

# Kotlin OOP Eğitimi

## Nesne Tabanlı Programlama

Kasım ADALAN

Elektronik ve Haberleşme Mühendisi  
Freelance Software Developer

# Eğitim içeriği

1. Sınıf ( Class ve Structure ) Nedir ?
2. Nesne ( Object ) Nedir ?
3. Nullable Type
4. Fonksiyonlar - Metodlar
5. Initialization – Constructor
  - Shadowing ( Gölgeleme )
6. Class vs Structure Farkı
7. Import
8. Visibility Modifier
9. Data Class
10. Static değişken & metodlar
11. Enumeration
12. Composition
13. Kalıtım
14. Kalıtım İlişkisinde Constructor
15. Metodları Ezme ( Override)
16. PolyMorphism
17. Nesnelerin Tip Dönüşümü
18. Interface

# Nesne Tabanlı Programlama

# Nesne Tabanlı Programlama

- Hayatlarımız nesneler çevresinde kuruludur



- Bu nesneleri soyutlayarak yazılım projelerine yansıtırız
- Birden çok kez kullanım için nesneler soyutlanarak bilgisayar koduna dönüştürülür
- Oluşan soyut taslaklara sınıf (class) denir

# Sınıf ( Class ) Nedir ?

- Araba Analojisi
  - Mühendisler yeni bir araba üretmek için öncelikle proje planları oluşturur
  - Benzin emisyonu, motorun nasıl çalıştığı gibi ayrıntılar bu planlara yansıtılır
  - Planlar arabanın nasıl hareket edeceği, arabayı oluşturacak parçalar gibi birçok detayı içerir
- Herhangi bir sürücünün tüm bu detayları bilmesine gerek var mıdır?
- Hayır! Yalnızca ehliyetinin olması ve arabayı sürmeyi bilmesi yeterlidir.

# Nesne ( Object ) Nedir ?

- Nesneler sınıfların somutlaşmış halleridir
- Nesneleri durumu (state) ve davranış biçimleri vardır (behaviour)



Arabanın  
-renk, hız, kapasite } Durum (state)  
-Hızlanmak ve  
yavaşlamak için pedallar } Davranış (behaviour)

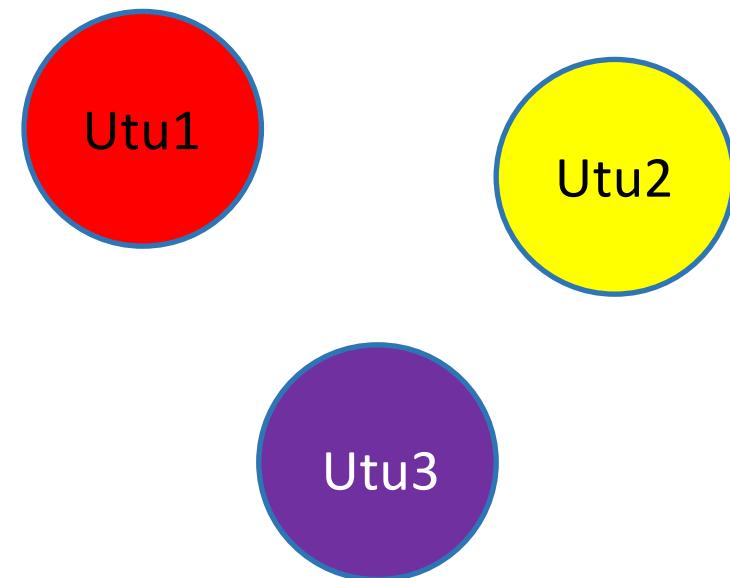
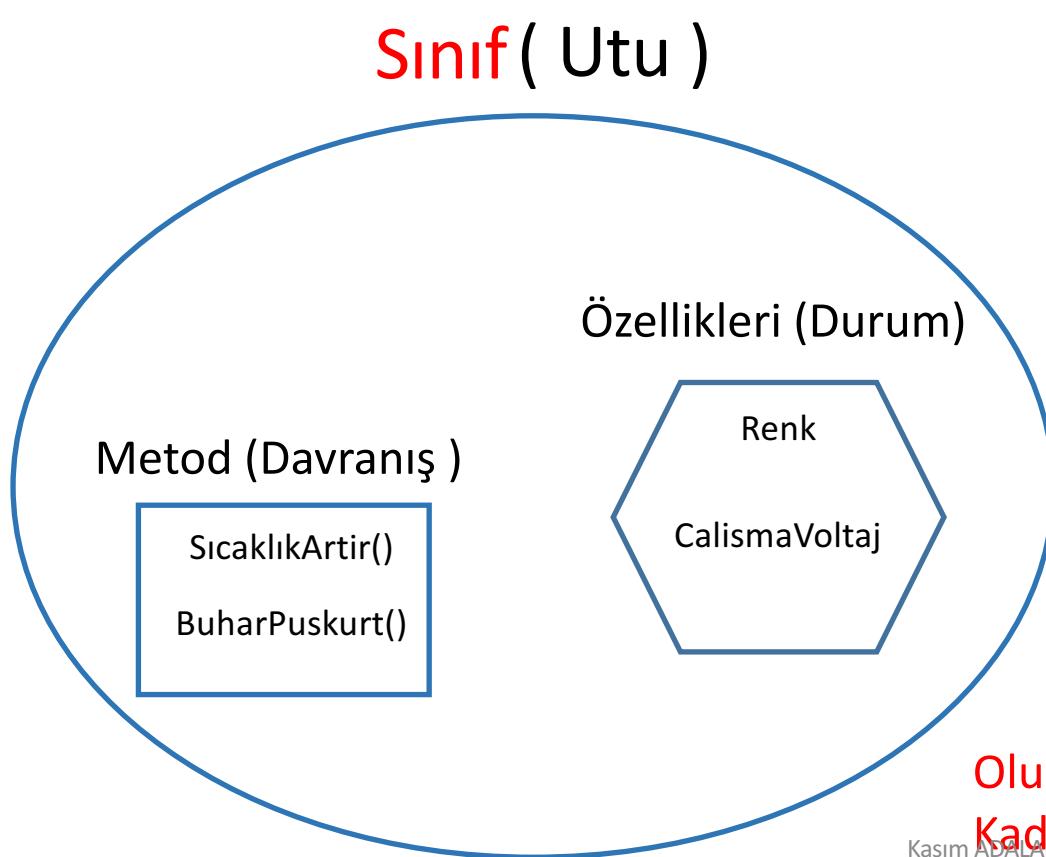
- Sınıflar ise nesnelerin özellikleri ve davranışları ile ilgili ayrıntıları içerir Araba sınıfı.  
-Frene basıldığında ne olur?



# Nesnenin Durumu ve Davranışı

- **Durum (State):** Sınıfların bir – birçok özelliği olabilir
  - Somut değişkenler (instance variables) belirler
  - Nesneyle birlikte taşınır
- **Davranış (Behavior):** Sınıflar bir ya da birden çok metoda sahip olabilir
  - Metod program içindeki bir işi temsil eder
    - Görevlerin gerçekleştirileceği adımları tanımlar
    - Kullanıcıdan kompleks işlemleri gizler
    - Metodu çağrırmak, metodun bu işlemleri gerçekleştirmesini sağlar

# Nesne ( Object )



Oluşturduğumuz Ütü sınıfı taslağında istediğimiz Kadar Nesne üretebiliriz.

## Class Tanımlama

Keywords

class Araba {

İsim

}

# Nesne Tanımlama

```
class Araba {  
    var renk = "kırmızı"  
    var kapasite = 4  
}
```

Nesnenin  
Adı

---

var bmw = Araba()

Nesnenin  
Oluşturulduğu  
Sınıf

---

# Class Yapısına Erişim.

- Class yapısı içindeki metod ve özelliklerine erişmek için **ilk şart** bulunduğu Class ' dan nesne oluşturmalı.

```
class Araba {  
    var renk = "kırmızı"  
    var kapasite = 4  
}
```

```
val bmw = Araba()  
  
println(bmw.renk)//Kırmızı  
println(bmw.kapasite)//4  
  
//Yeni değer yükleme  
bmw.renk = "yeni kırmızı"  
bmw.kapasite = 99  
  
println(bmw.renk)//yeni kırmızı  
println(bmw.kapasite)//99
```

# Class Yapısına Erişim.

- Her nesne kendine ait özelliklere erişebilir ve veri aktarımı yapabilir.

```
//Nesne oluşturma  
val bmw = Araba()
```

```
//Yeni değer yükleme  
bmw.renk = "mavi"  
bmw.kapasite = 2
```

```
println(bmw.renk)//mavi  
println(bmw.kapasite)//2
```

```
//Nesne oluşturma  
val limuzin = Araba()
```

```
//Yeni değer yükleme  
limuzin.renk = "beyaz"  
limuzin.kapasite = 10
```

```
println(limuzin.renk)//beyaz  
println(limuzin.kapasite)//10
```

# Değeri olmayan özellikle Class oluşturma

- Boş değerli oluşturmak için constructor içerisine tanımlama yapmalıyız.
- Aksi halde boş değer tanımlayamıyoruz.

```
class Araba ( var renk:String ,var kapasite:Int ) {  
}  
  
val bmw = Araba("mavi",2)  
//Nesne oluşturma  
//Değerler nesne oluştururken aktarılır.
```

# Değeri olmayan özellikle Class oluşturma

```
class Araba ( var renk:String ,var kapasite:Int ) {  
}
```

```
val bmw = Araba("mavi",2)  
//Nesne oluşturma  
//Değerler nesne oluştururken aktarılır.  
  
println(bmw.renk)//mavi  
println(bmw.kapasite)//2
```

```
val limuzin = Araba("beyaz",10)  
//Nesne oluşturma  
//Değerler nesne oluştururken aktarılır.  
  
println(limuzin.renk)//beyaz  
println(limuzin.kapasite)//10
```

Özellikler **val** olursa ilk değer ataması olduktan sonra tekrar atama olmaz.

```
class Araba {  
    val renk = "Kırmızı"  
    val kapasite = 4  
}  
  
//Nesne  
val bmw = Araba()  
  
println(bmw.renk)//Kırmızı  
println(bmw.kapasite)//4  
  
// renk ve kapasite val olduğu için constructor içinde ilk  
//değer ataması yapılmıştır.Daha sonra değeri değiştiremez.  
bmw.renk = "yeni kırmızı"  
bmw.kapasite = 99  
  
println(bmw.renk)//yeni kırmızı  
println(bmw.kapasite)//99
```

```
class Araba ( val renk:String ,val kapasite:Int ) {  
}  
  
val bmw = Araba("mavi",2)  
//Nesne oluşturma  
//Değerler nesne oluştururken aktarılır.  
  
println(bmw.renk)//mavi  
println(bmw.kapasite)//2  
  
// renk ve kapasite val olduğu için constructor içinde ilk  
//değer ataması yapılmıştır.Daha sonra değeri değiştiremez.  
bmw.renk = "yeni mavi"  
bmw.kapasite = 99  
  
println(bmw.renk)//yeni mavi  
println(bmw.kapasite)//99
```

# Özellikler

# Nullable Type ?

- Global değişken oluştururken değişkene belirli bir değer vermeden oluşturmak isteyebiliriz.
  - Örn : `var str:String = 2` gibi bir değer yerine `var str:String = null`
- Bu durumda null yapabiliriz.
- Değişkeni **?** İşareti ile tanımlamalıyız.
- Tanımlandıktan sonra kullanılırken **?** İşareti kullanılırsa null hatasından kaynaklı olabilecek çökmelerden korunur.
- Tanımlandıktan sonra kullanılırken **!!** İşareti kullanılırsa bu değişkenin null olabileceğini belirtmiş oluruz.

# Null safety

KOTLIN

```
var str1: String? = null  
str1?.trim() // doesn't run
```

```
str1 = "Not null anymore"  
str1?.trim() // does run
```

```
str1!!.trim() // runs anyway
```

```
val str2: String = "I am not null"  
str2.trim() // no need for "?."
```

JAVA

```
String str1 = null;  
str1.trim(); // runs and crashes
```

```
str1 = "Not null anymore";  
str1.trim(); // runs
```

```
String str2 = "I am not null";  
str2.trim(); // runs
```

# Nullable Type ? Kullanım Örneği

Global değişken oluştururken değişkene bir değer vermeden oluşturmak isteyebiliriz.

```
var showAnswerButton: Button? = null
```

```
showAnswerButton = findViewById(R.id.showAnswerButton)
```

Tanımlandıktan sonra kullanılırken ? İşareti kullanılırsa null hatasından kaynaklı olabilecek çökmelerden korunur

```
showAnswerButton?.setOnClickListener { /* */ }
```

Javadaki karşılığı

```
if (showAnswerButton != null) {  
    showAnswerButton.setOnClickListener(/* */);  
}
```

Kotlindeki !! karşılığı

```
showAnswerButton!!.setOnClickListener { /* */ }
```

Çökme ihtimalini bilerek !! ile null olabileceği belirtilmiştir.

## lateinit

- Bir değişken oluştururken hiç bir değer atamak istemeyebiliriz.
  - Örn : **lateinit var a:Sinif** gibi
- Bu durumda lateinit keyword'ü kullanılabılır.
- Lateinit ile birlikte **?** veya **!!** kullanmak zorunda değiliz fakat değişken null olduğunda çökme olabilir bunu göze almalıyız.
- Primitif tipler için kullanılamaz.
- Referans tiplerinde geçerlidir yani nesne olmalıdır.

# lateinit Kullanım Örneği

Global değişken oluştururken değişkene hiç bir değer vermeden oluşturmak isteyebiliriz.

```
lateinit var showAnswerButton: Button  
  
showAnswerButton = findViewById(R.id.showAnswerButton)  
  
showAnswerButton.setOnClickListener { /* */ }
```

# Fonksiyonlar - Metodlar

# Fonksiyonlar

- Belirli işlemleri temsil eden yapılardır.
- Kullanma amacımız tekrarlanan kod yapılarının önüne geçmektir.
- **fun** kelimesi tanımlanırlar.
- Programlamayı daha pratik bir hale getirir.
- Kodun okunmasına faydası vardır.
- Class içinde yer alan fonksiyonlara denir.
- Özellikler gibi bulunduğu Class veya Structure'dan nesne oluşturulursa erişilebilir.

```
fun    fonksiyon adı ( Parametre ) : dönüş türü {
```

```
//Kodlama buraya yazılır
```

```
return dönüş değeri
```

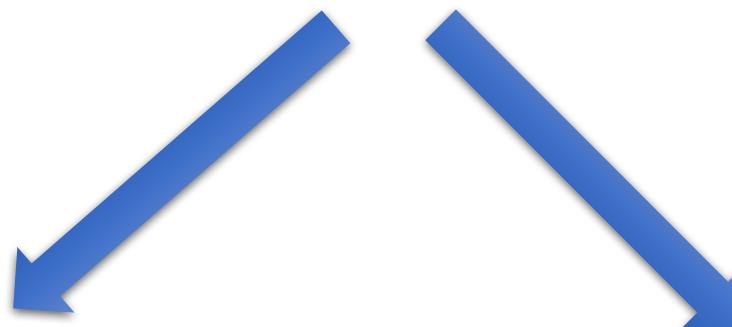
```
}
```

# Class Metodlarına Erişim.

```
class Araba (var renk:String,var kapasite:Int) {  
  
    fun calistir(){  
        println("Araba çalıştı")  
    }  
  
    fun durdur(){  
        println("Araba durdu")  
    }  
}
```

```
var bmw = Araba(renk: "kirmizi", kapasite: 5)  
  
//Atanan değerleri alma  
  
println(bmw.renk) // kirmizi  
println(bmw.kapasite) // 5  
  
//Metodlara erişim  
  
bmw.calistir() // Araba çalıştı  
  
bmw.durdur() //Araba durdu
```

## Foksiyonlar



Geri dönüş değeri olan

Geri dönüş değeri olmayan

# Geri Dönüş değeri olmayan fonksiyonlar

- Sadece yaptırılmak istenen işlemi yapan metodу kullanana veri döndürmeyen fonksiyonlardır.

```
fun selamla(){
    val sonuc = "Merhaba Ahmet"
    println(sonuc)
}

selamla()
```

# Geri Dönüş değeri olan fonksiyonlar

- Yapılan işlem sonucunda metodu kullanana veri dönüşü yapan fonksiyonlardır.

```
fun selamla() : String {  
    val sonuc = "Merhaba Ahmet"  
    return sonuc  
}
```

```
val gelenSonuc = selamla()
```

```
println(gelenSonuc)
```

# Fonksiyonların Parametre Alması

- Parametre fonksiyonlara dışarıdan verilen değerlerdir.
- Her fonksiyonun parametresi olmak zorunda değildir.
- Parametreler tanımlaması değişkeni tanımlar gibidir.
- Parametreler , virgül ile birden fazla tanımlanabilir.      (parametre)

- : işaretini ile ismi ve türü belirlenir.

• mesaj : String  
parametre adı      parametre türü

```
fun selamla(isim:String) : String {  
    val sonuc = "Merhaba $isim"  
    return sonuc  
}  
  
val gelenSonuc = selamla( isim: "Ahmet")  
println(gelenSonuc)
```

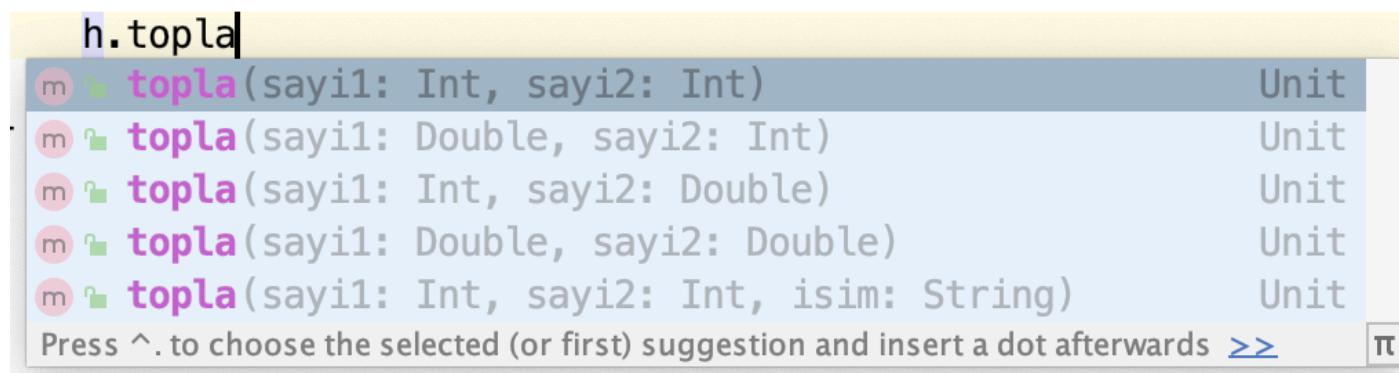
# Fonksiyonların Parametre Alması

- Birden fazla parametre kullanılabilir.

```
fun topla(sayı1:Int,sayı2:Int) : Int {  
    return sayı1 + sayı2  
}  
  
val gelenSonuc = topla( sayı1: 20, sayı2: 30)  
    println(gelenSonuc)
```

# Metodların Aşırı Yüklenmesi : Overloading

- Metodların aynı isimde tekrar kullanılması.
- Tek şart metodun parametre dizilimindeki türlerin farklı olmasıdır.
- Metodların kullanımında parametre çeşitliliği sağlar.



# Metodların Aşırı Yüklenmesi : Overloading

```
class Hesaplayici {

    fun topla(sayı1:Int,sayı2:Int) {
        println("Toplam : ${sayı1 + sayı2}")
    }

    fun topla(sayı1:Double,sayı2:Int) {
        println("Toplam : ${sayı1 + sayı2}")
    }

    fun topla(sayı1:Int,sayı2:Double) {
        println("Toplam : ${sayı1 + sayı2}")
    }

    fun topla(sayı1:Double,sayı2:Double) {
        println("Toplam : ${sayı1 + sayı2}")
    }

    fun topla(sayı1:Int,sayı2:Int,isim:String) {
        println("Toplama yapan : $isim , sonuç : ${sayı1 + sayı2}")
    }
}
```

```
val h = Hesaplayici()

h.topla( sayı1: 4, sayı2: 5, isim: "Ahmet")
```

# vararg Parametreler

- İstenildiği kadar veri girilebilen parametre

```
fun toplamVararg(vararg sayilar:Int) : Int {  
    var toplam = 0  
  
    for ( s in sayilar ) {  
        toplam = toplam +s  
    }  
  
    return toplam  
}  
  
val v1 = toplamVararg( ...sayilar: 1,2,3,4,5)  
  
println(v1)//15
```

# Parametrelerde Default (başlangıç) Değeri

- İstenirse parametrelere varsayılan değer verilebilir.
- Fonksiyonu kullanan kişi parametreyi koymaz ise varsayılan değer çalışır.
- Normal parametre tanımlama işlemi gibi olur ve değer verilir.

```
fun topla(sayi1:Int,sayi2:Int,sayi3:Int = 2) : Int {  
    return sayi1 + sayi2 + sayi3  
}  
  
//3. parametre girilmez ise varsayılan değer kullanılır.  
  
val t1 = topla( sayi1: 10, sayi2: 20)  
  
println(t1)//32  
  
//3. parametre girilir ise varsayılan değer kullanılmaz ve girilen parametre kullanılır.  
  
val t2 = topla( sayi1: 10, sayi2: 20, sayi3: 30)  
  
println(t2)//60
```

# Extension Fonksiyon

- Extension fonksiyonlar kotlin içinde bulunan sınıfları genişletmek için kullanılan pratik bir yöntemdir.
- Örneğin int sınıfına özellik katabiliriz.

```
fun Int.carpi(sayı: Int) : Int{  
    return this * sayı  
    //this metodun isminin önündeki sayıyı ifade eder.  
}
```

```
val sonuc = 5.carpi(sayı: 2)
```

```
println(sonuc)//10
```

# infix Fonksiyon

- Extension fonksiyonları daha pratik kullanmamızı sağlar.

```
infix fun Int.carpı(sayı: Int) : Int{  
    return this * sayı  
    //this metodun isminin önündeki sayıyı ifade eder.  
}
```

extension karşılığı

```
val sonuc = 5 carpı 2  
val sonuc = 5.carpı( sayı: 2)
```

```
println(sonuc)//10
```

# Nesne Tabanlı ÖDEVLER

# Ödevler

1. Parametre olarak girilen dereceyi fahrenheit'a dönüştürdükten sonra geri döndüren bir metod yazınız.  $T_{(^\circ\text{F})} = T_{(^\circ\text{C})} \times 1.8 + 32$
2. Kenarları parametre olarak girilen ve dikdörtgenin çevresini hesaplayan bir metod yazınız..
3. Parametre olarak girilen sayının faktoriyel değerini hesaplayıp geri döndüren metodu yazınız.
4. Parametre olarak girilen kelime ve harf için harfin kelime içinde kaç adet olduğunu gösteren bir metod yazınız.

# Ödevler

5. Parametre olarak girilen kenar sayısına göre iç açılar toplamını hesaplayıp sonucu geri gönderen metod yazınız.

Formül n: kenar sayısı  $(n-2) \cdot 180$

6 . Parametre olarak girilen gün sayısına göre maaş hesabı yapan ve elde edilen değeri geri döndüren metod yazınız.

1 Günde 8 saat çalışılabilir.

Çalışma saat ücreti : 10 tl

Mesai saat ücreti : 20tl

160 saat üzeri mesai sayılır.

7. Parametre olarak girilen kota miktarına göre ücreti hesaplayarak geri döndüren metodu yazınız.

- 50GB 100 TL
- Kota aşımından sonra her 1GB 4 TL

# Constructor

# Constructor

- Bir sınıfından ( class ) nesne oluşturmak için gerekli olan yapıdır.
- **constructor** kelimesi ile tanımlanır.
- Bir sınıfından ( class ) nesne oluşturma işleminde parametre alabilir.
- Nesne oluşurken istenilen kodlamalar bu metod içinde yapılabilir.

```
class Kisiler {  
    var ad:String = "Ahmet"  
    var yas:Int = 18  
}
```

```
val kisi = Kisiler()  
println(kisi.ad)  
println(kisi.yas)
```

Bu parantezler boş constructor'ı temsil etmektedirler.

# Primary Constructor

- Nesne oluştururken değişkenlere veri aktarmak için kullanılır.
- Değişken atama işleminden başka bir işlem yapılmaz.
- Bu yapı ile javadaki setter getter metodlarına ihtiyaç yoktur.

```
class Kisiler(var ad:String, var yas:Int) {  
}  
  
val kisi = Kisiler(ad: "Ahmet", yas: 18)  
    println(kisi.ad)  
    println(kisi.yas)
```

Nesne oluşturulurken  
değer aktarılmıştır.

# Primary Constructor için `init` kullanımı

- Nesne oluşturulurken değişkenlere değer atamanın haricinde başka bir işlem yapmak için kullanılır.
- Nesne oluşturulduğu anda çalışır.

```
class Kisiler(var ad:String,var yas:Int) {  
    init {  
        println("Primary Constructor Çalıştı")  
    }  
}
```

```
val kisi = Kisiler(ad: "Ahmet", yas: 18)  
println(kisi.ad)      Nesne olduğu anda  
println(kisi.yas)    init çalışır.
```

ÇIKTI :

```
Primary Constructor Çalıştı  
Ahmet  
18
```

# Secondary Constructor

- Primary Constructor değişken atama işleminden başka bir işlem yapılmaz.
- Secondary Constructor içinde istenilen kodlama yapılır ve nesne oluşturulduğunda çalışır.
- Primary Constructor gibi nesne oluştururken değişkenlere veri aktarmak mümkündür aynı zamanda başka kodlamada yapılabilir.

```
class Kisiler {  
    var ad:String  
    var yas:Int  
  
    constructor(ad:String,yas:Int){  
        this.ad = ad  
        this.yas = yas  
  
        println("Secondary Constructor Çalıştı")  
    }  
}
```

```
val kisi = Kisiler( ad: "Ahmet", yas: 18)  
println(kisi.ad)      Nesne olduğu anda  
println(kisi.yas)    init çalışır.
```

ÇIKTI :

```
Secondary Constructor Çalıştı  
Ahmet  
18
```

# Shadowing - Gölgeleme

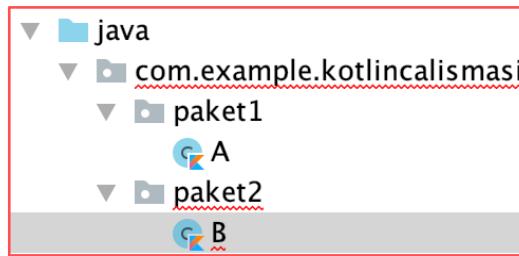
- Bir metod içinde aynı isimde oluşturulmuş global değişkeni lokal değişken gölgelemektedir.
- Bu gölgelemeyi engellemek için metod içinde global değişken self kelimesi tanımlanır.

```
class Kisiler {  
    var ad:String  
    var yas:Int  
  
    constructor(ad:String,yas:Int){  
        this.ad = ad  
        this.yas = yas  
  
        println("Secondary Constructor Çalıştı")  
    }  
}
```

# Paketler & Import

# Paketler & Import

- **Paketler** birden fazla sınıfı kümlelediğimiz yapılardır.
- **Paketler** ile daha düzenli projeler oluşturulabilir.
- **Import** bir sınıfı başka bir sınıf içinde kullanıcaksa o sınıfı **import** etmeliyiz.
- Bir sınıfın tüm uzantısı kullanıldığında **import'a** gerek yoktur.



```

package com.example.kotlincalismasi.paket2

import com.example.kotlincalismasi.paket1.A

class B {

    fun fonksiyon() {
        val nesne = A()

        println(nesne.varsayılanDegişken)
        println(nesne.publicDegişken)
        println(nesne.privateDegişken) //Hata : Aynı sınıf içinde olmadığımız için
        println(nesne.internalDegişken)
        println(nesne.protectedDegişken)//Hata : Aynı sınıf veya bulunduğu sınıfın türemediği için
    }
}
  
```

# Visibility Modifier

# Visibility Modifier ( Erişim )

<ul style="list-style-type: none"><li>• <b>Public</b><ul style="list-style-type: none"><li>• Sınıf</li><li>• Metod</li><li>• Değişken</li></ul></li><li>• (Bütün sınıflardan ve paketlerden ulaşılabilir.)</li></ul>	<ul style="list-style-type: none"><li>• <b>Internal</b><ul style="list-style-type: none"><li>• Metod</li><li>• Değişken</li></ul></li><li>• (Yalnızca bulunduğu modül içinde yer alan sınıflar ulaşılabilir.)</li><li>• Paket ile modül aynı kavram değildir.</li></ul>	<ul style="list-style-type: none"><li>• <b>Private</b><ul style="list-style-type: none"><li>• Metod</li><li>• Değişken</li></ul></li><li>• (Yalnızca tanımlandıkları sınıfın içinden ulaşılabilir.)</li></ul>	<ul style="list-style-type: none"><li>• <b>Protected</b><ul style="list-style-type: none"><li>• Metod</li><li>• Değişken</li></ul></li><li>• (Bulundukları sınıfın içinden veya bulundukları sınıfın türetildiklerinde ulaşılabilir.)</li></ul>
--	---	---	---

**Not :** Hiçbir access modifier kullanılmiyorsa public anlamına gelir.

# Visibility Modifier Örnek

```
class A {  
    var varsayılanDegisen = 1  
    public var publicDegisen = 2  
    private var privateDegisen = 3  
    internal var internalDegisen = 4  
    protected var protectedDegisen = 5  
}
```

```
class B {  
  
    fun fonksiyon() {  
  
        val nesne = A()  
  
        println(nesne.varsayılanDegisen)  
        println(nesne.publicDegisen)  
        println(nesne.privateDegisen) //Hata : Aynı sınıf içinde olmadığımız için  
        println(nesne.internalDegisen)  
        println(nesne.protectedDegisen)//Hata : Aynı sınıf veya bulunduğu sınıfın从中 türemediği için  
    }  
}
```

# Visibility Modifier Örnek

Eğer çalıştığımız sınıfı protected değişkenin bulunduğu sınıfın türetirsek protected hatası kalkar.

```
open class A {  
    var varsayılanDegisen = 1  
    public var publicDegisen = 2  
    private var privateDegisen = 3  
    internal var internalDegisen = 4  
    protected var protectedDegisen = 5  
}
```

```
class B : A() {  
  
    //Kalıtım olduğu için A sınıfı içindeki değişkenlere direkt erişim sağlanabilir.  
  
    fun fonksiyon() {  
  
        println(varsayılanDegisen)  
        println(publicDegisen)  
        println(privateDegisen) //Hata : Aynı sınıf içinde olmadığımdır için  
        println(internalDegisen)  
        println(protectedDegisen).  
    }  
}
```

# Visibility Modifier Örnek

Eğer değişkenleri kendi sınıfı içinden çağrırsak hiç sorun yaşamayız.

```
class A {  
    var varsayılanDegisen = 1  
    public var publicDegisen = 2  
    private var privateDegisen = 3  
    internal var internalDegisen = 4  
    protected var protectedDegisen = 5  
  
    fun fonksiyon() {  
  
        println(varsayılanDegisen)  
        println(publicDegisen)  
        println(privateDegisen)  
        println(internalDegisen)  
        println(protectedDegisen)  
    }  
}
```

# Data Class

# Data Class

- Java üzerinde çalışma yaparken oluşturduğumu model sınıflarında dolu constructor,setter getter>equals,hascode gibi yapıları kendimiz koymak zorundaydık.
- Kotlin üzerinde sınıfı data keyword'u eklendiğinde bu özelliklerin hepsine sahip olmuş oluyor.
- Genelde veritabanı tablolarının modelleri için kullanılır.

```
data class Kisiler(var ad:String) {  
}
```

# Java ile Data Class Karşılaştırması

JAVA

```
public class Person {  
    private String ad;  
  
    public Person(String ad) {  
        this.ad = ad;  
    }  
  
    public String getAd() {  
        return ad;  
    }  
  
    public void setAd(String ad) {  
        this.ad = ad;  
    }  
  
    //toString  
    //hashCode  
    //equals  
}
```

Kotlin

```
data class Kisiler(var ad:String) {  
}
```

# Static Değişkenler ve Metodlar

# Static Değişkenler ve Metodlar

- Bir değişkenin veya metodun, bulunduğu sınıfın ( class ) nesne oluşturmaya gerek kalmadan erişilmek istenirse kullanılır.

```
class Asinifi {  
  
    companion object {  
        //Direk erişim için bu alan içine kodlama yapılır.  
        var x = 10  
  
        fun metod(){  
            println("Merhaba")  
        }  
    }  
}
```

```
println(Asinifi.x)  
Asinifi.metod()
```

# Enumeration

# Enumeration

- **enum** ifadesi gösterilir.
- Parametrelerde kullanılır.
- Verilerin eşleşmesi sonucunda bir işlem yapılır.
- Kodlama yapan yazılımcıyı detaydan kurtarmaktadır.

```
enum class Renkler {  
    Beyaz, Siyah  
}
```

```
val renk = Renkler.Beyaz  
  
when (renk) {  
    Renkler.Beyaz -> print("#FFFFFF")  
    Renkler.Siyah -> print("#000000")  
}
```

# Composition

# Composition

- Başka sınıflardan ( class ) olmuş nesneler bir sınıfın değişkeni olabilir.

```
class Adres(var il:String,var ilce:String) {  
}
```

```
class Kisiler(var ad:String,var yas:Int,var adres:Adres) {  
}
```

```
val adres = Adres( il: "Bursa", ilce: "Osmangazi")  
  
val kisi = Kisiler( ad: "Ahmet", yas: 18,adres)  
  
println("Kişi ad    : ${kisi.ad}")  
println("Kişi yaş   : ${kisi.yas}")  
println("Adres il   : ${kisi.adres.il}")  
println("Adres ilçe : ${kisi.adres.ilce}")
```

# Composition

Kategoriler Tablosu

kategori_id	kategori_ad
1	Dram
2	Komedi
3	Bilim Kurgu

Yonetmenler Tablosu

yonetmen_id	yonetmen_ad
1	Nuri Bilge Ceylan
2	Quentin Tarantino
3	2013

Filmler Tablosu

film_id	film_ad	film_yil	kategori_id	yonetmen_id
1	Django	2013	1	2
2	Inception	2006	3	3

```
data class Filmler(  
    var film_id:Int,  
    var film_ad:String,  
    var film_yil:Int,  
    var kategori: Kategoriler,  
    var yonetmen: Yonetmenler) {  
}
```

```
data class Yonetmenler(  
    var yonetmen_id:Int,  
    var yonetmen_ad:String) {  
}
```

```
data class Kategoriler(  
    var kategori_id:Int,  
    var kategori_ad:String) {  
}
```

```
val k1 = Kategoriler( kategori_id: 1, kategori_ad: "Dram")  
val k2 = Kategoriler( kategori_id: 2, kategori_ad: "Komedi")  
val k3 = Kategoriler( kategori_id: 3, kategori_ad: "Bilim Kurgu")  
  
val y1 = Yonetmenler( yonetmen_id: 1, yonetmen_ad: "Nuri Bilge Ceylan")  
val y2 = Yonetmenler( yonetmen_id: 2, yonetmen_ad: "Quentin Tarantino")  
val y3 = Yonetmenler( yonetmen_id: 3, yonetmen_ad: "Christopher Nolan")  
  
val f1 = Filmler( film_id: 1, film_ad: "Django", film_yil: 2013,k1,y2)  
val f2 = Filmler( film_id: 2, film_ad: "Inception", film_yil: 2013,k3,y3)
```

```
println("Film id : ${f1.film_id}")  
println("Film ad : ${f1.film_ad}")  
println("Film yıl : ${f1.film_yil}")  
println("Film kategori : ${f1.kategori.kategori_ad}")  
println("Film yönetmen : ${f1.yonetmen.yonetmen_ad}")
```

```
Film id : 1  
Film ad : Django  
Film yıl : 2013  
Film kategori : Dram  
Film yönetmen : Quentin Tarantino
```

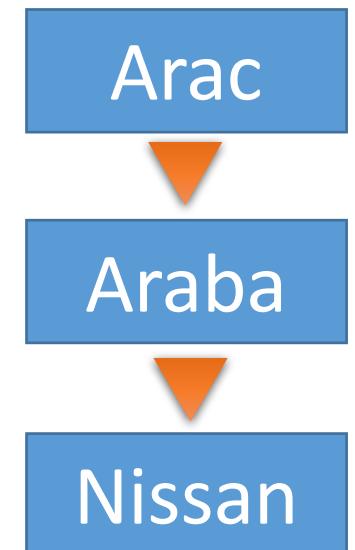
Veritabanı modeli olduğu  
için data class olması iyi  
olacaktır.

# Kalıtım ( Inheritance )

# OOP Kuralı – Kalıtım ( Inheritance )

- Mevcut bir sınıfından başka bir sınıf türetmek için kullanılır.
- Kodun tekrar kullanabilirliğini artırır.
- Sadece **class** için geçerlidir.
- Super class **open** kelimesi ile tanımlanır.
- : işaretü ile tanımlanır.
- Bir sınıfın tek kalıtımı olabilir.
- Bir sınıfa birden fazla sınıf kalıtım yolu ile bağlanamaz.
- Üst sınıfa **superclass** denir.
- Alt sınıfa **subclass** denir.

```
open class Arac {  
}  
  
open class Araba : Arac() {  
}  
  
class Nissan:Araba() {  
}
```



## Örnek :

```
open class Arac (var renk:String, var vites:String) {
```

Kalıtım yoluyla oluşturulan sınıfın constructor'ı üst sınıfın özelliklerini almalıdır.

```
open class Araba(renk:String, vites:String, var kasaTipi:String) : Arac(renk, vites) {
```

```
class Nissan(renk:String, vites:String, kasaTipi:String, var model:String):Araba(renk, vites, kasaTipi) {
```

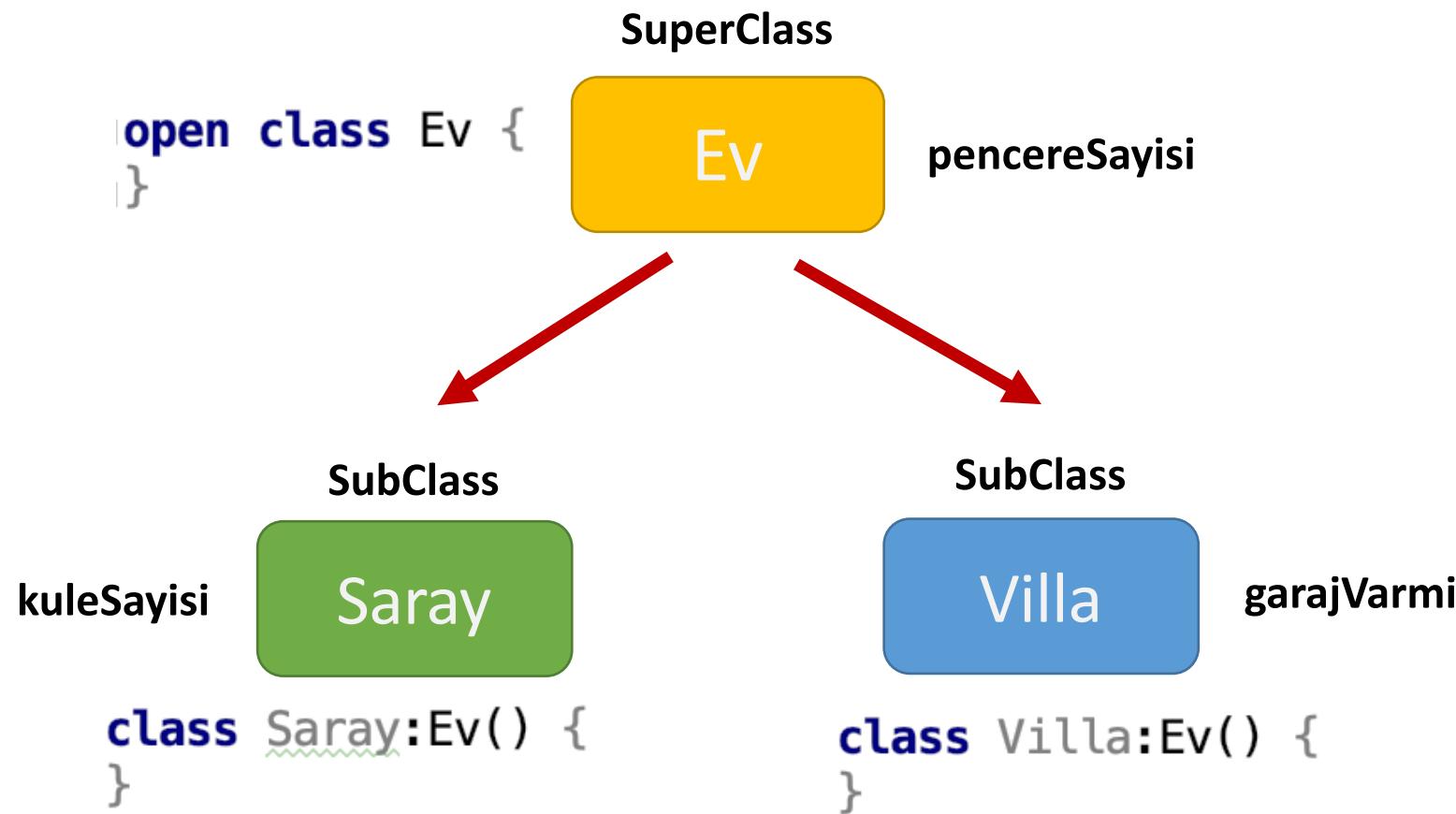
Araç

Araba

Nissan

```
var araba = Araba( renk: "Kırmızı", vites: "Otomatik", kasaTipi: "Sedan")  
println(araba.renk)  
println(araba.vites)  
println(araba.kasaTipi)  
println(araba.model)
```

# Kalıtım Hiyerarşisi Örnek



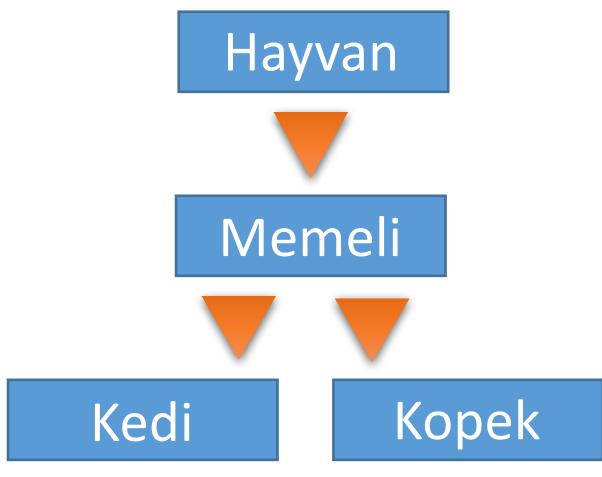
# Kalıtım Hiyerarşisi Örnek

```
open class Ev(var pencereSayisi:Int) {  
}  
  
class Villa(pencereSayisi:Int, var garajVarmi:Boolean):Ev(pencereSayisi) { }  
class Saray(pencereSayisi:Int, var kuleSayisi:Int):Ev(pencereSayisi) { }  
  
val topkapiSarayı = Saray( pencereSayisi: 10, kuleSayisi: 5)  
val bogazVilla = Villa( pencereSayisi: 4, garajVarmi: true)  
  
println(topkapiSarayı.pencereSayisi)  
println(topkapiSarayı.kuleSayisi)  
  
println(topkapiSarayı.garajVarmi)  
// Saray ile Villa arasında kalıtım olmadığı için erişim olmaz  
  
println(bogazVilla.pencereSayisi)  
println(bogazVilla.garajVarmi)  
  
println(bogazVilla.kuleSayisi)  
// Saray ile Villa arasında kalıtım olmadığı için erişim olmaz
```

# Override

# Metodları Ezme : Overriding

- Kalıtım ilişkisinde üst sınıfın metodlarının alt sınıf tarafından tekrar kullanılmasıdır.
  - override edilecek metod'a **open** kelimesi eklenmelidir.



```
open class Hayvan {  
    open fun sesCikar(){  
        println("Sesim Yok")  
    }  
}
```

```
open class Memeli:Hayvan() {  
}
```

```
class Kedi:Memeli() {  
    override fun sesCikar() {  
        println("Miyav Miyav")  
    }  
}
```

```
class Kopek:Memeli() {  
    override fun sesCikar() {  
        println("Hav Hav")  
    }  
}
```

```
val hayvan = Hayvan()  
println(hayvan.sesCikar())  
//Üst sınıf doğal olarak kendi metodunu çalıştırır.
```

```
val memeli = Memeli()  
println(memeli.sesCikar())  
//Alt sınıf üst sınıfın metodunu override etmesede  
//üst sınıfın metodunu çalıştırır.
```

```
val kedi = Kedi()  
println(kedi.sesCikar())  
//Alt sınıf üst sınıfın metodunu override ettiyse  
//kendi sınıfının metodunu çalıştırır.
```

```
val kopek = Kopek()  
println(kopek.sesCikar())  
//Alt sınıf üst sınıfın metodunu override ettiyse  
//kendi sınıfının metodunu çalıştırır.
```

**Sesim Yok**

**Sesim Yok**

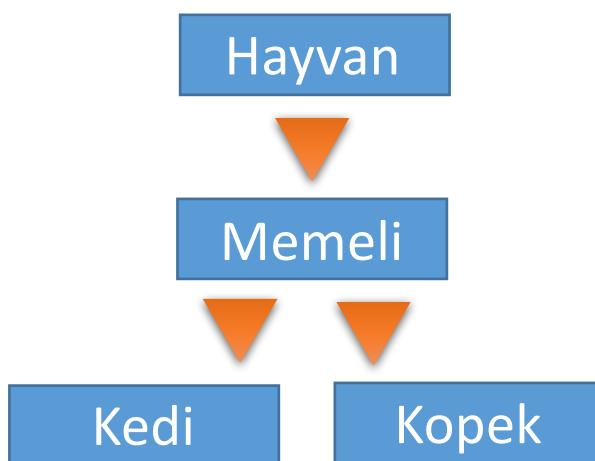
**Miyav Miyav**

**Hav Hav**

# PolyMorphism

# PolyMorphism

- PolyMorphism olması için iki sınıf arasında kalıtım ilişkisi olmalıdır.
  - Daha kapsayıcı bir kullanım sağlamak için kullanılır.
  - Özellikle metodların parametrelerinde PolyMorphism kullanılarak daha kapsayıcı veriler alınabilir.
  - Superclass gibi görünüp subclass gibi davranır.

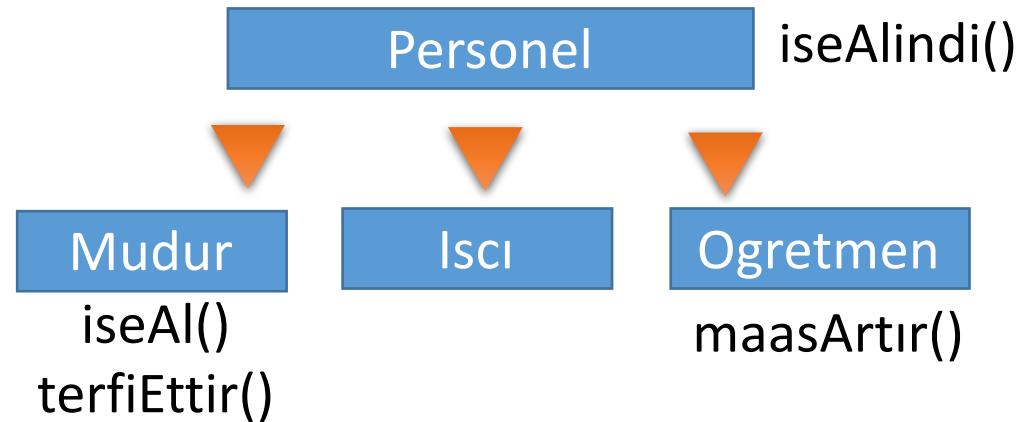


**val** hayvan:Hayvan = Kopek()

```
println(hayvan.sesCikar())
```

Hav Hav

# PolyMorphism Örnek



# PolyMorphism Örnek

```
open class Personel {  
    fun iseAlindi(){  
        println("Personel Mutlu")  
    }  
}  
  
class Mudur:Personel() {  
    fun iseAl(p:Personel){  
        p.iseAlindi()  
    }  
}  
  
class Isci:Personel() {  
}  
  
class Ogretmen:Personel() {  
}
```

**KULLANIM**

```
val ogretmen:Personel = Ogretmen()  
//Polymorfism  
  
valisci:Personel = Isci()  
//Polymorfism  
  
val mudur = Mudur()  
mudur.iseAl(ogretmen) Personel Mutlu  
mudur.iseAl(isci) Personel Mutlu
```

Özellikle metodların parametrelerinde  
PolyMorphism kullanılarak daha  
kapsayıcı veriler alınabilir.

# PolyMorphism Örnek

```
open class Personel {
    fun iseAlindi(){
        println("Personel Mutlu")
    }
}

class Mudur:Personel() {
    fun iseAl(p:Personel){
        p.iseAlindi()
    }

    fun terffiEttir(p:Personel){
        (p as Ogretmen).maasArttir()
        //Downcasting
        //maasArttir() metoduna erişmek için
        //Ogretmen sınıfından nesneye ihtiyaç vardır.
    }
}

class Isci:Personel() {
    class Ogretmen:Personel() {
        fun maasArttir(){
            println("Maas arttı.Öğretmen Mutlu :)")
        }
    }
}
```

**KULLANIM**

```
val ogretmen:Personel = Ogretmen()
//Polymorfism

valisci:Personel = Isci()
//Polymorfism

val mudur = Mudur()

mudur.terffiEttir(ogretmen)
mudur.terffiEttir(isci)
```

Casting hatası oluşacaktır çünküisci  
Ogretmen sınıfına dönüşemez.

# Nesnelerin Tip Dönüşümü

# Tip Kontrolü - **is**

- Tip kontrolü **is** ile yapılabilir. `is true` `false` şeklinde bilgi verir.

```
val topkapiSarayı = Saray( pencereSayisi: 100, kuleSayisi: 10)  
if (topkapiSarayı is Saray){  
    println("Saraydır")  
}else{  
    println("Saray Değildir")  
}
```

ÇIKTI :

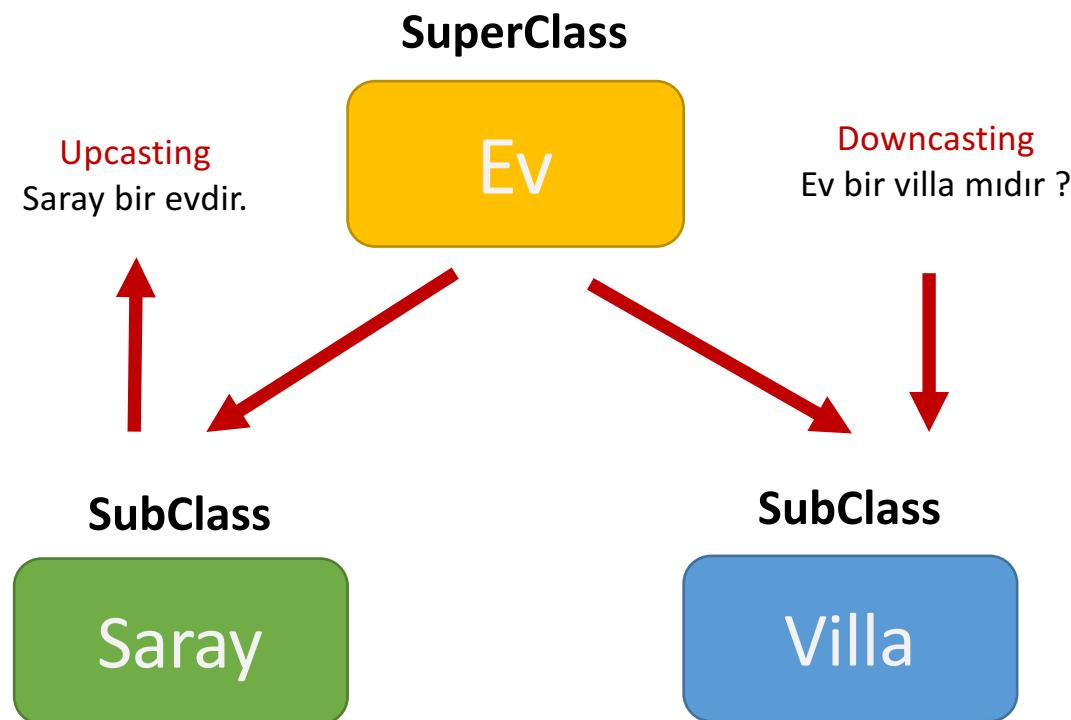
Bu saraydır!

# Downcasting – UpCasting



# Upcasting

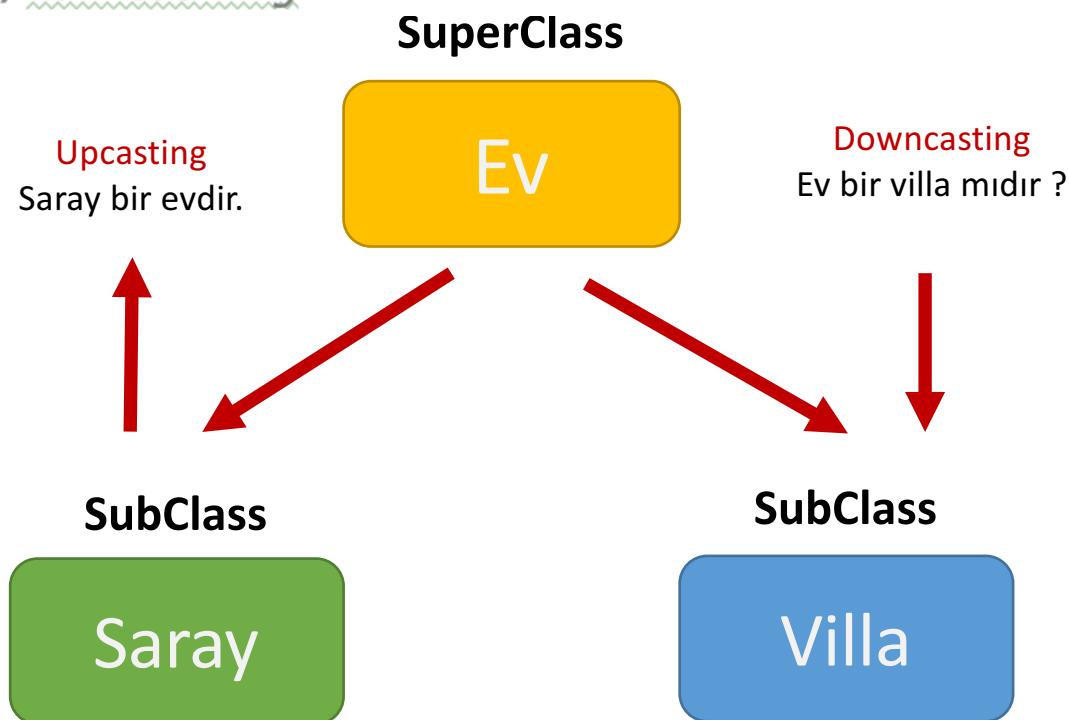
```
val ev1:Ev = Saray( pencereSayisi: 5, kuleSayisi: 3)  
//Upcasting
```



# Downcasting – **as**

```
val ev2 = Ev( pencereSayisi: 2)
```

```
val saray1 = ev2 as Saray  
//Downcasting
```

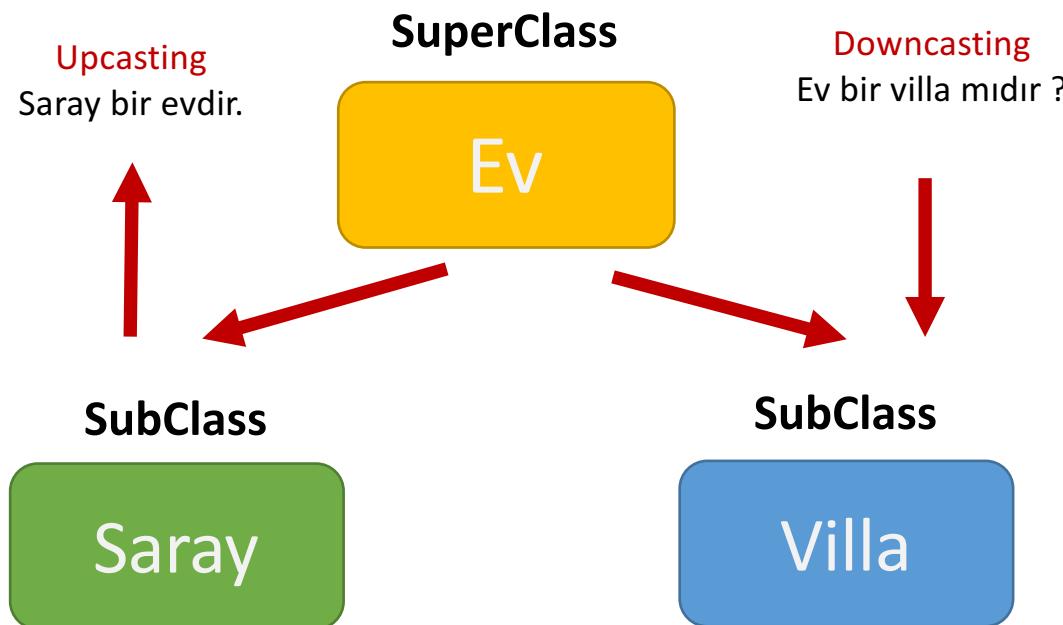


# Downcasting – as? – (nullable) cast

```
val ev2 = Ev( pencereSayisi: 2)
```

```
val saray2 = ev2 as? Saray  
//Downcasting  
//? işaretini ile güvenli castin olmuş olur.  
//Eğer dönüşümde sorun olursa  
//saray2 değişkeni içine null aktarılır.
```

? işaretini ile güvenli casting olmuş olur.  
Dönüşüm olurken hata oluşması engellenir.  
Eğer dönüşümde sorun olursa saray2 değişkeni içine null aktarılır.



# PolyMorphism Casting Hatası Önleme

## Tip Kontrolü **is**

```
class Mudur:Personel() {  
    fun iseAl(p:Personel){  
        p.iseAlindi()  
    }  
  
    fun terffiEttir(p:Personel){  
        if (p is Ogretmen){  
            p.maasArttir()  
            //is kontrolü yapıldığı için otomatik olarak casting olmuştur.  
        }  
  
        if (p is Isci){  
            println("İşçiler terfi alamaz.")  
        }  
    }  
}
```

### KULLANIM

```
val ogretmen:Personel = Ogretmen()  
//Polymorfism  
  
valisci:Personel = Isci()  
//Polymorfism  
  
valmudur = Mudur()  
  
mudur.terffiEttir(ogretmen)  
mudur.terffiEttir(isci)
```

Maaş Arttı. Öğretmen Mutlu :)  
İşçiler terfi alamaz

# Interface

# Interface

- Class yapısında kullanılabilir.
- Bir birden fazla interface alabilir.
- : ile eklenirler.
- Kalıtım gibi sınıfın sonuna () konulmaz.
- Hazır taslaklar gibi düşünebilirsiniz.
- Interface'ler sınıflara özellik katar.

```
interface Interface1 {  
    val degsiken: Int  
    fun metod1()  
    fun metod2(): String  
}
```

```
class ClassA:Interface1 {  
    override val degsiken: Int = 10  
    override fun metod1() {  
        println("Interface Merhaba")  
    }  
    override fun metod2(): String {  
        return "Interface çalışması"  
    }  
}
```

# Interface Örnek

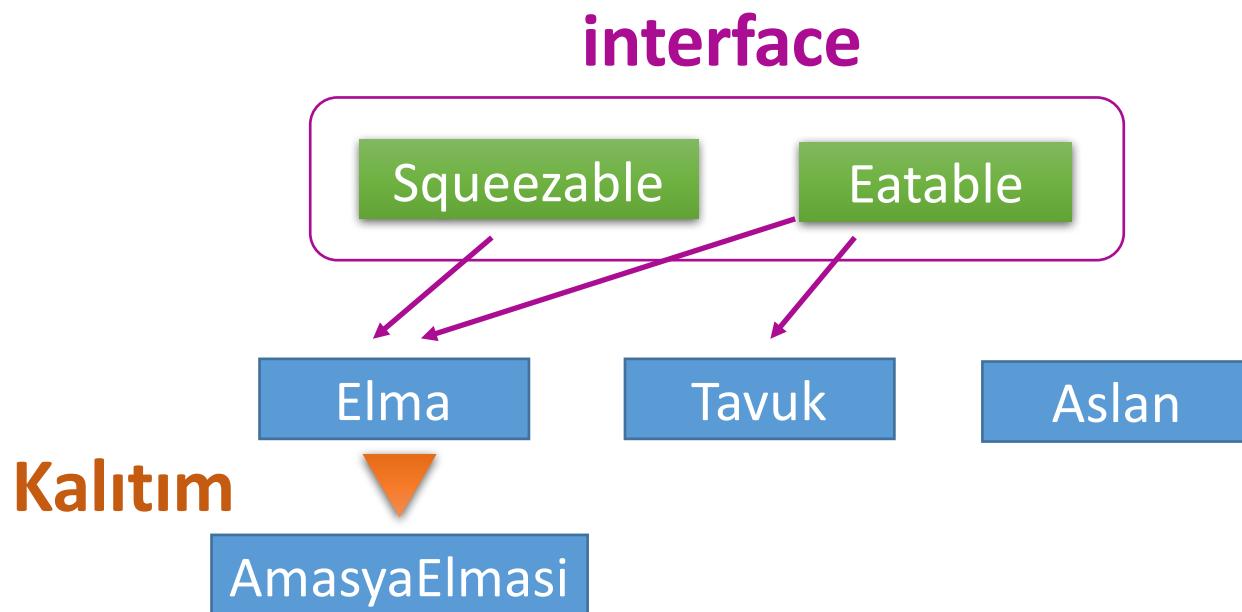
```
interface Interface1 {  
    val degsiken: Int  
    fun metod1()  
    fun metod2(): String  
}
```

```
class ClassA:Interface1 {  
    override val degsiken: Int = 10  
    override fun metod1() {  
        println("Interface Merhaba")  
    }  
    override fun metod2(): String {  
        return "Interface çalışması"  
    }  
}
```

```
val a = ClassA()  
println(a.degsiken)  
a.metod1()  
println(a.metod2())
```

```
10  
Interface Merhaba  
Interface çalışması
```

# Interface Örnek



# Interface Örnek

```
interface Squeezable {  
    fun howTosqueeze()  
}
```

```
interface Eatable {  
    fun howToEat()  
}
```

```
open class Elma:Eatable,Squeezable {  
    override fun howToEat() {  
        println("Dilimle ve ye")  
    }  
  
    override fun howTosqueeze() {  
        println("Blendır ile sık")  
    }  
}
```

```
class Tavuk : Eatable {  
    override fun howToEat() {  
        println("Fırında kızart")  
    }  
}
```

```
class Aslan {  
}
```

```
class AmasyaElmasi:Elma() {  
    override fun howToEat() {  
        println("Yıka ve ye")  
        //Bu metod interface ile değil  
        //kalıtım ile geldi.  
    }  
}
```

```
val aslan = Aslan()  
val amasyaElmasi:Elma = AmasyaElmasi()  
val elma = Elma()  
val tavuk:Eatable = Tavuk()  
  
val nesneler = arrayOf(aslan,amasyaElmasi,elma,tavuk)  
  
for (nesne in nesneler){  
    if (nesne is Eatable){  
        nesne.howToEat()  
    }  
  
    if (nesne is Squeezable){  
        nesne.howTosqueeze()  
    }  
}
```

Yıka ve ye  
Blendır ile sık  
Dilimle ve ye  
Blendır ile sık  
Fırında kızart

Teşekkürler...



kasim-adalan



kasimadalan@gmail.com



kasimadalan