System.out.println(kume.size());
15 System.out.println(kume.contains(new Nokta(4, 3)));
```



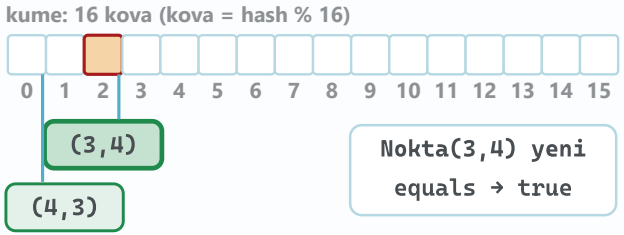
Yeni bir `Nokta(3, 4)` nesnesi: hash yine 1058, aynı kova 2. Kova dolu: içindekiyle `equals` sorulur.

Adım 4 / 7

ÇIKTI

HashSet İçinde equals ve hashCode: Kod İzleme

```
▶ public boolean equals(Object o) {
2     if (!(o instanceof Nokta)) return false;
3     Nokta n = (Nokta) o;
▶     return x == n.x && y == n.y;
5 }
6 public int hashCode() {
7     return Objects.hash(x, y);  // 31·(31 + x) + y
8 }
9 // main içinde:
10 HashSet<Nokta> kume = new HashSet<>();
11 kume.add(new Nokta(3, 4));
12 kume.add(new Nokta(4, 3));
13 kume.add(new Nokta(3, 4));
14 System.out.println(kume.size());
15 System.out.println(kume.contains(new Nokta(4, 3)));
```



`3 == 3 && 4 == 4 → true: eklenmez`

`equals` doğru: küme aynı öğeyi ikinci kez almaz, `add` `false` döndürür.

Adım 5 / 7

ÇIKTI

HashSet İçinde equals ve hashCode: Kod İzleme

```
1  public boolean equals(Object o) {
2      if (!(o instanceof Nokta)) return false;
3      Nokta n = (Nokta) o;
4      return x == n.x && y == n.y;
5  }
6  public int hashCode() {
7      return Objects.hash(x, y);  // 31·(31 + x) + y
8  }
9  // main içinde:
10 HashSet<Nokta> kume = new HashSet<>();
11 kume.add(new Nokta(3, 4));
12 kume.add(new Nokta(4, 3));
13 kume.add(new Nokta(3, 4));
▶  System.out.println(kume.size());
15 System.out.println(kume.contains(new Nokta(4, 3)));
```



Kümede iki öğe var: **2**.

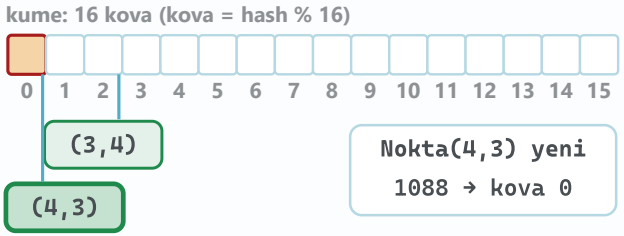
Adım 6 / 7

ÇIKTI

```
2
```

HashSet İçinde equals ve hashCode: Kod İzleme

```
1  public boolean equals(Object o) {
2      if (!(o instanceof Nokta)) return false;
3      Nokta n = (Nokta) o;
▶      return x == n.x && y == n.y;
5  }
6  public int hashCode() {
▶      return Objects.hash(x, y);  // 31·(31 + x) + y
8  }
9  // main içinde:
10 HashSet<Nokta> kume = new HashSet<>();
11 kume.add(new Nokta(3, 4));
12 kume.add(new Nokta(4, 3));
13 kume.add(new Nokta(3, 4));
14 System.out.println(kume.size());
▶  System.out.println(kume.contains(new Nokta(4, 3)));
```



`kova 0 → equals true → bulundu`

`hashCode yoksa: farklı kova, equals sorulmaz`

`contains` da önce kovayı bulur, sonra `equals` sorar: **true**. `hashCode` yazılmasaydı eşit nesneler farklı kovalara düşerdi.

Adım 7 / 7

ÇIKTI

```
2
true
```

ÖZET

Özet

Kavram	Özü
Kalıtım	`class Alt extends Ust`; alt sınıf üst sınıfın üyelerini kalıtımla alır, yenilerini ekler (is-a)
Kalıtımla alınmayan	Yapıcılar; `private` üyeler nesnede vardır ama doğrudan erişilemez
`super`	`super(...)`: üst sınıf yapıcısı (ilk satır); `super.m()`: üst sınıfın sürümü
Yapıcı zinciri	Object → … → Ust → Alt sırasıyla tamamlanır
Geçersiz kılma	Aynı imza, alt sınıfta yeni gövde; `@Override` ile derleyici denetler
`protected`	Sınıf + alt sınıflar + aynı paket
`final`	Metot: geçersiz kılınamaz; sınıf: alt sınıfı olamaz
Has-a	Oluşum: başka sınıf tipinde alan; çoklu kalıtım yerine oluşum ve arayüzler
Object	Her sınıfın atası; `toString`, `equals`, `hashCode` geçersiz kılınır
`==` / `equals`	Referans karşılaştırma / içerik karşılaştırma

ÖZET

Sık Yapılan Hatalar

super(...) çağırmamak

Üst sınıfta parametresiz yapıcı yoksa: `constructor Hayvan in class Hayvan cannot be applied to given types`.

private alana doğrudan erişmek

`saglik += 5;` alt sınıfta: `saglik has private access in Hayvan`. Çözüm: getter ya da protected metot.

equals(Nokta) yazmak

Parametre `Object` değilse aşırı yüklemedir; `@Override` ile: `method does not override or implement a method from a supertype`.

Nesneleri == ile karşılaştırmak

İçerikleri aynı iki nesne için `==` → `false`. `String` dahil, içerik `equals` ile karşılaştırılır.

equals var, hashCode yok

Derleme hatası yoktur ama `HashSet` / `HashMap` yanlış çalışır: aynı nokta iki kez eklenir.

İki sınıftan türetmeye çalışmak

`class D extends B, C` → `'{' expected`. Çözüm: oluşum ya da arayüz.

Erişimi daraltmak

`public void ciz()` alt sınıfta `void ciz()` olursa: `attempting to assign weaker access privileges`.

Kontrolsüz aşağı dönüşüm

`(Lastik) p`, p bir Lastik değilse: `ClassCastException`. Önce `instanceof`.

ÖZET

Sık Yapılan Hatalar (2)

this(...) ile super(...) birlikte

İkisi de ilk satır olmak ister: `call to super must be first statement in constructor`. Zincirin sonundaki yapıcı `super(...)` çağırır.

Dönüş tipini değiştirmek

`String yavrula()` : `return type String is not compatible with Hayvan`. Yalnızca alt tip (kovaryant) olabilir.

private metodu geçersiz kıldığını sanmak

Hata vermez ama üst sınıfın kodu yine kendi `private` sürümünü çağırır. `@Override` bu yanılgıyı yakalar.

protected üyeye üst tip referansıyla erişmek

Başka paketteki alt sınıfta `h.isim` : `isim has protected access in Hayvan`. Erişim alt sınıfın kendi tipi üzerinden yapılır.

İstisna listesini genişletmek

`void f() throws Exception`, üstte `throws` yoksa: `overridden method does not throw Exception`.

Alan sırasını yanlış sanmak

Alt sınıfın alan ilk değerleri, üst sınıf yapıcısı **bittikten sonra** atanır; üst yapıcı çalışırken hâlâ varsayılandır.

Kod paylaşmak için kalıtım

is-a doğru değilse alt sınıf, istenmeyen `public` metotları da alır: `Yigin extends ArrayList` ortadan eklemeye izin verir. Çözüm: oluşum.

Kendini çağıran metodu geçersiz kılmak

Üst sınıfın öteki metotları da alt sınıfın sürümünü çağırabilir: sayaç iki kez artar. Çözüm: üst sınıfın belgesine bakmak ya da oluşum.

ÖZET

Örnek Problemler

```
class A {
    A() { System.out.println("A"); }
    void yaz() { System.out.println("A.yaz"); }
}
class B extends A {
    B() { System.out.println("B"); }
    @Override void yaz() { System.out.println("B.yaz"); }
}
class C extends B {
    C() { super(); System.out.println("C"); }
    @Override void yaz() {
        super.yaz();
        System.out.println("C.yaz");
    }
}
```

1. `C c = new C();` ne yazdırır?
2. Ardından `c.yaz();` ne yazdırır?
3. `A a = new B(); a.yaz();` ne yazdırır?
4. `c instanceof A` nedir?
5. `equals`'ı geçersiz kılmayan bir `Ogrenci` sınıfında, aynı adlı iki nesne için `o1.equals(o2)` nedir?

▶ **Cevapları göster**

ÖZET

Örnek Problemler (2)

```
class A {
    static { System.out.print("As "); }
    { System.out.print("Ab "); }
    A() { System.out.print("A "); }
}
class B extends A {
    static { System.out.print("Bs "); }
    { System.out.print("Bb "); }
    B() { this(0); System.out.print("B "); }
    B(int x) { System.out.print("B" + x + " "); }
}
```

1. Programdaki ilk `new B();` ne yazdırır? (Süslü parantezli satırlar örnek ilklendirici bloklarıdır.)
2. Ardından `new B(7);` ne yazdırır?
3. Yerine koyma örneğindeki `Kare` ile `Dikdortgen d = new Kare(); d.setBoy(4); d.setEn(5);` sonrasında `d.alan()` kaçtır?
4. `Ust` 'te `protected Object kopya()` var. `Alt extends Ust` içinde `@Override public Alt kopya()` derlenir mi?

▶ **Cevapları göster**

ÖZET

Alıştırmalar

1. `Calisan` (`ad` , `maas`) ve ondan türeyen `Yonetici` (`prim`) yazın; `toplamGelir()` ve `super.toString()` kullanan bir `toString()` ekleyin.
2. `Hayvan` 'ın tek yapıcısı `Hayvan(String isim)` . `Kus(String isim, boolean ucabilir)` yapıcısını yazın.
3. `Ogrenci` (`int no` , `String ad`) için `equals` ve `hashCode` yazın.
4. Başka paketteki, alt sınıf **olmayan** bir sınıf `protected` bir alana erişebilir mi?
5. is-a mı, has-a mı? Öğrenci–Kişi, Araba–Tekerlek, Kütüphane–Kitap, Kare–Şekil.

▶ **Cevaplar / ipuçları**

ÖZET

Kaynaklar

- Oracle, *The Java Tutorials*: Inheritance; Overriding and Hiding Methods; Using the Keyword super; Object as a Superclass.
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21 Edition (8.1.4 Superclasses and Subclasses, 8.4.8 Inheritance, Overriding, and Hiding).
- P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (Bölüm 9: Object-Oriented Programming: Inheritance).
- K. Sierra ve B. Bates, *Head First Java*, 3. baskı, O'Reilly (Bölüm 7: Inheritance and Polymorphism).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson (Bölüm 5: Inheritance).
- J. Bloch, *Effective Java*, 3. baskı, Addison-Wesley (Madde 10–11: equals ve hashCode; Madde 18: oluşumu kalıtıma tercih edin).

Son

Bölüm 4.3: Kalıtım

Sonraki bölüm: Bölüm 4.4: Çok Biçimlilik · Terim sözlüğü