

Java ve Inheritance

Kalıtım (Inheritance) Nedir?

- Kalıtım (inheritance) **bilginin hiyerarşik sınıflandırılmasıdır**
- İki sınıf **ortak bazı özelliklere ve davranışlara** sahip olabilir, biri diğerine göre **ilave özellikler ve davranışlarda** içerebilir
- Bu iki sınıf arasında **hiyerarşik bir ilişki** kurulabilir
- Hiyerarşide üst sınıfa **superclass**, alt sınıfa **subclass** adı verilir

Kalıtım (Inheritance) Nedir?

- Bir üst sınıf kendinden türeyen diğer bütün alt sınıfların **ortak özellik ve davranışlarını** kendinde barındırır
- Alt sınıflar ise üst sınıfın sahip olduğu özellik ve davranışların yanı sıra **kendi tanımlandıkları özellik ve davranışlara** da sahiptirler
- Alt sınıflar ayrıca **üst sınıftaki bazı davranışları kendilerine göre özelleştirebilirler**

Kalıtım ve super Anahtar Kelimesi

```
public class Honda extends Araba {  
  
    public Honda() {  
        super("Honda");  
    }  
  
    @Override  
    public Integer hizlan() {  
        Integer hiz = motor.getHiz() + 1;  
        motor.setHiz(hiz);  
        return hiz;  
    }  
}
```

Lokal değişken
Programcı tarafından
initialize edilmeli

Kalıtım ve super Anahtar Kelimesi

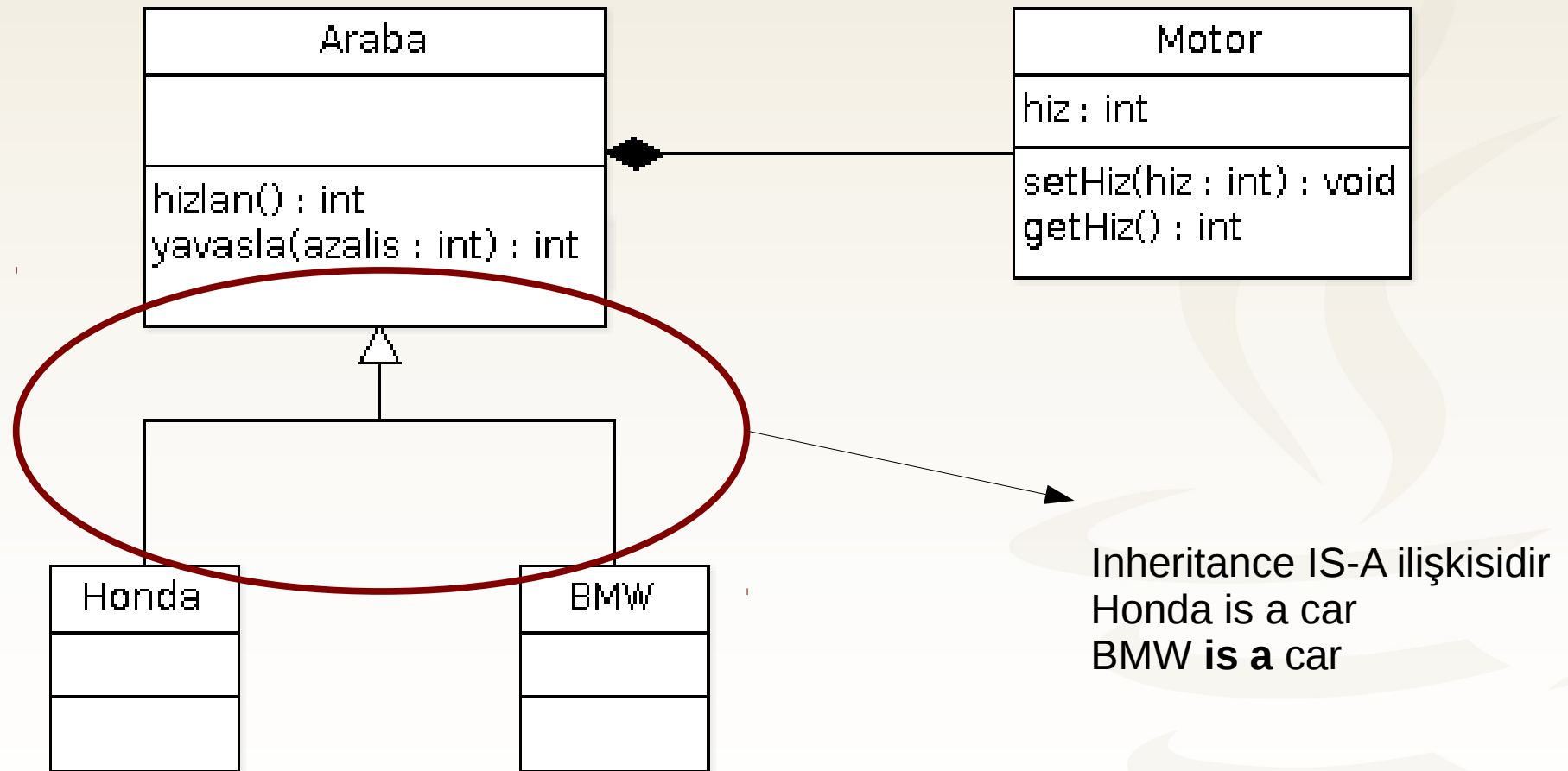
- Nesnenin inherit ettiği **üst sınıfa karşılık gelen nesneye erişim** sağlar
- Metot veya constructor'ların içerisinde kullanılabilir
- **Üst nesnenin** constructor'ını veya metotlarını çağırmayı sağlar,
- Ya da üst nesnede tanımlanmış bir **field'a** erişim sağlar

Kalıtım ve super Anahtar Kelimesi

```
public class BMW extends Araba {  
  
    public BMW() {  
        this("BMW");  
    }  
  
    public BMW(String adi) {  
        super(adi);  
        System.out.println("Adi :" + adi);  
    }  
  
    @Override  
    public Integer hizlan() {  
        Integer hiz = motor.getHiz() + 3;  
        motor.setHiz(hiz);  
        return hiz;  
    }  
}
```

- Üst sınıfta **sadece default constructor** varsa alt sınıflarda constructor tanımlanmasına gerek yoktur
- Üst sınıfta sadece input argüman alan constructor(lar) tanımlı ise, bunların arasından **en az bir constructor**'ın da alt sınıfta tanımlanması gerekir
- Constructor içerisinde **this(..)**, **super(..)** ifadeleri yazılacak ise ilk ifade olmalıdırlar

UML ve Inheritance



java.lang.Object Sınıfı

- Java'da bütün sınıfların ortak superclass'ı **java.lang.Object** sınıfıdır
- Object sınıfında bütün nesnelerin sahip olduğu ortak metotlar bulunur
 - getClass()
 - equals(Object o)
 - hashCode()
 - toString()
- Sınıfların Object sınıfından **explicit** biçimde extend etmelerine gerek yoktur

instanceof Operatörü

- Bir nesnenin belirli **bir sınıfa ait olup olmadığını** test eder

```
Araba kirmiziAraba = new Honda();
```

```
Araba yesilAraba = new BMW();
```

```
...
```

```
System.out.println(kirmiziAraba instanceof Araba); —————> true
```

```
System.out.println(kirmiziAraba instanceof Honda); —————> true
```

```
System.out.println(yesilAraba instanceof Araba); —————> true
```

```
System.out.println(yesilAraba instanceof BMW); —————> true
```

```
System.out.println(kirmiziAraba instanceof BMW); —————> false
```

== ve equals(..) Karşılaştırmaları

- == ve != operatörleri **nesne referanslarının memory adreslerini** karşılaştırır
- Aynı memory adresini gösteren iki değişken karşılaştırılmadığı müddetçe sonuç **false** olacaktır

```
Integer i1 = new Integer(1);  
Integer i2 = new Integer(1);
```

```
System.out.println(i1 == i2); → false
```

```
int a = 1;  
int b = 1;
```

```
System.out.println(a == b); → true
```

Değer Eşitliği ve equals(..) Metodu

- Nesnelerin değerlerini karşılaştırmak için **equals** metodu kullanılmalıdır
- equals metodu **java.lang.Object** sınıfında tanımlıdır
- **Default implementasyonda ==** gibi nesne referanslarının adreslerini kontrol eder
- Ancak built-in **Java tiplerinde** equals metodu bizim için override edilmiştir

```
Integer i1 = new Integer(1);  
Integer i2 = new Integer(1);  
  
System.out.println(i1.equals(i2));
```

→ true

Uygulamaya Özel Sınıflarda equals(..) Metodu

- Ancak kendi yazdığınız sınıflarda equals metodunu **override etmediğiniz takdirde** karşılaştırma yine **nesne referans adreslerine göre** olacaktır
- Uygulama içerisinde yazılan her sınıf equals metodunu **uygulamaya özel iş mantığına göre** override etmelidir

equals(..) Metodu Olmadan Nesnelerin Karşılaştırılması




```
public class Kisi {  
    private String ad;  
    private String soyad;  
  
    public Kisi(String ad, String soyad) {  
        this.ad = ad;  
        this.soyad = soyad;  
    }  
  
    public String getAd() {  
        return ad;  
    }  
  
    public String getSoyad() {  
        return soyad;  
    }  
}
```

equals metodu override edilmediği için default davranış hafıza adreslerinin kontrol karşılaştırılmasıdır

```
Kisi k1 = new Kisi("kenan", "sevindik");  
Kisi k2 = new Kisi("kenan", "sevindik");  
  
System.out.println(k1.equals(k2));
```

→ false

equals(..) Metodu Nasıl Yazılmalı?

```
public boolean equals(Object o) {  
    if (o == null) return false;  k1.equals(null)  
  
    if (this == o) return true;  k1.equals(k1)  
  
    if (o instanceof Kisi) {  k1.equals(k2)  
        Kisi k = (Kisi) o;  
        return this.getAd() != null  
            && this.getAd().equals(k.getAd())  
            && this.getSoyad() != null  
            && this.getSoyad().equals(k.getSoyad());  
    }  
  
    return false;  
}
```

equals(..) Metodu Nasıl Yazılmalı?

```
public boolean equals(Object o) {  
    if (o == null) return false;  
  
    if (this == o) return true;  
  
    if (!Kisi.class.isAssignableFrom(o.getClass()))  
        return false;  
  
    Kisi k = (Kisi) o;  
  
    return this.getAd() != null  
        && this.getAd().equals(k.getAd())  
        && this.getSoyad() != null  
        && this.getSoyad().equals(k.getSoyad());  
}
```

instanceof kullanımına alternatiftir

Abstract Class ve Concrete Class Nedir?



Peki Meyve nedir?

Soyutlama/Abstraction Kavramı

- Burada “meyve”, elma, armut, portakal gibi **farklı türde olguların** sahip oldukları **ortak özellik ve davranışların ifade edilmesini** sağlar
- Soyutlama sadece problemi ilgilendiren, problem için önemli noktalara, **özellik ve davranışlara odaklanması** işlemidir

```
public abstract class Meyve {  
    private String adi;  
    private String kokusu;  
    private float agirligi;  
  
    public String getAdi() {  
        return adi;  
    }  
  
    //...  
}
```

Abstract Class & Abstract Metot

- Soyut sınıflar “**abstract**” anahtar kelimesi ile tanımlanır
- Soyut sınıflar içerisinde attribute veya metot tanımlanabilir
- Bu metotlardan bazılarının davranışlarının alt sınıflar tarafından tanımlanması istenebilir
- Bu tür metotlara “**abstract metot**” adı verilir
- “abstract” anahtar kelimesi ile tanımlanırlar

Abstract Class & Abstract Metot

```
public abstract class Araba {  
    protected Integer hiz = 0;  
  
    public Integer yavasla(Integer azalis) {  
        return hiz-=azalis;  
    }  
  
    public abstract Integer hizlan();  
}
```

~~Araba kirmiziAraba = new Araba();~~

→ Abstract class'dan nesne yaratılamaz

Abstract vs Concrete Sınıflar

- Bir sınıf içerisinde en az bir abstract metot var ise bu sınıf abstract olmak zorundadır
- Abstract sınıf içerisinde hiç abstract metot da olmayabilir
- Eğer bir sınıf abstract anahtar kelimesi ile tanımlı değil ise “**concrete sınıf**” olarak adlandırılır

Alt Sınıflar

- Bir üst sınıftan **türetilen alt sınıf tanımıdır**
- Alt sınıfın hangi üst sınıftan türediği **“extends”** ile belirtilir
- Bu alt sınıf üst sınıfın sahip olduğu **bütün özelliklere sahiptir**
- Ayrıca üst sınıfta olmayan **yeni attribute ve metotlar** da alt sınıfa **eklenebilir**
- Ya da **üst sınıfta yer alan metotlardan bir veya daha fazlasının davranışı değiştirilebilir**

Alt Sınıflar

- Alt sınıflar **herhangi bir üst sınıftan türetilebilirler**
- Türetilecek olan **üst sınıfın “abstract” olmasına gerek yoktur**

Alt Sınıflar

Extends anahtar kelimesi ile üst sınıf belirtilir, sadece bir sınıftan extend edilebilir

```
public class Honda extends  
Araba {
```

```
    public Integer hizlan() {  
        return hiz += 3;  
    }  
}
```

```
public class Toyota extends  
Araba {
```

```
    public Integer hizlan() {  
        return hiz += 2;  
    }  
}
```

```
public class Formula1 {  
    public static void main(String[] args) {
```

```
        Araba kirmiziAraba = new Honda();  
        Araba yesilAraba = new Toyota();
```

Sadece concrete sınıflardan nesne yaratılabilir

Metot İmzası (Method Signature)

Access modifier

Return tipi

Metot ismi

Input argüman tipleri ve sırası

throws clause

```
public Integer hizlan(Integer miktar) throws IllegalArgumentException  
{  
    return hiz + miktar;  
}
```

Metot Override

- Üst sınıfta tanımlı bir metodun aynı imza ile alt sınıfta da tanımlanmasına “**method overriding**” adı verilmektedir
- Metot overriding ile üst sınıftaki metodun davranışı alt sınıfta değiştirilebilir
- Artık alt sınıftan bir nesne yaratılıp bu metot çağrıldığı vakit **alt sınıfta yeniden tanımlanmış olan metot davranışı etkin** olacaktır

Metot Overload

- Aynı ismin farklı input argümanlarla kullanılarak iki veya daha fazla metot tanımlama işlemine “**method overloading**” adı verilir
- Input argümanların farklı sayıda ve sırada olması gerekir
- Method overload **aynı sınıf içerisinde de** gerçekleşebilir, üst ve alt sınıflarda da gerçekleşebilir
- Method override ise sadece üst sınıftaki metodun **alt sınıfta** tekrar tanımlanmasıdır

Metot Override vs Metot Overload

```
public class ModifyeBMW extends BMW {
```

```
@Override
```

```
public Integer hizlan() {  
    return hiz + 1;  
}
```

Method override

```
public Integer hizlan(Integer miktar) {  
    return hiz + miktar;  
}
```

Method overload

@Override Anotasyonu Ne İşe Yarar?

- Java SE5 ile gelen bir anotasyon **@Override** mevcuttur
- Bir metodu alt sınıfta override etmeye karar verdiğinizde alt sınıfta tanımlanan metot üzerinde bu anotasyonu kullanabilirsiniz
- Bu sayede derleyici metodu override etmek yerine **yanlışlıkla overload etmeniz durumunda hata verecektir**

Metot Override'da Dikkat Edilecek Noktalar

- **Metot ismi, parametrelerin sayısı, sırası ve tipleri** üst sınıftaki tanımla birebir aynı olmalıdır
- **Return tipi** ise ya aynı olmalıdır, ya da **üst sınıftaki metodun return tipi ile uyumlu** bir return tipi olmalıdır
- Örneğin Araba dönen bir metot, alt sınıfta override edildiği vakit Honda tipinde bir nesne de dönebilir

Metot Override'da Dikkat Edilecek Noktalar

- **Access modifier**'da ya üst sınıftaki metodun modifier'ı ile **aynı olmalıdır**, ya da **daha gevşek bir access modifier** olabilir
- Örneğin, üst sınıfta protected tanımlanmış bir metot alt sınıfta public tanımlanabilir
- Ancak tersi mümkün değildir

Metot Override'da Dikkat Edilecek Noktalar

- Eğer üst sınıftaki metot **checked exception fırlatıyor ise**, alt sınıftaki metot da
 - ya aynı exception'ları fırlatmalı
 - ya daha az sayıda exception fırlatmalı
 - ya da üst sınıfın fırlattığı checked exception'ın daha spesifik bir alt-tipi olmalıdır
- Alt sınıfta tanımlanan metot **herhangi bir unchecked exception** (RuntimeException) fırlatabilir

Constructor Overload

- Bir sınıf içerisinde **farklı input parametrelerle birden fazla constructor** tanımlanması işlemine constructor overload adı verilir
- Bu açıdan **metot overload** gibidir

Tip Dönüşümleri ve Upcast/Downcast İşlemleri

- Java **strongly-typed** bir dildir, yani tip dönüşümleri ile ilgili net kuralları vardır ve bu kurallar derleme zamanında kontrol edilir
- Eğer tip tanımları, değer atamalar ve tipler arasında dönüşümlerle ilgili **herhangi bir uyumsuzluk varsa derleyici hata verecektir**
- Tipler arasındaki dönüşümlere **upcast/downcast** adı verilir

Tip Dönüşümleri ve Upcast/Downcast İşlemleri

```
Araba kirmiziAraba = new  
Honda();
```

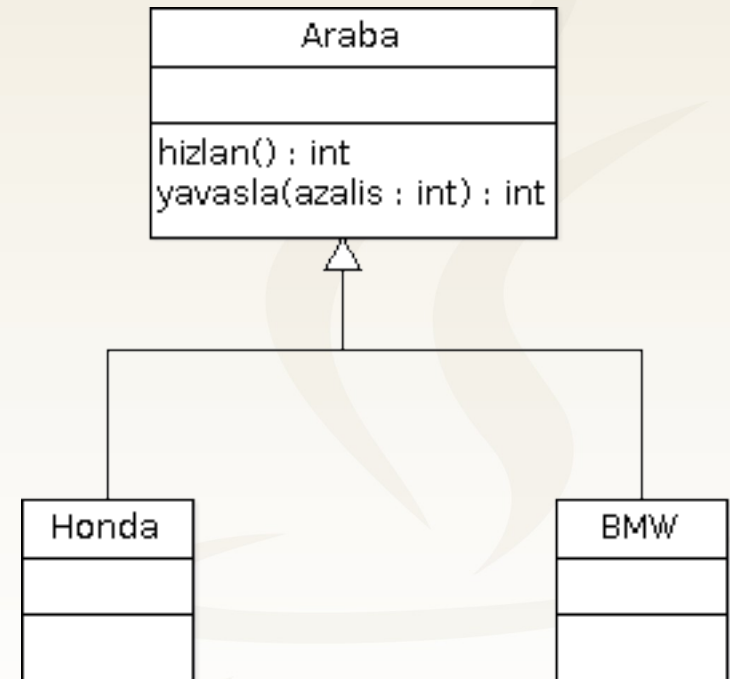
```
Araba yesilAraba = new  
BMW();
```

```
BMW bmw = (BMW)yesilAraba;
```

```
Honda honda = (Honda)  
kirmiziAraba;
```

```
yesilAraba = honda;
```

UPCAST (compiler tarafından otomatik yapılır)



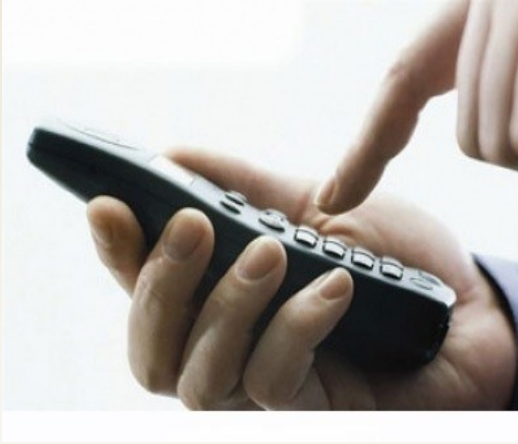
DOWNCAST (programcı tarafından explicit biçimde yapılmalıdır)



Tip Dönüşümleri ve Upcast/Downcast İşlemleri

- **Upcast** derleyici tarafından otomatik olarak gerçekleştiği için **çalışacağı garantidir**, hata olmaz
- Hata olsaydı derleme zamanında ortaya çıkardı
- Ancak **downcast** işlemi programcı tarafından uyumsuz tipler arasında yapılmış olabilir
- Böyle bir durumda **hata çalışma zamanında (runtime)** ortaya çıkacaktır

Interface Kavramı



- Telefonların **tuş takımları** vardır, telefon numarasını çevirmemizi sağlar
- Telefonların **mikrofonları** vardır, sözlerimizi diğer tarafa iletir
- Telefonların **hoparlörleri** vardır, karşı taraftakinin konuşmalarını duymamızı sağlar

Interface Nasıl Tanımlanır?

Interface, bir tipten nesnelerin **davranışlarının (kabiliyetlerini)** ne olacağını tanımlar

```
public interface Telefon {
    public void cevir(Long telefonNumarasi);
    public void mikrofon(String mesaj);
    public abstract String hoparlör();
}
```

interface anahtar kelimesi ile tanımlanırlar

İnterfacelerde opsiyonel'dir, interface'de tanımlanan metotlar default olarak public ve abstract modifier'larına sahiptir

Interface Nasıl Tanımlanır?

```
public interface Vasita {
    Integer hizlan();
    Integer yavasla(Integer azalis);
}

public abstract class Araba implements Vasita {
    protected Integer hiz = 0;

    public Integer yavasla(Integer azalis) {
        return hiz-=azalis;
    }

    public abstract Integer hizlan();
}

public class Toyota extends Araba {
    public Integer hizlan() {
        return hiz += 2;
    }
}
```

Interface ve Sınıflar

- Abstract sınıflar veya concrete sınıflar interface'leri **implement** edebilirler
- Bir interface başka bir interface'i **extend** edebilir
- Bir sınıf **birden fazla interface**'i implement edebilir
- Bir concrete sınıf sadece **tek bir** concrete sınıfı veya abstract sınıfı extend edebilir

Interface ve Default Metotlar

- Java 8'e kadar interface'lerde **sadece metot tanımları** yapılabilmekteydi
- Metotların implemantasyonu ise interface'i implement eden sınıflarda gerçekleştirilmekteydi
- **Java 8'de** interface içerisinde metot tanımının yanı sıra bu metot için **default bir implemantasyona** da izin verilmektedir

Interface ve Default Metotlar

- Böylece interface'i implement eden sınıflar **default metot davranışlarına** sahip olabilirler
- Bu sayede interface'e sonradan **bir metot tanımı eklendiğinde** bu interface'i implement eden sınıfların **metodu implement etme zarureti** ortadan kaldırılabilir

```
public interface LocaleResolver {  
    default Locale resolveLocale() {  
        return Locale.getDefault();  
    }  
}
```

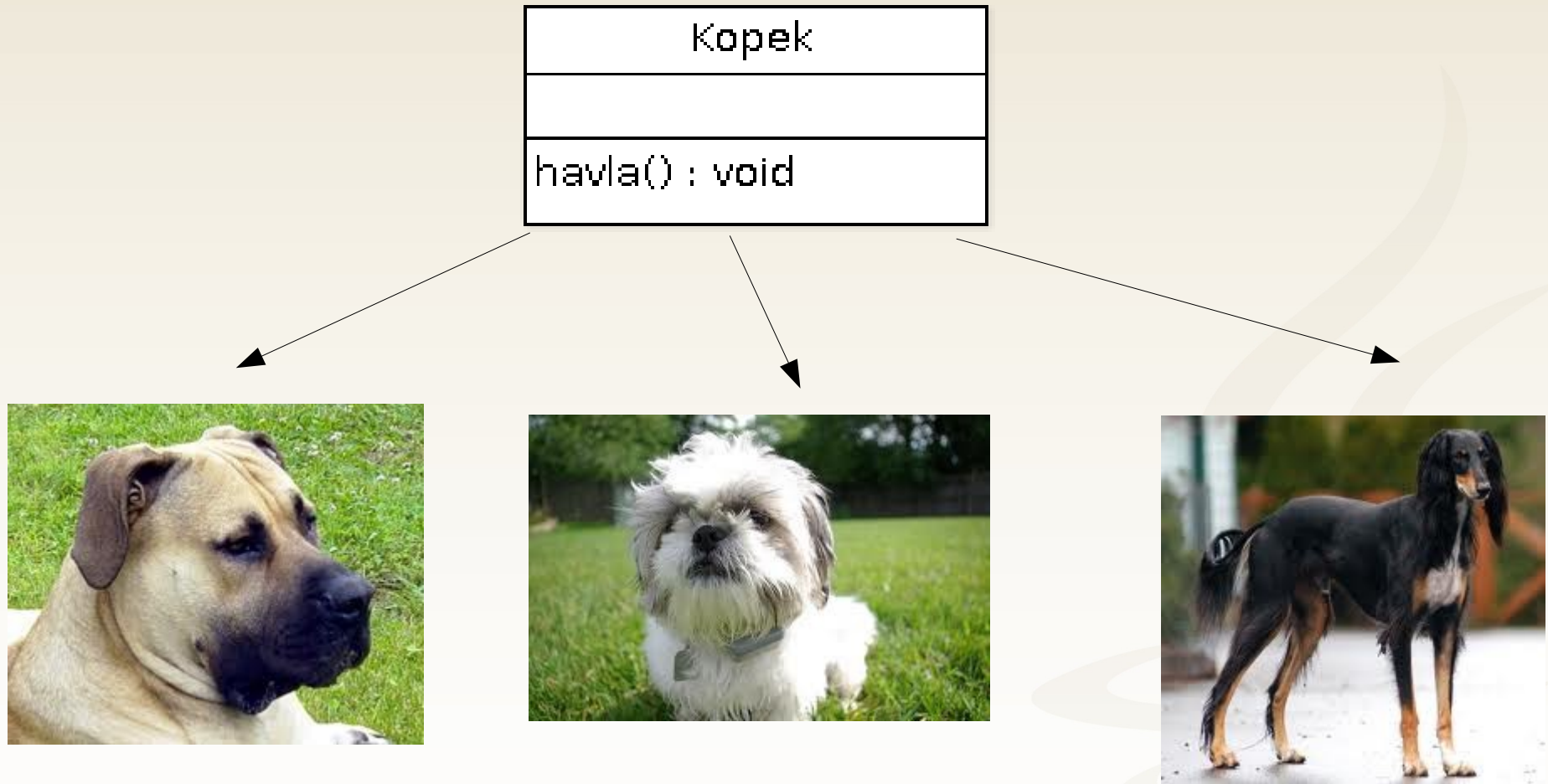
Interface ve Statik Metotlar

- Java 8 öncesi statik helper metotlar sadece sınıflarda yazılabilmekteydi
- Java 8 ile birlikte **statik metot tanımları** da interface'lerde yapılabilir

```
public interface LocaleResolver {
    static Locale getDefaultLocale() {
        return Locale.getDefault();
    }

    default Locale resolveLocale() {
        return getDefaultLocale();
    }
}
```

Çokformluluk(Polymorphism) Kavramı



Aynı davranışın (havlamak) farklı
biçimlerde/formlarda sergilenmesidir

Polymorphism Örneği

```
public interface Kopek {
    public void havla();
}
```

```
public class Kangal implements Kopek {
    @Override
    public void havla() {
        System.out.println("Kangal HAV HAV HAV...");
    }
}
```

```
public class Tazi implements Kopek {
    @Override
    public void havla() {
        System.out.println("Tazi hav hav havvv...");
    }
}
```

```
public class Dalmacyali implements Kopek {
    @Override
    public void havla() {
        System.out.println("Dalmacyalı hav hav hav...");
    }
}
```

Polymorphism Örneği

```
Kopek[] kopekler = new Kopek[]{  
    new Kangal(), new Dalmacyali(), new Tazi()};
```

```
for(Kopek kopek:kopekler) {
```

```
    kopek.havla();
```

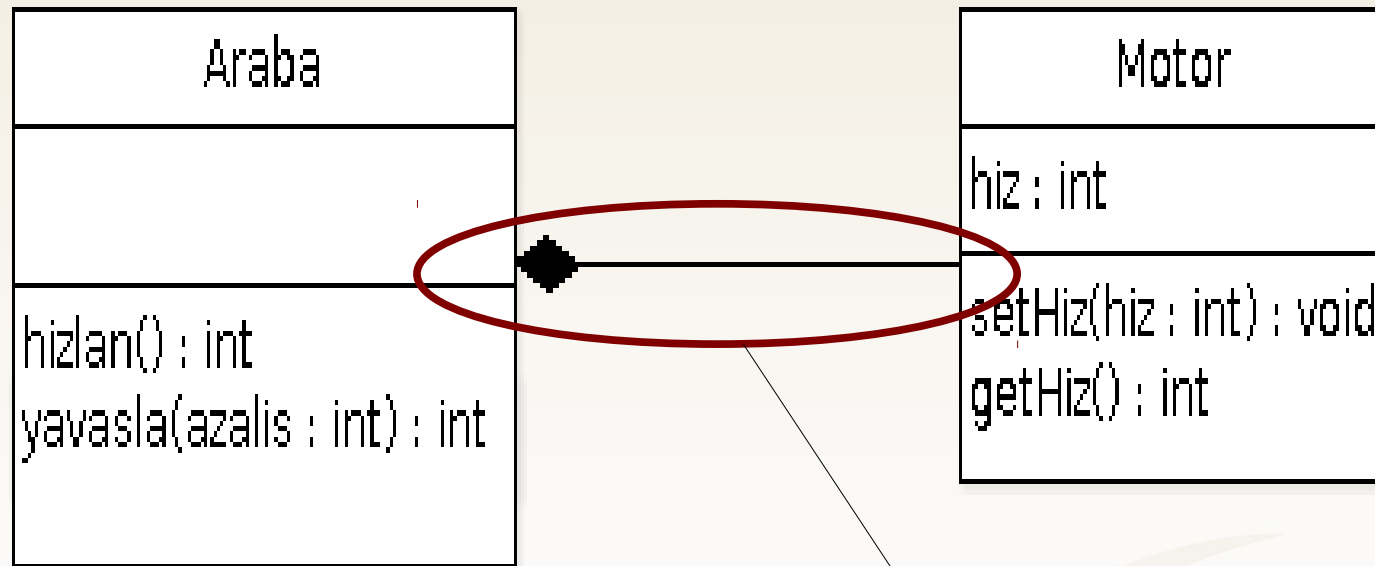
```
}
```

```
Kangal HAV HAV HAV...  
Dalmacyalı hav hav hav...  
Tazi hav hav havvv...
```

Composition Kavramı

- Bir sınıfın diğer bir takım sınıflar kullanılarak oluşturulması işlemine “**composition**” adı verilir
- Örneğin bir otomobilin bir motora sahip olması gibi

UML ve Composition



Composition HAS-A ilişkisidir
Car **has an** engine

Composition Örneği

```
public abstract class Araba implements Vasita {  
  
    private Motor motor;  
  
    public Integer yavasla(Integer azalis) {  
        Integer mevcutHiz = motor.getHiz();  
        motor.setHiz(mevcutHiz - azalis);  
        return motor.getHiz();  
    }  
  
    public void setMotor(Motor motor) {  
        this.motor = motor;  
    }  
  
    public abstract Integer hizlan();  
}
```

Composition Örneği

```
public class Motor {  
    private Integer hiz = 0;  
  
    public Integer getHiz() {  
        return hiz;  
    }  
  
    public void setHiz(Integer hiz) {  
        this.hiz = hiz;  
    }  
}
```

```
Araba kirmiziAraba = new Honda();  
Motor motor = new Motor();  
kirmiziAraba.setMotor(motor);
```

Scope Kavramı

```

public abstract class Araba implements Vasita {
    protected Motor motor;
    private String adi;

    public Araba() {
    }

    public Araba(String adi) {
        this.adi = adi;
    }

    public Integer yavasla(Integer azalis) {
        Integer mevcutHiz = motor.getHiz();
        motor.setHiz(mevcutHiz - azalis);
        return motor.getHiz();
    }

    public void setMotor(Motor motor) {
        this.motor = motor;
    }

    public abstract Integer hizlan();
}

```

constructor scope

instance method scope

Statik ve Instance Bloklar

```

public class Honda extends Araba {

    static {
        Date yuklemeZamani = new Date();
        System.out.println("Yükleme Zamanı : " +
            yuklemeZamani);
    }

    {
        Date imalatTarihi = new Date();
        String sasiNo = "SN" +
            imalatTarihi.getTime();
        System.out.println("Şasi No : " + sasiNo);
    }

    public Honda() {
        super("Honda");
        System.out.println("honda constructor");
    }
}

```

Statik initialization blok
(sınıf yüklenirken bir kere çalıştırılır)

instance initialization blok
(her yeni nesne yaratılışında çalıştırılır)

Sıralamada initialization bloklardan sonra çalışır

Final Sınıf, Method ve Attribute'lar

- Final sınıf **extend** edilemez
- Final metot **override** edilemez
- Final değişkenin **değeri** değiştirilemez
- Final değişkenin değeri nesnenin **yaratılma anında** veya sınıfın **yüklenme anında** set edilir
- Final değişkenin değeri nesne ise bu nesnenin **attribute'larının değerlerini** değiştirmede hiçbir kısıt yoktur

Nesne Yaratım Süreci ve Kalıtım

- Eğer nesnesi oluşturulan sınıf başka bir sınıfı extend ediyorsa **önce superclass** yüklenir ve initialize edilir
- Initialization **en üst sınıftan** başlar
- Önce **statik bloklar** ve **statik değişkenler** initialize edilir
- Bu hiyerarşideki **en alt sınıfa kadar** böyle devam eder

Nesne Yaratım Süreci ve Kalıtım

- Ardından sıra ile **hiyerarşide bir üstteki instance blok ve constructor'lar** çağrılır
- En üst seviyeye varıldığında **önce instance blok ardından constructor** çalışır
- Bu işlem hiyerarşide **en alt seviyeye kadar** böyle tekrarlanır

İletişim



www.harezmi.com.tr

www.java-egitimleri.com



info@harezmi.com.tr

info@java-egitimleri.com



[@HarezmiBilisim](https://twitter.com/HarezmiBilisim)

[@JavaEgitimleri](https://twitter.com/JavaEgitimleri)