

Bölüm 4: Çok Biçimlilik

Java ile Nesne Yönelimli Programlama

Çok biçimlilik kavramı

Referans kuralı ve is-a

Yukarı ve aşağı dönüşüm

instanceof

Dinamik bağlama

Aşırı yükleme

Şekil ve tanksavar örnekleri

toString ve equals

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; **is-a** ilişkisini kullanmak.

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; metod
aramasını hiyerarşide göstermek.

Aşırı yükleme ve geçersiz kılma

Overloading ile overriding'i karşılaştırmak;
hangisinin derleyici, hangisinin JVM tarafından
seçildiğini bilmek.

Tasarım

Şekil dizisi, tanksavar oyunu ve
`toString` / `equals` örnekleriyle üst sınıf
tipiyle genel kod yazmak.

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 otomobili sürersiniz: "otomobil" arayüzüyle konuşursunuz, 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 (<code>javac</code>)	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
```

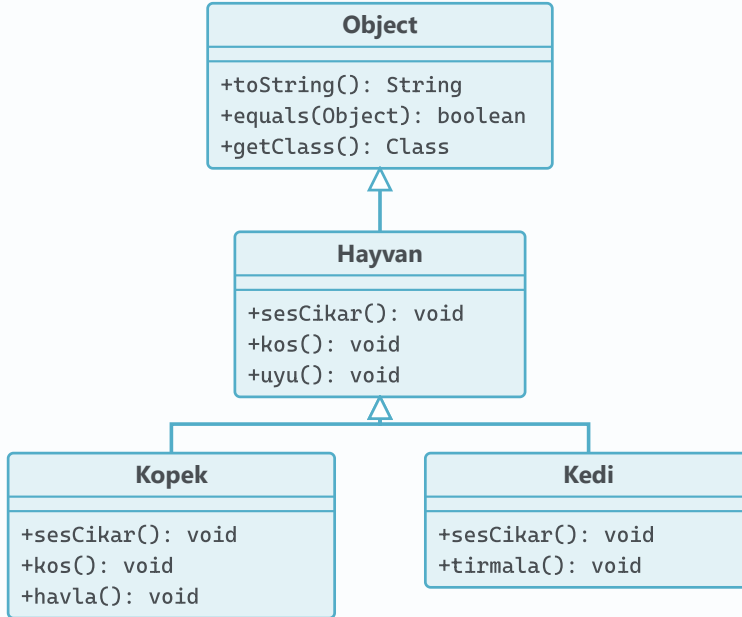
AYNI İMZA, FARKLI SINIFLAR

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

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

Bu bölümün odağı çalışma zamanı çok biçimliliğidir; "çok biçimlilik" denince çoğunlukla bu kastedilir.

Bölüm Boyunca Kullanılan Hiyerarşi



- Boş üçgenli ok **kalıtımı** (`extends`) gösterir: ok alt sınıftan üst sınıfa yönelir.
- Her sınıf, yazmasak da `Object` 'ten türer; `toString()` ve `equals()` herkeste vardır.
- `Kopek` , `sesCikar()` ve `kos()` metotlarını **geçersiz kılar** (override), `havla()` metodunu **ekler**.
- `Kedi` yalnızca `sesCikar()` 'ı geçersiz kılar ve `tirmala()` 'yı ekler; `kos()` ve `uyu()` `Hayvan` 'dan gelir.

Kopek **bir** Hayvan'dır (is-a); Kedi de **bir** Hayvan'dır. Bu yüzden ikisi de Hayvan tipindeki bir değişkende tutulabilir.

ÇOK BİÇİMLİLİK KAVRAMI

Sınıfların Kodu

```
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 Kedi extends Hayvan {  
    @Override  
    public void sesCikar() {  
        System.out.println("Miyav!");  
    }  
    public void tirmala() {  
        System.out.println("Kedi tırmalıyor");  
    }  
}
```

```
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");  
    }  
}
```

@Override notu (annotation), derleyiciye "bu metod üst sınıftaki bir metodu geçersiz kılar" der; imza yanlış yazılırsa derleme hatası alınır.

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

Referans değişkenleri

kopek
tip: Kopek

Heap'teki nesne

Kopek nesnesi

Object kısmı

Hayvan kısmı

Kopek kısmı

`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
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

Referans değişkenleri

kopek
tip: Kopek

hayvan
tip: Hayvan

Heap'teki nesne

Kopek nesnesi

Object kısmı

Hayvan kısmı

Kopek kısmı

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

Referans değişkenleri

kopek
tip: Kopek

hayvan
tip: Hayvan

nesne
tip: Object

Heap'teki nesne

Kopek nesnesi

Object kısmı

Hayvan kısmı

Kopek kısmı

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;
7 ► System.out.println(kopek == hayvan);
8 System.out.println(hayvan == nesne);
9 System.out.println(nesne.getClass());
```

ÇIKTI

```
true
```

Referans değişkenleri

kopek
tip: Kopek

hayvan
tip: Hayvan

nesne
tip: Object

Heap'teki nesne

Kopek nesnesi

Object kısmı

Hayvan kısmı

Kopek kısmı

== 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);
8 System.out.println(hayvan == nesne);
9 System.out.println(nesne.getClass());
```

ÇIKTI

```
true
true
```

Referans değişkenleri

kopek
tip: Kopek

hayvan
tip: Hayvan

nesne
tip: Object

Heap'teki nesne

Kopek nesnesi

Object kısmı

Hayvan kısmı

Kopek kısmı

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
```

Referans değişkenleri

kopek
tip: Kopek

hayvan
tip: Hayvan

nesne
tip: Object

Heap'teki nesne

Kopek nesnesi

Object kısmı

Hayvan kısmı

Kopek kısmı

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

Adım 6 / 6

DEĞİŞKENLER

kopek = → **Kopek nesnesi**

hayvan = → **aynı nesne**

nesne = → **aynı nesne**

Nesne Oluşturulurken Yapıcılar Zinciri

```
1 class Hayvan {  
2     public Hayvan() {  
3         System.out.println("Hayvan yapıcısı");  
4     }  
5 }  
6  
7 class Kopek extends Hayvan {  
8     public Kopek() {  
9         super(); // derleyici ekler  
10        System.out.println("Kopek yapıcısı");  
11    }  
12 }  
13  
14 // main içinde:  
▶ Kopek kopek = new Kopek();
```

ÇIKTI



`new` heap'te yer ayırır ve `Kopek()` yapıcısını çağırır.

Adım 1 / 7

ÇAĞRI YİĞİNİ

main()

DEĞİŞKENLER

kopek = -

Nesne Oluşturulurken Yapıcılar Zinciri

```
1 class Hayvan {  
2     public Hayvan() {  
3         System.out.println("Hayvan yapıcısı");  
4     }  
5 }  
6  
7 class Kopek extends Hayvan {  
8     public Kopek() {  
9         super(); // derleyici ekler  
10        System.out.println("Kopek yapıcısı");  
11    }  
12 }  
13  
14 // main içinde:  
15 Kopek kopek = new Kopek();
```

ÇIKTI



`Kopek()` başlar. İlk satır her zaman `super(...)` ya da `this(...)` çağrısıdır.

Adım 2 / 7

ÇAĞRI YİĞİNİ

`Kopek()`

`main()`

DEĞİŞKENLER

`kopek = -`

Nesne Oluşturulurken Yapıcılar Zinciri

```
1 class Hayvan {  
2     public Hayvan() {  
3         System.out.println("Hayvan yapıcısı");  
4     }  
5 }  
6  
7 class Kopek extends Hayvan {  
8     public Kopek() {  
9         ▶ super(); // derleyici ekler  
10        System.out.println("Kopek yapıcısı");  
11    }  
12 }  
13  
14 // main içinde:  
15 Kopek kopek = new Kopek();
```

ÇIKTI



`super()` : önce üst sınıfın parçası kurulmalıdır.

Adım 3 / 7

ÇAĞRI YİĞİNİ

Kopek()

main()

DEĞİŞKENLER

kopek = -

Nesne Oluşturulurken Yapıcılar Zinciri

```
1 class Hayvan {  
2     public Hayvan() {  
3         System.out.println("Hayvan yapıcısı");  
4     }  
5 }  
6  
7 class Kopek extends Hayvan {  
8     public Kopek() {  
9         super(); // derleyici ekler  
10        System.out.println("Kopek yapıcısı");  
11    }  
12 }  
13  
14 // main içinde:  
15 Kopek kopek = new Kopek();
```

ÇIKTI

```
Hayvan yapıcısı  
Kopek yapıcısı
```



Hayvan() çalışır; o da gizli `super()` ile `Object()` 'i çağırmış ve bitirmiştir.

Adım 4 / 7

ÇAĞRI YİĞİNİ

Hayvan()

Kopek()

main()

DEĞİŞKENLER

kopek = -

Nesne Oluşturulurken Yapıcılar Zinciri

```
1 class Hayvan {  
2     public Hayvan() {  
3         System.out.println("Hayvan yapıcısı");  
4     }  
5 }  
6  
7 class Kopek extends Hayvan {  
8     public Kopek() {  
9         super(); // derleyici ekler  
10        System.out.println("Kopek yapıcısı");  
11    }  
12 }  
13  
14 // main içinde:  
15 Kopek kopek = new Kopek();
```

ÇIKTI

```
Hayvan yapıcısı
```



Hayvan kısmının gövdesi çalışır.

Adım 5 / 7

ÇAĞRI YIĞINI

```
Hayvan()
```

```
Kopek()
```

```
main()
```

DEĞİŞKENLER

```
kopek = -
```

Nesne Oluşturulurken Yapıcılar Zinciri

```
1 class Hayvan {  
2     public Hayvan() {  
3         System.out.println("Hayvan yapıcısı");  
4     }  
5 }  
6  
7 class Kopek extends Hayvan {  
8     public Kopek() {  
9         super(); // derleyici ekler  
10        System.out.println("Kopek yapıcısı");  
11    }  
12 }  
13  
14 // main içinde:  
15 Kopek kopek = new Kopek();
```

ÇIKTI

```
Hayvan yapıcısı  
Kopek yapıcısı
```



Hayvan() döner; şimdi Kopek() 'in kalan gövdesi çalışır.

Adım 6 / 7

ÇAĞRI YIĞINI

Kopek()

main()

DEĞİŞKENLER

kopek = -

Nesne Oluşturulurken Yapıcılar Zinciri

```
1 class Hayvan {
2     public Hayvan() {
3         System.out.println("Hayvan yapıcısı");
4     }
5 }
6
7 class Kopek extends Hayvan {
8     public Kopek() {
9         super(); // derleyici ekler
10        System.out.println("Kopek yapıcısı");
11    }
12 }
13
14 // main içinde:
15 ▶ Kopek kopek = new Kopek();
```

ÇIKTI

```
Hayvan yapıcısı
Kopek yapıcısı
```



Nesne tamamlandı; adresi `kopek` 'e atanır. Yapıcılar yukarıdan aşağıya çalışır.

Adım 7 / 7

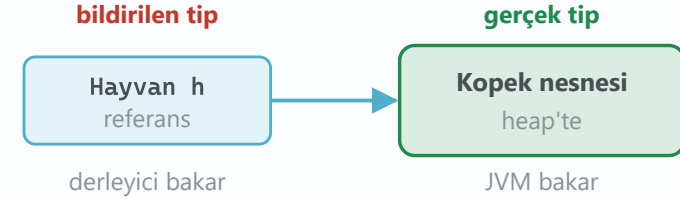
ÇAĞRI YİĞİNİ

```
main()
```

DEĞİŞKENLER

```
kopek = → Kopek nesnesi
```

Referans Tipi ve Nesne Tipi



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

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

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

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

Kumandada düğmesi olmayan özelliği kullanamazsınız: 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

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

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

Hatalı

```
// Hayvan, Köpek'in üst sınıfıdır  
Köpek köpek = new Hayvan();
```

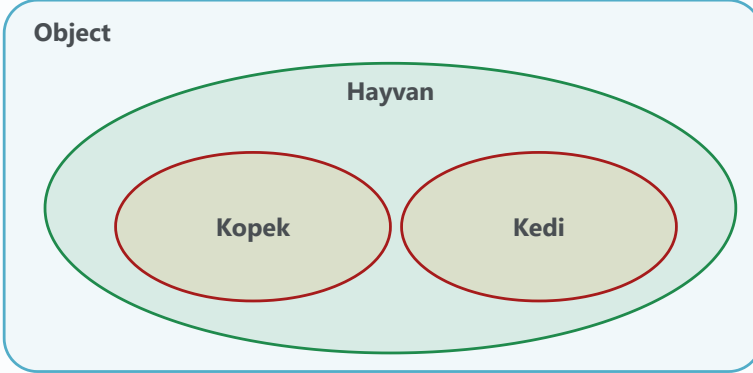
error: incompatible types:
Hayvan cannot be converted to Köpek

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

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



Üst Sınıf ve Alt Sınıf: is-a İlişkisi



- is-a ilişkisi yalnızca **alt sınıftan üst sınıfa** doğru geçerlidir.
- **Köpek is a Hayvan**: her köpek bir hayvandır.
- **Hayvan is not only a Köpek**: her hayvan bir köpek değildir.
- Bir üst sınıf nesnesi, alt sınıflardan herhangi birinin nesnesi **değildir**.

instanceof İLE SINAMA

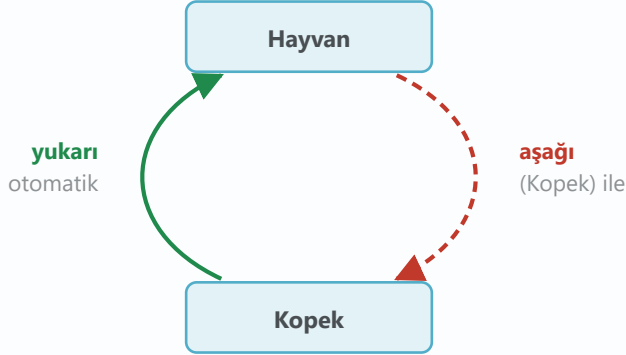
```
Hayvan h2 = new Köpek();  
Hayvan h3 = new Hayvan();  
Hayvan bos = null;
```

İfade	Sonuç
h2 instanceof Köpek	true
h2 instanceof Hayvan	true
h2 instanceof Object	true
h2 instanceof Kedi	false
h3 instanceof Köpek	false
bos instanceof Hayvan	false

null hiçbir tipin örneği değildir.

TİP DÖNÜŞÜMÜ

Yukarı ve Aşağı Dönüşüm



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

	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)

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

TIP DÖNÜŞÜMÜ

Aşağı Dönüşüm (Downcasting)

- Bir üst sınıf referansı, alt sınıf tipine **açıkça** (explicitly) dönüştürülebilir: **(AltSınıf) 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();**

Dikkatli Kullanın: 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
```

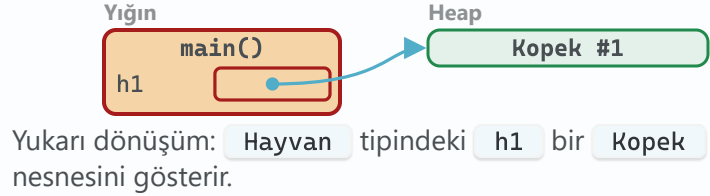
- **Derleyici** izin verir: `Hayvan` tipindeki bir referans pekâlâ bir `Kopek` 'i gösteriyor olabilir.
- **JVM**, dönüşüm anında nesnenin gerçek sınıfını denetler: nesne bir `Kedi` ya da düz bir `Hayvan` ise program istisnaıyla durur.
- Yani üst sınıf **nesnesi**, alt sınıf referansına hiçbir zaman atanamaz;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

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

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");
```

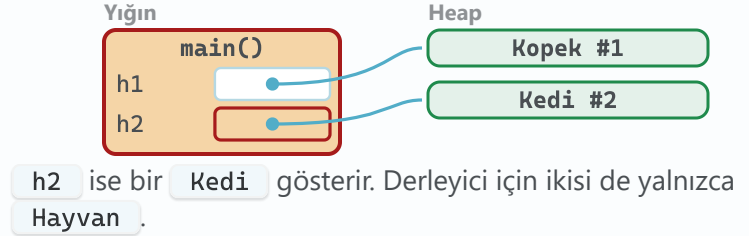


Adım 1 / 6

ÇIKTI

ClassCastException: Kod İzleme

```
1 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");
```

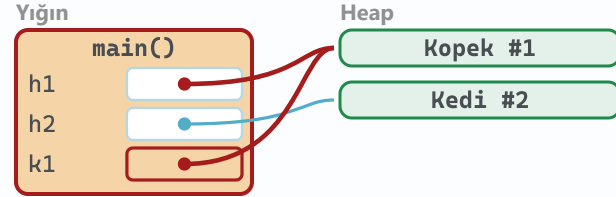


Adım 2 / 6

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.

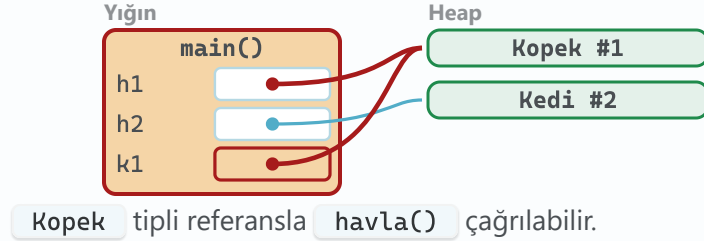
Adım 3 / 6

ÇIKTI



ClassCastException: Kod İzleme

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



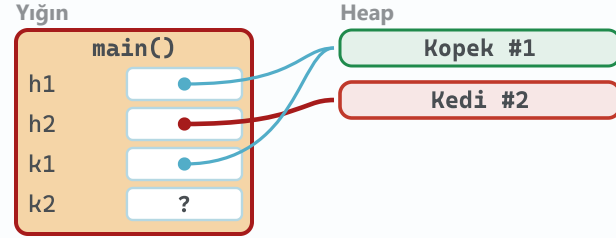
Adım 4 / 6

ÇIKTI

Köpek havlıyor

ClassCastException: Kod İzleme

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



JVM denetimi: Kedi bir Kopek mi?



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

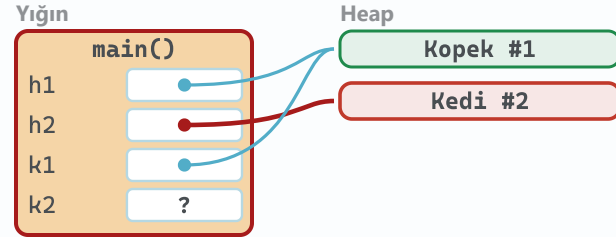
Adım 5 / 6

ÇIKTI

Köpek havlıyor

ClassCastException: Kod İzleme

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



ClassCastException → 6. ve 7. satır çalışmaz

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
```

TIP 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 sına, sonra dönüştür:
ClassCastException oluşamaz.

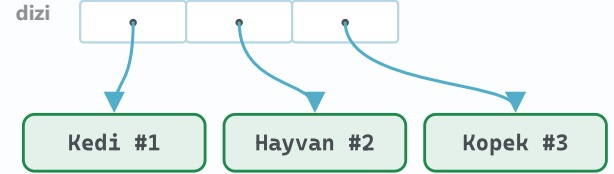
JAVA 16+: ÖRÜNTÜ EŞLEME (PATTERN MATCHING)

```
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 ögesi de **Hayvan** tipinde referans; gösterdikleri nesneler farklı sınıflardan.

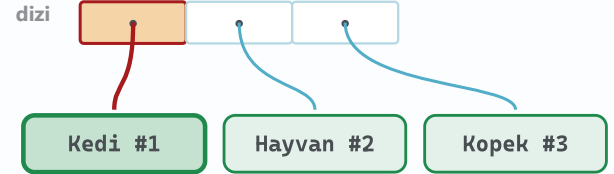
Adım 1 / 6

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.

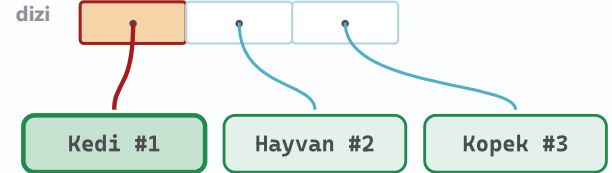
Adım 2 / 6

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();
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**

x instanceof Kedi **true**

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

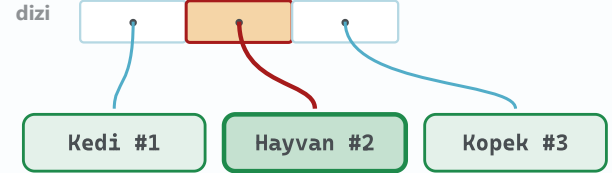
Adım 3 / 6

ÇIKTI

Kedi tırmalıyor

instanceof ile Güvenli Dönüşüm: Kod İzleme

```
1 Hayvan[] dizi = {new Kedi(), new Hayvan(), new Kopek()};
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 }
```



x → **Hayvan #2**

x instanceof Kopek **false**

x instanceof Kedi **false**

İkinci tur: düz bir **Hayvan**. Hiçbir alt sınıfın örneği değil: iki sına da yanlı.

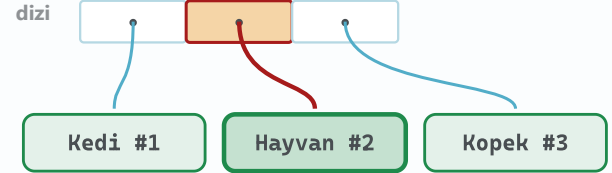
Adım 4 / 6

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 {
8         ▶ System.out.println("Özel davranış yok");
9     }
10 }
```



x → Hayvan #2

x instanceof Kopek **false**

x instanceof Kedi **false**

else dalı çalışır; hiçbir dönüşüm denetlenmediği için hata riski yok.

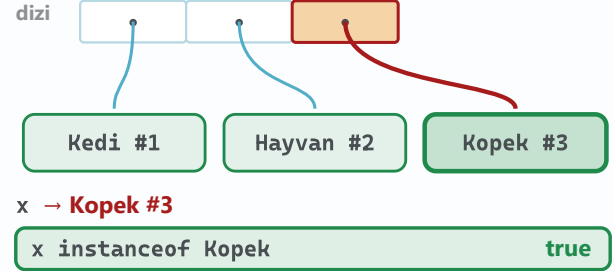
Adım 5 / 6

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
```

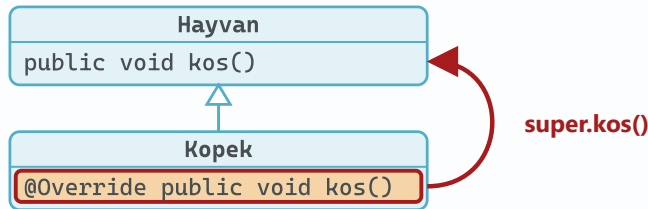
Hatırlatma: Metot Geçersiz Kılma (Overriding)

- Alt sınıf, üst sınıftan aldığı bir metodu **aynı imzayla** (ad + parametre listesi) yeniden yazar.
- Dönüş tipi aynı olmalıdır (ya da bir alt tipi: kovaryant dönüş).
- Erişim belirleyici **daraltılamaz**: `public` metot `public` kalmalıdır.
- `private`, `static` ve `final` metotlar geçersiz kılınmaz.
- `@Override` yazmak zorunlu değildir, ama yazım hatalarını derleme zamanında yakalar.
- Geçersiz kılınan sürüme alt sınıfın içinden `super.metot()` ile ulaşılabilir.

```
class Kopek extends Hayvan {  
    @Override  
    public void kos() {  
        super.kos(); // Hayvan sürümü  
        System.out.println("...ve havlıyor");  
    }  
}
```

Geçersiz kılma ≠ aşırı yükleme

Geçersiz kılma: **aynı** imza, **farklı sınıf**. Aşırı yükleme: **aynı ad**, **farklı parametreler**.



- ✓ ad ve parametreler aynı
- ✓ dönüş tipi aynı (ya da alt tip)
- ✓ erişim daralmadı (public)
- ✗ private / static / final olmaz

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
```

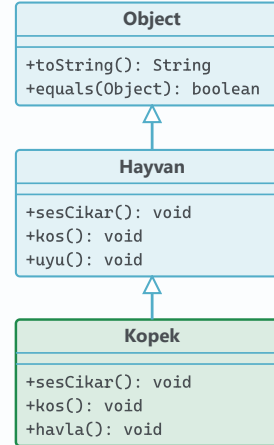
ÇAĞRI YİĞİNİ

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

ÇAĞRI YİĞİNİ

Kopek.kos()

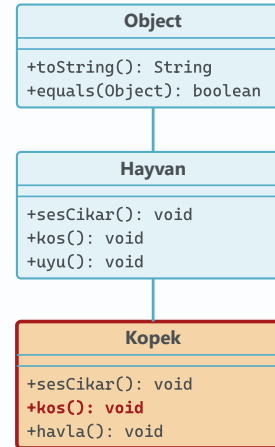
main()

DEĞİŞKENLER

h = → Kopek nesnesi

ÇIKTI

Köpek koşuyor



✓ bulundu

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

Adım 2 / 5

Metot Araması Hiyerarşinin En Altından Başlar

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

ÇAĞRI YİĞİNİ

Hayvan.uyu()

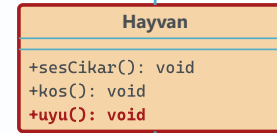
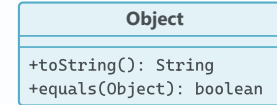
main()

DEĞİŞKENLER

h = → Kopek nesnesi

ÇIKTI

Köpek koşuyor
Hayvan uyuyor



✓ bulundu



X yok, yukarı

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

Adım 3 / 5

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 YİĞİNİ

Object.toString()

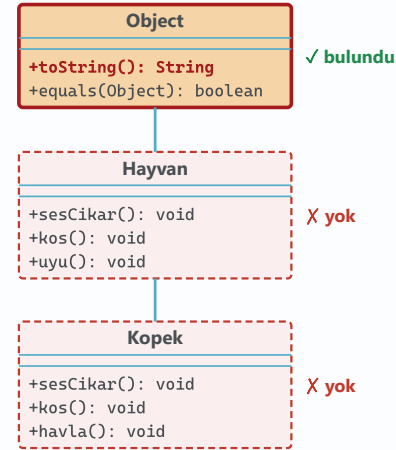
main()

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 YİĞİNİ

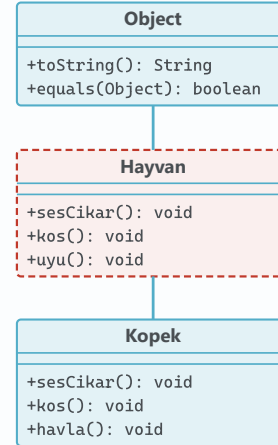
main()

DEĞİŞKENLER

h = → Kopek nesnesi

ÇIKTI

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



✗ havla() yok

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

Çok Biçimli Dizi: Kodu Adım Adım İzle

```
► 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 YİĞİNİ

```
main()
```

DEĞİŞKENLER

```
i = -
```

ÇIKTI



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

Adım 1 / 11

Çok Biçimli Dizi: Kodu Adım Adım İzle

```
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  }
```

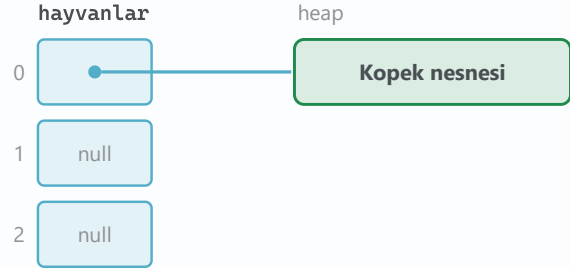
ÇAĞRI YİĞİNİ

main()

DEĞİŞKENLER

i = -

ÇIKTI



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

Çok Biçimli Dizi: Kodu Adım Adım İzle

```
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++)
6 {
7     hayvanlar[i].sesCikar();
8 }
```

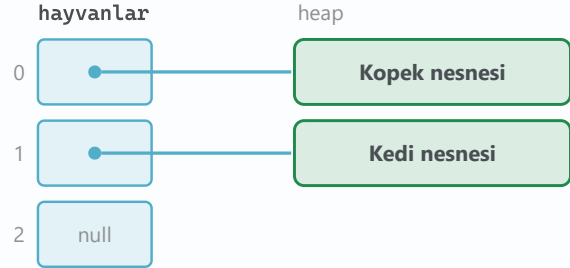
ÇAĞRI YIĞINI

main()

DEĞİŞKENLER

i = -

ÇIKTI



1. öge bir **Kedi** 'yi gösterir.

Adım 3 / 11

Çok Biçimli Dizi: Kodu Adım Adım İzle

```
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++)
6 {
7     hayvanlar[i].sesCikar();
8 }
```

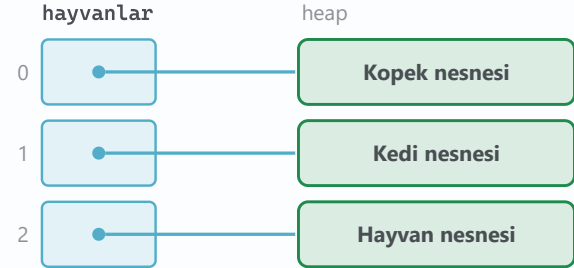
ÇAĞRI YIĞINI

main()

DEĞİŞKENLER

i = -

ÇIKTI



2. öge düz bir **Hayvan** nesnesidir. Dizi artık üç farklı biçim tutuyor.

Adım 4 / 11

Çok Biçimli Dizi: Kodu Adım Adım İzle

```
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 }
```

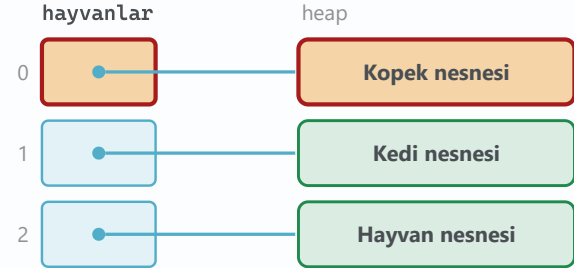
ÇAĞRI YIĞINI

main()

DEĞİŞKENLER

i = 0

ÇIKTI



i = 0 , 0 < 3 doğru → döngü gövdesi.

Adım 5 / 11

Çok Biçimli Dizi: Kodu Adım Adım İzle

```
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++)
6 {
7     ▶ hayvanlar[i].sesCikar();
8 }
```

ÇAĞRI YIĞINI

Kopek.sesCikar()

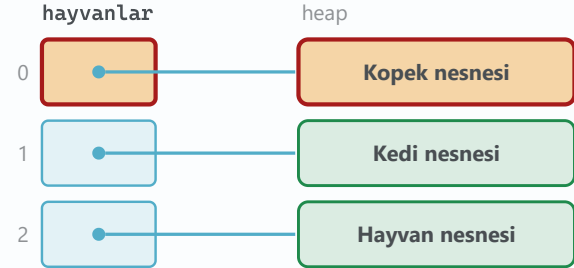
main()

DEĞİŞKENLER

i = 0

ÇIKTI

Hav hav!



Aynı satır, farklı nesne: JVM nesnenin sınıfına bakar →

Kopek.sesCikar()

Adım 6 / 11

Çok Biçimli Dizi: Kodu Adım Adım İzle

```
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

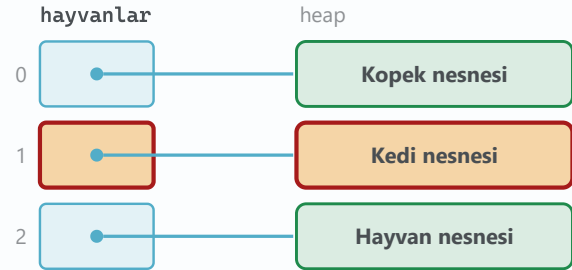
main()

DEĞİŞKENLER

i = 1

ÇIKTI

Hav hav!



i = 1 , 1 < 3 doğru → döngü gövdesi.

Adım 7 / 11

Çok Biçimli Dizi: Kodu Adım Adım İzle

```
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++)
6 {
7     ▶ hayvanlar[i].sesCikar();
8 }
```

ÇAĞRI YIĞINI

Kedi.sesCikar()

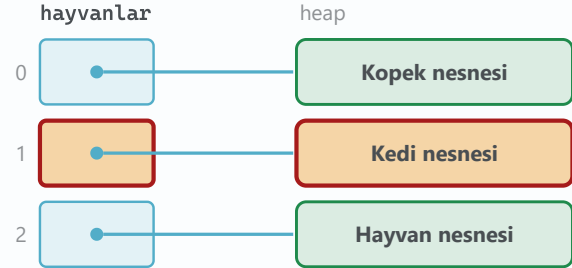
main()

DEĞİŞKENLER

i = 1

ÇIKTI

```
Hav hav!
Miyav!
```



Aynı satır, farklı nesne: JVM nesnenin sınıfına bakar →

Kedi.sesCikar() .

Adım 8 / 11

Çok Biçimli Dizi: Kodu Adım Adım İzle

```
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

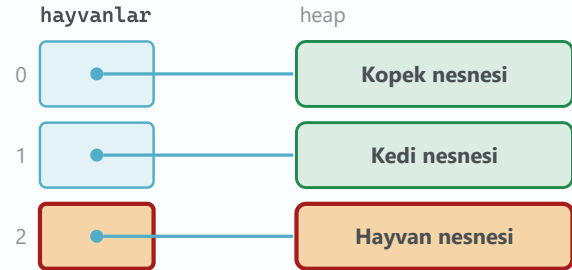
main()

DEĞİŞKENLER

i = 2

ÇIKTI

Hav hav!
Miyav!



i = 2 , 2 < 3 doğru → döngü gövdesi.

Adım 9 / 11

Çok Biçimli Dizi: Kodu Adım Adım İzle

```
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()

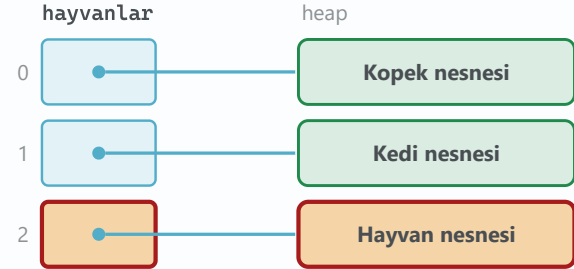
main()

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: Kodu Adım Adım İzle

```
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

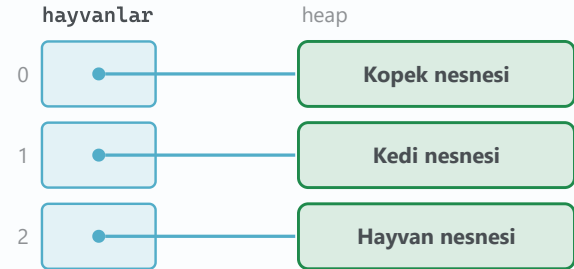
main()

DEĞİŞKENLER

i = 3

ÇIKTI

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



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

Adım 11 / 11

DİNAMİK BAĞLAMA

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
```

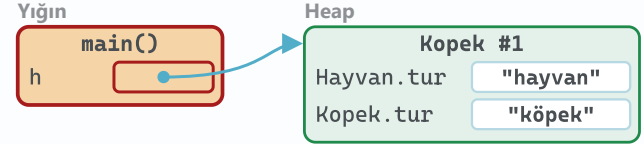
Üye	Neye göre seçilir?
Örnek metodu	Nesne tipi (dinamik)
Alan (değişken)	Referans tipi (statik)
<code>static</code> metod	Referans tipi (statik)
<code>private</code> / <code>final</code>	Geçersiz kılınamaz, static

static metodu sınıf adıyla çağırın:

`Kopek.bilgi()`. Alt sınıfta aynı adla alan tanımlamaktan kaçının.

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 sınıfinkini ezmez, **gizler**:
nesnede iki ayrı alan vardır.

Adım 1 / 5

ÇIKTI

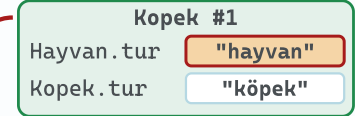
Alan Gizleme ve `static`: Kod İzleme

```
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);
```

Yığın



Heap

alan → **bildirilen tip** (Hayvan)

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

Adım 2 / 5

ÇIKTI

hayvan

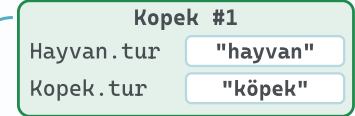
Alan Gizleme ve `static`: Kod İzleme

```
1 class Hayvan {  
2     String tur = "hayvan";  
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());  
15 System.out.println(((Kopek) h).tanit());
```

Yığın



Heap



`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"; }  
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());  
15 System.out.println(((Kopek) h).tur);
```

Yığın

Kopek.tanit()

this

main()

h

Heap

Kopek #1

Hayvan.tur

"hayvan"

Kopek.tur

"köpek"

örnek metodu → gerçek tip (Kopek)

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
```

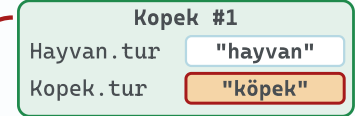
Alan Gizleme ve `static`: Kod İzleme

```
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);
```

Yığın



Heap



(Kopek) h → bildirilen tip Kopek → Kopek.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
```

Aşırı Yükleme (Overloading)

```
public class HesapMakinesi {
    public int topla(int sayi1, int sayi2) {
        return sayi1 + sayi2;
    }
    public double topla(double sayi1, double sayi2) {
        return sayi1 + sayi2;
    }
    public int topla(int sayi1, int sayi2, int sayi3) {
        return sayi1 + sayi2 + sayi3;
    }
    // dizinin elemanlarını topla
    public int topla(int[] sayilar) {
        int toplam = 0;
        for (int sayi : sayilar) {
            toplam += sayi;
        }
        return toplam;
    }
}
```

- Aynı sınıfta **aynı adlı, parametre listesi farklı** (sayı, tip, sıra) metotlar.
- Derleyici argümanlara bakarak **derleme zamanında** seçer.
- Yalnızca dönüş tipi farklı olan iki metot aşırı yükleme **değildir**.

```
HesapMakinesi hm =
    new HesapMakinesi();
int[] sayilar = {1, 2, 3, 4, 5};
int s1 = hm.topla(5, 3);
double s2 = hm.topla(3.5, 2.7);
int s3 = hm.topla(4, 6, 8);
int s4 = hm.topla(sayilar);
```

DEĞERLER

```
s1 = 8, s2 = 6.2,
s3 = 18, s4 = 15
```

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

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:
10 ▶ Hayvan hayvan = new Kopek();
11 ▶ Hayvan diger = new Kedi();
12 yazdir(hayvan);
13 yazdir(diger);
14 yazdir((Kopek) hayvan);
```

Derleyici
bildirilen tipe bakar

yazdir(Hayvan)

yazdir(Kopek)

derleme zamanı

JVM
nesnenin sınıfına bakar

Kopek.sesCikar()

Kedi.sesCikar()

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);
```

Derleyici

bildirilen tipe bakar

argüman tipi: Hayvan

yazdir(Hayvan)

yazdir(Kopek)

derleme zamanı

JVM

nesnenin sınıfına bakar

Kopek.sesCikar()

Kedi.sesCikar()

çalışma zamanı

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): ");  
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);  
13 yazdir(diger);  
14 yazdir((Kopek) hayvan);
```

Derleyici

bildirilen tipe bakar

argüman tipi: Hayvan

yazdir(Hayvan)

yazdir(Kopek)

derleme zamanı

JVM

nesnenin sınıfına bakar

nesne: Kopek

Kopek.sesCikar()

Kedi.sesCikar()

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);
```

Derleyici

bildirilen tipe bakar

argüman tipi: Hayvan

yazdir(Hayvan)

yazdir(Kopek)

derleme zamanı

JVM

nesnenin sınıfına bakar

Kopek.sesCikar()

Kedi.sesCikar()

çalışma zamanı

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): ");  
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);  
13 yazdir(diger);  
14 yazdir((Kopek) hayvan);
```

Derleyici

bildirilen tipe bakar

argüman tipi: Hayvan

yazdir(Hayvan)

yazdir(Kopek)

derleme zamanı

JVM

nesnenin sınıfına bakar

nesne: Kedi

Kopek.sesCikar()

Kedi.sesCikar()

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);
```

Derleyici
bildirilen tipe bakar
argüman tipi: Kopek

yazdir(Hayvan)

yazdir(Kopek)

derleme zamanı

JVM
nesnenin sınıfına bakar

Kopek.sesCikar()

Kedi.sesCikar()

çalışma zamanı

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): ");  
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
argüman tipi: Kopek

yazdir(Hayvan)

yazdir(Kopek)

derleme zamanı

JVM
nesnenin sınıfına bakar
nesne: Kopek

Kopek.sesCikar()

Kedi.sesCikar()

çalışma zamanı

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

Adım 7 / 7

ÇIKTI

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

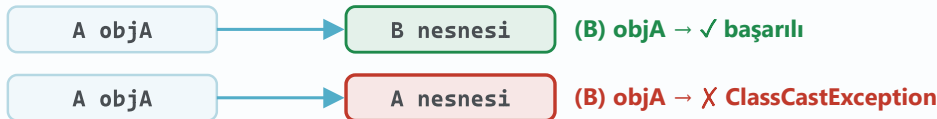
Kalıtım ve Çok Biçimlilik

- 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.

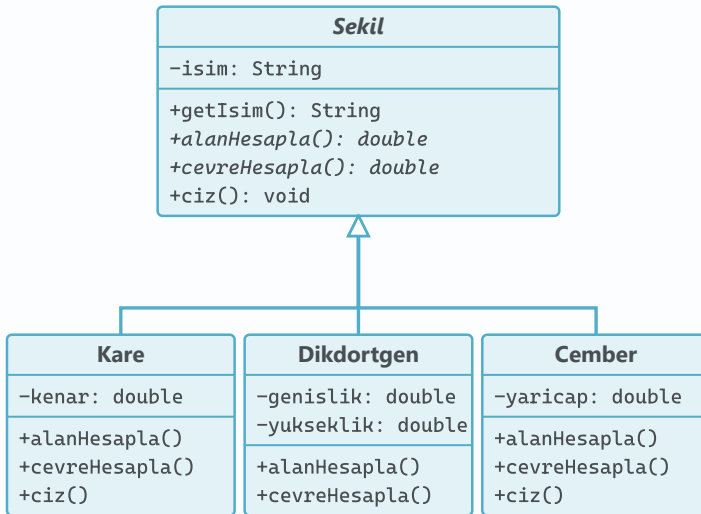
```
A objA = new B(); // B, A gibi davranır  
B objB = (B) objA; // geçerli, ancak  
// dikkatli kullanılmalı
```

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



```
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. Ayrıntılar bir sonraki bölümde.

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);  
    }  
}
```

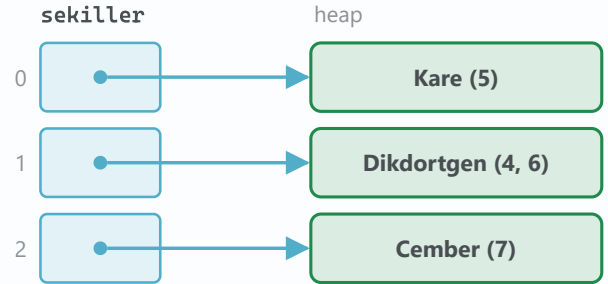
Cember : **Math.PI** ile hesaplar; **ciz()** 'i kendi bilgisiyle geçersiz kılar.

Şekiller Dizisi: Kodu Adım Adım İzle

```
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());
9     sekil.ciz();
10 }
```

ÇIKTI

[Redacted output]



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

Adım 1 / 7

ÇAĞRI YİĞİNİ

main()

DEĞİŞKENLER

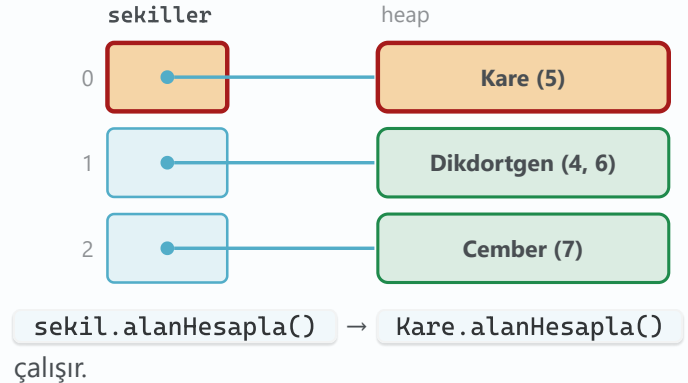
sekil = -

Şekiller Dizisi: Kodu Adım Adım İzle

```
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());
9     sekil.ciz();
10 }
```

ÇIKTI

```
Kare alanı: 25.0
```



Adım 2 / 7

ÇAĞRI YİĞİNİ

```
Kare.alanHesapla()
```

```
main()
```

DEĞİŞKENLER

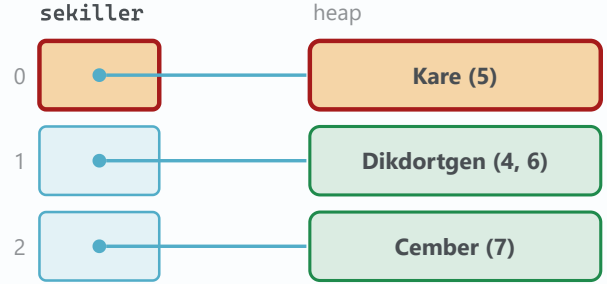
```
sekil = → Kare (5)
```

Şekiller Dizisi: Kodu Adım Adım İzle

```
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());
9     sekil.ciz();
10 }
```

ÇIKTI

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



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

Adım 3 / 7

ÇAĞRI YİĞİNİ

Kare.ciz()

main()

DEĞİŞKENLER

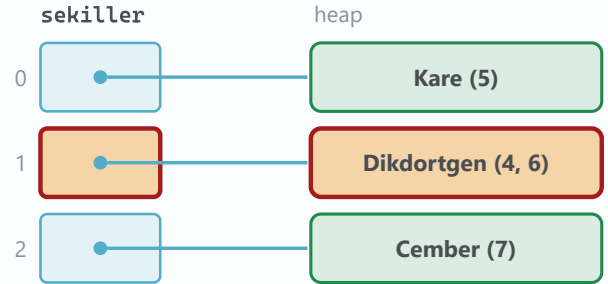
sekil = → Kare (5)

Şekiller Dizisi: Kodu Adım Adım İzle

```
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());
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 YİĞİNİ

Dikdortgen.alanHesapla()

main()

DEĞİŞKENLER

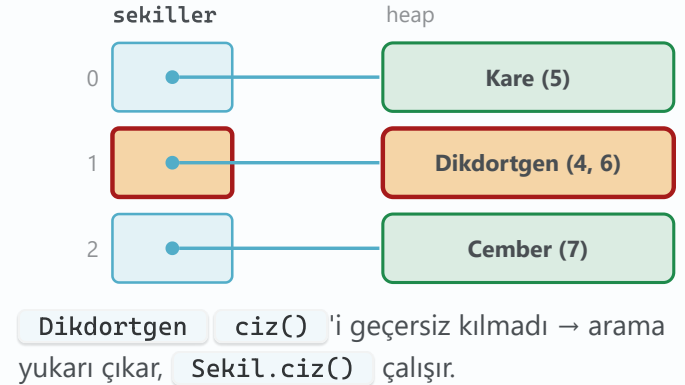
sekil = → Dikdortgen (4, 6)

Şekiller Dizisi: Kodu Adım Adım İzle

```
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());
9     sekil.ciz();
10 }
```

ÇIKTI

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



Adım 5 / 7

ÇAĞRI YIĞINI

`Sekil.ciz()`

`main()`

DEĞİŞKENLER

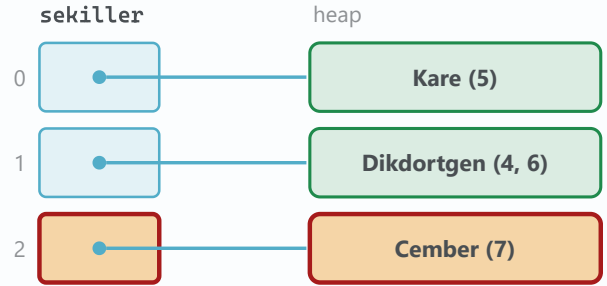
`sekil = → Dikdortgen (4, 6)`

Şekiller Dizisi: Kodu Adım Adım İzle

```
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());
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

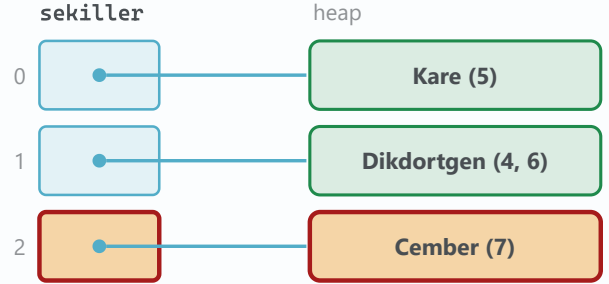
sekil = → Cember (7)

Şekiller Dizisi: Kodu Adım Adım İzle

```
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());
9     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 YİĞİNİ

Cember.ciz()

main()

DEĞİŞKENLER

sekil = → Cember (7)

Neden Çok Biçimlilik?

Tip sınavarak

```
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 metod da değişmeli; unutulursa `alanKotu(new DikUcgen(3, 4))` sessizce **0.0** döner.

Çok biçimli

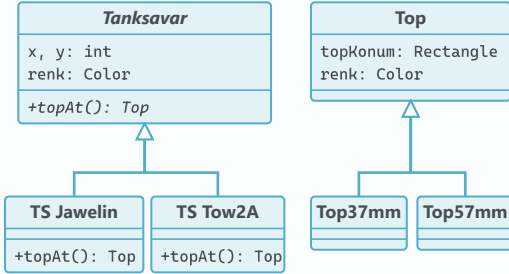
```
static double toplamAlan(Sekil[] dizi) {  
    double toplam = 0;  
    for (Sekil sekil : dizi) {  
        // her nesne kendi alanını  
        toplam += sekil.alanHesapla();  
    }  
    return toplam;  
}
```

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

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

KALITIM VE ÇOK BİÇİMLİLİK

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.

Object Metotlarını Geçersiz Kılmak: toString ve equals

```
public class Kisi {  
    private String ad;  
    private int yas;  
    public Kisi(String ad, int yas) {  
        this.ad = ad;  
        this.yas = yas;  
    }  
    @Override  
    public boolean equals(Object obj) {  
        if (this == obj) return true;  
        if (!(obj instanceof Kisi)) return false;  
        Kisi kisi = (Kisi) obj; // aşağı dönüşüm  
        return yas == kisi.yas  
            && ad.equals(kisi.ad);  
    }  
  
    @Override  
    public String toString() {  
        return "Kisi[Ad=" + ad  
            + ", Yas=" + yas + "];"  
    }  
}
```

```
Kisi k1 = new Kisi("Ali", 25);  
Kisi k2 = new Kisi("Ali", 25);  
System.out.println(k1 == k2);  
System.out.println(k1.equals(k2));  
Object nesne = k1;  
System.out.println(nesne);  
System.out.println(nesne.equals(k2));
```

```
false  
true  
Kisi[Ad=Ali, Yas=25]  
true
```

- `println(nesne)`, `toString()` 'i çağırır: dinamik bağlamayla `Kisi` sürümünü.
- `equals(Object)` : önce `instanceof`, sonra aşağı dönüşüm.
- `equals` geçersiz kılınınca `hashCode()` da geçersiz kılınmalıdır.

ÖZET

Özet

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

İlgili Sayfalar

Soyut Sınıflar ve Arayüzler

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

Bu bölümle bağlantısı

Bir sınıf tek bir üst sınıftan türeyebilir, ama birçok arayüzü gerçekleştirebilir: `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
<code>Kopek k = new Hayvan();</code>	<i>incompatible types: Hayvan cannot be converted to Kopek</i>	Her hayvan köpek değildir; ters atama yapılamaz.
<code>Kopek k = hayvan;</code>	<i>incompatible types: Hayvan cannot be converted to Kopek</i>	Nesne köpek olsa bile açık dönüşüm gerekir: <code>(Kopek) hayvan</code> .
<code>hayvan.havla();</code>	<i>cannot find symbol ... variable hayvan of type Hayvan</i>	Referans tipinde metot yok; önce dönüştürün.
<code>(Kopek) h.havla();</code>	<i>not a statement</i>	Parantez eksik: <code>((Kopek) h).havla();</code>
<code>(Kedi) kopek</code>	<i>incompatible types: Kopek cannot be converted to Kedi</i>	Kardeş sınıflar arasında dönüşüm yoktur.
<code>@Override void sescikar()</code>	<i>method does not override or implement a method from a supertype</i>	Ad yanlış yazıldı (büyük/küçük harf).
<code>void sesCikar()</code> (üstte <code>public</code>)	<i>attempting to assign weaker access privileges; was public</i>	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"). Alanları `private` yapıp getter metodu geçersiz kılın.

Aşırı yüklemeyi geçersiz kılma sanmak

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

`==` ile nesne karşılaştırmak

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

Her yerde `instanceof` zinciri

Yeni alt sınıf eklendikçe kırılır. Davranışı geçersiz kılınan bir metoda taşıyın.

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

ÖZET

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)
```

► Cevap

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

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

Alıştırmalar (1)

1. `Calisan` (`maas()` = temel maaş), `Mudur` (temel $\times$ 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 = $a \cdot b / 2$, çevre = $a + b + \text{hipotenüs}$).
`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

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: Çok Biçimlilik

Sonraki bölüm: Bölüm 4: Soyut Sınıflar ve Arayüzler · [Terim sözlüğü](#)