Bölüm 4.4: Çok Biçimlilik

Java ile Nesne Yönelimli Programlama

Çok biçimlilik kavramı | Referans kuralı | Yukarı ve aşağı dönüşüm | instanceof | Dinamik bağlama | Metot tablosu | Aşırı yükleme | Şekil ve tanksavar örnekleri | equals ve simetri

Sercan KÜLCÜ · Giresun Üniversitesi · Elektrik-Elektronik Mühendisliği

GİRİŞ

Öğrenme Çıktıları

Kavram
Çok biçimliliği (polymorphism) tanımlamak; derleme zamanı ve çalışma zamanı çok biçimliliğini ayırt etmek.

Referans kuralı
Üst sınıf referansına alt sınıf nesnesi atanabildiğini, tersinin neden derleme hatası verdiğini açıklamak.

Tip dönüşümü
Yukarı ve aşağı dönüşümü (upcasting, downcasting) yazmak; `ClassCastException`'ı `instanceof` ile önlemek.

Dinamik bağlama
Çağrılan metodun nesnenin **gerçek sınıfına** göre seçildiğini adım adım izlemek; metot aramasını hiyerarşide göstermek.

Aşırı yükleme ve geçersiz kılma
Overloading ile overriding'i karşılaştırmak; hangisinin derleyici, hangisinin JVM tarafından seçildiğini bilmek.

Tasarım
Şekil dizisi ve tanksavar oyunu örnekleriyle üst sınıf tipiyle genel kod yazmak; `equals`'ın simetri tuzağını tanımak.

ÇOK BİÇİMLİLİK KAVRAMI

Çok Biçimlilik (Polymorphism)

- Yunanca **poly** (çok) + **morph** (biçim): aynı ifadenin, farklı nesnelerde **farklı davranışa** yol açmasıdır.
- Bir referans, kendi tipindeki **veya bir alt sınıfa ait** bir nesneye atıfta bulunabilir.
- `hayvan.sesCikar()` çağrısı köpek nesnesinde `Hav hav!`, kedi nesnesinde `Miyav!` yazdırır.
- Kodu **üst sınıf tipiyle** bir kez yazarız; her nesne çağrıya **kendi sürümüyle** cevap verir.

Nesne yönelimli programlamanın dört temel ilkesi: **sarmalama**, **kalıtım**, **çok biçimlilik** ve **soyutlama**. Çok biçimlilik, kalıtımın üzerine kurulur.

Uzaktan kumanda

Aynı "aç" düğmesi televizyonu, klimayı ve projektörü açar; her cihaz bunu kendi bildiği gibi yapar.

Sürücü belgesi

B sınıfı belgeyle her marka otomobil sürülür: sürücü "otomobil" arayüzünü kullanır, motorun ayrıntısı markaya aittir.

Hayvanat bahçesi

Bakıcı "herkes ses çıkarsın" der; her hayvan kendi sesini çıkarır.

ÇOK BİÇİMLİLİK KAVRAMI

Çok Biçimliliğin İki Türü

	Derleme zamanı (statik)	Çalışma zamanı (dinamik)
Mekanizma	Aşırı yükleme (overloading)	Metot geçersiz kılma (overriding) + dinamik bağlama
Seçimi yapan	Derleyici (`javac`)	JVM, program çalışırken
Neye bakılır?	Argümanların sayısı ve **bildirilen** tipleri	Nesnenin **gerçek** sınıfı
Kalıtım gerekir mi?	Hayır, tek sınıfta olur	Evet, sınıf hiyerarşisi gerekir

AYNI AD, FARKLI PARAMETRELER

```
int topla(int a, int b) { ... }
double topla(double a, double b) { ... }

topla(5, 3);     // int sürümü
topla(3.5, 2.7); // double sürümü
```

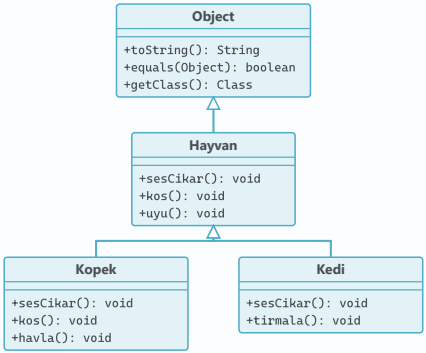
AYNI İMZA, FARKLI SINIFLAR

```
Hayvan h1 = new Kopek();
Hayvan h2 = new Kedi();

h1.sesCikar(); // Hav hav!
h2.sesCikar(); // Miyav!
```

"Çok biçimlilik" denince çoğunlukla çalışma zamanı çok biçimliliği kastedilir.

Örnek Sınıf Hiyerarşisi




```
class Hayvan {
  public void sesCikar() { System.out.println("Hayvan sesi"); }
  public void kos()      { System.out.println("Hayvan koşuyor"); }
  public void uyu()      { System.out.println("Hayvan uyuyor"); }
}
class Kopek extends Hayvan {
  @Override public void sesCikar() { System.out.println("Hav hav!"); }
  @Override public void kos() { System.out.println("Köpek koşuyor"); }
  public void havla() { System.out.println("Köpek havlıyor"); }
}
class Kedi extends Hayvan {
  @Override public void sesCikar() { System.out.println("Miyav!"); }
  public void tirmala() { System.out.println("Kedi tırmalıyor"); }
}
```

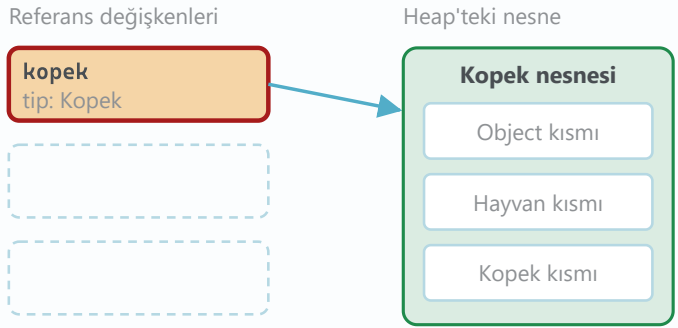
- `Kopek`, `sesCikar()` ve `kos()`'u **geçersiz kılar**, `havla()`'yı **ekler**; `Kedi` yalnızca `sesCikar()`'ı geçersiz kılar, `tirmala()`'yı ekler.
- Kopek de Kedi de **bir** Hayvan'dır (is-a); ikisi de `Hayvan` tipindeki bir değişkende tutulabilir.

Referans, Alt Sınıf Nesnesini Gösterebilir

```
1 // Köpek nesnesine referans
2 Kopek kopek = new Kopek();
3 // Hayvan referansı Kopek'i gösterebilir
4 Hayvan hayvan = kopek;
5 // Object referansı da gösterebilir
6 Object nesne = hayvan;
7 System.out.println(kopek == hayvan);
8 System.out.println(hayvan == nesne);
9 System.out.println(nesne.getClass());
```

ÇIKTI





`new Kopek()` heap'te **bir** nesne oluşturur; `kopek` referansı onu gösterir.

Adım 1 / 6

DEĞİŞKENLER

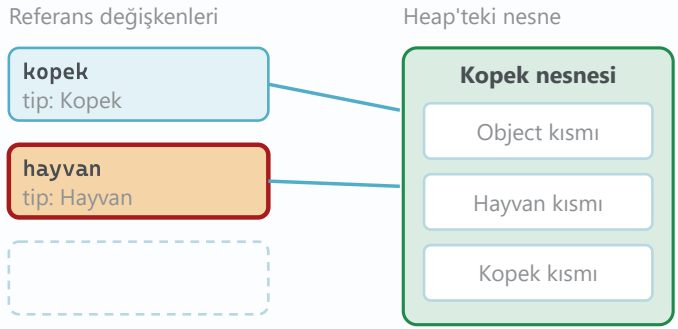
`kopek = → Kopek nesnesi`

Referans, Alt Sınıf Nesnesini Gösterebilir

```
1 // Köpek nesnesine referans
2 Kopek kopek = new Kopek();
3 // Hayvan referansı Kopek'i gösterebilir
▶ Hayvan hayvan = kopek;
5 // Object referansı da gösterebilir
6 Object nesne = hayvan;
7 System.out.println(kopek == hayvan);
8 System.out.println(hayvan == nesne);
9 System.out.println(nesne.getClass());
```

ÇIKTI





Yeni nesne **oluşmaz**: `hayvan`, aynı nesneyi gösteren ikinci referanstır. Kopek bir Hayvan olduğu için atama geçerlidir.

Adım 2 / 6

DEĞİŞKENLER

`kopek = → Kopek nesnesi`

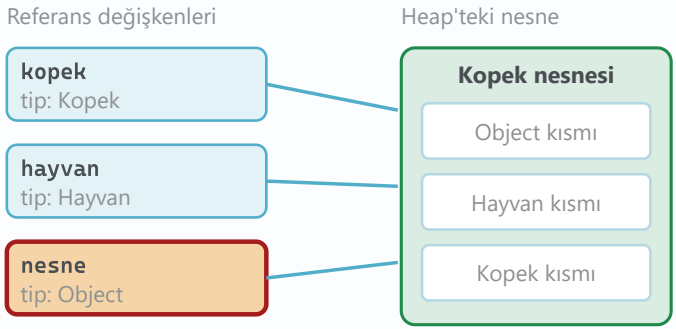
`hayvan = → aynı nesne`

Referans, Alt Sınıf Nesnesini Gösterebilir

```
1 // Köpek nesnesine referans
2 Kopek kopek = new Kopek();
3 // Hayvan referansı Kopek'i gösterebilir
4 Hayvan hayvan = kopek;
5 // Object referansı da gösterebilir
6 Object nesne = hayvan;
7 System.out.println(kopek == hayvan);
8 System.out.println(hayvan == nesne);
9 System.out.println(nesne.getClass());
```

ÇIKTI





Her sınıf `Object`'ten türediği için `Object` referansı da aynı nesneyi gösterebilir.

Adım 3 / 6

DEĞİŞKENLER

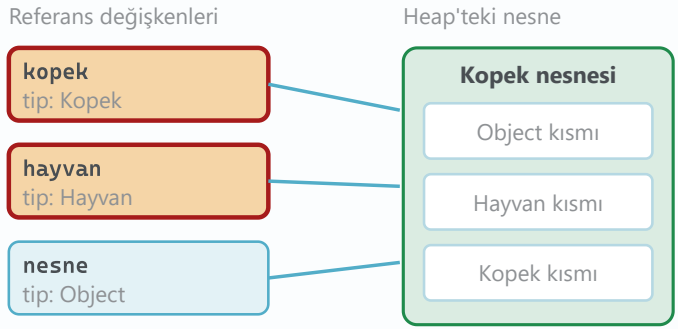
- kopek = → **Kopek nesnesi**
- hayvan = → **aynı nesne**
- nesne = → **aynı nesne**

Referans, Alt Sınıf Nesnesini Gösterebilir

```
1 // Köpek nesnesine referans
2 Kopek kopek = new Kopek();
3 // Hayvan referansı Kopek'i gösterebilir
4 Hayvan hayvan = kopek;
5 // Object referansı da gösterebilir
6 Object nesne = hayvan;
▶ System.out.println(kopek == hayvan);
8 System.out.println(hayvan == nesne);
9 System.out.println(nesne.getClass());
```

ÇIKTI

```
true
```



`==` referansları karşılaştırır: ikisi de **aynı** nesneyi gösterdiği için `true`.

Adım 4 / 6

DEĞİŞKENLER

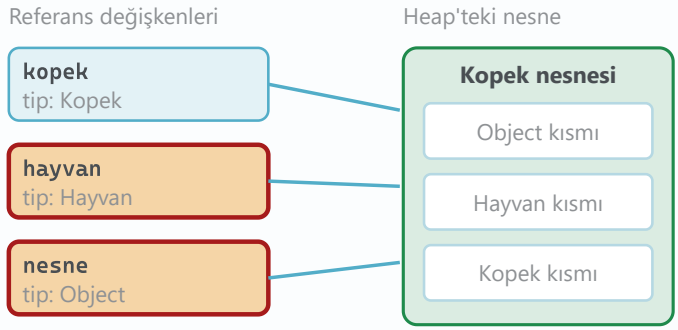
- kopek = → **Kopek nesnesi**
- hayvan = → **aynı nesne**
- nesne = → **aynı nesne**

Referans, Alt Sınıf Nesnesini Gösterebilir

```
1 // Köpek nesnesine referans
2 Kopek kopek = new Kopek();
3 // Hayvan referansı Kopek'i gösterebilir
4 Hayvan hayvan = kopek;
5 // Object referansı da gösterebilir
6 Object nesne = hayvan;
7 System.out.println(kopek == hayvan);
▶ System.out.println(hayvan == nesne);
9 System.out.println(nesne.getClass());
```

ÇIKTI

```
true
true
```



Yine aynı nesne: `true`. Üç referans, tek nesne.

Adım 5 / 6

DEĞİŞKENLER

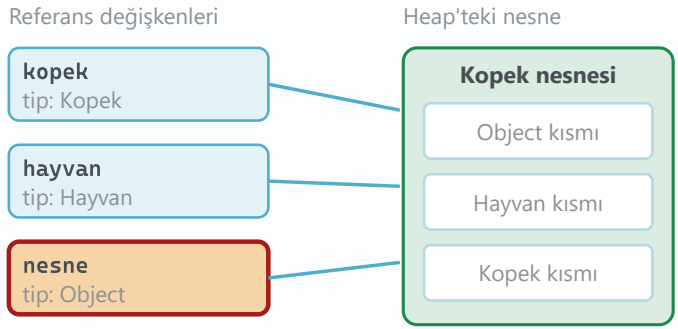
- kopek = → **Kopek nesnesi**
- hayvan = → **aynı nesne**
- nesne = → **aynı nesne**

Referans, Alt Sınıf Nesnesini Gösterebilir

```
1 // Köpek nesnesine referans
2 Kopek kopek = new Kopek();
3 // Hayvan referansı Kopek'i gösterebilir
4 Hayvan hayvan = kopek;
5 // Object referansı da gösterebilir
6 Object nesne = hayvan;
7 System.out.println(kopek == hayvan);
8 System.out.println(hayvan == nesne);
▶ System.out.println(nesne.getClass());
```

ÇIKTI

```
true
true
class Kopek
```



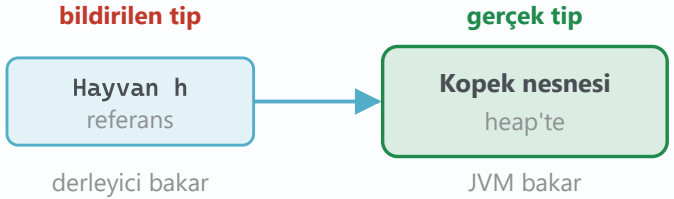
Referansın tipi `Object` olsa da nesnenin **gerçek sınıfı** değişmez: `Kopek`.

Adım 6 / 6

DEĞİŞKENLER

- kopek = → **Kopek nesnesi**
- hayvan = → **aynı nesne**
- nesne = → **aynı nesne**

Referans Tipi ve Nesne Tipi



Soru	Kim karar verir?
Bu metot **çağrılabilir mi?**	Derleyici, **referans tipine** bakar
Hangi sürüm çalışır?	JVM, **nesne tipine** bakar
Hangi atamalar geçerli?	Derleyici, referans tipine bakar

```
Hayvan h = new Kopek();
// bildirilen tür: Hayvan
// gerçek tür:     Kopek
```

```
Hayvan hayvan = new Kopek();
hayvan.havla(); // derleme hatası
```

```
error: cannot find symbol
  symbol:   method havla()
  location: variable hayvan of type Hayvan
```

Kumandada düğmesi olmayan özellik kullanılamaz: nesne bir köpek olsa da `Hayvan` "kumandasında" `havla()` düğmesi yoktur.

Referans Kuralı

- Bir nesnenin **alt tipini**, üst sınıf tipindeki bir referansa atayabiliriz.
- Ancak üst sınıfın nesnesini, alt sınıf tipindeki bir referansa **atayamayız**.

Geçerli

```
// Kopek, Hayvan'ın alt türüdür
Hayvan hayvan = new Kopek();
Object nesne = new Kedi();
```

Her köpek bir hayvandır: hayvandan beklenen her şeyi köpek de yapabilir.

Hatalı

```
// Hayvan, Kopek'in üst sınıfıdır
Kopek kopek = new Hayvan();
```

```
error: incompatible types:
  Hayvan cannot be converted to Kopek
```

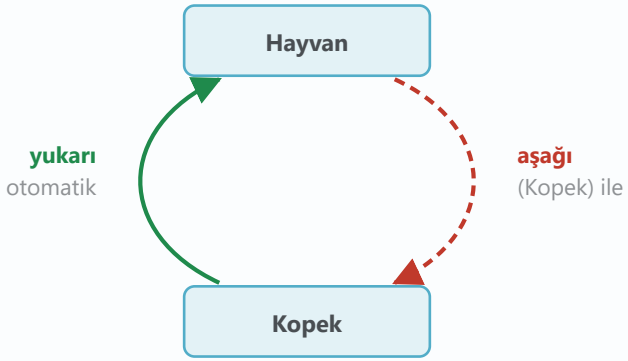
Bir hayvan `havla()` bilmeyebilir; derleyici buna izin vermez.

`Hayvan h = new Kopek(); Kopek k = h;` de aynı hatayı verir. Derleyici `h` 'nin gösterdiği nesneyi değil, **bildirilen tipini** (`Hayvan`) görür; çözüm açık dönüşümdür.



TİP DÖNÜŞÜMÜ

Yukarı ve Aşağı Dönüşüm



	Yukarı (upcasting)	Aşağı (downcasting)
Yön	Alt → üst	Üst → alt
Yazım	Otomatik (örtük)	Açık: `(Kopek) h`
Güvenli mi?	**Her zaman**	**Nesneye bağlı**
Diğer adı	Genişletme (widening)	Daraltma (narrowing)

```
Kopek kopek = new Kopek();
Hayvan h = kopek;     // yukarı: otomatik
Hayvan h1 = (Hayvan) kopek; // gereksiz
```

```
Hayvan h = new Kopek();
Kopek k = (Kopek) h; // aşağı: açık
k.havla();           // Köpek havlıyor
```

TİP DÖNÜŞÜMÜ

Aşağı Dönüşüm (Downcasting)

- Bir üst sınıf referansı, alt sınıf tipine **açıkça** (explicitly) dönüştürülebilir: `(AltSinif) referans`.
- Üst sınıf referansı, ancak bu dönüşümle bir alt sınıf değişkenine atanabilir.
- Amaç: üst sınıfta **bulunmayan** (yalnızca alt sınıfa ait) metotları çağırmak.
- Dönüşüm **nesneyi değiştirmez**; yalnızca derleyiciye "bu referansı Kopek olarak gör" der.

```
Hayvan h2 = new Kopek();
Kopek k = (Kopek) h2;
k.havla();
((Kopek) h2).havla(); // tek satırda
```

```
Köpek havlıyor
Köpek havlıyor
```

ÖNCELİK TUZAĞI

Nokta (`.`) dönüşümden önce uygulanır. Parantezler unutulursa `(Kopek) havla()` çağrısının **sonucuna** uygulanmak istenir:

```
(Kopek) h2.havla();   // hatalı
```

```
error: not a statement
```

```
String s = (Kopek) h2.isim();
// h2.isim() önce aranır → Hayvan'da yok
```

```
error: cannot find symbol
  symbol:   method isim()
  location: variable h2 of type Hayvan
```

Doğrusu: `((Kopek) h2).havla();`

Hatalı Aşağı Dönüşüm: `ClassCastException`

```
Hayvan h4 = new Kedi();
Kopek k4 = (Kopek) h4; // derlenir!
k4.havla();
```

```
Exception in thread "main"
java.lang.ClassCastException:
class Kedi cannot be cast to class Kopek
```

```
Kopek k5 = (Kopek) new Hayvan(); // derlenir!
```

```
java.lang.ClassCastException:
class Hayvan cannot be cast to class Kopek
```

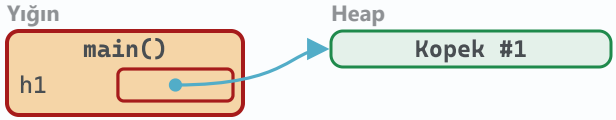
- **Derleyici** izin verir: `Hayvan` tipindeki bir referans pekâlâ bir `Kopek`'i gösteriyor olabilir.
- **JVM**, dönüşüm anında nesnenin gerçek sınıfını denetler: nesne bir `Kedi` ya da düz bir `Hayvan` ise program istisnayla durur.
- Yani üst sınıf **nesnesi**, alt sınıf referansına hiçbir zaman atanamaz; dönüşüm yalnızca nesne gerçekten o alt sınıftansa başarılı olur.

`B objB = (B) objA;` derlenir, ama yalnızca `objA` gerçekten bir `B` nesnesini gösteriyorsa çalışır. Emin olunamıyorsa önce `instanceof` ile sınanır.

İlgisiz tipler arasında dönüşümü derleyici hiç kabul etmez: `(Kedi) kopek` → *incompatible types: Kopek cannot be converted to Kedi.*

`ClassCastException`: Kod İzleme

```
▶ Hayvan h1 = new Kopek();
2 Hayvan h2 = new Kedi();
3 Kopek k1 = (Kopek) h1;
4 k1.havla();
5 Kopek k2 = (Kopek) h2;
6 k2.havla();
7 System.out.println("Bitti");
```



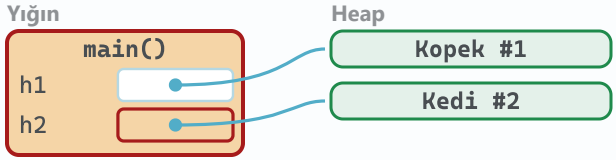
Yukarı dönüşüm: `Hayvan` tipindeki `h1` bir `Kopek` nesnesini gösterir.

Adım 1 / 6

ÇIKTI

`ClassCastException`: Kod İzleme

```
1 Hayvan h1 = new Kopek();
▶ Hayvan h2 = new Kedi();
3 Kopek k1 = (Kopek) h1;
4 k1.havla();
5 Kopek k2 = (Kopek) h2;
6 k2.havla();
7 System.out.println("Bitti");
```



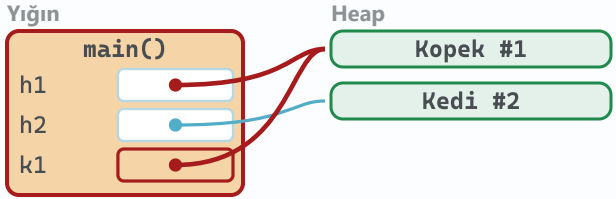
`h2` ise bir `Kedi` gösterir. Derleyici için ikisi de yalnızca `Hayvan`.

Adım 2 / 6

ÇIKTI

`ClassCastException`: Kod İzleme

```
1 Hayvan h1 = new Kopek();
2 Hayvan h2 = new Kedi();
▶ Kopek k1 = (Kopek) h1;
4 k1.havla();
5 Kopek k2 = (Kopek) h2;
6 k2.havla();
7 System.out.println("Bitti");
```



JVM denetimi: nesnenin sınıfı

`Kopek` = **Kopek** ✓

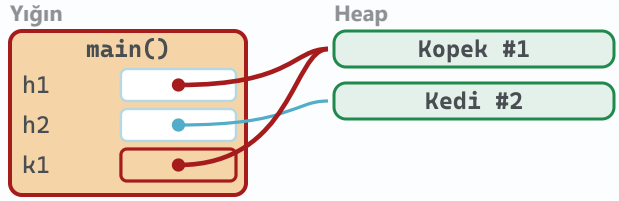
Derleyici dönüşüme izin verir. Çalışırken JVM nesneye bakar: gerçekten `Kopek`. `k1` aynı nesneyi gösterir.

Adım 3 / 6

ÇIKTI

`ClassCastException`: Kod İzleme

```
1 Hayvan h1 = new Kopek();
2 Hayvan h2 = new Kedi();
3 Kopek k1 = (Kopek) h1;
▶ k1.havla();
5 Kopek k2 = (Kopek) h2;
6 k2.havla();
7 System.out.println("Bitti");
```



`Kopek` tipli referansla `havla()` çağrılabilir.

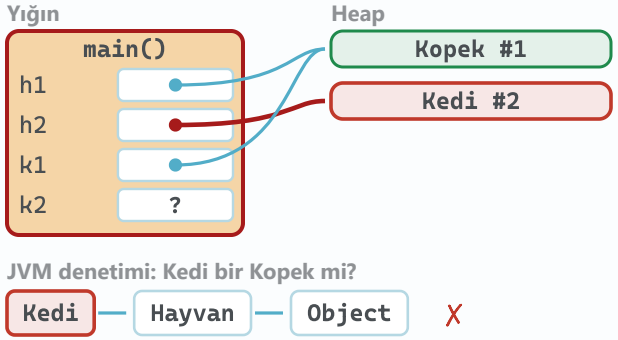
Adım 4 / 6

ÇIKTI

```
Köpek havlıyor
```

`ClassCastException`: Kod İzleme

```
1 Hayvan h1 = new Kopek();
2 Hayvan h2 = new Kedi();
3 Kopek k1 = (Kopek) h1;
4 k1.havla();
▶ Kopek k2 = (Kopek) h2;
6 k2.havla();
7 System.out.println("Bitti");
```



Bu satır da **derlenir**. Ama JVM, `Kedi`'den yukarı çıkan zincirde `Kopek` bulamaz.

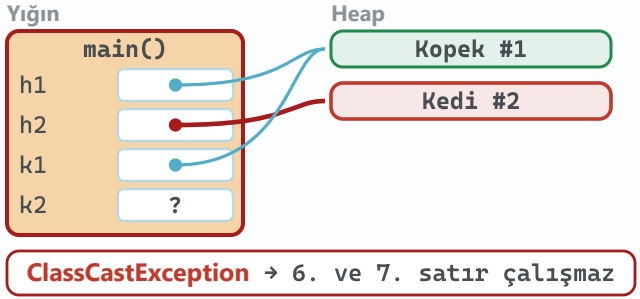
Adım 5 / 6

ÇIKTI

```
Köpek havlıyor
```

`ClassCastException`: Kod İzleme

```
1 Hayvan h1 = new Kopek();
2 Hayvan h2 = new Kedi();
3 Kopek k1 = (Kopek) h1;
4 k1.havla();
▶ Kopek k2 = (Kopek) h2;
6 k2.havla();
7 System.out.println("Bitti");
```



`ClassCastException` fırlatılır, `k2` 'ye hiçbir şey atanmaz; yakalayan olmadığı için program durur, **Bitti** yazılmaz.

Adım 6 / 6

ÇIKTI

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

TİP DÖNÜŞÜMÜ

`instanceof` ile Güvenli Dönüşüm

`hayvanlar` = { `new Kopek()`, `new Kedi()`, `new Hayvan()` }

```
for (Hayvan x : hayvanlar) {
    if (x instanceof Kopek) {
        Kopek kk = (Kopek) x;
        kk.havla();
    } else if (x instanceof Kedi) {
        ((Kedi) x).tirmala();
    } else {
        System.out.println("Özel davranış yok");
    }
}
```

```
Köpek havlıyor
Kedi tırmalıyor
Özel davranış yok
```

- `referans instanceof Tur` : nesne `Tur` 'den ya da onun bir alt sınıfındansa `true` .
- Önce sınama, sonra dönüşüm: `ClassCastException` oluşamaz. `null instanceof Tur` her zaman `false` .

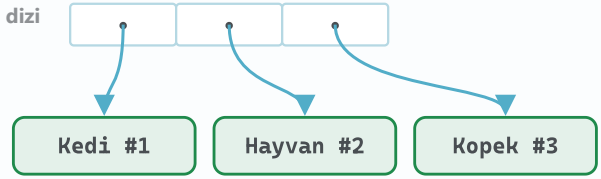
JAVA 16+: ÖRÜNTÜ EŞLEME (PATTERN MATCHING)

```
if (x instanceof Kopek kk) {
    kk.havla(); // dönüşüm otomatik
}
```

Uzun `instanceof` zincirleri çoğunlukla bir tasarım uyarısıdır: davranışı alt sınıflarda **geçersiz kılınan bir metoda** taşımak daha iyidir (Alıştırma 5).

`instanceof` ile Güvenli Dönüşüm: Kod İzleme

```
▶ Hayvan[] dizi = {new Kedi(), new Hayvan(), new Kopek()};
2 for (Hayvan x : dizi) {
3     if (x instanceof Kopek) {
4         ((Kopek) x).havla();
5     } else if (x instanceof Kedi) {
6         ((Kedi) x).tirmala();
7     } else {
8         System.out.println("Özel davranış yok");
9     }
10 }
```



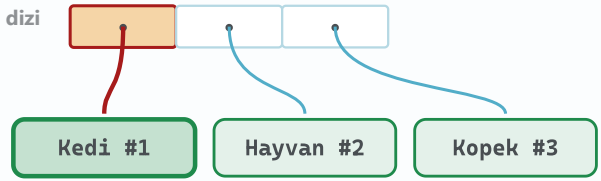
Dizinin üç öğesi de `Hayvan` tipinde referans; gösterdikleri nesneler farklı sınıflardan.

Adım 1 / 6

ÇIKTI

`instanceof` ile Güvenli Dönüşüm: Kod İzleme

```
1  Hayvan[] dizi = {new Kedi(), new Hayvan(), new Kopek()};
▶ for (Hayvan x : dizi) {
▶     if (x instanceof Kopek) {
4         ((Kopek) x).havla();
5     } else if (x instanceof Kedi) {
6         ((Kedi) x).tirmala();
7     } else {
8         System.out.println("Özel davranış yok");
9     }
10 }
```



x → Kedi #1

`x instanceof Kopek` **false**

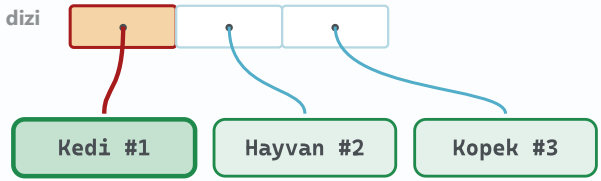
İlk tur: `x` bir `Kedi`. `Kopek` sınaması yanlış.

Adım 2 / 6

ÇIKTI

`instanceof` ile Güvenli Dönüşüm: Kod İzleme

```
1  Hayvan[] dizi = {new Kedi(), new Hayvan(), new Kopek()};
2  for (Hayvan x : dizi) {
3      if (x instanceof Kopek) {
4          ((Kopek) x).havla();
▶      } else if (x instanceof Kedi) {
▶          ((Kedi) x).tirmala();
7      } else {
8          System.out.println("Özel davranış yok");
9      }
10 }
```



x → **Kedi #1**

`x instanceof Kopek`	false
`x instanceof Kedi`	true

`Kedi` sınaması doğru; önce sınandığı için `(Kedi) x` dönüşümü güvenli: **Kedi tırmalıyor**.

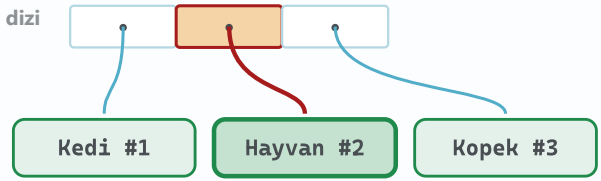
Adım 3 / 6

ÇIKTI

```
Kedi tırmalıyor
```

`instanceof` ile Güvenli Dönüşüm: Kod İzleme

```
1  Hayvan[] dizi = {new Kedi(), new Hayvan(), new Kopek()};
▶  for (Hayvan x : dizi) {
▶      if (x instanceof Kopek) {
4          ((Kopek) x).havla();
▶      } else if (x instanceof Kedi) {
6          ((Kedi) x).tirmala();
7      } else {
8          System.out.println("Özel davranış yok");
9      }
10 }
```



x → Hayvan #2

`x instanceof Kopek`	**false**
`x instanceof Kedi`	**false**

İkinci tur: düz bir `Hayvan`. Hiçbir alt sınıfın örneği değil: iki sınama da yanlış.

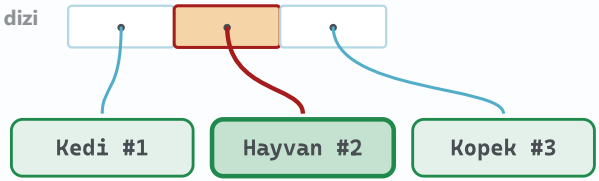
Adım 4 / 6

ÇIKTI

```
Kedi tırmalıyor
```

`instanceof` ile Güvenli Dönüşüm: Kod İzleme

```
1  Hayvan[] dizi = {new Kedi(), new Hayvan(), new Kopek()};
2  for (Hayvan x : dizi) {
3      if (x instanceof Kopek) {
4          ((Kopek) x).havla();
5      } else if (x instanceof Kedi) {
6          ((Kedi) x).tirmala();
7      } else {
▶          System.out.println("Özel davranış yok");
9      }
10 }
```



x → **Hayvan #2**

`x instanceof Kopek`	**false**
`x instanceof Kedi`	**false**

`else` dalı çalışır; hiçbir dönüşüm denenmediği için hata riski yok.

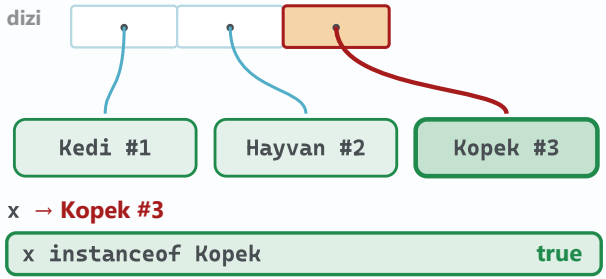
Adım 5 / 6

ÇIKTI

```
Kedi tırmalıyor
Özel davranış yok
```

`instanceof` ile Güvenli Dönüşüm: Kod İzleme

```
1  Hayvan[] dizi = {new Kedi(), new Hayvan(), new Kopek()};
▶ for (Hayvan x : dizi) {
▶     if (x instanceof Kopek) {
▶         ((Kopek) x).havla();
5      } else if (x instanceof Kedi) {
6          ((Kedi) x).tirmala();
7      } else {
8          System.out.println("Özel davranış yok");
9      }
10 }
```



Üçüncü tur: `Kopek` sınaması doğru, `havla()` çalışır. Döngü biter; hiçbir turda `ClassCastException` oluşmadı.

Adım 6 / 6

ÇIKTI

```
Kedi tırmalıyor
Özel davranış yok
Köpek havlıyor
```

Diziler ve `ArrayStoreException`

```
Kopek[] kopekler = new Kopek[2];
Hayvan[] hayvanlar = kopekler;   // derlenir
hayvanlar[0] = new Kopek();
System.out.println("ilk atama tamam");
hayvanlar[1] = new Kedi();       // derlenir ama...
System.out.println("buraya gelinmez");
```

```
ilk atama tamam
Exception in thread "main"
java.lang.ArrayStoreException: Kedi
```

Aynı fikir jenerik listede

```
List<Hayvan> l = new ArrayList<Kopek>();
```

```
error: incompatible types: ArrayList<Kopek>
  cannot be converted to List<Hayvan>
```

- Diziler **kovaryanttır**: `Kopek` bir `Hayvan` olduğundan `Kopek[]` de bir `Hayvan[]` sayılır ve yukarı dönüşüm derlenir.
- Ama dizi nesnesi gerçekte `Kopek[]` 'dir. `Hayvan[]` referansı üzerinden içine bir `Kedi` yazmak derleyicinin gözünde geçerlidir; JVM her yazmada öğenin tipini denetler ve `ArrayStoreException` fırlatır.
- Jenerik koleksiyonlar bu açığı derleme zamanında kapatır: `List<Kopek>`, `List<Hayvan>` değildir.
- Okumak için kovaryans güvenlidir (her öğe bir `Hayvan` 'dır); sorun yalnızca **yazarken** çıkar.

Dinamik Bağlama (Dynamic Binding)

- Referans bir alt sınıf nesnesini gösteriyorsa, **alt sınıfın** metodu çağrılır.
- Çağrılacak sürüm, değişkenin bildirilen tipine göre değil, **nesnenin gerçek sınıfına** göre belirlenir.
- Bu seçim derleme sırasında değil, **çalışma zamanında** yapıldığı için dinamik (geç) bağlama denir.

```
Hayvan h = new Kopek();
h.kos();      // Kopek sınıfının metodu
h.sesCikar(); // Kopek sınıfının metodu
h.uyu();      // Kopek'te yok → Hayvan'ınki
```

```
Köpek koşuyor
Hav hav!
Hayvan uyuyor
```

`h.kos()` İKİ AŞAMADA ÇÖZÜLÜR

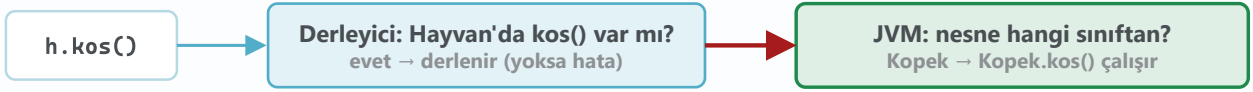
1. Derleme zamanı

Derleyici `h`'nin tipine (`Hayvan`) bakar: `kos()` var mı? **Var** → çağrı derlenir. Yoksa derleme hatası.

2. Çalışma zamanı

JVM nesneye bakar: gerçek sınıf `Kopek` → `Kopek.kos()` çalışır.

Kural: **derleyici "var mı?"** diye, **JVM "hangisi?"** diye sorar.



Metot Araması Hiyerarşinin En Altından Başlar

```
▶ Hayvan h = new Kopek();
2 h.kos();
3 h.uyu();
4 System.out.println(h.toString());
5 // h.havla(); → derleme hatası
```

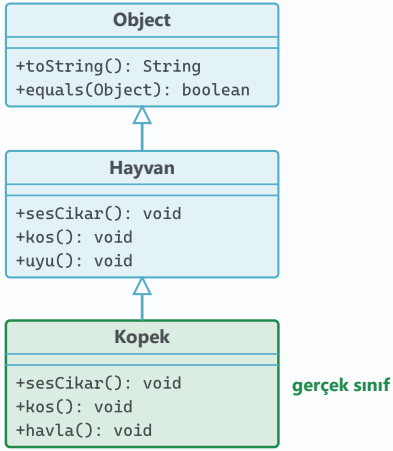
ÇAĞRI YIĞINI

`main()`

DEĞİŞKENLER

`h = → Kopek nesnesi`

ÇIKTI



Nesne bir `Kopek`: aramalar hep **en alttan**, gerçek sınıftan başlar.

Metot Araması Hiyerarşinin En Altından Başlar

```
1 Hayvan h = new Kopek();
▶ h.kos();
3 h.uyu();
4 System.out.println(h.toString());
5 // h.havla(); → derleme hatası
```

ÇAĞRI YIĞINI

`Kopek.kos()`

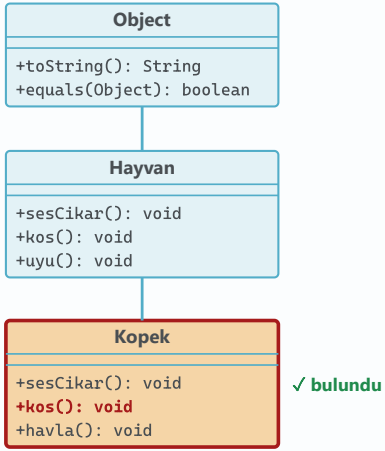
`main()`

DEĞİŞKENLER

`h = → Kopek nesnesi`

ÇIKTI

```
Köpek koşuyor
```



`kos()` `Kopek` 'te var → arama biter, Kopek sürümü çalışır.

Metot Araması Hiyerarşinin En Altından Başlar

```
1 Hayvan h = new Kopek();
2 h.kos();
▶ h.uyu();
4 System.out.println(h.toString());
5 // h.havla(); → derleme hatası
```

ÇAĞRI YIĞINI

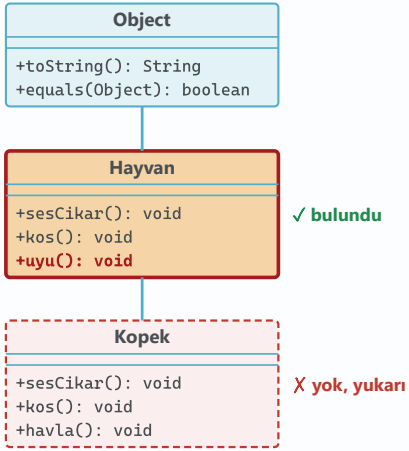
- `Hayvan.uyu()`
- `main()`

DEĞİŞKENLER

`h` = → **Kopek nesnesi**

ÇIKTI

```
Köpek koşuyor
Hayvan uyuyor
```



`uyu()` `Kopek`'te yok → bir üste çık: `Hayvan`'da bulundu.

Metot Araması Hiyerarşinin En Altından Başlar

```
1 Hayvan h = new Kopek();
2 h.kos();
3 h.uyu();
▶ System.out.println(h.toString());
5 // h.havla(); → derleme hatası
```

ÇAĞRI YIĞINI

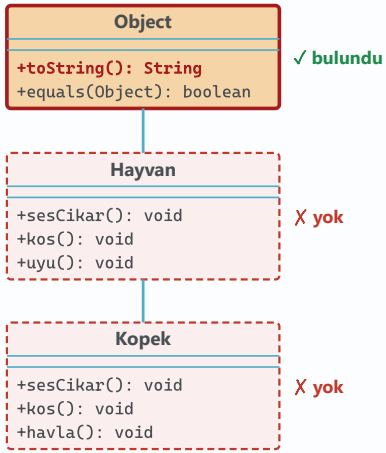
- **`Object.toString()`**
- `main()`

DEĞİŞKENLER

`h` = → **Kopek nesnesi**

ÇIKTI

```
Köpek koşuyor
Hayvan uyuyor
Kopek@6b884d57
```



`toString()` en tepede, `Object`'te bulunur: sınıf adı + `@` + onaltılık hash kodu (çalıştırmaya göre değişir).

Metot Araması Hiyerarşinin En Altından Başlar

```
1 Hayvan h = new Kopek();
2 h.kos();
3 h.uyu();
4 System.out.println(h.toString());
▶ // h.havla(); → derleme hatası
```

ÇAĞRI YIĞINI

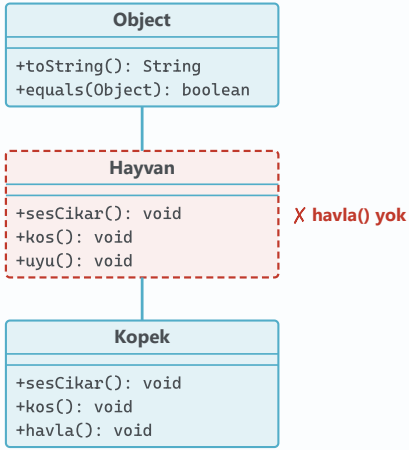
`main()`

DEĞİŞKENLER

`h` = → **Kopek nesnesi**

ÇIKTI

```
Köpek koşuyor
Hayvan uyuyor
Kopek@6b884d57
```



`h.havla()` derlenmez: derleyici aramayı **referans tipi** `Hayvan` 'dan yukarı yapar ve bulamaz.

Çok Biçimli Dizi: Adım Adım Kod İzleme

```
▶ Hayvan[] hayvanlar = new Hayvan[3];
2 hayvanlar[0] = new Kopek();
3 hayvanlar[1] = new Kedi();
4 hayvanlar[2] = new Hayvan();
5 for (int i = 0; i < hayvanlar.length; i++)
  {
6     hayvanlar[i].sesCikar();
7 }
```

ÇAĞRI YIĞINI

`main()`

DEĞİŞKENLER

`i = –`

ÇIKTI





`Hayvan` tipinde 3 öğeli dizi: öğeler referanstır, başlangıçta `null`.

Adım 1 / 11

Çok Biçimli Dizi: Adım Adım Kod İzleme

```
1  Hayvan[] hayvanlar = new Hayvan[3];
▶  hayvanlar[0] = new Kopek();
3  hayvanlar[1] = new Kedi();
4  hayvanlar[2] = new Hayvan();
5  for (int i = 0; i < hayvanlar.length; i++)
   {
6      hayvanlar[i].sesCikar();
7  }
```

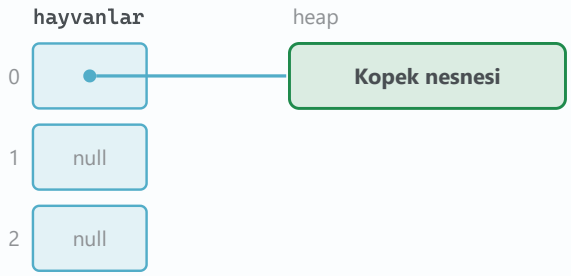
ÇAĞRI YIĞINI

`main()`

DEĞİŞKENLER

`i = -`

ÇIKTI



0. öğe bir `Kopek` 'i gösterir (yukarı dönüşüm, otomatik).

Çok Biçimli Dizi: Adım Adım Kod İzleme

```
1 Hayvan[] hayvanlar = new Hayvan[3];
2 hayvanlar[0] = new Kopek();
▶ hayvanlar[1] = new Kedi();
4 hayvanlar[2] = new Hayvan();
5 for (int i = 0; i < hayvanlar.length; i++)
  {
6     hayvanlar[i].sesCikar();
7 }
```

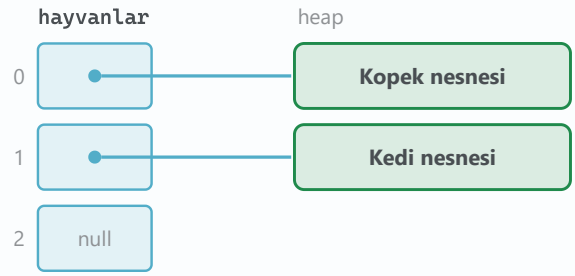
ÇAĞRI YIĞINI

`main()`

DEĞİŞKENLER

`i = –`

ÇIKTI



1. öğe bir `Kedi` 'yi gösterir.

Çok Biçimli Dizi: Adım Adım Kod İzleme

```
1 Hayvan[] hayvanlar = new Hayvan[3];
2 hayvanlar[0] = new Kopek();
3 hayvanlar[1] = new Kedi();
▶ hayvanlar[2] = new Hayvan();
5 for (int i = 0; i < hayvanlar.length; i++)
  {
6     hayvanlar[i].sesCikar();
7 }
```

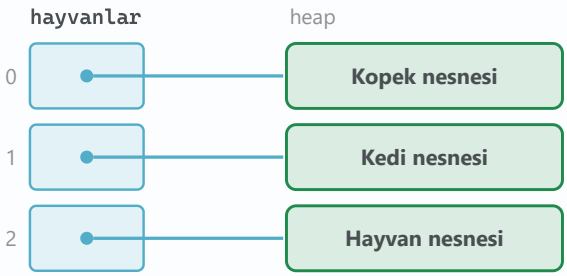
ÇAĞRI YIĞINI

`main()`

DEĞİŞKENLER

`i = –`

ÇIKTI



2. öğe düz bir `Hayvan` nesnesidir. Dizi artık üç farklı biçim tutuyor.

Çok Biçimli Dizi: Adım Adım Kod İzleme

```
1 Hayvan[] hayvanlar = new Hayvan[3];
2 hayvanlar[0] = new Kopek();
3 hayvanlar[1] = new Kedi();
4 hayvanlar[2] = new Hayvan();
▶ for (int i = 0; i < hayvanlar.length; i++)
  {
6     hayvanlar[i].sesCikar();
7 }
```

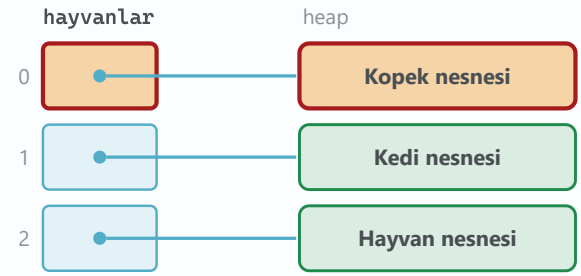
ÇAĞRI YIĞINI

`main()`

DEĞİŞKENLER

`i = 0`

ÇIKTI



`i = 0`, `0 < 3` doğru → döngü gövdesi.

Çok Biçimli Dizi: Adım Adım Kod İzleme

```
1 Hayvan[] hayvanlar = new Hayvan[3];
2 hayvanlar[0] = new Kopek();
3 hayvanlar[1] = new Kedi();
4 hayvanlar[2] = new Hayvan();
5 for (int i = 0; i < hayvanlar.length; i++)
  {
▶     hayvanlar[i].sesCikar();
7 }
```

ÇAĞRI YIĞINI

`Kopek.sesCikar()`

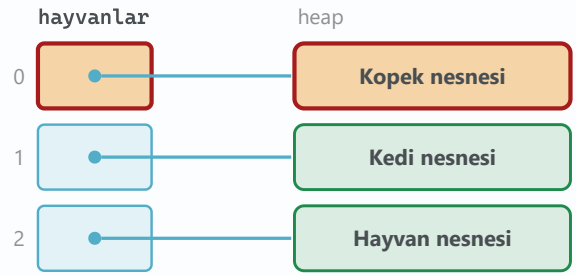
`main()`

DEĞİŞKENLER

`i = 0`

ÇIKTI

```
Hav hav!
```



Aynı satır, farklı nesne: JVM nesnenin sınıfına bakar → `Kopek.sesCikar()`.

Çok Biçimli Dizi: Adım Adım Kod İzleme

```
1 Hayvan[] hayvanlar = new Hayvan[3];
2 hayvanlar[0] = new Kopek();
3 hayvanlar[1] = new Kedi();
4 hayvanlar[2] = new Hayvan();
▶ for (int i = 0; i < hayvanlar.length; i++)
  {
6     hayvanlar[i].sesCikar();
7 }
```

ÇAĞRI YIĞINI

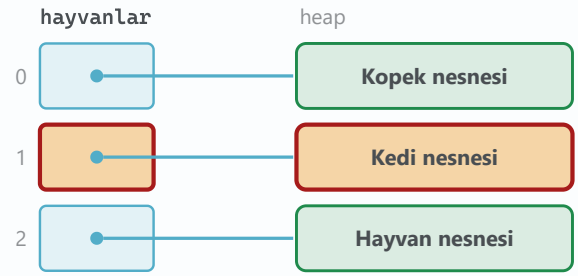
`main()`

DEĞİŞKENLER

`i = 1`

ÇIKTI

```
Hav hav!
```



`i = 1`, `1 < 3` doğru → döngü gövdesi.

Çok Biçimli Dizi: Adım Adım Kod İzleme

```
1 Hayvan[] hayvanlar = new Hayvan[3];
2 hayvanlar[0] = new Kopek();
3 hayvanlar[1] = new Kedi();
4 hayvanlar[2] = new Hayvan();
5 for (int i = 0; i < hayvanlar.length; i++)
  {
▶     hayvanlar[i].sesCikar();
7 }
```

ÇAĞRI YIĞINI

`Kedi.sesCikar()`

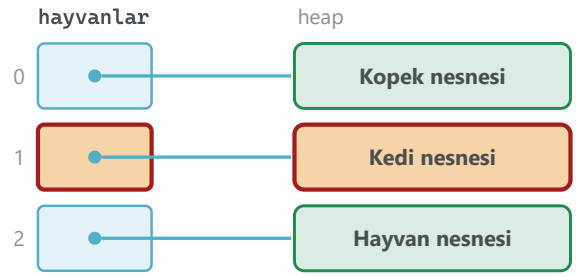
`main()`

DEĞİŞKENLER

`i = 1`

ÇIKTI

```
Hav hav!
Miyav!
```



Aynı satır, farklı nesne: JVM nesnenin sınıfına bakar → `Kedi.sesCikar()`.

Çok Biçimli Dizi: Adım Adım Kod İzleme

```
1 Hayvan[] hayvanlar = new Hayvan[3];
2 hayvanlar[0] = new Kopek();
3 hayvanlar[1] = new Kedi();
4 hayvanlar[2] = new Hayvan();
▶ for (int i = 0; i < hayvanlar.length; i++)
  {
6     hayvanlar[i].sesCikar();
7 }
```

ÇAĞRI YIĞINI

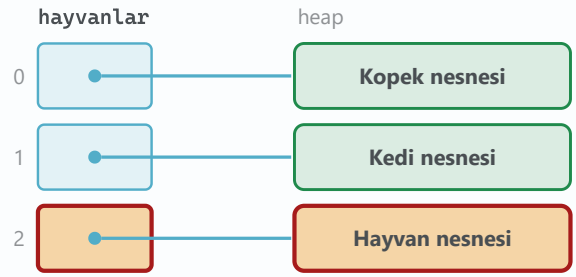
`main()`

DEĞİŞKENLER

`i = 2`

ÇIKTI

```
Hav hav!
Miyav!
```



`i = 2`, `2 < 3` doğru → döngü gövdesi.

Çok Biçimli Dizi: Adım Adım Kod İzleme

```
1 Hayvan[] hayvanlar = new Hayvan[3];
2 hayvanlar[0] = new Kopek();
3 hayvanlar[1] = new Kedi();
4 hayvanlar[2] = new Hayvan();
5 for (int i = 0; i < hayvanlar.length; i++)
  {
▶     hayvanlar[i].sesCikar();
7 }
```

ÇAĞRI YIĞINI

`Hayvan.sesCikar()`

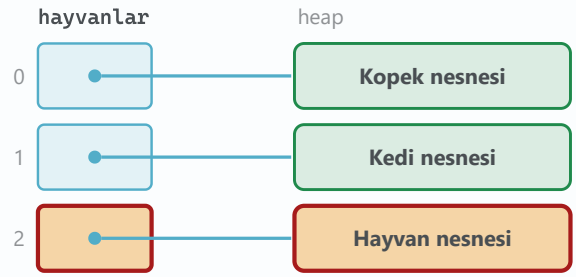
`main()`

DEĞİŞKENLER

`i = 2`

ÇIKTI

```
Hav hav!
Miyav!
Hayvan sesi
```



Aynı satır, farklı nesne: JVM nesnenin sınıfına bakar → `Hayvan.sesCikar()`.

Adım 10 / 11

Çok Biçimli Dizi: Adım Adım Kod İzleme

```
1 Hayvan[] hayvanlar = new Hayvan[3];
2 hayvanlar[0] = new Kopek();
3 hayvanlar[1] = new Kedi();
4 hayvanlar[2] = new Hayvan();
▶ for (int i = 0; i < hayvanlar.length; i++)
  {
6     hayvanlar[i].sesCikar();
7 }
```

ÇAĞRI YIĞINI

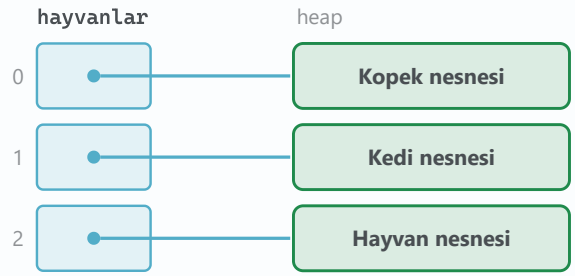
`main()`

DEĞİŞKENLER

`i = 3`

ÇIKTI

```
Hav hav!
Miyav!
Hayvan sesi
```



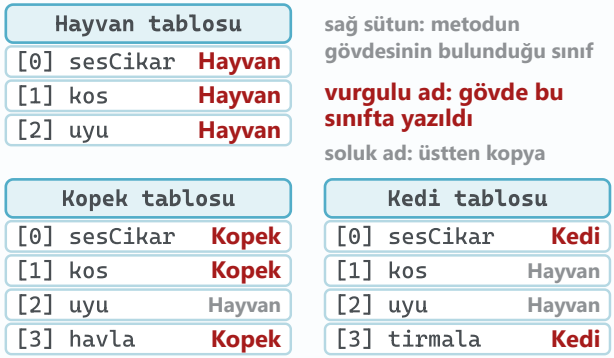
`i = 3`, `3 < 3` yanlış → döngü biter. Tek satır kod, üç farklı davranış.

Perde Arkası: Metot Tablosu

- JVM'ler dinamik bağlamayı genellikle sınıf başına bir **metot tablosuyla** (virtual method table) uygular: her geçersiz kılınabilir metodun sabit bir sıra numarası (yuvası) vardır.
- Alt sınıfın tablosu üst sınıfınkinin kopyasıyla başlar; geçersiz kılınan yuvalar yeni gövdeyle **değiştirilir**, yeni metotlar sona **eklenir**.
- Her nesne kendi sınıfını bilir. `h.kos()` için JVM: nesne → sınıfı → tablodaki `kos` yuvası → gövde. Arama tek bir dizi erişimidir, hiyerarşide yukarı yürümek gerekmez.

`javap -c Ana` çıktısından

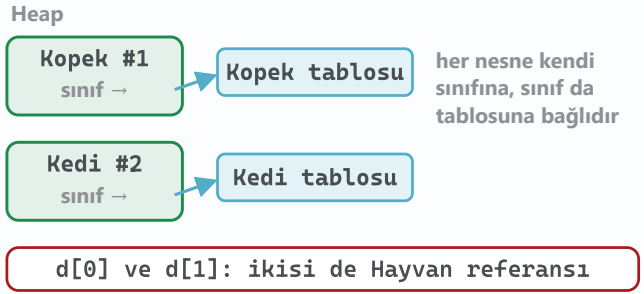
```
47: invokevirtual #15  // Method Hayvan.kos:()V
52: invokevirtual #18  // Method Hayvan.sesCikar:()V
```



Derleyici bayt koduna yalnızca `Hayvan.kos` yazar; gövdeyi `invokevirtual`, nesnenin tablosuna bakarak seçer. Yapıcılar ve `super.m()` `invokespecial`, `static` metotlar `invokestatic` ile çağrılır: tabloya bakılmaz. `private` metotlar Java 11'den beri `invokevirtual` kullanır, ama tek adayları vardır.

Metot Tablosu ile Çağrı: Kod İzleme

```
1 Hayvan[] d = {new Kopek(), new Kedi()};
2 for (Hayvan h : d) {
3     h.kos();
4     h.sesCikar();
5 }
```



Dizideki iki nesne farklı sınıflardandır; JVM için her biri kendi sınıfının tablosuna giden bir bağlantı taşır.

Adım 1 / 5

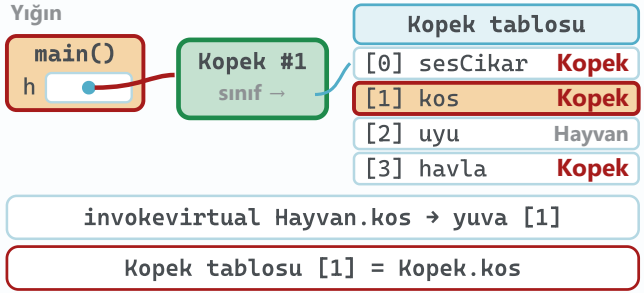
ÇIKTI

DEĞİŞKENLER

—

Metot Tablosu ile Çağrı: Kod İzleme

```
1 Hayvan[] d = {new Kopek(), new Kedi()};
2 for (Hayvan h : d) {
▶     h.kos();
4     h.sesCikar();
5 }
```



Bayt kodu "`Hayvan.kos`, yuva 1" der. Nesne bir `Kopek`: onun tablosunun 1. yuvasında `Kopek.kos` vardır.

Adım 2 / 5

ÇIKTI

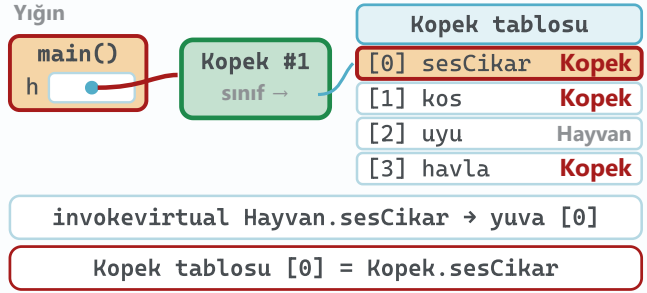
```
Köpek koşuyor
```

DEĞİŞKENLER

h = →Kopek #1

Metot Tablosu ile Çağrı: Kod İzleme

```
1 Hayvan[] d = {new Kopek(), new Kedi()};
2 for (Hayvan h : d) {
3     h.kos();
▶     h.sesCikar();
5 }
```



Aynı yol, 0. yuva: `Hav hav!`

Adım 3 / 5

ÇIKTI

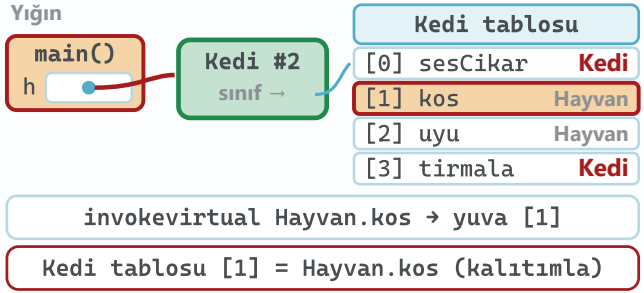
```
Köpek koşuyor
Hav hav!
```

DEĞİŞKENLER

h = →Kopek #1

Metot Tablosu ile Çağrı: Kod İzleme

```
1 Hayvan[] d = {new Kopek(), new Kedi()};
2 for (Hayvan h : d) {
▶     h.kos();
4     h.sesCikar();
5 }
```



Aynı bayt kodu, başka nesne. `Kedi` `kos`'u geçersiz kılmadığı için tablosunun 1. yuvası `Hayvan.kos`'u gösterir.

Adım 4 / 5

ÇIKTI

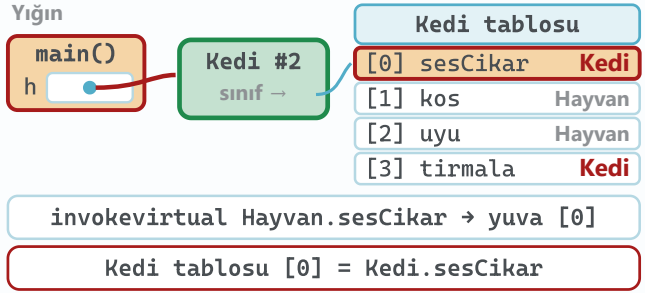
```
Köpek koşuyor
Hav hav!
Hayvan koşuyor
```

DEĞİŞKENLER

h = →Kedi #2

Metot Tablosu ile Çağrı: Kod İzleme

```
1 Hayvan[] d = {new Kopek(), new Kedi()};
2 for (Hayvan h : d) {
3     h.kos();
▶     h.sesCikar();
5 }
```



`Miyav!` Dört çağrının hiçbirinde `if` ya da tip sınaması yapılmadı: seçimi tablolar yaptı.

Adım 5 / 5

ÇIKTI

```
Köpek koşuyor
Hav hav!
Hayvan koşuyor
Miyav!
```

DEĞİŞKENLER

h = →Kedi #2

Neler Dinamik Bağlanmaz?

```
class Hayvan {
    String tur = "hayvan";

    static void bilgi() {
        System.out.println("Hayvan.bilgi()");
    }

    void tanit() {
        System.out.println("Ben bir " + tur);
    }
}

class Kopek extends Hayvan {
    String tur = "köpek"; // gizler, ezmez

    static void bilgi() {
        System.out.println("Kopek.bilgi()");
    }

    @Override
    void tanit() {
        System.out.println("Ben bir " + tur);
    }
}
```

```
Hayvan h = new Kopek();
System.out.println(h.tur);
h.bilgi();
h.tanit();
```

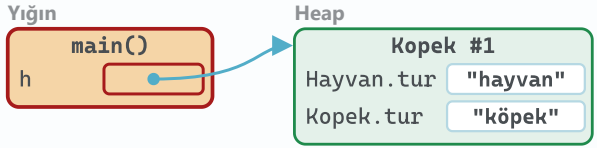
```
hayvan
Hayvan.bilgi()
Ben bir köpek
```

Üye	Neye göre seçilir?
Örnek metodu	**Nesne** tipi (dinamik)
Alan (değişken)	Referans tipi (statik)
`static` metot	Referans tipi (statik)
`private` / `final`	Geçersiz kılınamaz, static

static metot sınıf adıyla çağrılır: `Kopek.bilgi()`. Alt sınıfta aynı adla alan tanımlamaktan kaçınılır.

Alan Gizleme ve `static`: Kod İzleme

```
1  class Hayvan {
2      String tur = "hayvan";
3      static String bilgi() { return "Hayvan.bilgi"; }
4      String tanit() { return "Ben bir " + tur; }
5  }
6  class Kopek extends Hayvan {
7      String tur = "köpek";            // gizler
8      static String bilgi() { return "Kopek.bilgi"; }
9      String tanit() { return "Ben bir " + tur; }
10 }
▶  Hayvan h = new Kopek();   // main
12 System.out.println(h.tur);
13 System.out.println(h.bilgi());
14 System.out.println(h.tanit());
15 System.out.println(((Kopek) h).tur);
```



`bir nesne, iki ayrı tur alanı`

Alt sınıftaki `tur` üst sınıfınkini ezmez, **gizler**: nesnede iki ayrı alan vardır.

Adım 1 / 5

ÇIKTI

Alan Gizleme ve `static`: Kod İzleme

```
 1 class Hayvan {
 2     String tur = "hayvan";
 3     static String bilgi() { return "Hayvan.bilgi"; }
 4     String tanit() { return "Ben bir " + tur; }
 5 }
 6 class Kopek extends Hayvan {
 7     String tur = "köpek";            // gizler
 8     static String bilgi() { return "Kopek.bilgi"; }
 9     String tanit() { return "Ben bir " + tur; }
10 }
11 Hayvan h = new Kopek();   // main
▶ System.out.println(h.tur);
13 System.out.println(h.bilgi());
14 System.out.println(h.tanit());
15 System.out.println(((Kopek) h).tur);
```



`alan → bildirilen tip (Hayvan)`

Alanlar dinamik bağlanmaz: `h` 'nin bildirilen tipi `Hayvan` olduğu için `Hayvan.tur` okunur: **hayvan**.

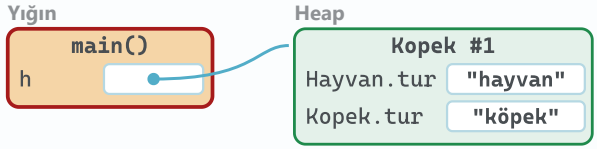
Adım 2 / 5

ÇIKTI

```
hayvan
```

Alan Gizleme ve `static`: Kod İzleme

```
 1 class Hayvan {
 2     String tur = "hayvan";
 ▶     static String bilgi() { return "Hayvan.bilgi"; }
 4     String tanit() { return "Ben bir " + tur; }
 5 }
 6 class Kopek extends Hayvan {
 7     String tur = "köpek";            // gizler
 8     static String bilgi() { return "Kopek.bilgi"; }
 9     String tanit() { return "Ben bir " + tur; }
10 }
11 Hayvan h = new Kopek();   // main
12 System.out.println(h.tur);
▶ System.out.println(h.bilgi());
14 System.out.println(h.tanit());
15 System.out.println(((Kopek) h).tur);
```



`static metot → bildirilen tip; nesneye bakılmaz`

`static` metot sınıfa aittir; derleyici `h.bilgi()`'yi `Hayvan.bilgi()` olarak bağlar.

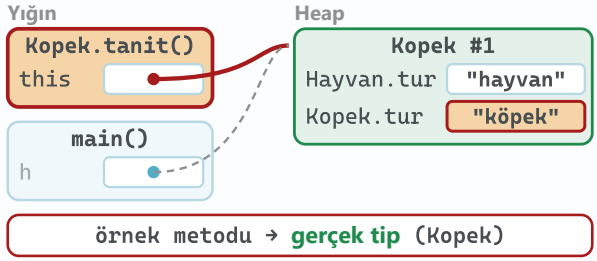
Adım 3 / 5

ÇIKTI

```
hayvan
Hayvan.bilgi
```

Alan Gizleme ve `static`: Kod İzleme

```
 1 class Hayvan {
 2     String tur = "hayvan";
 3     static String bilgi() { return "Hayvan.bilgi"; }
 4     String tanit() { return "Ben bir " + tur; }
 5 }
 6 class Kopek extends Hayvan {
 7     String tur = "köpek";            // gizler
 8     static String bilgi() { return "Kopek.bilgi"; }
 ▶     String tanit() { return "Ben bir " + tur; }
10 }
11 Hayvan h = new Kopek();   // main
12 System.out.println(h.tur);
13 System.out.println(h.bilgi());
 ▶ System.out.println(h.tanit());
15 System.out.println(((Kopek) h).tur);
```



Örnek metodu dinamik bağlanır: `Kopek.tanit()` çalışır; onun içinde `tur`, `Kopek`'in alanıdır: **Ben bir köpek**.

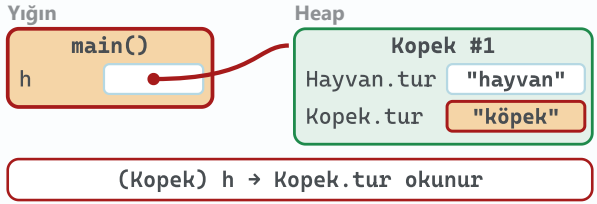
Adım 4 / 5

ÇIKTI

```
hayvan
Hayvan.bilgi
Ben bir köpek
```

Alan Gizleme ve `static`: Kod İzleme

```
1  class Hayvan {
2      String tur = "hayvan";
3      static String bilgi() { return "Hayvan.bilgi"; }
4      String tanit() { return "Ben bir " + tur; }
5  }
6  class Kopek extends Hayvan {
7      String tur = "köpek";            // gizler
8      static String bilgi() { return "Kopek.bilgi"; }
9      String tanit() { return "Ben bir " + tur; }
10 }
11 Hayvan h = new Kopek();   // main
12 System.out.println(h.tur);
13 System.out.println(h.bilgi());
14 System.out.println(h.tanit());
▶  System.out.println(((Kopek) h).tur);
```



Dönüşüm bildirilen tipi `Kopek` yapar; aynı nesnenin diğer alanı okunur: **köpek**. Bu karışıklık yüzünden alt sınıfta aynı adlı alan tanımlanmaz.

Adım 5 / 5

ÇIKTI

```
hayvan
Hayvan.bilgi
Ben bir köpek
köpek
```

Yapıcıdan Geçersiz Kılınabilir Metot Çağırmak

- Dinamik bağlama nesne **kurulurken** de çalışır: üst sınıfın yapıcısındaki `ciz()` çağrısı, nesne bir `Etiket` olduğu için `Etiket.ciz()` 'i çalıştırır.
- Ama o anda alt sınıfın alan ilk değerleri henüz atanmamıştır: `metin` hâlâ `null`. Alt sınıfın metodu, yarı kurulmuş bir nesne üzerinde çalışır.
- Metot alana yalnızca yazdırmak için değil, kullanmak için dokunsaydı program çökerdi.
- Kural: yapıcıdan yalnızca `private`, `final` ya da `static` metotlar çağrılmalıdır; bunlar geçersiz kılınamaz.

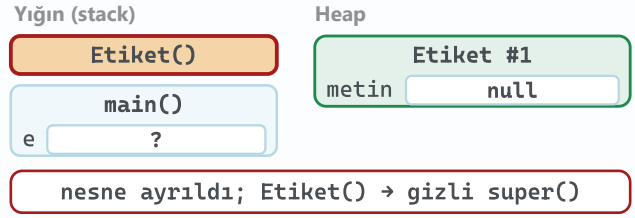
`ciz()` içinde `metin.length()` kullanılsaydı

```
Exception in thread "main"
java.lang.NullPointerException: Cannot invoke
"String.length()" because "this.metin" is null
        at Etiket.ciz(Ana.java:21)
        at Bilesen.<init>(Ana.java:16)
        at Etiket.<init>(Ana.java:19)
```

Yığın izindeki sıra durumu özetler: `Etiket` yapıcısı → `Bilesen` yapıcısı → yine `Etiket` 'in metodu. `<init>` , bayt kodunda yapıcının adıdır.

Yapıcıdaki Çağrı: Kod İzleme

```
 1 class Bilesen {
 2     Bilesen() {
 3         System.out.println("Bilesen kuruluyor");
 4         ciz();
 5     }
 6     void ciz() { System.out.println("boş bileşen"); }
 7 }
 8 class Etiket extends Bilesen {
 9     private String metin = "Merhaba";
▶      Etiket() { System.out.println("Etiket kuruldu"); }
11     @Override void ciz() {
12         System.out.println("Etiket: " + metin);
13     }
14 }
▶  Etiket e = new Etiket();   // main
16 e.ciz();
```



`new` nesneye yer ayırır, `metin` varsayılan `null`. `Etiket()`'in ilk işi gizli `super()` çağrısıdır.

Adım 1 / 6

ÇIKTI

DEĞİŞKENLER

`metin = null`

Yapıcıdaki Çağrı: Kod İzleme

```
1  class Bilesen {
2      Bilesen() {
▶          System.out.println("Bilesen kuruluyor");
4          ciz();
5      }
6      void ciz() { System.out.println("boş bileşen"); }
7  }
8  class Etiket extends Bilesen {
9      private String metin = "Merhaba";
10     Etiket() { System.out.println("Etiket kuruldu"); }
11     @Override void ciz() {
12         System.out.println("Etiket: " + metin);
13     }
14 }
15 Etiket e = new Etiket();   // main
16 e.ciz();
```

Yığın (stack)

- `Bilesen()`
- `Etiket()`
- `main()` — e = ?

Heap

- Etiket #1 — metin = null

`üst sınıf kısmı kuruluyor`

`Bilesen()` gövdesi çalışır ve ilk satırı yazar.

Adım 2 / 6

ÇIKTI

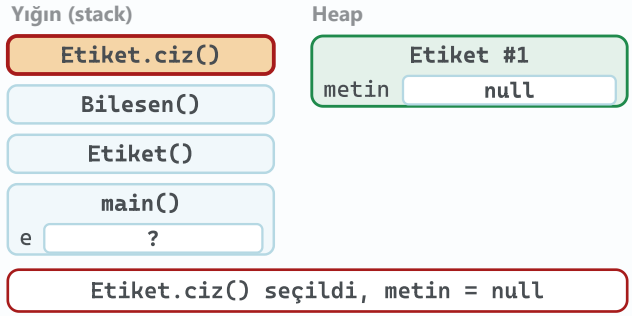
```
Bilesen kuruluyor
```

DEĞİŞKENLER

`metin = null`

Yapıcıdaki Çağrı: Kod İzleme

```
 1 class Bilesen {
 2     Bilesen() {
 3         System.out.println("Bilesen kuruluyor");
 4         ciz();
 5     }
 6     void ciz() { System.out.println("boş bileşen"); }
 7 }
 8 class Etiket extends Bilesen {
 9     private String metin = "Merhaba";
10     Etiket() { System.out.println("Etiket kuruldu"); }
11     @Override void ciz() {
12         System.out.println("Etiket: " + metin);
13     }
14 }
15 Etiket e = new Etiket();   // main
16 e.ciz();
```



Etiket.ciz() seçildi, metin = null

`ciz()` = `this.ciz()`. Nesnenin sınıfı `Etiket` olduğu için geçersiz kılınmış sürüm seçilir; ama `metin` henüz atanmadı: `Etiket: null`.

Adım 3 / 6

ÇIKTI

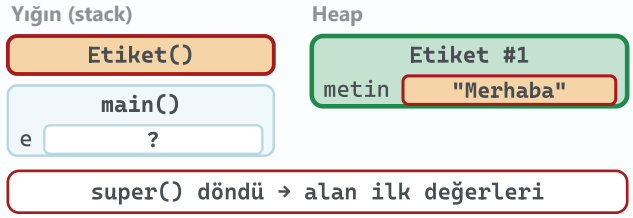
```
Bilesen kuruluyor
Etiket: null
```

DEĞİŞKENLER

`metin = null`

Yapıcıdaki Çağrı: Kod İzleme

```
 1 class Bilesen {
 2     Bilesen() {
 3         System.out.println("Bilesen kuruluyor");
 4         ciz();
 5     }
 6     void ciz() { System.out.println("boş bileşen"); }
 7 }
 8 class Etiket extends Bilesen {
 9     private String metin = "Merhaba";
10     Etiket() { System.out.println("Etiket kuruldu"); }
11     @Override void ciz() {
12         System.out.println("Etiket: " + metin);
13     }
14 }
15 Etiket e = new Etiket();   // main
16 e.ciz();
```



Üst sınıf yapıcısı döndükten sonra alt sınıfın alan ilk değeri çalışır: `metin = "Merhaba"`. Çok geç: `ciz()` çoktan çağrıldı.

Adım 4 / 6

ÇIKTI

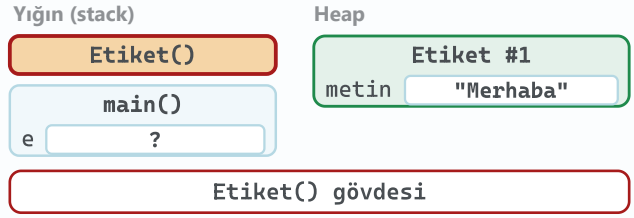
```
Bilesen kuruluyor
Etiket: null
```

DEĞİŞKENLER

`metin = "Merhaba"`

Yapıcıdaki Çağrı: Kod İzleme

```
 1 class Bilesen {
 2     Bilesen() {
 3         System.out.println("Bilesen kuruluyor");
 4         ciz();
 5     }
 6     void ciz() { System.out.println("boş bileşen"); }
 7 }
 8 class Etiket extends Bilesen {
 9     private String metin = "Merhaba";
▶11     Etiket() { System.out.println("Etiket kuruldu"); }
11     @Override void ciz() {
12         System.out.println("Etiket: " + metin);
13     }
14 }
15 Etiket e = new Etiket();   // main
16 e.ciz();
```



`Etiket()` gövdesi çalışır; nesne artık tamamen kuruldu.

Adım 5 / 6

ÇIKTI

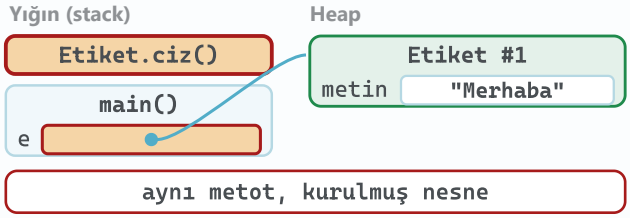
```
Bilesen kuruluyor
Etiket: null
Etiket kuruldu
```

DEĞİŞKENLER

`metin = "Merhaba"`

Yapıcıdaki Çağrı: Kod İzleme

```
 1 class Bilesen {
 2     Bilesen() {
 3         System.out.println("Bilesen kuruluyor");
 4         ciz();
 5     }
 6     void ciz() { System.out.println("boş bileşen"); }
 7 }
 8 class Etiket extends Bilesen {
 9     private String metin = "Merhaba";
10     Etiket() { System.out.println("Etiket kuruldu"); }
11     @Override void ciz() {
▶           System.out.println("Etiket: " + metin);
13     }
14 }
15 Etiket e = new Etiket();   // main
▶ e.ciz();
```



Kurulmuş nesnede aynı metot doğru sonucu verir. Fark yalnızca çağrının **ne zaman** yapıldığıdır.

Adım 6 / 6

ÇIKTI

```
Bilesen kuruluyor
Etiket: null
Etiket kuruldu
Etiket: Merhaba
```

DEĞİŞKENLER

`metin = "Merhaba"`

Aşırı Yükleme Derleme Zamanında Seçilir

```
static void yazdir(Hayvan h) {
    System.out.println("yazdir(Hayvan)");
}

static void yazdir(Kopek k) {
    System.out.println("yazdir(Kopek)");
}

public static void main(String[] args) {
    Kopek kopek = new Kopek();
    Hayvan hayvan = kopek;
    yazdir(kopek);
    yazdir(hayvan);
    yazdir((Kopek) hayvan);
}
```

```
yazdir(Kopek)
yazdir(Hayvan)
yazdir(Kopek)
```

- `hayvan` bir `Kopek`'i gösterse de `yazdir(hayvan)` → **`yazdir(Hayvan)`**.
- Aşırı yüklenmiş sürümler arasındaki seçim argümanın **bildirilen tipine** göre, derleyici tarafından yapılır.
- Dönüşüm bildirilen tipi değiştirir: `(Kopek) hayvan` → `yazdir(Kopek)`.

	Aşırı yükleme	**Geçersiz kılma**
Nerede?	Aynı sınıf	Alt sınıf
İmza	Farklı parametre	Aynı imza
Seçim	Derleyici	JVM
Ölçüt	Bildirilen tip	Gerçek tip

Aşırı Yükleme: En Özel Sürüm ve Belirsizlik

```
static void goster(Object o) { System.out.println("goster(Object)"); }
static void goster(Hayvan h) { System.out.println("goster(Hayvan)"); }
static void goster(String s) { System.out.println("goster(String)"); }
// main içinde:
goster(new Kedi());
goster("a");
Object o = "a";
goster(o);
goster((Hayvan) null);
```

```
goster(Hayvan)
goster(String)
goster(Object)
goster(Hayvan)
```

`goster(null);`

```
error: reference to goster is ambiguous
  both method goster(Hayvan) in Ana and
  method goster(String) in Ana match
```

- Birden çok sürüm uyarsa derleyici **en özel** olanı seçer: `Kedi` hem `Hayvan` 'a hem `Object` 'e uyar; `Hayvan` daha özeldir.
- `null` her başvuru tipine uyar. `Hayvan` ile `String` arasında kalıtım ilişkisi olmadığı için hiçbiri ötekinden özel değildir: **belirsiz** çağrı, derleme hatası.
- Uyan sürümler tek bir zincir üzerindeyse (`Object` , `Hayvan` , `Kedi`) belirsizlik olmaz: `goster(null)` en alttakini, `goster(Kedi)` 'yi seçer.
- Çözüm, ifadenin bildirilen tipini açık dönüşümle belirlemektir: `goster((Hayvan) null)` .

Derleyici mi, JVM mi? Kod İzleme

```
1  static void yazdir(Hayvan h) {
2      System.out.print("yazdir(Hayvan): ");
3      h.sesCikar();
4  }
5  static void yazdir(Kopek k) {
6      System.out.print("yazdir(Kopek): ");
7      k.sesCikar();
8  }
9  // main içinde:
▶  Hayvan hayvan = new Kopek();
▶  Hayvan diger = new Kedi();
12 yazdir(hayvan);
13 yazdir(diger);
14 yazdir((Kopek) hayvan);
```

Derleyici bildirilen tipe bakar	JVM nesnenin sınıfına bakar
`yazdir(Hayvan)`	`Kopek.sesCikar()`
`yazdir(Kopek)`	`Kedi.sesCikar()`
derleme zamanı	çalışma zamanı

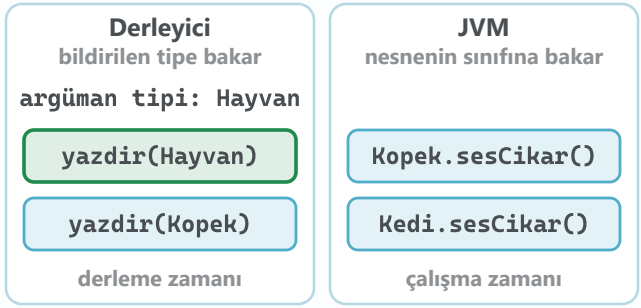
İki değişkenin bildirilen tipi aynı (`Hayvan`); gösterdikleri nesneler farklı: `Kopek` ve `Kedi` .

Adım 1 / 7

ÇIKTI

Derleyici mi, JVM mi? Kod İzleme

```
▶ static void yazdir(Hayvan h) {
▶     System.out.print("yazdir(Hayvan): ");
3     h.sesCikar();
4 }
5 static void yazdir(Kopek k) {
6     System.out.print("yazdir(Kopek): ");
7     k.sesCikar();
8 }
9 // main içinde:
10 Hayvan hayvan = new Kopek();
11 Hayvan diger = new Kedi();
▶ yazdir(hayvan);
13 yazdir(diger);
14 yazdir((Kopek) hayvan);
```



`hayvan` bir `Kopek`'i gösterse de derleyici yalnızca bildirilen tipi bilir: `yazdir(Hayvan)`.

Adım 2 / 7

ÇIKTI

```
yazdir(Hayvan):
```

Derleyici mi, JVM mi? Kod İzleme

```
1  static void yazdir(Hayvan h) {
2      System.out.print("yazdir(Hayvan): ");
▶      h.sesCikar();
4  }
5  static void yazdir(Kopek k) {
6      System.out.print("yazdir(Kopek): ");
7      k.sesCikar();
8  }
9  // main içinde:
10 Hayvan hayvan = new Kopek();
11 Hayvan diger = new Kedi();
12 yazdir(hayvan);
13 yazdir(diger);
14 yazdir((Kopek) hayvan);
```

Derleyici bildirilen tipe bakar	JVM nesnenin sınıfına bakar
`argüman tipi: Hayvan`	`nesne: Kopek`
`yazdir(Hayvan)`	`Kopek.sesCikar()`
`yazdir(Kopek)`	`Kedi.sesCikar()`
derleme zamanı	çalışma zamanı

Metodun içinde `h.sesCikar()` geçersiz kılınmış bir metot: JVM nesneye bakar, **Hav hav!**

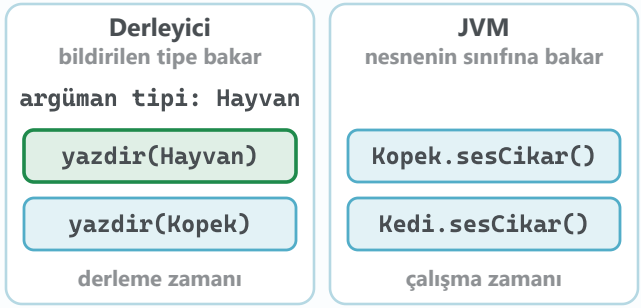
Adım 3 / 7

ÇIKTI

```
yazdir(Hayvan): Hav hav!
```

Derleyici mi, JVM mi? Kod İzleme

```
▶ static void yazdir(Hayvan h) {
▶     System.out.print("yazdir(Hayvan): ");
  3     h.sesCikar();
  4 }
  5 static void yazdir(Kopek k) {
  6     System.out.print("yazdir(Kopek): ");
  7     k.sesCikar();
  8 }
  9 // main içinde:
 10 Hayvan hayvan = new Kopek();
 11 Hayvan diger = new Kedi();
 12 yazdir(hayvan);
▶ yazdir(diger);
 14 yazdir((Kopek) hayvan);
```



`diger` da `Hayvan` tipinde: aynı sürüm seçilir.

Adım 4 / 7

ÇIKTI

```
yazdir(Hayvan): Hav hav!
yazdir(Hayvan):
```

Derleyici mi, JVM mi? Kod İzleme

```
1  static void yazdir(Hayvan h) {
2      System.out.print("yazdir(Hayvan): ");
▶      h.sesCikar();
4  }
5  static void yazdir(Kopek k) {
6      System.out.print("yazdir(Kopek): ");
7      k.sesCikar();
8  }
9  // main içinde:
10 Hayvan hayvan = new Kopek();
11 Hayvan diger = new Kedi();
12 yazdir(hayvan);
13 yazdir(diger);
14 yazdir((Kopek) hayvan);
```

Derleyici bildirilen tipe bakar	**JVM** nesnenin sınıfına bakar
`argüman tipi: Hayvan`	`nesne: Kedi`
`yazdir(Hayvan)`	`Kopek.sesCikar()`
`yazdir(Kopek)`	`Kedi.sesCikar()`
derleme zamanı	çalışma zamanı

Aynı satır, farklı nesne: bu kez `Kedi.sesCikar()`, **Miyav!**

Adım 5 / 7

ÇIKTI

```
yazdir(Hayvan): Hav hav!
yazdir(Hayvan): Miyav!
```

Derleyici mi, JVM mi? Kod İzleme

```
1  static void yazdir(Hayvan h) {
2      System.out.print("yazdir(Hayvan): ");
3      h.sesCikar();
4  }
▶  static void yazdir(Kopek k) {
▶      System.out.print("yazdir(Kopek): ");
7      k.sesCikar();
8  }
9  // main içinde:
10 Hayvan hayvan = new Kopek();
11 Hayvan diger = new Kedi();
12 yazdir(hayvan);
13 yazdir(diger);
▶  yazdir((Kopek) hayvan);
```



Açık dönüşüm ifadenin bildirilen tipini `Kopek` yapar: derleyici bu kez `yazdir(Kopek)`'u seçer.

Adım 6 / 7

ÇIKTI

```
yazdir(Hayvan): Hav hav!
yazdir(Hayvan): Miyav!
yazdir(Kopek):
```

Derleyici mi, JVM mi? Kod İzleme

```
1  static void yazdir(Hayvan h) {
2      System.out.print("yazdir(Hayvan): ");
3      h.sesCikar();
4  }
5  static void yazdir(Kopek k) {
6      System.out.print("yazdir(Kopek): ");
▶      k.sesCikar();
8  }
9  // main içinde:
10 Hayvan hayvan = new Kopek();
11 Hayvan diger = new Kedi();
12 yazdir(hayvan);
13 yazdir(diger);
14 yazdir((Kopek) hayvan);
```

Derleyici	JVM
bildirilen tipe bakar	nesnenin sınıfına bakar
argüman tipi: Kopek	nesne: Kopek
yazdir(Hayvan)	Kopek.sesCikar()
yazdir(Kopek)	Kedi.sesCikar()
derleme zamanı	çalışma zamanı

Aşırı yükleme seçimi bildirilen tiple **derlemede**, geçersiz kılma seçimi gerçek tiple **çalışırken** yapıldı.

Adım 7 / 7

ÇIKTI

```
yazdir(Hayvan): Hav hav!
yazdir(Hayvan): Miyav!
yazdir(Kopek): Hav hav!
```

AŞIRI YÜKLEME

Aşırı Yükleme Seçiminin Üç Aşaması

1. Kutulamasız

Yalnızca tam eşleşme, ilkel genişletme (`int` → `long`) ve başvuru genişletmesi (`String` → `Object`) denenir.

2. Kutulama

1. aşamada aday yoksa kutulama ve kutudan çıkarma da denenir: `double` → `Double` → `Object` .

3. Değişken sayıda

Hâlâ aday yoksa `int...` gibi değişken sayıda parametreli sürümler devreye girer.

- Bir aşamada en az bir aday bulunursa sonraki aşamalara **geçilmez**; o aşamanın adayları arasından en özel olan seçilir.
- Sonuç: `f(long)` ve `f(Integer)` varken `f(5)` çağrısı `f(long)` 'u seçer. Genişletme kutulamadan önce gelir, çünkü kutulama Java 5'te eklendi ve eski programların davranışı değişmesin istendi.
- Daraltma (`long` → `int`) hiçbir aşamada denenmez; üç aşamada da aday yoksa: `no suitable method found for yazdir(boolean)` .
- Bu sıra derleme zamanında uygulanır; seçilen **imza** bayt koduna yazılır. Seçilen imzanın hangi sınıftaki gövdesinin çalışacağı ise çalışma zamanında belirlenir.

Aşırı Yükleme Aşamaları: Kod İzleme

```
1  static void f(long x)    { p("f(long)"); }
2  static void f(Integer x) { p("f(Integer)"); }
3  static void f(Object x)  { p("f(Object)"); }
4  static void f(int... x)  { p("f(int...)"); }
5  // main içinde (p: System.out.println):
6  byte b = 1;
▶  f(5);
8  f(b);
9  f(Integer.valueOf(5));
10 f("x");
11 f(2.5);
12 f();
13 f(1, 2);
```

Argüman: 5 (int)

1. kutulamasız	2. kutulama	3. değişken sayıda

`f(long)`	✓ int → long genişletme
`f(Integer)`	✗ kutulama gerekir
`f(Object)`	✗ kutulama gerekir
`f(int ... )`	

`1. aşamada tek aday: f(long)`

`5` bir `int`. 1. aşamada yalnızca genişletme ile `long` uyar; `Integer` ve `Object` kutulama ister, bu aşamada denenmez.

Adım 1 / 6

ÇIKTI

```
f(long)
```

Aşırı Yükleme Aşamaları: Kod İzleme

```
1  static void f(long x)    { p("f(long)"); }
2  static void f(Integer x) { p("f(Integer)"); }
3  static void f(Object x)  { p("f(Object)"); }
4  static void f(int... x)  { p("f(int...)"); }
5  // main içinde (p: System.out.println):
6  byte b = 1;
7  f(5);
▶ f(b);
9  f(Integer.valueOf(5));
10 f("x");
11 f(2.5);
12 f();
13 f(1, 2);
```

Argüman: b (byte)

1. kutulamasız	2. kutulama	3. değişken sayıda

`f(long)`	✓ byte → long genişletme
`f(Integer)`	✗ Byte ≠ Integer
`f(Object)`	✗ kutulama gerekir
`f(int ... )`	

`yine f(long)`

`byte` da `long` 'a genişler. Kutulansaydı bile `Byte` olurdu, `Integer` değil.

Adım 2 / 6

ÇIKTI

```
f(long)
f(long)
```

Aşırı Yükleme Aşamaları: Kod İzleme

```
1  static void f(long x)    { p("f(long)"); }
2  static void f(Integer x) { p("f(Integer)"); }
3  static void f(Object x)  { p("f(Object)"); }
4  static void f(int... x)  { p("f(int...)"); }
5  // main içinde (p: System.out.println):
6  byte b = 1;
7  f(5);
8  f(b);
▶  f(Integer.valueOf(5));
10 f("x");
11 f(2.5);
12 f();
13 f(1, 2);
```

Argüman: Integer

1. kutulamasız	2. kutulama	3. değişken sayıda

`f(long)`	✗ kutudan çıkarma gerekir
`f(Integer)`	✓ tam eşleşme
`f(Object)`	✓ Integer → Object
`f(int ... )`	

`iki aday: en özel olan Integer`

Argüman zaten bir `Integer` : 1. aşamada iki aday var. `Integer` , `Object` 'ten özel olduğu için seçilir.

Adım 3 / 6

ÇIKTI

```
f(long)
f(long)
f(Integer)
```

Aşırı Yükleme Aşamaları: Kod İzleme

```
1  static void f(long x)    { p("f(long)"); }
2  static void f(Integer x) { p("f(Integer)"); }
3  static void f(Object x)  { p("f(Object)"); }
4  static void f(int... x)  { p("f(int...)"); }
5  // main içinde (p: System.out.println):
6  byte b = 1;
7  f(5);
8  f(b);
9  f(Integer.valueOf(5));
▶  f("x");
11 f(2.5);
12 f();
13 f(1, 2);
```

Argüman: "x" (String)

1. kutulamasız	2. kutulama	3. değişken sayıda

`f(long)`	✗
`f(Integer)`	✗
`f(Object)`	✓ String → Object
`f(int ... )`	

`başvuru genişletmesi: f(Object)`

`String` yalnızca `Object`'e uyar.

Adım 4 / 6

ÇIKTI

```
f(long)
f(long)
f(Integer)
f(Object)
```

Aşırı Yükleme Aşamaları: Kod İzleme

```
1  static void f(long x)    { p("f(long)"); }
2  static void f(Integer x) { p("f(Integer)"); }
3  static void f(Object x)  { p("f(Object)"); }
4  static void f(int... x)  { p("f(int...)"); }
5  // main içinde (p: System.out.println):
6  byte b = 1;
7  f(5);
8  f(b);
9  f(Integer.valueOf(5));
10 f("x");
▶  f(2.5);
12 f();
13 f(1, 2);
```

Argüman: 2.5 (double)

1. kutulamasız	2. kutulama	3. değişken sayıda

`f(long)`	✗ double → long daraltma
`f(Integer)`	✗ Double ≠ Integer
`f(Object)`	✓ double → Double → Object
`f(int ... )`	

`1. aşama boş → 2. aşamada kutulama`

`double` , `long` 'a daraltılamaz; 1. aşamada aday yok. 2. aşamada `Double` 'a kutulanır ve `Object` 'e uyar.

Adım 5 / 6

ÇIKTI

```
f(long)
f(long)
f(Integer)
f(Object)
f(Object)
```

Aşırı Yükleme Aşamaları: Kod İzleme

```
1  static void f(long x)    { p("f(long)"); }
2  static void f(Integer x) { p("f(Integer)"); }
3  static void f(Object x)  { p("f(Object)"); }
4  static void f(int... x)  { p("f(int...)"); }
5  // main içinde (p: System.out.println):
6  byte b = 1;
7  f(5);
8  f(b);
9  f(Integer.valueOf(5));
10 f("x");
11 f(2.5);
▶  f();
▶  f(1, 2);
```

Argüman: () ve (1, 2)

1. kutulamasız	2. kutulama	3. değişken sayıda

`f(long)`	✗ parametre sayısı
`f(Integer)`	✗ parametre sayısı
`f(Object)`	✗ parametre sayısı
`f(int ... )`	✓ 0 ya da 2 int

`yalnızca 3. aşamada aday var`

Sıfır ya da iki argümanı yalnızca `int...` karşılar; ona en son, 3. aşamada başvurulur.

Adım 6 / 6

ÇIKTI

```
f(long)
f(long)
f(Integer)
f(Object)
f(Object)
f(int...)
f(int...)
```

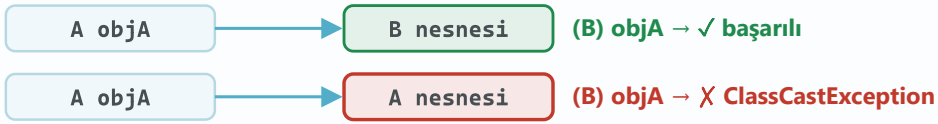
Kalıtım ve Çok Biçimlilik

- Kalıtım, **sınıf hiyerarşileri** oluşturmak için gereklidir.
- Çok biçimlilik için kalıtım **gereklidir**: bir sınıf hiyerarşisine sahip olmalıyız.
- Kalıtımı kullanmak için çok biçimlilik **gerekli değildir**: yalnızca kod yeniden kullanımı için de kalıtım yapılabilir.
- B sınıfı A'nın alt sınıfıysa, **B'nin nesnesi A'nın nesnesi gibi davranabilir** (tersi değil).
- A referansını B referansına atamak için **daraltma dönüşümü** (casting) gerekir; parantezle yapılır.

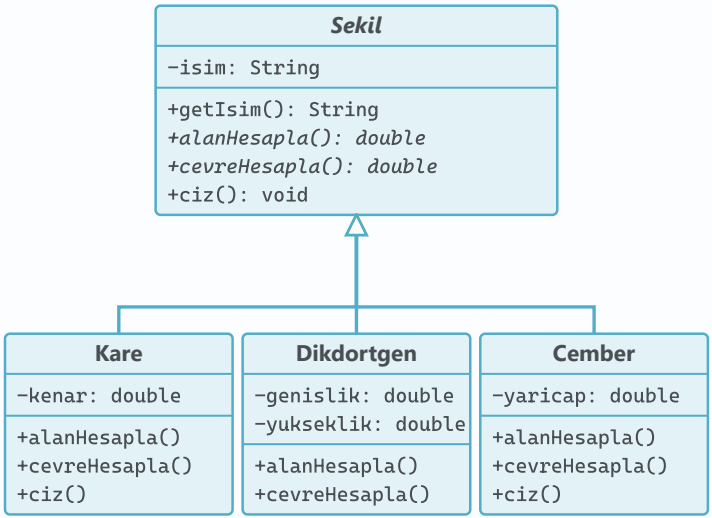
```
A objA = new B(); // B, A gibi davranır
B objB = (B) objA; // geçerli, ancak
                   // dikkatli kullanılmalı
```

Son satır derlenir; çalışırken `objA` gerçekten bir `B` nesnesini göstermiyorsa `ClassCastException` fırlatılır.

Java'da arayüzler (interface) de çok biçimlilik sağlar: `implements` eden her sınıf, arayüz tipindeki referansa atanabilir.



Örnek: Şekil Hiyerarşisi




```java
abstract class Sekil {
    private String isim;

    public Sekil(String isim) {
        this.isim = isim;
    }

    public String getIsim() {
        return isim;
    }

    public abstract double alanHesapla();

    public abstract double cevreHesapla();

    public void ciz() {
        System.out.println(isim + " çiziliyor.");
    }
}
```

`abstract` : gövdesiz metot; her alt sınıf yazmak **zorundadır**, `new Sekil(…)` yapılamaz.

Alt Sınıflar Metotları Geçersiz Kılar

```
class Kare extends Sekil {
    private double kenar;
    public Kare(double kenar) {
        super("Kare");
        this.kenar = kenar;
    }
    @Override
    public double alanHesapla() {
        return kenar * kenar;
    }
    @Override
    public double cevreHesapla() {
        return 4 * kenar;
    }
    @Override
    public void ciz() {
        System.out.println("Kare çiziliyor: kenar " + kenar);
    }
}
```

`Kare` : yapıcı `super("Kare")` ile ismi üst sınıfa verir; üç metodu da geçersiz kılar.

Alt Sınıflar Metotları Geçersiz Kılar

```
class Dikdortgen extends Sekil {
    private double genislik;
    private double yukseklik;
    public Dikdortgen(double genislik, double yukseklik) {
        super("Dikdörtgen");
        this.genislik = genislik;
        this.yukseklik = yukseklik;
    }
    @Override
    public double alanHesapla() {
        return genislik * yukseklik;
    }
    @Override
    public double cevreHesapla() {
        return 2 * (genislik + yukseklik);
    }
    // ciz() ezilmedi → Sekil'inki kullanılır
}
```

`Dikdortgen` : alan ve çevreyi geçersiz kılar, `ciz()` 'i geçersiz kılmaz → `Sekil.ciz()` kullanılır.

Alt Sınıflar Metotları Geçersiz Kılar

```
class Cember extends Sekil {
    private double yaricap;
    public Cember(double yaricap) {
        super("Çember");
        this.yaricap = yaricap;
    }
    @Override
    public double alanHesapla() {
        return Math.PI * yaricap * yaricap;
    }
    @Override
    public double cevreHesapla() {
        return 2 * Math.PI * yaricap;
    }
    @Override
    public void ciz() {
        System.out.println("Çember çiziliyor: yarıçap " + yaricap);
    }
}
```

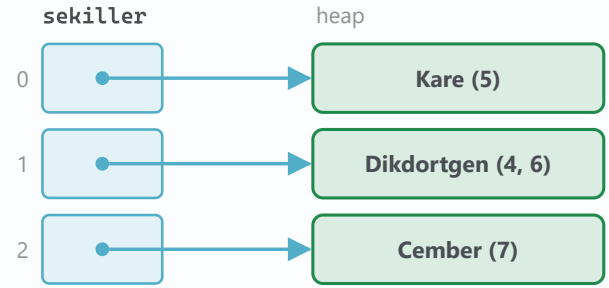
`Cember` : `Math.PI` ile hesaplar; `ciz()` 'i kendi bilgisiyle geçersiz kılar.

Şekiller Dizisi: Adım Adım Kod İzleme

```
1  Sekil[] sekiller = new Sekil[3];
2  sekiller[0] = new Kare(5);
3  sekiller[1] = new Dikdortgen(4, 6);
▶  sekiller[2] = new Cember(7);
5
6  for (Sekil sekil : sekiller) {
7      System.out.println(sekil.getIsim()
8          + " alanı: " + sekil.alanHesapla());
9      sekil.ciz();
10 }
```

ÇIKTI





Dizi `Sekil` tipinde; öğeler üç farklı alt sınıf nesnesini gösterir.

Adım 1 / 7

ÇAĞRI YIĞINI

`main()`

DEĞİŞKENLER

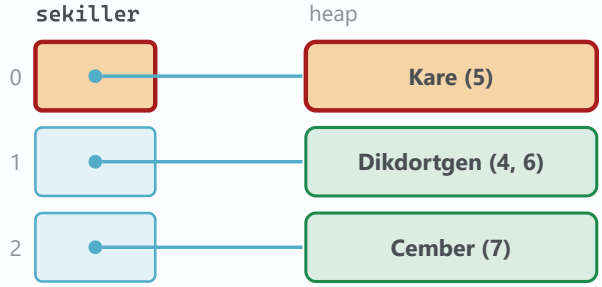
`sekil = –`

Şekiller Dizisi: Adım Adım Kod İzleme

```
1  Sekil[] sekiller = new Sekil[3];
2  sekiller[0] = new Kare(5);
3  sekiller[1] = new Dikdortgen(4, 6);
4  sekiller[2] = new Cember(7);
5
6  for (Sekil sekil : sekiller) {
7      System.out.println(sekil.getIsim()
▶          + " alanı: " + sekil.alanHesapla());
9      sekil.ciz();
10 }
```

ÇIKTI

```
Kare alanı: 25.0
```



`sekil.alanHesapla()` → `Kare.alanHesapla()` çalışır.

Adım 2 / 7

ÇAĞRI YIĞINI

- `Kare.alanHesapla()`
- `main()`

DEĞİŞKENLER

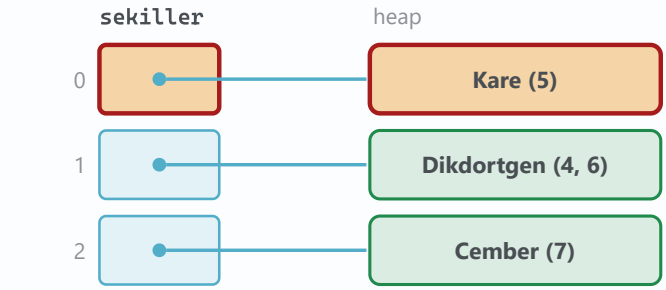
`sekil = → Kare (5)`

Şekiller Dizisi: Adım Adım Kod İzleme

```
1  Sekil[] sekiller = new Sekil[3];
2  sekiller[0] = new Kare(5);
3  sekiller[1] = new Dikdortgen(4, 6);
4  sekiller[2] = new Cember(7);
5
6  for (Sekil sekil : sekiller) {
7      System.out.println(sekil.getIsim()
8          + " alanı: " + sekil.alanHesapla());
▶      sekil.ciz();
10 }
```

ÇIKTI

```
Kare alanı: 25.0
Kare çiziliyor: kenar 5.0
```



`ciz()` geçersiz kılınmış → `Kare.ciz()` çalışır.

Adım 3 / 7

ÇAĞRI YIĞINI

- `Kare.ciz()`
- `main()`

DEĞİŞKENLER

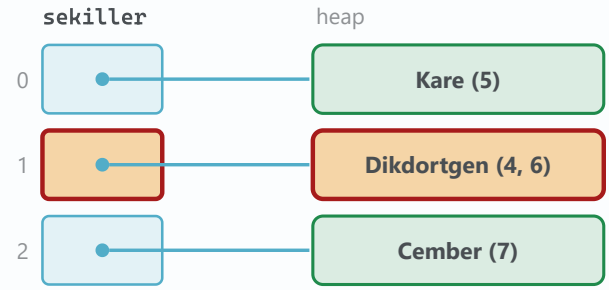
`sekil` = → **Kare (5)**

Şekiller Dizisi: Adım Adım Kod İzleme

```
1  Sekil[] sekiller = new Sekil[3];
2  sekiller[0] = new Kare(5);
3  sekiller[1] = new Dikdortgen(4, 6);
4  sekiller[2] = new Cember(7);
5
6  for (Sekil sekil : sekiller) {
7      System.out.println(sekil.getIsim()
▶          + " alanı: " + sekil.alanHesapla());
9      sekil.ciz();
10 }
```

ÇIKTI

```
Kare alanı: 25.0
Kare çiziliyor: kenar 5.0
Dikdörtgen alanı: 24.0
```



`sekil.alanHesapla()` → `Dikdortgen.alanHesapla()` çalışır.

Adım 4 / 7

ÇAĞRI YIĞINI

- **`Dikdortgen.alanHesapla()`**
- `main()`

DEĞİŞKENLER

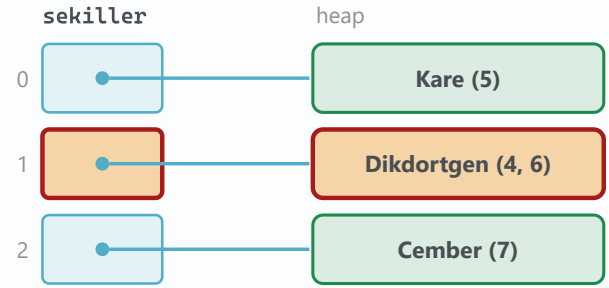
`sekil` = → **`Dikdortgen (4, 6)`**

Şekiller Dizisi: Adım Adım Kod İzleme

```
1  Sekil[] sekiller = new Sekil[3];
2  sekiller[0] = new Kare(5);
3  sekiller[1] = new Dikdortgen(4, 6);
4  sekiller[2] = new Cember(7);
5
6  for (Sekil sekil : sekiller) {
7      System.out.println(sekil.getIsim()
8          + " alanı: " + sekil.alanHesapla());
▶      sekil.ciz();
10 }
```

ÇIKTI

```
Kare alanı: 25.0
Kare çiziliyor: kenar 5.0
Dikdörtgen alanı: 24.0
Dikdörtgen çiziliyor.
```



`Dikdortgen` `ciz()` 'i geçersiz kılmadı → arama yukarı çıkar, `Sekil.ciz()` çalışır.

Adım 5 / 7

ÇAĞRI YIĞINI

- **`Sekil.ciz()`**
- `main()`

DEĞİŞKENLER

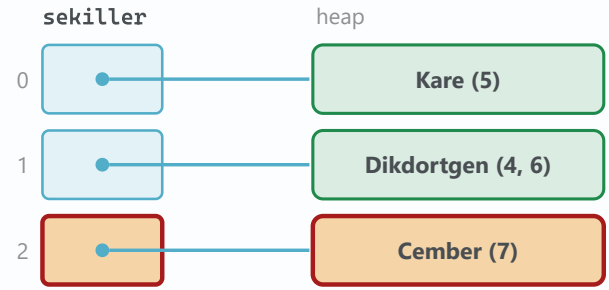
`sekil` = → **`Dikdortgen (4, 6)`**

Şekiller Dizisi: Adım Adım Kod İzleme

```
1  Sekil[] sekiller = new Sekil[3];
2  sekiller[0] = new Kare(5);
3  sekiller[1] = new Dikdortgen(4, 6);
4  sekiller[2] = new Cember(7);
5
6  for (Sekil sekil : sekiller) {
7      System.out.println(sekil.getIsim()
           + " alanı: " + sekil.alanHesapla());
9      sekil.ciz();
10 }
```

ÇIKTI

```
Kare alanı: 25.0
Kare çiziliyor: kenar 5.0
Dikdörtgen alanı: 24.0
Dikdörtgen çiziliyor.
Çember alanı: 153.93804002589985
```



`sekil.alanHesapla()` → `Cember.alanHesapla()` çalışır.

Adım 6 / 7

ÇAĞRI YIĞINI

- `Cember.alanHesapla()`
- `main()`

DEĞİŞKENLER

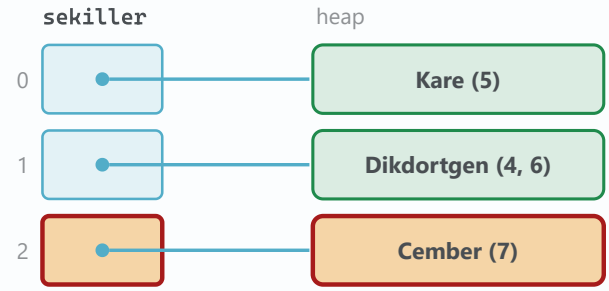
`sekil = → Cember (7)`

Şekiller Dizisi: Adım Adım Kod İzleme

```
1  Sekil[] sekiller = new Sekil[3];
2  sekiller[0] = new Kare(5);
3  sekiller[1] = new Dikdortgen(4, 6);
4  sekiller[2] = new Cember(7);
5
6  for (Sekil sekil : sekiller) {
7      System.out.println(sekil.getIsim()
8          + " alanı: " + sekil.alanHesapla());
▶      sekil.ciz();
10 }
```

ÇIKTI

```
Kare alanı: 25.0
Kare çiziliyor: kenar 5.0
Dikdörtgen alanı: 24.0
Dikdörtgen çiziliyor.
Çember alanı: 153.93804002589985
Çember çiziliyor: yarıçap 7.0
```



`ciz()` geçersiz kılınmış → `Cember.ciz()` çalışır.

Adım 7 / 7

ÇAĞRI YIĞINI

- `Cember.ciz()`
- `main()`

DEĞİŞKENLER

`sekil = → Cember (7)`

Neden Çok Biçimlilik?

Tip sınayarak

```
static double alanKotu(Sekil s) {
    double alan;
    if (s instanceof Kare) {
        alan = ((Kare) s).alanHesapla();
    } else if (s instanceof Dikdortgen) {
        alan = ((Dikdortgen) s).alanHesapla();
    } else if (s instanceof Cember) {
        alan = ((Cember) s).alanHesapla();
    } else {
        alan = 0; // yeni tür unutuldu!
    }
    return alan;
}
```

Yeni bir şekil eklenince bu metot da değişmeli; unutulursa `alanKotu(new DikUcgen(3, 4))` sessizce **0.0** döner.

Çok biçimli

```
static double toplamAlan(Sekil[] dizi) {
    double toplam = 0;
    for (Sekil sekil : dizi) {
        // her nesne kendi alanını
        toplam += sekil.alanHesapla();
    }
    return toplam;
}
```

+ Yeni `DikUcgen extends Sekil` sınıfı eklemek yeter; bu kod **hiç değişmez**.
+ { Kare(5), Dikdortgen(4, 6), DikUcgen(3, 4) } → **55.0**
+ Kod kısalır, `ClassCastException` riski ortadan kalkar.

İlke: genişletmeye açık, değiştirmeye kapalı kod (open/closed principle).

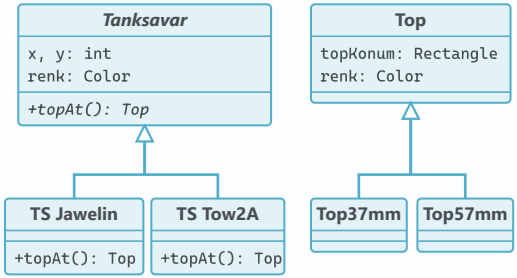
Çok Biçimli Parametre ve Dönüş Değeri

```
class Veteriner {
    void muayene(Hayvan h) {     // her alt sınıf
        System.out.print("Muayene: ");
        h.sesCikar();
    }
    static Hayvan sahiplen(String tur) { // gizli
        if (tur.equals("kedi")) return new Kedi();
        return new Kopek();
    }
}
// main içinde:
Veteriner v = new Veteriner();
v.muayene(new Kedi());
Hayvan h = Veteriner.sahiplen("köpek");
v.muayene(h);
System.out.println(h.getClass().getSimpleName());
```

```
Muayene: Miyav!
Muayene: Hav hav!
Kopek
```

- **Parametre** üst sınıf tipinde tanımlanırsa metot, bugün var olan ve ileride yazılacak her alt sınıfın nesnesini kabul eder. `muayene` yeni bir `Kus` sınıfı için değişmez.
- **Dönüş tipi** üst sınıf olursa metot hangi alt sınıfı ürettiğini gizler. Çağıran yalnızca `Hayvan` sözleşmesine güvenir; `sahiplen` başka bir alt sınıf döndürmeye başlasa kullanan kod etkilenmez.
- Nesneleri üreten bu tür metotlara **fabrika metodu** (factory method) denir.
- Genel ilke: parametre ve değişken tipleri, işi görmeye yeten **en genel** tip olarak seçilir.

Örnek: Tanksavar Oyunu



TS Jawelin: `TanksavarJawelin`, TS Tow2A: `TanksavarTow2A`. `Tank` sınıfının da `TankT90MS` ve `TankLeopard2A4` alt sınıfları vardır.

- Swing ile yazılmış oyunda düşen tanklar vurulur; skor 10'un katı olunca tanksavar türü değişir.
- Oyun döngüsü yalnızca `Tanksavar` ve `Tank` tiplerini bilir.

```
Tanksavar tanksavar = new TanksavarJawelin(x, y);
...
// hangi top? nesneye bağlı (dinamik bağlama)
toplar.add(tanksavar.topAt());
...
if (tanksavar instanceof TanksavarJawelin) {
    tanksavar = new TanksavarTow2A(tanksavar.x, tanksavar.y);
}
```

SINAMA ÇIKTISI

```
Top t1 = tanksavar.topAt(); // Jawelin
// → Top37mm, yükseklik 30
tanksavar = new TanksavarTow2A(100, 540);
Top t2 = tanksavar.topAt();
// → Top57mm, yükseklik 40
```

Aynı `topAt()` çağrısı, tanksavar değişince farklı bir top üretir; oyun kodunda hiçbir `if` gerekmez.

`equals` ve Kalıtım: Simetri Tuzağı

`Object` metotları da dinamik bağlanır: `Object o = k1; System.out.println(o);` nesnenin kendi `toString` 'ini, `o.equals(x)` kendi `equals` 'ını çalıştırır.

```
class Nokta {
    final int x, y;
    Nokta(int x, int y) { this.x = x; this.y = y; }
    @Override public boolean equals(Object o) {
        if (!(o instanceof Nokta)) return false;
        Nokta n = (Nokta) o;
        return x == n.x && y == n.y;
    }
    @Override public int hashCode() { return 31 * x + y; }
}
class RenkliNokta extends Nokta {
    final String renk;
    RenkliNokta(int x, int y, String renk) {
        super(x, y);
        this.renk = renk;
    }
    @Override public boolean equals(Object o) {
        if (!(o instanceof RenkliNokta)) return false;
        RenkliNokta r = (RenkliNokta) o;
        return super.equals(r) && renk.equals(r.renk);
    }
}
```

```
Nokta p = new Nokta(1, 2);
RenkliNokta k =
    new RenkliNokta(1, 2, "kırmızı");
System.out.println(p.equals(k)
    + " " + k.equals(p));
```

```
true false
```

- Hangi `equals` 'ın çalışacağını **alıcı nesne** belirler: `p.equals(k)` `Nokta` 'nınkini, `k.equals(p)` `RenkliNokta` 'nınkini çalıştırır.
- Sonuçlar farklı: simetri bozuldu. Bir `HashSet` ya da `contains`, elemanların sırasına göre farklı cevap verebilir.
- Çözümler: iki sınıfta da `getClass()` karşılaştırması; ya da kalıtım yerine oluşum (`RenkliNokta` bir `Nokta` alanı tutar).

ÖZET

Özet

Kavram	Kural	Örnek
Referans kuralı	Üst sınıf referansı alt sınıf nesnesini gösterebilir; tersi derlenmez	`Hayvan h = new Kopek();`
is-a	Alt sınıf **bir** üst sınıftır; tersi değil	`h instanceof Hayvan` → true
Yukarı dönüşüm	Otomatik ve her zaman güvenli	`Object o = kopek;`
Aşağı dönüşüm	Açık yazılır; nesne uygun değilse `ClassCastException`	`Kopek k = (Kopek) h;`
Erişim	Hangi metotların çağrılabileceğini **referans tipi** belirler	`h.havla()` → derleme hatası
Dinamik bağlama	Hangi sürümün çalışacağını **nesne tipi** belirler; arama alttan yukarı	`h.kos()` → Kopek sürümü
Statik bağlama	Alanlar, `static` / `private` / `final` metotlar ve aşırı yükleme seçimi	`h.tur` → "hayvan"
Fayda	Üst sınıf tipiyle bir kez yazılan kod, yeni alt sınıflarla değişmeden çalışır	`toplamAlan(sekiller)`

ÖZET

İlgili Sayfalar

Dinamik Bağlama

Simülatör: üst sınıf tipindeki referansla yapılan çağrıda metot aramasının nesnenin gerçek sınıfından başlayıp yukarı çıkması.

Soyut Sınıflar ve Arayüzler

Arayüz tanımlama ve `implements`, arayüz tipindeki referanslarla çok biçimlilik, çoklu arayüz, `default` / `static` / `private` arayüz metotları, jenerikler.

Arayüzlerle bağlantısı

Bir sınıf tek bir üst sınıftan türeyebilir, ama birçok arayüzü uygulayabilir: `Hayvan h` yerine `SesCikarabilen s` gibi arayüz tipleriyle de aynı dinamik bağlama çalışır.

ÖZET

Sık Yapılan Hatalar (1): Derleme Hataları

Kod	Derleyici mesajı	Neden / çözüm
`Kopek k = new Hayvan();`	*incompatible types: Hayvan cannot be converted to Kopek*	Her hayvan köpek değildir; ters atama yapılamaz.
`Kopek k = hayvan;`	*incompatible types: Hayvan cannot be converted to Kopek*	Nesne köpek olsa bile açık dönüşüm gerekir: `(Kopek) hayvan`.
`hayvan.havla();`	*cannot find symbol … variable hayvan of type Hayvan*	Referans tipinde metot yok; önce aşağı dönüşüm gerekir.
`(Kopek) h.havla();`	*not a statement*	Parantez eksik: `((Kopek) h).havla();`
`(Kedi) kopek`	*incompatible types: Kopek cannot be converted to Kedi*	Kardeş sınıflar arasında dönüşüm yoktur.
`@Override void sescikar()`	*method does not override or implement a method from a supertype*	Ad yanlış yazıldı (büyük/küçük harf).
`void sesCikar()` (üstte `public`)	*attempting to assign weaker access privileges; was public*	Geçersiz kılan metot erişimi daraltamaz.

ÖZET

Sık Yapılan Hatalar (2): Çalışma Zamanı ve Mantık

Denetimsiz aşağı dönüşüm

`(Kopek) h` yazıp `h`'nin bir `Kedi` gösterebileceğini unutmak → `ClassCastException`. Önce `instanceof`.

"Dönüşüm nesneyi değiştirir" sanmak

`(Hayvan) kopek` köpeği hayvana çevirmez; `sesCikar()` yine "Hav hav!" yazar.

Alanların da geçersiz kılındığını sanmak

`h.tur` referans tipine göre okunur ("hayvan"). Çözüm: alanlar `private` yapılır, getter metodu geçersiz kılınır.

Aşırı yüklemeyi geçersiz kılma sanmak

Alt sınıfta `equals(Kisi k)` yazmak `equals(Object)`'i geçersiz kılmaz, aşırı yükler; koleksiyonlar bu metodu çağırmaz. `@Override` bu hatayı yakalar.

`==` ile nesne karşılaştırmak

`==` referansları karşılaştırır; içerik için geçersiz kılınmış `equals` kullanılır.

Her yerde `instanceof` zinciri

Yeni alt sınıf eklendikçe kırılır. Davranış, geçersiz kılınan bir metoda taşınmalıdır.

ÖZET

Sık Yapılan Hatalar (3)

Yapıcıdan geçersiz kılınabilir metot çağırmak

Alt sınıfın metodu, alanları henüz atanmamış nesnede çalışır: `null` çıktısı ya da `NullPointerException`.

Dizi kovaryansına güvenmek

`Hayvan[] d = new Kopek[2]; d[1] = new Kedi();` derlenir, çalışırken `ArrayStoreException: Kedi`.

null ile belirsiz çağrı

`goster(null)`, ilişkisiz iki aşırı yükleme varken: `reference to goster is ambiguous`. Seçim açık dönüşümle yapılır.

Kutulamanın önce geldiğini sanmak

`f(long)` ve `f(Integer)` varken `f(5)` → `f(long)`: genişletme 1. aşamadadır.

equals'ı alt sınıfta tek yönlü genişletmek

`p.equals(k)` true, `k.equals(p)` false: simetri bozulur, koleksiyonlar tutarsızlaşır.

Jenerik listeyi dizi gibi sanmak

`List<Hayvan> l = new ArrayList<Kopek>();` → `incompatible types`. Jenerikler kovaryant değildir.

ÖZET

Örnek Problem 1: Çıktı Nedir?

```
class A {
    void yaz() {
        System.out.println("A.yaz");
    }

    void cagir() {
        yaz(); // this.yaz()
    }
}

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

// main içinde:
A a = new B();
a.cagir();
new A().cagir();
```

`cagir()` yalnızca `A`'da tanımlı ve içinden `yaz()`'ı çağırıyor. `B` ise yalnızca `yaz()`'ı eziyor. Program ne yazdırır?

▶ **Cevap**

Örnek Problem 2: Derlenir mi, Çalışır mı?

```
Hayvan h = new Kedi();
Kedi k1 = h;               // (1)
Kedi k2 = (Kedi) h;        // (2)
Kopek k3 = (Kopek) h;      // (3)
h.tirmala();               // (4)
((Kedi) h).tirmala();      // (5)
Object o = new Kopek();
o.sesCikar();              // (6)
((Hayvan) o).sesCikar();   // (7)
```

Her satır için karar verin: **derleme hatası**, **çalışma zamanı hatası** ya da **sorunsuz** (çıktısıyla).

▶ **Cevap**

ÖZET

Örnek Problem 3: Aşırı Yükleme + Geçersiz Kılma

```
class Ust {
    void f(Object o) {
        System.out.println("Ust.f(Object)");
    }

    void f(String s) {
        System.out.println("Ust.f(String)");
    }
}

class Alt extends Ust {
    @Override
    void f(Object o) {
        System.out.println("Alt.f(Object)");
    }
}

// main içinde:
Ust u = new Alt();
Object o = "merhaba";
u.f(o);
u.f("merhaba");
```

`Ust`'te iki aşırı yüklenmiş `f` var; `Alt` yalnızca `f(Object)`'i eziyor. İki çağrı ne yazdırır?

▶ **Cevap**

ÖZET

Örnek Problem 4: Kurulum, Aşama, Dizi

```
class Ust {
    Ust() { goster(); }
    void goster() { System.out.println("Ust"); }
}
class Alt extends Ust {
    private int[] dizi = {1, 2, 3};
    @Override void goster() {
        System.out.println(dizi == null
            ? "dizi yok" : "uzunluk " + dizi.length);
    }
}
```

1. `new Alt();` ve ardından `new Alt().goster();` ne yazdırır?
2. `f(long)`, `f(Integer)`, `f(Object)` ve `f(int...)` varken `short s = 3; f(s);`, `f('A');` ve `f(Long.valueOf(4));` hangi sürümleri seçer?
3. `Object[] d = new String[1]; d[0] = 1;` derlenir mi, çalışır mı?

▶ **Cevapları göster**

ÖZET

Alıştırmalar (1)

1. `Calisan` (`maas()` = temel maaş), `Mudur` (temel × 1,5) ve `Stajyer` (sabit 5000) sınıflarını yazın. `Calisan[]` = { Ayşe 20000, Müdür Mehmet 30000, Stajyer Can } için toplam maaş?
2. `X` : `ad()` → "X", `yaz()` → `println(ad())`. `Y extends X` : `ad()` → "Y". `Z extends Y` : `yaz()` → `println("Z:" + ad())`. { new X(), new Y(), new Z() } için her birinde `yaz()` çağrılırsa çıktı?
3. `Object o = new Kopek();` ardından `o.sesCikar();` neden derlenmez? Nasıl düzeltilir, düzeltilince ne yazar?

▶ **Cevaplar / ipuçları**

ÖZET

Alıştırmalar (2)

4. Şekil hiyerarşisine `DikUcgen(a, b)` ekleyin (alan = a·b/2, çevre = a + b + hipotenüs). `DikUcgen(3, 4)` için alan, çevre ve `ciz()` çıktıları nedir? Şekil döngüsünde ne değişir?
5. `instanceof` zincirli döngüyü (Kopek → havla, Kedi → tırmala, diğer → "Özel davranış yok") hiç `instanceof` kullanmadan yazın.
6. `yazdir(Hayvan)` ve `yazdir(Kopek)` aşırı yüklemeleri varken `Object o = new Kopek(); yazdir(o);` derlenir mi?

▶ **Cevaplar / ipuçları**

ÖZET

Kaynaklar

- Oracle, *The Java Tutorials*: Polymorphism; Overriding and Hiding Methods; Inheritance.
- J. Gosling ve diğ., *The Java Language Specification*, Java SE 21 Edition (5.5 Casting Contexts, 8.4.8 Inheritance, Overriding, and Hiding, 15.12 Method Invocation Expressions).
- P. Deitel ve H. Deitel, *Java: How to Program*, 11. baskı, Pearson (Bölüm 10: Object-Oriented Programming: Polymorphism and Interfaces).
- K. Sierra ve B. Bates, *Head First Java*, 3. baskı, O'Reilly (Inheritance and Polymorphism).
- C. S. Horstmann, *Core Java, Volume I: Fundamentals*, 12. baskı, Pearson (Bölüm 5: Inheritance).
- A. Downey ve C. Mayfield, *Think Java*, 2. baskı, O'Reilly.

Son

Bölüm 4.4: Çok Biçimlilik

Sonraki bölüm: Bölüm 4.5: Soyut Sınıflar ve Arayüzler · Terim sözlüğü