

## **BÖLÜM 1**

### **JAVA NEDİR ?**

Java™ platformu bilgisayar ağının varlığı da göz önüne alınarak uygulamaların/programların farklı işletim sistemleri üzerinde çalıştırılabilmesi düşüncesiyle geliştirilmiş yeni bir teknolojidir. Java teknolojisi kullanılarak aynı uygulama farklı ortamlarda çalıştırılabilir. Örneğin kişisel bilgisayarlarda, *Macintosh* bilgisayarlarda, üstelik cep telefonlarında...

#### **Java'nın Başarılı Olmasındaki Anahtar Sözcükler**

- ❶. Nitelikli bir programlama dili olması
  - C/C++ da olduğu gibi bellek problemlerinin olmaması.
  - Nesneye yönelik (*Object Oriented*) olması.
  - C/C++/VB dillerinin aksine doğal dinamik olması.
  - Güvenli olması.
  - İnternet uyg. için elverişli olması. (*Applet*, *JSP*, *Servlet*, *EJB*, *Corba*, *RMI*).
- ❷. Platform bağımsız olması: **Bir kere yaz her yerde çalıştır!**

#### **Java ile Neler Yapılabilir?**

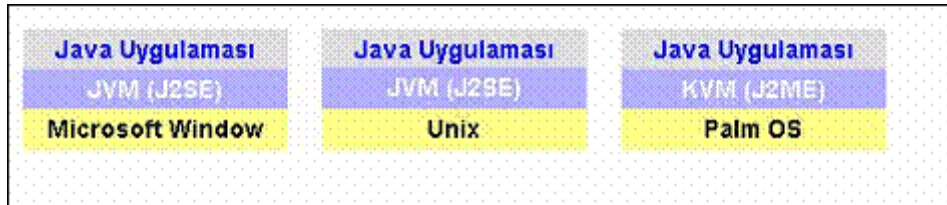
Java diliyle projeler diğer programlama dillerine göre daha kolay, sağlıklı ve esnek şekilde yapılması mümkün olur. Kısaca göz atılırsa Java diliyle,

- GUI (Grafiksel Kullanıcı Arayüzü) uygulamaları, *Applet*'ler
- Veri tabanına erişimle ilgili uygulamalar
- *Servlet*, *Jsp* (Web tabanlı uygulamalar).
- Dağıtık bileşenler (*Distributed components*) (örneğin *EJB*, *RMI*, *CORBA*).
- Cep telefonları, *Smart* kartlar için uygulamalar.
- Ve daha niceleri...

için uygulamalar yazmamız mümkündür.

#### **Java Nasıl Çalışır?**

Java uygulamaları JVM tarafından yorumlanır; JVM, işletim sisteminin üstünde bulunur. Bu nedenle, Java uygulamaları farklı işletim sistemlerinde herhangi bir değişiklik yapılmadan çalışır. Böylece Java programlama dilinin felsefesi olan “Bir kere yaz her yerde çalıştır” sözü gerçekleştirilmiş olunur.



Şekil-1.1. İşletim sistemleri ve JVM'in konumu

Şekil-1.2.'de Java kaynak kodunun nasıl çalıştırıldığı aşamalarıyla gösterilmiştir. *Byte* (sekizli) koduna çevrilen kaynak kod, JVM tarafından yorumlanır ve uygulama çalıştırılmış olur. Kısa bir Java uygulaması üzerinde olayları daha ayrıntılı bir şekilde incelenirse...

## Örnek: *Selam.java*

```
public class Selam {  
    public static void main(String args[]) {  
        System.out.println("Selamlar !");  
    }  
}
```

### Derleme anı (Compile time)

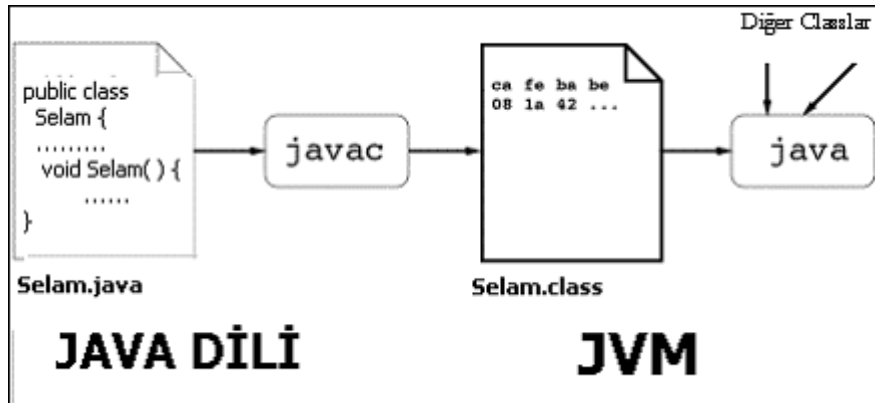


### Çalıştırma anı (Run time)



Şekil-1.2. JAVA kodunun çalıştırılma aşamaları

Yukarıda yazılan uygulamanın hangi aşamalardan geçtiği şekil üzerinde ilerleyen paragraflarda gösterilmiştir:



Şekil-1.3. Selam.java'nın geçtiği aşamalar

Yazılan kaynak kodları ilk önce **javac** komutuyla derlenir; ardından **java** komutuyla çalıştırılır. Fiziksel dosyanın içerisindeki her sınıf (*class*) için fiziksel olarak bir **.class** dosyası oluşturulur.

## Çöp Toplayıcı (*Garbage Collector*)

Çöp toplayıcı devamlı olarak takip halindedir; Java uygulamasının çalışma süresince ortaya çıkan ve sonradan kullanılmayan gereksiz nesneleri bulur ve onları temizler. Böylece bellek yönetim (*memory management*) yükü tasarımcıdan JVM'e geçmiş olur. Diğer dillerde, örneğin C++ programlama dilinde, oluşturulan nesnelerin yok edilme sorumluluğu tasarımcıya aittir.

Çöp toplayıcının ne zaman ortaya çıkıp temizleme yapacağı belirli değildir; eğer bellekte JVM için ayrılan kısım dolmaya başlamışsa çöp toplayıcı devreye girerek kullanılmayan nesneleri bellekten siler. Çöp toplayıcısı JVM'in gerçekleşmesine göre farklılık gösterebilir; nedeni, her JVM üreticisinin farklı algoritmalar kullanmasından ileri gelmektedir.

### Java'da Açıklama Satırı (*Comment Line*)

Java kaynak kodunun içerisine kod değeri olmayan açıklama yazılabilmesi için belirli bir yol izlenmesi gerekir.

- `/* yorum */`
- `// yorum;`
- 

### Herşey Nesne

Elimizde bir kumanda cihazının bulunması, maket uçağımızda olması anlamına gelmez. Her durumda bir referansı tek başına da tanımlanabilir. İşte kanıtı,

#### Gösterim-1.1:

```
String kumanda; // kumanda referansı şu an için String nesnesine bağlı değil.
```

Bir referansa mesaj göndermek istiyorsak onu bir nesneye bağlamamız gerekir.

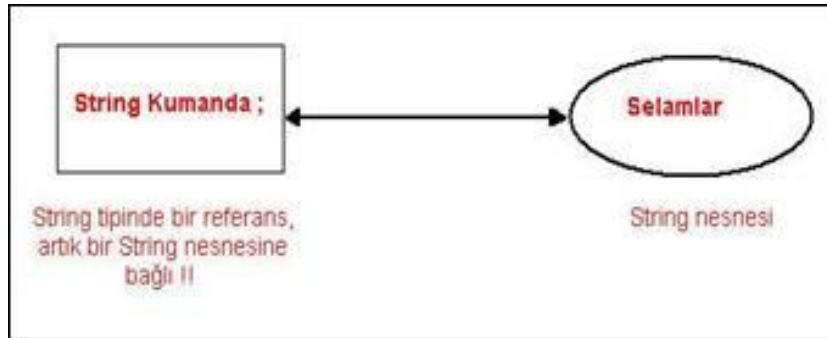
#### Gösterim-1.2:

```
String kumanda= new String("maket uca");
```

#### Gösterim-1.3:

```
String kumanda="maket uca";
```

Bu gösterimlerin şekil üzerindeki ifadesi aşağıdaki gibi olur:



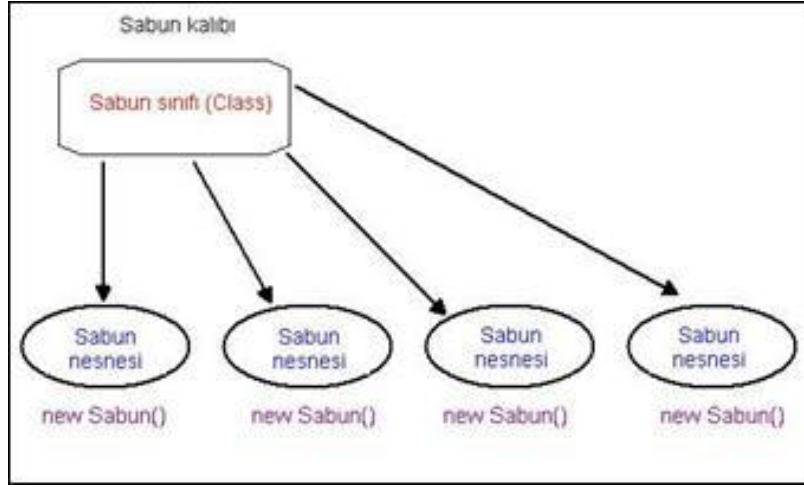
Şekil-1.5. Referans nesne bağlantısı

Verilen gösterimlerde *String* tipindeki referanslara *String* tipindeki nesneler bağlanmıştır. Gösterim-1.2 ile 1.3 arasında herhangi bir fark yoktur. Java'da *String* nesnesinin özel bir yeri vardır.

### Sınıf (*Class*) Nedir? Nesne (*Object*) Nedir?

Sınıf ve nesne kavramı bir benzetme ile açıklanırsa: Sabun fabrikasında yeni bir sabun tasarımı üzerinde çalıştığımızı hayal edelim; ortaya yeni bir kalıp çıkarttık... Artık son aşama olan üretime geçmek istiyoruz. Bu kalıp içerisinde sabun nesnelerinin hangi boyutlarda, hangi renkte olacağı, nasıl kokacağı vs. gibi bilgilerin hepsi bizim tarafımızdan belirlenmiş durumda olacaktır. Üretim aşamasına geçildiğinde hep aynı sabun kalıbını kullanılarak yeni sabun nesneleri üretmemiz mümkün olacaktır. Buradaki önemli nokta,

sabun kalıbı tasarımı birkez yapılmış olmasıdır; ancak, bu kalıp ile  $N$  tane sabun nesnesi üretilebilmektedir. Buradan yola çıkılarak sabun kalıbını sınıfa, sabunlarsa nesnelere benzetilebilir.



Şekil-1.6. Sınıf ve nesne'nin gösterilmesi

### Depolanan (*Storage*) Veriler Nerede Durmaktadır?

Depo toplam 4 alandan oluşur, bu 4 alan aşağıdaki gibi açıklanabilir:

- **Yığın (*Stack*):** Bulunduğu nokta bellek içerisinde; yani RAM üzerinde tutulur. Yığın üzerinde referansların kendileri bulunur.
- **Heap:** Genel amaçlı bir bellek havuzudur. *Heap* alanında nesnelerin kendisi durur.
- **Statik Alan:** Bu alan da RAM üzerinde bulunur. Statik alanda yer alan veriler, programın çalışması süresince orada yaşarlar. Tüm nesneler bu statik verileri görebilirler, burayı ortak bir alan gibi düşünebiliriz. Veriyi statik yapmak için `static` kelimesini global değişkenin (*referans*) önüne getirmemiz yeterli olur. Nesnelerin kendileri bu alanda yer almazlar.
- **Sabit Disk:** Bazı durumlarda uygulamaların içerisinde oluşturduğumuz nesnelerin, uygulama sonlandıktan sonra bile varlıklarını sürdürmelerini isteriz.

**Akışkan Nesneler (*Streamed Objects*):** Bu nesneler genel olarak ağ (*network*) üzerindeki başka bir sisteme gönderilmek üzere *byte* (sekizli) ırmaklarına dönüştürülürler.

**Kalıcı Nesneler (*Persistent Objects*):** Bu nesneler kendi durumlarını saklarlar; saklamaktan kasıt edilen ise özelliklerinin (*attribute*) değerlerinin korunmasıdır.

## Temel Tipler

**Tablo-1.2. JAVA programlama dilinin temel tipleri**

| Temel tip      | Boyut   | Minimum   | Maximum            | Sarmalayıcı sınıf |
|----------------|---------|-----------|--------------------|-------------------|
| <b>boolean</b> | —       | —         | —                  | Boolean           |
| <b>char</b>    | 16- bit | Unicode 0 | Unicode $2^{16}-1$ | Character         |
| <b>byte</b>    | 8- bit  | -128      | +127               | Byte              |
| <b>short</b>   | 16- bit | $-2^{15}$ | $+2^{15}-1$        | Short             |
| <b>int</b>     | 32- bit | $-2^{31}$ | $+2^{31}-1$        | Integer           |
| <b>long</b>    | 64- bit | $-2^{63}$ | $+2^{63}-1$        | Long              |
| <b>float</b>   | 32- bit | IEEE754   | IEEE754            | Float             |
| <b>double</b>  | 64- bit | IEEE754   | IEEE754            | Double            |
| <b>void</b>    | —       | —         | —                  | Void              |

Bu temel tiplerin birer adet sarmalayıcı (*wrapper*) sınıfı bulunur. Örneğin, temel `int` tipinin sarmalayıcısı *Integer* sınıfıdır; benzer şekilde `double` tipinin sarmalayıcısı *Double* sınıfıdır. Temel tipler ile sarmalayıcıları sınıfları arasındaki farklar ilerleyen bölümlerde ele alınacaktır.

### Gösterim-1.4:

```
int i = 5; // temel tip
```

### Gösterim-1.5:

```
Integer in = new Integer(5); // sarmalayıcı sınıf
```

### **Geçerlilik Alanı (Scope)**

Her programlama dilinde değişkenlerin geçerlilik alanı kavramı bulunur. Java ile C ve C++ dillerindeki değişkenlerin geçerlilik alanlarının nasıl olduğunu görüp bir karşılaştırma yapalım:

### Gösterim-1.6:

```
{
    int a = 12; /* sadece a mevcut */
    {
        int b = 96; /* a ve b mevcut */
    }
    /* sadece a mevcut */
    /* b geçerlilik alanının dışına çıktı */
}
```

Temel `int` tipinde olan **a** değişkeninin geçerlilik alanı kendisinden daha iç tarafta olan alanlar da bile geçerlidir; ancak, aynı tipte olan **b** değişkeni incelenirse, kendisinden daha dış tarafta olan alanlarda geçerli olmadığı görülür.. Şimdi aşağıdaki gösterimi inceleyelim, bu ifade C ve C++ için doğru ama Java programlama dili için yanlış olur.

### Gösterim-1.7:

```
{ // dış alan
  int a = 12;
  { // iç alan
    int a = 96; /* java için yanlış, C ve C++ doğru*/
  } // iç alanın sonu
} //dış alanın sonu
```

### **Nesnelerin Geçerlilik Alanları**

Java programlama dilinde nesnelerin ömürleri, temel tiplere göre daha farklıdır.

### Gösterim-1.8:

```
if (true){
  String s = new String("Selamlar");
} /* geçerlilik alanının sonu*/
```

Yukarıdaki gösterimde `if` koşuluna kesinlikle girilecektir.; girildiği anda *String* nesnesi *heap* alanında oluşturulacaktır. Bu yeni oluşturulan *String* nesnesi, *String* tipindeki `s` referansı (değişken) ile denetlenmektedir. Geçerlilik alanı sona erdiğinden `s` referansı artık kullanılamayacak hale gelecektir; Çöp “toplayıcı” devreye girdiği an *heap* alanındaki bu **erişilemez** ve **çöp haline** gelmiş olan *String* nesnesini bellekten silecektir.

### **Yeni Sınıf Oluşturma**

### Gösterim-1.9:

```
public class YeniBirSinif {
  // gerekli tanımlar...
}
```

### **Alanlar ve Yordamlar**

Bir sınıf (*class*) tanımladığı zaman bu sınıfın iki şey tanımlanabilir:

- ❶ Global **Alanlar** yani global **değişkenler**: temel (*primitive*) bir tip veya bir başka sınıf tipinde olabilirler.

### Gösterim-1.10:

```
public class YeniBirSinif {
  public int i;
  public float f;
  public boolean b;
}
```

Global değişkenlerin başlangıç değeri yoksa Tablo 1.3’te verilen default değerleri alır. Global değişkenlere başlangıç değeri verilmek isteniyorsa,

### Gösterim-1.11:

```
public class YeniBirSinif {  
    public int i = 5;  
    public float f = 3.23;  
    public boolean b = true;}  
}
```

**Tablo-1.3. Java temel tiplerin başlangıç değerleri**

| Temel Tip      | Varsayılan (Default) Değer |
|----------------|----------------------------|
| <b>Boolean</b> | false                      |
| <b>Char</b>    | '\u0000' (null)            |
| <b>Byte</b>    | (byte)0                    |
| <b>Short</b>   | (short)0                   |
| <b>Int</b>     | 0                          |
| <b>Long</b>    | 0L                         |
| <b>Double</b>  | 0.0d                       |
| <b>Float</b>   | 0.0f                       |

(Not: Sınıf tipindeki referanslara o tipteki nesne bağlanmamış ise değeri **null**'dir )  
*YeniBirSinif* sınıfına ait bir nesne oluşturulsun:

### Gösterim-1.12:

```
YeniBirSinif ybs = new YeniBirSinif();
```

ybs ismini verdiğimiz referansımız, *heap* alanındaki *YeniBirSinif* nesnesine bağlı bulunmaktadır. Eğer biz *heap* alanındaki bu *YeniBirSinif* nesnesiyle temas kurulması istenirse ybs referansı kullanılması gerekir. Nesne alanlarına ulaşılması için “.” (nokta) kullanılır.

### Gösterim-1.13:

```
ybs.i;  
ybs.f;  
ybs.b;
```

Eğer nesnenin alanlarındaki değerler değiştirilmek isteniyorsa,

### Gösterim-1.14:

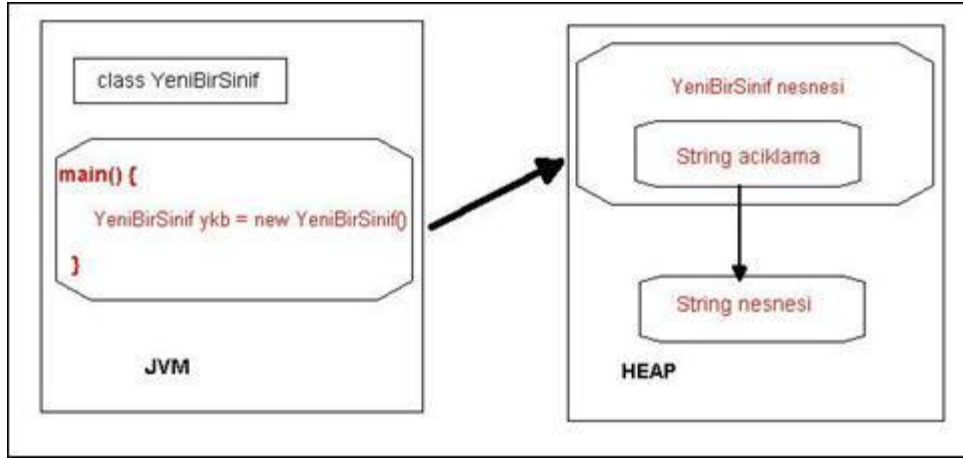
```
ybs.i = 5;  
ybs.f = 5.3f;  
ybs.b = false;
```

**Örnek:** *YeniBirSinif.java*

```
class YeniBirSinif {
    public int i;
    public float f;
    public boolean b;
    public String aciklama = new String("nesnemizin
    aciklamasi"); }

```

Örnekte verilen *YeniBirSinif* sınıfının içerisinde temel tipteki global değişkenlerin dışında, başka sınıf tipinde olan *aciklama* değişkeni yer almaktadır. Temel tiplerle sınıf tipindeki değişkenlerin arasındaki fark, aşağıda verilen şekil üzerinden incelenirse,



Şekil-1.7. Sınıf Tipindeki değişken

Şekildeki **main()** yordamı Java uygulamaları için başlama noktasıdır. *YeniBirSinif* sınıfına ait bir nesne oluştururken görüyoruz ki *aciklama* global değişkenine bağlı olan *String* nesnesi de *heap* bölgesinde yerini alıyor; yani, *heap* bölgesinde 2 adet nesne oluşmuş oluyor. Biri *YeniBirSinif* sınıfına, diğeri ise *String* sınıfına ait nesnelerdir.

❷ **Yordamlar:** Nesnelerin işe yarar hareketler yapmasına olanak veren kısımlar.

#### Gösterim-1.15:

```
DönüşTipi yordamınİsmi ( /* parametre listesi */ ) {

    /* Yordam gövdesi */

}

```

Yukarıdaki yordam iskeletinde tanımlanmış olan kısımlar birer birer açıklanırsa:

- **dönüşTipi** = Yordamlarınikişansıdır:
  - Değer döndürürler
    - Temel(primitive) bir tipde değer (int, double, short vb..)
    - Sınıf tipinde bir değer (String, Double, Short vb...)
  - Değer döndürmezler = **void**
- **yordamınİsmi** = Java'nın kendisine ait olan anahtar sözcükleri (**if**, **else**, **import**, **class**, **return** vs gibi) ve Türkçe karakter içermeyen herhangi bir isim kullanılabilir; ancak, yordamlar bir eylem içerdikleri için yordam isimlerinin de bir eylemi belirtmesi önerilir. Örneğin, *sayiSiralama()*, *enBuyuguBul()*, *sqlCalistir()* gibi; burada dikkat edilirse, yordam isimlerinin ilk



harfleri küçük sonra gelen ek sözcüğün ilk harfi ise büyüktür. Bu ismin anlamını daha kolay görmek içindir.

- **parametre listesi** = Yordam içerisinde işlemler yapabilmek için gerekli olan parametrelerdir. Bu parametreler temel tipte veya sınıf tipinde olabilirler.
- **yordam gövdesi** = Bu kısım kodu yazan kişinin hayal gücüne bağlı olarak değişmektedir.

Yordam (metot) örneği verebilirse,

#### **Gösterim-1.16:**

```
int uzunlukDondur(String kelime) {  
    return kelime.length();  
} // yordamın sonu
```

`uzunlukDondur()` yordamı *String* tipinde parametre alıyor ve *String* nesnesinin uzunluğunu geri döndürüyor. Yordamımızın geri döndürdüğü değer temel `int` tipindedir. Herhangi bir değer geri döndürülebilmesi için `return` anahtar kelimesi kullanılır.

#### **Gösterim-1.17:**

```
String elmaHesapla( int elmasayisi) {  
    return new String(" toplam elma sayisi = " + elmasayisi*18);  
}
```

Gösterim-1.17’de verilen `elmaHesapla()` yordamı tamsayı tipinde parametre alıyor; sonra yeni bir *String* nesnesi oluşturup bu nesnenin bağlı bir referansı geri döndürüyor. Buradaki ilginç olabilecek olan unsur `int` olan bir değişkeni 18 ile çarpılıp sonradan `+` operatörü ile *String* bir ifadenin sonuna eklenmiş olmasıdır. Java dilinde *String* bir ifadeden sonra gelen herhangi bir tipteki değişken otomatik olarak *String* nesnesine dönüştürülür.

#### **Gösterim-1.18:**

```
void hesapla(String kelime, int kdv ) {  
    int sondeger = 0;  
    int kelimeboyut = 0;  
    int toplamboyut; //Hatali !!!!  
    toplamboyut++;//Hatali !!!!  
    sondeger = kelimeboyut + kdv;  
}
```

**`hesapla()`** yordamı iki adet parametre almaktadır ve geriye hiçbir şey döndürmeyeceğini `void` anahtar kelimesi belirtmektedir. Bu örnekte dikkat edilmesi gereken ikinci unsur ise yordamların içerisinde tanımlanan yerel değişkenlerine başlangıç değerlerinin kesinlikle programcı tarafından belirtilmesi gerekliliğidir.

Sınıflara (*Class*) ait global değişkenlere başlangıç değerleri verilmediği zaman bu değişkenlere varsayılan değerleri verilir (bkz. Tablo-1.3); ancak, yordam içerisinde tanımlanan yerel değişkenler için aynı durum söz konusu değildir. Bu nedenle `toplamboyut` değişkeninin tanımlanma şekli yanlıştır.

### **Gösterim-1.19:**

```
void uniteKontrol(int deger) {  
    if (deger == 1 ) {          // eğer deger 1'e eşitse yordamı terk et  
        return;  
    }else {  
        // gerekli işlemler  
    }  
}
```

return anahtar kelimesi tek başına kullanıldığında ilgili yordamın içerisinden çıkarılır.

### **İlk Java Programımız**

**Örnek:** *Merhaba.java*

```
public class Merhaba {  
    public static void main(String args[]) {  
        System.out.println("Merhaba !");  
    }  
}
```

**public class merhaba:** Bu kısımda yeni bir sınıf oluşturuluyor.

**public static void main(String args[]):** Java'da bir sınıfın tek başına çalışması isteniyorsa (*stand alone*) bu yordam yazılmak zorundadır. Bu yordam sınıflar için başlangıç noktası gibi varsayılabilir.

**statik yordamlar:** Statik yordamlar nesneye bağımlı olmayan yordamlardır. Bunların kullanılması için sınıfa ait nesnenin oluşturulmuş olması gerekmez.

**Örnek:** *TestNormal.java*

```
public class TestNormal {  
    public void uyariYap() {  
        System.out.println("Dikkat Dikkat");  
    }  
    public static void main(String args[]) {  
        TestNormal tn = new TestNormal();    // dikkat  
        tn.uyariYap();  
    }  
}
```

*TestNormal.java* uygulamamızda *uyariYap()* yordamı statik değildir; bu nedenle bu yordamın çağrılabilmesi için *TestNormal* sınıfına ait bir nesne oluşturulması gerekir.

### **Örnek:** *TestStatik.java*

```
public class TestStatik {  
  
    public static void uyariYap() {  
        System.out.println("Dikkat Dikkat statik metod");  
    }  
  
    public static void main(String args[]) {  
        TestStatik.uyariYap();  
    }  
}
```

Bu örnekteki tek fark **uyariYap()** yordamının statik olarak değiştirilmesi değildir; çağrılma şekli de değiştirilmiştir. **uyariYap()** yordamı artık *TestStatik* nesnesine bağlı bir yordam değildir, yani **uyariYap()** yordamını çağırabilmemiz için *TestStatik* sınıfına ait bir nesne oluşturulması gerekmez. **main()** yordamında da işler aynıdır, fakat **main()** yordamının Java içerisinde çok farklı bir yeri vardır. **main()** yordamı tek başına çalışabilir uygulamalar için bir başlangıç noktasıdır.

**Diziler (Arrays):** **main()** yordamı parametre olarak *String* tipinde dizi alır. Bu *String* dizisinin içerisinde konsoldan Java uygulamasına gönderilen parametreler bulunur.

- **args[0]:** konsoldan girilen 1. parametre değerini taşır.
- **args[1]:** konsoldan girilen 2. parametre değerini taşır.
- **args[n-1]:** konsoldan girilen *n*. parametre değerini taşır.

Java'da diziler sıfırdan başlar; ilerleyen bölümlerde ayrıntılı olarak ele alınmaktadır.

**System.out.println ("Merhaba !"):** Bu satır, bilgilerin ekrana yazılmasını sağlar.

### **JAVA'nın Windows İşletim Sisteminde Kurulumu**

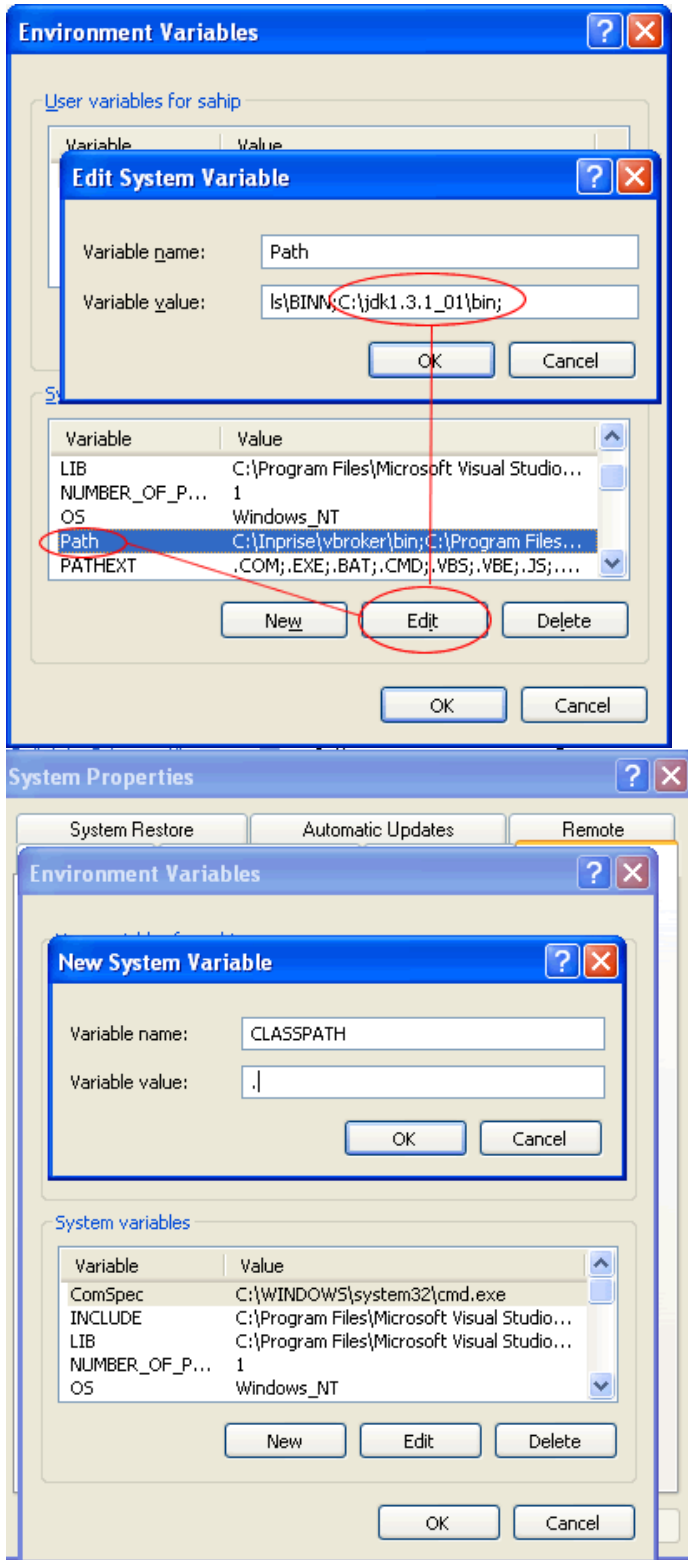
Java'nın Windows için hazır bir kurulum dosyası bulunur; bu dosya üzerine çift tıklanarak yapılması gereken işlerin büyük bir kısmı gerçekleştirilmeye başlar; ancak, hala yapılması gereken ufak tefek işler vardır:

Windows 95-98 için autoexec.bat dosyasında aşağıdaki değişiklikleri yapmanız yeterli olacaktır.

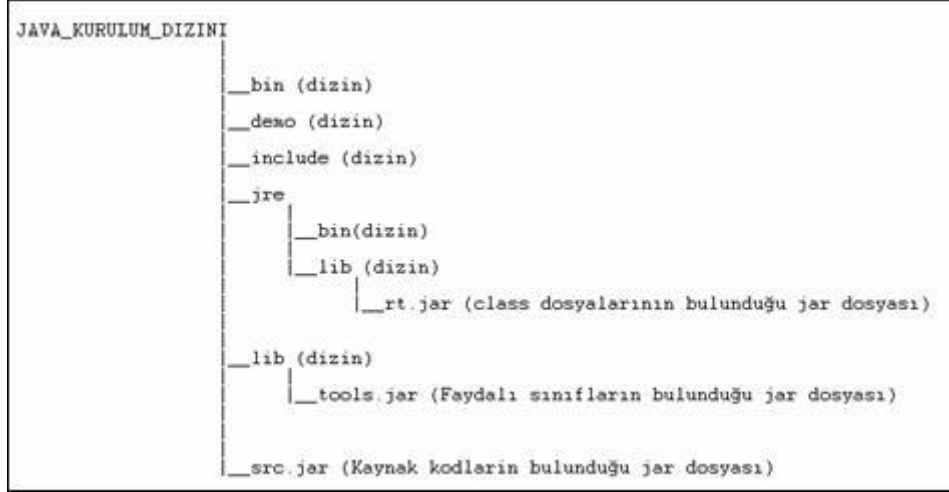
```
SET PATH=C:\WINDOWS;C:\WINDOWS\COMMAND;C:\JAVA\BIN;  
C:\ultraedit;.  
SET CLASSPATH=;C:\JAVA\lib\tools.jar;.
```

(**Not:** Tekrar oldu ama olsun çok önemli bir kısım: Dikkat edilirse CLASSPATH tanımlanırken en sona bir nokta koyuldu, bu nokta hayati bir önem taşır. Bu noktanın anlamı, bulunulan dizindeki **.class** dosyalarının görülebilmesini sağlamaktadır, böylece java komutunu çalıştırırken saçma hatalar almazız.)

Windows 2000 ve Xp için PATH ve CLASSPATH tanımlarını *Environment Variables* kısmına ulaşarak yapabilirsiniz. Bu bölüme ulaşmak için **Control Panel --> System --> Advanced --> Environment Variables** yolunu izlemeniz yeterli olacaktır.



Şekil-1.8’de Java dosyalarının yapısı görülmektedir.



Şekil-1.8. Java dosya düzenlenmesi

## JAVA Kodlarını Derleme ve Çalıştırma

Elimizdeki *Merhaba.java* dosyasını nasıl derleyeceğiz? Cevabı aşağıdaki gibidir:

```
$ javac Merhaba.java
```

Artık elimizde *Merhaba.class* dosyasının oluşmuş olması gerekir. Şimdi sıra geldi bu dosyayı çalıştırmaya,

```
$ java Merhaba
```

Ekrana çıkan yazın:

```
Merhaba!
```

## Nedir bu args[], Ne İşe Yarar ?

Tek başına çalışabilir Java uygulamasına, komut satırından (konsoldan) nasıl çalıştığını anladıktan sonra komut satırından Java uygulamamıza parametre göndermeyi öğrenebiliriz. Diğer bir uygulama örneği,

**Örnek:** *ParametreUygulaması.java*

```
public class ParametreUygulaması {
    public static void main(String[] args) {
        System.out.println("Merhaba Girdiğiniz Parametre = " +args[0]);
    }
}
```

Anımsanırsa Java'da dizilerin indis sıfırdan başlarlar. Şimdi *ParametreUygula.java* kaynak dosyası incelenirse,

```
$ javac ParametreUygulaması.java
```

Elimizde *ParametreUygulaması.class* dosyası oluştu; şimdi uygulamamızı çalıştırabiliriz, yalnız farklı bir şekilde,

```
$ java ParametreUygulaması test
```

Uygulamamızı çalıştırma tarzımız değişmedi, burada ki tek fark, en sona yazdığımız **test** kelimesini uygulamaya parametre olarak gönderilmesidir. İşte programın çıktısı:

```
Merhaba Girdiğiniz Parametre = test
```

Tek başına çalışabilir Java uygulamasına konsoldan birden fazla parametreler de gönderebiliriz.

### **Örnek:** *ParametreUygulaması2.java*

```
public class ParametreUygulaması2 {  
  
    public static void main(String[] args) {  
        System.out.println("Merhaba Girdiğiniz ilk Parametre  
        = " + args[0]);  
        System.out.println("Merhaba Girdiğiniz ikinci  
        Parametre = " + args[1]);  
  
    }  
}
```

Uygulamamızı öncelikle derleyelim:

```
$ javac ParametreUygulaması2.java
```

ve şimdi de çalıştıralım:

```
$ java ParametreUygulaması2 Test1 Test2
```

Ekranda görülen:

```
Merhaba Girdiğiniz ilk Parametre = Test1  
Merhaba Girdiğiniz ikinci Parametre = Test2
```

Peki bu uygulamaya dışarıdan hiç parametre girmeseydik ne olurdu ? Deneyelim:

```
$ java ParametreUygulaması
```

```
Exception in thread "main"  
java.lang.ArrayIndexOutOfBoundsException  
at  
ParametreUygulaması2.main(ParametreUygulaması2.java:5)
```

Sorun şu, Java uygulamamız bizden kesin olarak iki adet parametre girmemizi bekliyor. Beklediğini aşağıdaki kısımlarda görebiliyoruz:

### **Gösterim-1.20:**

```
System.out.println("Merhaba Girdiğiniz ilk Parametre =  
" + args[0]);  
System.out.println("Merhaba Girdiğiniz ikinci  
Parametre = " + args[1]);
```

### **Javadoc = Yorum ile Dokümantasyon Oluşturmak**

Uygulamalar yazılırken büyük bir oranla dokümantasyon işi ikinci plana itilir veya unutulur veya en kötüsü hiç yapılmaz. Dokümantasyon kodu yazan kişiler için çok ağır iştir (yazarımızda bu fikri katılıyor...). Java, dokümantasyon hazırlama işini daha kolay ve sevimli bir hale getirmiştir. Bu özelliğe "Javadoc" denir. *Javadoc* kodun içersine yazılmış olan yorum satırlarını alarak bunları HTML biçimine dönüştürmektedir. Fakat yorum satırını yazarken bazı kurallara uymamız gerekmektedir.

### **Sözdizimi Kuralları (Syntax)**

Bizim yazdığımız yorum satırlarının *Javadoc* tarafından dikkate alınması için:

/\*\* ile başlaması ve \*/ ile bitmesi gerekmektedir. *Javadoc* mekanizmasını kullanmanın iki etkili yolu vardır. Bunlardan birincisi **gömülü html** (*embedded html*) ikincisi ise *doc tags*.

*Doc tag* “@” ile başlarlar. Sınıfa, global değişkenlere ve yordamlara ait üç adet yorum dokümanı tipi vardır. Verilen örnek şablon bunu açıklamaktadır.

### **Gösterim-1.21**

```
/** A sınıf ait yorum */

public class DocTest {

/** i global değişkenin yorumu */

    public int i;

/** isYap() yordamunun yorumu */

    public void isYap() {}

}
```

Unutulmaması gereken başka bir nokta ise *Javadoc* varsayılan (*default*) *public* ve *protected* olan üyelerin dokümanını üretir. *private* ve *friendly* üyelerin dokümantasyonu yapmaz. Bunların sebeplerine ilerleyen konularda değineceğiz. Eğer *private* üyelere ait bilgilerinde yazılmasına istiyorsak o zaman komut satırından,

```
$ javadoc a -private .....
```

ekini vererek bunu başarabiliriz.

### **Gömülü HTML (*Embedded Html*)**

*JavaDoc* mekanizmasında kendinize ait HTML komutları kullanılması mümkündür:

### **Gösterim-1.22:**

```
/**
 * <pre>
 * System.out.println("Selamlar");
 * </pre>
 */
```

### **Gösterim-1.23:**

```
/**
 * Çok güzel <em>hatta</em> liste bile
 * yerleştirebilirsiniz:*<ol>
 * <li> madde bir
 * <li> madde iki
 * <li> Madde üç* </ol>
 */
```

<h1> <hr> gibi etiketlerini (*tag*) kullanmayın çünkü *Javadoc* bunları sizin yerinize zaten yerleştirmektedir. Gömülü HTML özelliği sınıf, global değişkenler ve yordamlar tarafından desteklenir.

## Ortak Kullanılan Yorum Ekleri

- **@see:** Başka bir sınıfın, global değişkenin veya yordamın dokümantasyonunu göstermek için kullanabilirsiniz.

(sınıf, global değişken, ve yordam) dokümantasyonlarında, **@see** etiketini (*tag*) kullanabilirsiniz. Bu etiketin anlamı, “başka bir dokümantasyona gönderme yapıyorum” demektir.

### Gösterim-1.24:

```
@see classismi  
@see classismi#yordam-ismi
```

*JavaDoc*, gönderme yaptığınız başka dokümantasyonların var olup olmadığını kontrol etmez. Bunun sorumluluğu size aittir.

## Sınıflara Ait *JavaDoc* Etiketleri

Sınıflara ait etiketler arayüzler içinde kullanılabilir.

- **@version:** Uyarlama numaraları dokümantasyonlamada yaşamsal bir rol oynar. Uyarlama etiketinin görünmesini istiyorsanız:

```
$ javadoc -version
```

Parametresi ile çalıştırmamız gerekmektedir. Versiyon etiketini kodumuza yerleştirmek için;

### Gösterim-1.25:

```
@version versiyon-bilgisi
```

- **@author:** Sınıfı yazan yazar hakkında bilgi verir. *author* etiketi *Javadoc* tarafından dikkate alınmasını istiyorsanız:

```
$ javadoc -author
```

Parametresini koymanız gerekir. Bir sınıfa ait birden fazla *author* etiketi yerleştirebilirsiniz. *author* etiketini kodumuza yerleştirmek için:

### Gösterim-1.26:

```
@author author-bilgisi
```

- **@since:** Bir sınıfın belli bir tarihten veya belli bir uyarlamadan itibaren var olduğunu belirtmek için kullanılır. *since* etiketini kodumuza yerleştirmek için:

### Gösterim-1.27:

```
@since 05.03.2001
```

## Global Değişkenlere Ait *JavaDoc* Etiketleri

Global değişkenlere ait dokümantasyonlarda sadece gömülü html veya **@see** etiketi kullanılabilir.

## Yordamlara Ait *JavaDoc* Etiketleri

Yordam dokümantasyonunda gömülü html veya **@see** etiketini kullanabilirsiniz. Ayrıca aşağıdaki etiketleri kullanabilirsiniz.

- **@param:** Yordamın aldığı parametrenin açıklamasını belirtir. *param* etiketini kodumuza yerleştirmek için:



### **Gösterim-1.28:**

```
@param parametre-ismi açıklaması
```

- **@return:** Yordamın döndürdüğü değerin açıklamasını belirtir. *return* etiketini kodumuza yerleştirmek için:

### **Gösterim-1.29:**

```
@return açıklama
```

- **@throws:** İstisnalar (*exception*) konusunu ilerleyen bölümlerde inceleyeceğiz. İstisnalar kısa olarak açıklarsak, yordamlarda hata oluşursa oluşturulan özel nesnelerdir. *throws* etiketini kodumuza yerleştirmek için:

### **Gösterim-1.30:**

```
@throws class-ismi açıklaması
```

- **@deprecated:** Önceden yazılmış fakat artık yerine yenisi yazılmış bir yordam *deprecated* olur (tedavülünden kaldırılır). *deprecated* yordamı kullandığınız zaman derleyici tarafından uyarı alırsınız. *deprecated* etiketini kodumuza yerleştirmek için:

## **Dokümantasyon Örneği**

### **Örnek:** *SelamDoc.java*

```
/** İlk Java Kodumuzun Dokümantasyonu
 * Ekranı Selamlar diyen bir uygulama
 * @author Altug B. Altintas (altug.altintas@koubm.org)
 * @version 1.0
 * @since 09.01.2002
 */

public class SelamDoc {

    /**sayıyı artırmak için değişkenler için bir örnek*/

    public int sayac = 0;

    /** sınıflarda & uygulamalarda giriş noktası olan yordam
     * @param args dışarıdan girilen parametreler dizisi
     * @return dönen değer yok
     * @exception Hic istisna fırlatılmıyor */

    public static void main(String[] args) {
        System.out.println("Selamlar !");
    }
}
```

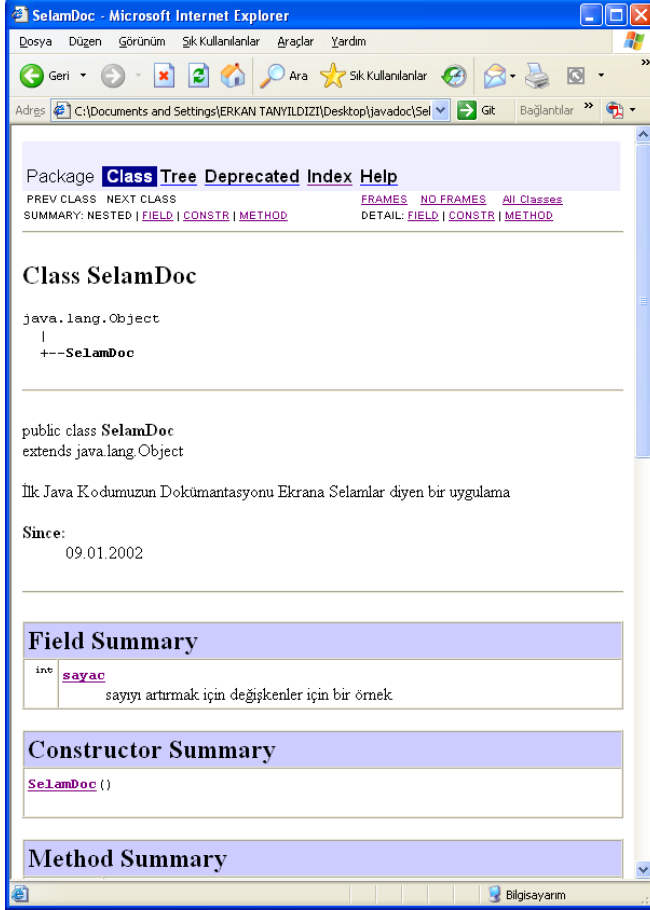
Oluşturulan bu kaynak koduna ait doküman çıkartmak için komut satırına aşağıdaki yazılı komutun yürütülmesi yeterlidir:

```
$ javadoc -d javadoc SelamDoc.java
```

Bu komut sayesinde HTML dokümanları bulunduğumuz dizinin altındaki *javadoc* dizinin içinde oluşturulmuş oldu. *Javadoc* 'un komutlarıyla daha fazla bilgiyi,

```
$ javadoc -help
```

yazarak ulaşabilirsiniz.



## Sınıf İsimleri ve Yordam İsimleri

Sınıf isimleri uzun olursa alt çizgi ile ayrılmalıdır. Sınıf isimleri fiil cümlesi içermemelidir.

### Gösterim-1.31:

```
public class En_Kisa_Yol_Bulma_Algoritmasi {  
  
}
```

Yordamlar fiil cümlesi içermelidirler. İlk kelime küçük harf ile başlamalıdır.

### Gösterim-1.32:

```
public void yolBul() {  
  
}
```

## BÖLÜM 2

## ***JAVA' DA PROGRAM DENETİMİ VE OPERATÖRLER***

### **2.1. Atamalar**

#### **Gösterim-2.1:**

```
int a ;  
a=4 ;    // doğru bir atama  
4=a ;    // yanlış bir atama!
```

#### **2.1.1. Temel Tiplerde Atama**

Atama işlemi, temel (*primitive*) tipler için basittir. Temel tipler, değerleri doğrudan kendileri üzerlerinde tuttukları için, bir temel tipi diğerine atadığımız zaman değişen sadece içerikler olur.

#### **Gösterim-2.2:**

```
int a, b ;  
a=4 ;  
b=5 ;  
a=b ;
```

#### **2.1.2. Nesneler ve Atamalar**

Nesneler için atama işlemleri, temel tiplere göre biraz daha karmaşıktır. Nesneleri yönetmek için referanslar kullanılır; eğer, nesneler için bir atama işlemi söz konusu ise, akla gelmesi gereken ilk şey, bu nesnelere bağlı olan referansın gösterdiği hedeflerde bir değişiklik olacağıdır.

#### **Örnek:** *NesnelerdeAtama.java*

```
class Sayi {  
    int i;  
}  
  
public class NesnelerdeAtama {  
    public static void main(String[] args) {  
        Sayi s1 = new Sayi();  
        Sayi s2 = new Sayi();  
        s1.i = 9;  
        s2.i = 47;  
        System.out.println("1: s1.i: " + s1.i +", s2.i: " + s2.i);  
        s1 = s2; //referanslar kopyalanıyor.. nesneler değil  
        System.out.println("2: s1.i: " + s1.i +", s2.i: " + s2.i);  
        s1.i = 27;  
        System.out.println("3: s1.i: " + s1.i +", s2.i: " + s2.i);  
    }  
}
```

Yukarıda verilen uygulama adım adım açıklanırsa: Önce 2 adet *Sayı* nesnesi oluşturulmaktadır; bunlar *Sayı* tipindeki referanslara bağlı durumdadırlar; **s1** ve **s2**..

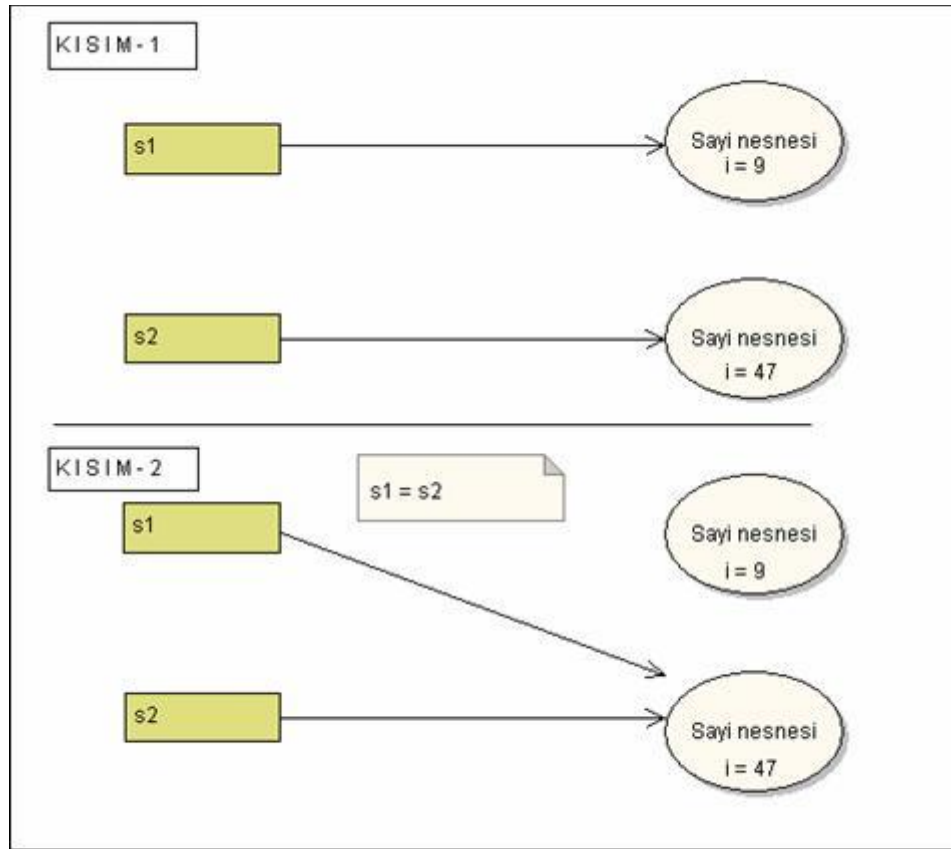
1: s1.i: 9, s2.i: 47

Şu ana kadar bir sorun olmadığını anlaşıp rahatladıktan sonra önemli hamleyi yapıyoruz.

### **Gösterim-2.3:**

```
s1 = s2 ; // referanslar kopyalanıyor... nesneler değil
```

Burada gerçekleşen olay şudur; s1 artık s2'nin işaret ettiği nesneyi göstermektedir. Şekil-2.1, verilen bu örneğin daha iyi anlaşılmasına yardımcı olabilir. Kısım-1 durumun, s2'nin s1'e atanmadan önceki halini göstermektedir. Kısım-2 ise s2'nin s1'e atandıktan sonraki halini göstermektedir.



**Şekil-2.1. Nesnelerde atama ve referans değişikliği**

Kalınan yerden devam edilirse, şimdi s1 ve s2'nin değerlerini ekrana yazdırılırsa, s1.i ve s2.i alanları aynı içeriği taşıdığı görülür.

2: s1.i: 47, s2.i: 47

Bunun nedeni ise bu iki referansın (s1 ve s2) aynı nesneyi göstermeleridir. Son olarak s1 referansının işaret ettiği nesnenin i alanı değiştirilip ekrana yazdırıldığında...

3: s1.i: 27, s2.i: 27

Görüldüğü gibi `s2.i` alanının da değeri değişmiş oldu; nedeni ise yine `s1` ve `s2` referanslarının aynı nesneyi göstermeleridir.

Peki, `s1` referansının daha önceden işaret etmiş olduğu *Sayı* nesnesine ne olacaktır? Cevap vermek için henüz erken ama yinede söylenirse, bu nesne kullanılmayacağından dolayı çöp haline gelecektir ve çöp toplayıcısı (*Garbage Collector*) tarafından temizlenecektir. Görüldüğü gibi tasarımcının nesne temizliği konusunda endişeye kapılmasına gerek yoktur. Çöp toplayıcısını ilerleyen bölümlerde daha ayrıntılı ele alınacaktır.

Bu örneğimizde `s1` referansının `s2`'nin işaret etmiş olduğu nesneyi göstermesini istemeyip yalnızca `s2.i` alanı değerinin `s1.i` alanı değerine atanmasını istenmiş olsaydı, aşağıdaki gibi yazılması yeterli olacaktı...

#### **Gösterim-2.4:**

```
s1.i = s2.i;
```

Bu ifade referansların gösterdikleri nesnelerde herhangi bir değişiklik yapmayacaktır; değişen sadece `s1.i` alanının değeri olacaktır.

## **2.2. Yordamların (*Method*) Çağrılması**

Yordamların parametre kabul ettiklerini ve bu parametreleri alarak işlemler gerçekleştirdiğini biliyoruz. Peki yordamlara parametre olarak neler gitmektedir? Nesnelerin kendisi mi? Yoksa nesnelere ait referanslar mı?

#### **Örnek:** *Pas.java*

```
class Harf {
    char c;
}

public class Pas {
    static void f(Harf h) {
        /* Harf nesnesine yeni bir referans bağlandı (h), yoksa oluşturulan Harf nesnesinin veya
        yeni bir Harf nesnesinin bu yordama gönderilmesi gibi birşey söz konusu değildir. */
        h.c = 'z';
    }

    public static void main(String[] args) {
        Harf x = new Harf(); // Harf nesnesini oluşturuluyor.
        x.c = 'a';           // Harf nesnesinin c alanına değer atandı
        System.out.println("1: x.c: " + x.c);

        f(x); // dikkat

        System.out.println("2: x.c: " + x.c);
    }
}
```

Yukarıda verilen örnekte *Harf* ve *Pas* olarak adlandırılan 2 adet sınıf bulunmaktadır. *Pas* sınıfı **public** olduğu için fiziksel dosyanın ismi *Pas.java*'dır. Bu kadar ön bilgiden sonra program açıklamasına

geçilebilir: İlk olarak *Harf* nesnesi oluşturuluyor ve *Harf* nesnesin `char` tipinde olan `c` alanına `'a'` karakteri atanıyor. Yapılan işlem ekrana yazdırıldıktan sonra *Harf* nesnesine bağlanmış olan *Harf* sınıfı tipindeki `x` referansı `f()` yordamına gönderiliyor; sonra, `f()` yordamı içerisinde daha önceden oluşturulan *Harf* nesnesinin `char` tipinde olan `c` alanına `'z'` karakteri atanıyor. Bilgiler yeniden ekrana yazdırıldığında *Harf* nesnesinin `char` tipinde olan `c` alanındaki değerin değişmiş olduğu görülür. Burada yapılan işlem, kesinlikle, *Harf* nesnesinin yer değiştirmesi değildir; nesnenin bellekteki yeri her zaman sabittir... Yalnızca `f()` yordamı içerisinde *Harf* nesnesine kısa süreli olarak başka bir *Harf* sınıfı tipindeki bir referansın işaret etmiş olmasıdır (böylece *Harf* nesnesine toplam iki referans bağlı olmaktadır biri `x` diğeri ise `h`). Yordamın sonuna gelindiğinde ise `h` referansı geçerlilik alanı bitmektedir; ancak, bu kısa süre içerisinde *Harf* nesnesinin `char` tipinde olan `c` alanını değiştirilmiştir. Uygulamanın sonucu aşağıdaki gibi olur:

```
1: x.c: a
2: x.c: z
```

Yordam çağrımları temel tipler için biraz daha açıktır.

**Örnek:** *IlkelPas.java*

```
public class IlkelPas {
    static void f(double a) {
        System.out.println(a + " gönderildi");
        a = 10 ;
        System.out.println("gonderilen parametrenin degeri 10'a"
                           + "esitlendi");
    }
    public static void main(String[] args) {
        double a = 5 ;
        f(a);
        System.out.println("a --> " + a );
    }
}
```

Yukarıdaki uygulamada `double` tipindeki `a` değişkenine 5 değeri atanmaktadır; daha sonra bu değişkenimiz `double` tipinde parametre kabul eden `f()` yordamına gönderilmektedir. `f()` yordamı içerisinde `a=10` ifadesi, gerçekte, `main()` yordamı içerisindeki `double` tipindeki `a` değişkeni ile hiç bir ilgisi yoktur. Bu iki değişken birbirlerinden tamamen farklıdır. Sonuç olarak, temel tipler değerleri kendi üzerlerinde taşırlar. Yordamlara gönderilen parametreler yerel değişkenler gibidir. Uygulamanın sonucu aşağıdaki gibi olur:

```
5.0 gönderildi
gonderilen parametrenin degeri 10'a esitlendi
a --> 5.0
```

## 2.3. Java Operatörleri

Operatörler programlama dillerinin en temel işlem yapma yeteneğine sahip simgesel isimlerdir. Tüm programlama dillerinde önemli bir yere sahip olup bir işlem operatör ile gerçekleştirilebiliyorsa “en hızlı ve

verimli ancak bu şekilde yapılır” denilebilir. Yalnızca bir operatör ile gerçekleştirilemeyen işlemler, ya bir grup operatörün bir araya getirilmesiyle ya da o işlemi gerçekleştirecek bir yordam (*method*) yazılmasıyla sağlanır. Java dili oldukça zengin ve esnek operatör kümesine sahiptir; örneğin matematiksel, mantıksal, koşulsal, bit düzeyinde vs. gibi birçok operatör kümesi vardır; ve, bunlar içerisinde çeşitli operatörler bulunmaktadır:

- Aritmetik Operatör
- İlişkisel Operatör
- Mantıksal Operatörler
- Bit düzeyinde (*bitwise*) Operatörler

Operatörler, genel olarak, üzerinde işlem yaptığı değişken/sabit sayısına göre tekli operatör (*unary operator*) veya ikili operatör (*binary operator*) olarak sınıflanmaktadır; 3 adet değişken/sabite ihtiyaç duyan operatörlere de üçlü operatör denilir. Yani, tek değişken/sabit üzerinde işlem yapan operatör, iki değişken/sabit üzerinde işlem yapan operatör gibi... Tekli operatörler hem ön-ek (*prefix*) hem de son-ek (*postfix*) işlemlerini desteklerler. Ön-ek’ten kastedilen anlam operatörün değişkenden önce gelmesi, son-ek’ta de operatörden sonra gelmesidir..

→ **operatör** değişken //ön-ek ifadesi

Son-ek işlemlerine örnek olarak,

→ değişken **operatör** // son-ek ifadesi

İkili operatörlerde operatör simgesi ara-ek (*infix*) olarak iki değişkenin ortasında bulunur:

→ değişken1 **operatör** değişken2 //ara-ek

Üçlü operatörlerde ara-ek (*infix*) işlemlerde kullanılır. Java’da üçlü operatör bir tanedir.

→ değişken1 ? değişken2 : değişken3 //ara ek

### 2.3.1. Aritmetik Operatörler

Java programlama dili kayan-noktalı (*floating-point*) ve tamsayılar için birçok aritmetik işlemi destekleyen çeşitli operatörlere sahiptir. Bu işlemler toplama operatörü (+), çıkartma operatörü (-), çarpma operatörü (\*), bölme operatörü (/) ve son olarak da artık bölme (%) operatörüdür.

**Tablo-2.1. Java’da aritmetik operatörler**

| Operatör | Kullanılış            | Açıklama   |
|----------|-----------------------|--|
| +        | değişken1 + değişken2 | değişken1 ile değişken2’yi toplar                              |
| -        | değişken1 - değişken2 | değişken1 ile değişken2’yi çıkarır                             |
| *        | değişken1 * değişken2 | değişken1 ile değişken2’yi çarpır                              |
| /        | değişken1 / değişken2 | değişken1, değişken2 tarafından bölünür                        |
| %        | değişken1 % değişken2 | değişken1’in değişken2 tarafından bölümünden kalan hesaplanır. |

Verilenler bir Java uygulamasında aşağıdaki gibi gösterilebilir:

**Örnek:** *AritmetikOrnek.java*

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

        // Değişkenler atanan değerler
        int a = 57, b = 42;
        double c = 27.475, d = 7.22;
        System.out.println("Degisken Degerleri...");
        System.out.println(" a = " + a);
        System.out.println(" b = " + b);
        System.out.println(" c = " + c);
        System.out.println(" d = " + d);

        // Sayıları topluyoruz
        System.out.println("Toplama...");
        System.out.println(" a + b = " + (a + b));
        System.out.println(" c + d = " + (c + d));

        // Sayıları çıkartıyoruz
        System.out.println("Cikartma...");
        System.out.println(" a - b = " + (a - b));
        System.out.println(" c - d = " + (c - d));

        // Sayıları Çarpıyoruz.
        System.out.println("Carpma...");
        System.out.println(" a * b = " + (a * b));
        System.out.println(" c * d = " + (c * d));

        // Sayıları bölüyoruz
        System.out.println("Bolme...");
        System.out.println(" a / b = " + (a / b));
        System.out.println(" c / d = " + (c / d));

        // Bölme işlemlerinden kalan sayıyı hesaplıyoruz
        System.out.println("Kalan sayıyı hesaplama...");
        System.out.println(" a % b = " + (a % b));
        System.out.println(" c % d = " + (c % d));

        // double ve int tiplerini karışık şekilde kullanıyoruz.
        System.out.println("Karisik tipler...");
        System.out.println(" b + d = " + (b + d));
        System.out.println(" a * c = " + (a * c));
    }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

Degisken Degerleri...

a = 57  
b = 42  
c = 27.475  
d = 7.22

Toplama...

a + b = 99  
c + d = 34.695

Cikartma...

a - b = 15



```

c - d = 20.2550000000000003
Carpma...
a * b = 2394
c * d = 198.369500000000002
Bolme...
a / b = 1
c / d = 3.805401662049862
Kalan sayiyi hesaplama...
a % b = 15
c % d = 5.8150000000000002
Karisik tipler...
b + d = 49.22
a * c = 1566.075

```

Verilen örnek dikkatlice incelenecek olursa, tamsayı ile kayan noktalı sayılar bir operatörün değişkenleri olursa sonuç kayan noktalı sayı olmaktadır. Bu işlemde tamsayı, kendiliğinden kayan noktalı sayıya çevrilir. Aşağıdaki tabloda böylesi dönüştürme işleminde izlenen yol gösterilmiştir:

**Tablo-2.2. Operatörlerin veri tipini etkilemesi/dönüştürmesi**

| Sonuç Veri Tipi | Değişkenlerin Veri Tipleri   |
|-----------------|--|
| long            | Değişkenlerin float veya double tipinden <u>farklı olması</u> ve en az bir değişkenin long tipinde olması    |
| int             | Değişkenlerin float veya double tipinden <u>farklı olması</u> ve değişkenlerin long tipinden farklı olması   |
| double          | En az bir değişkenin double tipinde olması   |
| float           | Değişkenlerin hiçbirinin double tipinde <u>olmaması</u> ve değişkenlerden en az birinin float tipinde olması |

+ ve – operatörleri, aynı zamanda, karakter tipindeki verileri sayısal tipe dönüştürme misyonları da vardır.

**Tablo-2.3. Toplama ve Çıkartma operatörlerinin tip etkilemesi**

| Operatör | Kullanılış Şekli | Açıklama  |
|----------|------------------|---|
| +        | + değişken       | Eğer değişken char, byte veya short tipinde ise int tipine dönüştürür |
| -        | - değişken       | Değişkenin değerini eksi yapar (-1 ile çarpar).                       |

Anlaşılması açısından kısa bir uygulama örneği yazılırsa,

**Örnek:** *OperatorTest.java*

```

public class OperatorTest {
    public static void main(String args[] ) {
        char kr = 'a' ;
        int b = +kr ;           // otomatik olarak int temel tipine çevrildi
        int c = -b ;           // değeri eksi yaptı
        System.out.println("kr = " + kr );
        System.out.println("b = " + b );
        System.out.println("c = " + c );
    }
}

```

**char** temel (*primitive*) bir tiptir; ve, bu tiplere değer atanırken veri tek tırnak içerisinde verilmelidir. Bu örnekte girilen değer 'a' harfidir. Daha sonra + operatörü kullanılarak char değerini int tipine dönüştürülüyor ve son olarak ta bu int değeri - operatörüyle eksi hale getiriliyor. Uygulamanın sonucu aşağıdaki gibi olur:

```
kr = a
b = 97
c = -97
```

### **Dönüştürme (Casting) İşlemi**

Temel bir veri tipi diğer bir temel tipe dönüştürebilir; fakat, oluşacak değer kayıplarından tasarımcı sorumludur.

**Örnek:** *IlkelDonusum.java*

```
public class IlkelDonusum {
    public static void main(String args[]) {
        int a = 5;
        double b = (double) a;
        double x = 4.15 ;

        int y = (int) x ;
        long z = (long) y ;

        System.out.println("b = " + b + " y = " + y + " z = " + z);
    }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

```
b = 5.0 y = 4 z = 4
```

### **Bir Arttırma ve Azaltma**

Java dilinde, aynı C dilinde olduğu gibi, birçok kısaltmalar vardır; bunlar yaşamı bazen daha güzel, bazen de çekilmez kılabilmektedir... İşe yarayan kısaltmalardan iki tanesi arttırma ve azaltma operatörleridir; bu operatörler değişkenin içeriğini bir arttırmak veya azaltmak için kullanılır.

Arttırma ve azaltma operatörleri iki farklı konumda kullanılabilirler: Birincisi ön-ek (*prefix*) -ki bu (-- veya ++) operatörünün, kullanılan değişkenin önüne gelmesi anlamını taşır, diğeri ise son-ek'dir (*postfix*), bu da (-- veya ++) operatörünün değişkenin sonuna gelmesi anlamına gelir. Peki bu operatörlerin değişkenin başına gelmesi ile sonuna gelmesi arasında ne gibi farklar vardır?

**Tablo-2.4. Arttırma ve azaltma operatörü**

| Operatör | Kullanış Şekli    | Açıklama  |
|----------|-------------------|---|
| ++       | <b>değişken++</b> | Önce değişkenin değerini hesaplar sonra değişkenin değerini bir arttırır. |
| ++       | <b>++değişken</b> | Önce değişkenin değerini arttırır sonra değişkenin değerini hesaplar.     |
| --       | <b>değişken--</b> | Önce değişkenin değerini hesaplar sonra değişkenin değerini bir azaltır.  |
| --       | <b>--değişken</b> | Önce değişkenin değerini azaltır sonra değişkenin değerini hesaplar.      |

Örneğin (++a veya --a) şeklinde verilmesinde, önce matematiksel toplama/çıkartma işlemi gerçekleşir; daha sonra değer üretilir. Ancak, (a++ veya a--) şeklinde verilmesi durumunda ise, önce değer üretilir; daha sonra matematiksel toplama/çıkartma işlemi gerçekleşir. Aşağıda verilen kısa program bunu güzel bir şekilde ifade etmektedir:

**Örnek:** *OtomatikArtveAz.java*

```
public class OtomatikArtveAz {

    static void ekranaYaz(String s) {
        System.out.println(s);
    }

    public static void main(String[] args) {

        int i = 1;
        ekranaYaz("i : " + i);
        ekranaYaz("++i : " + ++i);           // önek artırım
        ekranaYaz("i++ : " + i++);           // sonek artırım
        ekranaYaz("i : " + i);
        ekranaYaz("--i : " + --i);           // önek azaltma
        ekranaYaz("i-- : " + i--);           // sonek azaltma
        ekranaYaz("i : " + i);
    }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

```
i : 1
++i : 2
i++ : 2
i : 3
--i : 2
i-- : 2
i : 1
```

### 2.3.2. İlişkisel Operatörler

İlişkisel operatörler iki değeri karşılaştırarak bunlar arasındaki mantıksal ilişkiyi belirlemeye yararlar. Örneğin iki değer birbirine eşit değilse, == operatörüyle bu ilişki sonucu yanlış (**false**) olur. Tablo-2.5’de ilişkisel operatörler ve anlamları verilmiştir:

**Tablo-2.5. İlişkisel operatörler**

| Operatör | Kullanış Şekli         | True değeri döner eğer ki.....               |
|----------|------------------------|--|
| >        | değişken1 > değişken2  | değişken1, değişken2’den büyükse             |
| >=       | değişken1 >= değişken2 | değişken1, değişken2’den büyükse veya eşitse |
| <        | değişken1 < değişken2  | değişken1, değişken2’den küçükse             |
| <=       | değişken1 <= değişken2 | değişken1, değişken2’den küçükse veya eşitse |
| ==       | değişken1 == değişken2 | değişken1, değişken2’ye eşitse               |
| !=       | değişken1 != değişken2 | değişken1, değişken2’ye eşit değilse         |

İlişkisel operatörlerin kullanılması bir Java uygulaması üzerinde gösterilirse,

**Örnek:** *IliskiselDeneme.java*

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

        // değişken bildirimleri
        int i = 37, j = 42, k = 42;

        System.out.println("Degisken degerleri...");
        System.out.println(" i = " + i);
        System.out.println(" j = " + j);
        System.out.println(" k = " + k);

        //Büyüktür
        System.out.println("Buyuktur...");
        System.out.println(" i > j = " + (i > j)); //false - i, j den küçüktür
        System.out.println(" j > i = " + (j > i)); //true - j, i den Büyüktür
        System.out.println(" k > j = " + (k > j)); //false - k, j ye eşit

        //Büyüktür veya eşittir
        System.out.println("Buyuktur veya esittir...");
        System.out.println(" i >= j = " + (i >= j)); //false - i, j den küçüktür
        System.out.println(" j >= i = " + (j >= i)); //true - j, i den büyüktür
        System.out.println(" k >= j = " + (k >= j)); //true - k, j ye eşit

        //Küçüktür
        System.out.println("Kucuktur...");
        System.out.println(" i < j = " + (i < j)); //true - i, j' den küçüktür
        System.out.println(" j < i = " + (j < i)); //false - j, i' den büyüktür
        System.out.println(" k < j = " + (k < j)); //false - k, j ye eşit

        //Küçüktür veya eşittir
        System.out.println("Kucuktur veya esittir...");
        System.out.println(" i <= j = " + (i <= j)); //true - i, j' den küçüktür
        System.out.println(" j <= i = " + (j <= i)); //false - j, i den büyüktür
        System.out.println(" k <= j = " + (k <= j)); //true - k, j ye eşit

        //Eşittir
        System.out.println("Esittir...");
        System.out.println(" i == j = " + (i == j)); //false - i, j' den küçüktür
        System.out.println(" k == j = " + (k == j)); //true - k, j ye eşit

        //Eşit değil
        System.out.println("Esit degil...");
        System.out.println(" i != j = " + (i != j)); //true - i, den küçüktür
        System.out.println(" k != j = " + (k != j)); //false - k, ye eşit
    }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

Degisken degerleri...

```
i = 37
j = 42
k = 42
```

Buyuktur...

```
i > j = false
j > i = true
k > j = false
```

Buyuktur veya esittir...

```
i >= j = false
j >= i = true
k >= j = true
```

Kucuktur...

```
i < j = true
j < i = false
k < j = false
```

Kucuktur veya esittir...

```
i <= j = true
j <= i = false
k <= j = true
```

Equal to...

```
i == j = false
k == j = true
```

Not equal to...

```
i != j = true
k != j = false
```

### 2.3.3. Mantıksal Operatörler

Mantıksal operatörler birden çok karşılaştırma işlemini birleştirip tek bir koşul ifadesi haline getirilmesi için kullanılır. Örneğin bir koşul sağlanması için hem **a**'nın 10'dan büyük olması hem de **b**'nin 55 küçük olması gerekiyorsa iki karşılaştırma koşulu VE mantıksal operatörüyle birleştirilip `a>10 && b<55` şeklinde yazılabilir. Aşağıda, Tablo-2.6'da mantıksal operatörlerin listesi ve anlamları verilmiştir:

**Tablo-2.6. İlişkisel ve koşul operatörlerinin kullanımı**

| Operatör          | Kullanılış Şekli                      | İşlevi/Anlamı           |
|-------------------|---------------------------------------|-------------------------|
| <b>&amp;&amp;</b> | <b>değişken1 &amp;&amp; değişken2</b> | VE operatörü            |
| <b>  </b>         | <b>değişken1    değişken2</b>         | VEYA operatörü          |
| <b>^</b>          | <b>değişken1 ^ değişken2</b>          | YA DA operatörü         |
| <b>!</b>          | <b>! değişken</b>                     | DEĞİLini alma operatörü |

Aşağıda birden çok ilişkisel karşılaştırmanın mantıksal operatörler birleştirilmesi için örnekler verilmiştir (m, n, r ve z değişken adları):

```
m>10 && m<55
(m>0 && r<55) && z==10
a>10 && b<55 || r<99
```

**Not:** Mantıksal operatörlerden **&&** (VE), **||** (VEYA) operatörleri sırasıyla tek karakterli olarak ta kullanılabilir. Örneğin **a&&b** şeklinde bir ifade **a&b** şeklinde de yazılabilir. Aralarında fark, eğer tek karakterli, yani **&** veya **|** şeklinde ise, operatörün heriki yanındaki işlemler/karşılaştırmalar kesinkes yapılır. Ancak, çift karakterli kullanılırsa, yani **&&** veya **||** şeklinde ise, işleme soldan başlanır; eğer tüm ifade bitmeden kesin sonuca ulaşırsa ifadenin geri kalan kısmı gözardı edilir. Örneğin VE işleminde sol taraf yanlış (*false*) ise sonuç kesinkes yanlış olacaktır ve ifadenin sağına bakmaya gerek yoktur.

Mantıksal operatörlerin doğruluk tabloları bit düzeyinde operatörler ayrıtında Tablolarda verilmiştir.

**Örnek:** *KosulOp.java*

```
public class KosulOp {
    public static void main( String args[] ) {
        int a = 2 ;
        int b = 3 ;
        int c = 6 ;
        int d = 1 ;

        /* (a < b) = bu ifadenin doğru (true) olduğunu biliyoruz
           (c < d) = bu ifadenin yanlış (false) olduğunu biliyoruz */

        System.out.println(" (a<b)&&(c<d) --> " + ((a<b)&&(c<d)) );
        System.out.println(" (a<b)|| (c<d) --> " + ((a<b)|| (c<d)) );
        System.out.println(" ! (a<b) --> " + (! (a<b)) );
        System.out.println(" (a<b)&(c<d) --> " + ((a<b)&(c<d)) );
        System.out.println(" (a<b)|(c<d) --> " + ((a<b)|(c<d)) );
        System.out.println(" (a<b)^(c<d) --> " + ((a<b)^(c<d)) );
    }
}
```

Uygulamamızın çıktısı aşağıdaki gibidir.

```
(a < b) && (c < d) --> false
(a < b) || (c < d) --> true
! (a < b) --> false
(a < b) & (c < d) --> false
(a < b) | (c < d) --> true
(a < b) ^ (c < d) --> true
```

#### 2.3.4. bit Düzeyinde (*bitwise*) Operatörler

bit düzeyinde operatörler, adı üzerinde, değişkenlerin/sabitlerin tuttuğu değerlerin doğrudan ikili kodlarının bitleri üzerinde işlem yaparlar. Örneğin 6 sayısının iki karşılığı 0110'dır. Bu değer sonuç 4 bit üzerinde kalmak koşuluyla bir sola kaydırılırsa 1100, tümleyenini alınırsa 1001 olur. Görüldüğü gibi bit düzeyinde operatörler veriyi bit düzeyde etkilemektedir. bit düzeyinde operatörlerin listesi Tablo-2.8.'da ve doğruluk tabloları da sırasıyla Tablo-2.9, 2-10 ve 2-11'de gösterilmişlerdir.

**Tablo-2.7. bit düzeyinde operatörler**

| Operatör | Kullanılış Şekli        | Açıklama                                    |
|----------|-------------------------|---|
| &        | değişken1 & değişken2   | bit düzeyinde VE                            |
|          | değişken1   değişken2   | bit düzeyinde VEYA                          |
| ^        | değişken1 ^ değişken2   | bit düzeyinde YA DA                         |
| ~        | ~değişken               | bit düzeyinde tümleme                       |
| >>       | değişken1 >> değişken2  | bit düzeyinde sağa öteleme                  |
| <<       | değişken1 << değişken2  | bit düzeyinde sağa öteleme                  |
| >>>      | değişken1 >>> değişken2 | bit düzeyinde sağa öteleme (unsigned) ????? |

### ▪ VE (AND) İşlemi/Operatörü

VE işleminde heriki taraf ta doğru (*true*) ise sonuç doğru diğer durumlarda sonuç yanlış (*false*) olur. VE işlemi doğruluk tablosu Tablo-2.9’da verildiği gibidir. bit düzeyinde VE işleminde operatörün hemen sağ ve sol yanında bulunan parametrelerin ikili karşılıkları bit bit VE işlemine sokulur; işlem en sağdaki bitten başlanır. Örneğin 10 ve 9 sayılarının ikili karşılıkları sırasıyla 1010 ve 1011’dir. Herikisi bit düzeyinde VE işlemine sokulursa, sonuç 1000 çıkar:

```
    1010      ⇒  10
    & 1001      ⇒  9
    -----
    1000
```

**Tablo-2.8. VE (AND) işlemi doğruluk tablosu**

| değişken1 | değişken2 | Sonuç |
|-----------|-----------|-------|
| 0         | 0         | 0     |
| 0         | 1         | 0     |
| 1         | 0         | 0     |
| 1         | 1         | 1     |

### ▪ VEYA (OR) İşlemi/Operatörü

VEYA işleminde heriki taraftan birinin doğru (*true*) olması sonucun doğru çıkması için yeterlidir. VEYA işlemi doğruluk tablosu Tablo-2.10’da verildiği gibidir. bit düzeyinde VEYA işleminde operatörün hemen sağ ve sol yanında bulunan parametrelerin ikili karşılıkları bit bit VEYA işlemine sokulur; işlem en sağdaki bitten başlatılır. Örneğin 10 ve 9 sayılarının ikili karşılıkları sırasıyla 1010 ve 1011’dir. Herikisi bit düzeyinde VE işlemine sokulursa, sonuç 1011 çıkar:

```
    1010      ⇒  10
    | 1001      ⇒  9
    -----
    1011
```

**Tablo-2.9. VEYA (OR) işlemi doğruluk tablosu**

| değişken1 | değişken2 | Sonuç |
|-----------|-----------|-------|
| 0         | 0         | 0     |
| 0         | 1         | 1     |
| 1         | 0         | 1     |
| 1         | 1         | 1     |

### ▪ YA DA (*Exclusive Or*) İşlemi/Operatörü

YA DA işleminde heriki taraftan yalnızca birinin doğru (*true*) olması sonucu doğru yapar; heriki tarafın aynı olması durumunda sonuç yanlış (*false*) çıkar. YA DA işlemi doğruluk tablosu Tablo-2.10’da verildiği gibidir.

**Tablo-2.10. Dışlayan YA DA işlemi doğruluk tablosu**

| Değişken1 | değişken2 | Sonuç |
|-----------|-----------|-------|
| 0         | 0         | 0     |
| 0         | 1         | 1     |
| 1         | 0         | 1     |
| 1         | 1         | 0     |

VE ve VEYA işlemlerinde kullanılan örnek sayıları YA DA için de gösterilirse, sonuç aşağıdaki gibi olur:

```
    1010      ⇨  10
   ^ 1001      ⇨   9
-----
    0011
```

### TÜMLEME (NOT) İşlemi/Operatörü

a bir değişken adı ise **~a** ifadesi açılımı :  $\sim a = (-a) - 1$ , yani,  $\sim 10 = (-10) - 1 = -11$  sonucunu verir.

**Örnek:** *BitwiseOrnek2.java*

```
public class BitwiseOrnek2 {

    public static void main( String args[] ) {
        int a = 10, b = 9, c = 8 ;
        System.out.println(" (a & b) --> " + (a & b) );
        System.out.println(" (a | b) --> " + (a | b) );
        System.out.println(" (a ^ b) --> " + (a ^ b) );

        System.out.println(" ( ~a ) --> " + ( ~a ) );
        System.out.println(" ( ~b ) --> " + ( ~b ) );
        System.out.println(" ( ~c ) --> " + ( ~c ) );
    }
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

```
(a & b) --> 8
(a | b) --> 11
(a ^ b) --> 3
( ~a ) --> -11
( ~b ) --> -10
( ~c ) --> -9
```

VE, VEYA ve YA DA (*Exclusive Or*) operatörleri birden çok mantıksal sonuç içeren ifadelerde kullanılabilir!



## Öteleme (Shift) Operatörleri

bit düzeyinde işlem yapan bir grup operatörün adı öteleme operatörleri olarak adlandırılırlar; bunlar >>, >>> ve <<< simgeleriyle gösterilmektedir. Öteleme operatörleri veri üzerindeki bitlerin sağa veya sola kaydırılması amacıyla kullanılır.

Aşağıdaki örneğimiz bu operatörlerin Java uygulamalarında nasıl kullanılacaklarına ait bir fikir verebilir.

### Örnek: Bitwise.java

```
public class Bitwise {  
    public static void main( String args[] ) {  
        int a = 9 ;  
  
        System.out.println(" (a >> 1) -->" + (a >> 1) );  
        System.out.println(" (a >> 2) -->" + (a >> 2) );  
  
        System.out.println(" (a << 1) -->" + (a << 1) );  
        System.out.println(" (a << 2) -->" + (a << 2) );  
  
        System.out.println(" (a >>> 2) -->" + (a >>> 2) );  
    }  
}
```

Verilen örnekte a değişkenine 9 sayısı atanmıştır; bu sayının ikili karşılığı aşağıdaki gibi bulunur:

$$9 = (1001)_2 = 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0$$

Yani, 910 sayısının ikili tabandaki karşılığı 1001 olmaktadır. Buna göre a değişkeni üzerinde öteleme operatörünün etkisi aşağıda açıklandığı gibi olur:

(a >> 1) şeklinde ifade ile, 9 sayısı ikili karşılığı olan 1001 bitleri sağa doğru 1 basamak kaydırılır; \_100, boşlan yere 0 yerleştirildiğinde sonuç elde edilir; dolayısıyla 0100 elde edilir ve bunun ondalık karşılığı 4 çıkar.

(a >> 2) şeklinde ifade ile 9 sayısı ikili karşılığı olan 1001 bitlerini sağa doğru 2 basamak kaydırılır; \_\_10, boşalan yerlere 0 yerleştirildiğinde sonuç elde edilir; dolayısıyla 0010 elde edilir ve bunun ondalık karşılığı 2 çıkar.

(a << 1) şeklinde ifade ile 9 sayısı ikili karşılığı olan 1001 bitlerini sola doğru 1 basamak kaydırılır; 1001\_, boşalan yere 0 yerleştirildiğinde sonuç elde edilir; dolayısıyla 10010 elde edilir ve bunun ondalık karşılığı 18 çıkar.

(a << 2) şeklinde ifade ile 9 sayısı ikili karşılığı olan 1001 bitlerini sola doğru 2 basamak kaydırılır; 1001\_\_, boşalan yerlere 0 yerleştirildiğinde sonuç elde edilir; dolayısıyla 100100 elde edilir ve bunun ondalık karşılığı 36 çıkar.

(a >>> 2) şeklinde verilen ifadenin (a >> 2) ile arasında sonuç olarak bir fark yoktur, sonuç olarak yine 2 elde edilir. ">>>" operatörü işaretli (signed) sağa doğru kaydırıp yapar.

Eğer char, byte, veya short tiplerinde kaydırma işlemi yaparsanız bu tipler ilk önce int tipine dönüştürülürler. Eğer long tipinde kaydırma işlemi gerçekleştiriyorsanız o zaman yine long tipinde bir sonuç elde ederseniz.

Uygulamanın sonucu aşağıdaki gibi olur:.

```
(a >> 1) -->4  
(a >> 2) -->2
```

```
(a << 1) -->18
(a << 2) -->36
(a >>> 2) -->2
```

bit düzeyinde operatörleri tamsayı veriler üzerinde uygulamak anlamlıdır.

### 2.3.5. Atama Operatörleri

Atama operatörü en temel operatördür denilebilir; atama işlemi, bir değeri veya değişkenini içeriğini bir başka değişkene yerleştirmektir. Hemen hem tüm programlama dillerinde atama operatörü olarak = simgesi kullanılır; yalnızca Pascal ve benzeri dillerde := karakter çifti kullanılır.

#### Örnek: EnBuyukSayilar.java

```
public class EnBuyukSayilar {

    public static void ekranaBas(String deger) {
        System.out.println(deger);
    }
    public static void main( String args[] ) {

        // tamsayılar
        byte enbuyukByte = Byte.MAX_VALUE;
        short enbuyukShort = Short.MAX_VALUE;
        int enbuyukInteger = Integer.MAX_VALUE;
        long enbuyukLong = Long.MAX_VALUE;

        ekranaBas("enbuyukByte-->" + enbuyukByte );
        ekranaBas("enbuyukShort-->" + enbuyukShort );
        ekranaBas("enbuyukInteger-->" + enbuyukInteger );
        ekranaBas("enbuyukLong-->" + enbuyukLong );
        ekranaBas("");

        // gerçek sayılar
        float enbuyukFloat = Float.MAX_VALUE;
        double enbuyukDouble = Double.MAX_VALUE;
        ekranaBas("enbuyukFloat-->" + enbuyukFloat );
        ekranaBas("enbuyukDouble-->" + enbuyukDouble );
        ekranaBas("");

        // diğer temel (primitive) tipler
        char birChar = 'S';
        boolean birBoolean = true;

        ekranaBas("birChar-->" + birChar );
        ekranaBas("birBoolean-->" + birBoolean );
    }

}
```

Java'da C dilinde olduğu gibi bitişik atama operatörleri de vardır; bunlar atama operatörüyle diğer operatörlerden birinin birleştirilmesinden oluşurlar. Böylece kısa bir yazılımla hem aritmetik, öteleme gibi işlemler yaptırılır hem de atama yapılır. Yani, ifade yazımı kolaylaştırır. Örneğin, int tipinde olan toplam değişkeninin değeri 1 arttırmak için aşağıda gibi bir ifade kullanılabilir:

```
toplam = toplam + 1 ;
```

Bu ifade bitişik atama operatörüyle aşağıdaki gibi yazılabilir. Görüldüğü gibi değişken adı yukarıdaki yazımda 2, aşağıda yazımda ise 1 kez yazılmıştır...

toplam += 1 ;

Tablo-2-12’de bitişik atama operatörlerinin listesi görülmektedir; bu operatör, özellikle, uzun değişken kullanıldığı durumlarda yazım kolaylığı sağlarlar.

**Tablo-2.11. Java’daki bitişik atama operatörleri**

| Operatör | Kullanılış Şekli         | Eşittir                             |
|----------|--------------------------|-------------------------------------|
| +=       | değişken1 += değişken2   | değişken1 = değişken1 + değişken2   |
| -=       | değişken1 -= değişken2   | değişken1 = değişken1 – değişken2   |
| *=       | değişken1 *= değişken2   | değişken1 = değişken1 * değişken2   |
| /=       | değişken1 /= değişken2   | değişken1 = değişken1 / değişken2   |
| %=       | değişken1 %= değişken2   | değişken1 = değişken1 % değişken2   |
| &=       | değişken1 &= değişken2   | değişken1 = değişken1 & değişken2   |
| =        | değişken1  = değişken2   | değişken1 = değişken1   değişken2   |
| ^=       | değişken1 ^= değişken2   | değişken1 = değişken1 ^ değişken2   |
| <<=      | değişken1 <<= değişken2  | değişken1 = değişken1 << değişken2  |
| >>=      | değişken1 >>= değişken2  | değişken1 = değişken1 >> değişken2  |
| >>>=     | değişken1 >>>= değişken2 | değişken1 = değişken1 >>> değişken2 |

### String (+) Operatörü

“+” operatörü String verilerde birleştirme görevi görür; eğer ifade String ile başlarsa, onu izleyen veri tipleri de kendiliğinden String’e dönüştürülür. Bu dönüştürme sırrı ve ayrıntısı ilerleyen bölümlerde ele alınmaktadır:

**Örnek:** OtomatikCevirim.java

```
public class OtomatikCevirim {  
  
    public static void main(String args[]) {  
        int x = 0, y = 1, z = 2;  
        System.out.println("Sonuc =" + x + y + z);  
    }  
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

Sonuc =012

Görüldüğü gibi String bir ifadeden sonra gelen tamsayılar toplanmadı; doğrudan String nesnesine çevrilip ekrana çıktı olarak gönderildiler...

### Nesnelerin Karşılaştırılması

Nesnelerin eşit olup olmadığını == veya != operatörleriyle sıranabilir!

**Örnek:** Denklik.java

```
public class Denklik {  
  
    public static void main(String[] args)  
    {  
        Integer a1 = new Integer(47);  
        Integer a2 = new Integer(47);  
  
        System.out.println(a1 == a2);  
        System.out.println(a1 != a2);  
    }  
}
```

Önce Integer sınıfı tipinde olan n1 ve n2 referansları, içlerinde 47 sayısını tutan Integer nesnelere bağlı durumdadırlar. Uygulamanın sonucu olarak aşağıdaki gibi değerler bekliyor olabiliriz...

True

False

Ancak ne yazık ki, sonuç yukarıdaki gibi değildir! Nedeni ise, elimizde iki adet farklı Integer nesnesi bulunmaktadır. Bu nesnelerin taşıdıkları değerler birbirlerine eşittir; ancak, `a1==a2` ifadesi kullanılarak şu denilmiş oldu “a1 ve a2 referanslarının işaret etmiş oldukları nesneler aynı mı?” Yanıt tahmin edilebileceği gibi hayırdır. Yani, false’dur. a1 ve a2 ayrı Integer nesnelerini işaret etmektedirler; eşit olan tek şey, bu iki ayrı nesnenin tuttukları değer 47 olmasıdır (-ki bu eşitliği `a1=a2` ifadesi ile yakalayamayız). Programımızın çıktısı aşağıdaki gibidir.

False

True

Peki, verilen örnekteki Integer nesneleri yerine temel tip olan int tipi kullanılsaydı sonuç ne olurdu?

**Örnek:** *IntIcinDenklik.java*

```
public class IntIcinDenklik {  
  
    public static void main(String[] args) {  
  
        int s1 = 47;  
        int s2 = 47;  
        System.out.println(s1 == s2);  
        System.out.println(s1 != s2);  
    }  
}
```

Bu uygulamanın sonucu aşağıdaki gibi olur:

True

False

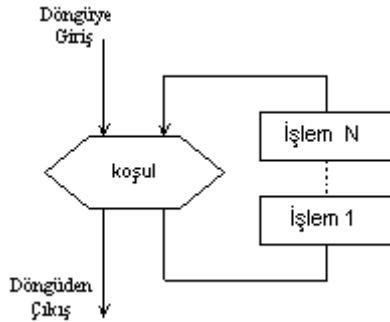
Temel (*primitive*) tipler değerleri doğrudan kendi üzerlerinde taşıdıkları için `==` operatörüyle `s1` ve `s2` değişimleri değerleri karşılaştırıldı ve doğru (*true*) yanıtı döndürüldü. Benzer şekilde `!=` operatörü de `s1` ve `s2` değişimleri değerlerini karşılaştırdı ve yanlış (*false*) döndürüldü. Sonuçlar beklendiği gibi...

## 2.4. Kontrol Deyimleri/İfadeler

Kontrol deyimleri bir uygulamanın yürütülmesi sırasında program akışını yönlendiren yapılar/kalıplardır. Kontrol deyimi olmaksızın bir uygulama yazılması neredeyse olanaksızdır denilebilir. Java programlama dilinde toplam 4 adet kontrol ifadesi bulunur:

- Döngü : **while, do-while, for**
- Karşılaştırma : **if-else, switch-case**
- Dallanma : **break, continue, label:, return**
- İstisna : **try-catch-finally, throw**

### 2.4.1. Döngü Deyimleri



Döngü deyimleri aynı işlemin farklı parametre değerleri üzerinde yapılması için kullanılan yineleme/tekrarlama işleri için kullanılır. Java'da C dilinde olduğu gibi `while`, `do-while` ve `for` olarak adlandırılan üç farklı döngü deyimi vardır.

#### ▪ **while Döngü Deyimi**

`while` deyimi belirli bir grup kod öbeğini döngü koşulu doğru (*true*) olduğu sürece devamlı yineler. Genel yazım şekli aşağıdaki gibidir:

```
while (koşul) {
    çalışması istenen kod bloğu
}
```

Program akışı `while` deyimine geldiğinde döngü koşuluna bakılır; olumlu/doğru ise çevrime girerek çalışması istenen kod öbeği yürütülür; yineleme döngü koşulu olumsuz/yanlış olana kadar sürer.

**Örnek:** *WhileOrnek.java*

```

public class WhileOrnek {
    int i = 0 ; //döngü kontrol değişkeni

    while (i < 5 ) {
        System.out.println("i = " + i);
        i++ ;
    }
    System.out.println("Sayma islemi tamamlandı.");
}

```

Uygulamanın sonucu aşağıdaki gibi olur:

```

i = 0
i = 1
i = 2
i = 3
i = 4

```

Sayma islemi tamamlandı.

#### ▪ **do-while Döngü Deyimi**

Bu döngü deyiminde koşul sınaması döngü sonunda yapılır; dolayısıyla çevrim kod öbeği en az birkez yürütülmüş olur. Genel yazım şekli ve çizimsel gösterimi aşağıdaki gibidir:

```

do {

    çalışması istenen kod bloğu

} while(koşul);

```

**Örnek:** *WhileDoOrnek.java*

```

public class WhileDoOrnek {

    public static void main(String args[])
    {
        int i = 0 ;          //döngü koşul değişkeni
        do {
            System.out.println("i = " + i);
            i++ ;
        } while ( i < 0 );
        System.out.println("Sayma islemi tamamlandı.");
    }
}

```

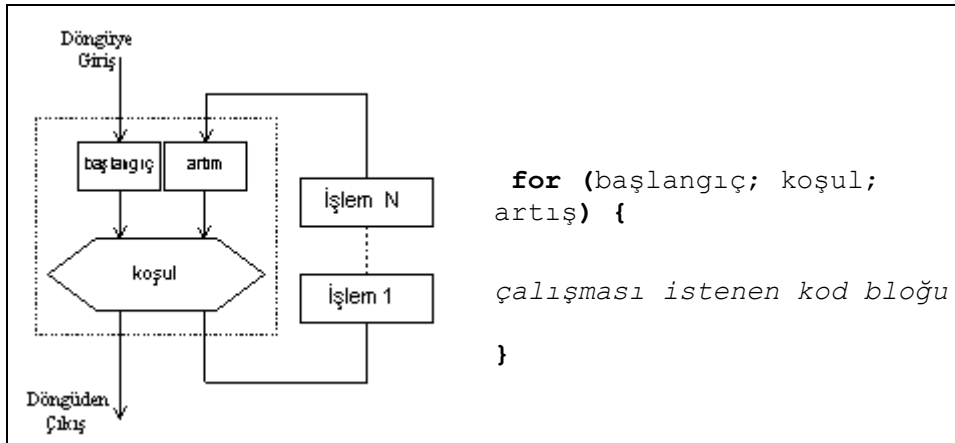
Verilen örnekte `while` kullanılmış olsaydı ekranda sadece “Sayma işlemi tamamladı” cümlesi görülecekti; ancak, `do-while` kullanılmasıyla yürütülmesi istenen kod öbeği koşul değişkeni çevri m koşulunu sağlamasa da çevrime girilir...

`while` ve `do-while` döngü deyimleri kullanırken dikkat edilmesi gereken unsurlar aşağıdaki gibi belirtilebilir;

1. Döngü koşul değişkenine uygun bir şekilde değer atandığına dikkat ediniz.
2. Döngü durumunun doğru (*true*) ile başlamasına dikkat ediniz.
3. Döngü koşul değişkeninin çevrim içerisinde güncellediğinden emin olunuz; aksi durumda sonsuz çevrime girilebilir!

#### ▪ **for Döngü Deyimi**

`for` deyimde çevrim işlemleri daha bir derli toplu yapılabilir; bu döngü deyiminde koşulda kullanılan çevrim değişkeni, koşul ifadesi ve çevrim sayacı artımı `for` ifadesi içerisinde verilir. Genel yazım şekli ve çizimle gösterilmesi aşağıdaki gibi verilebilir:



Görüldüğü gibi `for` deyiminde “;” ile ayrılmış 3 parametre vardır; birincisi çevrim sayacı, ikincisi koşul ifadesi ve üçüncüsü de sayacın artım miktarı ifadesidir. Eğer, kodun daha önceki kısımlarda sayacı değişkeni halihazırda varsa `başlangıç`, `artış` kod öbeği kısmında yapıyorsa `artış` bilgisi verilmeyebilir. Bu durumda bu alanlar bol bırakılır.

#### **Örnek:** *ForOrnek.java*

```
public class ForOrnek {  
    public static void main(String args[]) {  
        for (int i=0 ; i < 5 ; i ++ ) {  
            System.out.println("i = " + i);  
        }  
    }  
}
```

Uygulamanın çıktısı aşağıdaki gibi olur:

i = 0

```
i = 1  
i = 2  
i = 3  
i = 4
```

for deyimi kullanılarak sonsuz çevrim oluşturulmak istenirse aşağıdaki gibi yazılması yeterlidir.

```
for ( ;1; ) {           // sonsuz döngü  
    ...  
}
```

for ifadesinde birden fazla değişken kullanabilirsiniz; yani, birden çok sayaç değişkeni olabilir veya koşul mantıksal operatörler kullanılarak birden çok karşılaştırma içerebilir.

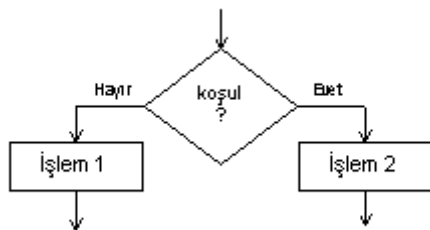
### **Örnek:** *ForOrnekVersiyon2.java*

```
public class ForOrnekVersiyon2 {  
    public static void main(String args[]) {  
        for ( int i = 0, j = 0 ; i < 20 ; i++, j++ ) {  
            i *= j ;  
            System.out.println("i = " + i + " j = " + j);  
        }  
    }  
}
```

Uygulamamızın çıktısı aşağıdaki gibidir:

```
i = 0 j = 0  
i = 1 j = 1  
i = 4 j = 2  
i = 15 j = 3  
i = 64 j = 4
```

### 2.4.2. Karşılaştırma Deyimleri



**<![endif]>**Karşılaştırma deyimleri belirli bir koşula göre farklı işlemler yaptırılacağı zaman kullanılır. Örneğin *adet* adlı değişken değeri 5'ten küçük olduğundan farklı, 5'ten büyük veya eşit olduğunda farklı işler kodlar yürütülecekse bu işi yapabilmek için karşılaştırma deyimlerine gereksinim duyulur. Java'da *if-else* ve *switch-case* olmak üzere iki farklı karşılaştırma deyimi vardır.

#### ▪ **if-else Deyimi**



Koşula göre program akışı değiştirilmek isteniyorsa **if** kullanılabilir. Genel yazım ifadesi aşağıdaki gibidir:

```
if (koşul) {  
    durum true olduğunda çalışması istenen kod bloğu  
} else {  
    durum false olduğunda çalışması istenen kod bloğu  
}
```

**Örnek:** *IfElseTest.java*

```
public class IfElseTest {  
    public static void main(String[] args) {  
  
        int puan = 76;  
        char sonuc;  
  
        if (puan >= 90) {  
            sonuc = 'A';  
        } else if (puan >= 80) {  
            sonuc = 'B';  
        } else if (puan >= 70) {  
            sonuc = 'C';  
        } else if (puan >= 60) {  
            sonuc = 'D';  
        } else {  
            sonuc = 'F';  
        }  
        System.out.println("Sonuc = " + sonuc);  
    }  
}
```

`int` tipindeki *puan* değişkeninin değeri 70'den büyük olduğu için sonuç aşağıdaki gibi olacaktır:

Sonuc = C

- **3'lü if-else:** 3'lü *if-else* deyimi önceki *if-else* deyimine alternatif olarak kullanılabilir. Genel yazılış biçimi;

**mantıksal-ifade? deger0: deger1**

Eğer mantıksal ifade doğru (*true*) ise *deger0* hesaplanır; eğer yanlış (*false*) ise *deger1* hesaplanır.

- **Kestirme sonuç:** *VE* işleminde (bkz. Mantıksal Operatörler) iki değer doğru (*true*) olması durumunda sonuç doğru oluyordu... Eğer *if* deyiminde *VE* işlemi kullanılmış ise ve ilk değerden yanlış dönmüş ise, ikinci değer kesinlikle hesaplanmaz. Bunun nedeni, iki değer sonucunun *VE* işlemine göre doğru dönmesi imkansızlığıdır. Kestirme sonuç özelliği sayesinde uygulamalar gereksiz hesaplamalardan kurtulmuş olur; bununda getirisi performansdır.

**Örnek:** *Kestirme.java*

```
public class Kestirme {
```

```

public static boolean hesaplaBir(int a) {
    System.out.println("hesaplaBir yordamına girildi");
    return a > 1 ; }

public static boolean hesaplaIki(int a) {
    System.out.println("hesaplaIki yordamına girildi");
    return a > 2 ; }

public static void main(String[] args) {
    System.out.println("Baslangic");
    //hesaplaBir(0) --> false deger doner
    //hesaplaIki(3) --> true deger doner

    System.out.println("hesaplaBir(0) && hesaplaIki(3)");
    if ( hesaplaBir(0) && hesaplaIki(3) ) {
        System.out.println(" 1 -true ");
    } else {
        System.out.println(" 1 -false "); }
    System.out.println("-----");
    System.out.println("hesaplaBir(0) || hesaplaIki(3)");

    if (hesaplaBir(0) || hesaplaIki(3)) {
        System.out.println(" 2 -true ");
    } else {
        System.out.println(" 2 -false "); }
    System.out.println("-----");
    System.out.println("hesaplaBir(0) & hesaplaIki(3)");

    if (hesaplaBir(0) & hesaplaIki(3)) {
        System.out.println(" 3 -true ");
    } else {
        System.out.println(" 3 -false "); }
    System.out.println("-----");
    System.out.println("hesaplaBir(0) | hesaplaIki(3)");

    if (hesaplaBir(0) | hesaplaIki(3)) {
        System.out.println(" 4 -true ");
    } else {
        System.out.println(" 4 -false "); }
    System.out.println("-----");
    System.out.println("hesaplaBir(0) ^ hesaplaIki(3)");

    if (hesaplaBir(0) ^ hesaplaIki(3)) {
        System.out.println(" 5 -true ");
    } else {
        System.out.println(" 5 -true "); }
    System.out.println("Son..");
}
}

```

Programı açıklanmaya çalışılırsa, bu uygulamada **hesaplaBir()** ve **hesaplaIki()** adında iki adet yordam bulunmaktadır. Bunlar **int** tipinde parametre kabul edip mantıksal sonuç döndürmektedirler. Bu yordamlara girildiği zaman ekrana kendilerini tanıtan bir yazı çıkartıyoruz -ki gerçekten kestirme olup olmadığını anlayalım:

**hesaplaBir()** yordamı kendisine gelen **int** tipindeki parametreyi alıp 1'den büyük mü diye bir sınamaktadır. Burada **hesaplaBir()** yordamına parametre olarak sıfır sayısı gönderildiğinden dönecek

değerin olumsuz olacağı biliniyor. **hesaplaIki()** yordamına da aynı şekilde üç sayısı gönderilerek bu yordamın bize olumlu değer döndüreceğinden emin olduktan sonra işlemlere başlıyoruz.

İlk önce, yordamlardan bize geri dönen değerler VE işlemine tabii tutuluyor. Görüldüğü gibi yalnızca **hesaplaBir()** yordamına giriyor. Çünkü **hesaplaBir()** yordamından olumsuz değer dönmektedir; VE işleminde olumlu dönebilmesi için iki değerinde olumlu olması gerektiğinden, **hesaplaIki()** yordamı çağrılmayarak kestirme özelliği kullanılmıştır.

İkinci olarak, gelen değerler VEYA işlemine tabii tutuluyor; görüldüğü gibi hem **hesaplaBir()** hem de **hesaplaIki()** yordamları çağrılmaktadır. VEYA tablosu hatırlanırsa, sonucun olumsuz olması için iki değerinde olumsuz olması gerekmektedir. Burada ilk değerden olumsuz değeri döndü; ancak, ikinci değer de hesaplanması gerek, aksi durumda sonucun öğrenilmesi imkansız olur. Bu nedenden dolayı, burada kestirme işlemi gerçekleşmedi. Ancak, ilk değer olumlu dönseydi, o zaman, ikinci yordam olan **hesaplaIki()** hiç çağrılmayacaktı. Çünkü VEYA işlemleri sonucunun olumlu olabilmesi için parametrelerden birisinin olumlu olması gereklidir.

Üçüncü olarak, değerler yine VE işlemine tabii tutuluyor; ancak, burada (&) operatörü kullanıldığı için kestirme işlemi ortadan kalkmaktadır; iki yordam da ayrı ayrı çağrılır.

Dördüncü olarak, değerler VEYA işlemine tabii tutuluyor; fakat (||) operatörü zaten kestirme işlemini ortadan kalkmaktadır ve iki yordamda ayrı ayrı çağrılır.

Son olarak, değerler YA DA (Exclusive Or) işlemine tabii tutuluyor; bu işlemde kesinlikle iki değere de bakılma zorunluluğu olduğundan kestirme işlemi söz konusu olmaz.

Uygulamanın sonucu aşağıdaki gibi olur:

```
Baslangic
hesaplaBir(0) && hesaplaIki(3)
hesaplaBir yordamına girildi
1 -false
-----
hesaplaBir(0) || hesaplaIki(3)
hesaplaBir yordamına girildi
hesaplaIki yordamına girildi
2 -true
-----
hesaplaBir(0) & hesaplaIki(3)
hesaplaBir yordamına girildi
hesaplaIki yordamına girildi
3 -false
-----
hesaplaBir(0) | hesaplaIki(3)
hesaplaBir yordamına girildi
hesaplaIki yordamına girildi
4 -true
-----
hesaplaBir(0) ^ hesaplaIki(3)
hesaplaBir yordamına girildi
hesaplaIki yordamına girildi
5 -true
Son..
```

#### ▪ switch Deyimi

switch deyimi tamsayıların karşılaştırılması ile doğru koşulların elde edilmesini sağlayan mekanizmadır. switch deyimini genel yazım biçimi aşağıdaki gibidir:

```
switch(tamsayı) {
    case uygun-tamsayı-deger1 : çalışması istenen kod bloğu; break;
    case uygun-tamsayı-deger2 : çalışması istenen kod bloğu; break;
    case uygun-tamsayı-deger3 : çalışması istenen kod bloğu; break;
    case uygun-tamsayı-deger4 : çalışması istenen kod bloğu; break;
    case uygun-tamsayı-deger5 : çalışması istenen kod bloğu; break;
    //...
    default: çalışması istenen kod bloğu ;
}
```

switch deyimi içersindeki tamsayı ile, bu tamsayıya karşılık gelen koşula girilir ve istenen kod bloğu çalıştırılır. Kod bloklarından sonra break koymak gerekir aksi takdirde uygun koşul bulduktan sonraki her koşula girilecektir. Eğer tamsayımız koşullardan hiçbirine uymuyorsa default koşulundaki kod bloğu çalıştırılarak son bulur.

**Örnek:** *AylarSwitchTest.java*

```
public class AylarSwitchTest {

    public static void main(String[] args) {

        int ay = 8;
        switch (ay) {
            case 1: System.out.println("Ocak"); break;
            case 2: System.out.println("Subat"); break;
            case 3: System.out.println("Mart"); break;
            case 4: System.out.println("Nisan"); break;
            case 5: System.out.println("Mayis"); break;
            case 6: System.out.println("Haziran"); break;
            case 7: System.out.println("Temmuz"); break;
            case 8: System.out.println("Agustos"); break;
            case 9: System.out.println("Eylul"); break;
            case 10: System.out.println("Ekim"); break;
            case 11: System.out.println("Kasim"); break;
            case 12: System.out.println("Aralik"); break;
        }
    }
}
```

Uygulamanın sonucu,

Agustos

Yukarıdaki uygulamadan break anahtar kelimelerini kaldırıp yeniden yazılırsa,

**Örnek:** *AylarSwitchTestNoBreak.java*

```
public class AylarSwitchTestNoBreak {
```

```

public static void main(String[] args) {

    int ay = 8;
    switch (ay) {
        case 1: System.out.println("Ocak");
        case 2: System.out.println("Subat");
        case 3: System.out.println("Mart");
        case 4: System.out.println("Nisan");
        case 5: System.out.println("Mayis");
        case 6: System.out.println("Haziran");
        case 7: System.out.println("Temmuz");
        case 8: System.out.println("Agustos");
        case 9: System.out.println("Eylul");
        case 10: System.out.println("Ekim");
        case 11: System.out.println("Kasim");
        case 12: System.out.println("Aralik");
    }
}
}

```

Uygulamanın çıktısı aşağıdaki gibi olur. Yalnız hemen uyaralım ki, bu istenmeyen bir durumdur:

```

Agustos
Eylul
Ekim
Kasim
Aralik

```

Aşağıda verilen uygulama ise switch deyiminde default kullanımı göstermek amacıyla yazılmıştır:

**Örnek:** *AylarSwitchDefaultTest.java*

```

public class AylarSwitchDefaultTest {

    public static void main(String[] args) {

        int ay = 25;
        switch (ay) {
            case 1: System.out.println("Ocak"); break;
            case 2: System.out.println("Subat"); break;
            case 3: System.out.println("Mart"); break;
            case 4: System.out.println("Nisan"); break;
            case 5: System.out.println("Mayis"); break;
            case 6: System.out.println("Haziran"); break;
            case 7: System.out.println("Temmuz"); break;
            case 8: System.out.println("Agustos"); break;
            case 9: System.out.println("Eylul"); break;
            case 10: System.out.println("Ekim"); break;
            case 11: System.out.println("Kasim"); break;
            case 12: System.out.println("Aralik"); break;
            default: System.out.println("Aranilan Kosul Bulunamadi!!");
        }
    }
}

```

Bu örneğimizde istenen durum hiçbir koşula uymadığı için default koşulundaki kod öbeği çalışmaktadır.

Uygulamanın sonucu aşağıdaki gibi olur:

```
Aranılan Kosul Bulunamadi !!
```

### 2.4.3. Dallandırma Deyimleri

Java programlama dilinde dallandırma ifadeleri toplam 3 adettir.

- break
- continue
- return

#### ▪ break Deyimi

break deyiminin 2 farklı uyarlaması bulunur; birisi etiketli (labeled), diğeri ise etiketsiz (unlabeled)'dir. Etiketsiz break, switch deyiminde nasıl kullanıldığı görülmüştü... Etiketsiz break sayesinde koşul sağlandığında switch deyimini sonlanması sağlanıyordu. break deyimi aynı şekilde while, do-while veya for deyimlerinden çıkılması için de kullanılabilir.

**Örnek:** *BreakTest.java*

```
public class BreakTest {
    public static void main(String[] args) {
        for ( int i = 0; i < 100; i++ ) {
            if ( i ==9 ) {        // for döngüsünü kırıyor
                break;
            }
            System.out.println("i =" +i);
        }
        System.out.println("Donguden cikti");
    }
}
```

Normalde 0 dan 99'a kadar dönmesi gereken kod bloğu, i değişkenin 9 değerine gelmesiyle for dönüşünün dışına çıktı. Uygulamanın çıktısı aşağıdaki gibidir.

```
i =0
i =1
i =2
i =3
i =4
i =5
i =6
i =7
i =8
Donguden cikti
```

Etiketiz break ifadeleri en içteki while, do-while veya for döngü ifadelerini sona erdirirken, etiketli break ifadeleri etiket (label) hangi döngünün başına konulmuş ise o döngü sistemini sona erdirir.

**Örnek:** *BreakTestEtiketli.java*

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

        kiril :
        for ( int j = 0 ; j < 10 ; j ++ ) {

            for ( int i = 0; i < 100; i++ ) {
                if ( i ==9 ) { // for dongusunu kiriyor
break kiril;
                }
                System.out.println("i =" +i);
            }
            System.out.println("Donguden cikti");
            System.out.println("j =" +j);

        }
    }
}
```

Yukarıdaki örneğimizde etiket kullanarak, daha geniş çaplı bir döngü sisteminden çıkmış olduk. Uygulamamızın çıktısı aşağıdaki gibidir.

```
i =0
i =1
i =2
i =3
i =4
i =5
i =6
i =7
i =8
```

▪ **continue Deyimi**

continue ifadesi, döngü içerisinde o anki devir işleminin pas geçilmesini ve bir sonraki devir işleminin başlamasını sağlayan bir mekanizmadır. continue ifadeleri de break ifadeleri gibi iki çeşide ayrılır. Etiketsiz continue ve etiketli continue. Etiketsiz continue en içteki döngü içerisinde etkili olurken, etiketli continue ise başına konulduğu döngü sisteminin etkiler.

**Örnek:** *ContinueTest.java*

```
public class ContinueTest {

    public static void main(String[] args) {
        for ( int i = 0; i < 10; i++ ) {
            if ( i == 5 ) {          // for döngüsünü kırıyor
                continue;
            }
            System.out.println("i =" +i);
        }
        System.out.println("Donguden cikti");
    }
}
```

```
}  
}
```

Uygulamanın sonucu aşağıdaki gibi olur:

```
i =0  
i =1  
i =2  
i =3  
i =4  
i =6  
i =7  
i =8  
i =9  
Donguden cikti
```

Ekrana yazılan sonuca dikkatli bakılırsa 5 değerinin olmadığı görülür; `continue` deyimi `break` gibi döngüleri kırmaz, yalnızca belli durumlardaki döngü işleminin atlanmasını sağlar sağlar.

**Örnek:** *ContinueTestEtiketli.java*

```
public class ContinueTestEtiketli {  
    public static void main(String[] args) {  
        pas :  
        for ( int j = 0 ; j < 6 ; j ++ ) {  
            for ( int i = 0; i < 5; i++ ) {  
                if ( i ==3 ) { // for döngüsünü kırıyor  
                    continue pas;  
                }  
                System.out.println("i =" +i);  
            }  
            System.out.println("Donguden cikti");  
            System.out.println("j =" +j);  
        }  
    }  
}
```

Bu uygulamada, *pas* etiketini kullanılarak `continue` işleminin en dışdaki döngüsel sistemden tekrardan başlamasını (ama kaldığı yerden) sağlandı... Uygulamanın sonucu aşağıdaki gibi olur:

```
i =0  
i =1  
i =2  
i =0  
i =1
```



```
i =2  
i =0  
i =1  
i =2  
i =0  
i =1  
i =2  
i =0  
i =1  
i =2  
i =0  
i =1  
i =2
```

**i** değişkeninin her seferinde yeniden 0'dan başladığını ve 2 de kesildiğini görmekteyiz. Bu işlem toplam 6 kez olmuştur. Yani en dışdaki döngünün sınırları kadar.

### ▪ return Deyimi

return deyimi daha önceden bahsedilmişti; ancak yeni bir anımsatma yararlı olacaktır. return deyiminin 2 tür kullanım şekli vardır: Birincisi değer döndürmek için -ki yordamlardan üretilen değerleri böyle geri döndürürüz, ikincisi ise eğer yordamın dönüş tipi buna izin vermiyorsa (*void* ise) herhangi bir taviz vermeden return yazıp ilgili yordamı terk edebiliriz:

**Örnek:** *ReturnTest.java*

```
public class ReturnTest {  
  
    public double toplamaYap(double a, double b) {  
  
        double sonuc = a + b ;  
        return sonuc ;    // normal return kullanımı  
    }  
  
    public void biseyYapma(double a) {  
  
        if (a == 0) {  
            return ; // yordamı acilen terk et  
        } else {  
            System.out.println("-->" + a);  
        }  
    }  
}
```

## BÖLÜM 3

### NESNELERİN BAŞLANGIÇ DURUMU VE TEMİZLİK

Bir nesnenin başlangıç durumuna getirilme işlemini bir sanatçının sahneye çıkmadan evvelki yaptığı son hazırlıklar gibi düşünebilir... Oluşturulacak olan her nesne kullanıma sunulmadan önce bazı bilgilere ihtiyaç duyabilir veya bazı işlemlerin yapılmasına gereksinim duyabilir. Uygulama programlarının çalışması sırasında oluşan hataların önemli nedenlerden birisi de, nesnelerin yanlış biçimde başlangıç durumlarına getirilmesidir; diğeri ise, temizlik işleminin doğru dürüst yapılmamasıdır. Tasarımcı, bilmediği kütüphanelere ait nesneleri yanlış başlangıç durumuna getirmesinden ötürü çok sıkıntılar yaşayabilir. Diğer bir husus ise temizliktir. Temizlik işleminin doğru yapılmaması durumunda, daha önceden oluşturulmuş ve artık kullanılmayan nesneler sistem kaynaklarında gereksiz yer kaplarlar; ve bunun sonucunda ciddi problemler oluşabilir. Bu bölümde nesnelere başlangıç değerleri verilmesi ve temizleme sürecinin Java programlama dilinde nasıl yapıldığı ele alınmıştır.

#### 3.1. Başlangıç Durumuna Getirme İşlemi ve Yapılandırıcılar (*Initialization and Constructor*)

Başlangıç durumuna getirme işlemlerin gerçekleştiği yer bir çeşit yordam (*method*) diyebileceğimiz yapılandırıcılardır (*constructor*); ancak, yapılandırıcılar normal yordamlardan çok farklıdır. Şimdi biraz düşünelim... Elimizde öyle bir yapılandırıcı olacak ki, biz bunun içerisinde nesnenin kullanılmasından önce gereken işlemleri yapacağız; yani, nesneyi başlangıç durumuna getireceğiz. Ayrıca, Java bu yapılandırıcıyı, ilgili nesneyi oluşturmadan hemen önce otomatik olarak çağırabilecek...

Diğer bir problem ise yapılandırıcının ismidir; bu isim öyle olmalıdır ki, diğer yordam (*method*) isimleriyle çakışmamalıdır. Ayrıca Java'nın bu yapılandırıcıyı otomatik olarak çağıracağı düşünülürse, isminin Java tarafından daha önce biliniyor olması gerekir.

Bu problemlere ilk çözüm C++ dilinde bulunmuştur. Çözüm, yapılandırıcıyla sınıf isimlerinin birebir aynı olmasıdır (büyük küçük harf dahil). Böylece Java sınıfın yapılandırıcısını bularak, buna ait bir nesne oluşturmak için ilk adımı atabilecektir. Küçük bir uygulama üzerinde açıklanmaya çalışırsa, bir sonraki sayfadaki örnek verilebilir:

#### **Örnek-3.1:** *YapilandirciBasitOrnek.java*

```
class KahveFincani {  
  
    public KahveFincani() { //yapılandırıcı kısmı  
        System.out.println("KahveFincani...");  
    }  
}  
  
public class YapilandirciBasitOrnek {  
    public static void main(String[] args) {  
        for(int i = 0; i < 5; i++)  
            new KahveFincani();  
    }  
}
```

Yukarıda verilen örnekte art arda 5 adet *KahveFincani* nesnesi oluşturuluyor; dikkat edilirse, nesneler oluşturulmadan önce (ellerimizi kahve fincanlarının üzerine götürmeden) ekrana kendilerini tanıtan ifadeler yazdılar; yani, nesnelere ilk değerleri verilmiş oldu...

Dikkat edilmesi gereken ikinci unsur, yapılandırıcıya (*constructor*) verilen isimdir; bu içinde bulunduğu sınıf ismi ile birebir aynıdır. Anımsarsanız, normalde yordam isimleri bir fiil cümlesi içermeliydi (`dosyaAc()`, `openFile()`, `dosyaOku()`, `readFile()`, `dosyaYaz()`, `arabaSur()` vb.); ancak, yapılandırıcılar bu kuralın da dışındadır. Yukarıda verilen uygulamanın sonucu aşağıdaki gibi olur:

```
KahveFincani...
KahveFincani...
KahveFincani...
KahveFincani...
KahveFincani...
```

Yapılandırıcılar, yordamlar (*methods*) gibi parametre alabilirler:

**Örnek:** *YapilandirciBasitOrnekVersiyon2.java*

```
class YeniKahveFincani {
    public YeniKahveFincani(int adet) {
        System.out.println(adet + " adet YeniKahveFincani");
    }
}

public class YapilandirciBasitOrnekVersiyon2 {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++)
            new YeniKahveFincani( i );
    }
}
```

Gönderilen parametre sayesinde nesnenin nasıl oluşacağı belirtebilmektedir. Bu örnekte olduğu gibi *YeniKahveFincani* nesnesi oluşturulurken kaç adet olacağı söylenebiliyor. Uygulamanın sonucu aşağıdaki gibi olur:

```
0 adet YeniKahveFincani
1 adet YeniKahveFincani
2 adet YeniKahveFincani
3 adet YeniKahveFincani
4 adet YeniKahveFincani
```

Yapılandırıcılar, yordamlardaki gibi değer döndürme mekanizmasına sahip değildirler; herhangi bir şekilde değer döndüremezler. Bu değer döndürülemez ibaresi yordamlardaki **void** ifadesine karşılık gelmemektedir. Yapılandırıcılardan çıkılmak isteniyorsa **return** kullanılabilir.

### 3.1.1. Bir İsmi Birden Çok Yordam İçin Kullanılması

#### - Adaş Yordamlar (*Overloaded Methods*)

İyi bir uygulama yazılması her zaman için iyi bir takım çalışması gerektirir; takım çalışmasının önemli kurallarından birisi de birinin yazdığı kodu diğer kişilerin de kolaylıkla anlayabilmesinden geçer. Uygulamalardaki yordam isimlerinin, yordam içerisinde yapılan işlemlerle uyum göstermesi önemlidir. Bu sayede bir başka kişi sadece yordamın ismine bakarak, bu yordam içerisinde oluşan olayları anlayabilme şansına sahip olabilir. Örneğin elimizde bulunan müzik, resim ve metin (*text*) formatındaki dosyaları açmak için yordamlar yazılmak istenirse, bunların isimlerinin ne olması gerekir? Müzik dosyasını açan yordamın ismi `muzikDosyasiAc()`, resim dosyası için `resimDosyasiAc()`, metin dosyasını açmak için ise

`textDosyasınıAc()` gibi üç ayrı yordam ismi kullanılması ne kadar akıllıca ve pratik olur? Sonuçta işlem sadece dosya açmaktır; dosyanın türü sadece bir ayrıntıdır. Bir ismin birçok yordam için kullanılması (**method overloading**) bize bu imkanı verebilmektedir. Aynı tür işleve sahip olan yordamların aynı isimlere sahip olabılme özelliği, bizi isim bulma sıkıntısından da kurtarmaktadır.

### **Örnek-3.2:** *YordamOverloadingDemo1.java*

```
class MuzikDosyasi {
    String m_tur = "Muzik Dosyasi" ;
}

class ResimDosyasi {
    String r_tur = "Resim Dosyasi" ;
}

class TextDosyasi {
    String t_tur = "Text Dosyasi" ;
}

public class YordamOverloadingDemo1 {
    public void dosyaAc(MuzikDosyasi md) {
        System.out.println( "Tur =" + md.m_tur );
    }

    public void dosyaAc(ResimDosyasi rd) {
        System.out.println( "Tur =" + rd.r_tur );
    }

    public void dosyaAc(TextDosyasi td) {
        System.out.println( "Tur =" + td.t_tur );
    }

    public static void main(String[] args) {
        YordamOverloadingDemo1 mod1 = new YordamOverloadingDemo1();
        MuzikDosyasi md = new MuzikDosyasi();
        ResimDosyasi rd = new ResimDosyasi();
        TextDosyasi td = new TextDosyasi();
        mod1.dosyaAc(md);
        mod1.dosyaAc(rd);
        mod1.dosyaAc(td);
    }
}
```

Uygulamamızın sonucu aşağıdaki gibi olur:

```
Tur =Muzik Dosyasi
Tur =Resim Dosyasi
Tur =Text Dosyasi
```

Yukarıdaki örnekte görüldüğü gibi aynı tür işlemleri yapan yordamların isimleri aynıdır. Peki, Java aynı isimde olan bu üç yordamı birbirinden nasıl ayırt etmektedir?

### **● Adaş Yordamlar Nasıl Ayırt Edilirler?**

Java aynı isme sahip olan yordamları nasıl ayırt edebilmektedir? Cevap olarak parametrelerine göre denilebilir. Konu biraz daha açılırsa, her yordamın kendisine ait özel ve tek parametresi veya parametre listesi olmak zorundadır.

#### **Örnek-3.4:** *YordamOverloadingDemo2.java*

```
public class YordamOverloadingDemo2 {
    public int toplamaYap(int a , int b){
        int sonuc = a + b ;
        System.out.println("sonuc - 1 = " + sonuc);
        return sonuc ;
    }

    public void toplamaYap(int a , double b){
        double sonuc = a + b ;
        System.out.println("sonuc - 2 = " + sonuc);
    }

    public double toplamaYap(double a , int b){
        double sonuc = a + b ;
        System.out.println("sonuc - 3= " + sonuc);
        return sonuc ;
    }

    public static void main(String[] args) {
        YordamOverloadingDemo2 mod2 = new YordamOverloadingDemo2();
        mod2.toplamaYap(3,4);
        mod2.toplamaYap(3,5.5);
        mod2.toplamaYap(6.8,4);
    }
}
```

Bu örnekte üç adet toplamaYap() yordamının parametreleri birbirinden bir şekilde farklıdır: toplamaYap() yordamının ilki, 2 adet temel **int** tipinde parametre olarak diğer adaş yordamlarından ayrılmaktadır; geriye kalan 2 adet toplamaYap() yordamı ise aynı tip parametreler almaktadır. Bunlar temel double tipi ve int tipi, bu iki yordamı birbirinden farklı kılan, parametrelerin sırasıdır. Uygulamanın çıktısı aşağıdaki gibidir:

```
sonuc - 1 = 7
sonuc - 2 = 8.5
sonuc - 3= 10.8
```

#### **Ø Dönüş Değerlerine Göre Adaş Yordamlar Ayırt Edilebilir mi ?**

Akıllara şöyle bir soru gelebilir: "Adaş yordamlar dönüş tiplerine göre ayırt edilebilir mi?" İnceleyelim:

#### **Gösterim-3.1:**

```
void toplamaYap();
double toplamaYap();
```

Elimizde 2 adet aynı isimde ve aynı işlemi yapan, fakat biri değer döndürmeyen (**void**) diğeri ise **double** tipinde değer döndüren yordamların olduğunu varsayalım.

### **Gösterim-3.2:**

```
double y = toplamayap();
```

Gösterim-3.2 için, Java bu yordamlardan hangisini seçeceğini tahmin edebilir; double toplamaYap() ... Peki aşağıdaki gibi bir durum için nasıl bir yol izlenmesi gerekir?

### **Gösterim-3.3:**

```
toplamayap() ;
```

Değer döndüren bir yordamı döndürdüğü tipe karşılık gelen değişkene atama zorunluluğu olmadığı hatırlatalım. Kısacası bu koşulda Java hangi yordamı çağıracağını bilemeyecektir. Bu nedenle, Java dilinde dönüş tiplerine göre yordamların ayırt edilmesi kabul görmez; ayırt edilmesini sağlayan tek şey parametrelerindeki farklılıktır.

### **3.1.2. Varsayılan Yapılandırıcılar (Default Constructors)**

Eğer uygulamaya herhangi bir yapılandırıcı koyulmazsa, Java bu işlemi kendiliğinden yapmaktadır. Varsayılan yapılandırıcılar aynı zamanda parametresiz yapılandırıcılar (*default constructor* veya "*no-args*" *constructor*) olarak ta anılmaktadır; bunları içi boş yordamlar olarak düşünebilirsiniz.

**Örnek-3.5:** *VarsayilanYapilandirici.java*

```
class Kedi {
    int i;
}

public class VarsayilanYapilandirici {
    public static void main(String[] args) {
        Kedi kd = new Kedi(); //Varsayılan
        yapılandırıcı çağrıldı
    }
}
```

Java'nın yerleştirmiş olduğu varsayılan yapılandırıcı açık bir şekilde gözükmemektedir. Açık şekilde görmek istenirse;

**Örnek-3.6:** *VarsayilanYapilandirici.java (değişik bir versiyon)*

```
class Kedi {
    int i;
    /* varsayılan yapılandırıcı bu yapılandırıcıyı eğer biz koymasaydık
    Java bizim yerimize zaten koyardı */
    public Kedi() {}
}

public class VarsayilanYapilandirici {
    public static void main(String[] args) {
        Kedi kd = new Kedi(); //Varsayılan yapılandırıcı
        çağrıldı
    }
}
```

### **● Büyünün Bozulması**

Eğer, yapılandırıcı kodu yazan kişi tarafından konulursa, Java varsayılan yapılandırıcı desteğini çekecektir. Bunun nedeni şöyledir: Eğer bir sınıfa ait herhangi bir yapılandırıcı belirtilmezse, Java devreye girip kendiliğinden varsayılan bir yapılandırıcı koyar; eğer, biz kendimize ait özel yapılandırıcılar tanımlarsak, şöyle demiş oluruz: "Ben ne yaptığımı biliyorum, lütfen karışma". Bu durumda olası tüm yapılandırıcılar bizim tarafımızdan yazılması gerekir. Şöyle ki:

### **Örnek-3.7:** *VarsayılanYapilandiriciVersiyon2.java*

```
class Araba {
    int kapi_sayisi;
    int vites_sayisi ;

    public Araba(int adet) {
        kapi_sayisi = adet ;
    }

    public Araba(int adet, int sayi) {
        kapi_sayisi = adet ;
        vites_sayisi = sayi ;
    }
}

public class VarsayılanYapilandiriciVersiyon2 {
    public static void main(String[] args) {
        Araba ar = new Araba();    // ! Hata var! Bu satır anlamlı
        değil; yapılandırıcısı yok
        Araba ar1 = new Araba(2);
        Araba ar2 = new Araba(4,5);
    }
}
```

Artık *Araba* sınıfına ait bir nesne oluşturmak için parametresiz yapılandırıcıyı (*default constructor*) çağırabiliriz; böylesi bir nesne oluşturmak isteniyorsa, artık, bu işlem için bir *Araba* sınıfına ait iki yapılandırıcıdan birisini seçip çağırmanız gerekir.

### 3.1.3. **this** Anahtar Sözcüğü

**this** anahtar sözcüğü, içinde bulunulan nesneye ait bir referans döndürür; bunun sayesinde nesnelere ait global alanlara erişme fırsatı bulunur. Şöyle ki:

### **Örnek-3.8:** *TarihHesaplama.java*

```
public class TarihHesaplama {
    int gun, ay, yil;
    public void gunEkle(int gun) {
        this.gun += gun ;
    }
    public void gunuEkranaBas() {
        System.out.println("Gun = " + gun);
    }
    public static void main(String[] args) {
        TarihHesaplama th = new TarihHesaplama();
        th.gunEkle(2); th.gunEkle(3);
        th.gunuEkranaBas();
    }
}
```

`gunEkle()` yordamı sayesinde parametre olarak gönderdiğimiz değer, global olan temel `int` tipindeki `gun` alanının değerini arttırmaktadır.

Nesnelere ait global alanlar, içinde bulundukları nesnelere ait alanlardır ve nesne içerisindeki her statik olmayan yordam tarafından doğrudan erişilebilirler. Yerel değişkenler ise yordamların içerisinde tanımlanırlar; ve, ancak tanımlandığı yordam içerisinde geçerlidir.

`gunEkle()` yordamına dikkat edilirse, `gun` ismini hem *TarihHesaplama* nesnesine ait global bir alanının adı olarak hem de yerel değişken adı olarak kullanıldığı görülür. Burada herhangi bir yanlışlık yoktur. Çünkü bunlardan biri nesneye ait bir alan, diğeri ise `gunEkle()` yordamına ait yerel değişkendir. Bizim gönderdiğimiz değer, `gunEkle()` yordamının yerel değişkeni sayesinde *TarihHesaplama* nesnesinin global olan alanına eklenmektedir. En sonunda `gunuEkranaBas()` yordamı ile global olan `gun` alanının değerini görebilmekteyiz.

Özet olarak, `gunEkle()` yordamının içerisinde kullandığımız `this.gun` ifadesiyle *TarihHesaplama* nesnesinin global olan `gun` alanına erişebilmekteyiz. Uygulamanın sonucu aşağıdaki gibi olur:

Gun = 5

Peki, `gunEkle()` yordamının içerisinde `this.gun` ifadesi yerine sadece `gun` ifadesi kullanılsaydı sonuç nasıl değişirdi?

### **Örnek-3.9:** *TarihHesaplama2.java*

```
public class TarihHesaplama2 {
    int gun, ay, yil;

    public void gunEkle(int gun) {
        gun += gun ;
    }

    public void gunuEkranaBas() {
        System.out.println("Gun = " + gun);
    }

    public static void main(String[] args) {
        TarihHesaplama2 th = new TarihHesaplama2();
        th.gunEkle(2);
        th.gunEkle(3);
        th.gunuEkranaBas();
    }
}
```

Uygulamanın çıktısı aşağıdaki gibi olur:.

Gun = 0

*TarihHesaplama* nesnesine ait olan global **gun** alanına herhangi bir değer ulaşmadığı için sonuç sıfır olacaktır.

### **● Yordam Çağrılarında *this* Kullanımı**

#### **Gösterim-3.4:**

```
class Uzum {
```



```

void sec() { /* ... */ }
...
void cekirdeginiCikar() { sec(); /* ... */ }

}

```

Bir yordamın içerisinde diğer yordamı çağırmak gayet basit ve açıktır ama sahne arkasında derleyici, çağrılan bu yordamın önüne **this** anahtar kelimesini gizlice yerleştirir: yani, fazladan **this.sec()** denilmesinin fazla bir anlamı yoktur.

Aşağıdaki örnek **this** anahtar kelimesinin, içinde bulunduğu nesneye ait nasıl bir referansın alındığını çok net bir biçimde göstermektedir.

### **Örnek-3.10:** *Yumurta.java*

```

public class Yumurta {

    int toplam_yumurta_sayisi = 0;

    Yumurta sepeteKoy() {
        toplam_yumurta_sayisi++;
        return this;
    }

    void goster() {
        System.out.println("toplam_yumurta_sayisi = "
                           + toplam_yumurta_sayisi);
    }

    public static void main(String[] args) {
        Yumurta y = new Yumurta();
        y.sepeteKoy().sepeteKoy().sepeteKoy().goster();
    }
}

```

sepeteKoy() yordamı *Yumurta* sınıfı tipinde değer geri döndürmektedir. `return this` diyerek, oluşturulmuş olan *Yumurta* nesnenin kendisine ait bir referans geri döndürülmektedir. `sepeteKoy()` yordamı her çağrıldığında *Yumurta* nesnesine ait, `toplam_yumurta_sayisi` global alanın değeri bir artmaktadır. Burada dikkat edilmesi gereken husus, `this` anahtar kelimesi ancak nesnelere ait olan yordamlar içinde kullanılabilir. Nesnelere ait olan yordamlar ve statik yordamlar biraz sonra detaylı bir şekilde incelenecektir. Uygulama sonucu aşağıdaki gibi olur:

```

toplam_yumurta_sayisi = 3

```

### ***Bir Yapılandırıcıdan Diğer Bir Yapılandırıcıyı Çağırarak***

Bir yapılandırıcıdan diğerini çağırmak `this` anahtar kelimesi ile mümkündür.

### **Örnek-3.11:** *Tost.java*

```

public class Tost {
    int sayi ;
    String malzeme ;

    Public Tost() {
        this(5);
        // this(5,"sucuklu");      !Hata!-iki this kullanılamaz
    }
}

```

```

        System.out.println("parametresiz yapilandirici");
    }

    public Tost(int sayi) {
        this(sayi,"Sucuklu");
        this.sayi = sayi ;
        System.out.println("Tost(int sayi) " );
    }

    public Tost(int sayi ,String malzeme) {
        this.sayi = sayi ;
        this.malzeme = malzeme ;
        System.out.println("Tost(int sayi ,String malzeme) " );
    }

    public void siparisGoster() {
        // this(5,"Kasarli");           !Hata!-sadece yapilandiricılarda kullanılır
        System.out.println("Tost sayisi="+sayi+ "malzeme =" + malzeme );
    }

    public static void main(String[] args) {
        Tost t = new Tost();
        t.siparisGoster();
    }
}

```

- Bir yapilandiricidan `this` ifadesi ile diğ er bir yapilandiriciy i çağ ırırken dikkat edilmesi gereken kurallar aşağıdaki gibidir:
- Yapilandiricılar içerisinde `this` ifadesi ile her zaman baş ka bir yapilandirici çağ ırılabilir.
- Yapilandirici içerisinde, diğ er bir yapilandiriciy i çağ ırırken `this` ifadesi her zaman ilk satırda yazılmalıdır.
- Yapilandiricılar içerisinde birden fazla `this` ifadesi ile baş ka yapilandirici çağ ırılmaz.

Uygulama sonucu aşağıdaki gibi olur:

```

Tost(int sayi,String malzeme)
Tost(int sayi)
parametresiz yapilandirici
Tost sayisi =5 malzeme =Sucuklu

```

### 3.1.4. Statik Alanlar (Sınıflara Ait Alanlar)

Sadece global olan alanlara statik özelliğ i verilebilir. Yerel değ iş kenlerin statik olma özellikleri yoktur. Global alanları tür olarak iki çeş ide ayırabiliriz: statik olan global alanlar ve nesnelere ait global alanlar. Statik alanlar, bir sınıfa ait olan alanlardır ve bu sınıfa ait tüm nesneler için ortak bir bellek alanında bulunurlar, ayrıca statik alanlara sadece bir kez ilk değ erleri atanır.

#### **Örnek-3.12:** *StatikDegisken.java*

```

public class StatikDegisken {
    public static int x ;
    public int y ;

    public static void ekranaBas(StatikDegisken sd ) {

```

```

        System.out.println("StatikDegisken.x = " + sd.x +
                           " StatikDegisken.y = " + sd.y );
    }

    public static void main(String args[]) {
        StatikDegisken sd1 = new StatikDegisken();
        StatikDegisken sd2 = new StatikDegisken();
        x = 10 ;
        //sd1.x = 10 ; // x = 10 ile ayni etkiyi yapar
        //sd2.x = 10 ; // x = 10 ile ayni etkiyi yapar
        sd1.y = 2 ;
        sd2.y = 8;
        ekranaBas (sd1);
        ekranaBas (sd2);
    }
}

```

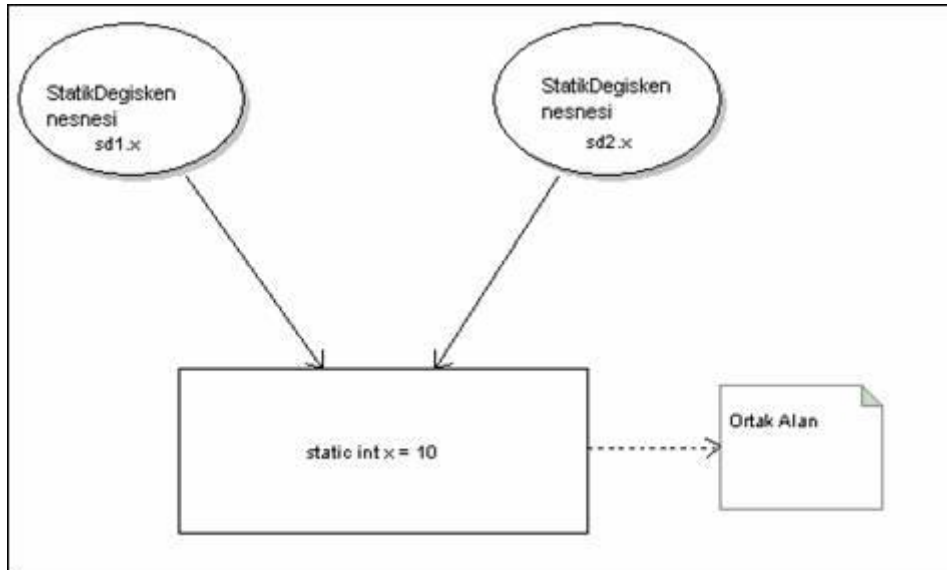
Bu uygulamada *StatikDegisken* sınıfına ait iki adet nesne oluřturulmaktadır; daha sonradan *StatikDegisken* sınıfının statik olan **x** alanına 10 deęeri atanmaktadır. Artık oluřturulacak olan tüm *StatikDegisken* tipindeki nesneler için bu **x** deęeri ortaktır. Yalnız dikkat edilecek olursa *StatikDegisken* nesnelerine ait olan global **y** alanı her bir *StatikDegisken* nesnesi için farklıdır. Uygulamanın çıktıısı ařağıdaki gibidir.

```

StatikDegisken.x = 10 StatikDegisken.y = 2
StatikDegisken.x = 10 StatikDegisken.y = 8

```

Statik alanların etkisi řekil-3.1’de verilen çizimden incelenebilir:



**řekil-3.1. Statik alanlar**

### 3.1.5. Statik Yordamlar (*Static Methods*)

Statik yordamlar (sınıf yordamları) nesnelerden bağımsız yordamlardır. Statik bir yordamı çağırarak için herhangi bir nesne oluřturulmak zorunda deęildir. Statik olmayan yordamlardan (nesneye ait yordamlar) statik yordamları rahatlıkla çağırabilmesine karřın, statik yordamlardan nesne yordamlarını doęrudan çağırılmaz.

**Örnek-3.13:** *StatikTest.java*

```

// StatikTest.java

```

```

public class StatikTest {

    public static void hesapla(int a , int b) {
        /*static yordam doğrudan nesneye ait bir yordamı çağırılmaz */
        // islemYap(a,b);        // !Hata!
    }

    public void islemYap(int a , int b) {
        /*doğru , nesneye ait bir yordam, static bir yordamı çağırabilir*/

        hesapla(a,b);
    }
}

```

### 3.1.6. Bir Yordamın Statik mi Yoksa Nesne Yordamı mı Olacağı Neye Göre Karar Verilecek?

Bir yordamı statik olarak tanımlamak için, “Bu yordamı çağırmak için, bu yordamın içerisinde bulunduğu sınıfa ait bir nesne oluşturmaya gerek var mı?” sorusuna yanıt vermemiz gerekir.

Acaba bir yordamı statik olarak mı tanımlasak? Yoksa nesne yordamı olarak mı tanımlasak? Bu sorunun cevabı, sınıf yordamları ile nesne yordamları arasındaki farkı iyi bilmekte gizlidir. Bu farkı anlamak için aşağıdaki uygulamamızı inceleyelim.

#### **Örnek-3.14:** *MutluAdam.java*

```

public class MutluAdam {

    private String ruh_hali = "Mutluyum" ;

    public void ruhHaliniYansit() {
        System.out.println( "Ben " + ruh_hali );
    }

    public void tokatAt() {
        if( ruh_hali.equals("Mutluyum" ) ) {
            ruh_hali = "Sinirlendim";
        }
    }

    public void kucakla() {
        if( ruh_hali.equals( "Sinirlendim" ) ) {
            ruh_hali = "Mutluyum";
        }
    }

    public static void main(String[] args) {
        MutluAdam obj1 = new MutluAdam();
        MutluAdam obj2 = new MutluAdam();

        obj1.ruhHaliniYansit();
        obj2.ruhHaliniYansit();

        obj1.kucakla();
        obj2.tokatAt();

        obj1.ruhHaliniYansit();
        obj2.ruhHaliniYansit();
    }
}

```

ruhHaliniYansit(), tokatAt(), kucakla()’nın hepsi birer nesne yordamlarıdır; yani, bu yordamları çağırmak için *MutluAdam* sınıfına ait bir nesne oluşturmalıdır.

*MutluAdam* tipindeki obj1 ve obj2 referanslarına *MutluAdam* nesnelerini bağladığı anda bu iki nesnenin ruh halleri aynıdır (ruh\_hali = "Mutluyum"). Fakat zaman geçtikçe her nesnenin kendisine ait ruh hali değişmektedir. Bu değişimin sebebi çevre koşulları olabilir. Örneğin obj1 referansına bağlı olan *MutluAdam* nesnesini kucaklandık, böylece mutlu olmaya devam etti ama obj2 referansına bağlı *MutluAdam* nesnesine tokat attığımızda, mutlu olan ruh hali değişti ve sinirli bir ruh haline sahip oldu.

Nesne yordamları, nesnenin durumuna ilişkin işlemleri yapmak için kullanılırlar ama statik yordamlar ilgili nesnenin durumuna ilişkin genel bir işlem yapmazlar; Şöyle ki:

**Örnek-3.15:** *Toplama.java*

```
public class Toplama {  
  
    public static double toplama(double a , double b ) {  
        double sonuc = a + b ;  
        return sonuc ;  
    }  
}
```

Bu örneğimizde görüldüğü üzere toplama() yordamının amacı sadece kendisine gelen iki double değerini toplamak ve sonucu geri döndürmektir. toplama() yordamı *Toplama* sınıfına ait bu nesnenin durumu ile ilgili herhangi bir görev üstlenmediği için statik yordam (sınıf yordamı) olarak tanımlanması gayet mantıklıdır. Ayrıca toplama() yordamını çağırmak için *Toplama* sınıfına ait bir nesne oluşturmak da çok gereksiz durmaktadır. Şimdi *Toplama* sınıfı içerisindeki statik olan toplama() yordamını kullanan bir uygulama yazalım.

**Örnek-3.16:** *ToplamaIslemi.java*

```
public class ToplamaIslemi {  
  
    public static void main(String args[]) {  
  
        if (args.length < 2) {  
            System.out.println("Ltf iki adet sayi  
giriniz");  
            System.exit(-1);          // uygulama sonlanacaktır  
        }  
  
        double a = Double.parseDouble(args[0]);  
        double b = Double.parseDouble(args[1]);  
  
        double sonuc = Toplama.toplama(a,b);          // dikkat  
        System.out.println("Sonuc : " + sonuc );  
    }  
}
```

*ToplamaIslemi* sınıfı, kullanıcıdan aldığı parametreleri öncelikle double tipine dönüştürmektedir. Daha sonra bu değerleri *Toplama* sınıfının toplama() yordamına göndererek, toplatmaktadır. Dikkat edileceği üzere toplama() yordamını çağırmak için *Toplama* sınıfına ait bir nesne oluşturma zahmetinde bulunmadık. Yukarıdaki uygulamamızı aşağıdaki gibi çalıştıracak olursak,

```
> java ToplamaIslemi 5.5 9.2  
  
14.7
```

Yukarıda uygulamaya gönderilen iki parametre toplanıp sonucu ekrana yazdırılmıştır.

Sonuç olarak eğer bir yordam, nesnenin durumuna ilişkin bir misyon yükleniyorsa, o yordamı nesne yordamı olarak tanımlamamız en uygun olanıdır. Yani ilgili sınıfa ait bir nesne oluşturulmadan, nesneye ait bir yordam (statik olmayan) çağrılmaz. Ama eğer bir yordam sadece atomik işlemler için kullanılacaksa (örneğin kendisine gelen *String* bir ifadenin hepsini büyük harfe çeviren yordam gibi) ve nesnenin durumuna ilişkin bir misyon yüklenmemiş ise o yordamı rahatlıkla statik yordam (sınıf yordamı) olarak tanımlayabiliriz. Dikkat edilmesi gereken bir başka husus ise uygulamalarınızda çok fazla statik yordam kullanılmasının tasarımsal açıdan yanlış olduğudur. Böyle bir durumda, stratejinizi baştan bir kez daha gözden geçirmenizi öneririm.

### 3.2. Temizlik İşlemleri: `finalize()` ve Çöp Toplayıcı (*Garbage Collector*)

Yapılandırıcılar sayesinde nesnelerimizi oluşturmadan önce, başlangıç durumlarının nasıl verildiğine değinildi; Peki, oluşturulan bu nesneler daha sonradan nasıl bellekten silinmektedir? Java programlama dilinde, C++ programla dilinde olduğu gibi oluşturulan nesneleri, işleri bitince yok etme özgürlüğü kodu yazan kişinin elinde değildir. Java programlama dilinde, bir nesnenin gerçekten çöp olup olmadığına karar veren mekanizma çöp toplayıcısıdır.

Çalışmakta olan uygulamanın içerisinde bulunan bir nesne artık kullanılmıyorsa, bu nesne çöp toplayıcısı tarafından bellekten silinir. Çöp toplama sistemi, kodu yazan kişi için büyük bir rahatlık oluşturmaktadır çünkü uygulamalardaki en büyük hataların ana kaynağı temizliğin (oluşturulan nesnelerin, işleri bitince bellekten silinmemeleri) doğru dürüst yapılamamasıdır.

#### 3.2.1. `finalize()` Yordamı

Çöp toplayıcı (*garbage collector*) bir nesneyi bellekten silmeden hemen önce o nesnenin `finalize()` yordamını çağırır. Böylece bellekten silinecek olan nesnenin yapması gereken son işlemler var ise bu işlemler `finalize()` yordamı içerisinde yapılır. Bir örnek ile açıklanırsa, örneğin bir çizim programı yaptık ve elimizde ekrana çizgi çizen bir nesnemiz olduğunu düşünelim. Bu nesnemiz belli bir süre sonra gereksiz hale geldiğinde, çöp toplayıcısı tarafından bellekten silinecektir. Yalnız bu nesnemizin çizdiği çizgilerin hala ekranda olduğunu varsayarsak, bu nesne çöp toplayıcısı tarafından bellekten silinmeden evvel, ekrana çizdiği çizgileri temizlemesini `finalize()` yordamında sağlayabiliriz.

Akıllarda tutulması gereken diğer bir konu ise eğer uygulamanız çok fazla sayıda çöp nesnesi (kullanılmayan nesne) üretmiyorsa, çöp toplayıcısı (*garbage collector*) devreye girmeyebilir. Bir başka nokta ise eğer `System.gc()` ile çöp toplayıcısını tetiklemezsek, çöp toplayıcısının ne zaman devreye girip çöp haline dönüşmüş olan nesneleri bellekten temizleyeceği bilinemez.

**Örnek-3.17:** *Temizle.java*

```
class Elma {  
  
    int i = 0 ;  
    Elma(int y) {  
        this.i = y ;  
        System.out.println("Elma Nesnesi Olusturuluyor = " + i );  
    }  
  
    public void finalize() {  
        System.out.println("Elma Nesnesi Yok Ediliyor = "+ i );  
    }  
}  
  
public class Temizle {  
  
    public static void main(String args[]) {  
        for (int y=0 ; y<5 ;y++) {  
            Elma e = new Elma(y);  
        }  
    }  
}
```

```

        for (int y=5 ; y<11 ;y++) {
            Elma e = new Elma(y);
        }
    }
}

```

*Temizle.java* örneğinde iki döngü içerisinde toplam 11 adet *Elma* nesnesi oluşturulmaktadır. Kendi bilgisayarımda 256 MB RAM bulunmaktadır; böyle bir koşulda çöp toplayıcısı devreye girmeyecektir. Nedeni ise, *Elma* nesnelerinin Java için ayrılan bellekte yeterince yer kaplamamasındandır. Değişik bilgisayar konfigürasyonlarına, sahip sistemlerde, çöp toplayıcısı belki devreye girebilir! Uygulamanın çıktısı aşağıdaki gibi olur:

```

Elma Nesnesi Olusturuluyor = 0
Elma Nesnesi Olusturuluyor = 1
Elma Nesnesi Olusturuluyor = 2
Elma Nesnesi Olusturuluyor = 3
Elma Nesnesi Olusturuluyor = 4
Elma Nesnesi Olusturuluyor = 5
Elma Nesnesi Olusturuluyor = 6
Elma Nesnesi Olusturuluyor = 7
Elma Nesnesi Olusturuluyor = 8
Elma Nesnesi Olusturuluyor = 9
Elma Nesnesi Olusturuluyor = 10

```

Uygulama sonucundan anlaşılacağı üzere, *Elma* nesnelerinin *finalize()* yordamı hiç çağrılmadı. Nedeni ise, çöp toplayıcısının hiç tetiklenmemiş olmasıdır. Aynı örneği biraz değiştirelim (bkz.. Örnek-3.18):

### **Örnek-3.18:** *Temizle2.java*

```

class Elma2 {

    int i = 0 ;
    Elma2(int y) {
        this.i = y ;
        System.out.println("Elma2 Nesnesi Olusturuluyor =
" + i );
    }

    public void finalize() {
        System.out.println("Elma2 Nesnesi Yok Ediliyor =
"+ i );
    }
}

public class Temizle2 {
    public static void main(String args[]) {
        for (int y=0 ; y<10 ;y++) {
            Elma2 e = new Elma2(y);
        }
        System.gc() ;           // çöp toplayıcısını çağırdık
        for (int y=10 ; y<21 ;y++) {
            Elma2 e = new Elma2(y);
        }
    }
}

```

```
}
```

*System* sınıfının statik bir yordamı olan **gc()**, çöp toplayıcısının kodu yazan kişi tarafından tetiklenmesini sağlar. Böylece çöp toplayıcısı, çöp haline gelmiş olan nesneleri (kullanılmayan nesneleri) bularak (eğer varsa) bellekten siler. Yukarıdaki uygulamamızı açıklamaya başlarsak; İlk **for** döngüsünde oluşturulan *Elma2* nesneleri, döngü bittiğinde çöp halini alacaklardır. Bunun sebebi, ilk **for** döngüsünün bitimiyle, oluşturulan 10 adet *Elma2* nesnesinin erişilemez bir duruma geleceğidir. Doğal olarak, erişilemeyen bu nesneler, çöp toplayıcısının iştahını kabartacaktır. Uygulamanın sonucu aşağıdaki gibi olur;

```
Elma2 Nesnesi Oluşturuluyor = 0
Elma2 Nesnesi Oluşturuluyor = 1
Elma2 Nesnesi Oluşturuluyor = 2
Elma2 Nesnesi Oluşturuluyor = 3
Elma2 Nesnesi Oluşturuluyor = 4
Elma2 Nesnesi Oluşturuluyor = 5
Elma2 Nesnesi Oluşturuluyor = 6
Elma2 Nesnesi Oluşturuluyor = 7
Elma2 Nesnesi Oluşturuluyor = 8
Elma2 Nesnesi Oluşturuluyor = 9
Elma2 Nesnesi Yok Ediliyor = 0
Elma2 Nesnesi Yok Ediliyor = 1
Elma2 Nesnesi Yok Ediliyor = 2
Elma2 Nesnesi Yok Ediliyor = 3
Elma2 Nesnesi Yok Ediliyor = 4
Elma2 Nesnesi Yok Ediliyor = 5
Elma2 Nesnesi Yok Ediliyor = 6
Elma2 Nesnesi Yok Ediliyor = 7
Elma2 Nesnesi Yok Ediliyor = 8
Elma2 Nesnesi Yok Ediliyor = 9
Elma2 Nesnesi Oluşturuluyor = 10
Elma2 Nesnesi Oluşturuluyor = 11
Elma2 Nesnesi Oluşturuluyor = 12
Elma2 Nesnesi Oluşturuluyor = 13
Elma2 Nesnesi Oluşturuluyor = 14
Elma2 Nesnesi Oluşturuluyor = 15
Elma2 Nesnesi Oluşturuluyor = 16
Elma2 Nesnesi Oluşturuluyor = 17
Elma2 Nesnesi Oluşturuluyor = 18
Elma2 Nesnesi Oluşturuluyor = 19
Elma2 Nesnesi Oluşturuluyor = 20
```

System.gc() komutu ile çöp toplayıcısı (garbage collector) tetiklendi.

Elma2 nesneleri bellekten siliniyor çünkü artık onlara gerek yok

**Not:** Sonuç üzerindeki oklar ve diğer gösterimler, dikkat çekmek amacıyla, özel bir resim uygulaması tarafından yapılmıştır.

*System.gc()* komutu ile çöp toplayıcısını tetiklediğimizde, gereksiz olan bu on adet *Elma2* nesnesi bellekten silinecektir. Bu nesneler bellekten silinirken, bu nesnelere ait *finalize()* yordamlarının nasıl çağrıldıklarına dikkat çekmek isterim.

### 3.2.2. Bellekten Hangi Nesneler Silinir?

Çöp toplayıcısı bellekten, bir referansa bağlı olmayan nesneleri siler. Eğer bir nesne, bir veya daha fazla sayıdaki referansa bağlıysa, bu nesnemiz uygulama tarafında kullanılıyordur demektir ve çöp toplayıcısı tarafından bellekten silinmez.

#### **Örnek-3.19:** *CopNesne.java*

```
public class CopNesne {

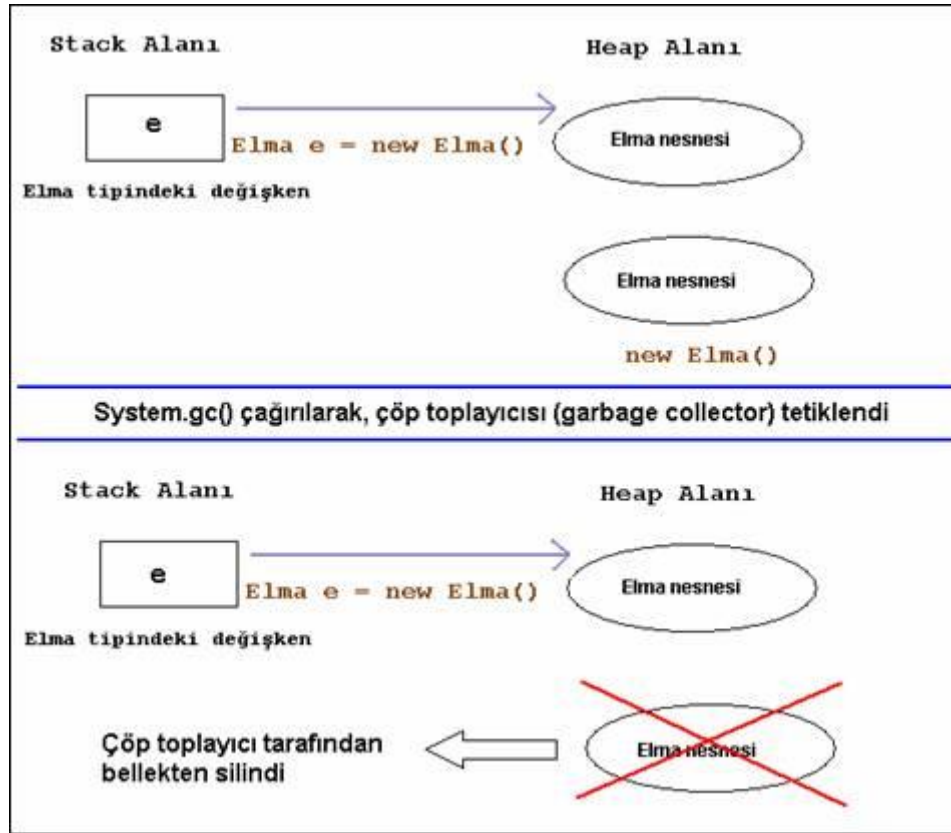
    public static void main(String args[]) {
        Elma e = new Elma(1);
        new Elma(2);
        System.gc(); // çöp toplayıcısını çağırdık
    }
}
```



Verilen örnekte 2 adet *Elma* nesnesinden biri çöp toplayıcısı tarafından bellekten silinirken, diğ er nesne bellekte yaşam anı sürdürmeye devam eder. `System.gc()` yordamının çağ ırılması ile çöp toplayıcımız tetiklenir. *Elma* sınıfı tipindeki **e** referansına bağ lı olan nesnemize çöp toplayıcısı tarafından dokunulmamıştır. Bunun sebebi bu *Elma* nesnesinin, *Elma* sınıfı tipindeki **e** referansına bağ lı olmasıdır. Yapılandırıcısına 2 sayısını göndererek oluşturduğ umuz *Elma* nesnesi ise çöp toplayıcısı tarafından bellekten silinmiştir çünkü bu nesnemize herhangi bir referans bağ lı değildir. Uygulama sonucu aşağıdaki gibi olur:

```
Elma Nesnesi Olusturuluyor = 1
Elma Nesnesi Olusturuluyor = 2
Elma Nesnesi Yok Ediliyor = 2
```

Uygulamamızı şek il üzerinde gösterirsek:



**Şekil-3.2. Hangi nesne bellekten silinir?**

### 3.2.3. `finalize()` Yordamına Güvenilirse Neler Olur?

**Örnek-3.20:** *BenzinDepo.java*

```
class Ucak {
    String ucak_isim ;
    boolean benzin_deposu_dolu = false ;
    boolean benzin_deposu_kapagi_acik_mi = false ;

    Ucak(boolean depoyu_doldur ,String ucak_isim) {
        benzin deposu kapagi acik mi = true ;    // kapağı
```

```

    açıyoruz
        benzin_deposu_dolu = depoyu_doldur ;           // depo dolu
        this.ucak_isim =ucak_isim ;
    }

    /* Kapakların kapatılmasını finalize() yordamına bıraktık */
    public void finalize() {
        if (benzin_deposu_kapagi_acik_mi) {           //
    kapak açıksa
        benzin_deposu_kapagi_acik_mi = false ;       //
    kapağı kapa
        System.out.println(ucak_isim + "- kapaklari
    kapatildi ");
        }
    }
}

public class BenzinDepo {

    public static void main(String args[]) {

        Ucak ucak_1 = new Ucak(true,"F-16");           //
    benzin doldur
        new Ucak(true,"F-14");                         // benzin
    doldur

        System.gc();                                   // kapakları kapat
        System.out.println("Ucaklara benzin dolduruldu,
    kapaklari"
                                                                    +
    "kapatildi");
        }
    }
}

```

Bu örnekte 2 adet *Ucak* nesnesi oluşturuluyor. Nesneler oluşturulur oluşturmaz kapaklar açılıp depolarına benzin doldurulmaktadır. Kapakları kapatma işlemi için `finalize()` yordamını kullanılıyor. Sonuçta `System.gc()` çağrılınca çöp toplayıcısının tetiklendiği bilinmektedir. Uygulamanın sonucu aşağıdaki gibi olur:

```

F-14 - kapaklari kapatildi
Ucaklara benzin dolduruldu,kapaklari kapatildi

```

Uygulama çıktısının gösterdiği gibi, sadece F-14 isimli *Ucak* nesnesine ait benzin deposunun kapağı kapatılmış durumda, F-16 isimli *Ucak* nesnesinin kapağı ise hala açık durmaktadır –ki istenen durum F-16 isimli *Ucak* nesnesinde kapağının kapalı olmasıdır. Bu yanlışlığın sebebi, F-16 isimli *Ucak* nesnesine, yine *Ucak* tipinde olan `ucak_1` referansının bağlanmış olmasıdır. Çöp toplayıcısı boşta olan nesneleri bellekten siler, yani herhangi bir nesne, bir referansa bağlanmamış ise o nesne boşta demektir. Bu örneğimizde boşta olan nesne ise F-14 isimli *Ucak* nesnesidir ve çöp toplayıcısı tarafından temizlenmiştir ve bu temizlik işleminden hemen önce F-14 isimli *Ucak* nesnesinin `finalize()` yordamı çağırılmıştır.

Özet olarak, `System.gc()` ile çöp toplayıcısını tetikleyebiliriz ama referanslar, ilgili nesnelere bağlı kaldığı sürece, bu nesnelerin bellekten silinmesi söz konusu değildir. Dolayısıyla `finalize()` yordamı kullanılırken dikkatli olunmalıdır.

### 3.2.4. Çöp Toplayıcısı (*Garbage Collector*) Nasıl Çalışır?

Çöp toplayıcısının temel görevi kullanılmayan nesneleri bularak bellekten silmektir. *Sun Microsystems* tarafından tanıtılan Java HotSpot VM (*Virtual Machine*) sayesinde *heap* bölgesindeki nesneler nesillerine göre ayrılmaktadır. Bunlar **eski nesil** ve **yeni nesil** nesneler olmak üzere iki çeşittir. Belli başlı parametreler kullanarak Java HotSpot VM mekanizmasını denetlemek mümkündür; bizlere sağlandığı hazır parametreler ile normal bir uygulama gayet performanslı çalışabilir. Eğer sunucu üzerinde uygulama geliştiriliyorsa, Java HotSpot VM parametrelerini doğru kullanarak uygulamanın performansının arttırmak mümkündür.

Daha önceden söz edildiği gibi nesnelerin bellekten silinmesi görevi programcıya ait değildir. Bu işlem tamamen çöp toplayıcısının sorumluluğundadır. Java HotSpot VM ait çöp toplayıcısı iki konuyu kullanıcılara garanti etmektedir.

- Kullanılmayan nesnelerin kesinlikle bellekten silinmesini sağlamak.
- Nesne bellek alanının parçalanmasını engellemek ve belleğin sıkıştırılmasını sağlamak.

Bu bölümde dört adet çöp toplama algoritmasından bahsedilecektir, bunlardan ilki ve en temel olanı referans sayma yöntemidir, bu yöntem modern JVM'ler (*Java Virtual Machine*) tarafından artık kullanılmamaktadır.

## ❶ Eski yöntem

### • Referans Sayma Yöntemi

Bu yöntemde, bir nesne oluşturulur oluşturulmaz kendisine ait bir sayaç çalıştırılmaya başlar ve bu sayacın ilk değeri birdir. Bu nesnemizin ismi **X** olsun. Bu sayacın saydığı şey, oluşturduğumuz nesneye kaç adet referansın bağlı olduğudur. Ne zaman yeni bir referans bu **X** nesnesine bağlanırsa, bu sayacın değeri bir artar. Aynı şekilde ne zaman bu **X** nesnesine bağlı olan bir referans geçerlilik alanı dışına çıksa veya bu referans `null` değerine eşitlenirse, **X** nesnesine ait bu sayacın değeri bir eksilir. Eğer sayaç sıfır değerini gösterirse, **X** nesnemizin artık bu dünyadan ayrılma zamanı gelmiş demektir ve çöp toplayıcısı tarafından bellekten silinir.

Bu yöntem, kısa zaman aralıkları ile çalıştırıldığında iyi sonuçlar vermektedir ve gerçek zamanlı uygulamalar için uygun olduğu söylenebilir. Fakat bu yöntemin kötü yanı döngüsel ilişkilerde referans sayacının doğru değerler göstermemesidir. Örneğin iki nesnemiz olsun, bunlardan biri **A** nesnesi diğeri ise **B** nesnesi olsun. Eğer **A** nesnesi, **B** nesnesine, **B** nesnesi de, **A** nesnesine döngüsel bir biçimde işaret ediyorsa ise bu nesneler artık kullanılmıyor olsa dahi bu nesnelere ait sayaçların değerleri hiç bir zaman sıfır olmaz ve bu yüzden çöp toplayıcısı tarafından bellekten silinmezler.

## ❷ Yeni Yöntemler

Toplam 3 adet yeni çöp toplama yönetimi vardır. Her üç yöntemin de yaşayan nesneleri bulma stratejisi aynıdır. Bu strateji bellek içinde yer alan statik ve yığın (*stack*) alanlarındaki referansların bağlı bulunduğu nesneler aranarak bulunur. Eğer geçerli bir referans, bir nesneye bağlıysa, bu nesne uygulama tarafından kullanılıyor demektir.

### • Kopyalama yöntemi

Oluşturulan bir nesne, ilk olarak *heap* bölgesindeki yeni nesil alanında yerini alır. Eğer bu nesne zaman içinde çöp toplayıcısı tarafından silinmemiş ise belirli bir olgunluğa ulaşmış demektir ve *heap* bölgesindeki eski nesil alanına geçmeye hak kazanır. Yeni nesil bölgeleri arasında kopyalanma işlemi ve bu alandan eski nesil alanına kopyalanma işlemi, kopyalama yöntemi sayesinde gerçekleşir.

### • İşaretle ve süpür yöntemi

Nesneler zaman içinde belli bir olgunluğa erişince *heap* bölgesindeki eski nesil alanına taşındıklarını belirtmiştik. Eski nesil alanındaki nesneleri bellekten silmek ve bu alandaki parçalanmaları engellemek için işaretle ve süpür yöntemi kullanılır. İşaretle ve süpür yöntemi, kopyalama yöntemine göre daha yavaş çalışmaktadır.

### • Artan (sıra) yöntem

Kopyalama yöntemi veya işaretle ve süpür yöntemi uygulamanın üretmiş olduğu büyük nesneleri bellekten silerken kullanıcı tarafından fark edilebilir bir duraksama oluşturabilirler. Bu fark edilir duraksamaları ortadan kaldırmak için Java HotSpot VM artan yönetimini geliştirmiştir.

Artan yöntem büyük nesnelerin bellekten silinmeleri için orta nesil alanı oluşturur. Bu alan içerisinde küçük küçük bir çok bölüm vardır. Bu sayede büyük nesneleri bellekten silerken oluşan fark edilir duraksamalar küçük ve fark edilmez duraksamalara dönüştürülmektedir.

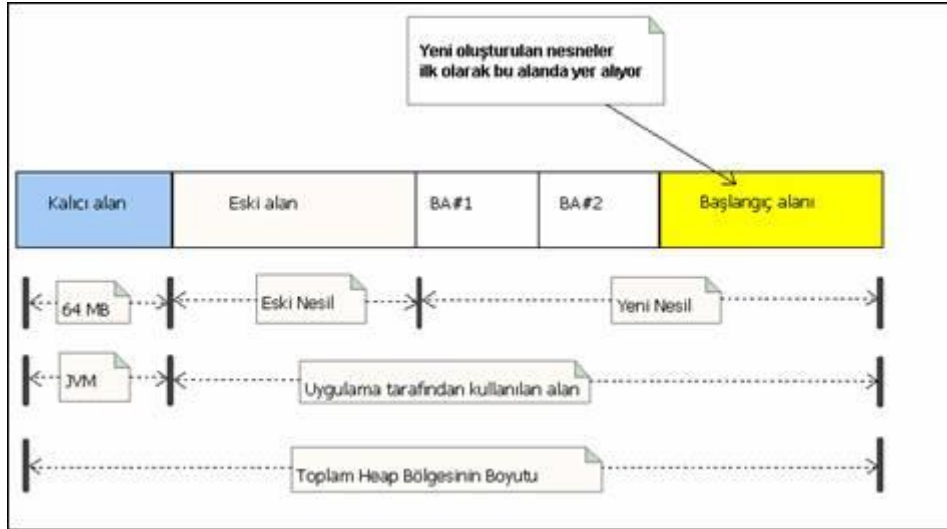
Artan yöntemi devreye sokmak için -Xincgc, çıkartmak için ise -Xnoincgc parametreleri kullanılır. Java HotSpot VM normal şartlarda bu yöntemi kullanmaz eğer kullanılması istiyorsak bu işlemi kendimiz yapmak zorundayız.

### **Gösterim-3.5:**

```
java -Xincgc BenzinDepo
```

#### **3.2.5. Heap bölgesi**

Java HotSpot VM, *heap* bölgesini nesillere göre yönetir. Bellek alanında değişik nesillere ait nesneler bulunur. Aşağıdaki şeklimizde *heap* bölgesinin nesillere göre nasıl ayrıldığını görebilirsiniz.



**Şekil-3.3. Heap bölgesi**

Kalıcı alan özel bir bölgedir (32 MB, 64 MB veya daha fazla olabilir). Bu bölgede JVM'e ait bilgiler bulunur. -XX:MaxPermSize=?M komutu ile bu alanın boyutları kontrol edilebilir. (??=*ne kadarlık bir alan gerektiği*, ör: java -X:MaxPermSize=64M)

#### **3.2.6. Yeni Nesil**

Yeni Nesil bölümü toplam 3 alandan oluşur: Başlangıç alanı ve iki adet boş alan (BA#1 ve BA#2). Bu iki boş alandan bir tanesi bir sonraki kopyalama (*kopyalama yöntemi sayesinde*) için her zaman boş tutulur. Başlangıç alanındaki nesneler belli bir olgunluğa ulaştıkları zaman boş olan alanlara kopyalanırlar.

#### **3.2.7. Eski Nesil**

Eski nesil nesneler, *heap'deki eski alanında* bulunurlar. Uygulama tarafından kullanılan uzun ömürlü nesneler yeni nesil alanından, eski nesil alanına taşınırlar. Eski nesil alan içerisinde de zamanla kullanılmayan nesneler olabilir. Bu nesnelerin silinmesi için işaretle ve süpür yöntemi kullanılır.

### 3.2.8. **Heap bölgesi Boyutları Nasıl Denetlenir?**

*Heap* bölgesine minimum veya maksimum değerleri vermek için `-Xms` veya `-Xmx` parametreleri kullanılır. Performansı arttırmak amacı ile geniş kapsamlı sunucu (*server-side*) uygulamalarında minimum ve maksimum değerler birbirlerine eşitlenerek sabit boyutlu *heap* bölgesi elde edilir.

JVM herhangi bir çöp toplama yöntemini çağırdıktan sonra *heap* bölgesini, boş alanlar ile yaşayan nesneler arasındaki farkı ayarlamak için büyütür veya azaltır. Bu oranı yüzdesel olarak minimum veya maksimum değerler atamak istenirse `-Xminf` veya

`-Xmaxf` parametreleri kullanılır. Bu değer (*SPARC Platform versiyonu*) için hali hazırda minimum %40, maksimum %70'dir.

#### **Gösterim-3.6:**

```
java -Xms32mb Temizle
```

#### **Gösterim-3.7:**

```
java -Xminf%30 Temizle
```

*Heap* bölgesi JVM tarafından büyütülüp küçültüldükçe, eski nesil ve yeni nesil alanları da `NewRatio` parametresine göre yeniden hesaplanır. `NewRatio` parametresi eski nesil alan ile yeni nesil alan arasındaki oranı belirlemeye yarar. Örneğin `-X:NewRatio=3` parametresinin anlamı eskinin yeniye oranının 3:1 olması anlamına gelir; yani, eski nesil alanı *heap* bölgesinin 3/4'inde, yeni nesil ise 1/3'inde yer kaplayacaktır. Bu şartlarda kopyalama yönteminin daha sık çalışması beklenir. Eğer ki eski nesil alanını daha küçük yaparsak o zaman işaretle ve süpür yöntemi daha sık çalışacaktır. Daha evvelden belirtildiği gibi işaretle ve süpür yöntemi, kopyalama yöntemine göre daha yavaş çalışmaktadır.

#### **Gösterim-3.8:**

```
java -XX:NewRatio=8 Temizle
```

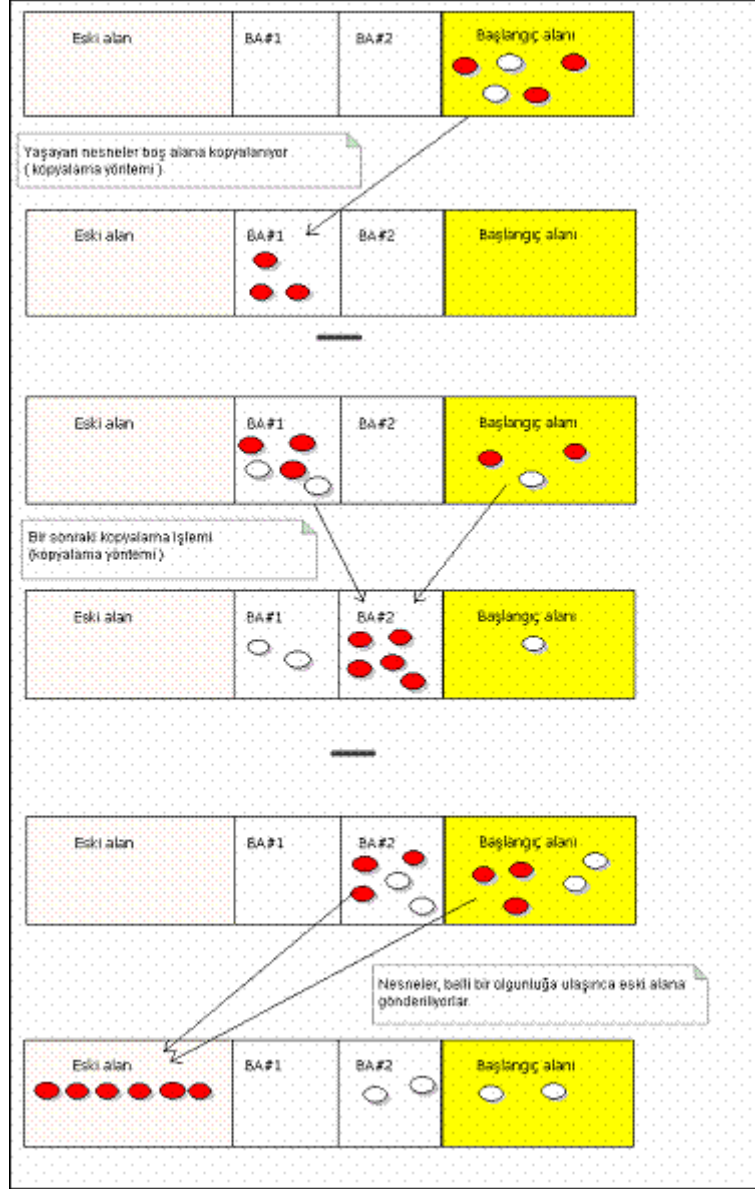
`NewSize` ve `MaxNewSize` parametreleri, yeni nesil alanının minimum ve maksimum değerlerini belirlemek için kullanılır. Eğer bu parametreler birbirlerine eşitlenirse, sabit uzunlukta yeni nesil alanı elde edilmiş olur. Aşağıdaki gösterimde yeni nesil alanlarının 32MB olacağı belirtilmiştir.

#### **Gösterim-3.9:**

```
java -XX:MaxNewSize=32m Temizle
```

### 3.2.9. **Kopyalama yönteminin gösterimi**

Aşağıda, Şekil 3-4'de yaşayan nesneler koyu renk ile gösterilmişlerdir:

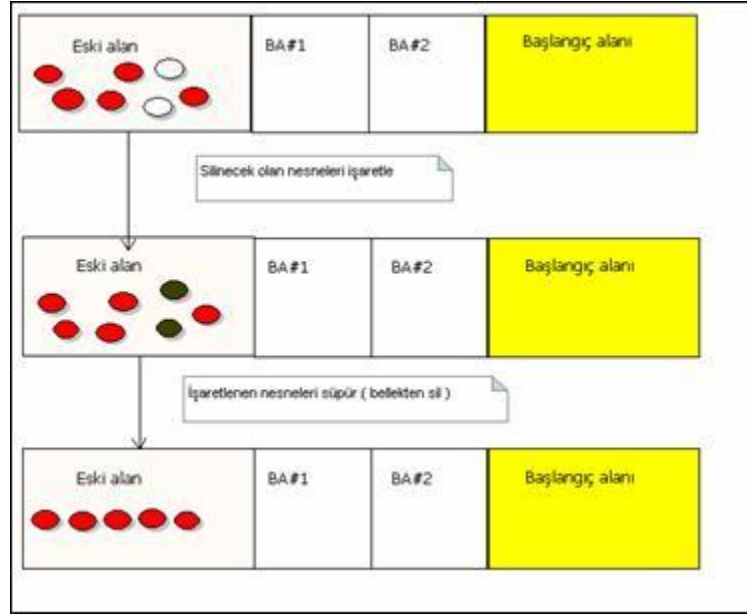


**Şekil-3.4. Heap Bölgesindeki Gelişmeler**

Nesneler belirli bir olgunluğa ulaşıncaya kadar, yeni nesil alanının bölümleri arasında geçirilirler. Bu sırada kopyalama yöntemi kullanılmaktadır. Belli bir olgunluğa ulaşan nesneler son aşamada eski alana gönderilirler.

### 3.2.10. İşaretle ve süpür yönteminin gösterimi

İyi tasarlanmış bir sistemde, çöp toplayıcısı bir kaç defa devreye girmesiyle, kullanılmayan nesneler bellekten silinir; geriye kalan yaşayan nesneler ise, eski nesil alanına geçmeye hak kazanırlar. Eski nesil alanında, işaretle ve süpür yöntemi kullanılır; bu yöntem kopyalama yöntemine göre daha yavaş fakat daha etkilidir.



**Şekil-3.5. İşaretle ve Süpür Yönteminin Gösterimi**

Aşağıdaki örnekte, kopyalama yönteminin ne zaman çalıştığını, işaretle ve süpür yönteminin ne zaman çalıştığını gösterilmektedir:

**Örnek-3.21:** *HeapGosterim.java*

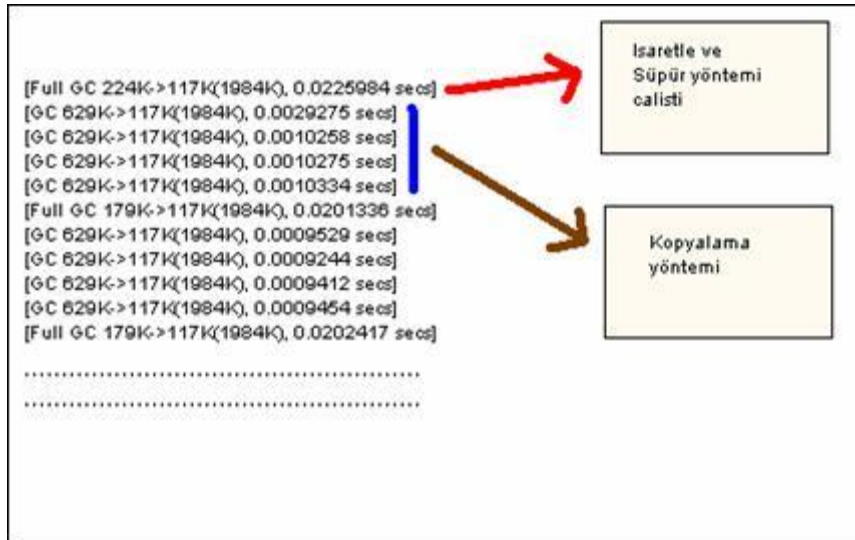
```
public class HeapGosterim {
    public static void main(String args[]) {
        for (int y=0 ; y<100000 ;y++) {
            new String("Yer Kaplamak icin");
            new String("Yer Kaplamak icin");
            new String("Yer Kaplamak icin");
            if ( (y%10000) == 0 ) {
                System.gc();
            }
        }
    }
}
```

Bu örnekte 3 adet *String* nesnesi *for* döngüsünün her bir çevriminde oluşturularak bellekten yer alınmaktadır. Nesneler herhangi bir referansa bağlı olmadıklarından dolayı, çöp toplayıcısının devreye girmesiyle birlikte bellekten silineceklerdir. Eğer uygulama aşağıdaki gibi çalıştırılacak olursa, gerçekleşen olaylar daha iyi görülebilir:

**Gösterim-3.10:**

```
java -verbosegc HeapGosterim
```

Uygulamanın sonucu aşağıdaki gibi olur:



**Şekil-3.6. Yöntemlerin Gösterimi**

**Not:** Ekran sonuçları üzerindeki oklar ve diğer gösterimler, dikkat çekmek amacıyla özel bir resim uygulaması tarafından yapılmıştır.

### 3.2.11. İlk Değerlerin Atanması

Java uygulamalarında üç tür değişken çeşidi bulunur: yerel (*local*) değişkenler, nesneye ait global alanlar ve son olarak sınıfa ait global alanlar (*statik alanlar*). Bu değişkenlerin tipleri temel (*primitive*) veya herhangi bir sınıf tipi olabilir.

#### **Örnek-3.22:** *DegiskenGosterim.java*

```

public class DegiskenGosterim {
    int x ;                //nesneye ait global alan
    static int y ;         // sınıfa ait global alan
    public void metod () {
        int i ; //yerel degisken
        //static int y = 5 ;    //yanlış
    }
}

```

### 3.2.12. Yerel Değişkenler

Yerel değişkenlere ilk değerleri programcı tarafından verilmelidir; ilk değeri verilmeden kullanılan yerel değişkenlere ilk tepki derleme (*compile-time*) anında verilir.

#### **Gösterim-3.11:**

```

public int hesapla () { // yerel değişkenlere ilk değerleri
                        // her zaman verilmelidir.

    int i ;
    i++;                // ! Hata ! ilk deger verilmeden üzerinde
işlem yapılamaz
    return i ;
}

```

### 3.2.13. Nesnelere Ait Global Alanlar



Nesnelere ait global alanlara ilk değerleri programcının vermesi zorunlu değildir; Java bu alanlara ilk değerleri kendiliğinden verir.

### 3.2.13.1. Nesnelere Ait Global Alanların Temel Bir Tip Olması Durumunda

Java'nın temel tipteki global alanlara hangi başlangıç değerleri atadığını görmek için aşağıdaki örneği inceleyelim,

#### **Örnek-3.23:** *IlkelTipler.java*

```
public class IlkelTipler {
    boolean mantiksal_deger;
    char krakter_deger;
    byte byter_deger;
    short short_deger;
    int int_deger;
    long long_deger;
    float float_deger;
    double double_deger;
    public void ekranaBas() {
        System.out.println("Veri Tipleri Ilk Degerleri");
        System.out.println("boolean " + mantiksal_deger );
        System.out.println("char [" + krakter_deger + "]
"+
        (int)krakter_deger );
        System.out.println("byte " + byter_deger );
        System.out.println("short " + short_deger );
        System.out.println("int " + int_deger );
        System.out.println("long " + long_deger );
        System.out.println("float " + float_deger );
        System.out.println("double " + double_deger);
    }
    public static void main(String args[]) {
        new IlkelTipler().ekranaBas();

        /*
        // yukaridaki ifade yerine
        // asagidaki ifadeyide kullanabilirsiniz.
        IlkelTipler it = new IlkelTipler();
        it.ekranaBas();
        */
    }
}
```

Bu uygulama sonucu aşağıdaki gibidir; buradan nesne alanlarına hangi başlangıç değerleri görülebilir:

|              |               |
|--------------|---------------|
| Veri Tipleri | Ilk Degerleri |
| boolean      | false         |
| char [ ]     | 0             |
| byte         | 0             |
| short        | 0             |
| int          | 0             |
| long         | 0             |
| float        | 0.0           |
| double       | 0.0           |

### 3.2.13.2. Nesnelere Ait Global Alanların Sınıf Tipi Olması Durumunda

Aksi belirtilmediği sürece nesnelere ait global alanlara, herhangi bir sınıf tipinde olması durumunda, başlangıç değeri olarak "null" atanır; yani boş değer... Eğer bu alanlar bu içeriğiyle kullanılmaya

kalkışılırsa, çalışma anında (*run time*) hata ile karşılaşılır. Hata ile karşılaşmamak için ilgili alanları uygun bir nesnelere bağlamak gerekir.

**Örnek-3.24:** *NesneTipleri.java*

```
public class NesneTipleri {
    String s ;
    public static void main(String args[]) {
        NesneTipleri nt = new NesneTipleri();
        System.out.println("s = " + nt.s );
        // nt.s = nt.s.trim(); //hata
    }
}
```

Uygulama sonucu aşağıdaki gibi olur:

```
s = null
```

### 3.2.14. Sınıflara Ait Global Alanlar (Statik Alanlar)

Sınıflara ait global alanlara (statik alanlar) değer atamakla nesnelere ait global alanlara değer atamak arasında bir fark yoktur. Buradaki önemli nokta statik alanların ortak olmasıdır.

#### 3.2.14.1. Sınıflara Ait Global Alanların (Statik Alanların) Temel Bir Tip Olması Durumunda

**Örnek-3.25:** *IlkelTiplerStatik.java*

```
public class IlkelTiplerStatik {
    static boolean mantiksal_deger;
    static char krakter_deger;
    static byte byter_deger;
    static short short_deger;
    static int int_deger;
    static long long_deger;
    static float float_deger;
    static double double_deger;
    public void ekranaBas() {
        System.out.println("Veri Tipleri İlk Değerleri");
        System.out.println("static boolean " +
mantiksal_deger );
        System.out.println("static char [" + krakter_deger
+ "]" );
        System.out.println("static byte " + byter_deger );
        System.out.println("static short " + short_deger
);
        System.out.println("static int " + int_deger );
        System.out.println("static long " + long_deger );
        System.out.println("static float " + float_deger
);
        System.out.println("static double " +
double_deger);
    }
    public static void main(String args[]) {
        new IlkelTiplerStatik().ekranaBas();
    }
}
```

```

/*
// yukaridaki ifade yerine
// asagidaki ifadeyi de kullanabilirsiniz.
    IlkelTiplerStatik its = new
    IlkelTiplerStatik();
    its.ekranaBas();    */
}
}

```

Uygulamanın sonucu aşağıdaki gibi olur:

| Veri Tipleri    | Ilk Degerleri |
|-----------------|---------------|
| static boolean  | false         |
| static char [ ] | 0             |
| static byte     | 0             |
| static short    | 0             |
| static int      | 0             |
| static long     | 0             |
| static float    | 0.0           |
| static double   | 0.0           |

### 3.2.14.2. Sınıflara Ait Global Alanların (Statik Alanların) Sınıf Tipi Olması Durumunda

Aksi belirtilmediği sürece, sınıflara ait global alanlar herhangi bir sınıf tipinde olması durumunda, bu alana başlangıç değeri olarak "null" atanır.

#### **Örnek-3.26:** *StatikNesneTipleri.java*

```

public class StatikNesneTipleri {
    static String s ;
    public static void main(String args[]) {
        StatikNesneTipleri snt = new
        StatikNesneTipleri();
        System.out.println("s = " + snt.s );
        // snt.s = snt.s.trim(); //hata
    }
}

```

Uygulamanın çıktısı aşağıdaki gibi olur:

S = null

### 3.2.15. İlk Değerler Atanırken Yordam (Method) Kullanımı

Global olan nesnelere ait alanlara veya statik alanlara ilk değerleri bir yordam aracılığı ile de atanabilir:

#### **Örnek-3.27:** *KarisikTipler.java*

```

public class KarisikTipler {
    boolean mantiksal_deger = mantiksalDegerAta(); //
    doğru (true) değerini alır

    static int int_deger = intDegerAta(); // 10 değerini alır
    String s ;
    double d = 4.17 ;
    public boolean mantiksalDegerAta() {
        return true ;
    }
    public static int intDegerAta() {
        return 5*2 ;
    }
    public static void main(String args[]) {
        new KarisikTipler();
    }
}

```

Dikkat edilirse, statik olan `int_deger` alanına başlangıç değeri statik bir yordam tarafından verilmektedir.

### 3.2.16. İlk Değer Alma Sırası

Nesnelere ait global alanlara başlangıç değerleri hemen verilir; üstelik, yapılandırıcılardan (*constructor*) bile önce... Belirtilen alanların konumu hangi sırada ise başlangıç değeri alma sırasında aynı olur.

#### **Örnek-3.28:** *Defter.java*

```

class Kagit {
    public Kagit(int i) {
        System.out.println("Kagit (" + i + ") ");
    }
}
public class Defter {
    Kagit k1 = new Kagit(1); // dikkat
    public Defter() {
        System.out.println("Defter() yapilandirici ");
        k2 = new Kagit(33); //artık başka bir Kagit nesnesine bağlı
    }
    Kagit k2 = new Kagit(2); //dikkat

    public void islemTamam() {
        System.out.println("Islem tamam");
    }

    Kagit k3 = new Kagit(3); //dikkat

    public static void main (String args[]) throws
    Exception {
        Defter d = new Defter();
        d.islemTamam();
    }
}

```

Uygulama sonucu aşağıdaki gibi olur:

```

Kagit (1)
Kagit (2)
Kagit (3)
Defter() yapilandirici

```

```
Kagit (33)
Islem tamam
```

Görüleceği gibi ilk olarak k1 daha sonra k2 ve k3 alanlarına değerler atandı; sonra sınıfa ait yapılandırıcı çağrıldı... Ayrıca k3 alanı, yapılandırıcının içerisinde oluşturulan başka bir *Kagit* nesnesine bağlanmıştır. Peki?, k3 değişkeninin daha önceden bağlandığı *Kagit* nesnesine ne olacaktır? Yanıt: çöp toplayıcısı tarafından bellekten silinecektir. Uygulama `islemTamam()` yordamı çağrılarak çalışması sona ermektedir...

### 3.2.17. Statik ve Statik Olmayan Alanların Değer Alma Sırası

Statik alanlar, sınıflara ait olan alanlardır ve statik olmayan alanlara (*nesne alanları*) göre başlangıç değerlerini daha önce alırlar.

**Örnek-3.29:** *Kahvalti.java*

```
class Peynir {
    public Peynir(int i, String tur) {
        System.out.println("Peynir (" + i + ") -->" + tur);
    }
}
class Tabak {
    public Tabak(int i, String tur) {
        System.out.println("Tabak (" + i + ") -->" + tur);
    }
    static Peynir p1 = new Peynir(1, "statik alan");
    Peynir p2 = new Peynir(2, "statik-olmayan alan");
}
class Catal {
    public Catal(int i, String tur) {
        System.out.println("Catal (" + i + ") --> " + tur);
    }
}
public class Kahvalti {
    static Catal c1 = new Catal(1, "statik alan"); //
    dikkat!
    public Kahvalti() {
        System.out.println("Kahvalti() yapilandirici");
    }
    Tabak t1 = new Tabak(1, "statik-olmayan alan"); //
    dikkat!
    public void islemTamam() {
        System.out.println("Islem tamam");
    }
    static Catal c2 = new Catal(2, "statik alan"); //
    dikkat!
    public static void main (String args[] ) throws
    Exception {
        Kahvalti d = new Kahvalti();
        d.islemTamam();
    }
    static Tabak t4 = new Tabak(4, "statik alan"); //
    dikkat!
    static Tabak t5 = new Tabak(5, "statik alan"); //
    dikkat!
}
```

Sırayla gidilirse, öncelikle c1, c2, t4, t5 alanlarına başlangıç değerleri verilecektir. *Catal* nesnesini oluşturulurken, bu sınıfın yapılandırıcısına iki adet parametre gönderilmektedir; birisi `int` tipi, diğeri ise *String* tipindedir. Böylece hangi *Catal* tipindeki alanların, hangi *Catal* nesnelerine bağlandığı anlaşılabilir.

*Catal* nesnesini oluştururken olaylar gayet açıktır; yapılandırıcı çağrılır ve ekrana gerekli bilgiler yazdırılır...

```
Catal (1) --> statik alan
Catal (2) --> statik alan
```

*Tabak* sınıfına ait olan yapılandırıcıda iki adet parametre kabul etmektedir. Bu sayede oluşturulan *Tabak* nesnelerinin ne tür alanlara (*statik mi? Değil mi?*) bağlı olduğunu öğrenme şansına sahip olunur. Olaylara bu açıdan bakılırsa; statik olan *Catal* sınıfı tipindeki c1 ve c2 alanlarından sonra yine statik olan *Tabak* sınıfı tipindeki t4 ve t5 alanları ilk değerlerini alacaklardır. Yalnız *Tabak* sınıfını incelersek, bu sınıfın içerisinde de global olan statik ve nesnelere ait alanların olduğunu görürüz. Bu yüzden *Tabak* sınıfına ait yapılandırıcının çağrılmasından evvel *Peynir* sınıfı tipinde olan bu alanlar ilk değerlerini alacaklardır. Doğal olarak bu alanlardan ilk önce statik olan p1 daha sonra ise nesneye ait olan p2 alanına ilk değerleri verilecektir. Statik olan alanların her zaman daha önce ve yalnız bir kere ilk değerini aldığı kuralını hatırlarsak, ekrana yansıyan sonucun şaşırtıcı olmadığını görölür.

```
Peynir (1) →statik alan
```

Daha sonradan da statik olmayan *Peynir* tipindeki alana ilk değeri verilir.

```
Peynir (2) -->statik-olmayan alan
```

Ve nihayet *Tabak* sınıfının yapılandırıcısı çalıştırılarak, *Tabak* nesnesi oluşturulur.

```
Tabak (4) -->statik alan
```

Bu işlemlerde sonra sıra statik t5 alanına, *Tabak* nesnesini bağlanır. Bu işlemde sadece statik olmayan *Peynir* tipindeki p2 alanına ilk değeri atanır. Statik olan p1 alanına bu aşamada herhangi bir değer atanmaz (*daha evvelden değer atanmıştı*). Bunun sebebi ise statik alanlara sadece bir kere değer atanabilmesidir (*daha önceden belirtildiği üzere*). Bu işlemden sonra *Tabak* nesnesinin yapılandırıcısı çağrılır.

```
Peynir (2) -->statik-olmayan alan
Tabak (5) -->statik alan
```

Daha sonra *Kahvalti* nesnesine ait olan *Tabak* sınıfı tipindeki t1 alanına ilk değeri verilir:

```
Peynir (2) -->statik-olmayan alan
Tabak (1) -->statik-olmayan alan
```

Dikkat edilirse, *Tabak* sınıfının içerisinde bulunan *Peynir* tipindeki global p2 isimli alan (*nesneye ait alan*), her yeni *Tabak* nesnesini oluşturulduğunda başlangıç değerini almaktadır.

Sondan bir önceki adım ise, *Kahvalti* sınıfının yapılandırıcısının çağırılması ile *Kahvalti* nesnesinin oluşturulmasıdır:

```
Kahvalti() yapilandirici
```

Son adım, *Kahvalti* nesnesinin *islemTamam()* yordamını çağırarak ekrana aşağıdaki mesajı yazdırmaktadır:

```
Islem tamam
```

Ekrana yazılanlar toplu bir halde aşağıdaki gibi görülür:

```
Catal (1) --> statik alan
Catal (2) --> statik alan
Peynir (1) -->statik alan
Peynir (2) -->statik-olmayan alan
```

Tabak (4) -->statik alan  
Peynir (2) -->statik-olmayan alan  
Tabak (5) -->statik alan  
Peynir (2) -->statik-olmayan alan  
Tabak (1) -->statik-olmayan alan  
Kahvalti() yapilandirici  
Islem tamam

### 3.2.18. Statik Alanlara Toplu Değer Atama

Statik alanlara toplu olarak da değer atanabilir; böylesi bir örnek :

**Örnek-3.30:** *StatikTopluDegerAtama.java*

```
class Kopek {  
    public Kopek() {  
        System.out.println("Hav Hav");  
    }  
}  
public class StatikTopluDegerAtama {  
    static int x ;  
    static double y ;  
    static Kopek kp ;  
    {  
        x = 5 ;  
        y = 6.89 ;  
        kp = new Kopek();  
    }  
    public static void main(String args[]) {  
        new StatikTopluDegerAtama();  
    }  
}
```

Statik alanlara toplu değer vermek için kullanılan bu öbek, yalnızca, *StatikTopluDegerAtama* sınıfından nesne oluşturulduğu zaman veya bu sınıfa ait herhangi bir statik alana erişilmeye çalışıldığı zaman (*statik alanlara erişmek için ilgili sınıfa ait bir nesne oluşturmak zorunda değilsiniz*), bir kez çağrılır.

### 3.2.19. Statik Olmayan Alanlara Toplu Değer Atama

Statik olmayan alanlara toplu değer verme, şekilsel olarak statik alanlara toplu değer verilmesine benzer:

**Örnek-3.31:** *NonStatikTopluDegerAtama.java*

```
class Dinozor {  
    public Dinozor() {  
        System.out.println("Ben Denver");  
    }  
}  
public class NonStatikTopluDegerAtama {  
    int x ;  
    double y ;  
    Dinozor dz ;  
    {  
        x = 5 ;  
        y = 6.89 ;  
        dz= new Dinozor();  
    }  
    public static void main(String args[]) {  
        new NonStatikTopluDegerAtama();  
    }  
}
```

```
}  
}
```

### 3.3. Diziler (Arrays)

Diziler nesnedir; içerisinde belirli sayıda eleman bulunur. Eğer bu sayı sıfır ise, dizi boş demektir. Dizinin içerisindeki elemanlara eksi olmayan bir tam sayı ile ifade edilen dizi erişim indisi ile erişilir. Bir dizide **n** tane eleman varsa dizinin uzunluğu da **n** kadardır; ilk elemanın indisi/konumu **0**'dan başlar, son elemanı ise **n-1**'dir.

Dizi içerisindeki elemanlar aynı türden olmak zorundadır. Eğer dizi içerisindeki elemanların türü `double` ise, bu dizinin türü için `double` denilir. Bu `double` tipinde olan diziye `String` tipinde bir nesne atanması denenirse hata ile karşılaşılır. Diziler temel veya herhangi bir sınıf tipinde olabilir...

#### 3.3.1. Dizi Türündeki Referanslar

Dizi türündeki referanslar, dizi nesnelere bağlanmaktadır. Dizi referansları tanımlamak bu dizinin hemen kullanılacağı anlamına gelmez...

##### Gösterim-3.12:

```
double[] dd ;      // double tipindeki dizi  
double dd[] ;      // double tipindeki dizi  
float [] fd ;      // float tipindeki dizi  
Object[] ao ;      // Object tipindeki dizi
```

Gösterim-3.12'de yalnızca dizi nesnelere bağlanacak olan referanslar tanımlandı; bu dizi nesneleri bellek alanında henüz yer kaplamamışlardır. Dizi nesnelerini oluşturmak için **new** anahtar sözcüğü kullanılması gereklidir.

#### 3.3.2. Dizileri Oluşturmak

Diziler herhangi bir nesne gibi oluşturulabilir:

##### Gösterim-3.13:

```
double[] d = new double[20]; // 20 elemanlı double tipindeki dizi  
double dd[] = new double[20]; // 20 elemanlı double tipindeki dizi  
float []fd = new float [14]; // 14 elemanlı float tipindeki dizi  
Object[]ao = new Object[17]; // 17 elemanlı Object tipindeki dizi  
String[] s = new String[25]; // 25 elemanlı String tipindeki dizi
```

Örneğin, **new double[20]** ifadesiyle 20 elemanlı temel `double` türünde bir dizi elde edilmiş oldu; bu dizi elemanları başlangıç değerleri **0.0**'dır. Java'nın hangi temel türe hangi varsayılan değeri atadığını görmek için Bölüm 1'e bakılabilir.

#### 3.3.2.1. Temel Türlerden Oluşan Bir Dizi

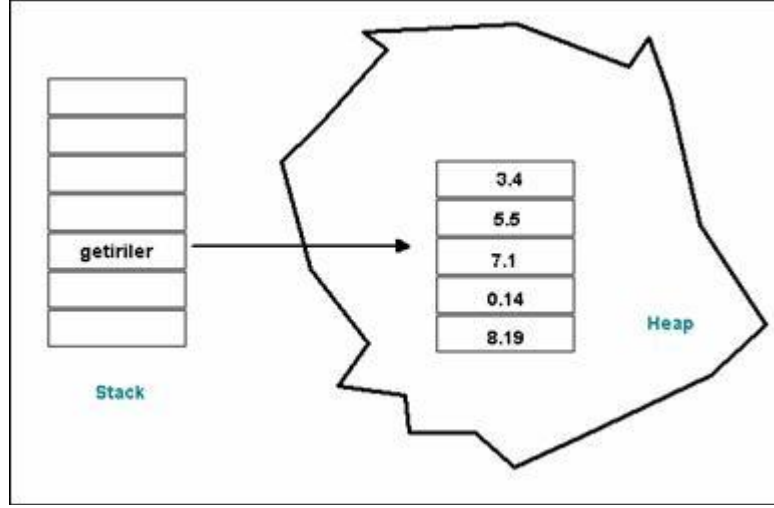
Bir dizi temel türde ise, elemanları aynı temel türde olmak zorundaydı; bu tekrar anımsatıldıktan sonra aşağıdaki gösterim yakından incelenirse,

##### Gösterim-3.14:

```
double[] getiriler = { 3.4, 5.5, 7,1, 0.14, 8.19 } ;
```

Gösterim-3.14'deki dizi tanımını, aşağıdaki şekil üzerinde açıklanabilir:





**Şekil-3.7. Temel (*primitive*) türde bir dizi**

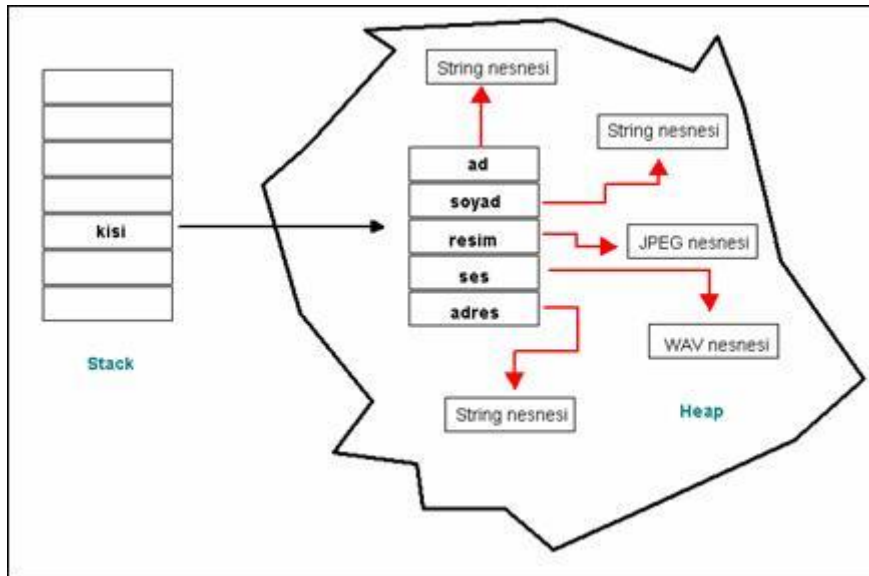
Temel `double` türdeki `getiriler` referansı yığın (*stack*) alanında yer almaktadır. Bu referansın bağlı olduğu dizi nesnesi ise *heap* alanında bulunmaktadır. Bu dizimizin elemanları temel `double` tipinde olduğu için dizi elemanlarının değerleri kendi üzerlerinde dururlar.

### 3.3.2.2. Nesnelerden Oluşan Bir Dizi

Nesnelerden oluşan bir dizi, temel türlerden oluşan bir diziden çok daha farklıdır. Nesnelerden meydana gelmiş bir dizi oluşturulduğu zaman, bu dizinin elemanlarının içerisinde, ilgili nesnelere ait referanslar tutulur. Örneğin aşağıdaki gibi bir dizi yapısı bulunsun:

- Ad, *String* sınıfı tipinde
- Soyad, *String* sınıfı tipinde
- Resim, *JPEG* sınıfı tipinde (*böyle bir sınıf olduğunu varsayalım*)
- Ses, *WAV* sınıfı tipinde (*böyle bir sınıf olduğunu varsayalım*)
- Adres, *String* sınıfı tipinde.

Bu dizinin içerisinde 5 adet farklı nesne bulunmaktadır. Tüm nesnelerin içerisine koyulacak olan bu dizinin tipi *Object* sınıfı tipinde olmalıdır. *Object* sınıfı, Java programlama dilinde, sınıf ağacının en tepesinde bulunur. *Object* sınıfının ayrıntılarını bölümlerde ele alınacaktır... Verilen dizi aşağıdaki gibi gösterilebilir:



**Şekil-3.8. *Object* sınıfı türündeki bir dizi**

Bu dizinin içerisinde ilgili nesnelerin kendileri değil, bu nesnelere bağlı olan referanslar durur.

### 3.3.3. Dizi Boyunun Değiştirilmesi

Dizi boyutu birkez verildi mi, artık değiştirilemezler!

#### **Gösterim-3.15:**

```
int liste[] = new int[5] ;
liste = new int[15] ; // yeni bir dizi nesnesi bağlandı
```

Gösterim-3.15’de dizi boyutlarının büyütüldüğünü sanmayın; burada, yalnızca yeni bir dizi nesnesi daha oluşturulmaktadır. `liste` dizi referansının daha evvelden bağlanmış olduğu dizi nesnesi (`new int[5]`), çöp toplama işlemi sırasında çöp toplayıcısı tarafından bellekten silinecektir.

### 3.3.4. Dizi Elemanlarına Erişim

Java dilinde dizi kullanımı diğer dillere nazaran daha az risklidir; anlamı, eğer tanımladığımız dizinin sınırları aşılsak, çalışma-anında (*runtime*) hata ile karşılaşacağımızdır. Örneğin 20 elemanlı bir `double` dizisi tanımlanmış olsun. Eğer bu dizinin 78. elemanına ulaşmak istenirse (- ki böyle bir indisli eleman yok), olanlar olur ve çalışma-anında hata alınır; böylesi hatanın (*ArrayIndexOutOfBoundsException*) çalışma-anında alınması güvenlik açısından güzel bir olaydır. Böylece dizi için ayrılmış bellek alanından dışarı çıkılıp başka verilere müdahale edilmesi engellenmiş olunur.

#### **Örnek-3.32:** *DiziElemanlariGosterimBir.java*

```
public class DiziElemanlariGosterimBir {
    public static void main(String args[]) {
        double[] d = { 2.1, 3.4, 4.6, 1.1, 0.11 } ;
        String[] s = { "defter", "kalem", "sarman", "tekir", "boncuk" } ; ;
        // double tipindeki dizimizi ekrana yazdırıyoruz
        for (int i = 0 ; i < d.length ; i++) {
            System.out.println("d["+i+"] = " + d[i] );
            // System.out.println("d["+78+"] = " + d[78] );    // Hata !
        }
        System.out.println("-----");

        // String tipindeki dizimizi ekrana yazdırıyoruz
        for (int x = 0 ; x < s.length ; x++) {
            System.out.println("s["+x+"] = " + s[x] );
            // System.out.println("s["+78+"]=" + s[78] );    // Hata !
        }
    }
}
```

`length` ifadesiyle bir dizinin içerisindeki eleman sayısı öğrenilir. Bu örnekte iki adet dizi tanımlandı: `double` ve `String` türündeki dizilerin içerisine 5’er adet eleman yerleştirildi ve sonradan bunları `for` döngüsü ile ekrana yazdırıldı. `i < d.length` ifadesine dikkat edilirse, döngü sayacın 4’e kadar artmaktadır; ancak, döngü sayacının 0’dan başladığı unutulmamalıdır.

Eğer 5 elemana sahip olan dizinin 78. elemanına erişilmeye kalkışılırsa, derleme anında (*compile-time*) bir hata ile karşılaşılmaz; ancak, uygulama yürütüldüğü zaman; yani, çalışma-anında (*runtime*) hata ile karşılaşılır. Uygulamanın sonucu aşağıdaki gibi olur:

```
d[0] = 2.1
d[1] = 3.4
d[2] = 4.6
d[3] = 1.1
d[4] = 0.11
-----
s[0] = defter
```

```
s[1] = kalem
s[2] = sarman
s[3] = tekir
s[4] = boncuk
```

Bir önceki uygulamanın çalışma anına hata vermesi istenmiyorsa, yorum satırı olan yerler açılması ve uygulamanın baştan derlenip çalıştırması gerekmektedir. Aynı örnek daha değişik bir şekilde ifade edilebilir:

**Örnek-3.33:** *DiziElemanlariGosterimIki.java*

```
public class DiziElemanlariGosterimIki {
    double[] d ;
    String[] s ;
    public DiziElemanlariGosterimIki() { // double tipindeki diziye eleman atanıyor
        d = new double[5];
        d[0] = 2.1 ;
        d[1] = 3.4 ;
        d[2] = 4.6 ;
        d[3] = 1.1 ;
        d[4] = 0.11 ;
        // d[5] = 0.56 ; // Hata !
        // String tipindeki diziye eleman atanıyor
        s = new String[5] ;
        s[0] = new String("defter");
        s[1] = new String("kalem");
        s[2] = new String("sarman");
        s[3] = new String("tekir");
        s[4] = new String("boncuk");
        // s[5] = new String("duman"); // Hata !
    }
    public void ekranaBas() { // double tipindeki diziye ekrana yazdırıyoruz
        for (int i = 0 ; i < d.length ; i++) {
            System.out.println("d["+i+"] = " + d[i] );
        }
        System.out.println("-----"); // String dizi ekrana yazdırılıyor
        for (int x = 0 ; x < s.length ; x++) {
            System.out.println("s["+x+"] = " + s[x] );
        }
    }
    public static void main(String args[]) {
        DiziElemanlariGosterimIki deg = new
            DiziElemanlariGosterimIki();
        deg.ekranaBas();
    }
}
```

Bu örnekte 5 elemanlı dizilere 6. eleman eklenmeye çalışıldığında, derleme anında (*compile-time*) herhangi bir hata ile karşılaşmayız. Hata ile karşılaşacağımız yer çalışma anındadır. Çünkü bu tür hatalar çalışma anında kontrol edilir. Yalnız çalışma anında hata oluşturabilecek olan satırlar kapatıldığı için şu an için herhangi bir tehlike yoktur; ancak, çalışma anında bu hatalar ile tanışmak isterseniz, bu satırların başında “//” yorum ekini kaldırmanız yeterli olacaktır. Uygulamanın sonucu aşağıdaki gibi olacaktır:

```
d[0] = 2.1
d[1] = 3.4
d[2] = 4.6
d[3] = 1.1
d[4] = 0.11
-----
s[0] = defter
s[1] = kalem
```

```
s[2] = sarman
s[3] = tekir
s[4] = boncuk
```

### 3.3.5. Diziler Elemanlarını Sıralama

Dizi elemanlarını büyükten küçüğe doğru sıralatmak için *java.util* paketini altındaki *Arrays* sınıfı kullanılabilir. Bu sınıfın statik **sort()** yordamı sayesinde dizilerin içerisindeki elemanlar sıralanabilir:

#### **Örnek-3.34:** *DiziSiralama.java*

```
import java.util.*; // java.util kütüphanesini kullanmak için
public class DiziSiralama {
    public static void ekranaBas(double[] d) {
        for (int i = 0 ; i < d.length ; i++) {
            System.out.println("d["+i+"] = " + d[i]);
        }
    }
    public static void main(String args[]) {
        double d[] = new double[9];
        d[0] = 2.45; d[1] = 3.45 ; d[2] = 4.78;
        d[3] = 1.45; d[4] = 15.12; d[5] = 1;
        d[6] = 9; d[7] = 15.32 ; d[8] = 78.17;
        System.out.println("Karisik sirada"); ekranaBas(d);
        Arrays.sort( d ) ;
        System.out.println("Siralanmis Sirada"); ekranaBas(d);
    }
}
```

Uygulama sonucu aşağıdaki gibi olur:

```
Karisik sirada
d[0] = 2.45
d[1] = 3.45
d[2] = 4.78
d[3] = 1.45
d[4] = 15.12
d[5] = 1.0
d[6] = 9.0
d[7] = 15.32
d[8] = 78.17
Siralanmis Sirada
d[0] = 1.0
d[1] = 1.45
d[2] = 2.45
d[3] = 3.45
d[4] = 4.78
d[5] = 9.0
d[6] = 15.12
d[7] = 15.32
d[8] = 78.17
```

### 3.3.6. Dizilerin Dizilere Kopyalanması

Bir dizi tümünden diğer bir diziye kopyalanabilir:

### **Örnek-3.35:** *DizilerinKopyalanmasi.java*

```
public class DizilerinKopyalanmasi {
    public static void main(String args[]) {
        int[] dizi1 = { 1,2,3,4 }; //ilk dizi
        int[] dizi2 = { 100,90,78,45,40,30,20,10}; // daha geniş olan 2. dizi

        // kopyalama işlemi başlıyor
        // 0. indisinden dizi1 uzunluğu kadar kopyalama yap
        System.arraycopy(dizi1,0,dizi2,0,dizi1.length);
        for (int i = 0 ; i < dizi2.length ; i++) {
            System.out.println("dizi2["+i+"] = "+ dizi2[i] );
        }
    }
}
```

*System* sınıfının statik yordamı olan `arraycopy()` sayesinde `dizi1` `dizi2`'ye kopyalandı. Sonuç aşağıdaki gibi olur:

```
dizi2[0] = 1
dizi2[1] = 2
dizi2[2] = 3
dizi2[3] = 4
dizi2[4] = 40
dizi2[5] = 30
dizi2[6] = 20
dizi2[7] = 10
```

### 3.3.7. Çok Boyutlu Diziler

Çok boyutlu diziler, Java'da diğer programlama dillerinden farklıdır. Sonuçta dizinin tek türde olması gerekir; yani, dizi içerisinde diziler (*dizilerin içerisinde dizilerin içerisindeki diziler şeklinde de gidebilir...*) tanımlayabilirsiniz.

#### **Gösterim-3.16:**

```
int[][] t1 = {
    { 1, 2, 3, },
    { 4, 5, 6, },
};
```

**Gösterim-3.16** 'de ifade edildiği gibi iki boyutlu temel türden oluşmuş çok boyutlu dizi oluşturulabilir. Çok boyutlu dizileri oluşturmanın diğer bir yolu ise,

#### **Gösterim-3.17:**

```
int [] [] t1 = new int [3][4] ;
int [] [] t1 = new int [][4] ; //!Hata !
```

Çok boyutlu dizileri bir uygulama üzerinde incelersek;

#### **Örnek:** *CokBoyutluDizilerOrnekBir.java*

```
public class CokBoyutluDizilerOrnekBir {
    public static void main(String args[]) {
        int ikiboyutlu[][] = new int[3][4] ;
        ikiboyutlu[0][0] = 45 ;
        ikiboyutlu[0][1] = 83 ;
        ikiboyutlu[0][2] = 11 ;
        ikiboyutlu[0][3] = 18 ;
        ikiboyutlu[1][0] = 17 ;
        ikiboyutlu[1][1] = 56 ;
        ikiboyutlu[1][2] = 26 ;
```

```

        ikiboyutlu[1][3] = 79 ;
        ikiboyutlu[2][0] = 3 ;
        ikiboyutlu[2][1] = 93 ;
        ikiboyutlu[2][2] = 43 ;
        ikiboyutlu[2][3] = 12 ;
        // ekrana yazdırıyoruz
        for (int i = 0 ; i<ikiboyutlu.length ; i++ ) {
            for (int j = 0 ; j < ikiboyutlu[i].length ; j++
) {
                System.out.println(" ikiboyutlu["+i+"] ["+j+"]
= "
                                +
        ikiboyutlu[i][j] );
            }
        }
    }
}

```

Verilen örnekte int türünde 3'e 4'lük (3x4) çok boyutlu dizi oluşturuldu; bu diziyi 3'e 4'lük bir matris gibi de düşünülebilir. Uygulama sonucu aşağıdaki gibi olur:

```

        ikiboyutlu[0][0] =45
        ikiboyutlu[0][1] =83
        ikiboyutlu[0][2] =11
        ikiboyutlu[0][3] =18
        ikiboyutlu[1][0] =17
        ikiboyutlu[1][1] =56
        ikiboyutlu[1][2] =26
        ikiboyutlu[1][3] =79
        ikiboyutlu[2][0] =3
        ikiboyutlu[2][1] =93
        ikiboyutlu[2][2] =43
        ikiboyutlu[2][3] =12

```

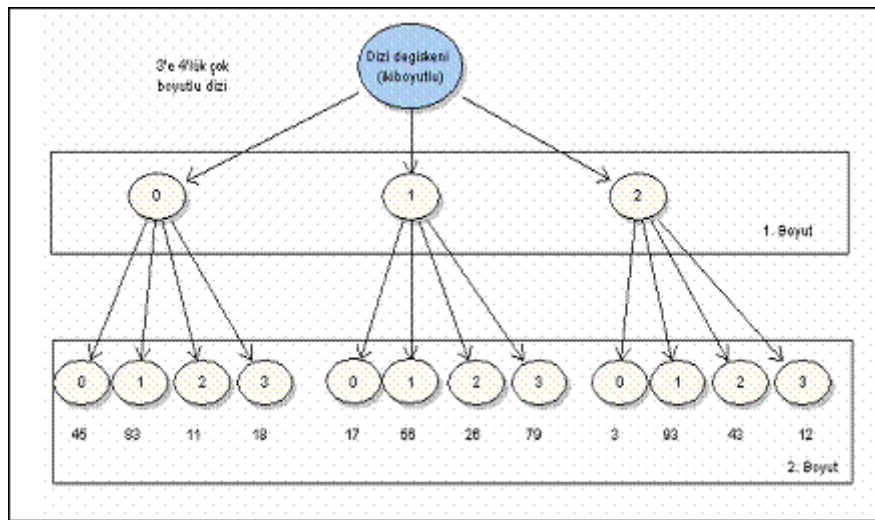
Uygulama sonucu matris gibi düşünülürse aşağıdaki gibi olur:

```

        45 83 11 18
        17 56 26 79
        3 93 43 12

```

Uygulama şekilsel olarak gösterilirse:



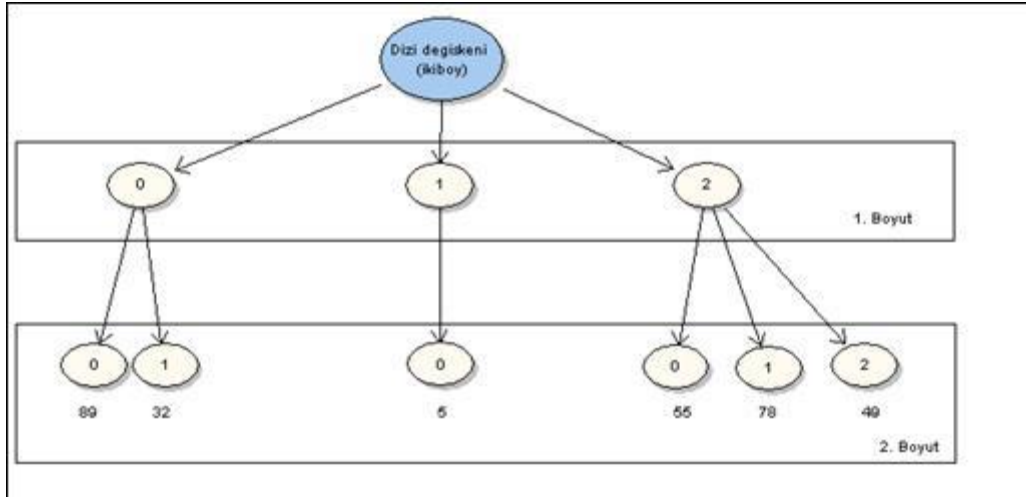
**Şekil-3.9. Aynı elemana sahip çok boyutlu dizi**

Dizilere bağlı diğer dizilerin aynı boyutta olma zorunluluğu yoktur.

### Örnek-3.36: CokBoyutluDiziler.java

```
public class CokBoyutluDiziler {  
    public static void main(String args[]) {  
        int ikiboy[][] = new int[3][];  
        ikiboy[0] = new int[2] ;  
        ikiboy[1] = new int[1] ;  
        ikiboy[2] = new int[3] ;  
        ikiboy[0][0] = 89 ;  
        ikiboy[0][1] = 32 ;  
        ikiboy[1][0] = 5 ;  
        // ikiboy[1][1]=3 ;// ! Hata !  
        ikiboy[2][0] = 55 ;  
        ikiboy[2][1] = 78 ;  
        ikiboy[2][2] = 49 ;  
    }  
}
```

Yukarıda verilen örnekte görüldüğü gibi, dizilere bağlı her farklı dizinin boyutları birbirinden farklı olmuştur. Şekil üzerinde ifade etmeye çalışırsak.



Şekil-3.10. Farklı elemana sahip çok boyutlu dizi

Çok boyutlu dizilerin içerisine sınıf tiplerinin yerleştirilmesi de olasıdır. Örneğin *String* sınıfı tipinde olan çok boyutlu bir dizi oluşturulabilir:

### Örnek-3.37: HayvanlarAlemi.java

```
public class HayvanlarAlemi {  
    String isimler[][][] ;  
    public HayvanlarAlemi() {  
        isimler = new String[2][2][3] ;  
        veriAta();  
    }  
    public void veriAta() {  
        isimler[0][0][0] = "aslan" ;  
        isimler[0][0][1] = "boz AyI" ;  
        isimler[0][0][2] = "ceylan";  
        isimler[0][1][0] = "deniz AnasI" ;  
        isimler[0][1][1] = "essek" ;  
        isimler[0][1][2] = "fare" ;  
        isimler[1][0][0] = "geyik" ;  
        isimler[1][0][1] = "hamsi" ;  
        isimler[1][0][2] = "inek" ;  
    }  
}
```

```

        isimler[1][1][0] = "japon baligi" ;
        isimler[1][1][1] = "kedi" ;
        isimler[1][1][2] = "lama" ;
        ekranaBas() ;
    }
    public void ekranaBas() {
        for (int x = 0 ; x < isimler.length ; x++) {
            for (int y = 0 ; y < isimler[x].length ; y++) {
                for (int z = 0 ; z < isimler[x][y].length ; z ++ ) {
                    System.out.println("isimler["+x+"]["+y+"]["+z+"] =" +
                                        isimler[x][y][z]);
                }
            }
        }
    }
    public static void main(String args[]) {
        HayvanlarAlemi ha = new HayvanlarAlemi();
    }
}

```

Bu örnekte *HayvanlarAlemi* nesnesinin oluşturulmasıyla olaylar tetiklenmiş olur. İster tek boyutlu olsun ister çok boyutlu olsun, diziler üzerinde işlem yapmak isteniyorsa, onların oluşturulması (*new* *anahtar kelimesi* ile) gerektiği daha önceden belirtilmişti... Bu örnekte olduğu gibi, dizi ilk oluşturulduğunda, dizi içerisindeki *String* tipindeki referansların ilk değeri *null*'dir. Uygulamanın sonucu aşağıdaki gibi olur:

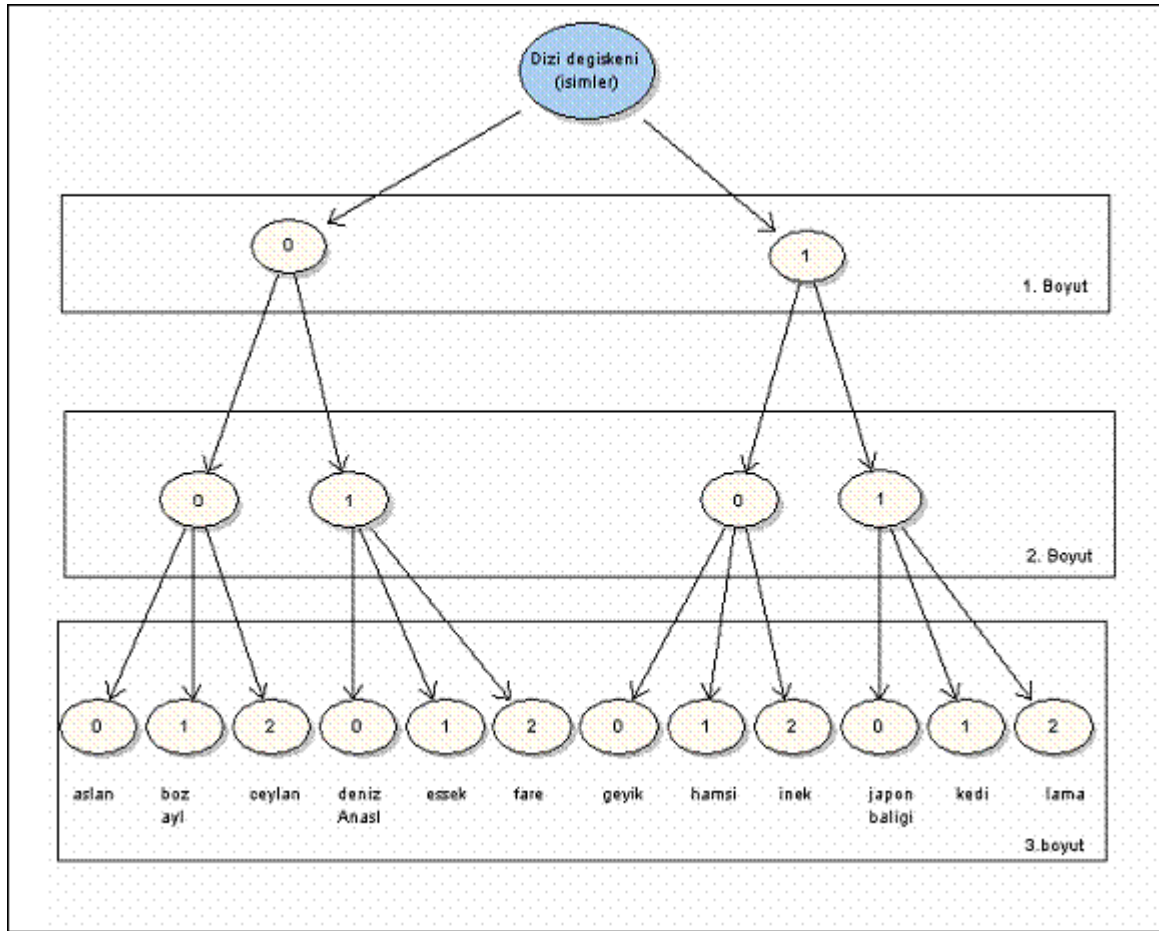
```

isimler[0][0][0] =aslan
isimler[0][0][1] =boz AyI
isimler[0][0][2] =ceylan
isimler[0][1][0] =deniz AnasI
isimler[0][1][1] =essek
isimler[0][1][2] =fare
isimler[1][0][0] =geyik
isimler[1][0][1] =hamsi
isimler[1][0][2] =inek
isimler[1][1][0] =japon baligi
isimler[1][1][1] =kedi
isimler[1][1][2] =lama

```

Oluşan olaylar şekilsel olarak gösterilmeye çalışılırsa:





Şekil-3.11. *String* sınıfı tipindeki çok boyutlu dizi

## **BÖLÜM 4**

### ***PAKET ERİŞİMLERİ***

Erişim konusu kütüphaneler için büyük önem taşır. Erişimde iki taraf bulunur; birisi kütüphaneyi kullanan kişiler (*client*), diğeri ise bu kütüphaneyi yazanlardır. Olaylara, hazır kütüphane kullanarak uygulama geliştiren tarafından bakılırsa, örneğin finans uygulaması yazan bir programcı, işlerin daha da hızlı yürütmesi için daha önceden yazılmış ve denenmiş bir finans kütüphanesini kullanmak isteyebilir. Finans için özel geliştirilmiş kütüphanenin 1.0 uyarlamasını kullanan tasarımcı, bunun yeni uyarlamaları (2.0, 3.0...) çıktığında da hemen alıp kendi uygulamasına entegre etmek istemesi çok doğaldır. Tahmin edilebileceği üzere her yeni uyarlama bir öncekine göre daha az hata içerir ve yeni özellikler sunar. Bu nedenden dolayı, kütüphane uyarlamaları arasındaki tutarlılık çok önemlidir. Sonuç olarak, kütüphanenin her yeni uyarlamasında sisteme entegre edilmesi aşamasında, bu kütüphaneyi kullanan uygulamaları teker teker değiştirmek yaşamı çekilmez kılabilir!

Olaylara, birde kütüphane tasarımcısı açısından bakılırsa... Bir kütüphane yazdınız ve bunun kullanılması için İnternet'e koydunuz. Aradan belirli bir zaman geçti ve sizin yazdığınız kütüphane birçok kişi tarafından kullanılmaya başladı... Fakat, daha sonra, kütüphane içerisinde bazı hatalar olduğunu fark ettiniz; veya, bazı kısımları, daha verimli çalışması için geliştirilmesini istiyorsunuz. Bu arzularınız, ilgili kütüphane üzerinde nasıl bir etki oluşturur? Kütüphaneyi kullanan kişiler bundan zarar görebilir mi veya zarar görmemesi için ne yapılması gerekir?

Böylesi sorunların çözümü için devreye erişim kavramı girer. Java dili 4 adet erişim belirleyicisi sunar. Erişim belirleyiciler, en erişilebilirden erişilmeze doğru sıralanırsa, `public`, `protected`, `friendly` ve `private`'dir. Bunlar sayesinde hem kütüphane tasarımcıları özgürlüklerine kavuşur hem de kütüphaneyi kullanan programcılar kullandıkları kütüphanenin yeni bir uyarlamalarını tasarımlarına kolayca entegre edebilirler.

Kütüphane tasarlayan kişiler, ileride değişebilecek olan sınıfları veya sınıflara ait yordamları, kullanıcı tarafından erişilmez yaparak hem kütüphanenin rahatça gelişimini sağlarlar hem de kütüphaneyi kullanan programcılarının endişelerini gidermiş olurlar.

#### **4.1. Paket (*Package*)**

Paketler kütüphaneyi oluşturan elemanlardır. Paket mantığının var olmasında ana nedenlerden birisi sınıf ismi karmaşasının getirmiş olduğu çözümdür. Örneğin elimizde *X* ve *Y* adlı 2 sınıf bulunsun. Bunlar içerisinde aynı isimli 2 yordam (*method*) olması, örneğin `f()` yordamı, herhangi bir karmaşıklığa neden olmayacaktır. Çünkü aynı isimdeki yordamlar ayrı sınıflarda bulunurlar. Peki sınıf isimleri? Sistemimizde bulunan aynı isimdeki sınıflar karmaşıklığa sebep vermez; eğer ki, aynı isimdeki sınıflar değişik paketlerin içerisinde bulunurlarsa...

#### **Gösterim-4.1:**

```
import java.io.BufferedReader;
```

Yukarıdaki gösterimde *BufferedReader* sınıf isminin *java.io* paketinde tek olduğunu anlıyoruz (*java.io* Java ile gelen standart bir pakettir). Fakat, başka paketlerin içerisinde *BufferedReader* sınıf ismi rahatlıkla kullanılabilir. Yukarıdaki gösterim *java.io* paketinin içerisinde bulunan *BufferedReader* sınıfını kullanacağını ifade etmektedir. Paketin içerisindeki tek bir sınıfı kullanmak yerine ilgili paketin içerisindeki tüm sınıfları tek seferde kullanmak için:

#### **Gösterim-4.2:**

```
import java.io.* ;
```

*java.io* paketi içerisindeki sınıfların uygulamalarda kullanılması için *import java.io.\** denilmesi yeterli olacaktır. Anlatılanlar uygulama içerisinde incelenirse,

#### **Örnek-4.1: *PaketKullanim.java***

```

import java.io.*;

public class PaketKullanım {

    public static void main(String args[]) throws
    IOException{
        BufferedReader sf = new BufferedReader(
                                new
        InputStreamReader(System.in));
        System.out.print("Bilgi Giriniz : ");
        String bilgi = sf.readLine();
        System.out.println("bilgi --> " + bilgi);
    }
}

```

*PaketKullanım.java* uygulamasında görmediğimiz yeni kavram mevcuttur, "throws Exception". Bu kavram istisnalar (*Exception*- 8. bölüm) konusunda detaylı bir şekilde incelenecektir.

#### 4.2. Varsayılan Paket (*Default Package*)

Öncelikle belirtmek gerekirse, *.java* uzantılı fiziksel dosya derlendiği zaman buna tam karşılık *.class* fiziksel dosyası elde edilir. (\**.java* dosyasında hata olmadığı varsayılırsa). Fiziksel *.java* dosyasında birden fazla sınıf tanımlanmış ise, tanımlanan her sınıf için ayrı ayrı fiziksel *.class* dosyaları üretilir.

##### Örnek-4.2: *Test1.java*

```

public class Test1 {

    public void kos() {
    }
}

class Test2 {

    public void kos() {
    }
}

```

Yukarıda verilen örnek, *Test1.java* adıyla herhangi dizine kayıt edilebilir (fiziksel *.java* uzantılı dosya ile *public* sınıfın ismi birebir aynı olmalıdır). Bu dosya *javac* komutu ile derlendiğinde adları *Test1.class* ve *Test2.class* olan 2 adet fiziksel *.class* dosyası elde edilir. *Test1.java* dosyanın en üstüne herhangi bir paket ibaresi yerleştirilmediğinden dolayı Java bu sınıfları varsayılan paket (*default package*) olarak algılayacaktır.

##### Örnek-4.3: *Test3.java*

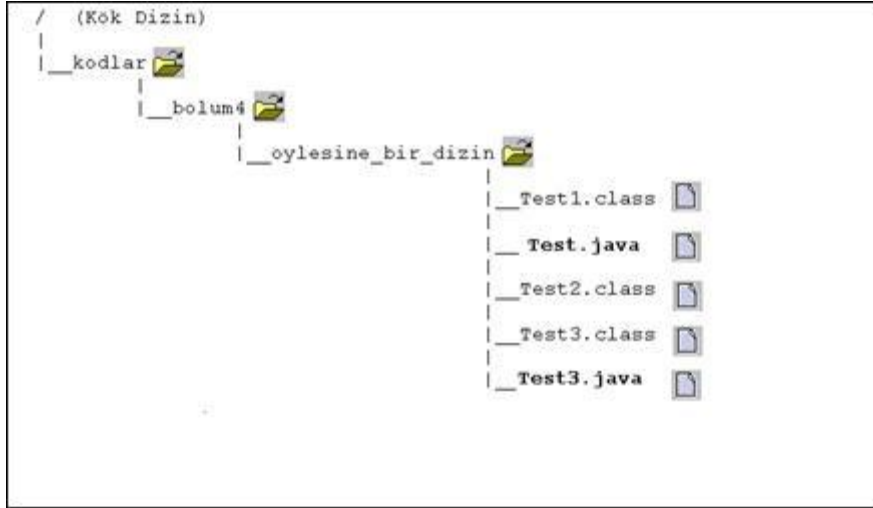
```

public class Test3 {

    public void kos() { }
}

```

Benzer şekilde verilen örnekte, aynı dizine *Test3.java* adıyla kayıt edilebilir; derleme (*compile*) işleminden sonra genel ifade Şekil-4.1.'de gösterildiği gibi olur:



**Şekil-4.1. Varsayılan paket**

#### 4.3. Paket Oluşturma

Kendi paketlerimizi oluşturmanın temel amaçlarından birisi, aynı amaca yönelik iş yapan sınıfları bir çatı altında toplamaktır; böylece yazılan sınıflar daha derli toplu olurlar. Ayrıca aranan sınıflar daha kolay bulunabilir. Peki eğer kendimiz paket oluşturmak istersek, bunu nasıl başaracağız?

##### **Örnek-4.4:** *Test1.java*

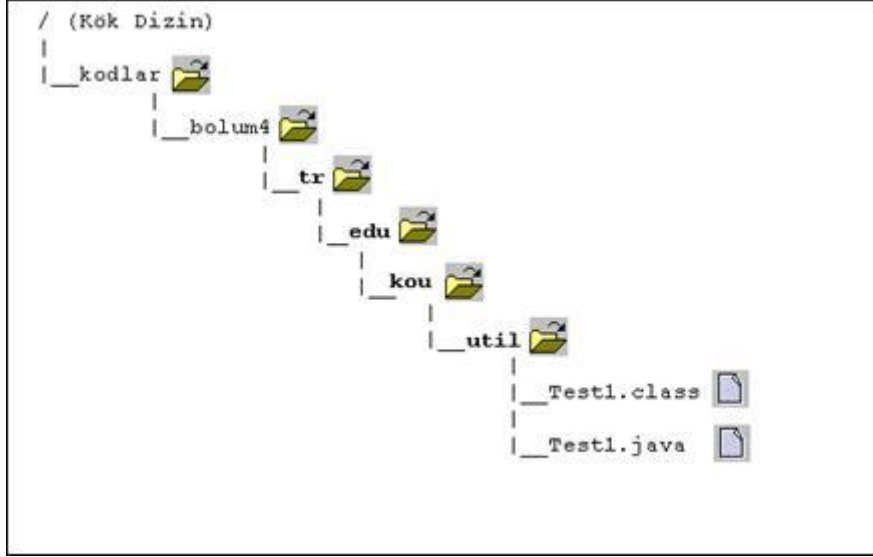
```
package tr.edu.kou.util ;

public class Test1 {

    public Test1() {
        System.out.println("tr.edu.kou.util.Test1
nesnesi" +
"olusturuluyor");
    }

    public static void main(String args[]) {
        Test1 pb = new Test1();
    }
}
```

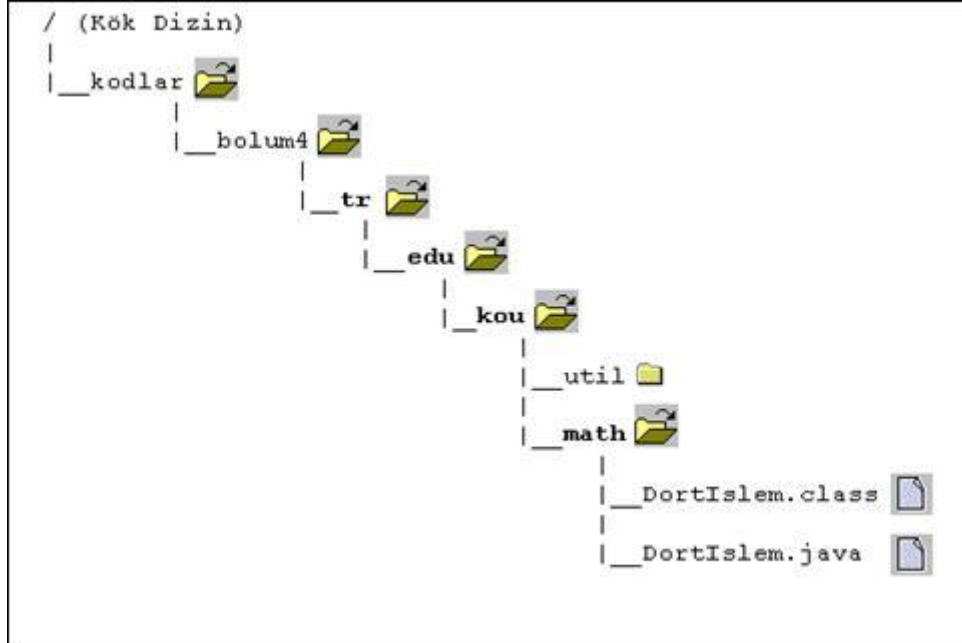
Bu örneğimizde, *Test1.java* dosyası işletim sisteminin herhangi bir dizinine yerleştirilemez; çünkü, o artık *tr.edu.kou.util* paketine ait bir sınıftır. Bundan dolayı *Test1.java* dosyası işletim sisteminin bu paket ismiyle paralel olan bir dizin yapısına kayıt edilmesi gerekir. Önemli diğer bir unsur ise sınıfın ismidir; *Test1.java* dosyası içerisinde belirtilen *Test1* sınıfının ismi artık *tr.kou.edu.util.Test1*'dir.



**Şekil-4.2. *tr.edu.kou.util.Test1* sınıfı**

Paket isimleri için kullanılan yapı İnternet alan isim sistemiyle (*Internet DNS*) aynıdır. Örneğin, Kocaeli Üniversitesinde matematik paketi geliştiren tasarımcı, bu paketin içerisindeki sınıfların, başka kütüphane paketleri içerisindeki sınıf isimleriyle çakışmaması için İnternet alan adı sistemini kullanmalıdır. İnternet alan adı sistemi, <http://www.kou.edu.tr/> adresinin dünya üzerinde tek olacağını garantiler. Aynı mantık, paket isimlerine de uygulanarak, paket içerisindeki sınıf isimleri çakışma sorununa çözüm bulunmuş olunuyor. Dikkat edilirse, paket isimleri İnternet alan adlarının tersten yazılmış halleridir.

İşletim sistemleri farkı gözönüne alınarak, *math* paketine ait sınıfların içinde bulunması gereken dizin Unix veya Linux için *tr/edu/kou/math*, Windows için *tr\edu\kou\math* şekilde olmalıdır.



**Şekil-4.3. *tr.edu.kou.math.DortIslem* sınıfı**

#### 4.4. CLASSPATH Ayarları

Java yorumlayıcısı (*interpreter*) işletim sistemi çevre değişkenlerinden CLASSPATH'e bakarak import ifadesindeki paketi bulmaya çalışır...

#### **Gösterim-4.3:**

```
import tr.edu.kou.math.*;
```

Diyelim ki; *math* paketi aşağıdaki dizinde bulunsun:

#### **Gösterim-4.4:**

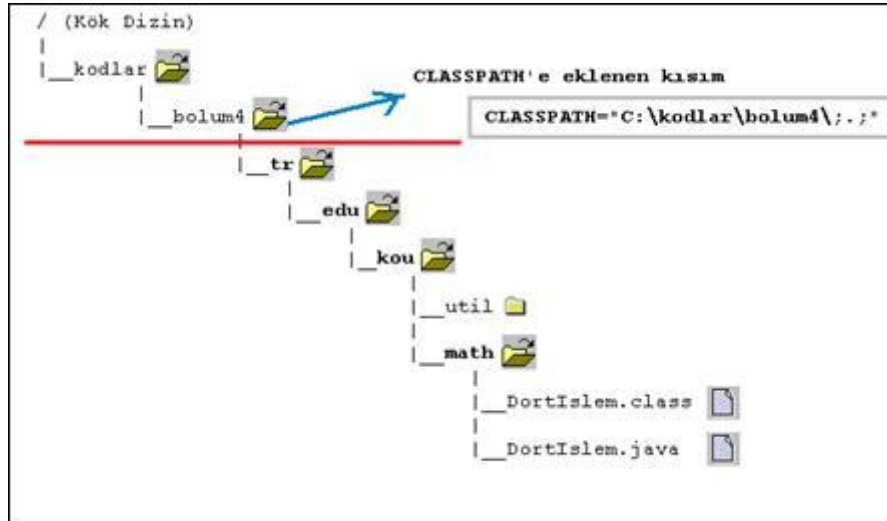
```
C:\kodlar\bolum4\tr\edu\kou\math\
```

Yorumlayıcının import ifadesindeki paketi bulması için aşağıdaki tanımın CLASSPATH'e eklenmesi gerekir.

#### **Gösterim-4.5:**

```
CLASSPATH="C:\kodlar\bolum4\;."
```

CLASSPATH'e eklenen bu tanım sayesinde yorumlayıcı *C:\kodlar\bolum4\* dizinine kadar gidip *tr\edu\kou\math* dizinini arayacaktır; bulunursa, bu dizinin altındaki tüm sınıflar uygulama tarafından kullanılabilir hale gelecektir.



**Şekil-4.4. CLASSPATH ayarları**

*DortIslem.java* dosyasını *C:\kodlar\bolum4\tr\edu\kou\math* dizinine yerleştirelim:

#### **Örnek-4.5: DortIslem.java**

```
package tr.edu.kou.math ;
public class DortIslem {

    public static double topla(double a, double b) {
        return (a + b) ;
    }
}
```

```

public static double cikart(double a, double b) {
    return (a - b) ;
}

public static double carp(double a, double b) {
    return (a * b) ;
}

public static double bol(double a, double b) {
    if ( (a != 0) && (b != 0) ) {
        return (a / b);
    } else {
        return 0;
    }
}
}

```

Gerekli CLASSPATH tanımları yapıldıktan sonra bu paketi kullanacak olan uygulama herhangi bir dizinine yerleştirebilir.

#### Örnek-4.6: Hesaplama.java

```

import tr.edu.kou.math.*;      //dikkat!

public class Hesaplama {

    public static void main(String args[]) {

        double sonuc = DortIslem.topla(9.6 , 8.7);
        System.out.println("9.6 + 8.7 = " + sonuc );

        Sonuc = DortIslem.cikart(9.6 , 8.7);
        System.out.println("9.6 - 8.7 = " + sonuc );

        Sonuc = DortIslem.carp(5.6 , 8.7);
        System.out.println("9.6 * 8.7 = " + sonuc );

        Sonuc = DortIslem.bol(5.6 , 8.7);
        System.out.println("9.6 / 8.7 = " + sonuc );

    }
}

```

Uygulamanın sonucu aşağıdaki gibi olur:

```

9.6 + 8.7 = 18.299999999999997
9.6 - 8.7 = 0.90000000000000004
9.6 * 8.7 = 48.719999999999999
9.6 / 8.7 = 0.6436781609195402

```

#### 4.4.1. Önemli Nokta

Dikkat edilirse CLASSPATH değişkenine değer atanırken en sona "." nokta koyuldu.



**Şekil-4.5. Noktanın Önemi**

Bu önemli bir ayrıntıdır. Bu noktanın konmasındaki neden varsayılan paketlerin içindeki sınıfların birbirlerini görebilmesini sağlamaktır; unutulursa, anlamsız hata mesajlarıyla karşılaşılabilir.

Java'yı sisteme ilk yüklediği zaman, basit örneklerin, CLASSPATH değişkenine herhangi bir tanım eklemeyen bile çalıştırabildiği görülür; nedeni, Java'nın, temel kütüphanelerinin bilinmesindendir.

#### 4.5. Çakışma

Aynı paket içerisinde aynı isimdeki sınıflar uygulamada kullanılırsa ne olur? Adları aynı olsa bile değişik paketlerde bulundukları için bir sorun yaşanmaması gerekecektir. Öncelikle *tr.edu.kou.util* paketinin içerisine kendi *ArrayList* sınıfımızı oluşturalım:

**Örnek-4.7:** *ArrayList.java*

```
package tr.edu.kou.util;

public class ArrayList {

    public ArrayList() {
        System.out.println("tr.edu.kou.util.ArrayList
nesnesi" +
                                "
olusturuluyor");
    }
}
```

Aşağıdaki örneği işletim sisteminin herhangi bir dizinine kayıt edebiliriz.

**Örnek-4.8:** *Cakisma.java*

```
import java.util.*;
import tr.edu.kou.util.*;

public class Cakisma {

    public static void main(String args[]) {
        System.out.println("Baslagic..");
        ArrayList al = new ArrayList();
        System.out.println("Bitis..");
    }
}
```

*Cakisma.java* dosyası *javac* komutu ile derlendiğinde şu hata mesajıyla karşılaşılır:



```
Cakisma.java:8: reference to ArrayList is ambiguous,
both class tr.edu.kou.util.
ArrayList in tr.edu.kou.util and class
java.util.ArrayList in java.util match
ArrayList al = new ArrayList();
^
Cakisma.java:8: reference to ArrayList is ambiguous,
both class tr.edu.kou.util.

ArrayList in tr.edu.kou.util and class
java.util.ArrayList in java.util match
ArrayList al = new ArrayList();
^
2 errors
```

Bu hata mesajı, *ArrayList*'in hem *java.util* paketinde hem de *tr.edu.kou.util* paketinde bulunmasından kaynaklanan bir ikilemi göstermektedir. *Cakisma* sınıfının içerisinde *ArrayList* sınıfı kullanılmıştır; ancak, hangi paketin içerisindeki *ArrayList* sınıfı? Bu sorunu çözmek için aşağıdaki örneği inceleyelim:

#### Örnek-4.9: *Cakisma2.java*

```
import java.util.*;
import tr.edu.kou.util.*;

public class Cakisma2 {

    public static void main(String args[]) {
        System.out.println("Baslagic..");
        tr.edu.kou.util.ArrayList al = new
tr.edu.kou.util.ArrayList();
        System.out.println("Bitis..");
    }
}
```

Eğer ortada böyle bir ikilem varsa, gerçekten hangi sınıfı kullanmak istiyorsanız, o sınıfın içinde bulunduğu paket ismini de açık bir biçimde yazarak oluşan bu ikilemi ortadan kaldırabilirsiniz.

#### 4.6. **Paket İçerisindeki Tek Başına Yürütülebilir Uygulamaları (Standalone) Çalıştırmak**

Paket içerisindeki tek başına çalışabilen uygulamaları (*standalone*) herhangi bir dizin içerisinde çalışmak için komut satırına, ilgili “paket ismi+sınıf ismi” girilmesi yeterlidir. *Hesaplama.java*'nın yeni uyarlamasını *C:\kodlar\bolum4\tr\edu\kou\math* dizinine kaydedelim.

#### Örnek-4.10: *Hesaplama.java*

```
package tr.edu.kou.math ;    // dikkat!

public class Hesaplama {

    public static void main(String args[]) {

        double sonuc = DortIslem.topla(9.6 , 8.7);
```

```

        System.out.println("9.6 + 8.7 = " + sonuc );

        sonuc = DortIslem.cikart(9.6 , 8.7);
        System.out.println("9.6 - 8.7 = " + sonuc );

        sonuc = DortIslem.carp(5.6 , 8.7);
        System.out.println("9.6 * 8.7 = " + sonuc );

        sonuc = DortIslem.bol(5.6 , 8.7);
        System.out.println("9.6 / 8.7 = " + sonuc );
    }
}

```

Artık *Hesaplama* sınıfımız *tr.edu.kou.math* paketinin yeni bir üyesidir. *Hesaplama* sınıfımızı **java** komutu kullanarak çalıştırmayı deneyelim. *C:\kodlar\bolum4* dizininin CLASSPATH değişkeninde tanımlı olduğunu varsayıyorum.

#### **Gösterim-4.6:**

```
java Hesaplama
```

Bir aksilik var! Ekranaya yazılan hata mesajı aşağıdaki gibidir:

```

Exception in thread "main" java.lang.NoClassDefFoundError:
Hesaplama (wrong name: tr/edu/kou/math/Hesaplama)
    at java.lang.ClassLoader.defineClass0(Native Method)
    at
java.lang.ClassLoader.defineClass(ClassLoader.java:509)
    at java.security.SecureClassLoader.defineClass
(SecureClassLoader.java:123)
    at java.net.URLClassLoader.defineClass
(URLClassLoader.java:246)
    at
java.net.URLClassLoader.access$100(URLClassLoader.java:54)
    at
java.net.URLClassLoader$1.run(URLClassLoader.java:193)
    at java.security.AccessController.doPrivileged(Native
Method)
    at
java.net.URLClassLoader.findClass(URLClassLoader.java:186)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:306)
    at sun.misc.Launcher$AppClassLoader.loadClass
(Launcher.java:265)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:262)
    at java.lang.ClassLoader.loadClassInternal
(ClassLoader.java:322)

```

*Hesaplama* sınıfı bulunamıyor diye bir hata mesajı? Nasıl olur ama orada... Evet orada; ama, o artık *Hesaplama* sınıfı değildir; *tr.edu.kou.math.Hesaplama* sınıfıdır. Yani, artık bir paketin üyesi olmuştur. Aşağıdaki ifade sayesinde herhangi bir dizinden (tabii CLASSPATH değişkeninin değeri doğru tanımlanmış ise) *tr.edu.kou.math.Hesaplama* sınıfına ulaşip onu **java** komutu ile çalıştırabiliriz.

#### Gösterim-4.7:

```
java tr.edu.kou.math.Hesaplama
```

Uygulamanın çıktısı aşağıdaki gibidir.

```
9.6 + 8.7 = 18.299999999999997
9.6 - 8.7 = 0.90000000000000004
9.6 * 8.7 = 48.719999999999999
9.6 / 8.7 = 0.6436781609195402
```

#### 4.7. JAR Dosyaları (*The Java™ Archive File*)

JAR dosya formatı dosyaların arşivlenmesine ve sıkıştırılmasına olanak tanır. Olağan durumunda JAR dosyaları içerisinde sınıf dosyaları (\*.class) bulunur; bazen, özellikle *Appletler* de, yardımcı dosyalar da (gif, jpeg...) JAR dosyası içerisine konulabilir.

JAR dosyasının sağladığı yararlar şöyledir:

- **Güvenlik:** Dijital olarak JAR dosyasının içeriğini imzalayabilirsiniz. Böylece sizin imzanızı tanıyan kişiler JAR dosyasının içeriğini rahatlıkla kullanabilirler.
- **Sıkıştırma:** Bir çok dosyayı güvenli bir şekilde arşivleyip sıkıştırılabilir.
- **İndirme (download) zamanını azaltması:** Arşivlenmiş ve sıkıştırılmış dosyalar internet üzerinde daha çabuk indirilebilir.
- **Paket mühürleme (versiyon 1.2):** Versiyon uyumluluğunu sağlamak amacı ile JAR dosyasının içerisindeki paketler mühürlenebilir. JAR dosyasının içerisinde paket mühürlemekten kasıt edilen paket içerisinde bulunan sınıfların aynı JAR dosyasında bulunmasıdır.
- **Paket uyarılama (versiyon 1.2):** JAR dosyaları, içindeki dosyalar hakkında bilgiler saklayabilirler, örneğin üretici firmaya ait bilgiler, versiyon bilgileri gibi.
- **Taşınabilirlik:** Java Platformunun standart bir üyesi olan JAR dosyaları kolaylıkla taşınabilir.

Oluşturulan paketler JAR dosyası içerisine yerleştirilerek daha derli toplu bir görüntü elde etmiş olunur; *tr.edu.kou.math* ve *tr.edu.kou.util* paketlerini tek bir JAR dosyasında birleştirmek için Gösterim-4.8’de verilen komutun kullanılması yeterli olur; ancak, JAR dosyası oluşturmak için komutun hangi dizinde yürütüldüğü önemlidir. Tablo-4.1’de JAR dosyası işlemleri için gerekli olan bazı komutlar verilmiştir:

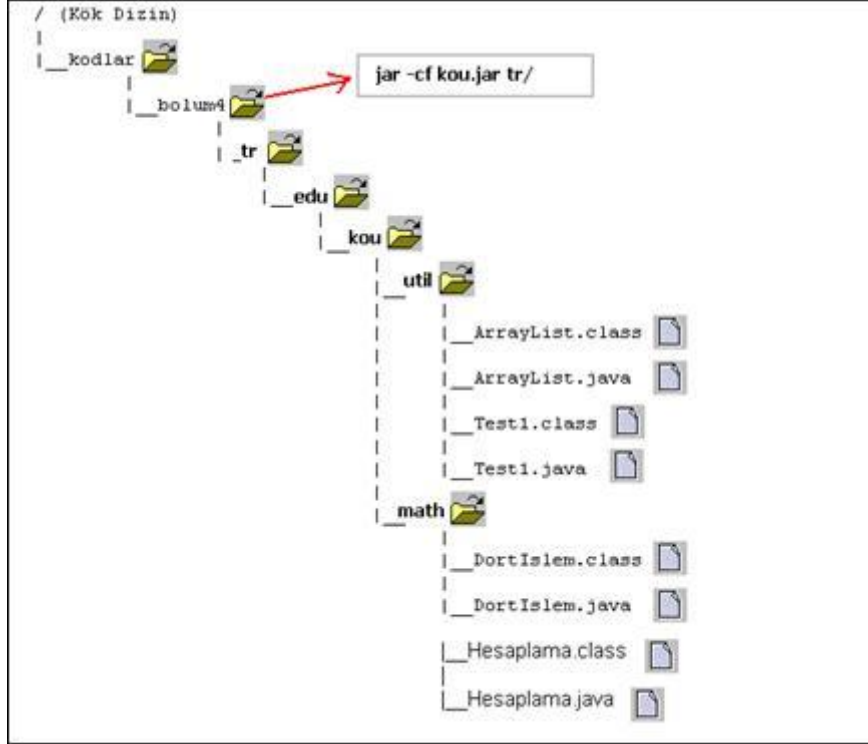
**Tablo-4.1. JAR dosyaları**

| Açıklama  | Komut   |
|---|---|
| JAR dosyası oluşturmak için                             | jar -cf <i>jar-dosya-ismi içeriye-atılacak-dosya(lar)</i> |
| JAR dosyasının içeriği bakmak için                      | jar -tf <i>jar-dosya-ismi</i>                             |
| JAR dosyasının içeriği toptan dışarı çıkartmak için     | jar -xf <i>jar-dosya-ismi</i>                             |
| Belli bir dosyayı JAR dosyasından dışarı çıkartmak için | jar -xf <i>jar-dosya-ismi arşivlenmiş dosya(lar)</i>      |
| JAR olarak paketlenmiş uygulamayı çalıştırmak için      | java -classpath <i>jar-dosya-ismi MainClass</i>           |

#### Gösterim-4.8:

```
jar -cf kou.jar tr/
```

Gösterim-4.8'deki ifadeyi *C:\kodlar\bolum4* dizinin içerisinde iken çalıştırmalıyız ki, JAR dosyasının içerisine doğru yapıdaki dizinleri yerleştirelim.



**Şekil-4.6. Dizinlerin JAR dosyasına atılmasına**

Oluşmuş olan bu JAR dosyasını CLASSPATH ekleyerek, Java'nın bu paketleri bulmasını sağlayabilirsiniz. Aşağıdaki ifade yerine:

#### **Gösterim-4.9:**

```
CLASSPATH="C:\kodlar\bolum4\;."
```

Artık Gösterim-4.10'daki ifade kullanılabilir; *kou.jar* dosyası, *C:\kodlar\bolum4*'nin altındaki dizin yapılarının aynısını kendi içerisinde barındırır. Bu nedenle *kou.jar* dosyası en alakasız dizine kopyalanabilir; ancak, tek bir koşulu unutmamak kaydıyla... Bu koşul da *kou.jar* dosyası sistemin CLASSPATH değişkeninin de tanımlı olmasıdır.

#### **Gösterim-4.10:**

```
CLASSPATH="C:\muzik\kou.jar;."
```

Java, CLASSPATH değerlerinden yola çıkarak JAR dosyasını bulup açar; *tr\edu\kou\util* ve *tr\edu\kou\math* dizinlerine erişebileceğinden bir sorun yaşanmayacaktır. Yani, JAR dosyasının hangi dizinde olduğu önemli değildir, önemli olan ilgili *jar* dosyasının sistemin CLASSPATH değişkeninin tanımlı olmasıdır. Tabii, paketlerin içerisindeki sınıflar geliştikçe güncelliği korumak adına JAR dosyasını tekrardan oluşturmak (*jar -cvf.....*) gerekebilir.

#### 4.7.1. JAR Dosyası İçerisindeki Bir Uygulamayı Çalıştırmak

JAR dosyası içeriğini dışarı çıkartılmadan tek başına çalışabilir (*standalone*) java uygulamalarını yürütmek olasıdır. Örneğin, JAR dosyası içerisinde *tr.edu.kou.math* paketi altındaki *Hesaplama* sınıfı çalıştırılsın,

##### Gösterim-4.11:

```
> java -classpath C:\muzik\kou.jar  
tr.edu.kou.math.Hesaplama
```

Eğer *kou.jar* dosyası, sistemin CLASSPATH değişkeninde tanımlı değilse ve CLASSPATH ayarlarıyla uğraşılacak istenmiyorsa, *java* komutuna kullanılacak JAR dosyası adresi tam olarak *-classpath* parametresiyle birlikte verilebilir. Daha sonra, ilgili JAR dosyasındaki hangi sınıf çalıştırılmak isteniyorsa, bu dosya düzgün bir biçimde yazılmalıdır. Gösterim-4.11'deki komutun oluşturacağı ekran sonucu aşağıdaki gibi olur:

```
9.6 + 8.7 = 18.299999999999997  
9.6 - 8.7 = 0.90000000000000004  
9.6 * 8.7 = 48.719999999999999  
9.6 / 8.7 = 0.6436781609195402
```

#### 4.8. Erişim Belirleyiciler

Java dilinde 4 tür erişim belirleyicisi vardır; bunlar *friendly*, *public*, *protected* ve *private*'dir. Bu erişim belirleyiciler global alanlar (statik veya değil) ve yordamlar (statik veya değil) için kullanılabilir. Ayrıca sınıflar içinde (dahili sınıflar hariç *-inner class*) sadece *public* ve *friendly* erişim belirleyicilerini kullanılabilir.

##### 4.8.1. **friendly**

*friendly* erişim belirleyicisi global alanlara (statik veya değil), yordamlara (statik veya değil) ve sınıflara atanabilir. *friendly* türünde erişim belirleyicisine sahip olan global alanlar (statik veya değil) içerisinde bulundukları paketin diğer sınıfları tarafından erişilebilirler. Fakat, diğer paketlerin içerisindeki sınıflar tarafından erişilemezler. Yani, diğer paketlerin içerisindeki sınıflara karşı *private* erişim belirleyici etkisi oluşturmuş olurlar.

*friendly* yordamlarda, yalnız, paketin kendi içerisindeki diğer sınıflar tarafından erişilebilirler. Diğer paketlerin içerisindeki sınıflar tarafından erişilemezler. Aynı şekilde, sınıflara da *friendly* erişim belirleyicisi atayabiliriz, böylece *friendly* erişim belirleyicisine sahip bu sınıfa, aynı paket içerisindeki diğer sınıflar tarafından erişilebilir; ancak, diğer paketlerin içerisindeki sınıflar tarafından erişilemezler.

Şimdi, *tr\edu\kou\* dizini altına yeni bir dizin oluşturalım; ve, ismini *gerekli* verelim. Yani *tr\edu\kou\gerekli* paketini oluşturmuş olduk; bunun içerisine adları *Robot* ve *Profesor* olan 2 adet *friendly* sınıf yazalım:

##### Örnek-4.11: *Robot.java*

```
package tr.edu.kou.gerekli;  
class Robot {  
  
    int calisma sure = 0;
```

```
String renk = "beyaz";
int motor_gucu = 120;

Robot() {
    System.out.println("Robot olusturuluyor");
}
}
```

#### **Örnek-4.12:** *Profesor.java*

```
package tr.edu.kou.gerekli;

class Profesor {
    public void kullan() {
        Robot upuaut = new Robot(); // sorunsuz
    }
}
```

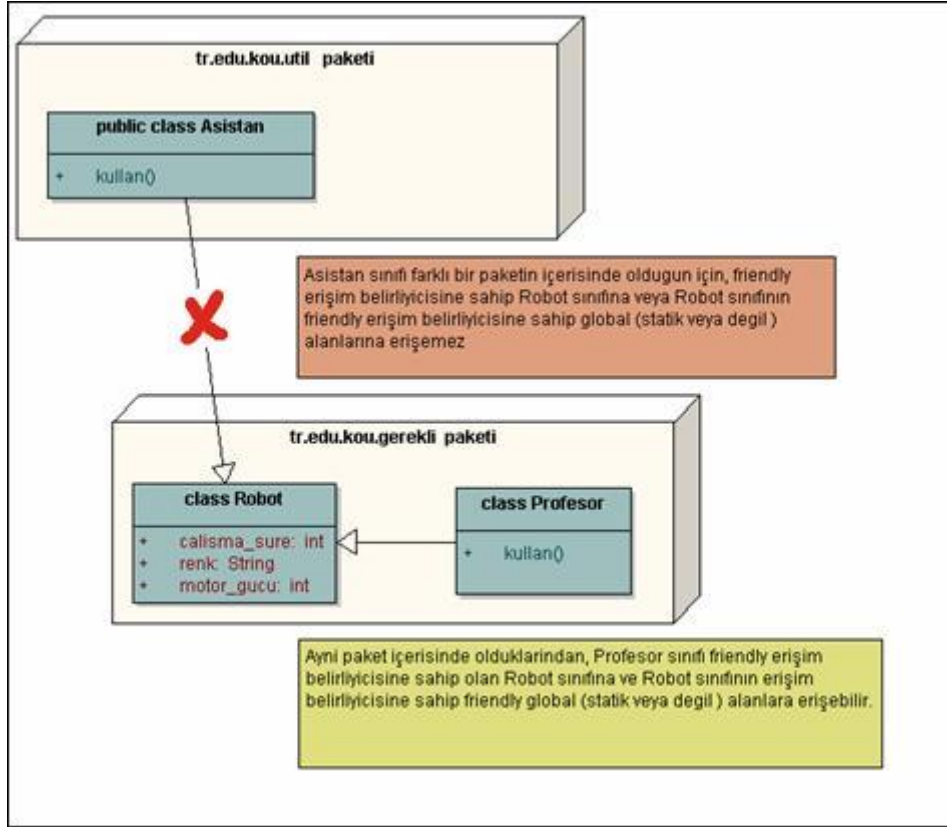
Verilen örneklerden anlaşılacağı gibi, bir global alan veya sınıf *friendly* yapılmak isteniyorsa önüne hiç bir erişim belirleyicisi konulmaz. Şimdi, bu iki sınıf aynı paketin içerisinde olduklarına göre *Profesor* sınıfı rahatlıkla *Robot* sınıfına erişebilecektir. Peki başka bir paket içerisindeki sınıf *Robot* sınıfına erişebilir mi?

#### **Örnek-4.13:** *Asistan.java*

```
package tr.edu.kou.util;
import tr.edu.kou.gerekli.*;
public class Asistan {
    public void arastir() {
        System.out.println("Asistan arastiriyor");
    }

    public void kullan() {
        //Robot upuaut = new Robot(); Hata! erişemez
    }
}
```

Daha önce belirtildiği gibi, *friendly* erişim belirleyicisine sahip olan global alanlara veya sınıflara yalnızca içerisinde bulunduğu paketin diğer sınıfları tarafından erişilebilirdi. Diğer paketlerin içerisindeki sınıflar tarafından erişilemezler. Yukarıdaki örneğimizde, *Asistan* sınıfı *tr.edu.kou.util* paketi altında tanımlandığı için *tr.edu.kou.gerekli* paketi altında tanımlı olan *Robot* sınıfına hiç bir şekilde erişemez. Anlatılanlar Şekil-4.7’de çizimsel olarak gösterilmeye çalışılmıştır.



**Şekil-4.7. friendly erişim belirleyicisinin etkisi**

#### 4.8.1.1. Varsayılan Paketlerde (Default Package) Erişim

Aşağıdaki 2 sınıfı herhangi bir dizini kaydedelim:

##### **Örnek-4.14:** *AltKomsu.java*

```
class AltKomsu {
    public static void main(String[] args) {
        UstKomsu uk = new UstKomsu();
        uk.merhaba();
    }
}
```

##### **Örnek-4.15:** *UstKomsu.java*

```
class UstKomsu {
    void merhaba() {
        System.out.println("Merhaba");
    }
}
```

Bu iki sınıf friendly erişim belirleyicisine sahiptir yani aynı paketin içerisindeki sınıflar tarafından erişilebilirler ama ortada herhangi bir paket ibaresi bulunmamaktadır. Aynı dizinde olan fakat bir paket olarak tanımlanmamış sınıflar, Java tarafından varsayılan paket çatısı altında toplanmaktadır. Bu iki sınıfın

birbirini görebilmesi için CLASSPATH değişkeninin değerinde "." (nokta) ibaresinin olması şarttır (bkz: Şekil-4.5.1 önemli nokta).

#### 4.8.2. **public** (Herkese Açık)

public erişim belirleyicisi sahip olabilen sınıflar, global alanlar ve yordamlar herkes tarafından erişilebilir. Bu erişim belirleyicisi yerleştirilmeden önce iki kez düşünülmelidir! Bu erişim belirleyicisine sahip olan global alanlar veya yordamlar herhangi bir yerden doğrudan çağrılabilirdiklerinden dolayı dış dünya ile arasındaki arabirim rolünü üstlenirler.

##### **Örnek-4.16:** *Makine.java*

```
package tr.edu.kou.util;

public class Makine {

    int devir_sayisi;
    public String model = "2002 model" ;

    public int degerAl() {
        return devir_sayisi;
    }

    public void degerAta(int deger) {
        this.devir_sayisi = deger;
        calis();
    }

    void calis() {
        for (int i = 0 ; i < devir_sayisi ; i++) {
            System.out.println("Calisiyor, devir_sayisi = " + i);
        }
    }
}
```

*tr.edu.kou.util* paketinin içerisindeki *Makine* sınıfının 2 adet global alanı bulunmaktadır; bunlardan *int* türündeki *devir\_sayisi* alanı friendly erişim belirleyicisine sahiptir. Yani, sadece *tr.edu.kou.util* paketinin içerisindeki diğer sınıflar tarafından doğrudan erişilebilir. Diğer *String* tipindeki *model* alanı ise her yerden erişilebilir. Çünkü *public* erişim belirleyicisine sahiptir. *degerAl()* yordamı *public* erişim belirleyicisine sahiptir yani her yerden erişilebilir. Aynı şekilde *degerAta(int deger)* yordamı da her yerden erişilebilir; ancak, *calis()* yordamı friendly belirleyicisine sahip olduğundan sadece *tr.edu.kou.util* paketinin içerisindeki sınıflar tarafından erişilebilir.

##### **Örnek-4.17:** *UstaBasi.java*

```
import tr.edu.kou.util.*;

public class UstaBasi {

    public UstaBasi() {
        Makine m = new Makine();
        // int devir_sayisi = m.devir_sayisi ; ! Hata ! erişemez
        m.degerAta(6) ;
    }
}
```



```

        int devir_sayisi = m.degerAl();
        String model = m.model;
        // m.calis() ; !Hata! erişemez
    }
}

```

Yukarıdaki uygulamada *tr.edu.kou.util* paketinin altındaki tüm sınıflar kullanılmak istendiği belirtilmiştir. *Ustabasi* sınıfının yapılandırıcısında *public* erişim belirleyicisine sahip olan *Makine* sınıfına ait bir nesne oluşturulabilmesine karşın, bu nesnenin **friendly** erişim belirleyicisine sahip olan *devir\_sayisi* alanına ve *calis()* yordamına erişilemez. Çünkü, *Ustabasi* sınıfı *tr.edu.kou.util* paketinin içerisinde değildir.

#### 4.8.3. **private (Özel)**

*private* olan global alanlara veya yordamlara (sınıflar *private* olamazlar; dahili sınıflar-*inner class* hariç) aynı paket içerisinden veya farklı paketlerden erişilemez. Ancak, ait olduğu sınıfın içinden erişilebilir. *private* belirleyicisine sahip olan yordamların içerisinde devamlı değişebilecek/geliştirilebilecek olan kodlar yazılmalıdır.

#### **Örnek-4.18:** *Kahve.java*

```

package tr.edu.kou.gerekli;

class Kahve {
    private int siparis_sayisi;
    private Kahve() {
    }

    private void kahveHazirla() {
        System.out.println(siparis_sayisi + " adet
kahve hazırlandı");
    }

    public static Kahve siparisGarson(int sayi) {
        Kahve kahve = new Kahve(); //dikkat
        kahve.siparis_sayisi = sayi ;
        kahve.kahveHazirla();
        return kahve;
    }
}

```

#### **Örnek-4.19:** *Musteri.java*

```

package tr.edu.kou.gerekli;

public class Musteri {

    public static void main(String args[]) {
        // Kahve kh = new Kahve() ;    // Hata !
        // kh.kahveHazirla() ;          // Hata !
        // kh.siparis_sayisi = 5 ;      // Hata !
    }
}

```

```

        Kahve kh = Kahve.siparisGarson(5);
    }
}

```

*Kahve* sınıfının yapılandırıcısı (*constructor*) *private* olarak tanımlanmıştır. Bundan dolayı herhangi bir başka sınıf, *Kahve* sınıfının yapılandırıcısını doğrudan çağırabilir; aynı paketin içerisinde olsa bile... Ancak, bu *private* yapılandırıcı aynı sınıfın içerisindeki yordamlar tarafından rahatlıkla çağırılabilir (*Kahve* sınıfının statik *siparisGarson()* yordamına dikkat ediniz). Aynı şekilde *private* olarak tanımlanmış global alanlara veya yordamlara aynı paket içerisinde olsun veya olmasın, kesinlikle erişilemez. Anlatılanlar Şekil-4.8’de çizimsel olarak gösterilmeye çalışılmıştır.

#### 4.8.4. **protected** (Korumalı Erişim)

Sadece global alanlar ve yordamlar *protected* erişim belirleyicisine sahip olabilirler. Sınıflar *protected* erişim belirleyicisine sahip olmazlar (dahili sınıflar-inner class hariç); ancak, sınıflar *friendly* veya *public* erişim belirleyicisine sahip olabilirler. *protected* erişim belirleyicisi kalıtım (*inheritance*) konusu ile sıkı sıkıya bağlıdır. Kalıtım konusunu bir sonraki ayırtta ele alınmıştır. Kalıtım konusu hakkında kısaca, ana sınıftan diğer sınıfların türemesi denilebilir.

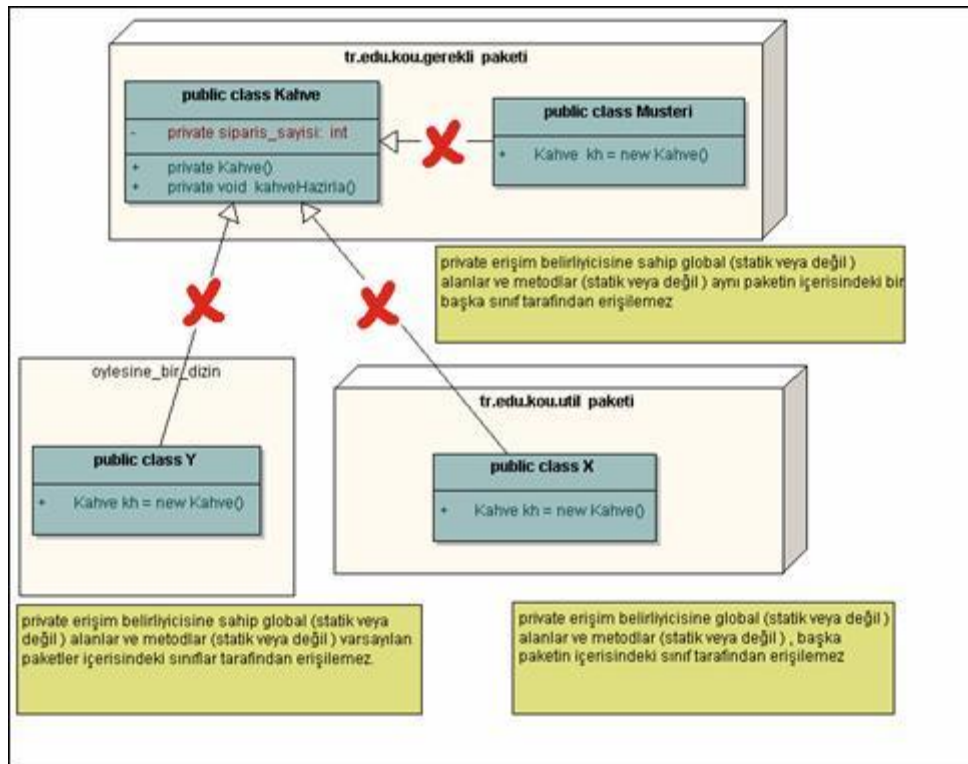
#### Gösterim-4.12:

```

class Kedi extends Hayvan

```

Yukarıda şu ifade edilmiştir: Her Kedi bir Hayvandır. Yani *Hayvan* sınıfından *Kedi* üretildi; bizim oluşturacağımız her *Kedi* nesnesi bir *Hayvan* olacaktır. Ancak, kendisine has kedisel özellikler de taşıyacaktır.



Şekil-4.8. private erişim belirleyicisi

protected erişim belirleyicisini evdeki buzdolabımızın kapısına vurulmuş bir kilit olarakda düşünebiliriz. Örneğin evimizde bir buzdolabı var ve içinde her türlü yiyecek ve içecek mevcut. Biz aile büyüğü olarak bu buzdolabına herkesin erişmesini istemiyoruz. (public yaparsak) aksi takdirde yiyecek ve içecek kısa sürede bitip ailenin aç kalma tehlikesi oluşacaktır, aynı şekilde bu buzdolabına erişimi tamamen kesersek de aile bireyleri aç kalacaktır (private yaparsak). Tek çare özel bir erişim belirleyicisi kullanmaktır yani sadece aileden olanlara (aynı paketin içerisindeki sınıflara) bu buzdolabının erişmesine izin veren bir erişim belirleyicisi yani protected erişim belirleyicisi.

#### **Örnek-4.20:** *Hayvan.java*

```
package tr.edu.kou.util;

public class Hayvan {
    protected String a = "Hayvan.a";
    String b = "Hayvan.b"; //friendly
    private String c = "Hayvan.c";
    public String d = "Hayvan.d"; ;
}
```

*Hayvan* sınıfından türetilen *Kedi* sınıfı *tr.edu.kou.gerekli* paketi içerisine yerleştirildi.

#### **Örnek-4.21:** *Kedi.java*

```
Package tr.edu.kou.gerekli;
Import tr.edu.kou.util.*;

Public class Kedi extends Hayvan {
    public Kedi() {
        System.out.println("Kedi olusturuluyor");
        System.out.println(a);
        // System.out.println(b); // ! Hata ! erisemez
        // System.out.println(c); // ! Hata ! erisemez
        System.out.println(d);
    }

    public static void main(String args[]) {
        Kedi k = new Kedi();
    }
}
```

Kedi.java dosyasını önce derleyip (*compile*)

#### **Gösterim-4.13:**

```
javac Kedi.java
```

Sonrada çalıştırılır.

#### **Gösterim-4.14:**

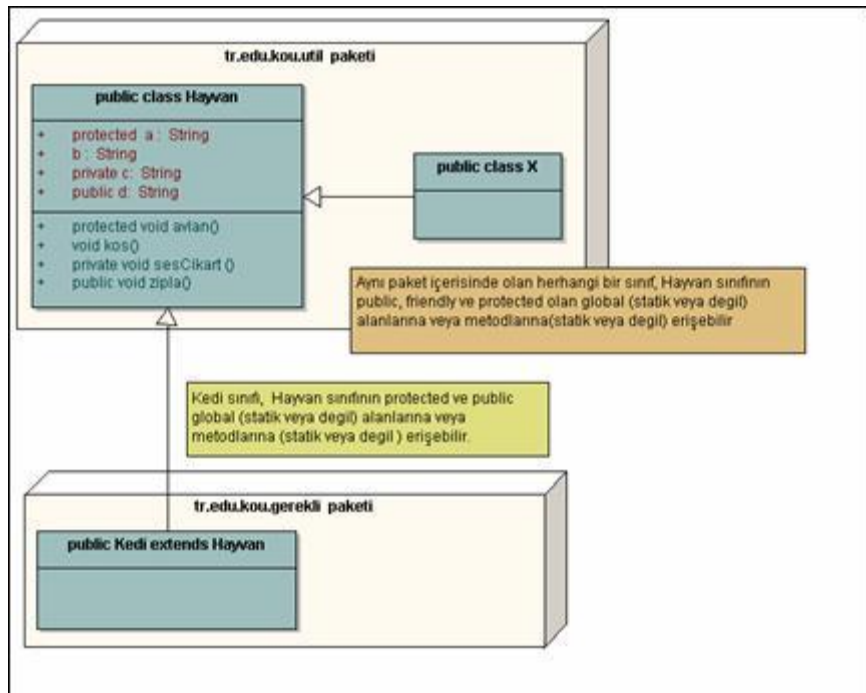
```
java tr.edu.kou.gerekli.Kedi
```

Uygulamanın sonucu aşağıdaki gibi olacaktır:

```
Kedi oluşturuluyor  
Hayvan.a  
Hayvan.d
```

Anlatılanları şekil üzerinde incelenirse,

Şekil-4.9'dan görülebileceği gibi, *tr.edu.kou.gerekli.Kedi* sınıfı, *tr.edu.kou.util. Hayvan* sınıfının *public* ve *protected* erişim belirleyicilerine sahip olan global alanlarına ve yordamlarına erişebilme fırsatı bulmuştur.



Şekil-4.9. protected erişim belirleyicisi

#### 4.9. Kapsüllenme (Encapsulation)

Nesneye yönelik programlama özelliklerinden birisi kapsüllenme; bu, dışarıdaki başka bir uygulamanın bizim nesnemiz ile sadece arabirimler (*public*) sayesinde iletişim kurması gerektiğini, ancak, arka planda işi yapan esas kısmın gizlenmesi gerektiğini söyler. Olaylara bu açıdan bakılırsa, nesneleri 2 kısma bölmeliyiz; arabirimler -ki nesnenin dünya ile iletişim kurabilmesi için gerekli kısımlar ve gemiyi yürüten kısım...

#### Örnek-4.22: *Makine2.java*

```
package tr.edu.kou.util;  
public class Makine2 {  
    private int alinan = 0;  
    private int geridondurulen = 0 ;  
    public int get() {
```

```

        return geridondurulen; }
    public void set(int i ) {
        alinan = i; calis();
    }
    private void calis() {
        for (int j = 0 ; j < alinan ; j++ ) {
            System.out.println("Sonuc = "+j);
        }
        geridondurulen = alinan * 2 ;
    }
}

```

Bir önce verilen örnekte bu *Makine2* türündeki nesneye yalnızca `get()` ve `set()` yordamlarıyla ulaşılabiliriz; geriye kalan global nesne alanlarına veya `calis()` yordamına ulaşım söz konusu değildir. Kapsüllenme kavramının dediği gibi nesneyi 2 kısımdan oluşturduk: ara birimler (- `get()`, `set()` -) ve gemiyi yürüten kısım (- `calis()` -).

Başka paket içerisinde olan başka bir uygulama, *tr.edu.kou.util.Makine2* sınıfının sadece iki yordamına erişebilir, `get()` ve `set()`.

#### **Örnek-4.23:** *GetSet.java*

```

package tr.edu.kou.gerekli;
import tr.edu.kou.util.*;
public class GetSet {
    public static void main(String args[]) {
        Makine2 m2 = new Makine2() ;
        m2.set(5);
        int deger = m2.get();
        // m2.calis() ;      // Hata !
        // m2.alinan ;      // Hata !
        // m2.geridondurulen; // Hata !
        System.out.println("Deger =" + deger);
    }
}

```

#### 4.10. Genel bir bakış

Sınıflar için erişim tablosu aşağıdaki gibidir:

|           | <b>Aynı Paket</b> | <b>Ayrı Paket</b> | <b>Ayrı paket-türetilmiş</b> |
|-----------|-------------------|-------------------|------------------------------|
| public    | erişebilir        | erişebilir        | erişebilir                   |
| protected | -                 | -                 | -                            |
| friendly  | erişebilir        | erişemez          | erişemez                     |
| private   | -                 | -                 | -                            |

Sınıflar `protected` veya `private` olamazlar, bu bağlamda bu tablomuzu şöyle okuyabiliriz; örneğin elimizde *A* sınıfı bulunsun

- `public A` sınıfına aynı paketin içerisindeki başka bir sınıf tarafından erişebilir.

- `public A sınıfına` ayrı paketin içerisindeki başka bir sınıf tarafından erişilebilir.
- `public A sınıfına` ayrı paketten erişebildiğinden buradan yeni sınıflar türetilir.
- `friendly A sınıfına` aynı paketin içerisindeki başka bir sınıf tarafından erişilebilir.
- `friendly A sınıfına` ayrı paketin içerisindeki başka bir sınıf tarafından erişilemez.
- `friendly A'ya` ayrı paketten erişilemediğinden, buradan yeni sınıflar türetilmez.

Statik veya statik olmayan yordamlar için, erişim tablosu aşağıdaki gibidir.

|           | Aynı Paket | Ayrı Paket | Ayrı paket-türetilmiş |
|-----------|------------|------------|-----------------------|
| public    | erişebilir | erişebilir | public                |
| protected | erişebilir | erişemez   | erişebilir            |
| friendly  | erişebilir | erişemez   | erişemez              |
| private   | erişemez   | erişemez   | erişemez              |

Yordamlar `public`, `protected`, `friendly` ve `private` olabilirler. Örneğin, `public X` sınıfının içerisinde `f()` yordamı olsun:

- `public f()` yordamı, aynı paket içerisinde erişilebilir.
- `protected f()` yordamı, hem aynı paket içerisinde, hem de `X` sınıfından türetilmiş ayrı paketteki bir sınıf tarafından erişilebilir.
- `friendly f()` yordamı, yalnızca aynı paket içerisinde erişilebilir.
- `private f()` yordamına, yalnızca kendi sınıfı içerisinde erişilebilir. Başka bir sınıfın bu yordama erişmesi mümkün değildir.

Statik veya statik olmayan global alanlar için erişim tablosu aşağıdaki gibidir:

|           | Aynı Paket | Ayrı Paket | Ayrı paket-türetilmiş |
|-----------|------------|------------|-----------------------|
| public    | erişebilir | erişebilir | erişebilir            |
| protected | erişebilir | erişemez   | erişebilir            |
| friendly  | erişebilir | erişemez   | erişemez              |
| private   | erişemez   | erişemez   | erişemez              |

Global alanlar `public`, `protected`, `friendly`, `private` olabilirler. Örneğin `public X` sınıfının içerisindeki `String` sınıfı tipindeki `uzunluk` adında bir alanımız olsun:

- `public uzunluk` alanı, aynı paket içerisinde erişilebilir.
- `protected uzunluk` alanı, hem aynı paket içerisinde, hem de `X` sınıfından türetilmiş ayrı paketteki bir sınıf tarafından erişilebilir.
- `friendly uzunluk` alanı, yalnızca aynı paket içerisinde erişilebilir.
- `private uzunluk` alanı, yalnızca kendi sınıfı içerisinde erişilebilir. Başka bir sınıfın bu alana erişmesi mümkün değildir.

## **BÖLÜM 5**

### ***SINIFLARIN TEKRAR KULLANILMASI***

Belli bir amaç için yazılmış ve doğruluğu kanıtlanmış olan sınıfları, yeni uygulamaların içerisinde kullanmak hem iş süresini kısaltacaktır hem de yeni yazılan uygulamalarda hata çıkma riskini en aza indirecektir. Uygulamalarımızda daha evvelden yazılmış ve doğruluğu kanıtlanmış olan sınıfları tekrardan kullanmanın iki yöntemi bulunur.

Birinci yöntem kompozisyon'dur. Bu yöntem sayesinde daha önceden yazılmış ve doğruluğu kanıtlanmış olan sınıf/sınıfları, yeni yazılan sınıfın içerisinde doğrudan kullanabilme şansına sahip oluruz. Daha önceki bölümlerde kompozisyon yöntemini çokça kullandık.

İkinci yöntem ise kalıttır (*inheritance*). Bu yöntemde yeni oluşturacağımız sınıfı, daha evvelden yazılmış ve doğruluğu kanıtlanmış olan sınıftan türetilir; böylece yeni oluşan sınıf, türetildiği sınıfın özelliklerine sahip olur; Ayrıca oluşan bu yeni sınıfın kendisine ait yeni özellikleri de olabilir.

#### **5.1. Kompozisyon**

Kompozisyon yönetimini daha önceki örneklerde kullanıldı. Şimdi bu yöntemin detaylarını hep beraber inceleyelim.

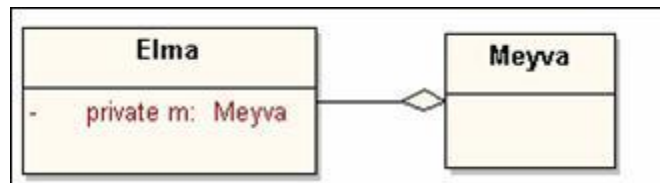
##### **Gösterim-5.1:**

```
class Meyva {  
    //...  
}
```

##### **Gösterim-5.2:**

```
class Elma {  
    private Meyva m = new Meyva();  
    //...  
}
```

*Elma* sınıfı, *Meyva* sınıfını doğrudan kendi içerisinde tanımlayarak, *Meyva* sınıfının içerisindeki erişilebilir olan özellikleri kullanabilir. Buradaki yapılan iş *Elma* sınıfını *Meyva* sınıfına bağlamaktır. Sınıfların arasındaki ilişki UML diyagramında gösterilirse;



**Şekil-5.1. Kompozisyon-I**

Başka bir örnek verilirse,

### Örnek-5.1: *Motor.java*

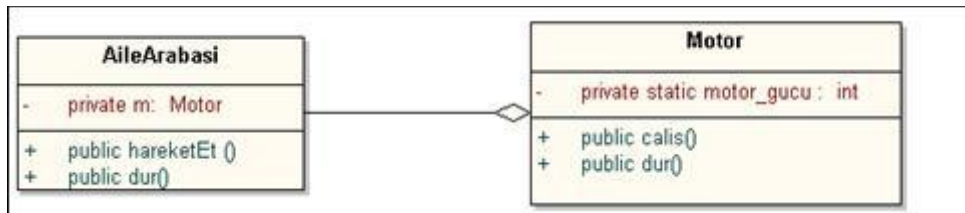
```
public class Motor {  
    private static int motor_gucu = 3600;  
  
    public void calis() {  
        System.out.println("Motor Calisiyor") ;  
    }  
  
    public void dur() {  
        System.out.println("Motor Durdu") ;  
    }  
}
```

Şimdi bu *Motor* sınıfını, arabamızın içerisine yerleştirelim;

### Örnek-5.2: *AileArabasi.java*

```
public class AileArabasi {  
    private Motor m = new Motor();  
    public void hareketEt() {  
        m.calis();  
        System.out.println("Aile Arabasi Calisti");  
    }  
    public void dur() {  
        m.dur();  
        System.out.println("Aile Arabasi Durdu");  
    }  
    public static void main(String args[]) {  
        AileArabasi aa = new AileArabasi() ;  
        Aa.hareketEt();  
        Aa.dur();  
    }  
}
```

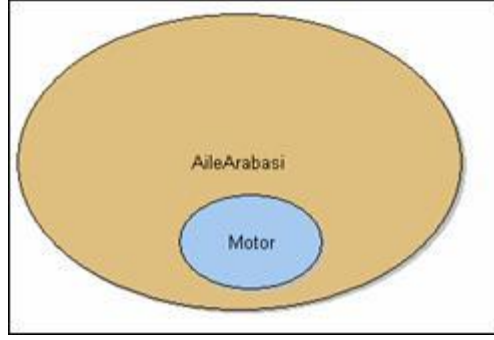
*AileArabasi* sınıfının içerisine, *Motor* tipinde global bir alan yerleştirilerek, bu iki sınıf birbirine bağlanmış oldu. *AileArabasi* sınıfının *hareketEt()* ve *dur()* metodlarında, önce *Motor* sınıfına ait yordamlar (methods) direk olarak çağrıldı. Bu ilişki UML diyagramında incelenirse:



Şekil-5.2. Kompozisyon-II

*Motor* sınıfının private erişim belirleyicisine sahip olan *motor\_gucu* alanına, *AileArabasi* sınıfının içerisinde ulaşamayız. Bunun nedenlerini bir önceki bölümlerde incelemiştik. *AileArabasi* sınıfı *Motor* sınıfının sadece iki adet public yordamına (method) erişebilir: *calis()* ve *dur()*. Olaylara kuş bakışı bakarsak, karşımızdaki manzara aşağıdaki gibidir.





**Şekil-5.3. Kuş Bakışı Görünüş**

*AileArabasi* sınıfı çalıştırılırsa, ekrana görülen sonucun aşağıdaki gibi olması gerekir:

```
Motor Calisiyor  
Aile Arabasi Calisti  
Motor Durdu  
Aile Arabasi Durdu
```

Kompozisyon yöntemine en iyi örnek bir zamanların ünlü çizgi filmi *Voltran*'dır. Bu çizgi filmi hatırlayanlar bileceklerdir ki, büyük ve yenilmez olan robotu (*Voltran*) oluşturmak için değişik ufak robotlar bir araya gelmekteydi. Kollar, bacaklar, gövde ve kafa bölümü... Bizde kendi *Voltran* robotumuzu oluşturmak istersek,

**Örnek-5.3:** *Voltran.java*

```
class Govde {  
    void benzinTankKontrolEt() {}  
}  
  
class SolBacak {  
    void maviLazerSilahiAtesle() {}  
}  
  
class SagBacak {  
    void kirmiziLazerSilahiAtesle() {}  
}  
  
class SagKol {  
    void hedeHodoKalkaniCalistir() {}  
}  
  
class SolKol {  
    void gucKaynagiKontrolEt() {}  
}  
  
class Kafa {  
    void tumBirimlereUyariGonder() {}  
    void dusmanTanimlamaSistemiDevreyeSok() {}  
}  
  
public class Voltran {  
    Govde gv = new Govde();  
    SolBacak slb = new SolBacak();  
    SagBacak sgb = new SagBacak();
```

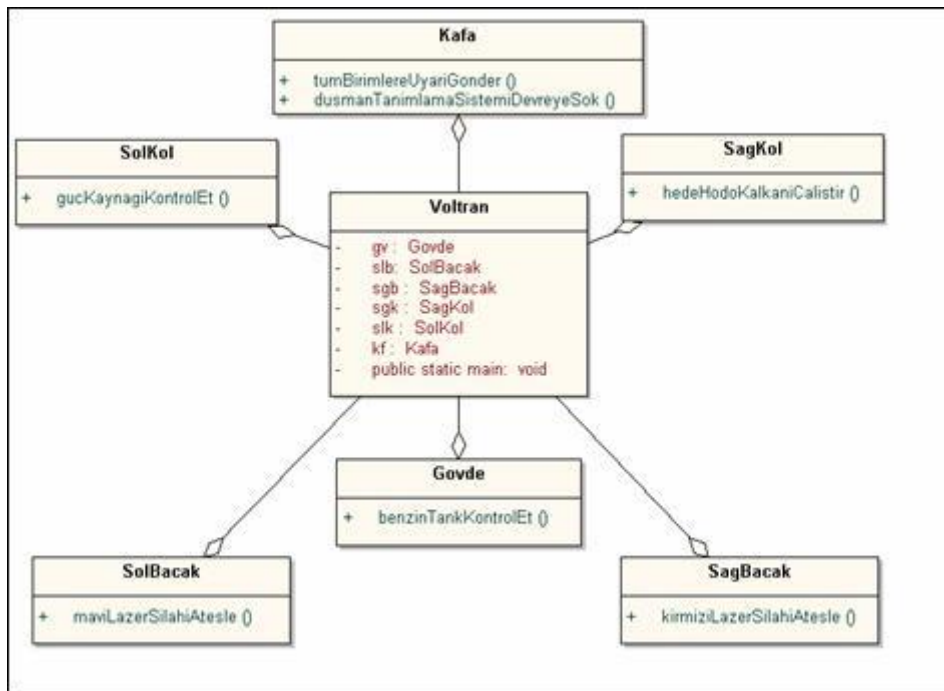
```

SagKol sgk = new SagKol() ;
SolKol slk = new SolKol() ;
Kafa kf = new Kafa() ;

public static void main(String args[]) {
    Voltran vr = new Voltran() ;
    vr.kf.dusmanTanimlamaSistemiDevreyeSok() ;
    vr.kf.tumBirimlereUyariGonder() ;
    vr.sgb.kirmiziLazerSilahiAtesle() ;
}
}

```

*Voltran* sınıfı 6 değişik sınıf tarafından oluşturulmaktadır; bu sınıflara ait özellikler daha sonradan *Voltran* sınıfının içerisinde ihtiyaçlara göre kullanılıyor. Oluşan olaylar UML diyagramında tanımlanırsa:



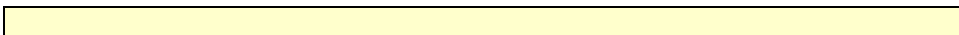
**Şekil-5.4. Kompozisyon - III**

## 5.2. Kalıtım

Kalıtım konusu nesneye yönelik programlamanın (*object oriented programming*) en önemli kavramlarından bir tanesidir. Kalıtım kavramı, kısaca bir sınıftan diğer bir sınıfın türemesidir. Yeni türeyen sınıf, türetilen sınıfın global alanlarına ve yordamlarına (statik veya değil) otomatik olarak sahip olur (*private* olanlar hariç).

Unutulmaması gereken unsur, yeni türeyen sınıf, türetilen sınıfın *private* global alanlarına ve yordamlarına (statik veya değil) otomatik olarak sahip olamaz. Ayrıca yeni türeyen sınıf eğer türetilen sınıf ile ayrı paketlerde ise yeni türeyen sınıf, türetilen sınıfın sadece *public* ve *protected* erişim belirleyicisine sahip olan global alanlarına (statik veya değil) ve yordamlarına (statik veya değil) otomatik olarak sahip olur.

### Gösterim-5.3:



```

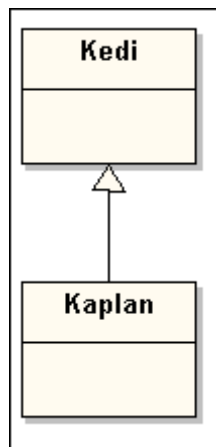
class Kedi {
    //..
}

class Kaplan extends Kedi {
    //..
}

```

*Kedi* sınıfından türeyen *Kaplan* sınıfı... İki sınıf arasındaki ilişkiyi şöyle tarif edebiliriz, her *Kaplan* **bir** *Kedi* dir. Yani her kaplan kedisel özellikler taşıyacaktır ama bu özelliklerin üzerine kendisine bir şeyler eklemiştir.

Yazılış ifadesi olarak, türeyen sınıf isminin yanına **extends** ifadesini koyarak, hemen sonrasında kendisinden türetilme yapılan sınıfın kendisini yerleştiririz (bkz: gösterim-5.3). Yukarıdaki örneğimizi UML diyagramında göstermeye çalışırsak;



**Şekil-5.5. Kalıtım İlişkisi-I**

*Kedi* ve *Kaplan* sınıflarımızı biraz daha geliştirelim,

#### **Örnek-5.4:** *KediKaplan.java*

```

class Kedi {

    protected int ayakSayisi = 4 ;
    public void yakalaAv() {
        System.out.println("Kedi sinifi Av
yakaladi");
    }

    public static void main(String args[]) {
        Kedi kd= new Kedi() ;
        kd.yakalaAv() ;
    }
}

class Kaplan extends Kedi {

    public static void main(String args[] ) {
        Kaplan kp = new Kaplan();
    }
}

```

```

        kp.yakalaAv();
        System.out.println("Ayak Sayisi = " +
        kp.ayakSayisi);
    }
}

```

*Kaplan* sınıfı *Kedi* sınıfından türemiştir. Görüldüğü üzere *Kaplan* sınıfının içerisinde ne *yakalaAv()* yordamı ne de *ayaksayisi* alanı tanımlanmıştır. *Kaplan* sınıfı bu özelliklerini kendisinin ana sınıfı olan *Kedi* sınıfından miras almıştır.

*Kedi* sınıfının içerisinde tanımlanmış *ayaksayisi* alanı, *protected* erişim belirleyicisine sahiptir. Bunun anlamı, bu alana aynı paket içerisinde olan sınıflar ve ayrı paket içerisinde olup bu sınıftan türetilmiş olan sınıfların erişebileceğidir. Böylece *Kaplan* sınıfı ister *Kedi* sınıfı ile aynı pakette olsun veya olmasın, *Kedi* sınıfına ait global *int* ilkel (*primitive*) tipindeki alanına (*ayaksayisi*) erişebilir.

Her sınıfın içerisine *main* yordamı yazarak onları tek başlarına çalışabilir bir hale sokabiliriz (*standalone application*); bu yöntem sınıfları test etmek açısından iyidir. Örneğin *Kedi* sınıfını çalıştırmak için komut satırından *java Kedi* veya *Kaplan* sınıfını çalıştırmak için *java Kaplan* yazılması yeterli olacaktır.

#### 5.2.1. Gizli Kalıtım

Oluşturduğumuz her yeni sınıf otomatik ve gizli olarak *Object* sınıfından türer. *Object* sınıfı Java programlama dili içerisinde kullanılan tüm sınıfların tepesinde bulunur.

#### **Örnek-5.5:** *YeniBirSinif.java*

```

public class YeniBirSinif {
    public static void main(String[] args) {
        YeniBirSinif ybs1 = new YeniBirSinif();
        YeniBirSinif ybs2 = new YeniBirSinif();
        System.out.println("YeniBirSinif.toString()" + ybs1 );
        System.out.println("YeniBirSinif.toString()" + ybs2 );
        System.out.println("ybs1.equals(ybs2)"+ybs1.equals(ybs2)) ;
        // ....
    }
}

```

Uygulamamızın çıktısı aşağıdaki gibi olur:

```

YeniBirSinif.toString() YeniBirSinif@82f0db
YeniBirSinif.toString() YeniBirSinif@92d342
ybs1.equals(ybs2) false

```

*YeniBirSinif* sınıfımızda, *toString()* ve *equals()* yordamları tanımlanmamasına rağmen bu yordamları kullandık, ama nasıl ? Biz yeni bir sınıf tanımladığımızda, Java gizli ve otomatik olarak *extends Object*, ibaresini yerleştirir.

#### **Gösterim-5.4:**

```

public class YeniBirSinif extends Object {

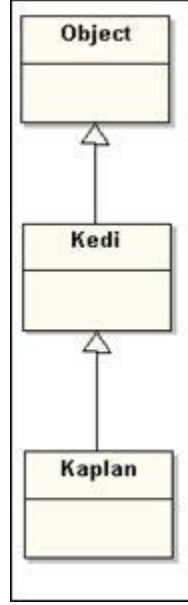
```

Bu sayede *Object* nesnesine ait erişebilir yordamları kullanabiliriz. *Object* nesnesine ait yordamlar aşağıdaki gibidir:

- **clone()** : Bu nesnenin (*this*) aynısını klonlar ve yeni nesneyi döndürür.
- **equals(Object obj)** : *obj* referansına bağlı olan nesnenin, kendisine (*this*) eşit olup olmadığı kontrolü yapan yordam.
- **finalize()** : Çöp toplayıcısı tarafından silinmeden önce çalıştırılan yordam.
- **getClass()** : Bu nesnenin (*this*) çalışma anındaki sınıf bilgilerini *Class* nesnesi şeklinde geri döner.
- **hashCode()** : Bu nesnenin (*this*) hash kodunu geri döner.
- **notify()** : Bu nesnenin (*this*), monitöründe olan tek bir iş parçacığını (*thread*) uyandırır. (ilerleyen bölümlerde inceleyeceğiz)
- **notifyAll()** : Bu nesnenin (*this*), monitöründe olan tüm iş parçacıklarını (*thread*) uyandırır. (ilerleyen bölümlerde incelenecektir)
- **toString()** : Bu nesnenin (*this*), *String* tipindeki ifadesini geri döner.
- **wait()** : O andaki iş parçacığının (*thread*) beklemesini sağlar; Bu bekleme *notify()* veya *notifyAll()* yordamları sayesinde sona erer.
- **wait(long zamanAsimi)** : O andaki iş parçacığının (*thread*), belirtilen süre kadar beklemesini sağlar (*zamanAsimi*); bu bekleme *notify()* veya *notifyAll()* yordamları sayesinde de sona erdirilebilir.
- **wait(long zamanAsimi, int nanos)** : O andaki iş parçacığının (*thread*), belirtilen gerçek süre kadar (*zamanAsimi*+ *nanos*) beklemesini sağlar; bu bekleme *notify()* veya *notifyAll()* yordamları sayesinde de sona erdirilebilir. *nanos* parametresi 0-999999 arasında olmalıdır.

Kısacası, oluşturulan her yeni sınıf, yukarıdaki, yordamlara otomatik olarak sahip olur. Bu yordamları yeni oluşan sınıfların içerisinde tekrardan istediğimiz gibi yazabiliriz (uygun olan yordamları iptal edebiliriz-*override*). Örneğin *finalize()* yordamı kendi sınıfımızın içerisinde farklı sorumluluklar verebiliriz (çizgi çizen bir nesnenin, bellekten silinirken çizdiği çizgileri temizlemesi gibi). Bu olaya, ana sınıfın yordamlarını iptal etmek (*override*) denir. Biraz sonra iptal etmek (*override*) konusunu daha detaylı bir şekilde incelenecektir.

Akıllara şöyle bir soru gelebilir, *Kaplan* sınıfı hem *Kedi* sınıfından hem de *Object* sınıfından mı türemiştir? Cevap hayır. Java programlama dilinde çoklu kalıtım (*multiple inheritance*) yoktur. Aşağıdan yukarıya doğru gidersek, *Kaplan* sınıfı *Kedi* sınıfından türemiştir, *Kedi* sınıfa da *Object* sınıfından (gizli ve otomatik olarak) türemiştir. Sonuçta *Kaplan* sınıfı hem *Kedi* sınıfının hem de *Object* sınıfına ait özellikler taşıyacaktır. Aşağıdaki şeklimizde görüldüğü üzere her sınıf sadece tek bir sınıftan türetilmiştir. *Object* sınıfı, Java programlama dilinde, sınıf hiyerarşinin en tepesinde bulunur.



**Şekil-5.6. Gizli Kalıtım**

Çoklu kalıtım (*multiple inheritance*), bazı konularda faydalı olmasının yanında birçok sorun oluşturmaktadır. Örneğin iki ana sınıf düşünün, bunların aynı isimde değişik işlemler yapan yordamları bulunsun. Bu olay türetilen sınıfın içerisinde birçok probleme yol açacaktır. Bu ve bunun gibi sebeplerden dolayı Java programlama dilinde çoklu kalıtım yoktur. Bu sebeplerin detaylarını ilerleyen bölümlerde inceleyeceğiz.

Java programlama dilinde çoklu kalıtımın faydalarından yararlanmak için Arayüzler (Interface) ve dahili sınıflar (inner class) kullanılır. Bu konular yine ilerleyen bölümlerde inceleyeceğiz.

### 5.2.2. Kalıtım ve İlk Değer Alma Sırası

Tek bir sınıf içerisinde ilk değerlerin nasıl alındığı 3. bölümde incelenmişti. Şimdi içerisinde birde kalıtım kavramı girince olaylar biraz karışabilir. Kalıtım (*inheritance*) kavramı bir sınıftan, başka bir sınıf kopyalamak değildir. Kalıtım kavramı, türeyen bir sınıfın, türetildiği sınıfa ait erişilebilir olan özellikleri alması ve ayrıca kendisine ait özellikleri tanımlayabilmesi anlamına gelir. Bir sınıfa ait nesne oluşurken, ilk önce bu sınıfa ait yapılandırıcının (*constructor*) çağrıldığını önceki bölümlerimizden biliyoruz.

Verilen örnekte, *UcanYarasa* nesnesi oluşmadan evvel, *UcanYarasa* sınıfının ana sınıfı olan *Yarasa* nesnesi oluşturulmaya çalışılacaktır. Fakat *Yarasa* sınıfında *Hayvan* sınıfından türetildiği için daha öncesinde *Hayvan* sınıfına ait olan yapılandırıcı çalıştırılacaktır. Bu zincirleme giden olayın en başında ise *Object* sınıfı vardır.

#### **Örnek-5.6:** *IlkDegerVermeSirasi.java*

```
class Hayvan {
    public Hayvan() {
        System.out.println("Hayvan Yapilandiricisi");
    }
}

class Yarasa extends Hayvan {
    public Yarasa() {
        System.out.println("Yarasa Yapilandiricisi");
    }
}
```

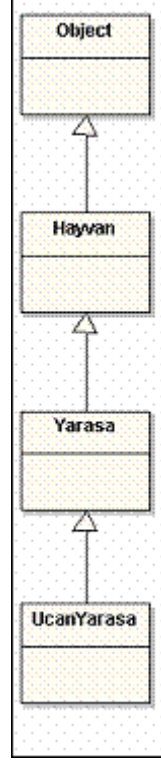
```

}

class UcanYarasa extends Yarasa{
    public UcanYarasa() {
        System.out.println("UcanYarasa Yapilandiricisi");
    }

    public static void main(String args[]) {
        UcanYarasa uy = new UcanYarasa();
    }
}

```



**Şekil-5.7. Kalıtım ve ilk değer alma sırası**

*Object* sınıfını bir kenara koyarsak, ilk olarak *Hayvan* sınıfının yapılandırıcısı çalışacaktır, daha sonra *Yarasa* sınıfının yapılandırıcısı çalışacaktır ve en son olarak *UcanYarasa* sınıfının yapılandırıcısı çalışacaktır. Bu yapılandırıcıların hepsi, fark edildiği üzere varsayılan yapılandırıcıdır (*default constructor*). Uygulamanın çıktısı aşağıdaki gibi olacaktır;

```

Hayvan Yapilandiricisi
Yarasa Yapilandiricisi
UcanYarasa Yapilandiricisi

```

### 5.2.3. Parametre Alan Yapılandırıcılar ve Kalıtım

Ana sınıfa ait yapılandırıcı çağırma işlemi, varsayılan yapılandırıcılar için otomatik olarak yürürken, parametre alan yapılandırıcılar için olaylar biraz daha değişiktir. Kısacası, ana sınıfın parametre alan yapılandırıcısını açık olarak *super* anahtar kelimesi ile çağırarak gereklidir. Şöyle ki;

#### **Örnek-5.7:** *IlkDegerVermeSirasiParametrelili.java*

```

class Insan {

```

```

    public İnsan(int par) {
        System.out.println("İnsan Yapilandiricisi " + par);
    }
}

class Zekiİnsan extends İnsan {
    public Zekiİnsan(int par) {
        super(par+1); //dikkat
        System.out.println("Zekiİnsan Yapilandiricisi " + par);
    }
}

class Hacker extends Zekiİnsan{
    public Hacker(int par) {
        super(par+1); //dikkat
        System.out.println("Hacker Yapilandiricisi " + par);
    }

    public static void main(String args[]) {
        Hacker hck = new Hacker(5);
    }
}

```

Yukarıdaki örneğimizde, her sınıf, yapılandırıcısına gelen değeri bir arttırıp ana sınıfının yapılandırıcısına göndermektedir. Fark edildiği üzere ana sınıfın parametre alan yapılandırıcısını çağırırken **super** anahtar kelimesini kullandık. Uygulamanın çıktısı aşağıdaki gibidir.

```

İnsan Yapilandiricisi 7
Zekiİnsan Yapilandiricisi 6
Hacker Yapilandiricisi 5

```

Dikkat edilmesi gereken bir başka husus, aynı `this` anahtar kelimesinin kullanılışı gibi `super` anahtar kelimesi de içinde bulunduğu yapılandırıcının ilk satırında yer almalıdır.

#### **Örnek-5.8:** *IlkDegerVermeSirasiParametreliliAmaHatali.java*

```

class İnsan2 {
    public İnsan2(int par) {
        System.out.println("İnsan2 Yapilandiricisi " + par);
    }
}

class Zekiİnsan2 extends İnsan2 {
    public Zekiİnsan2(int par) {
        System.out.println("Zekiİnsan2 Yapilandiricisi " + par);
        super(par+1);          // 2. satıra yaziliyor ! hata !
    }
}

class Hacker2 extends Zekiİnsan2 {
    public Hacker2(int par) {
        System.out.println("Hacker2 Yapilandiricisi " + par);
        System.out.println(".....selam.....");
        super(par+1);          // 3. satıra yaziliyor ! hata !
    }
}

```



```

    public static void main(String args[]) {
        Hacker2 hck2 = new Hacker2(5);
    }
}

```

*IlkDegerVermeSirasiParametrelili.java* örneğini derlenirse:

#### **Gösterim-5.5:**

```
javac IlkDegerVermeSirasiParametreliliAmaHatali.java
```

Aşağıdaki derleme-anı (*compile-time*) hatası ile karşılaşılır:

```

IlkDegerVermeSirasiParametreliliAmaHatali.java:11: cannot
resolve symbol
symbol : constructor İnsan2 ()
location: class İnsan2
    public Zekiİnsan2(int par) {
        ^
IlkDegerVermeSirasiParametreliliAmaHatali.java:14: call
to super must be first statement in constructor
        Super(par+1);           // 2. satıra yazılıyor ! hata !
        ^
IlkDegerVermeSirasiParametreliliAmaHatali.java:21: cannot
resolve symbol
symbol : constructor Zekiİnsan2 ()
location: class Zekiİnsan2
    public Hacker2(int par) {
        ^
IlkDegerVermeSirasiParametreliliAmaHatali.java:25: call
to super must be first statement in constructor
        super(par+1);           // 3. satıra yazılıyor ! hata !
        ^
4 errors

```

### **5.3. Kompozisyon mu? Kalıtım mı?**

Yeni oluşturduğunuz sınıfın içerisinde, daha evvelden yazılmış sınıfların özelliklerinden faydalanmak istiyorsanız bunun iki yolu olduğunu belirtmiştik; Kompozisyon ve kalıtım. Peki hangi yöntemi ne zaman tercih etmeliyiz? Kompozisyon, daha evvelden yazılmış sınıfların özelliklerini kullanmak için temiz bir yöntemdir.

#### **Örnek-5.9:** *Araba.java*

```

class ArabaMotoru {
    public void calis() { }
    public void dur() { }
}

class Pencere {

```

```

    public void asagiyaCek() { }
    public void yukariyaCek() { }
}

class Kapi {
    Pencere pencere = new Pencere();
    public void ac() { }
    public void kapa() { }
}

class Tekerlek {
    public void havaPompala(int olcek) { }
}

public class Araba {

    ArabaMotoru arbm = new ArabaMotoru();
    // 2 kapili spor bir araba olsun
    Kapi sag_kapi = new Kapi();
    Kapi sol_kapi = new Kapi();
    Tekerlek[] tekerlekler = new Tekerlek[4] ;
    public Araba() {
        for (int i = 0 ; i < 4 ; i++ ) {
            tekerlekler[i] = new Tekerlek();
        }
    }
    public static void main ( String args[] ) {
        Araba araba = new Araba();
        araba.sag_kapi.pencere.yukariyaCek();
        araba.tekerlekler[2].havaPompala(70);
    }
}

```

Peki, kalıtım kavramı ne zaman kullanılır? Daha evvelden yazılmış bir sınıfın, belli bir problem için yeni versiyonunu yazma işleminde, kalıtım kavramı kullanılabilir. Fakat kalıtım konusunda türetilen sınıf ile türeyen sınıf arasında bir ilişki olmalıdır. Bu ilişki "bir" ilişkisidir. Örneğin *Kedi* ve *Kaplan* sınıflarını göz önüne alırsak, şöyle bir söz yanlış olmaz sanırım, *Kaplan **bir** Kedidir*. Bu iki sınıf arasında "bir" (is -a) ilişkisi olduğundan, kalıtım kavramını bu sınıflar üzerinde rahatça kullanabiliriz.

Örnekleri çoğaltmak mümkündür; *UçanYarasa*, *Yarasa* ve *Hayvan* arasındaki ilişki açıklanışa,

- *UçanYarasa* **bir** Yarasadır;
- Yarasa **bir** Hayvandır;
- O zaman *UçanYarasa*'da **bir** Hayvandır.
- Hayvan'da bir Nesnedir. (bkz. Şekil-5.7.)

#### 5.4. İptal Etmek (*Overriding*)

Ana sınıf içerisinde tanımlanmış bir yordam, ana sınıftan türeyen bir alt sınıfın içerisinde iptal edilebilir.

**Örnek-5.10:** *KitapEvi.java*

```

class Kitap {
    public int sayfaSayisiOgren() {
        System.out.println("Kitap - sayfaSayisiOgren() ");
        return 440;
    }

    public double fiyatOgren() {
        System.out.println("Kitap - fiyatOgren() ");
        return 2500000 ;
    }

    public String yazarIsmiOgren() {
        System.out.println("Kitap - yazarIsmiOgren() ");
        return "xy";
    }
}

class Roman extends Kitap {

    public static void main( String args[] ) {
        Roman r = new Roman();
        int sayfasayisi = r.sayfaSayisiOgren();
        double fiyat = r.fiyatOgren();
        String yazar = r.yazarIsmiOgren();
    }

}

```

Uygulamamızı javac KitapEvi.java komutu ile derledikten sonra, java Roman komutunu çalıştırdığımızda, uygulamamızın çıktısı aşağıdaki gibi olur;

```

Kitap - sayfaSayisiOgren()
Kitap - fiyatOgren()
Kitap - yazarIsmiOgren()

```

*Roman* sınıfının içerisinde *sayfaSayisiOgren()*, *fiyatOgren()*, *yazar-IsmiOgren()* yordamları olmamasına rağmen çağırabildik. Bunun sebebinin kalıtım olduğu biliyoruz. Türeyen sınıf, türediği sınıfa ait global alanları (statik veya değil) ve yordamları (statik veya değil) kullanabilir. Tabii geçen bölümden hatırlayacağız üzere, ana sınıfa ait *private* erişim belirleyicisine sahip olan alanlara ve yordamlara, türeyen alt sınıf tarafından kesinlikle erişilemez. Aynı şekilde türeyen alt sınıf, türetildiği ana sınıf ile aynı paket içerisinde değilse, ana sınıfa ait *friendly* erişim belirleyicisine sahip olan alanlara ve yordamlara erişemez, sadece *protected* erişim belirleyicisine sahip olan alanlara ve yordamlara erişebilir.

*KitapEvi.java* örneğimizde *Roman* sınıfı da her özelliğini, kendisinin ana sınıfı olan *Kitap* sınıfından kalıtım yoluyla almıştır. Peki şimdi *Roman* sınıfının içerisinde *sayfaSayisiOgren()* ve *fiyatOgren()* adında iki yordam oluşturulabilir mi? Eğer oluşturulursa nasıl etkiler meydana gelir? Aynı örneğin ikinci bir versiyonunu yazılırsa,

#### **Örnek-5.11:** *KitapEvi2.java*

```

class Kitap2 {
    public int sayfaSayisiOgren() {

```

```

        System.out.println("Kitap2 - sayfaSayisiOgren() ");
        return 440;
    }

    public double fiyatOgren() {
        System.out.println("Kitap2 - fiyatOgren() ");
        return 2500000 ;
    }

    public String yazarIsmiOgren() {
        System.out.println("Kitap2 - yazarIsmiOgren() ");
        return "xy";
    }
}

class Roman2 extends Kitap2 {

    public int sayfaSayisiOgren() {
        System.out.println("Roman2 - sayfaSayisiOgren() ");
        return 569;
    }

    public double fiyatOgren() {
        System.out.println("Roman2 - fiyatOgren() ");
        return 8500000 ;
    }

    public static void main( String args[] ) {
        Roman2 r2 = new Roman2();
        int sayfasayisi = r2.sayfaSayisiOgren();
        double fiyat = r2.fiyatOgren();
        String yazar = r2.yazarIsmiOgren();
    }
}

```

sayfaSayisiOgren() ve fiyatOgren() yordamlarını hem ana sınıfın içerisine (Kitap2) hemde ana sınıftan türeyen yeni sınıfın içerisine (Roman2) yazmış olduk. Peki bu durumda uygulamanın ekrana basacağı sonuç nasıl olur? Uygulamayı derleyip, çalıştırınca, ekrana basılan sonuç aşağıdaki gibidir;

```

Roman2 - sayfaSayisiOgren()
Roman2 - fiyatOgren()
Kitap2 - yazarIsmiOgren()

```

*Roman2* sınıfının içerisinde, ana sınıfa ait yordamların aynılarını tanımladıktan sonra, *Roman2* sınıfının sayfaSayisiOgren() ve fiyatOgren() yordamlarını çağırınca, artık otomatik olarak ana sınıfın yordamları devreye girmedi. Bunun yerine *Roman2* sınıfının sayfaSayisiOgren() ve fiyatOgren() yordamları devreye girdi. Yani *Roman2* sınıfı, türetildiği sınıfın (Kitap2) sayfaSayisiOgren() ve fiyatOgren() yordamlarını iptal etmiş (override) oldu.

Ana sınıfa ait yordamları iptal ederken dikkat edilmesi gereken önemli hususlardan biri erişim belirleyicilerini iyi ayarlamaktır. Konuyu hatalı bir örnek üzerinde gösterirsek;

#### **Örnek-5.12:** Telefonlar.java

```


```

```

class Telefon {
    protected void aramaYap() {
        System.out.println("Telefon.aramaYap()");
    }
}
class CepTelefonu extends Telefon {
    private void aramaYap() { !! hatali !
        System.out.println("CepTelefon.aramaYap()");
    }
}

```

Bu örnek derlenmeye çalışılırsa, aşağıdaki hata mesajı ile karşılaşır

```

Telefonlar.java:10: aramaYap() in CepTelefonu cannot
override aramaYap() in Tele
fon; attempting to assign weaker access privileges; was
protected
private void aramaYap() {
^
1 error

```

Bu hatanın Türkçe açıklaması, iptal eden yordamın `CepTelefonu.aramaYap()`, iptal edilen yordamın `Telefon.aramaYap()` erişim belirleyicisi ile aynı veya daha erişilebilir bir erişim belirleyicisine sahip olması gerektiğini belirtir.

En erişilebilir erişim belirleyicisinden, en erişilemez erişim belirleyicisine doğru sıralarsak;

- **public**: Her yerden erişilmeyi sağlayan erişim belirleyicisi.
- **protected**: Aynı paket içerisinden ve bu sınıftan türemiş alt sınıflar tarafından erişilmeyi sağlayan erişim belirleyicisi.
- **friendly**: Yalnızca aynı paket içerisinden erişilmeyi sağlayan erişim belirleyicisi.
- **private**: Yalnızca kendi sınıfı içerisinden erişilmeyi sağlayan, başka her yerden erişimi kesen erişim belirleyicisi.

Olaylara bu açıdan bakarsak, ana sınıfa ait `a()` isimli `public` erişim belirleyicisine sahip bir yordam var ise, bu sınıftan türeyen bir alt sınıfın, ana sınıfa ait `a()` yordamını iptal etmek için, erişim belirleyicisi kesin kes `public` olmalıdır. Eğer aynı `a()` yordamı `protected` erişim belirleyicisine sahip olsaydı, o zaman türeyen alt sınıfın bu yordamı iptal edebilmesi için erişim belirleyicisini `public` veya `protected` yapması gerekecekti.

#### **Örnek-5.13:** *Hesap.java*

```

class HesapMakinesi {
    void hesapla(double a , double b) {
        System.out.println("HesapMakinesi.hesapla()");
    }
}

class Bilgisayar extends HesapMakinesi {
    protected void hesapla(double a , double b) {
        System.out.println("HesapMakinesi.hesapla()");
    }
}

```

```
}  
}
```

Yukarıdaki örnekte, *HesapMakinesi* sınıfı içerisinde tanımlanan ve *friendly* erişim belirleyicisi olan *hesapla()* yordamı, türeyen alt sınıf içerisinde iptal edilmiştir (*override*). Bu doğrudur; çünkü, *protected* erişim belirleyicisi, *friendly* erişim belirleyicisine göre daha erişilebilirdir. Fakat, bu 2 sınıf farklı paketlerde olsalardı -ki şu an varsayılan paketin içerisinde- *Bilgisayar* sınıfı, *HesapMakinesi* sınıfına erişemeyeceğinden dolayı (çünkü *HesapMakinesi* sınıfı *friendly* erişim belirleyicisine sahip) kalıtım kavramı söz konusu bile olmazdı.

#### 5.4.1. Sanki İptal Ettim Ama...

Şimdi farklı paketler içerisindeki sınıflar için iptal etmek kavramını nasıl yanlış kullanılabileceği konusunu inceleyelim. Öncelikle *HesapMakinesi* ve *Bilgisayar* sınıflarını *public* sınıf yapıp ayrı ayrı dosyalara kayıt edelim ve bunları farklı paketlerin altına kopyalayalım.

İki ayrı sınıfı farklı paketlere kopyaladık, özellikle *HesapMakinesi* sınıfını *public* sınıf yapmalıyız, yoksa değişik paketlerdeki sınıflar tarafından erişilemez, dolayısıyla kendisinden türetilme yapılamaz.

*HesapMakinesi* sınıfını *tr.edu.kou.math*, *Bilgisayar* sınıfını ise *tr.edu.kou.util* paketinin içerisine yerleştirelim;

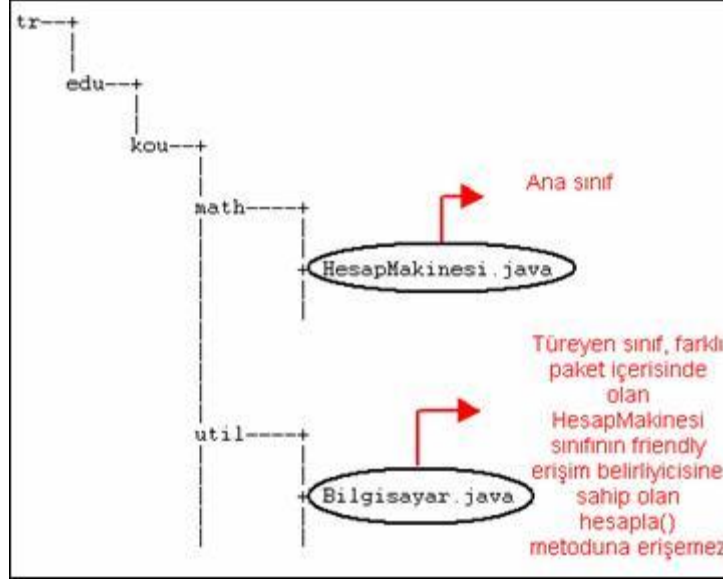
#### **Örnek-5.14:** *HesapMakinesi.java*

```
package tr.edu.kou.math;  
public class HesapMakinesi {  
    void hesapla(double a , double b) {  
        System.out.println("HesapMakinesi.hesapla()");  
    }  
}
```

#### **Örnek-5.15:** *Bilgisayar.java*

```
package tr.edu.kou.util;  
import tr.edu.kou.math.* ;  
  
public class Bilgisayar extends HesapMakinesi {  
    protected void hesapla(double a , double b) {  
        // dikkat  
        System.out.println("HesapMakinesi.hesapla()");  
    }  
  
    public static void main(String args[]) {  
        Bilgisayar b = new Bilgisayar();  
        b.hesapla(3.15, 5.6);  
        HesapMakinesi hm = new HesapMakinesi();  
        // hm.hesapla(3.15, 5.6); !Hata! başka paket içerisinden  
                                   // erişilemez  
    }  
}
```

Şu ana kadar yapılanların kuş bakışı görüntüsü aşağıdaki gibi olur:



**Şekil-5.8. Erişim Kavramının Önemi**

Yukarıdaki örnek derlenip *Bilgisayar* sınıfı çalıştırılırsa herhangi bir hata ile karşılaşılmaz.

#### **Gösterim-5.6:**

```
> java tr.edu.kou.util.Bilgisayar
```

*tr.edu.kou.util* paketinin içerisindeki türeyen *Bilgisayar* sınıfının *protected* erişim belirleyicisine sahip olan *hesapla()* yordamı, *tr.edu.kou.math* paketinin içerisindeki *HesapMakinesi* sınıfının *friendly* erişim belirleyicisine sahip olan *hesapla()* yordamını **iptal edemez**; çünkü türeyen sınıf (*Bilgisayar*) bu yordamın varlığından bile haberdar değildir. *Bilgisayar* sınıfının içerisindeki *hesapla()* yordamı, tamamen *Bilgisayar* sınıfına ait ayrı bir yordamdır. İşte bu yüzden *tr.edu.kou.util* paketinin içerisindeki türeyen *Bilgisayar* sınıfının içerisindeki *hesapla()* yordamı, kendisinin ana sınıfı olan *HesapMakinesi* sınıfının *hesapla()* yordamını **iptal etmekten (override)** gayet uzaktır. Ayrıca *tr.edu.kou.math* paketinin içerisindeki türetilen *HesapMakinesi* sınıfının *friendly* erişim belirleyicisine sahip olan *hesapla()* yordamına erişemediğimizi ispatlamak için *Bilgisayar.java* dosyasındaki yorum kısmını kaldırarak derlemeye çalışırsak, aşağıdaki hata mesajı ile karşılaşırız:

```
Bilgisayar.java:13: hesapla(double,double) is not public in
tr.edu.kou.math.HesapMakinesi; cannot be accessed from outside package
hm.hesapla(3.15, 5.6);
^
1 error
```

Bu hata mesajı, şu ana kadar anlatılanların kanıtı sayılabilir.

#### **5.4.2. İptal Etmek (Overriding) ve Adaş Yordamların (Overload) Birbirlerini Karıştırılması**

Ana sınıfa ait bir yordamı iptal etmek isterken yanlışlıkla adaş yordamlar yazılabilir.

#### **Örnek-5.16: *CalisanMudur.java***

```

class Calisan {
    public void isYap(double a) {
        System.out.println("Calisan.isYap()");
    }
}

class Mudur extends Calisan {
    public void isYap(int a) { // adas yordam (overloaded)
        System.out.println("Mudur.isYap()");
    }

    public static void main(String args[]) {
        Mudur m = new Mudur();
        m.isYap(3.3);
    }
}

```

Her *Müdür* bir *Çalışandır* ilkesinden yola çıkılarak yazılmış bu örneğimizdeki büyük hata iki kavramın - (iptal etmek ve adaş yordamlarının)- birbirlerine karıştırılmasıdır. Böyle bir hata çok kolay bir şekilde yapılabilir ve fark edilmesi de bir o kadar güçtür. Buradaki yanlışlık, yordamların parametrelerindeki farklılıktan doğmaktadır. Kodu yazan kişi, ana sınıfa ait olan `isYap()` yordamı iptal ettiğini kolaylıkla zannedebilir ama aslında farkına bile varmadan adaş yordam (*overloaded*) oluşturmuştur. Uygulamanın sonucu aşağıdaki gibi olur:

```
Calisan.isYap()
```

### 5.5. Yukarı Çevirim (*Upcasting*)

Kalıtım (*inheritance*) kavramı sayesinde, türeyen sınıf ile türetilen sınıf arasında bir ilişki kurulmuş olur. Bu ilişkiyi şöyle açıklayabiliriz “türeyen sınıfın tipi, türetilen sınıf tipindedir”. Yukarıdaki örnek tekrarlanırsa, “her kaplan bir kedidir” denilebilir. *Kaplan* ve *Kedi* sınıfları arasındaki ilişki kalıtım kavramı sayesinde sağlanmış olur. Her kaplan bir kedidir veya her müdür bir çalışandır örneklerimiz sadece sözel örnekler değildir, bu ilişki Java tarafından somut olarak desteklenmektedir.

Başka bir kalıtım örneğini şöyle açıklayabiliriz, her futbolcu bir sporcudur. Bu ifade bize, *Sporcu* sınıfının içerisindeki yordamların otomatik olarak *Futbolcu* sınıfının içerisinde olduğunu söyler, yani *Sporcu* sınıfına gönderilen her mesaj rahatlıkla *Futbolcu* sınıfına da gönderilebilir çünkü *Futbolcu* sınıfı *Sporcu* sınıfından türemiştir. Java’nın bu ilişkiye nasıl somut olarak destek verdiğini aşağıdaki örnekte görülebilir:

#### **Örnek-5.17:** *Spor.java*

```

class KontrolMerkezi {
    public static void checkUp(Sporcu s) {
        //..
        s.calis();
    }
}

class Sporcu {
    public void calis() {
        System.out.println("Sporcu.calis()");
    }
}

```



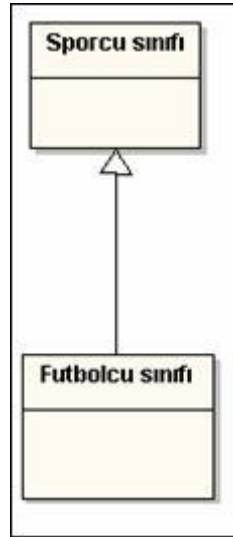
```

}

class Futbolcu extends Sporcu {
    public void calis() {                // iptal etti (Overriding)
        System.out.println("Futbolcu.calis()");
    }
    public static void main(String args[]) {
        Sporcu s = new Sporcu();
        Futbolcu f = new Futbolcu();
        KontrolMerkezi.checkUp(s);
        KontrolMerkezi.checkUp(f);    //dikkat
    }
}

```

*KontrolMerkezi* sınıfının statik bir yordamı olan *checkUp()*, *Sporcu* sınıfı tipinde parametre kabul etmektedir. Buradaki ilginç olan nokta *checkUp()* yordamına, *Futbolcu* sınıfı tipindeki referansı gönderdiğimizde hiç bir hata ile karşılaşmamamızdır. Burada bir hata yoktur çünkü her *Futbolcu* **bir** *Sporcudur*. Türetilmiş sınıfın (*Futbolcu*) içerisinde kendine has bir çok yordam olabilir ama en azından türediği sınıfın (*Sporcu*) içerisindeki yordamlara sahip olacaktır. *Sporcu* sınıfı tipinde parametre kabul eden her yordama *Futbolcu* sınıfı tipinde parametre gönderebiliriz. Bu ilişkiyi UML diyagramında gösterirsek;



**Şekil-5.9. Yukarı Çevirim**

*Sporcu.java* örneğimizde, türeyen sınıf (*Futbolcu*) türetildiği sınıfa (*Sporcu*) doğru çevrilmektedir; yani, yukarı çevrilmektedir. Yukarı çevrim her zaman güvenlidir. Çünkü daha özel bir tipten daha genel bir tipe doğru daralma vardır. Bu nedenle yukarı çevrimlerde özel bir ifade veya belirteç kullanmak zorunda değildir.

Yukarı çevirim (*Upcasting*) olayı "Kompozisyon mu, Kalıtım mı? kullanmalıyım" sorusuna da ışık tutmuştur. Eğer "yukarı doğru çevirime ihtiyacım var mı?" sorunun cevabı "evet" ise, kalıtım (*inheritance*) kullanılması gerekir.

## 5.6. Final Özelliği

Final kelimesinin sözlük anlamı "son" demektir. Java programlama dilindeki *final* özeliği de, sözlük anlamıyla paralel bir anlam taşır. Java programlama dilinde *final* anahtar kelimesi değiştirilemezliği

simgeler. Değiştirilemezliğin seçilmesi iki sebepten dolayı olabilir, birincisi tasarım ikincisi ise verimlilik; Global olan alanlara, yordamlara ve sınıflara final özelliğini uygulayabiliriz.

#### 5.6.1. Global Alanlar ve Final Özelliği

Global alanlar ile final özelliği birleştiği zaman, ortaya diğer programlama dillerindeki sabit değer özelliği ortaya çıkar. Global olan sabit alanlar ister statik olsun veya olmasın final özelliğine sahip olabilir. Java programlama dilinde final olan global alanların değerleri, derleme anında (*compile time*) veya çalışma anında (*run time*) belli olabilir ama dikkat edilmesi gereken husus, final global alanlara sadece bir kere değer atanabiliyor olmasıdır. Sonuç olarak global olan final alanları ikiye ayırabiliriz;

- Derleme anında değerlerini bilebildiğimiz final global alanlar.
- Çalışma anında değerlerini bilebildiğimiz final global alanlar.

#### **Örnek-5.18:** *FinalOrnek.java*

```
class Kutu {
    int i = 0 ;
}

public class FinalOrnek {

    final int X_SABIT_DEGER = 34 ;
    final static int Y_SABIT_DEGER = 35 ;

    final int A_SABIT_DEGER = (int) (Math.random()*50);

    final Kutu k = new Kutu() ;

    public static void main(String args[]) {
        FinalOrnek fo = new FinalOrnek();

        // fo.X_SABIT_DEGER = 15 ! Hata !
        // fo.Y_SABIT_DEGER = 16 ! Hata !
        // fo.A_SABIT_DEGER = 17 ! Hata !

        fo.k.i = 35 ;           // doğru

        // fo.k = new Kutu() ! hata !

        System.out.println("X_SABIT_DEGER = "+fo.X_SABIT_DEGER) ;
        System.out.println("Y_SABIT_DEGER = "+fo.Y_SABIT_DEGER) ;
        System.out.println("A_SABIT_DEGER = "+fo.A_SABIT_DEGER) ;
        System.out.println("Kutu.i = "+fo.k.i) ;
    }
}
```

Verilen örnekte X\_SABIT\_DEGER ve Y\_SABIT\_DEGER alanlarının değerlerini derleme anında bilenebilmesi mümkündür ama A\_SABIT\_DEGER alanının değerini derleme anında bilmek zordur (*Math* sınıfına ait statik bir yordam olan random(), 1 ile 50 arasında rasgele sayılar üretir), bu alanın değeri çalışma anında belli olacaktır. Bir global alana, final ve statik özellikler belirtirseniz, bu global alanımız, bu sınıfa ait olan tüm nesneler için tek olur (bkz: 3. bölüm, statik alanlar) ve değeri sonradan değiştirilemez.

Final özelliğinin etkisi, ilkel tipteki alanlar ve sınıf tipindeki alanlar farklıdır. Yukarıdaki örneğimizi incelerkeniz, X\_SABIT\_DEGER, Y\_SABIT\_DEGER, A\_SABIT\_DEGER alanları hep ilkel tipteydi; yani değerlerini kendi üzerlerinde taşıyorlardı. *Kutu* tipinde k alanımızı final yaptığımızda olaylar biraz değişir, *Kutu* sınıfı tipindeki k alanını final yaparak, bu alanın başka bir *Kutu* nesnesine tekrardan bağlanmasına izin vermeyiz ama *Kutu* sınıfı tipindeki k alanının bağlı olduğu nesnenin içeriği değişebilir. Uygulamanın sonucu aşağıdaki gibi olur:

```
X_SABIT_DEGER = 34
Y_SABIT_DEGER = 35
A_SABIT_DEGER = 39
Kutu.i = 35
```

### 5.6.2. Final Parametreler

Yordamlara gönderilen parametre değerlerinin değişmemesini istiyorsak, bu parametreleri final yapabiliriz.

#### **Örnek-5.19:** *FinalParametre.java*

```
public class FinalParametre {

    public static int topla(final int a , final int b) {
        // a=5 !Hata !
        // b=9 !Hata !
        return a+b;
    }

    public static void main(String args[] ) {
        if ( ( args.length != 2 ) ) {
            System.out.println("Eksik veri Girildi") ;
            System.exit(-1); // Uygulamayi sonlandir
        }

        int a = Integer.parseInt(args[0]);
        int b = Integer.parseInt(args[1]);
        int sonuc = FinalParametre.topla(a,b);
        System.out.println("Sonuc = " + sonuc );
    }
}
```

Bu uygulamamız, dışarıdan iki parametre olarak bunları ilkel olan int tipine çeviriyor. Eğer dışarıdan eksik veya fazla parametre girilmiş ise kullanıcı bu konuda uyarılıyor. Daha sonra elimizdeki değerleri *FinalParametre* sınıfının statik olan topla() yordamına gönderiyoruz. Bu yordama gönderilen parametrelerin değiştirilmesi, final ifadeden dolayı imkansızdır.

### 5.6.3. Boş (Blank) Final

Java, final olan nesneye ait alanlara ilk değeri verme konusunda acele etmez fakat final olan nesne alanları kullanılmadan önce ilk değerlerinin verilmiş olması şarttır.

#### **Örnek-5.20:** *BosFinal.java*

```

class Kalem {
}

public class BosFinal {
    final int a = 0;
    final int b; // Bos final
    final Kalem k;          // Blank final nesne alanı

    // Bos final alanlar ilk değerlerini yapılandırıcılarda içerisinde alırlar
    BosFinal() {
        k = new Kalem();
        b = 1; // bos final alanına ilk değeri ver
    }

    BosFinal(int x) {
        b = x; // bos final alanına ilk değeri ver
        k = new Kalem();
    }

    public static void main(String[] args) {
        BosFinal bf = new BosFinal();
    }
}

```

Boş final alanlara ilk değerleri yapılandırıcıların içerisinde verilemelidir; statik olan global alanlar boş final olma özelliğinden yararlanamazlar.

#### 5.6.4. final Yordamlar

Türetilen alt sınıfların, türetildikleri ana sınıflar içerisindeki erişilebilir olan yordamları iptal edebildiklerini (*override*) biliyoruz. Ana sınıf içerisindeki bir yordamın, alt sınıflar tarafından iptal edilmesi istenmiyorsa, o yordamı *final* yaparak korunabilir. Kısacası *final* yordamlar iptal edilemezler.

#### **Örnek-5.22:** *FinalMetod.java*

```

class A {
    public final void ekranaYaz() {
        System.out.println("A.ekranaYaz()");
    }
}

class B extends A {
    public void ekranaYaz() {
        System.out.println("B.ekranaYaz()");
    }
}

```

*A* sınıfına ait *ekranaYaz()* yordamı, *A* sınıfından türetilmiş *B* sınıfının *ekranaYaz()* yordamı tarafından iptal edilemez (*overriding*). *FinalMetod.java* örneğini derlemeye çalıştığımızda aşağıdaki hata mesajını alırız:

```
FinalMetod.java:9: ekranaYaz() in B cannot override
ekranaYaz() in A; overridden
method is final
    public void ekranaYaz() {
        ^
1 error
```

#### 5.6.5. **private ve final**

`final` ve `private` erişim belirleyicisine sahip olan bir yordam, başka bir yordam tarafından iptal ediliyormuş gibi gözükebilir.

#### **Örnek-5.23:** *SivilPolis.java*

```
class Polis {
    private final void sucluYakala() { // erişilemez gizli yordam
        System.out.println("Polis.sucluYakala()");
    }
}

public class SivilPolis extends Polis {
    public void sucluYakala() { //iptal etme söz konusu değildir
        System.out.println("SivilPolis.sucluYakala()");
    }
}
```

`private` erişim belirleyicisine sahip olan yordam dışarıdan erişilemeyeceğinden dolayı, türetilen sınıflar içerisindeki yordamlar tarafından iptal edilmesi söz konusu değildir. `private` erişim belirleyicisine sahip olan bir yordam, bir sınıfın gizli ve özel tarafıdır, yani o sınıfın dünyaya açılan bir penceresi değildir. Bir sınıfın dünyaya açılan pencereleri, o sınıfa ait `public`, `protected` veya `friendly` erişim belirleyicilerine sahip olan yordamlarıdır.

#### 5.6.6. **Final Sınıflar**

Bir sınıfı `final` yaparak, bu sınıftan türetilme yapılmasını engellemiş oluruz. Bir sınıfın `final` yapılmasının iki sebebi olabilir, birincisi tasarım, ikincisi ise verimlilik. `final` sınıflar kompozisyon yöntemi ile kullanabilirler.

#### **Örnek-5.24:** *Tv.java*

```
final class Televizyon {
    public void kanalBul() {
    }
}

/*
class SuperTelevizyon extends Televizyon{ // Hatalı
}
*/

class Ev {
    int oda sayisi = 5 ;
}
```

```

    Televizyon tv = new Televizyon() ;
    public static void main(String args[]) {
        Ev e = new Ev();
        e.tv.kanalBul();
    }
}

```

### 5.7. Kalıtım (*Inheritance*) ve İlk Değer Alma Sırası

Java programlama dilinde her sınıf kendi fiziksel dosyasında durur. Bir fiziksel *.java* dosyasının içerisinde birden fazla sınıf tanımlanabileceğini de hatırlatmak isterim. Uygulama tarafından kullanılan bu sınıflar, bulundukları fiziksel dosyalarından bir seferde topluca sınıf yükleyicisi tarafından belleğe yüklenmezler. Bunun yerine hangi sınıfa ihtiyaç duyuluyor ise, bu sınıf CLASSPATH değişkenin gösterdiği yerlere bakılarak yüklenilmeye çalışılır. Peki bir sınıf tam olarak ne zaman yüklenir ? Cevap, eğer bir sınıfa ait statik global alan veya statik bir yordam çağrıldığında, bu sınıf, sınıf yükleyicisi (*Class Loader*) tarafından yüklenir veya bir sınıfa ait bir nesne oluşturmak istersek yine sınıf yükleyicisi (*Class Loader*) devreye girerek bu sınıfı yükler.

#### **Örnek-5.25:** *Bocekcik.java*

```

class Bocek {
    int a = 10;
    int b;
    Bocek() {
        ekranaBas("a = " + a + ", b = " + b);
        b = 17;
    }
    static int x1 = ekranaBas("static Bocek.x1 ilk deger
verildi");

    static int ekranaBas(String s) {
        System.out.println(s);
        return 18;
    }
}

public class Bocekcik extends Bocek {
    int k = ekranaBas("Bocekcik.k ilk deger verildi");
    Bocekcik() {
        ekranaBas("k = " + k);
        ekranaBas("b = " + b);
    }
    static int x2= ekranaBas("static Bocekcik.x2 ilk deger
verildi");
    public static void main(String[] args) {
        ekranaBas("Bocekcik - basla..");
        Bocekcik b = new Bocekcik();
    }
}

```

Uygulamanın sonucu aşağıdaki gibi olur:

```

static Bocek.x1 ilk deger verildi

```

```
static Bocekcik.x2 ilk deger verildi
Bocekcik - basla..
a = 10, b = 0
Bocekcik.k ilk deger verildi
k = 18
b = 17
```

Gelişen olaylar adım adım açıklanırsa, öncelikle, *Bocekcik* sınıfına ait statik bir yordam olan `main()` çağrılıyor (`java Bocekcik` komutuyla). Sınıf yükleyici *Bocekcik.class* fiziksel dosyasını, sistemin CLASSPATH değerlerine bakarak bulmaya çalışır. Eğer bulursa bu sınıf yüklenir. *Bocekcik* sınıfının bulunduğunu varsayalım. Bu yükleme esnasında *Bocekcik* sınıfının türetildiği ortaya çıkar (*Bocekcik* extends *Bocek*). Kalıtım kavramından dolayı *Bocek* sınıfı da, sınıf yükleyicisi tarafından yüklenir (eğer *Bocek* sınıfı da türetilmiş olsaydı; türetildiği sınıfta yüklenecekti; böyle sürüp gidebilir...).

Daha sonra statik olan global alanlara ilk değerleri vermeye başlanır. Değer verme işlemi en yukarıdaki sınıftan başlar ve türemiş alt sınıflara doğru devam eder (aşağıya doğru). Burada en yukarıdaki sınıf *Bocek* sınıfıdır - (*Object* sınıfını hesaba katılmazsa). Bu anlatılanlar göz önüne alındığında ekrana çıkan ilk iki satırın aşağıdaki gibi olması bir şaşkınlığa sebebiyet vermez.

```
static Bocek.x1 ilk deger verildi
static Bocekcik.x2 ilk deger verildi
```

Sırada `main()` yordamının çağrılmasına gelmiştir. Ekrana çıkan üçüncü satır aşağıdaki gibidir;

```
Bocekcik - basla..
```

Daha sonra *Bocekcik* nesnesi oluşturulur (`Bocekcik b = new Bocekcik()`). Bu oluşturma sırasında ilk olarak en yukarıdaki sınıfa (*Bocek* sınıfı) ait statik olmayan (*non-static*) alanlara ilk değerleri verilir ve yapılandırıcısı çağrılır. Ekrana çıkan dördüncü satır aşağıdaki gibidir;

```
a = 10, b = 0
```

Son olarak *Bocekcik* sınıfının içerisindeki statik olmayan (*non-static*) alanlara ilk değerleri verilir ve *Bocekcik* sınıfının yapılandırıcısı çağrılır.

```
Bocekcik.k ilk deger verildi
k = 18
b = 17
```

ve mutlu son ...



## ***POLİMORFİZM***

Polimorfizm, nesneye yönelik programlamanın önemli kavramlarından biridir ve sözlük anlamı olarak "bir çok şekil" anlamına gelmektedir. Polimorfizm ile kalıtım konusu iç içedir. Kalıtım konusunu geçen bölüm incelenmişti; kalıtım konusunda iki taraf bulunmaktadır, ana sınıf ve bu sınıftan türeyen alt sınıf/sınıflar.

### **6.1. Detaylar**

Alt sınıf, türetildiği ana sınıfa ait tüm özellikleri alır; yani, ana sınıf ne yapıyorsa türetilen alt sınıfta bu işlemlerin aynısını yapabilir ama türetilen alt sınıfların kendilerine ait bir çok yeni özelliği de olabilir. Ayrıca türetilen alt sınıfa ait nesnenin, ana sınıf tipindeki referansa bağlamanın yukarı doğru (*upcasting*) işlemi olduğu geçen bölüm incelenmişti. Burada anlatılanları bir örnek üzerinde açıklarsak;

**Örnek:** *PolimorfizmOrnekBir.java*

```
class Asker {
    public void selamVer() {
        System.out.println("Asker Selam verdi");
    }
}

class Er extends Asker {
    public void selamVer() {
        System.out.println("Er Selam verdi");
    }
}

class Yuzbasi extends Asker {
    public void selamVer() {
        System.out.println("Yuzbasi Selam verdi");
    }
}

public class PolimorfizmOrnekBir {

    public static void hazirOl(Asker a) {
        a.selamVer(); //! Dikkat !
    }

    public static void main(String args[]) {
        Asker a = new Asker();
        Er e = new Er();
        Yuzbasi y = new Yuzbasi();
        hazirOl(a); // yukarı çevirim ! yok !
        hazirOl(e); // yukarı çevirim (upcasting)
        hazirOl(y); // yukarı çevirim (upcasting)
    }
}
```



Yukarıdaki örnekte üç kavram mevcuttur, bunlardan biri yukarı çevirim (*upcasting*) diğeri polimorfizm ve son olarak da geç bağlama (*late binding*). Şimdi yukarı çevirim ve polimorfizm kavramlarını açıklayalım. Bu örneğimizde ana sınıf *Asker* sınıfıdır; bu sınıftan türeyen sınıflar ise *Er* ve *Yuzbasi* sınıflarıdır. Bu ilişki "bir" ilişkisidir ;

- Er **bir** Askerdir, veya
- Yüzbaşı **bir** Askerdir, diyebiliriz.

Yani *Asker* sınıfının yaptığı her işi *Er* sınıfı veya *Yuzbasi* sınıfı da yapabilir artı türetilen bu iki sınıf kendisine has özellikler taşıyabilir, *Asker* sınıfı ile *Er* ve *Yuzbasi* sınıflarının arasında kalıtsal bir ilişki bulunmasından dolayı, *Asker* tipinde parametre kabul eden `hazirOl()` yordamına *Er* ve *Yuzbasi* tipindeki referansları paslayabildik, bu özelliğinde yukarı çevirim (*upcasting*) olduğunu geçen bölüm incelenmişti.

Polimorfizm ise `hazirOl()` yordamının içerisinde gizlidir. Bu yordamın (*method*) içerisinde *Asker* tipinde olan `a` referansı kendisine gelen 2 değişik nesneye (*Er* ve *Yuzbasi*) bağlanabildi; bunlardan biri *Er* diğeri ise *Yuzbasi*'dir. Peki bu yordamın içerisinde neler olmaktadır? Sırası ile açıklarsak; ilk önce *Asker* nesnesine bağlı *Asker* tipindeki referansı, `hazirOl()` yordamına parametre olarak gönderiyoruz, burada herhangi bir terslik yoktur çünkü `hazirOl()` yordamı zaten *Asker* tipinde parametre kabul etmektedir.

Burada dikkat edilmesi gereken husus, `hazirOl()` yordamının içerisinde *Asker* tipindeki yerel `a` değişkenimizin, kendi tipinden başka nesnelere de (*Er* ve *Yuzbasi*) bağlanabilmesidir; yani, *Asker* tipindeki yerel `a` değişkeni bir çok şekle girmiş bulunmaktadır. Aşağıdaki ifadelerin hepsi doğrudur:

- `Asker a = new Asker();`
- `Asker a = new Er();`
- `Asker a = new Yuzbasi();`

Yukarıdaki ifadelere, *Asker* tipindeki `a` değişkeninin açısından bakarsak, bu değişkenin bir çok nesneye bağlanabildiğini görürüz, bu özellik polimorfizm 'dir -ki bu özelliğin temelinde kalıtım (*inheritance*) yatar. Şimdi sıra geç bağlama (*late binding*) özelliğinin açıklanmasında....

## 6.2. Geç Bağlama (*Late Binding*)

Polimorfizm olmadan, geç bağlamadan bahsedilemez bile, polimorfizm ve geç bağlama (*late binding*) bir elmanın iki yarısı gibidir. Şimdi kaldığımız yerden devam ediyoruz, *Er* nesnesine bağlı *Er* tipindeki referansımızı (`e`) `hazirOl()` yordamına parametre olarak gönderiyoruz.

### Gösterim-6.1:

```
hazirOl(e); // yukari dogru cevrim (upcasting)
```

Bu size ilk başta hata olarak gelebilir, ama arada kalıtım ilişkisinden dolayı (*Er bir* Askerdir) nesneye yönelik programlama çerçevesinde bu olay doğrudur. En önemli kısım geliyor; şimdi, hangi nesnesin `selamVer()` yordamı çağrılacaktır? *Asker* nesnesinin mi? Yoksa *Er* nesnesinin mi ? Cevap: *Er* nesnesinin `selamVer()` yordamı çağrılacaktır. Çünkü *Asker* tipindeki yerel değişken (`a`) *Er* nesnesine bağlanmıştır. Eğer *Er* nesnesinin `selamVer()` yordamı olmasaydı o zaman *Asker* nesnesine ait olan

`selamVer()` yordamı çağrılacaktı fakat *Er* sınıfının içerisinde, ana sınıfa ait olan (*Asker* sınıfı) `selamVer()` yordamı iptal edildiğinden (*override*) dolayı, Java, *Er* nesnesinin `selamVer()` yordamını çağırılacaktır. Peki hangi nesnesinin `selamVer()` yordamının çağrılacağı ne zaman belli olur? Derleme anında mı (*compile-time*)? Yoksa çalışma anında mı (*run-time*)? Cevap; çalışma anında (*run-time*). Bunun sebebi, derleme anında `hazirOl()` yordamına hangi tür nesneye ait referansın gönderileceğinin belli olmamasıdır.

Son olarak, *Yuzbasi* nesnesine bağlı *Yuzbasi* tipindeki referansımızı `hazirOl()` yordamına parametre olarak gönderiyoruz. Artık bu bize şaşırtıcı gelmiyor... devam ediyoruz. Peki şimdi hangi nesneye ait `selamVer()` yordamı çağrılır? *Asker* nesnesinin mi? Yoksa *Yuzbasi* nesnesinin mi? Cevap *Yuzbasi* nesnesine ait olan `selamVer()` yordamının çağrılacağıdır çünkü *Asker* tipindeki yerel değişkenimiz *heap* alanındaki *Yuzbasi* nesnesine bağlıdır ve `selamVer()` yordamı *Yuzbasi* sınıfının içerisinde iptal edilmiştir (*override*). Eğer `selamVer()` yordamı *Yuzbasi* sınıfının içerisinde iptal edilmeseydi o zaman *Asker* sınıfına ait (ana sınıf) `selamVer()` yordamı çağrılacaktı. Aynı şekilde Java hangi nesnenin `selamVer()` yordamının çağrılacağına çalışma-anında (*run-time*) da karar verecektir yani geç bağlama özelliği devreye girmiş olacaktır. Eğer bir yordamın hangi nesneye ait olduğu çalışma anında belli oluyorsa bu olaya geç bağlama (*late-binding*) denir. Bu olayın tam tersi ise erken bağlamadır (*early binding*); yani, hangi nesnenin hangi yordamının çağrılacağı derleme anında bilinmesi. Bu örneğimiz çok fazla basit olduğu için, "Niye ! derleme anında hangi sınıf tipindeki referansın `hazirOl()` yordamına paslandığını bilemeyelim ki, çok kolay, önce *Asker* sınıfına ait bir referans sonra *Er* sınıfına ait bir referans ve en son olarak da *Yuzbasi* sınıfına ait bir referans bu yordama parametre olarak gönderiliyor işte..." diyebilirsiniz ama aşağıdaki örneğimiz için aynı şeyi söylemeniz bu kadar kolay olmayacaktır

### **Örnek:** *PolimorfizmOrnekIki.java*

```
class Hayvan {
    public void avYakala() {
        System.out.println("Hayvan avYakala");
    }
}

class Kartal extends Hayvan {
    public void avYakala() {
        System.out.println("Kartal avYakala");
    }
}

class Timsah extends Hayvan{
    public void avYakala() {
        System.out.println("Timsah avYakala");
    }
}

public class PolimorfizmOrnekIki {

    public static Hayvan rasgeleSec() {
        int sec = ( (int) (Math.random() *3) ) ;
        Hayvan h = null ;
        if (sec == 0) h = new Hayvan();
        if (sec == 1) h = new Kartal();
        if (sec == 2) h = new Timsah();
        return h;
    }

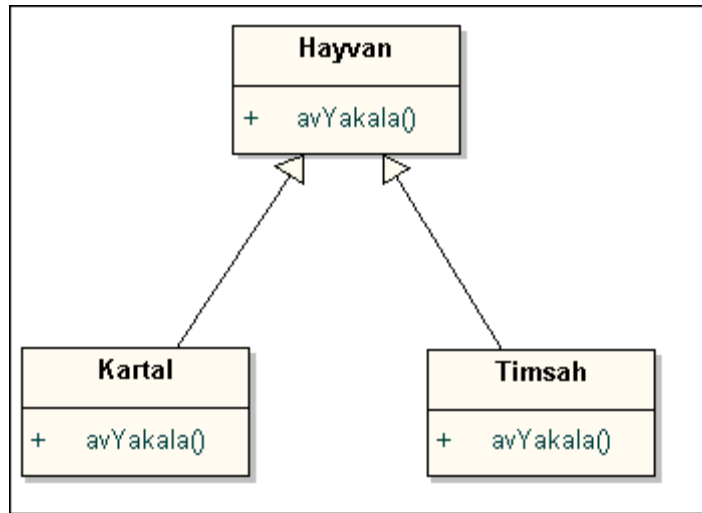
    public static void main(String args[]) {
        Hayvan[] h = new Hayvan[3];
    }
}
```

```

// diziyi doldur
for (int i = 0 ; i < 3 ; i++) {
    h[i] = rasgeleSec(); //upcasting
}
// dizi elemanlarini ekrana bas
for (int j = 0 ; j < 3 ; j++) {
    h[j].avYakala(); // !Dikkat!
}
}
}

```

Yukarıdaki örnekte bulunan kalıtım (*inheritance*) ilişkisini, UML diyagramında gösterirsek:



**Şekil-6.1. Kalıtım, Polimorfizm ve Geç Bağlama**

*PolimorfizmOrnekIki.java* örneğimizde `rasgeleSec()` yordamı, rasgele *Hayvan* nesneleri oluşturup geri döndürmektedir. Geri döndürülen bu *Hayvan* nesneleri, *Hayvan* tipindeki dizi içerisine atılmaktadır. *Hayvan* dizisine atılan *Kartal* ve *Timsah* nesnelerine Java'nın kızmamasındaki sebep kalıttır. *Kartal* **bir** *Hayvan*'dır diyebiliyoruz aynı şekilde *Timsah* **bir** *Hayvan*'dır diyebiliyoruz; olaylara bu açıdan bakarsak *Hayvan* tipindeki dizi içerisine eleman atarken yukarı çevirim (upcasting) olduğunu fark edilir.

Geç bağlama ise, *Hayvan* dizisinin içerisindeki elemanlara ait `avYakala()` yordamını çağırırken karşımıza çıkar. Buradaki ilginç nokta hangi nesnenin `avYakala()` yordamının çağrılacağını derleme anında (*compile-time*) bilinmiyor olmasıdır. Nasıl yani diyenler için konuyu biraz daha açalım. `rasgeleSec()` yordamını incelersek, `Math.random()` yordamının her seferinde 0 ile 2 arasında rasgele sayılar ürettiği görülür. Bu üretilen sayılar doğrultusunda *Hayvan* nesnesi *Kartal* nesnesi veya *Timsah* nesnesi döndürülebilir; bu sebepten dolayı uygulamamızı her çalıştırdığımızda *Hayvan* tipindeki dizinin içerisine değişik tipteki nesnelerin, değişik sırada olabilecekleri görülür. Örneğin *PolimorfizmIki* uygulamamızı üç kere üst üste çalıştırıp çıkan sonuçları inceleyelim; Uygulamamızı çalıştırıyorum.

#### **Gösterim-6.2:**

```
java PolimorfizmIki
```

Uygulamanın çıktısı aşağıdaki gibidir;

```
Kartal avYakala
Hayvan avYakala
Kartal avYakala
```

Aynı uygulamamızı tekrardan çalıştırıyorum;

```
Timsah avYakala
Timsah avYakala
Hayvan avYakala
```

Tekrar çalıştırıyorum;

```
Timsah avYakala
Hayvan avYakala
Kartal avYakala
```

Görüldüğü üzere dizi içerisindeki elemanlar her sefersinde farklı olabilmektedir, dizi içerisindeki elemanlar ancak çalışma anında (*runtime*) belli oluyorlar. `h[j].avYakala()` derken, derleme anında (*compile-time*) hangi nesnenin `avYakala()` yordamının çağrılacağını Java tarafından bilinemez, bu olay ancak çalışma anında (*run-time*) bilinebilir. Geç bağlama özelliği bu noktada karşımıza çıkar. Geç bağlamanın (*late-binding*) diğer isimleri, dinamik bağlama (*dynamic-binding*) veya çalışma anında bağlamadır. (*runtime-binding*).

### 6.3. Final ve Geç Bağlama

5. bölümde, *final* özelliğinin kullanılmasının iki sebebi olabileceğini belirtmiştik. Bunlardan bir tanesi tasarım diğeri ise verimlilik. Verimlilik konusu geç bağlama (*late binding*) özelliği ile aydınlatmış bulunmaktadır, şöyle ki, eğer biz bir sınıfı *final* yaparsak, bu sınıfa ait tüm yordamları *final* yapmış oluruz veya eğer istersek tek başına bir yordamı da *final* yapabiliriz. Bir yordamı *final* yaparak şunu demiş oluruz, bu yordam, türetilmiş olan alt sınıfların içerisindeki diğer yordamlar tarafından iptal edilemesin (*override*) Eğer bir yordam iptal edilemezse o zaman geç bağlama (*late binding*) özelliği de ortadan kalkar.

Uygulama içerisinde herhangi bir nesneye ait normal bir yordam (*final* olmayan) çağrıldığında, Java, acaba doğru nesnenin uygun yordam mı çağrılıyor diye bir kontrol yapar, daha doğrusu geç bağlamaya (*late-binding*) ihtiyaç var mı kontrolü yapılır. Örneğin *Kedi* sınıfını göz önüne alalım. *Kedi* sınıfı *final* olmadığından dolayı bu sınıftan türetilme yapabiliriz.

**Örnek:** *KediKaplan.java*

```
class Kedi {

    public void yakalaAv() {
        System.out.println("Kedi sinifi Av yakaladi");
    }

}

class Kaplan extends Kedi {

    public static void goster(Kedi k) {
        k.yakalaAv();
    }

}
```

```

    public void yakalaAv() {
        System.out.println("Kaplan sinifi Av
yakaladi");
    }

    public static void main(String args[] ) {
        Kedi k = new Kedi() ;
        Kaplan kp = new Kaplan();
        goster(k);
        goster(kp); // yukari dogru cevirim (upcasting)
    }
}

```

*Kaplan* sınıfına ait statik bir yordam olan `goster()` yordamının içerisinde *Kedi* tipindeki `k` yerel değişkene bağlı olan nesnenin, `yakalaAv()` yordamı çağrılmaktadır ama hangi nesnenin `yakalaAv()` yordamı? *Kedi* nesnesine ait olan mı? Yoksa *Kaplan* nesnesine ait olan mı? Java, `yakalaAv()` yordamını çağırmadan evvel geç bağlama (late-binding) özelliğini devreye sokarak, doğru nesneye ait uygun yordamı çağırmaya çalışır tabii bu işlemler sırasından verimlilik düşer. Eğer biz *Kedi* sınıfını `final` yaparsak veya sadece `yakalaAv()` yordamını `final` yaparsak geç bağlama özelliğini kapatmış oluruz böylece verimlilik artar.

**Örnek:** *KediKaplan2.java*

```

class Kedi2 {

    public final void yakalaAv() {
        System.out.println("Kedi sinifi Av yakaladi");
    }

}

class Kaplan2 extends Kedi2 {

    public static void goster(Kedi2 k) {
        // k.yakalaAv(); //! dikkat !
    }

    /* iptal edemez
    public void yakalaAv() {
        System.out.println("Kaplan sinifi Av yakaladi");
    }
    */

    public static void main(String args[] ) {
        Kedi2 k = new Kedi2() ;
        Kaplan2 kp = new Kaplan2();
        goster(k);
        goster(kp);
    }

}

```

*KediKaplan2.java* örneğimizde, `yakalaAv()` yordamını `final` yaparak, bu yordamın *Kaplan* sınıfının içerisinde iptal edilmesini engelleriz; yani, geç bağlama (late binding) özelliği kapatmış oluruz.

## 6.4. Neden Polimorfizm?

Neden polimorfizm sorusuna yanıt aramadan evvel, polimorfizm özelliği olmasaydı olayların nasıl gelişeceğini önce bir görelim. Nesneye yönelik olmayan programlama dillerini kullanan kişiler için aşağıdaki örneğimiz gayet normal gelebilir. `mesaiBasla()` yordamının içerisindeki `if-else` ifadelerine dikkat lütfen.

**Örnek:** *IsYeriNon.java*

```
class Calisan {
    public String pozisyon = "Calisan";
    public void calis() {
    }
}

class Mudur {

    public String pozisyon = "Mudur";

    public Mudur () { //yapilandirici
        pozisyon = "Mudur" ;
    }
    public void calis() { // iptal etme (override)
        System.out.println("Mudur Calisiyor");
    }
}

class Programci {

    public String pozisyon = "Programci";

    public Programci() { //yapilandirici
        pozisyon = "Programci" ;
    }
    public void calis() { // iptal etme (override)
        System.out.println("Programci Calisiyor");
    }
}

class Pazarlamaci {

    public String pozisyon = "Pazarlamaci";

    public Pazarlamaci() { //yapilandirici
        pozisyon = "Pazarlamaci" ;
    }

    public void calis() { // iptal etme (override)
        System.out.println("Pazarlamaci Calisiyor");
    }
}

public class IsYeriNon {
```

```

public static void mesaiBasla(Object[] o ) {

    for ( int i = 0 ; i < o.length ; i++ ) {

        if ( o[i] instanceof Calisan ) {
            Calisan c = (Calisan) o[i] ; //aşağıya çevirim
            c.calis();
        } else if ( o[i] instanceof Mudur ) {
            Mudur m = (Mudur) o[i] ; //aşağıya çevirim
            m.calis();
        } else if ( o[i] instanceof Programci ) {
            Programci p = (Programci) o[i] ; //aşağıya çevirim
            p.calis();
        } else if ( o[i] instanceof Pazarlamaci ) {
            Pazarlamaci paz = (Pazarlamaci) o[i]; //aşağıya
çevirim
            paz.calis();
        }

        //...
    }
}

public static void main(String args[]) {
    Object[] o = new Object[4];
    o[0] = new Calisan();      // yukarı çevirim (upcasting)
    o[1] = new Programci();    // yukarı çevirim (upcasting)
    o[2] = new Pazarlamaci();  // yukarı çevirim (upcasting)
    o[3] = new Mudur();        // yukarı çevirim (upcasting)
    mesaiBasla(o);
}
}

```

Yukarıdaki örneğimizde nesneye yönelik programlamaya yakışmayan davranıştır (OOP), `mesaiBasla()` yordamının içerisinde yapılmaktadır. Bu yordamımızda, dizi içerisindeki elemanların teker teker hangi tipte oldukları kontrol edilip, tiplerine göre `calis()` yordamları çağrılmaktadır. *Calisan* sınıfından türeteceğim her yeni sınıf için `mesaiBasla()` yordamının içerisinde ayrı bir `if-else` koşul ifade yazmak gerçekten çok acı verici bir olay olurdu. Polimorfizm bu durumda devreye girerek, bizi bu zahmet veren işlemden kurtarır. Öncelikle yukarıdaki uygulamamızın çıktısı inceleyelim.

```

Programci Calisiyor
Pazarlamaci Calisiyor
Mudur Calisiyor

```

*IsYeriNon.java* örneğimizi nesneye yönelik programlama çerçevesinde tekrardan yazılırsa

**Örnek:** *IsYeri.java*

```

class Calisan {
    public String pozisyon="Calisan" ;
    public void calis() {}
}

```

```

class Mudur extends Calisan {

    public Mudur () { //yapılandırıcı
        pozisyon = "Mudur" ;
    }
    public void calis() { //iptal etme (override)
        System.out.println("Mudur Calisiyor");
    }
}

class Programci extends Calisan {
    public Programci() { //yapılandırıcı
        pozisyon = "Programci" ;
    }
    public void calis() { //iptal etme (override)
        System.out.println("Programci Calisiyor");
    }
}

class Pazarlamaci extends Calisan {
    public Pazarlamaci() { //yapılandırıcı
        pozisyon = "Pazarlamaci" ;
    }
    public void calis() { //iptal etme (override)
        System.out.println("Pazarlamaci Calisiyor");
    }
}

public class IsYeri {
    public static void mesaiBasla(Calisan[] c ) {
        for (int i = 0 ; i < c.length ; i++) {
            c[i].calis(); //!Dikkat!
        }
    }
    public static void main(String args[]) {
        Calisan[] c = new Calisan[4];
        c[0] = new Calisan(); // yukarı çevirim gerekmiyor
        c[1] = new Programci(); // yukarı çevirim (upcasting)
        c[2] = new Pazarlamaci(); // yukarı çevirim (upcasting)
        c[3] = new Mudur(); // yukarı çevirim (upcasting)
        mesaiBasla(c);
    }
}

```

Görüldüğü üzere mesaiBasla() yordamı artık tek satır, bunu polimorfizm ve tabii ki geç bağlamaya borçluyuz. Bu sayede artık *Calisan* sınıfından istediğim kadar yeni sınıf türetebilirim, yani genişletme olayını rahatlıkla yapabilirim hem de mevcut yapıyı bozmadan. Uygulamanın çıktısında aşağıdaki gibidir.

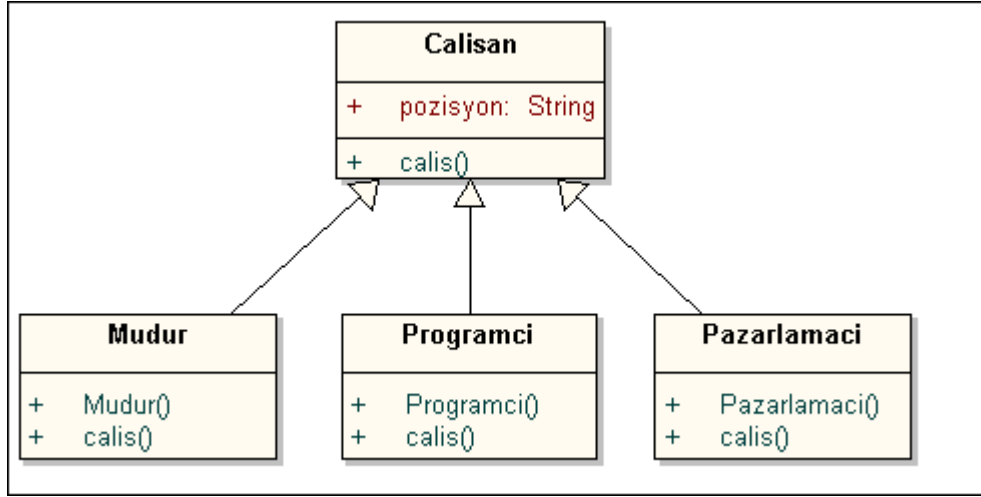
```

Programci Calisiyor
Pazarlamaci Calisiyor
Mudur Calisiyor

```

Bu uygulamadaki sınıflara ait UML diyagramı aşağıdaki gibidir.

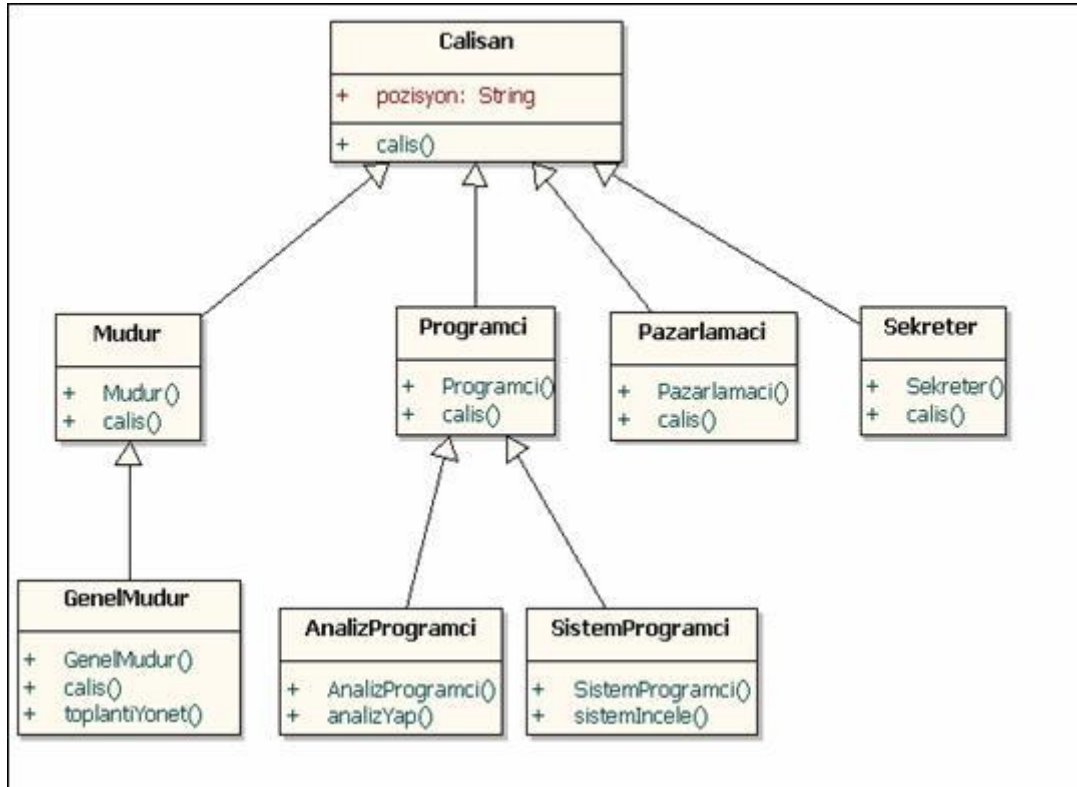




Şekil-6.2. Kullanılan sınıf yapısı

#### 6.5. Genişletilebilirlik (Extensibility)

Polimorfizm sayesinde genişletilebilirlik olayı çok basite indirgenmiş bulunmaktadır. Genişletilebilirlik, mevcut hiyerarşiyi kalıtım yolu ile genişletmedir. Şimdi *IsYeri.java* örneğimizi biraz daha genişletelim; Yeni uygulamamızın adını *BuyukIsYeri.java* yapalım, bu uygulamamız için, sınıflara ait UML diyagramı aşağıdaki gibidir;



Şekil-6.3. Büyük İş Yeri ve Çalışanlar

Yukarıdaki UML diyagramında görüldüğü üzere mevcut hiyerarşiyi genişlettik ve toplam 4 adet yeni sınıfı sistemimize eklendik (*GenelMudur*, *AnalizProgramci*, *SistemProgramci*, *Sekreter*). Yukarıdaki UML şeması Java uygulamasına çevrilirse:

**Örnek:** *BuyukIsyeri.java*

```
class Calisan {
    public String pozisyon ;
    public void calis() {}
}

class Mudur extends Calisan {

    public Mudur () { //yapılandırıcı
        pozisyon = "Mudur" ;
    }
    public void calis() { // iptal etme (override)
        System.out.println("Mudur Calisiyor");
    }
}

class GenelMudur extends Mudur {

    public GenelMudur () { //yapılandırıcı
        pozisyon = "GenelMudur" ;
    }
    public void calis() { // iptal etme (override)
        System.out.println("GenelMudur Calisiyor");
    }
    public void toplantıYonet() {
        System.out.println("GenelMudur toplantı
        yönetiyor");
    }
}

class Programci extends Calisan {

    public Programci() { //yapılandırıcı
        pozisyon = "Programci" ;
    }
    public void calis() { // iptal etme (override)
        System.out.println("Programci Calisiyor");
    }
}

class AnalizProgramci extends Programci {

    public AnalizProgramci() { //yapılandırıcı
        pozisyon = "AnalizProgramci" ;
    }
    public void analizYap() {
        System.out.println("Analiz Yapiliyor");
    }
}

class SistemProgramci extends Programci {

    public SistemProgramci() { //yapılandırıcı
        pozisyon = "SistemProgramci" ;
    }
}
```

```

    public void sistemIncele() {
        System.out.println("Sistem Inceleniyor");
    }
}

class Pazarlamaci extends Calisan {

    public Pazarlamaci() { //yapılandırıcı
        pozisyon = "Pazarlamaci" ;
    }
    public void calis() { //iptal etme (override)
        System.out.println("Pazarlamaci Calisiyor");
    }
}

class Sekreter extends Calisan {

    public Sekreter() { //yapılandırıcı
        pozisyon = "Sekreter" ;
    }
    public void calis() { //iptal etme (override)
        System.out.println("Sekreter Calisiyor");
    }
}

public class BuyukIsYeri {

    public static void mesaiBasla(Calisan[] c ) {
        for (int i = 0 ; i < c.length ; i++) {
            c[i].calis(); //! Dikkat !
        }
    }

    public static void main(String args[]) {
        Calisan[] c = new Calisan[7];
        c[0]=new Calisan(); //yukarı çevirim gerekmiyor
        c[1]=new Programci(); //yukarı çevirim (upcasting)
        c[2]=new Pazarlamaci(); // yukarı çevirim (upcasting)
        c[3]=new Mudur(); // yukarı çevirim (upcasting)
        c[4]=new GenelMudur(); // yukarı çevirim (upcasting)
        c[5]=new AnalizProgramci(); // yukarı çevirim (upcasting)
        c[6]=new SistemProgramci(); // yukarı çevirim (upcasting)
        mesaiBasla(c);
    }
}

```

Yukarıdaki örnekte dikkat edilirse mesaiBasla() yordamı hala tek satır. Uygulamanın çıktısı aşağıdaki gibidir;

```

Programci Calisiyor
Pazarlamaci Calisiyor
Mudur Calisiyor
GenelMudur Calisiyor
Programci Calisiyor
Programci Calisiyor

```

Burada yapılan iş, *Calisan* sınıfından yeni sınıflar türetmektir, bu yeni türetilmiş sınıfların (*GenelMudur*, *AnalizProgramci*, *SistemProgramci*, *Sekreter*) *calis()* yordamlarını çağırmak için ekstra bir yük üstlenilmedi (*mesaiBaslat()* yordamının içerisine dikkat edersek ). Polimorfizm ve tabii ki geç bağlama sayesinde bu işler otomatik olarak gerçekleşmektedir.

## 6.6. Soyut Sınıflar ve Yordamlar (Abstract Classes and Methods)

Soyut kavramını anlatmadan evvel, *IsYeri.java* ve *BuyukIsyeri.java* örneklerini inceleyelim. Bu uygulamaların içerisinde hiç bir iş yapmayan ve sanki boşuna oraya yerleştirilmiş hissi veren bir sınıf göze çarpar; evet bu *Calisan* sınıfıdır. *Calisan* sınıfını daha yakından bakılırsa;

### Gösterim-6.3:

```
class Calisan {  
    public String pozisyon = "Calisan";  
    public void calis() {}  
}
```

Yukarıda görüldüğü üzere *Calisan* sınıfı hiç bir iş yapmamaktadır. Akıllara şöyle bir soru daha gelebilir "Madem ki *Calisan* sınıfı hiç bir iş yapmıyor, ne diye onu oraya yazdık"; cevap: birleştirici bir rol oynadığı için, *Calisan* sınıfını oraya yazdık diyebiliriz. Olayları biraz daha detaylandıralım.

Soyut sınıflar, şu ana kadar bildiğimiz sınıflardan farklıdır. Soyut (*abstract*) sınıflarımızı direk *new()* ile oluşturamayız. Soyut sınıfların var olmasındaki en büyük sebeplerden biri birleştirici bir rol oynamalarıdır. Soyut bir sınıftan türetilmiş alt sınıflara ait nesneler, çok rahat bir şekilde yine bu soyut sınıf tipindeki referanslara bağlanabilirler (yukarı çevirim). Böylece polimorfizm ve geç bağlamanın kullanılması mümkün olur.

Bir sınıfın soyut olması için, bu sınıfın içerisinde en az bir adet soyut yordamının bulunması gerekir. Soyut yordamların gövdesi bulunmaz; yani, içi boş hiçbir iş yapmayan yordam görünümündedirler. Soyut bir sınıftan türetilmiş alt sınıflar, bu soyut sınıfın içerisindeki soyut yordamları kesin olarak iptal etmeleri (*override*) gerekmektedir. Eğer türetilmiş sınıflar, soyut olan ana sınıflarına ait bu soyut yordamları iptal etmezlerse, derleme anında (*compile-time*) hata ile karşılaşılır.

### Gösterim-6.4:

```
abstract void calis() ; // gövdesi olmayan soyut  
yordam
```

Soyut sınıfların içerisinde soyut yordamlar olacağı gibi, gövdeleri olan, yani iş yapan yordamlarda bulunabilir. Buraya kadar anlattıklarımızı bir uygulama üzerinde pekiştirelim;

### Örnek: *AbIsYeri.java*,

```
abstract class Calisan {  
    public String pozisyon="Calisan" ;  
    public abstract void calis() ;// soyut yordam  
    public void zamIste() { // soyut olmayan yordam  
        System.out.println("Calisan zamIste");  
    }  
}
```

```

class Mudur extends Calisan {

    public Mudur () { //yapılandırıcı
        pozisyon = "Mudur" ;
    }
    public void calis() { //iptal etme (override)
        System.out.println("Mudur Calisiyor");
    }
}

class Programci extends Calisan {

    public Programci() { //yapılandırıcı
        pozisyon = "Programci" ;
    }
    public void calis() { //iptal etme (override)
        System.out.println("Programci Calisiyor");
    }

    public void zamIste() { //iptal etme (override)
        System.out.println("Programci Zam Istiyor");
    }
}

class Pazarlamaci extends Calisan {

    public Pazarlamaci() { //yapılandırıcı
        pozisyon = "Pazarlamaci" ;
    }
    public void calis() { //iptal etme (override)
        System.out.println("Pazarlamaci Calisiyor");
    }
}

public class AbIsYeri {

    public static void mesaiBasla(Calisan[] c ) {
        for (int i = 0 ; i < c.length ; i++) {
            c[i].calis(); // !Dikkat!
        }
    }

    public static void main(String args[]) {
        Calisan[] c = new Calisan[3];
        // c[0] = new Calisan(); // soyut sınıflar new ile direk
        oluşturulamazlar
        c[0] = new Programci(); // yukarı çevirim (upcasting)
        c[1] = new Pazarlamaci(); // yukarı çevirim (upcasting)
        c[2] = new Mudur(); // yukarı çevirim (upcasting)
        mesaiBasla(c);
    }
}

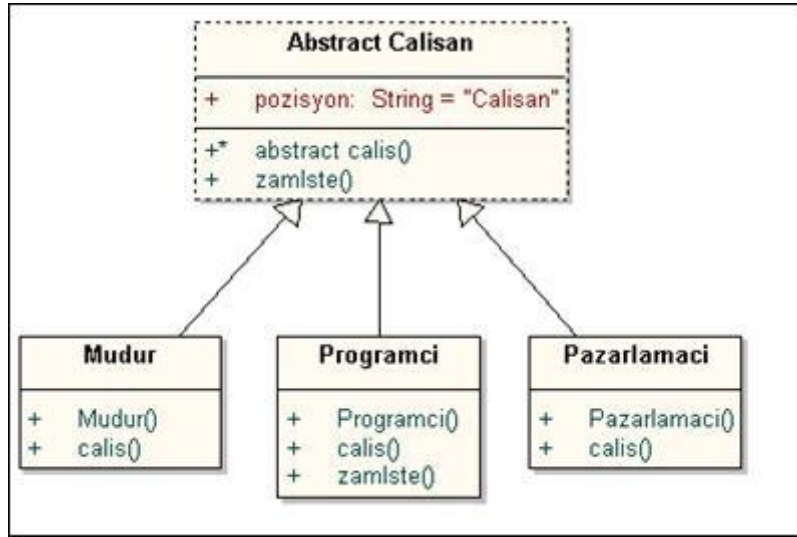
```

Yukarıdaki örneğimizi incelersek, soyut (*abstract*) olan *Calisan* sınıfının 2 adet yordamı olduğu görürüz. Bu iki yordamdan bir tanesi olan soyut (*abstract*) *calis()* yordamıdır. *Calisan* sınıfından türeyen sınıflar

tarafından bu yordam kesin kes\_iptal edilmek (*override*) zorundadır. Baştan açıklarsak, eğer bir sınıfın içerisinde soyut (*abstract*) bir yordam varsa o zaman bu sınıf da soyut (*abstract*) olmak zorundadır. Fakat soyut olan sınıfların içerisinde normal yordamlarda bulunabilir aynı `zamIste()` yordamının *Calisan* sınıfının içerisinde bulunduğu gibi.

*Calisan* sınıfından türeyen diğer sınıfları incelenirse, bu sınıfların hepsinin, `calis()` yordamını iptal ettikleri (*override*) görülür ama aynı zamanda `zamIste()` yordamı sadece *Programci* sınıfının içerisinde iptal edilmiştir (*override*). Eğer ana sınıfın içerisindeki bir yordamın türemiş sınıflar içerisinde iptal edilmeleri (*override*) şansa bırakılmak istenmiyorsa; o zaman bu yordamın soyut olarak tanımlanması gerekir.

Dikkat edilmesi gereken bir başka nokta, soyut sınıfların direk olarak `new()` ile oluşturulamıyor olmasıdır. Soyut sınıf demek birleştirici rol oynayan sınıf demektir. *AbIsYeri.java* uygulamasındaki sınıflara ait UML diyagramı aşağıdaki gibidir;



**Şekil-6.4. AbIsYeri.java uygulamasında kullanılan sınıflar**

UML diyagramından daha net bir biçimde görmekteyiz ki türemiş sınıflar, ana sınıfa ait `calis()` yordamını iptal etmek (*override*) zorunda bırakılmışlardır. Fakat `zamIste()` yordamı ise sadece *Calisan* sınıfından türemiş olan *Programci* sınıfı içerisinde iptal edilmiştir (*override*).

#### 6.6.1. Niye Soyut Sınıf ve Yordamlara İhtiyaç Duyarız?

Örneğin hem cep telefonunun ekranına hem de monitörün ekranına çizgi çizdirmek istiyoruz fakat cep telefonu ekranının özellikleri ile monitör ekranının özelliklerinin birbirinden tamamen farklı olması, karşımızda büyük bir problemdir. Bu iki ekrana çizgi çizdirmek için değişik sınıflara ihtiyaç duyulacağı kesindir. Peki nasıl bir yazılım tasarımı yapılmalıdır.

**Örnek:** *CizimProgrami.java*

```
abstract class Cizim {
    // soyut yordam
    public abstract void noktaCiz(int x , int y) ;

    // soyut olmayan yordam
```

```

    public void çizgiCiz(int x1 , int y1 , int x2 , int
y2) {
        // noktaCiz(x,y); // yordamını kullanarak ekrana çizgi çiz
    }
}

class CepTelefonuCizim extends Cizim {
    // iptal ediyor (override)
    public void noktaCiz(int x, int y) {
        // cep telefonu ekranı için nokta çiz.....
    }
}

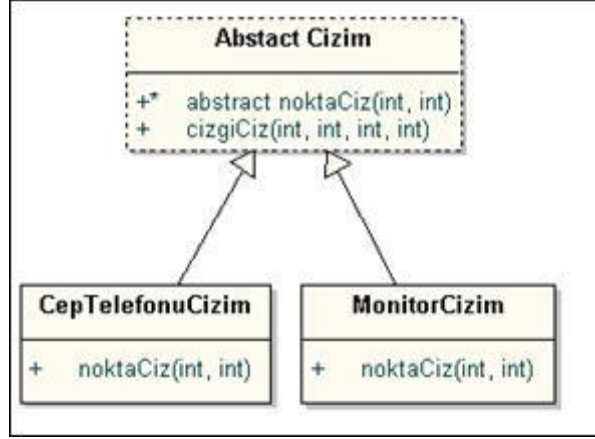
class MonitorCizim extends Cizim {
    // iptal ediyor (override)
    public void noktaCiz(int x, int y) {
        // Monitor ekranı için nokta çiz.....
    }
}

public class CizimProgrami {
    public void baslat(int x1 , int y1 , int x2 , int
y2) {
        // cep telefonunun ekranına çizgi çizmek için
        Cizim c1 = new CepTelefonuCizim();
        c1.cizgiCiz(x1 , y1 , x2 , y2);
        // Monitor ekranına çizgi çizmek için
        Cizim c2 = new MonitorCizim();
        c2.cizgiCiz(x1 , y1 , x2 , y2 );
    }
}

```

*Cizim* sınıfımızın içerisinde bulunan `çizgiCiz()` yordamı soyut (*abstract*) değildir fakat `noktaCiz()` yordamı soyuttur, neden? Sebebi, `çizgiCiz()` yordamının ekranlara çizgi çizmek için `noktaCiz()` yordamına ihtiyaç duymasından kaynaklanır. `çizgiCiz()` yordamının ihtiyaç duyduğu tek şey, ekran üzerinde tek bir noktanın nasıl çizileceğini bilmektir, bu bilgiler `çizgiCiz()` yordamına verildiği sürece sorun yaşanmayacaktır. Ekranı tek bir noktanın nasıl çizileceğini, *Cizim* sınıfından türemiş alt sınıflar tarafından verilmektedir.

*Cizim* sınıfından türemiş sınıflara dikkat edilirse (*CepTelefonuCizim* ve *MonitorCizim*), bu sınıfların içerisinde, ana sınıfa ait olan `noktaCiz()` yordamının iptal edilmiş (*override*) olduğunu görürüz. Bunun sebebi her bir ekrana (Monitörün ve Cep telefonu ekranı) ait nokta çiziminin farklı olmasından kaynaklanır. Yukarıdaki uygulamamıza ait sınıflar için UML diyagramı aşağıdaki gibidir.



**Şekil-6.5. CizimProgrami.java uygulamasında kullanılan sınıflar**

Yukarıdaki örneğimizden çıkartılacak olan ana fikir şöyledir; eğer bir işlem değişik verilere ihtiyaç duyup aynı işi yapıyorsa, bu işlem soyut (*abstract*) sınıfın içerisinde tanımlanmalıdır.

### 6.7. Yapılandırıcılar İçerisindeki İlginç Durumlar

Yapılandırıcılar içerisinde ne gibi ilginç durumlar olabilir ki diyebilirsiniz? Biraz sonra göstereceğimiz örnek içerisinde polimorfizm ve geç bağlamanın devreye girmesiyle olaylar biraz karışacaktır. Öncelikle yapılandırıcıların ne işe yaradıklarını bir tekrar edelim.

Bir nesne kullanıma geçmeden evvel bazı işlemler yapması gerekebilir, örneğin global olan alanlarına ilk değerlerinin verilmesi gerekebilir veya JDBC (ilerleyen bölümlerde inceleyeceğiz) bağlantısı ile veritabanı bağlanıp bazı işlemleri yerine getirmesi gerekebilir, örnekleri çoğaltmak mümkündür... Tüm bu işlemlerin yapılması için gereken yer yapılandırıcılardır. Buraya kadar sorun yoksa örneğimizi incelemeye başlayabiliriz.

**Örnek:** *Spor.java*

```

abstract class Sporcu {
    public abstract void calis();
    public Sporcu() { //yapılandırıcı yordam
        System.out.println("calis() cagrilmadan evvel");
        calis(); //!Dikkat !
        System.out.println("calis() cagrildikten sonra");
    }
}

class Futbolcu extends Sporcu {
    int antraman_sayisi = 4 ;
    public void calis() {
        System.out.println("Futbolcu calis() " +
        antraman_sayisi );
    }
    public Futbolcu() { //yapılandırıcı yordam
        System.out.println("Futbolcu yapilandirici" );
        calis();
    }
}

public class Spor {

```



```

public static void main( String args[] ) {
    Futbolcu f = new Futbolcu();
    // Sporcu s = new Sporcu(); //! Hata soyut sınıf!
}
}

```

Soyut sınıflara ait yapılandırıcılar olabilir. 5. bölümde de incelediği üzere, sınıflara ait nesneleri oluştururken işin içerisinde bir de kalıtım (*inheritance*) özelliği girdiğinde olayların nasıl değiştiği incelemiştik. Bir sınıfa ait nesne oluşturulacaksa, önce bu sınıfın ana sınıfı var mı diye kontrol edilir; yani, bu sınıf türetilmiş bir sınıf mı kontrolü yapılır. Eğer bu sınıfın türetildiği ana bir sınıf var ise önce bu ana sınıfa ait nesne oluşturulur daha sonra sıra türeyen sınıfımıza ait nesnenin oluşturulmasına gelir. Yukarıdaki örneğimizde kalıtım kavramı kullanılmıştır. Ana sınıf *Sporcu* sınıfıdır, bu sınıftan türetilmiş olan ise *Futbolcu* sınıfıdır - Futbolcu **bir** Sporcudur. Biz *Futbolcu* sınıfına ait bir nesne oluşturmak istersek, bu olayın daha öncesinde *Sporcu* sınıfına ait bir nesnenin oluşacağını açıktır.

Bu örneğimizdeki akıl karıştırıcı nokta, soyut bir sınıfa ait yapılandırıcı içerisinde soyut bir yordamın çağrılıyor olmasıdır. *Sporcu* sınıfının yapılandırıcısına dikkat ederseniz, bu yapılandırıcı içerisinde soyut bir yordam olan *calis()* yordamı çağrılmıştır. *calis()* yordamı hangi amaçla soyut yapılmış olabilir? Bu yordamın soyut yapılmasındaki tek amaç, alt sınıfların bu yordamı iptal etmelerini kesinleştirmek olabilir. Bu örneğimizdeki yanlış, soyut bir sınıfa ait yapılandırıcının içerisinde soyut bir yordamın çağrılmasıdır. Peki böyle bir yanlış yapıldığında nasıl sonuçlar oluşur? Uygulamamızın çıktısı aşağıdaki gibidir.

```

calis() cagrilmadan evvel
Futbolcu calis() 0 --> dikkat
calis() cagrildikten sonra
Futbolcu yapilandirici
Futbolcu calis() 4

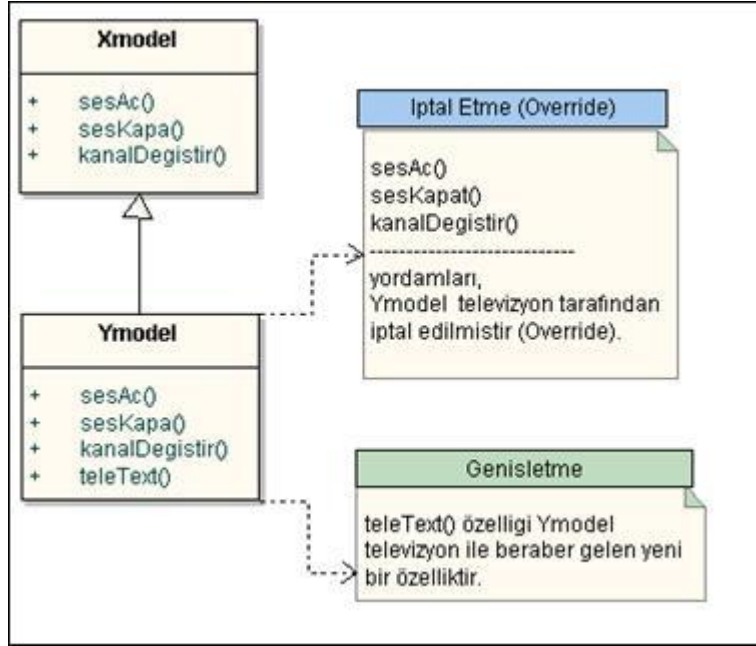
```

Uygulama çıktısındaki 2. satır çok ilginçtir. Oluşan olayları adım adım açıklarsak;

1. *Futbolcu* sınıfına ait bir nesne oluşturulmak istendi ama *Futbolcu* sınıfı *Sporcu* sınıfından türetildiği için, ilk önce **Sporcu** sınıfına ait yapılandırıcı çağrılacaktır.
2. *Sporcu* sınıfına ait yapılandırıcının içerisinde soyut olan *calis()* yordamı çağrıldı. Soyut olan *calis()* yordamı *Futbolcu* sınıfının içerisinde iptal edildiği (override) için, *Futbolcu* sınıfına ait olan *calis()* yordamı çağrılacaktır fakat temel (*primitive*) *int* tipinde olan antreman\_sayisi alanına henüz ilk değeri atanmadığından, Java tarafından varsayılan değer (*default value*) olan 0 sayısı verilmiştir. Buradaki ilginç olan nokta *Futbolcu* sınıfının içerisindeki *calis()* yordamı, *Futbolcu* sınıfına ait yapılandırıcıdan bile önce çağrılmış olmasıdır (-ki bu istenmeyen bir durumdur).
3. Ana sınıf olan *Sporcu* sınıfına ait yapılandırıcının çalışması sona erdikten sonra türetilmiş sınıf olan *Futbolcu* nesnesine ait global alan olan *antreman\_sayisi* alanının ilk değeri verilmiştir.
4. En son olarak *Futbolcu* sınıfına ait yapılandırıcı içerisindeki kodlar çalıştırılarak işlem sona erer.

## 6.8. Kalıtım ve Yukarı Çevirim (Upcasting)

Yukarı çevirim (upcasting) her zaman güvenlidir, sonuçta daha özellikli bir tipten daha genel bir tipe doğru çevirim gerçekleşmiştir. Örneğin elimizde iki tip televizyon bulunsun, biri Xmodel televizyon diğeri Xmodel'den türetilmiş ve daha yeni özelliklere sahip olan Ymodel televizyon. Bu ilişkiyi UML diyagramında gösterirsek.



**Şekil-6.6. Kalıtım ve Yukarı Çevirim**

UML diyagramı Java uygulamasına dönüştürürse;

**Örnek:** *Televizyon.java*

```

class Xmodel {
    public void sesAc() {
        System.out.println("X model televizyon sesAc()");
    }
    public void sesKapa() {
        System.out.println("X model televizyon sesKapa()");
    }
    public void kanalDegistir() {
        System.out.println("X model televizyon kanalDegistir()");
    }
}

class Ymodel extends Xmodel {
    public void sesAc() { // iptal etme (override)
        System.out.println("Y model televizyon sesAc()");
    }
    public void sesKapa() { // iptal etme (override)
        System.out.println("Y model televizyon sesKapa()");
    }
    public void kanalDegistir() { // iptal etme (override)
        System.out.println("Y model televizyon kanalDegistir()");
    }

    public void teleText() {
        System.out.println("Y model televizyon
  
```

```

teleText()");
    }
}

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

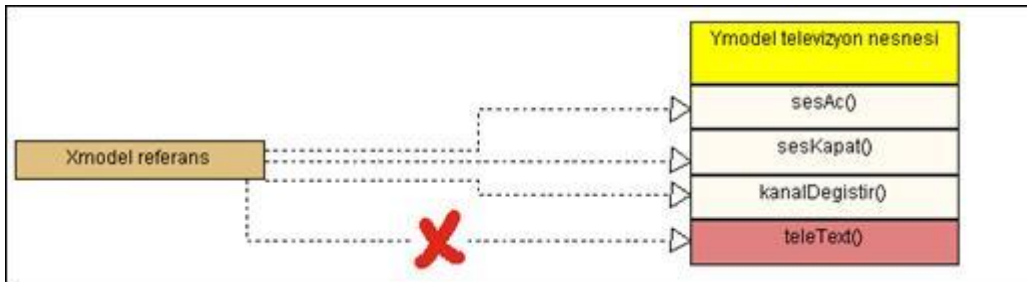
        // yukarı çevirim ( upcasting )
        Xmodel x_model_kumanda = new Ymodel();
        x_model_kumanda.sesAc();
        x_model_kumanda.sesKapa();
        x_model_kumanda.kanalDegistir();

        ///! hata !!, bu kumandanın böyle bir düğmesi yok :)
        // x_model_kumanda.teleText() ;

    }
}

```

Yukarı çevirim (*upcasting*) olayında iki taraf vardır, bir tanesi *heap* alanında nesnenin kendisi diğer tarafta yığın (*stack*) alanında bulunan referans. Olaylara televizyon ve kumanda boyutunda bakarsak işin sırrı çözülmeye başlar. Elimizde *Xmodel* televizyon kumandası olduğunu düşünün ama kumanda *Ymodel* bir televizyonu gösterirsin (gösterebilir çünkü arada kalıtım ilişkisi vardır (*Ymodel* televizyon bir *Xmodel* televizyondur), o zaman karşımızda duran *Ymodel* televizyonun *teleText()* özelliği olmasına rağmen bunu kullanamayız çünkü *Xmodel* bir televizyon kumandası, *Xmodel* televizyon için tasarlandığından, bu kumandanın üzerinde *teleText()* düğmesi olmayacaktır. Anlattıklar şekil üzerinde gösterilirse:



**Şekil-6.7. Kalıtım ve Yukarı Çevirim**

### 6.9. Aşağıya Çevirim (*Downcasting*)

Aşağıya çevirim (*downcasting*), yukarı çevirim (*upcasting*) işleminin tam tersidir. Aşağıya çevirim (*downcasting*), daha genel bir tipten, daha özellikli bir tipe doğru geçiş demektir ve tehlikelidir. Tehlikelidir çünkü çevrilmeye çalışılan daha özellikli tipe doğru çevirim esnasında sorun çıkma riski yüksektir. Java programlama dilinde aşağıya çevirim (*downcasting*) yaparken, hangi tipe doğru çevirim yapılacağı açık olarak belirtmelidir. Fakat yukarı çevirim (*upcasting*) işleminde böyle bir belirteç koyma zorunluluğu yoktur çünkü oradaki olay daha özellikli bir tipten daha genel bir tipe doğru çevirimdir; yani, güvenlidir. Anlattıkları örnek üzerinde gösterirsek:

**Örnek:** *Televizyon2.java*

```

class Xmodel {
    public void sesAc() {
        System.out.println("X model televizyon sesAc()");
    }
    public void sesKapa() {

```

```

        System.out.println("X model televizyon
sesKapa()");
    }
    public void kanalDegistir() {
        System.out.println("X model televizyon
kanalDegistir()");
    }
}

class Ymodel extends Xmodel {
    public void sesAc() { // iptal etme (override)
        System.out.println("Y model televizyon sesAc()");
    }
    public void sesKapa() { // iptal etme (override)
        System.out.println("Y model televizyon
sesKapa()");
    }
    public void kanalDegistir() { // iptal etme (override)
        System.out.println("Y model televizyon
kanalDegistir() ");
    }
    public void teleText() {
        System.out.println("Y model televizyon
teleText()");
    }
}

public class Televizyon2 {
    public static void main(String args[]) {

        Object[] ob = new Object[2] ;
        ob[0] = new Xmodel() ; // yukari cevirim
(upcasting)
        ob[1] = new Ymodel() ; // yukari cevirim
(upcasting)
        for (int i = 0 ; i < ob.length ; i++) {
            // asagiya cevirim (Downcasting)
            Xmodel x_model_kumanda = (Xmodel) ob[i] ;
            x_model_kumanda.sesAc();
            x_model_kumanda.sesKapa();
            x_model_kumanda.kanalDegistir();
            // x_model_kumanda.teleText(); // bu kumanda da böyle
bir düğme yok
            System.out.println("-----
-----");
        }
    }
}

```

Yukarıdaki uygulamanın çıktısı aşağıdaki gibidir;

```

X model televizyon sesAc()
X model televizyon sesKapa()
X model televizyon kanalDegistir()
-----
Y model televizyon sesAc()
Y model televizyon sesKapa()

```

```
Y model televizyon kanalDegistir()
```

Bu örneğimizde, *Xmodel* ve *Ymodel* nesnelerimiz, *Object* sınıfı tipindeki dizinin içerisine atılmaktadır. Her sınıf, *Object* sınıfından türetildiği göre, *Object* sınıfı tipindeki bir dizinin içerisine *Xmodel* ve *Ymodel* nesnelerini rahatlıkla atabiliriz buradaki olay yukarı doğru çevirimdir (*upcasting*). Artık elimizde *Object* sınıfı tipinde bir dizisi var. Bunun içerisindeki elemanları tekrardan eski hallerine sokmak için aşağıya doğru çevirim (*downcasting*) özelliğini kullanmak gereklidir. Daha evvelden de ifade edildiği gibi aşağıya doğru çevirim (*downcasting*) yapılacaksa, bunun hangi tipe olacağı açık olarak belirtilmelidir.

`for` ifadesinin içerisine dikkat edilirse, *Object* tipindeki dizinin içerisindeki elemanları *Xmodel* nesnesine dönüştürmek için aşağıdaki ifade kullanılmıştır.

#### **Gösterim-6.5:**

```
// asagiya dogru cevirim (Downcasting)
Xmodel x_model_kumanda = (Xmodel) ob[i] ;
```

Yukarıdaki ifade sayesinde *Object* sınıfı tipinde olan dizimizin içerisindeki elemanları, *Xmodel* nesnesine çevirmiş bulunmaktayız. Aslında bu örneğimizde zarardayız, niye dersiniz hemen açıklayalım. Sonuçta bizim iki adet hem *Xmodel* hem de *Ymodel* nesnelerimiz vardı. Biz bunları *Object* sınıfı tipine çevirerek yani yukarı çevirim yaparak *Object* sınıfı tipindeki dizinin içerisine yerleştirdik, buraya kadar sorun yok. Fakat *Object* dizisinin içerisinden elemanlarımızı çekerken, bu elemanlarımızın hepsini *Xmodel* nesnesine çevirdik yani aşağıya çevirim yapmış olduk. Biz böyle yapınca *Ymodel* nesnemiz yok olmuş gibi oldu.

Yukarı doğru çevirim (*upcasting*) yaparken, asıl nesnelerimiz değerlerinden birşey kaybetmezler. Örneğin bu uygulamamızda bir *Object* sınıfı tipindeki dizimizin içerisine *Xmodel* ve *Ymodel* nesnelerini atabildik (bunun sebebinin kalıttır). *Xmodel* ve *Ymodel* nesnelerini, *Object* nesnelere çevirerek, bu nesnelerimizin asıl tiplerini değiştirmeyiz, sadece *Object* dizisinin içerisine atma izni elde ederiz; yani, *Object* sınıfı tipindeki dizimizin içerisindeki nesnelerimizin orijinal halleri hala *Xmodel* ve *Ymodel* tipindedir.

Java çalışma anında (*run-time*) nesnelerin tiplerini kontrol eder. Eğer bu işlemlerde bir uyumsuzluk varsa bunu hemen kullanıcıya *ClassCastException* istisnası fırlatarak bildirir. Nesne tiplerinin çalışma anından (*run-time*) tanımlanması (*RTTI : Run Time Type Identification*), kodu yazan kişi açısından büyük faydalar içerir. Bu açıklamalardan yola çıkılarak, yukarıdaki uygulama değiştirilirse.

#### **Örnek:** *Televizyon3.java*

```
class Xmodel {
    public void sesAc() {
        System.out.println("X model televizyon sesAc()");
    }

    public void sesKapa() {
        System.out.println("X model televizyon
sesKapa()");
    }

    public void kanalDegistir() {
        System.out.println("X model televizyon
kanalDegistir()");
    }
}
```

```

}

class Ymodel extends Xmodel {
    public void sesAc() { //iptal ediyor (override)
        System.out.println("Y model televizyon sesAc()");
    }

    public void sesKapa() { //iptal ediyor (override)
        System.out.println("Y model televizyon
sesKapa()");
    }

    public void kanalDegistir() { //iptal ediyor (override)
        System.out.println("Y model televizyon
kanalDegistir() ");
    }

    public void teleText() {
        System.out.println("Y model televizyon
teleText()");
    }
}

public class Televizyon3 {
    public static void main(String args[]) {

        Object[] ob = new Object[2] ;
        ob[0] = new Xmodel() ;
        ob[1] = new Ymodel() ;

        for (int i = 0 ; i < ob.length ; i++) {

            Object o = ob[i] ;
            if (o instanceof Ymodel) { //RTTI
                Ymodel y_model_kumanda = (Ymodel) o ;
//artık güvende
                y_model_kumanda.sesAc();
                y_model_kumanda.sesKapa();
                y_model_kumanda.kanalDegistir();
                y_model_kumanda.teleText() ;
            } else if (o instanceof Xmodel) { //RTTI
                Xmodel x_model_kumanda = (Xmodel) o; //
artık güvenli
                x_model_kumanda.sesAc();
                x_model_kumanda.sesKapa();
                x_model_kumanda.kanalDegistir();

            }
        }
    }
}

```

*Object* sınıfı tipindeki dizi içerisinde bulunan elemanların hepsinin *Object* sınıfı tipinde olma zorunluluğu olduğunu 3. bölümdeki diziler başlığında incelemiştik. *Xmodel* ve *Ymodel* televizyonları yukarı doğru çevirim özelliği sayesinde *Object* sınıfı tipine çevirerek, *Object* sınıfı tipindeki dizi içerisine atabildik. Peki *Xmodel* ve *Ymodel* nesnelerimizin özelliklerini sonsuza kadar geri alamayacak mıyız? Geri almanın yolu aşağıya çevirimdir (*downcasting*).

Aşağıya çevirimin (*downcasting*) tehlikeli olduğunu biliyoruz. Eğer yanlış bir tipe doğru çevirim yaparsak, çalışma anında Java tarafından *ClassCastException* istisnası (ilerleyen bölümlerde inceleyeceğiz) ile durduruluruz. Çalışma anında (run-time) yanlış bir tipe çevirimden korkuyorsak *instanceof* anahtar kelimesini kullanmamız gerekir. Yukarıdaki örneğimizde (*Televizyon3.java*) *instanceof* anahtar kelimesi sayesinde çalışma anında, *Object* sınıfı tipindeki dizi içerisindeki elemanların asıl tiplerini kontrol ederek aşağıya doğru çevirim yapma imkanına sahip oluruz. Böylece hata oluşma riskini minimuma indiririz. Uygulamamızın çıktısı aşağıdaki gibidir;

```
X model televizyon sesAc()  
X model televizyon sesKapa()  
X model televizyon kanalDegistir()  
Y model televizyon sesAc()  
Y model televizyon sesKapa()  
Y model televizyon kanalDegistir()  
Y model televizyon teleText()
```

**Bu dökümanın her hakkı saklıdır.**

**© 2004**



## ARAYÜZLER VE DAHİLİ SINIFLAR

## (Interface and Inner Classes)

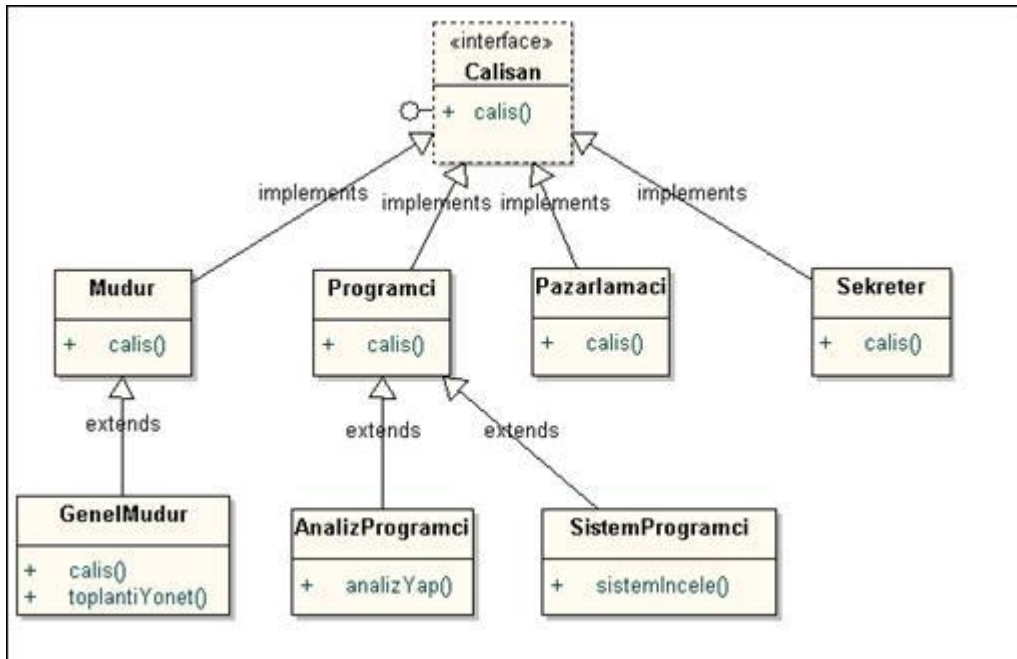
Diğer programlama dillerinde olan çoklu kalıtım (*multiple inheritance*) özelliği Java programlama dilinde yoktur. Java programlama dilinde çoklu kalıtım desteğinden faydalanmak için arayüz (*interface*) ve dahili sınıflar (*inner classes*) kullanılır.

## 7.1. Arayüz (Interface)

Arayüzler, soyut (abstract) sınıfların bir üst modeli gibi düşünelebilir, soyut sınıfların içerisinde hem iş yapan hem de hiçbir iş yapmayan sadece birleştirici rol üstlenen gövdesiz yordamlar (soyut yordamlar-*abstract methods*) vardı. Bu birleştirici rol oynayan yordamlar, soyut sınıftan (*abstract class*) türetilmiş alt sınıfların içerisinde iptal edilmeleri (*override*) gerektiğini geçen bölümde incelenmişti. Arayüzlerin içerisinde ise iş yapan herhangi bir yordam (*method*) bulunamaz; arayüzün içerisinde tamamen gövdesiz yordamlar (soyut yordamlar) bulunur. Bu açıdan bakılacak olursak, arayüzler, birleştirici bir rol oynamaları için tasarlanmıştır. Önemli bir noktayı hemen belirtelim; arayüzlere ait gövdesiz (*soyut*) yordamlar otomatik olarak `public` erişim belirleyicisine sahip olurlar ve sizin bunu değiştirme imkanınız yoktur. Aynı şekilde arayüzlere ait global alanlarda otomatik `public` erişim belirleyicisine sahip olurlar ek olarak, bu alanlar yine otomatik olarak `final` ve `statik` özelliği içerirler ve sizin bunlara yine müdahale etme imkanınız yoktur.

## 7.1.1. Birleştiricilik

Bölüm-6'da verilen *BüyükIsYeri.java* örneğini, arayüzleri kullanarak baştan yazmadan önce, yeni UML diyagramını inceleyelim;





### Şekil-7.1. Birleştiricilik

UML diyagramında görüldüğü üzere, *Calisan* arayüzü (*interface*), birleştirici bir rol oynamaktadır. *Calisan* arayüzünde tanımlanan ve soyut (gövdesiz) *calis()* yordamı (*method*), bu arayüze erişen tüm sınıfların içerisinde iptal edilmek zorundadır (*override*). UML diyagramımızı Java uygulamasına dönüştürülürse;

**Örnek:** *BuyukIsYeri.java*

```
interface Calisan { //arayuz
    public void calis() ;
}

class Mudur implements Calisan {
    public void calis() { // iptal etti (override)
        System.out.println("Mudur Calisiyor");
    }
}

class GenelMudur extends Mudur {
    public void calis() { // iptal etti (override)
        System.out.println("GenelMudur Calisiyor");
    }
    public void toplantıYonet() {
        System.out.println("GenelMudur toplantı
        yonetiyor");
    }
}

class Programci implements Calisan {
    public void calis() { // iptal etti (override)
        System.out.println("Programci Calisiyor");
    }
}

class AnalizProgramci extends Programci {
    public void analizYap() {
        System.out.println("Analiz Yapiliyor");
    }
}

class SistemProgramci extends Programci {
    public void sistemIncele() {
        System.out.println("Sistem Inceleniyor");
    }
}

class Pazarlamaci implements Calisan {
    public void calis() { // iptal etti (override)
        System.out.println("Pazarlamaci Calisiyor");
    }
}

class Sekreter implements Calisan {
    public void calis() { // iptal etti (override)
        System.out.println("Sekreter Calisiyor");
    }
}
```

```

}

public class BuyukIsYeri {
    public static void mesaiBasla(Calisan[] c ) {
        for (int i = 0 ; i < c.length ; i++) {
            c[i].calis(); //! Dikkat !
        }
    }

    public static void main(String args[]) {
        Calisan[] c = new Calisan[6];
        // c[0]=new Calisan(); ! Hata ! arayüz oluşturulamaz
        c[0]=new Programci(); // yukari cevrim (upcasting)
        c[1]=new Pazarlamaci(); // yukari cevrim (upcasting)
        c[2]=new Mudur(); // yukari cevrim (upcasting)
        c[3]=new GenelMudur(); // yukari cevrim (upcasting)
        c[4]=new AnalizProgramci(); // yukari cevrim (upcasting)
        c[5]=new SistemProgramci(); // yukari cevrim (upcasting)
        mesaiBasla(c);
    }
}

```

Yukarıdaki örneğimiz ilk etapta çekici gelmeyebilir, “Bunun aynısı soyut sınıflarla (*abstract class*) zaten yapılabiliyordu. Arayüzleri neden kullanayım ki... “ diyebilirsiniz. Yukarıdaki örnekte arayüzlerin nasıl kullanıldığı incelenmiştir; arayüzlerin sağladığı tüm faydalar birazdan daha detaylı bir şekilde incelenecektir.

Arayüzlerin devreye sokulmasını biraz daha yakından bakılırsa.

### **Gösterim-7.1:**

```

class Mudur implements Calisan {
    public void calis() { // iptal etti (override)
        System.out.println("Mudur Calisiyor");
    }
}

```

Olaylara *Mudur* sınıfının bakış açısından bakılsın. Bu sınıf *Calisan* arayüzünün gövdesiz yordamlarını iptal etmek (override) istiyorsa, *Calisan* arayüzüne ulaşması gerekir. Bu ulaşım *implements* anahtar kelimesi ile gerçekleşir. *Mudur* sınıfı bir kere *Calisan* arayüzüne ulaştı, buradaki gövdesiz yordamları (soyut yordamları) kesin olarak iptal etmek (override) zorundadır. Uygulamanın çıktısı aşağıdaki gibidir;

```

Programci Calisiyor
Pazarlamaci Calisiyor
Mudur Calisiyor
GenelMudur Calisiyor
Programci Calisiyor
Programci Calisiyor

```

### **7.1.2. Arayüz (Interface) ve Soyut Sınıflar (Abstract Classes)**

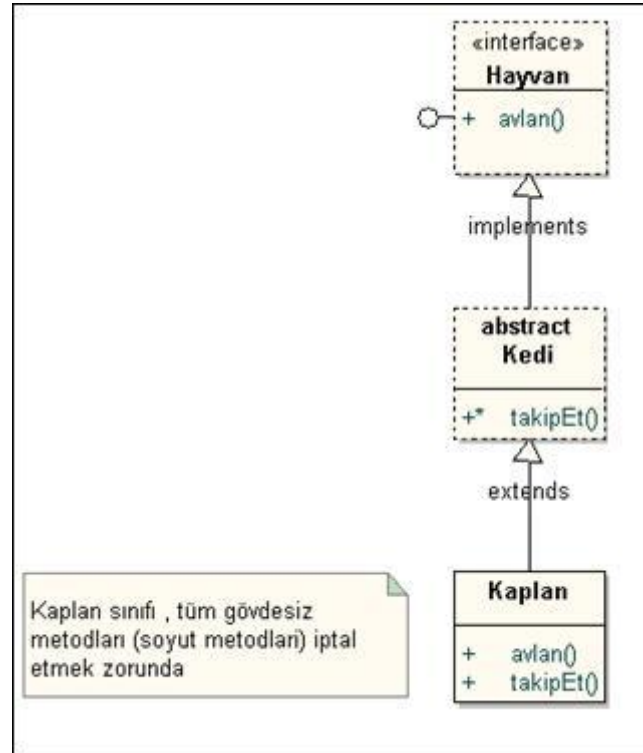
Eğer bir sınıf (soyut sınıflar dahil) bir arayüze (*interface*) ulaşmak istiyorsa, bunu *implements* anahtar kelimesi ile gerçekleştirebileceğini belirtmiştik. Ayrıca eğer bir sınıf bir kere arayüze ulaştı artık onun

tüm gövdesiz yordamlarını (soyut yordamlar) kesin olarak iptal etmesi (override) gerektiğini de belirttik. Peki eğer soyut bir sınıf (*abstract class*) bir arayüze ulaşırsa, arayüze ait gövdesiz yordamları kesin olarak, kendi içerisinde iptal etmeli mi? Bir örnek üzerinde incelersek;

**Örnek:** *Karisim.java*

```
interface Hayvan {  
    public void avlan() ;  
}  
  
abstract class Kedi implements Hayvan {  
}
```

Yukarıdaki örneğimizi derleyebilir (*compile*) miyiz? Derlense bile çalışma anında (*run-time*) hata oluşturur mu? Aslında olaylara kavramsal olarak bakıldığında çözüm yakalanmış olur. Soyut sınıfların amaçlarından bir aynı arayüz özelliğinde olduğu gibi birleştirici bir rol oynamaktır. Daha açık bir ifade kullanırsak, hem arayüzler olsun hem de soyut sınıflar olsun, bunların amaçları kendilerine ulaşan normal sınıflara, kendilerine ait olan gövdesiz yordamları iptal ettirmektir (*override*). O zaman yukarıdaki örnekte soyut olan *Kedi* sınıfı, *Hayvan* arayüzüne (*interface*) ait gövdesiz (soyut) *avlan()* yordamını iptal etmek zorunda değildir. Daha iyi anlaşılması açısından yukarıdaki örneği biraz daha geliştirelim ama öncesinde UML diyagramını çıkartalım;



**Şekil-7.2. Arayüzler ve Soyut Sınıflar**

UML diyagramından görüleceği üzere, *Kaplan* sınıfı, *avlan()* ve *takipEt()* yordamlarını (gövdesiz-soyut yordamlarını) iptal etmek zorundadır. UML diyagramını Java uygulamasına dönüştürülürse;

**Örnek:** *Karisim2.java*

```
interface Hayvan {
```

```

    public void avlan() ;
}

abstract class Kedi implements Hayvan {
    public abstract void takipEt() ;
}

class Kaplan extends Kedi {
    public void avlan() { // iptal etti (override)
        System.out.println("Kaplan avlanıyor...");
    }

    public void takipEt() { // iptal etti (override)
        System.out.println("Kaplan takip ediyor...");
    }
}

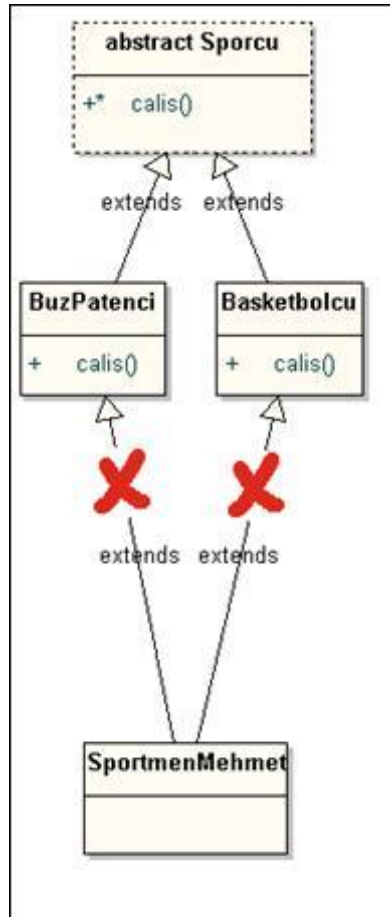
```

Soyut (*abstract*) olan *Kedi* sınıfının içerisinde, herhangi bir gövdesiz yordam (soyut yordam) iptal edilmemiştir (*override*). İptal edilme işlemlerinin gerçekleştiği tek yer *Kaplan* sınıfının içerisidir. Soru: *Kaplan* sınıfı *Hayvan* arayüzünde (interface) tanımlanmış soyut olan (gövdesiz) *avlan()* yordamını iptal etmek (*override*) zorunda mı? Cevap: Evet, çünkü *Kaplan* sınıfı *Kedi* sınıfından türetilmiştir. *Kedi* sınıfı ise *Hayvan* arayüzüne ulaşmaktadır. Bu sebepten dolayı *Kaplan* sınıfının içerisinde *avlan()* yordamı iptal edilmelidir.

En baştaki sorumuzun cevabı olarak, Karisim.java örneğimiz rahatlıkla derlenebilir (*compile*) ve çalışma anında (*run-time*) herhangi bir çalışma-anı istisnasına (*runtime-exception*) sebebiyet vermez. (Not: İstisnaları (Exception) 8. bölümde detaylı bir şekilde anlatılmaktadır.)

### 7.1.3. Arayüz (Interface) İle Çoklu Kalıtım (Multiple Inheritance)

İlk önce çoklu kalıtımın (*multiple inheritance*) niye tehlikeli ve Java programlama dili tarafından kabul görmediğini inceleyelim. Örneğin *Sporcu* soyut sınıfından türetilmiş iki adet sınıfımız bulunsun, *BuzPatenci* ve *Basketbolcu* sınıfları. Bu iki sınıftan türetilen yeni bir sınıf olamaz mı? Örneğin *SportmenMehmet* sınıfı; yani, *SportmenMehmet* sınıfı aynı anda hem *BuzPatenci*, hem de *Basketbolcu* sınıfından türetilir mi? Java programlama dilinde türetilemez. Bunun sebeplerini incelemeden evvel, hatalı yaklaşımı UML diyagramında ifade edilirse;



**Şekil-7.3. Çoklu Kalıtımın (Multiple Inheritance) Sakıncaları**

Java programlama dili niye çoklu kalıtımı bu şekilde desteklemez? UML diyagramını, hatalı bir Java uygulamasına dönüştürülürse;

**Örnek:** *Spor.java*

```
abstract class Sporcu {
    public abstract void calis();
}

class BuzPatenci extends Sporcu {
    public void calis() {
        System.out.println("BuzPatenci calisiyor....") ;
    }
}

class Basketbolcu extends Sporcu {
    public void calis() {
        System.out.println("Basketbolcu calisiyor....") ;
    }
}

/*
Bu ornegimiz derleme aninda hata alicaktir.
Java, coklu kalitimi desteklemez
*/
```

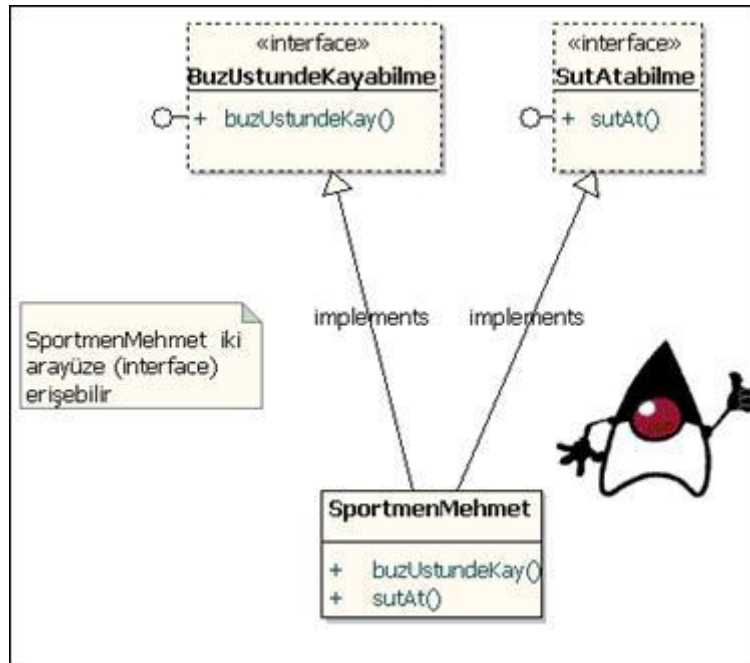
```
class SportmenMehmet extends BuzPatenci, Basketbolcu {  
}
```

*Spor.java* derleme anında hata alacaktır. Bu ufak ayrıntıyı belirttikten sonra, kaldığımız yerden devam edelim. Java'nın niye çoklu kalıtımı (*multiple inheritance*) desteklemediğini anlamak için aşağıdaki gösterim incelenmelidir.

#### **Gösterim-7.2:**

```
Sporcu s = new SportmenMehmet(); // yukari dogru cevrim  
s.calis(); // ??
```

Herhangi bir yerden, yukarıdaki gösterimde belirtildiği gibi bir ifade yazılmış olsa, sonuç nasıl olurdu? *Sporcu* tipinde olan referans, *SportmenMehmet* nesnesine bağlanmıştır (bağlanabilir çünkü arada kalıtım ilişkisi vardır). Fakat burada *s.calis()* ifadesi yazılırsa, hangi nesnenin *calis()* yordamı çağrılacaktır? *BuzPatenci* nesnesinin mi? Yoksa *Basketbolcu* nesnesinin mi? Sonuçta, *calis()* yordamı, *BuzPatenci* ve *Basketbolcu* sınıflarının içerisinde iptal edilmiştir. Bu sorunun cevabı yoktur. Fakat çoklu kalıtımın bu zararlarından arıtılmış versiyonunu yani arayüzleri (interface) ve dahili sınıflar (inner classes) kullanarak, diğer dillerinde bulunan çoklu kalıtım desteğini, Java programlama dilinde de bulmak mümkündür. 'Peki ama nasıl?' diyenler için hemen örneğimizi verelim. Yukarıdaki örneğimizi Java programlama diline uygun bir şekilde baştan yazalım ama öncesinde her zaman ki gibi işe UML diyagramını çizmekle başlayalım;



**Şekil-7.4.Arayüzlerin kullanılışı**

*SportmenMehmet*, belki aynı anda hem *BuzPatenci* hemde *Basketbolcu* olamaz ama onlara ait özelliklere sahip olabilir. Örneğin *BuzPatenci* gibi buz üzerinde kayabilir ve *Basketbolcu* gibi şut atabilir. Yukarıdaki UML diyagramı Java uygulamasına dönüştürülürse.

**Örnek:** *Spor2.java*

```

interface BuzUstundeKayabilme {
    public void buzUstundeKay();
}

interface SutAtabilme {
    public void sutAt();
}

class SportmenMehmet implements BuzUstundeKayabilme,
SutAtabilme {
    public void buzUstundeKay() {
        System.out.println("SportmenMehmet buz ustunde
kayiyor");
    }
    public void sutAt() {
        System.out.println("SportmenMehmet sut atiyor");
    }
}

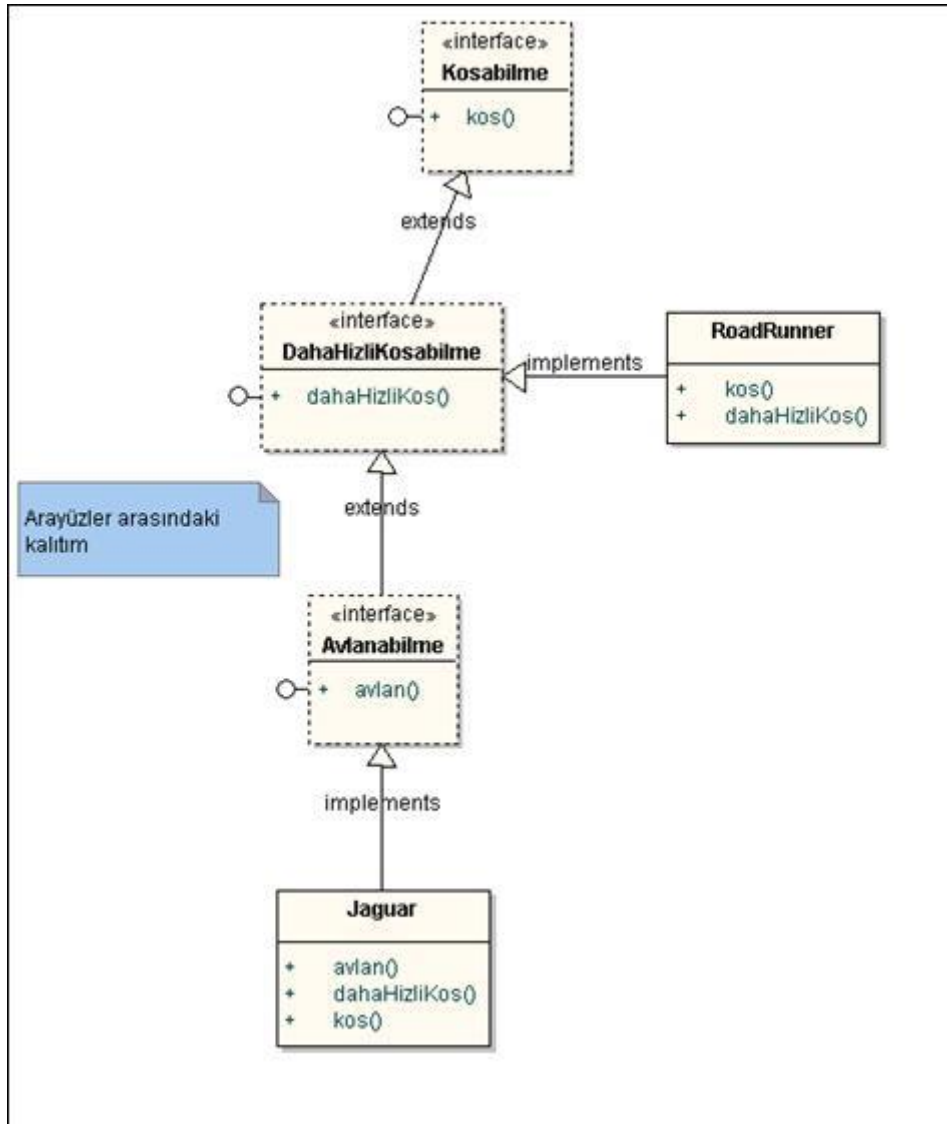
```

Bu örneğimizde *SportmenMehmet*, *BuzUstundeKayabilme* ve *SutAtabilme* özelliklerine sahip olmuştur. Arayüzler içerisindeki (*BuzUstundeKayabilme*, *SutAtabilme*) gövdesiz (soyut) yordamları (*buzUstundeKay()*, *sutAt()*), bu arayüzlere erişen sınıf tarafından kesinlikle iptal edilmelidir (*override*). Eğer iptal edilmez ise, derleme anında (*compile-time*) Java tarafından gerekli hata mesajı verilir.

Örneğimizden anlaşılabilceği üzere arayüz (interface) ile soyut (abstract) sınıf arasında büyük fark vardır. En başta kavramsal olarak bir fark vardır. Bu kavramsal fark nedir dersiniz hemen açıklayalım; Soyut bir sınıftan türetilme yapıldığı zaman, türetilen sınıf ile soyut sınıf arasında mantıksal bir ilişki olması gerekirdi, örnek vermek gerekirse "Yarasa **bir** Hayvandır" gibi veya "Müdür **bir** Çalışandır" gibi....Geçen bölümlerde incelediğimiz **bir** ilişkisi. Fakat arayüzler ile bunlara erişen sınıflar arasında kalıtsal bir ilişki bulunmayabilir.

#### 7.1.4. Arayüzlerin Kalıtım (Inheritance) Yoluyla Genişletilmesi

Bir arayüz başka bir arayüzden türetilerek yeni özelliklere sahip olabilir; böylece arayüzler kalıtım yoluyla genişletilmiş olur. Olayları daha iyi anlayabilmek için önce UML diyagramını çizip sonrada Java uygulamasını yazalım.



**Şekil-7.5. Arayüzlerin Kalıtım Yoluyla Genişletilmesi**

*Avlanabilme* arayüzü, *DahaHizliKosabilme* arayüzünden türemiştir. *DahaHizliKosabilme* arayüzünde *Kosabilme* arayüzünde türemiştir. Yukarıdaki UML diyagramımızı Java uygulamasına dönüştürelim;

**Örnek:** *Jaguar.java*

```

interface Kosabilme {
    public void kos();
}

interface DahaHizliKosabilme extends Kosabilme {
    public void dahaHizliKos();
}

interface Avlanabilme extends DahaHizliKosabilme {
    public void avlan();
}

class RoadRunner implements DahaHizliKosabilme {

```



```

    public void kos() {
        System.out.println("RoadRunner kosuyor, bip ");
    }

    public void dahaHizliKos() {
        System.out.println("RoadRunner kosuyor, bip
bippp");
    }
}

public class Jaguar implements Avlanabilme {
    public void avlan() {
        System.out.println("Juguar avlaniyor");
    }

    public void dahaHizliKos() {
        System.out.println("Juguar daha hizli kos");
    }

    public void kos() {
        System.out.println("Juguar Kosuyor");
    }
}

```

*Jaguar* sınıfı *Avlanabilme* arayüzüne ulaşarak, *avlan()*, *dahaHizliKos()*, *kos()* yordamlarını iptal etmek (override) zorunda bırakılmıştır. Bunun sebebi *Avlanabilme* arayüzünün *DahaHizliKosabilme* arayüzünden, *DahaHizliKosabilme* arayüzünde *Kosabilme* arayüzünden türemiş olmasıdır.

*RoadRunner* sınıfı ise sadece *DahaHizliKosabilme* arayüzüne erişerek, *kos()* ve *dahaHizliKos()* gövdesiz (soyut) yordamlarını iptal etmek (override) zorunda bırakılmıştır. Yine bunun sebebi *DahaHizliKosabilme* arayüzünün *Kosabilme* arayüzünden türemiş olmasıdır.

## Arayüzlere Özel

Şimdi birazdan inceleyeceğimiz olay sadece arayüzler söz konusu olduğunda yapılabilir. İlk olayımız açıklayalım; Bir arayüz (*interface*) birden çok arayüzden türetililebilir.

### Gösterim-7.3:

```

interface C {
    //..
}

interface B {
    //..
}

interface A extends B, C {
    //..
}

```

Yukarıdaki gösterimde, *A* arayüzü, birbirlerinden bağımsız iki arayüzden türetilmiş oldu. İkinci olay ise daha evvelde incelenmişti (*bkz:Spor2.java*) ama bir kere daha üzerine basa basa durmakta fayda var; bir sınıf birden fazla arayüze rahatlıkla erişebilir.

### 7.1.5. Çakışmalar

Arayüzlerin içerisinde dönüş tipleri haricinde her şeyleri aynı olan gövdesiz (soyut) yordamlar varsa, bu durum beklenmedik sorunlara yol açabilir. Yazılanları Java uygulaması üzerinde gösterilirse;

**Örnek:** *Cakisma.java*

```
interface A1 {
    public void hesapla();
}

interface A2 {
    public void hesapla(int d);
}

interface A3 {
    public int hesapla();
}

class S1 implements A1,A2 { // sorunsuz
    public void hesapla() { //adas yordamlar(overloaded)
        System.out.println("S1.hesapla");
    }
    public void hesapla(int d) { //adas yordamlar(overloaded)
        System.out.println("S1.hesapla " + d );
    }
}

/*
! Hatali !, adas yordamlar (overloaded)
donus tiplerine gore ayirt edilemez
*/

class S2 implements A1,A3 {
    public void hesapla() {
        System.out.println("S2.hesapla");
    }

    /* !Hata !
    public int hesapla() {
        System.out.println("S2.hesapla");
        return 123;
    }
    */
}
```

Cakisma.java örneğini derlenirse (*compile*), aşağıdaki hata mesajı ile karşılaşılır:

```
Cakisma.java:27: hesapla() in S2 cannot implement
hesapla() in A3; attempting to
    use incompatible return type
found : void
required: int
class S2 implements A1,A3 {
```

```
^
1 error
```

Bu hatanın oluşma sebebi, *A1* ve *A3* arayüzlerinin içerisindeki gövdesiz (soyut) yordamlarından kaynaklanır. Bu yordamların isimleri ve parametreleri aynıdır ama dönüş tipleri farklıdır.

#### **Gösterim-7.4:**

```
public void hesapla(); // A1 arayüzüne ait
public int hesapla(); // A3 arayüzüne ait
```

Bölüm-3’de incelendiği üzere iki yordamın adaş yordam (*overloaded method*) olabilmesi için bu yordamların parametrelerinde kesin bir farklılık olması gerekirdi. İki yordamın dönüş tipleri dışında herşeyleri aynıysa bunlar adaş yordam olamazlar. Olamamalarının sebebi, Java’nın bu yordamları dönüş tiplerine göre ayırt edememesinden kaynaklanır.

#### **7.1.6. Arayüzün (Interface) İçerisinde Alan Tanımlama**

Arayüzlerin içerisinde gövdesiz (soyut) yordamların dışında alanlarda bulunabilir. Bu alanlar uygulamalarda sabit olarak kullanılabilir. Çünkü arayüzün içerisinde tanımlanan bir alan (ilkel tipte veya sınıf tipinde olsun) otomatik olarak hem `public` erişim belirleyicisine hemde `final` ve `static` özelliğine sahip olur.

#### **Örnek:** *AyBul.java*

```
interface Aylar {
    int
    OCAK = 1, SUBAT = 2, MART = 3,
    NISAN = 4, MAYIS = 5, HAZIRAN = 6, TEMMUZ = 7,
    AGUSTOS = 8, EYLUL = 9, EKIM = 10,
    KASIM = 11, ARALIK = 12;
}

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

        int ay = (int)(Math.random()*13) ;
        System.out.println("Gelen ay = " + ay);
        switch ( ay ) {
            case Aylar.OCAK :
                System.out.println("Ocak");break;
            case Aylar.SUBAT :
                System.out.println("Subat");break;
            case Aylar.MART :
                System.out.println("Mart");break;
            case Aylar.NISAN :
                System.out.println("Nisan");break;
            case Aylar.MAYIS :
                System.out.println("Mayis");break;
            case Aylar.HAZIRAN :
                System.out.println("Haziran");break;
            case Aylar.TEMMUZ :
```

```

System.out.println("Temmuz");break;
    case Aylar.AGUSTOS :
System.out.println("Agustos");break;
    case Aylar.EYLUL :
System.out.println("Eylul");break;
    case Aylar.EKIM :
System.out.println("Ekim");break;
    case Aylar.KASIM :
System.out.println("Kasim");break;
    case Aylar.ARALIK :
System.out.println("Aralik");break;
    default: System.out.println("Tanimsiz Ay");
}
}
}

```

#### 7.1.6.1. **Arayüzün İçerisinde Tanımlanmış Alanlara İlk Değerlerinin Verilmesi**

Arayüzlerin içerisinde tanımlanmış alanların ilk değerleri, çalışma anında da (run-time) verilebilir. Aşağıdaki örneği inceleyelim.

**Örnek:** *Test.java*

```

interface A7 {

    int devir_sayisi = (int) ( Math.random() *6 ) ;
    String isim = "A7" ;
    double t1 = ( Math.random() * 8 ) ;
}

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

        System.out.println("devir_sayisi = " +
A7.devir_sayisi );
        System.out.println("isim = " + A7.isim );
        System.out.println("t1 = " + A7.t1 );
    }
}

```

A7 arayüzünün içerisindeki ilkel (primitive) int tipindeki devir\_sayisi ve t1 alanlarının değerlerini derleme anında bilebilmek imkansızdır. Bu değerler ancak çalışma anında bilenebilir.

Dikkat edilmesi gereken bir başka husus ise A7 arayüzünün içerisindeki alanların ne zaman ilk değerlerini aldıklarıdır. Bir arayüzün içerisindeki alanlar final ve statik oldukları için, A7 arayüzüne ilk kez erişildiği zaman, A7 arayüzünün içerisindeki tüm alanlar ilk değerlerini alırlar.

### 7.1.7. Genel Bakış

Arayüzler ve soyut sınıfların bizlere sağlamak istediği fayda nedir? Aslında ulaşılmak istenen amaç çoklu yukarı çevirimdir (*upcasting*). Bir sınıfa ait nesnenin bir çok tipteki sınıf referansına bağlanabilmesi, uygulama içerisinde büyük esneklik sağlar. Bir örnek üzerinde açıklayalım....

**Örnek:** *GenelBakis.java*

```
interface Arayuz1 {
    public void a1() ;
}

interface Arayuz2 {
    public void a2() ;
}

abstract class Soyut1 {
    public abstract void s1();
}

class A extends Soyut1 implements Arayuz1, Arayuz2 {
    public void s1() { //iptal etti (override)
        System.out.println("A.s1()");
    }
    public void a1() { //iptal etti (override)
        System.out.println("A.a1()");
    }
    public void a2() { //iptal etti (override)
        System.out.println("A.a2()");
    }
}

public class GenelBakis {
    public static void main(String args[]) {
        Soyut1 soyut_1 = new A();
        Arayuz1 arayuz_1 = (Arayuz1) soyut_1 ;
        Arayuz2 arayuz_2 = (Arayuz2) soyut_1 ;
        // Arayuz2 arayuz_2 = (Arayuz2) arayuz_1 ; //
dogru

        soyut_1.s1();
        // soyut_1.a1(); //! Hata !
        // soyut_1.a2(); //! Hata !

        arayuz_1.a1();
        // arayuz_1.a2(); //! Hata !
        // arayuz_1.s1(); //! Hata !

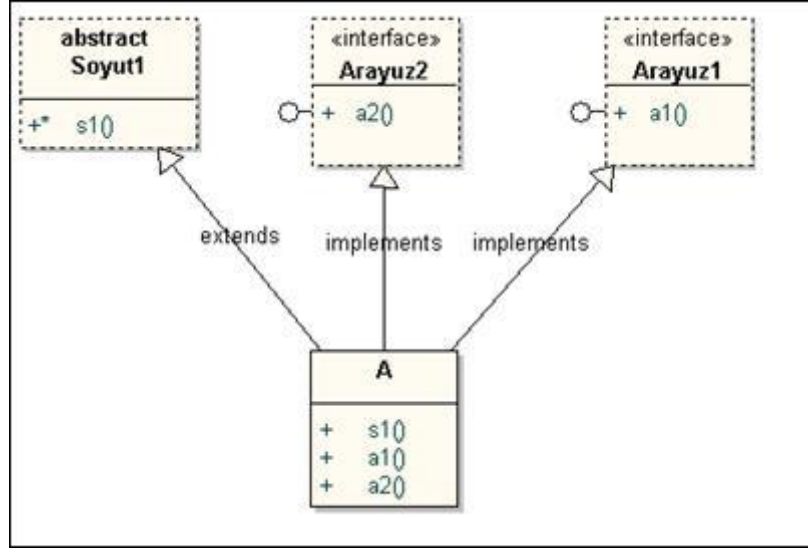
        arayuz_2.a2();
        // arayuz_2.a1(); //! Hata !
        // arayuz_2.s1(); //! Hata !
    }
}
```

A sınıfı *Soyut1* soyut sınıfından türetilmiştir, ayrıca A sınıfı *Arayuz1* ve *Arayuz2* arayüzlerine (interface) erişmektedir. Daha yakından incelenirse.

#### Gösterim-7.5:

```
class A extends Soyut1 implements Arayuz1, Arayuz2 {
```

Yukarıdaki gösterim şunu der: A sınıfına ait bir nesne, *Soyut1* sınıfı, *Arayuz1* veya *Arayuz2* arayüzü tipindeki referanslara bağlanabilir. Anlatılanlar UML diyagramına dönüştürülürse;



**Şekil-7.6. Genel Bakış**

Bu örneğimizde görüldüğü üzere, A sınıfı ait tek bir nesne oluşturulmuştur. Oluşturulan nesne farklı referanslara bağlanabilmektedir.

#### Gösterim-7.6:

```
Soyut1 soyut_1 = new A();
Arayuz1 arayuz_1 = (Arayuz1) soyut_1 ; // tip degisimi
Arayuz2 arayuz_2 = (Arayuz2) soyut_1 ; // tip degisimi
```

Oluşturulan tek bir A sınıfına ait nesne, bir çok farklı sınıf ve arayüz tipindeki referansa bağlanabilmektedir. Örneğin yukarıdaki gösterimde, A sınıfına ait nesnemiz ilk olarak *Soyut1* sınıfı tipindeki referansa bağlanmıştır. Bağlanabilir çünkü A sınıfı *Soyut1* sınıfından türemiştir. *Soyut1* sınıfı tipindeki referansa bağlı olan A sınıfına ait nesnenin sadece *s1()* yordamına ulaşılabilir.

Daha sonra *Soyut1* referansının bağlı olduğu A sınıfına ait nesneyi, *Arayuz1* tipindeki referansa bağlanıyor ama bu bağlama sırasında A sınıfına ait nesneyi *Arayuz1* tipindeki referansa bağlanacağını açık açık söylemek gerekir. A sınıfına ait nesnemiz *Arayuz1* tipindeki referansa bağlanırsa bu nesnenin sadece *a1()* yordamına erişilebilir.

Aynı şekilde *Soyut1* sınıfı tipindeki referansa bağlı olan A sınıfına ait nesne, *Arayuz2* tipindeki referansa bağlanabilir. A sınıfına ait nesne, *Arayuz2* tipindeki referansa bağlanırsa, bu nesnenin sadece *a2()*

yordamına erişebilir. Gösterim-7.6.'deki ifade yerine aşağıdaki gibi bir ifade de kullanılabilir fakat bu sefer üç ayrı A sınıfına ait nesne oluşturmuş olur.

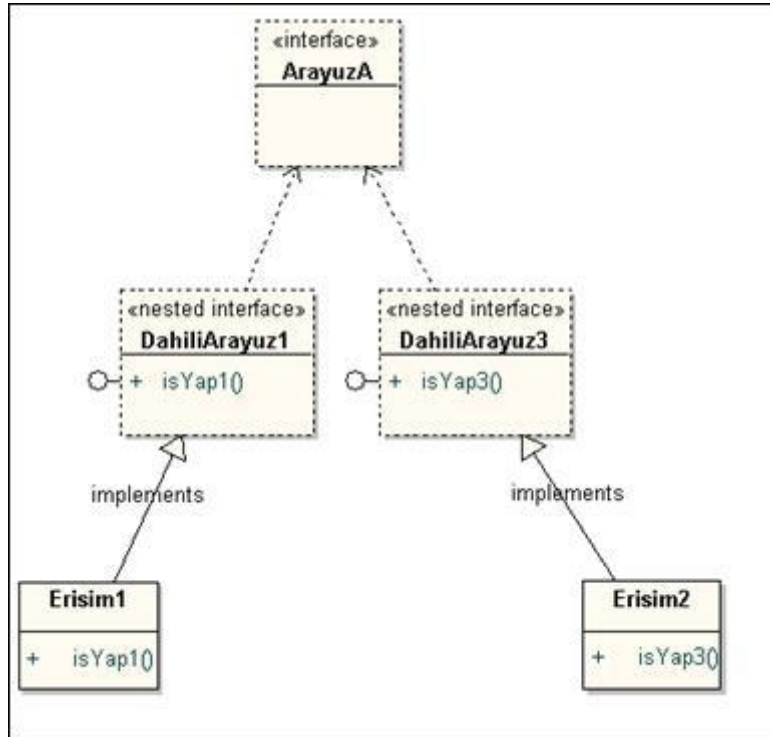
#### **Gösterim-7.7:**

```
Soyut1 soyut_1 = new A();  
Arayuz1 arayuz_1 = new A();  
Arayuz2 arayuz_2 = new A();
```

Görüldüğü üzere, A sınıfına ait üç adet nesne oluşturduk ve bu nesnelerin her birini farklı tipteki referanslara bağlayabildik. Bu olay nesneye yönelik tasarımlar yaparken işimize çokça yarayabilecek bir yaklaşımdır.

#### **7.1.8. Dahili Arayüzler (Nested Interface)**

Bir arayüz, başka bir arayüzün veya sınıfın içerisinde tanımlanabilir. Bir arayüzün içerisinde tanımlanan dahili arayüzler, protected, friendly veya private erişim belirleyicisine sahip olamaz. Örneğimize geçmeden evvel UML diyagramını inceleyelim.



**Sekil-7.7. Dahili arayüzler**

UML diyagramımızdan anlaşılacağı üzere, ArayuzA arayüzünün içerisinde iki adet dahili arayüz (nested interface) tanımlanmıştır. Dışarıdaki iki sınıfımız, dahili olarak tanımlanmış bu iki arayüze erişebilmektedir.

**Örnek:** DahiliArayuzTest.java

```

interface ArayuzA { //aslinda public erişim belirleyicisine sahip
    public interface DahiliArayuz1 {
        public void isYap1() ;
    }

    /* //! Hata !
    protected interface DahiliArayuz2 {
        public void isYap2() ;
    }
    */

    interface DahiliArayuz3 { //aslinda public erişim belirleyicisine sahip
        public void isYap3() ;
    }

    /* //! Hata !
    private interface DahiliArayuz4 {
        public void isYap4() ;
    }
    */
}

class Erisim1 implements ArayuzA.DahiliArayuz1 {
    public void isYap1() {
        System.out.println("Erisim1.isYap1()");
    }
}

class Erisim2 implements ArayuzA.DahiliArayuz3 {
    public void isYap3() {
        System.out.println("Erisim1.isYap3()");
    }
}

public class DahiliArayuzTest {
    public static void main(String args[]) {
        Erisim1 e1 = new Erisim1();
        Erisim2 e2 = new Erisim2();
        e1.isYap1();
        e2.isYap3();
    }
}

```

Dahili arayüzlere erişen sınıflar açısından olaylar aynıdır. Yine bu dahili arayüzlerin içerisindeki gövdesiz yordamları iptal etmeleri gerekmektedir. Uygulamanın çıktısı aşağıdaki gibidir;

```
Erisim1.isYap1()
```

```
Erisim1.isYap3()
```

#### 7.1.8.1. **Sınıfların İçerisinde Tanımlanan Dahili Arayüzler (Nested Interface)**

Bir arayüz diğer bir arayüzün içerisinde tanımlandığı gibi, bir sınıfın içerisinde de tanımlanabilir.



**Örnek:** *SinifA.java*

```
public class SinifA {

    public interface A1 {
        public void ekranaBas();
    } //arayüz

    public class DahiliSinif1 implements A1 {
        public void ekranaBas() {
            System.out.println("DahiliSinif1.ekranaBas()");
        }
    } // class DahiliSinif1

    protected class DahiliSinif2 implements A1 {
        public void ekranaBas() {
            System.out.println("DahiliSinif2.ekranaBas()");
        }
    } // class DahiliSinif2

    class DahiliSinif3 implements A1 {
        public void ekranaBas() {
            System.out.println("DahiliSinif3.ekranaBas()");
        }
    } // class DahiliSinif3

    private class DahiliSinif4 implements A1 {
        public void ekranaBas() {
            System.out.println("DahiliSinif4.ekranaBas()");
        }
    } // class DahiliSinif4

    public static void main(String args[]) {
        SinifA.DahiliSinif1 sad1 = new SinifA().new
DahiliSinif1();
        SinifA.DahiliSinif2 sad2 = new SinifA().new
DahiliSinif2();
        SinifA.DahiliSinif3 sad3 = new SinifA().new
DahiliSinif3();
        SinifA.DahiliSinif4 sad4 = new SinifA().new
DahiliSinif4();

        sad1.ekranaBas();
        sad2.ekranaBas();
        sad3.ekranaBas();
        sad4.ekranaBas();

        SinifB sb = new SinifB();
        sb.ekranaBas();
    }
}

class SinifB implements SinifA.A1{
    public void ekranaBas() {
        System.out.println("SinifB.ekranaBas()");
    }
}
```

*SınıfA* sınıfının içerisinde tanımlanan *AI* arayüzüne, *SınıfB* sınıfından ulaşılabilir. Bir sınıfın içerisinde dahili arayüz tanımlanabildiği gibi dahili sınıfta tanımlanabilir. Bu konu az sonra incelenecektir. Bu örneğimizdeki ana fikir, bir sınıfın içerisinde nasıl dahili arayüzün oluşturulduğu ve bu dahili arayüzün, dahili sınıf olsun veya dışarıdan başka bir sınıf tarafından olsun, nasıl erişilebildiğini göstermektir.

## 7.2. Dahili Sınıflar (Inner Classes)

Dahili sınıflar JDK 1.1 ile gelen bir özelliktir. Bu özellik sayesinde bir sınıf diğer bir sınıfın içerisinde tanımlanabilir; böylece mantıksal bir bütünü oluşturan bir çok sınıf tek bir çatı altında toplanır. Dahili sınıflar yapısal olarak 3 gruba ayrılabilir.

- Dahili üye sınıflar
- Yerel sınıflar (*Local classes*)
- İsimsiz sınıflar (*Anonymous classes*)

### 7.2.1. Dahili Üye Sınıflar

Bir sınıfın içerisinde, başka bir sınıfı tanımlamak mümkündür; Şöyle ki...

#### Gösterim-7.8:

```
class CevreliYiciSinif {  
    class DahiliSinif {  
        //....  
    }  
  
    //...  
}
```

Başka bir sınıfın içerisinde tanımlanan bu sınıfa dahili üye sınıf denir. Dahili sınıfları, çevreleyici sınıfların içerisinde kullanmak, geçen bölümlerde incelediğimiz kompozisyondan yönteminden farklıdır.

Dahili üye sınıflar, tek başlarına bağımsız sınıflar gibi düşünülebilir. Örnek üzerinde incelersek.

#### Örnek: Hesaplama.java

```
public class Hesaplama {  
    public class Toplama { //Dahili uye sinif  
        public int toplamaYap(int a, int b) {  
            return a+b ;  
        }  
    } //class Toplama  
  
    public static void main(String args[]) {  
        Hesaplama.Toplama ht = new Hesaplama().new  
        Toplama() ;  
    }  
}
```

```
int sonuc = ht.toplamaYap(3,5);
System.out.println("Sonuc = " + sonuc );
}
} // class Hesapla
```

*Hesaplama* sınıfının içerisinde tanımlanmış *Toplama* sınıfı bir dahili üye sınıfıdır. *Hesaplama* sınıfı ise çevreleyici sınıftır. *Toplama* sınıfına ait bir nesne oluşturmak için, önce *Hesaplama* sınıfına ait bir nesne oluşturmamız gerekir.

#### **Gösterim-7.9:**

```
Hesaplama.Toplama ht = new Hesaplama().new Toplama() ;
```

ht referansı *Toplama* dahili üye sınıfı tipindedir; artık bu referansı kullanarak *Toplama* nesnesine ait toplamaYap() yordamına ulaşabiliriz. Uygulamanın çıktısı aşağıdaki gibidir;

Sonuc = 8

#### **7.2.1.1. Dahili Üye Sınıflar ve Erişim**

Dahili üye sınıflara, public, friendly, protected veya private erişim belirleyicileri atanabilir, böylece dahili üye sınıflarımıza olan erişimi kısıtlamış/açmış oluruz. Dikkat edilmesi gereken diğer husus ise bir dahili üye sınıf private erişim belirleyicisine sahip olsa dahi, çevreleyici sınıf içerisindeki tüm yordamlar tarafından erişilebilir olmasıdır. Bu kısıt ancak başka sınıflar için geçerlidir.

#### **Örnek: Hesaplama1.java**

```
public class Hesaplama1 {
    public class Toplama { // Dahili üye sınıf - public
        public int toplamaYap(int a, int b) {
            return a + b ;
        }
    } // class Toplama

    protected class Cikartma { // Dahili üye sınıf - protected
        public int cikartmaYap(int a, int b) {
            return a - b ;
        }
    } // class Cikartma

    class Carpma { // Dahili üye sınıf - friendly
        public int carpmaYap(int a, int b) {
            return a * b ;
        }
    } // class Carpma

    private class Bolme { // Dahili üye sınıf - private
        public int bolmeYap(int a, int b) {
            return a / b ;
        }
    } // class Bolme
}
```

```

    public static void main(String args[]) {
        Hesaplama1.Toplama ht = new Hesaplama1().new
        Toplama() ;
        Hesaplama1.Cikartma hck = new Hesaplama1().new
        Cikartma() ;
        Hesaplama1.Carpma hcp = new Hesaplama1().new
        Carpma() ;
        Hesaplama1.Bolme hb = new Hesaplama1().new
        Bolme() ;

        int sonuc1 = ht.toplamaYap(10,5);
        int sonuc2 = hck.cikartmaYap(10,5);
        int sonuc3 = hcp.carpmaYap(10,5);
        int sonuc4 = hb.bolmeYap(10,5);

        System.out.println("Toplama Sonuc = " + sonuc1 );
        System.out.println("Cikartma Sonuc = " + sonuc2
    );
        System.out.println("Carpma Sonuc = " + sonuc3 );
        System.out.println("Bolme Sonuc = " + sonuc4 );
    }
} // class Hesaplama

```

*Hesaplama1* sınıfımızın içerisinde toplam 4 adet dahili üye sınıf mevcuttur. `public` erişim belirleyicisine sahip *Toplama* dahili üye sınıfı, `protected` erişim belirleyicisine sahip *Cikartma* dahili üye sınıfı, `friendly` erişim belirleyicisine sahip *Carpma* dahili üye sınıfı ve `private` erişim belirleyicisine sahip *Bolme* üye dahili sınıfı. *Hesaplama1* sınıfı, bu 4 adet dahili üye sınıfın çevreleyici sınıfıdır. Çevreleyici olan *Hesaplama1* sınıfının statik olan `main()` yordamına dikkat edilirse, bu yordamın içerisinde tüm (*private* dahil) dahili üye sınıflara erişilebildiğini görülür. Bunun sebebi, `main()` yordamı ile tüm dahili üye sınıfların aynı çevreleyici sınıfın içerisinde olmalarıdır. Uygulamanın çıktısı aşağıdaki gibidir:

```

Toplama Sonuc = 15
Cikartma Sonuc = 5
Carpma Sonuc = 50
Bolme Sonuc = 2

```

Yukarıdaki örneğin yeni bir versiyonu yazılıp, dahili üye sınıflar ile bunlara ait erişim belirleyicilerin nasıl işe yaradıklarını incelenirse...

### **Örnek:** *Hesaplama2Kullan.java*

```

class Hesaplama2 {
    public class Toplama2 { // Dahili uye sinif - public
        public int toplamaYap(int a, int b) {
            return a + b ;
        }
    } // class Toplama2

    protected class Cikartma2 { // Dahili uye sinif - protected
        public int cikartmaYap(int a, int b) {
            return a - b ;
        }
    }
}

```

```

    }
} // class Cikartma2

class Carpma2 { // Dahili üye sınıf - friendly
    public int carpmaYap(int a, int b) {
        return a * b ;
    }
} // class Carpma2

private class Bolme2 { // Dahili üye sınıf - private
    public int bolmeYap(int a, int b) {
        return a / b ;
    }
} // class Bolme2

} // class Hesaplama2

public class Hesaplama2Kullan {
    public static void main(String args[]) {

        Hesaplama2.Toplama2 ht=new Hesaplama2().new
        Toplama2() ;
        Hesaplama2.Cikartma2 hck=new Hesaplama2().new
        Cikartma2() ;
        Hesaplama2.Carpma2 hcp = new Hesaplama2().new
        Carpma2() ;
        // Hesaplama2.Bolme3 hb = new Hesaplama2().new
        Bolme2() ;
        // ! Hata !

        int sonuc1 = ht.toplamaYap(10,5);
        int sonuc2 = hck.cikartmaYap(10,5);
        int sonuc3 = hcp.carpmaYap(10,5);
        // int sonuc4 = hb.bolmeYap(10,5); // ! Hata !

        System.out.println("Toplama Sonuc = " + sonuc1 );
        System.out.println("Cikartma Sonuc = " + sonuc2
    );
        System.out.println("Carpma Sonuc = " + sonuc3 );
    }
}

```

*Hesaplama2* sınıfımız, toplam 4 adet olan dahili üye sınıflarının çevreleyicisidir. Dahili üye sınıfları ve onlara ait erişim belirleyicileri incelenirse:

- *Toplama2* sınıfı, `public` erişim belirleyicisine sahip olan dahili üye sınıfıdır.
- *Cikartma2* sınıfı, `protected` erişim belirleyicisine sahip olan dahili üye sınıfıdır.
- *Carpma2* sınıfı, `friendly` erişim belirleyicisine sahip olan dahili üye sınıfıdır.
- *Bolme2* sınıfı, `private` erişim belirleyicisine sahip olan dahili üye sınıfıdır.

*Hesaplama2Kullan* sınıfının statik olan `main()` yordamının içerisinde, *Hesaplama2* sınıfının içerisindeki dahili üye sınıflara erişilebilir mi? Erişilebilir ise hangi erişim belirleyicilerine sahip olan dahili üye sınıflara erişilebilir?

Normalde bir sınıf `private` veya `protected` erişim belirleyicisine sahip olamaz ancak dahili sınıflar `private` veya `protected` erişim belirleyicisine sahip olabilir. *Hesaplama2Kullan* sınıfı, *Hesaplama2* sınıfı ile aynı paket içerisinde (bkz: Bölüm 4-varsayılan paket) olduğu için, *Hesaplama2Kullan* sınıfı, *Hesaplama2* sınıfının içerisinde tanımlanmış olan `public`, `protected` ve `friendly` erişim belirleyicilerine sahip olan dahili üye sınıflara erişebilir ama `private` erişim belirleyicisine sahip olan *Bolme* dahili üye sınıfına erişemez. Uygulamanın çıktısı aşağıdaki gibidir;

```
Toplama Sonuc = 15
Cikartma Sonuc = 5
Carpma Sonuc = 50
```

#### 7.2.1.2. **Dahili Üye Sınıflar ve Bunları Çevreleyen Sınıflar Arasındaki İlişki**

Dahili üye sınıflar, içerisinde bulundukları çevreleyici sınıfların tüm alanlarına (statik veya değil-`private` dahil) ve yordamlarına (statik veya değil-`private` dahil) erişebilirler.

**Örnek:** *Hesaplama3.java*

```
public class Hesaplama3 {
    private int sabit1 = 2 ;
    private static int sabit2 = 1 ;

    public class Toplama3 { //Uye dahili sinif
        public int toplamaYap(int a, int b) {
            return (a+b) + sabit1 ; // dikkat
        }
    } // class Toplama3

    public class Cikartma3 { //Uye dahili sinif
        public int cikartmaYap(int a, int b) {
            dekontBilgileriGoster(); // dikkat
            return (a-b) - sabit2 ; // dikkat
        }
    } // class Cikartma3

    private void dekontBilgileriGoster() {
        System.out.println("Dekont Bilgileri
Gosteriliyor");
    }

    public void ekranaBas(int a , int b ) {
        int sonuc = new Toplama3().toplamaYap(a,b);
        System.out.println("Sonuc = " + a + " + " + b + "
+ sabit1 = "
+ sonuc);
    }

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

```

Hesaplama3 h3 = new Hesaplama3();
h3.ekranaBas(10,5);

// Toplama islemi
Hesaplama3.Toplama3 ht3 = h3.new Toplama3() ;
int sonuc = ht3.toplamaYap(11,6);
System.out.println("Sonuc = 11 + 6 + sabit1 = "
+ sonuc );

// Cikartma islemi
Hesaplama3.Cikartma3 hc3 = h3.new Cikartma3();
int sonuc1 = hc3.cikartmaYap(10,5);
System.out.println("Sonuc = 10 - 5 - sabit2 = " +
sonuc1);
}
} // class Hesaplama3

```

*Hesaplama3* sınıfının içerisinde iki adet dahili üye sınıf bulunmaktadır. Bunlar *Toplama3* ve *Cikartma3* sınıflarıdır. *Toplama3* dahili üye sınıfı, *Hesaplama3* sınıfı içerisinde global olarak tanımlanmış ilkel (primitive) int tipindeki ve `private` erişim belirleyicisine sahip olan `sabit1` alanına erişebilmektedir. *Toplama3* dahili üye sınıfı, *Hesaplama3* sınıfı içerisinde tanımlanmış olan `sabit1` alanını kullanırken sanki kendi içerisinde tanımlanmış bir alanmış gibi, hiç bir belirteç kullanmamaktadır.

Aynı şekilde *Cikartma3* dahili üye sınıfı, *Hesaplama3* sınıfının içerisinde statik olarak tanımlanmış, `private` erişim belirleyicisine sahip ilkel `int` tipindeki `sabit2` alanını ve `private` erişim belirleyicisine sahip `dekontBilgileriGoster()` yordamına direk olarak erişebilmektedir.

*Hesaplama3* sınıfının, nesne yordamı olan (-bu yordamın kullanılabilmesi için *Hesaplama3* sınıfına ait bir nesne oluşturmak gerekir) `ekranaBas()`, iki adet parametre alıp, geriye hiçbirşey döndürmez (void). Bu yordamın içerisinde *Toplama3* dahili üye sınıfına ait nesne oluşturularak, bu dahili üye sınıfın `toplamaYap()` yordamı çağrılmaktadır. *Toplama3* dahili üye sınıfının `toplamaYap()` yordamından dönen cevap, `ekranaBas()` yordamının içerisinde ekrana bastırılır.

Dikkat edilmeye değer diğer bir husus ise sadece bir adet çevreleyici sınıfa ait nesne oluşturup, Bu nesneye bağlı referansı kullanarak, çevreleyici sınıf içerisindeki diğer dahili üye sınıflara ait nesnelerin oluşturulmasıdır. Olaylara daha yakından bakılırsa;

#### **Gösterim-7.10:**

```

Hesaplama3 h3 = new Hesaplama3();
Hesaplama3.Toplama3 ht3 = h3.new Toplama3() ;
Hesaplama3.Cikartma3 hc3 = h3.new Cikartma3();

```

Sadece bir adet *Hesaplama3* sınıfına ait nesne oluşturuldu. Bu nesneye bağlı referansı kullanarak (`h3`), diğer dahili üye sınıflara ait nesneler oluşturulabilir. Buradaki anafikir, çevreleyici sınıfların içerisinde bulunan her dahili üye sınıfa ait bir nesne oluşturmak için, her seferinde yeni bir çevreleyici sınıfa ait nesne oluşturma zorunluluğu olmadığıdır. Yani çevreleyici sınıfa ait bir nesne, yine çevreleyici sınıf tipindeki bir referansa bağlanırsa, işler daha kestirmeden çözülebilir. Uygulamanın çıktısı aşağıdaki gibidir;

```

Sonuc = 10 + 5 + sabit1 = 17
Sonuc = 11 + 6 + sabit1 = 19
Dekont Bilgileri Gosteriliyor
Sonuc = 10 - 5 - sabit2 = 4

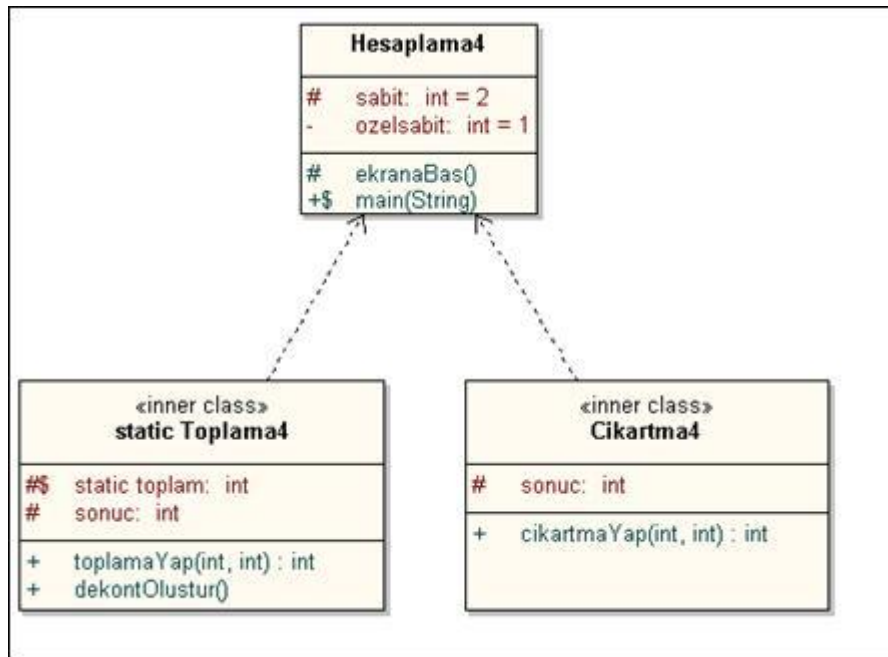
```

### 7.2.1.3. Statik Dahili Üye Sınıflar

Statik (`static`) olarak tanımlanmış dahili üye sınıflar, normal dahili üye sınıflardan farklıdır. Bu farklılıklar şöyledir:

- Statik dahili üye sınıfına ait nesne oluşturmak için, onu çevreleyen sınıfa ait bir nesne oluşmak zorunda değildir.
- Statik dahili üye sınıflar, kendilerini çevreleyen sınıfa ait bağlantıyı (`-this-`) kaybederler .

Statik dahili üye sınıflar, onları çevreleyen üst sınıfa ait global alanlara (statik veya değil) ve yordamlara (statik veya değil) direk ulaşım şansını kaybeder. Bunun sebebi, kendisini çevreleyen sınıf ile arasındaki bağı kopartmış olmasıdır. Buraya kadar ifade edilenleri örnek üzerinde inceleyelim, ama öncesinde UML diyagramı çizilirse...



Şekil-7.8. Statik Dahili Üye Sınıflar

*Hesaplama4* sınıfının içerisinde, 2 adet dahili üye sınıf oluşturulacaktır; fakat bu dahili üye sınıflardan biri statik olarak tanımlanacaktır. Bu örnekte statik tanımlanacak olan dahili üye sınıf, *Toplama4* sınıfıdır. *Toplama4* sınıfına ait bir nesne oluşturulmak istenirse, bunun hemen öncesinde *Hesaplama4* sınıfına ait bir nesne oluşturulmaz. UML diyagramı Java uygulamasına dönüştürülürse...

**Örnek:** *Hesaplama4.java*

```
public class Hesaplama4 {

    int sabit = 2 ;
    private int ozelsabit = 1 ;
    public static class Toplama4 {    // Statik üye dahili sınıf
        static int toplam ;    // dogru
        int sonuc ; //dogru
        public int toplamaYap(int a, int b) {
            // return (a+b) + sabit ; !Hata!
            sonuc = toplam = a+b ;
        }
    }
}
```



```

        return sonuc ;
    }

    public void dekontOlustur() {
        /* -sabit- alanina ve
        -ekranaBas() yordamına ulaşabilmek için
        Hesaplama4 sinifına ait nesne oluşturmamız gerekir.
        */
        Hesaplama4 hs4 = new Hesaplama4();
//dikkat
        int a = hs4.ozelsabit ; //dogru
        hs4.ekranaBas() ; //dogru
        System.out.println("Dekont olusturuyor
= " +
                                hs4.sabit + " -
" +a );
    }
} //class Toplama4

public class Cikartma4 { //Uye dahili sinif

    int sonuc ;
    // static int sonuc1 ; //!hata !
    public int cikartmaYap(int a, int b) {
        ekranBas() ; //dikkat
        sonuc = (a-b) - ozelsabit ;
        return sonuc ; //dikkat
    }
} //class Cikartma4

private void ekranBas() {
    System.out.println("Hesaplama4.ekranaBas()");
}

public static void main(String args[]) {

    //! Hata !
    // Hesaplama4.Toplama4 ht=new Hesaplama4().new
    Toplama4();
    Toplama4 tp4 = new Toplama4();
    tp4.dekontOlustur();
    int sonuc = tp4.toplamaYap(10,5);
    System.out.println("Sonuc = 10 + 5 = " + sonuc
);
}

} //class Hesaplama4

class Hesaplama4Kullan {
    public static void main(String args[]) {

        //! Hata !
        // Hesaplama4.Toplama4 ht=new Hesaplama4().new
        Toplama4() ;
        Hesaplama4.Toplama4 tp4 = new
        Hesaplama4.Toplama4();
        int sonuc = tp4.toplamaYap(10,5);
    }
}

```

```

        System.out.println("Sonuc  =  10 + 5  =  " +
sonuc );
    }

} // class Hesaplama4Kullan

```

Statik dahili üye sınıf olan *Toplama4* sınıfını yakın takibe alıp, neleri nasıl yaptığını inceleyelim. *Toplama4* statik dahili sınıfının içerisinde statik global alan tanımlayabiliriz. Statik olmayan dahili üye sınıfların içerisinde statik global alan tanımlanamaz.

*Toplama4* statik dahili üye sınıfının, *toplamaYap()* yordamının içerisinde, *Hesaplama4* sınıfına ait global olarak tanımlanmış ilkel (primitive) int tipindeki *sabit* alanına direk erişilemez. Statik dahili üye sınıflar ile bunları çevreleyen sınıflar arasında *this* bağlantısı yoktur. Eğer statik dahili üye sınıfın içerisinde, onu çevreleyen sınıfa ait bir alan (statik olmayan) veya yordam (statik olmayan) çağrılmak isteniyorsa, bu bizzat ifade edilmelidir. Aynı *Toplama4* statik dahili üye sınıfına ait *dekontOlustur()* yordamının içerisinde yapıldığı gibi.

*dekontOlustur()* yordamının içerisinde, *Hesaplama4* sınıfına ait nesne oluşturulmadan, *sabit*, *ozelsabit* alanlarına ve *ekranaBas()* yordamına ulaşamazdık. Buradaki önemli nokta, dahili üye sınıf statik olsa bile, kendisine çevreleyen sınıfın *private* erişim belirleyicisi sahip olan alanlarına (statik veya değil) ve yordamlarına (statik veya değil) erişebilmesidir.

*Hesaplama4* sınıfının statik olan *main()* yordamının içerisinde, *Toplama4* statik dahili üye sınıfına ait nesnenin nasıl oluşturulduğuna dikkat edelim. *Toplama4* statik dahili üye sınıfına ait nesne oluştururken, onu çevreleyen sınıfa ait herhangi bir nesne oluşturmak zorunda kalmadık.

Son olarak *Hesaplama4Kullan* sınıfında statik olarak tanımlanan *main()* yordamının içerisindeki olayları inceleyelim. Başka bir sınıfın içerisinde statik dahili üye sınıfı ulaşmak için, sadece tanımlama açısından, dahili üye sınıfı çevreleyen sınıfın ismi kullanılmıştır. Mantıklı olanda budur, statik de olsa sonuçta ulaşmak istenen dahili üye bir sınıftır.

Elimizde iki adet çalıştırılabilir sınıf mevcuttur (-*main()* yordamı olan). *Hesaplama4* sınıfını çalıştırdığımızda (java *Hesaplama4*), sonuç aşağıdaki gibi olur;

```

Hesaplama4.ekranaBas()
Dekont olusturuyor = 2 - 1
Sonuc = 10 + 5 = 15

```

Eğer *Hesaplama4Kullan* sınıfı çalıştırılırsa (java *Hesaplama4Kullan*), sonuç aşağıdaki gibi olur;

```

Sonuc = 10 + 5 = 15

```

#### 7.2.1.4. Statik Dahili Üye Sınıflar ve Statik Yordamlar

Statik dahili üye sınıfların içerisinde statik alanlar bulunduğu gibi, statik yordamlarda bulunabilir. Eğer statik dahili üye sınıfı içerisinde, statik bir yordam oluşturulmuş ise, bu yordamı çağırmak için ne statik dahili üye sınıfına ne de onu çevreleyen sınıfa ait herhangi bir nesne oluşturmak gerekmez.

**Örnek:** *Hesaplama5.java*

```

public class Hesaplama5 {

```

```

private static int x = 3 ;

public static class Toplama5 { //Statik üye dahili sınıf
    static int toplam ; //dogru
    int sonuc ; //dogru
    public static int toplamaYap(int a, int b) {
        // sonuc = a+b + x ; //!Hata !
        toplam = a + b + x ;
        return toplam ;
    }
} // class Toplama5

public static void main(String args[]) {
    int sonuc = Hesaplama5.Toplama5.toplamaYap(16,8);
// dikkat
    System.out.println("Sonuc = 16 + 8 = " +
sonuc );
}
} // class Hesaplama5

```

*Toplama5* statik dahili üye sınıfının, statik olan `toplamaYap()` yordamından, *Hesaplama5* çevreliyiçi sınıfına ait ilkel (primitive) `int` tipinde tanımlanmış `x` alanına ulaşılabilir. Bunun sebebi `x` alanında statik olarak tanımlanmış olmasıdır. `main()` yordamının içerisinde, `toplamaYap()` yordamının çağrılışına dikkat edilirse, ne *Hesaplama5* sınıfına ait nesne, ne de *Toplama5* statik dahili üye sınıfına ait bir nesnenin oluşturulmadığı görülür. Uygulamanın çıktısı aşağıdaki gibidir;

Sonuc = 16 + 8 = 27

#### 7.2.1.5. **Statik ve Final Alanlar**

Statik olmayan dahili üye sınıfların içerisinde, statik alanlar ve yordamlar tanımlanamaz; ama "statik ve final" alanlar tanımlanabilir. Bir alanın hem statik hemde final olması demek, onun SABİT olması anlamına geldiği için, Statik olmayan dahili üye sınıfların içerisinde statik ve final alanlar kullanılabilir.

**Örnek:** *StatikFinal.java*

```

class CevreliyiçiSinif1 {

    class DahiliSinif1 { //Dahili üye sınıflar
        // static int x = 10 ; //!Hata !
    }
}
// Dogru
class CevreliyiçiSinif2 {

    class DahiliSinif2 {
        int x; //Dogru
    }
}
// Dogru
class CevreliyiçiSinif3 {

```

```

class DahiliSinif3 {
    static final int x = 0; //Dogru
}

```

#### 7.2.1.6. Dahili Üye Sınıflar ve Yapılandırıcılar (*Constructors*)

Dahili üye sınıfların yapılandırıcıları olabilir.

**Örnek:** *BuyukA.java*

```

public class BuyukA {

    public class B {
        public B() { //yapilandirici
            System.out.println("Ben B sinifi ");
        }
    } // class B

    public BuyukA() {
        System.out.println("Ben BuyukA sinifi ");
    }

    public static void main(String args[]) {
        BuyukA ba = new BuyukA();
    }
}

```

Dahili üye sınıfını çevreleyen sınıfa ait bir nesne oluşturulduğu zaman, dahili üye sınıfına ait bir nesne otomatik oluşturulmaz. Yukarıdaki örneğimizde sadece *BuyukA* sınıfına ait bir nesne oluşturulmuştur ve bu yüzden sadece *BuyukA* sınıfına ait yapılandırıcı çağrılacaktır. Eğer dahili üye sınıf olan *B* sınıfına ait yapılandırıcının çağrılmasını isteseydik, *main()* yordamının içerisine : " *BuyukA.newB()* " dememiz gerekirdi.

#### 7.2.1.7. İç içe Dahili Üye Sınıflar

Bir sınıfın içerisinde dahili üye sınıf tanımlayabilirsiniz. Tanımlanan bu dahili üye sınıfın içerisinde, yine bir dahili üye sınıf tanımlayabilirsiniz... bu böyle sürüp gidebilir...

**Örnek:** *Abc.java*

```

public class Abc {

    public Abc() { //Yapilandirici
        System.out.println("Abc nesnesi olusturuluyor");
    }

    public class Def {
        public Def() { //Yapilandirici
            System.out.println("Def nesnesi olusturuluyor");
        }
    }
}

```

```

    }

    public class Ghi {
        public Ghi() {          //Yapilandirici
            System.out.println("Ghi nesnesi
olusturuluyor");
        }

    } //class Ghi

} //class Def

public static void main( String args[] ) {
    Abc.Def.Ghi  ici_ice = new Abc().new Def().new
Ghi();
}

} //class Abc

```

Bu örnekte iç içe geçmiş üç adet sınıf vardır. Uygulamanın çıktısı aşağıdaki gibi olur:

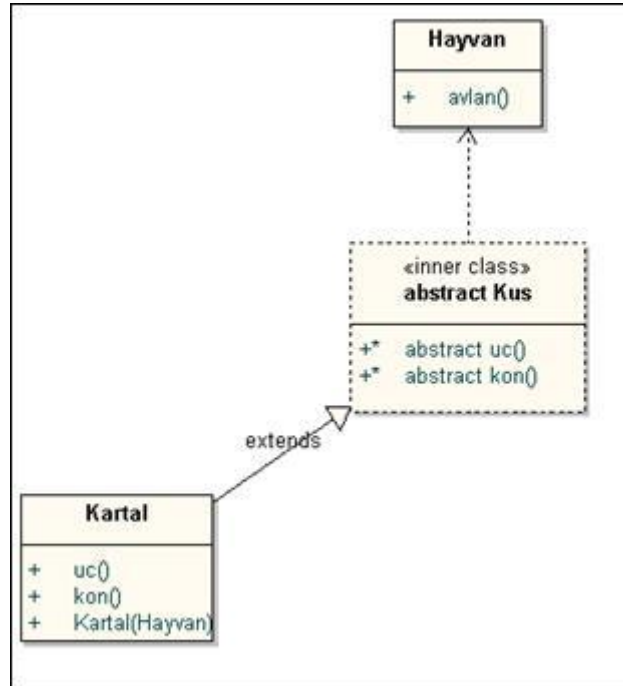
```

Abc nesnesi olusturuluyor
Def nesnesi olusturuluyor
Ghi nesnesi olusturuluyor

```

#### 7.2.1.8. Soyut (Abstract) Dahili Üye Sınıflar

Dahili üye sınıflar, soyut (*abstract*) sınıf olarak tanımlanabilir. Bu soyut dahili üye sınıflardan türeyen sınıflar, soyut dahili üye sınıfların içerisindeki gövdesiz (soyut) yordamları iptal etmeleri gerekmektedir. Örneğimize geçmeden evvel, UML diyagramını inceleyelim.



Şekil-7.9. Soyut Dahili Üye Sınıflar

*Hayvan* sınıfının içerisinde soyut (*abstract*) dahili üye sınıf olarak tanımlanmış *Kus* sınıfı iki adet gövdesiz (soyut-*abstract*) yordamı olsun, *uc()* ve *kon()*. *Kartal* sınıfı, soyut dahili üye sınıf olan *Kus* sınıfından türetilir.

**Örnek:** *HayvanKartal.java*

```
class Hayvan {  
  
    abstract class Kus {  
        public abstract void uc ();  
        public abstract void kon();  
    }  
  
    public void avlan() {  
        System.out.println("Hayvan avlanıyor...");  
    }  
}  
  
class Kartal extends Hayvan.Kus {  
    public void uc() {  
        System.out.println("Kartal Ucuyor...");  
    }  
    public void kon() {  
        System.out.println("Kartal Konuyor...");  
    }  
  
    // public Kartal() { } //!Hata!  
  
    public Kartal(Hayvan hv) {  
        hv.super(); //Dikkat  
    }  
  
    public static void main(String args[]) {  
        Hayvan h = new Hayvan(); //Dikkat  
        Kartal k = new Kartal(h);  
        k.uc();  
        k.kon();  
    }  
}
```

*Kartal* sınıfının içerisinde, soyut dahili üye sınıf olan *Kus* sınıfının, gövdesiz olan iki yordamı iptal edilmiştir. Olayları sırası ile inceleyelim, *Kartal* sınıfına ait bir nesne oluşturulmak istense bunun öncesinde *Kus* sınıfına ait bir nesnenin oluşturulması gerekir çünkü *Kartal* sınıfı *Kus* sınıfından türetilmiştir. Buraya kadar sorun yok, fakat asıl kritik nokta *Kus* sınıfının dahili üye sınıf olmasıdır. Daha açık bir ifade ile, eğer *Kus* sınıfına ait bir nesne oluşturulacaksa, bunun öncesinde elimizde *Kus* sınıfının çevreleyici sınıfı olan *Hayvan* sınıfına ait bir nesne bulunması zorunluluğudur. *Kus* sınıfı statik dahili üye sınıf olmadığından, *Hayvan* sınıfına bağımlıdır. Uygulamanın çıktısı aşağıdaki gibidir;

```
Kartal Ucuyor...  
Kartal Konuyor...
```

*Kartal* sınıfının statik olarak tanımlanmış *main()* yordamının içerisine dikkat edersek, önce *Hayvan* sınıfına ait bir nesne sonrada *Kartal* sınıfına ait bir nesne oluşturduk. Daha sonra *Hayvan* sınıfı tipinde parametre kabul eden, *Kartal* sınıfının yapılandırıcısına, bu referansı pasladık. *Kartal* sınıfına ait

yapılandırıcısının içerisinde `super()` anahtar kelimesi ile, *Hayvan* sınıfının varsayılan yapılandırıcısını çağırılmıştır.

#### **Gösterim-7.11:**

```
CevreliYiciSinif.super();
```

Eğer *Kus* sınıfı, statik dahili üye sınıfı yapılsaydı, `super()` anahtar kelimesini kullanılmak zorunda değildi. Bunun sebebi, statik olan dahili üye sınıfların onları çevreleyen sınıflara bağımlı olmamasıdır. Yukarıdaki örnek bu anlatılanlar ışığında tekrardan yazılırsa.

#### **Örnek:** *HayvanKartal1.java*

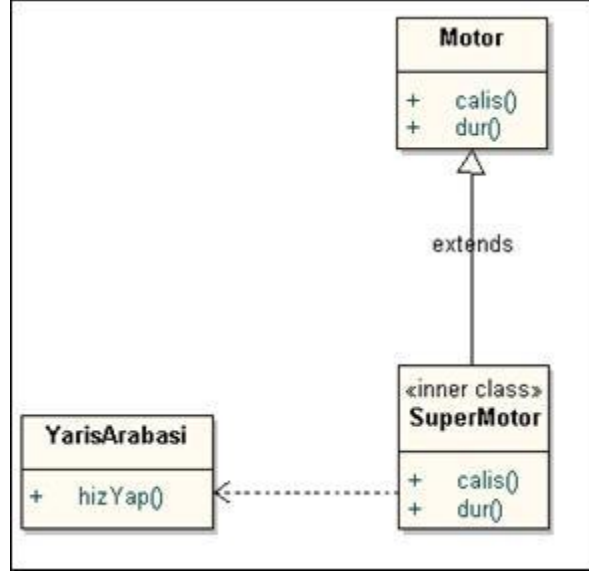
```
class Hayvan1 {  
    static abstract class Kus1 {  
        public abstract void uc ();  
        public abstract void kon();  
    }  
  
    public void avlan() {  
        System.out.println("Hayvan avlanıyor...");  
    }  
}  
  
class Kartal1 extends Hayvan1.Kus1 {  
    public void uc() {  
        System.out.println("Kartal1 Ucuyor...");  
    }  
    public void kon() {  
        System.out.println("Kartal1 Konuyor...");  
    }  
  
    public Kartal1() { } // dogru  
  
    public static void main(String args[]) {  
        Kartal1 k1 = new Kartal1();  
        k1.uc();  
        k1.kon();  
    }  
}
```

Yukarıdaki örneğimizden görüldüğü üzere, artık *Kus* sınıfına ait bir nesne oluşturmak istersek, bunun hemen öncesinde *Hayvan* sınıfına ait bir nesne oluşturmak zorunda değilizdir. Bunun sebebi, *Kus* sınıfının statik dahili üye sınıfı olmasından kaynaklanır. Uygulamanın çıktısı aşağıdaki gibidir;

```
Kartal1 Ucuyor...  
Kartal1 Konuyor...
```

#### **7.2.1.9. Türetilen Dahili Üye Sınıflar**

Dahili üye sınıflar, aynı normal sınıflar gibi başka sınıflardan türetilirler. Böylece diğer dillerde olan çoklu kalıtım desteğinin bir benzerini Java programlama dilinde de bulabiliriz. Dahili sınıfların varoluş sebeplerini biraz sonra detaylı bir şekilde inceleyeceğiz. Örneğimize geçmeden evvel, UML diyagramımızı inceleyelim;



**Şekil-7.10. Türetilen Dahili Üye Sınıflar**

Dahili üye sınıf olan *SuperMotor* sınıfı, *Motor* sınıfından türetilmiştir. UML diyagramını Java uygulamasını dönüştürüp, olayları daha somut bir şekilde incelersek.

**Örnek:** *YarisArabasi.java*

```
class Motor {
    public void calis() {
        System.out.println("Motor Calisiyor");
    }
    public void dur() {
        System.out.println("Motor Durdu");
    }
}

public class YarisArabasi {
    public void hizYap() {
        System.out.println("YarisArabasi hiz yapiyor");
    }
    public class SuperMotor extends Motor {
        public void calis() { // iptal etti (override)
            System.out.println("SuperMotor Calisiyor");
        }
        public void dur() { // iptal etti (override)
            System.out.println("SuperMotor Durdu");
        }
    }
}
```



Dahili üye sınıflar, başka sınıflardan türetilmediği gibi arayüzlere erişip, bunların içlerindeki gövdesiz yordamları iptal edebilir, aynı normal sınıflar gibi...

### 7.2.2. Yerel Sınıflar (*Local Classes*)

Yerel sınıflar, yapılandırıcıların (*constructor*), sınıf yordamlarının (statik yordam), nesne yordamların, statik alanlara toplu değer vermek için kullandığımız statik bloğun (bkz:bölüm 3) veya statik olmayan alanlara toplu değer vermek için kullandığımız bloğun (bkz:bölüm 3) içerisinde tanımlanabilir. Yerel sınıfların genel gösterimi aşağıdaki gibidir;

#### **Gösterim-7.12:**

```
public class Sinif {
    public void yordam() {
        public class YerelSinif {
            //...
        }
    }
}
```

Yerel sınıflar, yalnızca içinde tanımlandıkları, yordamın veya bloğun içerisinde geçerlidir. Nasıl ki dahili üye sınıfların çevreleyici sınıfları vardı, yerel sınıfların ise çevreleyici yordamları veya blokları vardır. Yerel sınıflar tanımlandıkları bu yordamların veya blokların dışarısından erişilemezler.

Yerel sınıflara ait ilk özellikleri verelim;

- Yerel sınıflar tanımlandıkları yordamın veya bloğun dışından erişilemezler.
- Yerel sınıflar başka sınıflardan türetilbilir veya arayüzlere (interface) erişebilir.
- Yerel sınıfların yapılandırıcıları olabilir.

Yukarıdaki özellikleri Java uygulamasında ispatlanırsa;

#### **Örnek:** Hesaplama6.java

```
interface Toplayici {
    public int hesaplamaYap() ;
}

public class Hesaplama6 {

    public int topla(int a, int b) {
        class Toplama6 implements Toplayici {
            private int deger1 ;
            private int deger2;
            public Toplama6(int deger1, int deger2) { //
yapilandirici
                this.deger1 = deger1;
                this.deger2 = deger2;
            }

            public int hesaplamaYap() { //iptal etti (override)
                return deger1+deger2;
            }
        }
    }
}
```

```

    }

    } // class Toplama6

    Toplama6 t6 = new Toplama6(a,b);
    return t6.hesaplamaYap();
}

public void ekranaBas() {
    // Toplama6 t6 = new Toplama6(2,6,); // !Hata!-Kapsama
    alanının dışı
}

public static void main(String args[]) {
    Hesaplama6 h6 = new Hesaplama6();
    int sonuc = h6.topla(5,9);
    System.out.println("Sonuc = 5 + 9 = " + sonuc
);
}
} // class Hesaplama6

```

Bu örneğimizde *Toplama6* yerel sınıftır. Yerel bir sınıf, başka bir sınıftan türetilbilir veya bir arayüze erişip, onun gövdesiz yordamlarını iptal edebilir, aynı normal sınıflar gibi. *Toplama6* yerel sınıfı, *Hesapliyci* arayüzüne eriştiğinden, bu arayüzün gövdesiz yordamı olan *hesaplamaYap()* yordamını iptal etmek zorundadır.

*Toplama6* yerel sınıfı, *Hesaplama6* sınıfının *topla()* yordamının içerisinde tanımlanmıştır. Bunun anlamı, *Toplama6* yerel sınıfına yalnızca *topla()* yordamının içerisinde erişilebileceğidir. *Hesaplama6* sınıfının nesne yordamı olan (bu yordama ulaşmak için *Hesaplama6* sınıfına ait nesne oluşturmamız gerektiği anlamında...) *ekranaBas()* yordamının içerisinde, *Toplama6* yerel sınıfına ulaşılamaz çünkü *Toplama6* yerel sınıfı, *ekranaBas()* yordamının kapsama alanının dışında kalmaktadır. Uygulamamızın çıktısı aşağıdaki gibi olur;

```
Sonuc = 5 + 9 = 14
```

Yerel sınıflara diğer özellikler aşağıdaki gibidir;

- Yerel sınıflar, içinde bulundukları yordamın sadece *final* olan değişkenlerine ulaşabilirler.
- Yerel sınıflar, statik veya statik olmayan yordamların içerisinde tanımlanabilirler.
- Yerel sınıflar, *private*, *protected* ve *public* erişim belirleyicisine sahip olamazlar sadece *friendly* erişim belirleyicisine sahip olabilirler.
- Yerel sınıflar, statik olarak tanımlanamaz.

Yukarıdaki kuralları, bir örnek üzerinde uygularsak...

**Örnek:** *Hesaplama7.java*

```

public class Hesaplama7 {
    public static int topla(int a, final int b) {
        int a_yedek = a ;
        class Toplama7 {
            private int x ; // dogru

```

```

    public int y ; //dogru
    // protected int z = a_yedek ; //!Hata !
    int p ; //dogru
    public int degerDondur() {
        // int degera = a ; //Hata
        int degerb = b ;
        return b ;
    }
} // class Toplama7

    Toplama7 t7 = new Toplama7();
    return t7.degerDondur();
}

public void ekranaBas() {

    /* yerel sınıflar sadece friendly erişim
        belirleyicisine sahip olabilirler

        public class Toplama8 {
            public void test() {}
        } //class Toplama8

    */
} // ekranaBas

public void hesaplamaYap() {

    /* yerel sınıf sadece friendly erişim
        belirleyicisine sahip olabilirler

        static class Toplama9 {
            public void abcd() {
            }
        } // class Toplama9

    */
} // hesaplamaYap

public static void main(String args[]) {

    int sonuc = Hesaplama7.topla(5,9);
    System.out.println("Sonuc " + sonuc );
}
} // class Hesaplama7

```

*Toplama7* yerel sınıfı, *Hesaplama7* sınıfının, statik olan `topla()` yordamının içerisinde tanımlanmıştır ve sadece `topla()` yordamının içerisinde geçerlidir. *Toplama7* yerel sınıfı, `topla()` yordamının içerisindeki `final` özelliğine sahip olan yerel değişkenlere erişip onları kullanabilir. Bu sebepten dolayı, ilkel (primitive) `int` tipinde tanımlanmış olan `a` ve `a_yedek` yerel değişkenlerine *Toplama7* yerel sınıfının içerisinden erişilemez, bu bir hatadır.

*Hesaplama7* sınıfının, nesne yordamı olan `ekranaBas()` içerisinde tanımlanmış olan *Toplama8* yerel sınıfı hatalıdır. Hatanın sebebi *Toplama8* yerel sınıfının `public` erişim belirleyicisine sahip olmasıdır. Yukarıda belirtildiği üzere, yerel sınıflar ancak `friendly` erişim belirleyicisine sahip olabilir.

Aynı şekilde *Hesaplama7* sınıfının, nesne yordamı olan `hesaplamaYap()` içerisinde tanımlanmış olan *Toplama9* yerel sınıfı hatalıdır. Bu hatanın sebebi, *Toplama9* yerel sınıfının statik yapılmaya çalışılmasıdır. Az evvel belirtildiği gibi, yerel yordamlar, statik olarak tanımlanamazlardı. Uygulamanın çıktısı aşağıdaki gibidir;

Sonuc 9

### 7.2.3. İsimsiz Sınıflar (*Anonymous Classes*)

İsimsiz sınıflar, isimsiz ifade edilebilen sınıflardır. İsimsiz sınıflar havada oluşturulabildiklerinden dolayı bir çok işlem için çok avantajlıdır, özellikle olay dinleyicilerin (*event listeners*) devreye sokulduğu uygulamalarda sıkça kullanılırlar. İsimsiz sınıfların özellikleri aşağıdaki gibidir;

- Diğer dahili sınıf çeşitlerinde olduğu gibi, isimsiz sınıflar direk `extends` ve `implements` anahtar kelimelerini kullanarak, diğer sınıflardan türetilemez ve arayüzlere erişemez.
- İsimsiz sınıfların herhangi bir ismi olmadığı için, yapılandırıcısında (*constructor*) olamaz.

Yukarıdaki kuralları, bir örnek üzerinde uygularsak...

#### Örnek: *Hesaplama8.java*

```
interface Toplayici {
    public int hesaplamaYap() ;
}

public class Hesaplama8 {

    public Toplayici topla(final int a, final int b)
    {
        return new Toplayici() {
            public int hesaplamaYap() {

                // final olan yerel degiskenlere ulasabilir.
                return a + b ;

            }
        }; // noktali virgul sart

    } // topla, yordam sonu

    public static void main(String args[]) {

        Hesaplama8 h8 = new Hesaplama8();
        Toplayici t = h8.topla(5,9);
        int sonuc = t.hesaplamaYap();
        System.out.println("Sonuc = 5 + 9 = "
+ sonuc );
    }
}
```

```
} // class Hesaplama8
```

*Hesaplama8* sınıfının, `topla()` yordamı *Toplayici* arayüzü tipindeki nesneye bağlı bir referans geri döndürmektedir. *Toplayici* arayüzü tipindeki nesneye bağlı bir referans geri döndürmek demek, *Toplayici* arayüzüne erişip onun gövdesiz olan yordamlarını iptal eden bir sınıf tipinde nesne oluşturmak demektir. Sonuçta bir arayüze ulaşan sınıf, ulaştığı arayüz tipinde olan bir referansa bağlanabilirdi. " Buraya kadar tamam ama isimsiz sınıfımız nerede... diyebilirsiniz. Olaylara daha yakından bakılırsa;

#### **Gösterim-7.13:**

```
return new Toplayici() {  
    public int hesaplamaYap() {  
  
        // final olan yerel degiskenlere ulasabilir.  
        return a + b ;  
    }  
}; // noktali virgul sart
```

İşte isimsiz sınıfımız !! .Yukarıdaki ifade yerine, `topla()` yordamın içerisinde yerel bir sınıf da yazılabilirdi.

#### **Gösterim-7.14:**

```
public Toplayici topla(final int a, final int b) {  
    public class BenimToplayicim implements Toplayici {  
        public int hesaplamaYap() {  
  
            // final olan yerel degiskenlere ulasabilir.  
            return a + b ;  
        }  
    } // yordam sonu  
    return new BenimToplayicim();  
}
```

İsimsiz sınıfları, yerel sınıfların kısaltılmışı gibi düşünebilirsiniz. Yerel sınıflarda `return new BenimToplayicim()` yerine, isimsiz sınıflarda hangi sınıf tipinde değer döndürüleceği en başta belirtilir.

#### **Gösterim-7.15:**

```
return new Toplayici() { ....  
    ...  
};
```

İsimsiz sınıflarda, yerel sınıflar gibi içinde bulundukları yordamın sadece `final` olarak tanımlanmış yerel değişkenlerine erişebilirler.

Yukarıdaki örneğimizde, isimsiz sınıfımız, *Toplayici* arayüzüne erişip onun gövdesiz sınıflarını iptal etmiştir, buraya kadar herşey normal. Peki eğer isimsiz sınıfımız, yapılandırıcısı parametre olan bir sınıftan türetilseydi nasıl olacaktı? Belirtildiği üzere isimsiz sınıfların yapılandırıcısı olamaz.

### **Örnek:** *Hesaplama9.java*

```
abstract class BuyukToplayici {
    private int deger ;
    public BuyukToplayici(int x) {
        deger = x ;
    }
    public int degerDondur() {
        return deger ;
    }
    public abstract int hesaplamaYap() ; // iptal edilmesi gerek
}

public class Hesaplama9 {
    public BuyukToplayici degerGoster( int gonderilen )
    {
        return new BuyukToplayici( gonderilen ) {
            public int hesaplamaYap() { //iptal etti (override)
                return super.degerDondur() + 5 ;
            }
        }; // noktali virgul sart
    } // degerGoster , yordam sonu

    public static void main(String args[]) {

        Hesaplama9 h9 = new Hesaplama9();
        BuyukToplayici bt = h9.degerGoster(5);
        int sonuc = bt.hesaplamaYap();
        System.out.println("Sonuc = " + sonuc );
    }
} // class Hesaplama9
```

*BuyukToplayici* sınıfı soyuttur, bunun anlamı bu sınıfın içerisinde en az bir tane gövdesiz yordam olduğudur. *BuyukToplayici* sınıfının içerisindeki `hesaplamaYap()` gövdesiz yordamını, *BuyukToplayici* sınıfından türetilen alt sınıflar tarafından iptal edilmek zorundadır.

Bu örneğimizde ilginç olan iki nokta vardır. Birincisi, isimsiz bir sınıfın, soyut bir yordam dan türetilmesi, ikincisi ise türetilme yapılan *BuyukToplayici* sınıfına ait yapılandırıcısının parametre almasıdır. İsimsiz sınıfımızın yapılandırıcısı olamayacağından dolayı, *BuyukToplayici* sınıfına ait parametre alan yapılandırıcıyı burada çağıramayız. Bu işlemi *BuyukToplayici* sınıfından türetilen isimsiz sınıfımızı oluştururken yapmalıyız.

İsimsiz sınıfların içerisinde, onları çevreleyen yordamların final olmayan yerel değişkenleri kullanılamaz. “Peki ama bu örnekte kullanılıyor...” diyebilirsiniz.

### **Gösterim-7.16:**

```
public BuyukToplayici degerGoster( int gonderilen ) {  
    return new BuyukToplayici( gonderilen ) {  
        public int hesaplamaYap() { //iptal etti (override)  
            return super.degerDondur() + 5 ;  
        }  
    }; // noktali virgul sart  
  
} // degerGoster , yordam sonu
```

İlkel int tipinde tanımlanmış gonderilen yerel değişkeni, degerGoster() yordamına aittir, isimsiz sınıfımızın içerisinde kullanılmamıştır. return new BuyukToplayici(gonderilen) ifadesi, degerGoster() yordamına dahil olduğundan bir sorun çıkmaz. Eğer uygulamamızı çalıştırsak, ekran çıktısı aşağıdaki gibi olacaktır.

```
Sonuc = 10
```

### **7.2.4. Fiziksel İfade**

İçerisinde Java kodları olan fiziksel bir dosya derlendiği (*compile*) zaman, bu fiziksel dosya içerisinde tanımlanmış her bir sınıf için, fiziksel bir *.class* dosyası oluşturulur. Peki olaylar dahili sınıflar içinde aynı mıdır? Her bir dahili sınıf için, bir fiziksel *.class* dosyası oluşturulur mu? Eğer oluşturuluyorsa ismi ne olur?

Her bir dahili sınıf için (3 çeşit dahili sınıf içinde geçerli) fiziksel *.class* dosyası oluşturulur. Bu *.class* dosyasının ismi ise, çevreleyen sınıfın ismi + \$ + dahili sınıfın ismi şeklindedir. *Hesaplama1.java* örneğimizden bir gösterim yaparsak;

### **Gösterim-7.17:**

```
Hesaplama1$1.class  
Hesaplama1$Bolme.class  
Hesaplama1$Carpma.class  
Hesaplama1$Cikartma.class  
Hesaplama1$Toplama.class  
Hesaplama1.class
```

*Hesaplama1* sınıfı, *Bolme*, *Carpma*, *Cikartma* ve *Toplama* sınıflarının çevreliyiçi sınıfıdır, böyle olunca dahili sınıflarımıza ait *.class* dosyasının ismi de *ÇevreliyiçiSınıf\$DahiliSınıf* biçiminde olduğunu görürüz.

Java, *Hesaplama9.java* örneğimizdeki isimsiz sınıf için nasıl bir *.class* dosyası oluşturur? Cevabı hemen aşağıdadır;

### **Gösterim-7.18.**

```
Hesaplama9$1.class  
Hesaplama9.class
```

Java, *Hesaplama9.java* içerisinde belirtilmiş isimsiz sınıfa ait *.class* dosyası oluştururken isim olarak *1 (bir)* kullanmıştır. Eğer aynı çevreliliyi sınıf içerisinde iki adet isimsiz sınıf olsaydı, bu isimsiz sınıfların ismi 1 ve 2 olacaktı.

#### **Gösterim-7.19:**

```
Hesaplama9$1.class  
Hesaplama9$2.class  
Hesaplama9.class
```

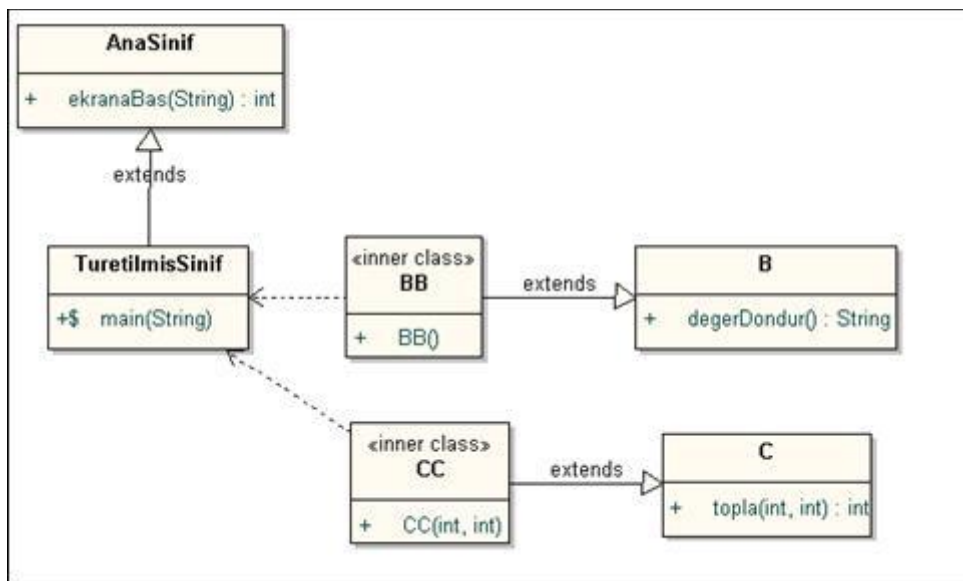
#### **7.2.5. Neden Dahili sınıflar?**

Dahili üye sınıflar, yerel sınıflar, isimsiz sınıflar hepsi çok güzel ama Java programlama dili neden bunlara ihtiyaç duymuş olabilir? Şimdiye kadar normal sınıflarımızla güzel güzel idare edebiliyorduk diyebilirsiniz. Dahili sınıfların var olmasındaki neden çoklu kalıtım (*multiple inheritance*) tam desteği sağlamaktır.

Arayüzler ile çoklu kalıtım desteğini kısmen bulabiliyorduk ama bu tam değildi. Tam değildi çünkü bir sınıf iki normal sınıftan türetiliyordu, bunun sakıncalarını tartışmıştık. Fakat bazı zamanlarda, arayüzler dışında, normal sınıflara ihtiyaç duyabiliriz. Normal sınıflar derken, soyut olmayan, problem çözmek için tasarlanmış işleyen sınıflardan bahsediyorum. Bu işleyen sınıfların iki tanesine aynı anda ulaşım türetme yapılmıyordu ama bu isteğimize artık dahili sınıflar ile ulaşabiliriz.

Java, dahili sınıflar ile çoklu kalıtım olan desteğini güvenli bir şekilde sağlamaktadır. Dahili sınıflar, kendilerini çevreleyen sınıfların hangi sınıftan türetildiğine bakmaksızın bağımsız şekilde ayrı sınıflardan türetilir veya başka arayüzlere erişebilir.

Örnek Java kodumuzu incelemeden evvel, UML diyagrama bir göz atalım.



**Şekil-7.11. Dahili sınıflara neden ihtiyaç duyarız ?**



UML diyagramından olayları kuş bakışı görebiliyoruz. *AnaSınıf* sınıfından türetilmiş *TuretilmisSınıf* sınıfının içerisinde iki adet dahili üye sınıf bulunmaktadır. Dahili üye sınıf olan *BB* ve *CC* sınıfları da, *B* ve *C* sınıflarından türetilmişlerdir. Bu örneğimizdeki ana fikir, bir sınıfın içerisinde dahili üye sınıflar kullanılarak çoklu kalıtımın güvenli bir şekilde yapılabildiğini göstermektir. UML diyagramını Java uygulamasına dönüştürsek;

**Örnek:** *TuretilmisSınıf.java*

```
class AnaSınıf {
    public void ekranaBas(String deger) {
        System.out.println( deger );
    }
}

class B {
    public String degerDondur() {
        return "B";
    }
}

class C {
    public int  topla(int a , int b) {
        return a+b ;
    }
}

public class TuretilmisSınıf  extends AnaSınıf {

    public class BB extends B {
        public BB() { //yapılandırıcı
            ekranaBas( "Sonuc = " + degerDondur() );
        }
    }

    public class CC extends C {
        public CC( int a , int b ) { //yapılandırıcı
            ekranaBas("Sonuc = " + topla(a,b) );
        }
    }

    public static void main( String args[] ) {
        TuretilmisSınıf.BB tbb= new TuretilmisSınıf().new
        BB();
        TuretilmisSınıf.CC tcc= new TuretilmisSınıf().new
        CC(6, 9);
    }
}
```

*TuretilmisSınıf* sınıfımız, *AnaSınıf* sınıfından türetilmiştir fakat bu dahili sınıfların başka sınıflardan türetilmelerine engel teşkil etmez. Her bir dahili sınıfın kendine ait bir durumu olabilir. Dahili sınıflar kendilerini çevreleyen sınıflardan bağımsızdır. Dahili sınıflar ile onları çevreleyen sınıflar arasında

kalıtsal bir ilişki olmak zorunda değildir, geçen bölümlerde incelediğimiz "bir" ilişkisi, *Kaplan bir Kedidir* gibi.

Örneğimize geri dönersek, *B* sınıfından türetilmiş *BB* sınıfı ve *C* sınıfından türetilmiş *CC* sınıfı, *Anasınıf* sınıfına ait *ekranaBas()* yordamını kullanarak sonuçlarını ekrana yansıtabilmektedirler. Olaylara bu açıdan bakacak olursa, *TüretilmisSınıf* sınıfın sanki üç ayrı işleyen (normal) sınıftan güvenli ve kolay bir şekilde türetilmiş olduğu görülür.

Uygulamamızın çıktısı aşağıdaki gibi olur;

```
Sonuc = B
Sonuc = 15
```

**Bu dökümanın her hakkı saklıdır.**

© 2004

**BÖLÜM**  
**8**



**Altuğ B.**  
**Altıntaş**  
**© 2004**

## ***Istisnalar (Exception)***

*“Diğerlerinin yazdığı programda hata olabilir ama benim yazdığım programda hata olmaz....” - Anonim*

Bu bölümde istisnalar üzerinde durulacaktır. İstisna deyince aklınıza ne geliyor? Yanlış yazılmış uygulama mı? Beklenmeyen durum mu? Yoksa her ikisi de mi? İstisna demek işlerin sizin kontrolünüzden çıkması anlamına gelir. Yani kaos ortamı, önceden kestirilemeyen... Birşeylerin ters gitmesi sonucu uygulamanın normal akışına devam edememesi demektir. Bu ters giden bir şeyler ne olabilir? Örneğin kullanıcının uygulamanıza istemeyen veri girmesi olabilir veya açmak istediğiniz dosyanın yerinde olmaması olabilir, örnekleri çoğaltmak mümkündür.

### **8.1. İstisnalara Giriş**

Gerçekten tam bir uygulama yazmak ne demektir? Uygulamadan beklenen görevleri yerine getirmesi onu tam bir uygulama yapar mı? Tabii ki yapmaz. Uygulama zaten kendisinden beklenen işi yapmalı, aksi takdirde zaten uygulama olmaz. Bir uygulamanın tam olmasının iki şartı vardır; Birincisi uygulamanın kendisinden beklenen görevleri doğru bir şekilde yerine getirmesidir yani doğruluk, ikincisi ise hatalı davranışlara karşı dayanıklı olmasıdır, sağlamlık. Örneğin bizden iki sayıyı bölmek için bir uygulama istense ne yapılmalıdır, *A/ B - A* bölüm *B* çok basit değil mi?. İlk etapta karşı tarafın bizden istediği şey, girilen iki sayının doğru şekilde bölünmesidir - doğruluk, bu öncelikli şarttır, bunda herkes hemfikir. Peki ikinci şart nedir? İkinci şart ise sağlamlıktır, ikinci şart olan sağlamlık genellikle önemsenmez. Bu örneğimizde karşı tarafın bizden istediği olay, iki sayının bölünmesidir ama dikkat edin sayı dedim, kullanıcı *int*, *double* veya *short* ilkel tiplerinde sayı girilebilir. Peki ya kullanıcı *String* bir ifadeyi uygulamanıza yollarsa ne olur? veya *A=5, B=0* girince uygulamanız buna nasıl bir tepki verir? (Not :5/0=sonsuz) Uygulamanız direk olarak kapanır mı? Veya uygulamanız bu anlamsız ifadeleri bölmeye mi çalışır? Eğer siz uygulamayı tasarlayan kişi olarak, bu hataları önceden tahmin etmiş ve önlemleri

almışsanız sorun ortaya çıksa bile, uygulama için sorun olmaz ama gerçek dünyada her şeyi öngörebilmek imkansızdır.

Java programlama dili, oluşabilecek hatalara karşı sert bir yaptırım uygular. Dikkat edin, oluşabilecek diyorum. Java programlama dili, ortada hata oluşmasına sebebiyet verebilecek bir durum var ise yazılan Java dosyasını derlemeyerek (*compile*) kodu yazan kişiye gerekli sert tavrı gösterir. Java programlama dilinin bu tavrı doğru mudur? Kimileriniz diyebilir ki, "Java sadece üstüne düşen görevi yapsın, oluşabilecek hataları bana söyleyerek canımı sıkmasın". Bu yaklaşım yanlıştır, Java programlama dilinin amacı kodu yazan kişiye maksimum şekilde yardımcı olmaktır, daha doğrusu insana dayalı oluşabilecek hataları kendi üstüne alıp, hatalı uygulama üretimini minimuma indirmeyi amaçlayarak tasarlanmıştır. Bunun ilk örneğini çöp toplama (*garbage collector*) mekanizmasında görmüştük. Diğer dillerde oluşturulan nesnelerin, daha sonradan işleri bitince bellekten silinmemelerinden dolayı bellek yetmezlikleri oluşmaktadır. " Kodu yazan insan, oluşturduğu nesneyi bellekten temizlemez mi? Ben bunu şahsen hiç yapmam. O zaman dalgın insanlar kod yazmasın aaa! " diye bir söz sakın demeyin, çünkü insanoğlu yeri geldiğinde çok dalgın olabilir ve bu dalgınlık uygulamayı bir bellek canavarına dönüştürebilir ayrıca bu tür hataları, uygulamanın içerisinde ayıklamak cidden çok zor bir iştir. Bu yüzden Java programlama dilinde, bir nesnenin bellekten silinmesi kodu yazan kişiye göre değil, çöp toplama algoritmalarına göre yapılır (bkz:Bölüm3). Java'nın oluşabilecek olan hatalara karşı bu sert tutumu da gayet mantıklıdır. Bu sert tutum sayesinde ileride oluşabilecek ve bulunması çok güç olan hataların erkenden engellenmesini sağlar.

### 8.1.1. İstisna Nasıl Oluşabilir?

İstisna oluşumuna en basit örnek olarak, yanlış kullanılmış dizi uygulamasını verebiliriz. Java programlama dilinde dizilere erişim her zaman kontrollüdür. Bunun anlamı, Java programlama dilinde dizilerin içerisine bir eleman atmak istiyorsak veya var olan bir elemana ulaşmak istiyorsak, bu işlemlerin hepsi Java tarafından önce bir kontrolden geçirilir. Bunun bir avantajı, bir de dezavantajı vardır. Avantaj olarak güvenli bir dizi erişim mekanizmasına sahip oluruz, dezavantaj olarak ufakta olsa hız kaybı meydana gelir. Fakat böyle bir durumda hız mı daha önemlidir yoksa güvenlik mi? Bu sorunun cevabı Java programlama dili için güvenlidir. Aşağıdaki örneğe dikkat edelim;

**Örnek:** *DiziErisim.java*

```
public class DiziErisim {  
  
    public static void main(String args[]) {  
  
        int sayilar[] = {1, 2, 3, 4};  
        System.out.println("Basla");  
        for (int i=0 ; i < 5 ; i++) {  
            System.out.println("--> " + sayilar[i]);  
        }  
        System.out.println("Bitti");  
    }  
}
```

`sayilar[]`, ilkel (*primitive*) `int` tipinde dizi değişkenidir ve bağlı bulunduğu dizi nesnesinin içerisinde 4 adet `int` tipinde eleman vardır. `for` döngüsü sayesinde dizi içerisindeki elemanları ekrana bastırmaktayız. Bu örneğimizdeki hata, `for` döngüsünün fazla dönmesiyle dizinin olmayan elemanına ulaşmak istememizden kaynaklanmaktadır. Böyle bir hareket, çalışma-anında (*run-time*) hata oluşmasına sebebiyet verip uygulamamızın aniden sonlanmasına sebebiyet verecektir. Uygulamayı çalıştırıp, sonuçları hep beraber görelim.

Basla

```
--> 1
--> 2
--> 3
--> 4
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException
    at DiziErisim.main(DiziErisim.java:10)
```

Bu örneğimizdeki istisna, *ArrayIndexOutOfBoundsException* istisnasıdır. Bu istisnanın sebebi, bir dizinin olmayan elemanına erişmeye çalıştığımızı ifade eder. Fark edildiği üzere Java programlama dilinde, oluşan istisnaları anlamak ve yerlerini belirlemek çok zor değildir. Örneğin bu uygulamada istisnanın 10. satırda ortaya çıktığı anlaşılabilmektedir.

### 8.1.2. Başka İstisnalar Neler Olabilir?

Bir uygulama içerisinde, başka ne tür istisnalar oluşabilir ? Bir kaç örnek verirsek;

- Açmak istediğiniz fiziksel dosya yerinde olmayabilir.
- Uygulamanıza kullanıcılar tarafında, beklenmedik bir girdi kümesi gelebilir.
- Ağ bağlantısı kopmuş olabilir.
- Yazmak istediğiniz dosya, başkası tarafından açılmış olduğundan yazma hakkınız olmayabilir.

Olabilir, olmayabilir, belki... Yukarıdaki istisnaların, bir uygulamanın başına gelmeyeceğini kim garanti edebilir? Kimse, peki Java program içerisinde tam bir uygulama nasıl yazılır. Başlayalım...

### 8.1.3. İstisna Yakalama Mekanizması

Bir istisna oluştuğu zaman uygulamamız aniden kapanmak zorunda mı? Oluşan bu istisnayı daha şık bir şekilde yakalayıp uygulamanın devam etmesini sağlamak mümkün mü? Cevap olarak evet;

#### Gösterim-8.1:

```
try {
    // İstisna sebebiyet verebilecek olan kod
} catch (Exception1 e1) {
    //Eğer Exception1 tipinde istisna fırlatılırsa buraya
} catch (Exception2 e2) {
    //Eğer Exception2 tipinde istisna fırlatılırsa buraya
}
```

İstisnaya sebebiyet verebilecek olan kod, `try` bloğunun içerisinde tutularak güvenlik altına alınmış olur. Eğer istisna oluşursa, istisna yakalama mekanizması devreye girer ve oluşan bu istisnanın tipine göre, uygulamanın akışı `catch` bloklarından birinin içerisine yönlendirilerek devam eder.

İstisnalar nesnedir. Bir istisna oluştuğu zaman bir çok olay gerçekleşir. İlk önce yeni bir istisna nesnesi belleğin heap alanında `new()` anahtar kelimesi ile oluşturulur. Oluşan bu istisna nesnesinin içerisine hatanın oluştuğu satır yerleştirilir. Uygulamanın normal seyri durur ve oluşan bu istisnanın yakalanması için `catch` bloğunun olup olmadığına bakılır. Eğer `catch` bloğu varsa uygulamanın akışı uygun `catch` bloğunun içerisinden devam eder. Eğer `catch` bloğu tanımlanmamış ise hatanın oluştuğu yordamı (*method*) çağırarak yordama istisna nesnesi paslanır, eğer bu yordam içerisinde de istisnayı yakalamak için

catch bloğu tanımlanmamış ise istisna nesnesi bir üst yordama paslanır, bu olay böyle devam eder ve en sonunda main() yordamına ulaşan istisna nesnesi için bir catch bloğu aranır eğer bu yordamın içerisinde de catch bloğu tanımlanmamış ise, uygulananın akışı sonlanır. Bu olayları detaylı incelemeden evvel temel bir giriş yapalım;

**Örnek:** *DiziErisim2.java*

```
public class DiziErisim2 {

    public void calis() {

        int sayilar[] = {1,2,3,4};
        for (int i=0 ; i < 5 ; i++) {
            try {
                System.out.println("--> " + sayilar[i]);
            } catch (ArrayIndexOutOfBoundsException ex) {
                System.out.println("Hata Olustu " + ex);
            }
        } // for
    }

    public static void main(String args[]) {

        System.out.println("Basla");
        DiziErisim2 de2 = new DiziErisim2();
        de2.calis();
        System.out.println("Bitti");
    }
}
```

Yukarıdaki uygulamamızda, dizi elemanlarına erişen kodu try bloğu içerisine alarak, oluşabilecek olan istisnaları yakalama şansına sahip olduk. Sahip olduk da ne oldu diyenler için gereken açıklamayı hemen yapalım. try-catch istisna yakalama mekanizması sayesinde istisna oluşsa bile uygulamanın akışı aniden sonlanmayacaktır. *DiziErisim.java* ile *DiziErisim2.java* uygulamalarının çıktısına bakılırsa aradaki kontrolü hemen fark edilecektir. *DiziErisim2.java* uygulama örneğimizin çıktısı aşağıdaki gibidir.

```
Basla
--> 1
--> 2
--> 3
--> 4
Hata Olustu java.lang.ArrayIndexOutOfBoundsException
Bitti
```

Kontrol nerede? Yukarıdaki *DiziErisim2.java* uygulamasının çıktısının son satırına dikkat ederseniz, "Bitti" yazısının ekrana yazıldığını görürsünüz oysaki bu ifade *DiziErisim.java* uygulamasının çıktısında görememiştik. İşte kontrol buradadır. Birinci kuralı daha net bir şekilde ifade edersek; try-catch istisna yakalama mekanizması sayesinde, istisna oluşsa bile uygulamanın akışı aniden sonlanmaz.

Yukarıdaki örneğimizde, try-catch mekanizmasını for döngüsünün içerisine koyulabileceği gibi, for döngüsünü kapsayacak şekilde de tasarlanıp yerleştirilebilir.

**Örnek:** *DiziErisim3.java*

```
public class DiziErisim3 {

    public void calis() {
        try {
            int sayilar[] = {1,2,3,4};
            for (int i=0 ; i < 5 ; i++) {
                System.out.println("--> " + sayilar[i]);
            }
        } catch (ArrayIndexOutOfBoundsException ex) {
            System.out.println("Hata Yakalandi");
        }
    }

    public static void main(String args[]) {

        System.out.println("Basla");
        DiziErisim3 de3 = new DiziErisim3();
        de3.calis();
        System.out.println("Bitti");
    }
}
```

Bu uygulama örneği ile *DiziErisim2.java* örneğimiz arasında sonuç bakımından bir fark yoktur. Değişen sadece tasarımıdır, try-catch bloğunun daha fazla kodu kapsamasıdır.

#### 8.1.4. İstisna İfadeleri

Bir yordam hangi tür istisna fırlatabileceğini önceden belirtebilir veya belirtmek zorunda kalabilir. Bu yordamı (*method*) çağıran diğer yordamlar da, fırlatılabilecek olan bu istisnayı, ya yakalarlar ya da bir üst bölüme iletirler. Bir üst bölümden kasıt edilen, bir yordamı çağıran diğer bir yordamdır. Şimdi bir yordamın önceden hangi tür istisna fırlatacağını nasıl belirtmek zorunda kaldığını inceleyelim.

**Örnek:** *IstisnaOrnek1.java*

```
import java.io.*;

public class IstisnaOrnek1 {

    public void cokCalis() {
        File f = new File("ornek.txt");
        BufferedReader bf = new BufferedReader( new
        FileReader( f ) );
        System.out.println(bf.readLine());
    }

    public void calis() {
        cokCalis();
    }

    public static void main(String args[]) {
        IstisnaOrnek1 io1 = new IstisnaOrnek1();
        io1.calis();
    }
}
```

```
}  
}
```

*java.io* paketinin içerisindeki sınıfları henüz incelemedik ama bu örneğimizde kullanılan sınıfların ne iş yaptıklarını anlamak çok zor değil. Burada yapılan iş, aynı dizinde bulunduğu farz edilen *ornek.txt* dosyasının ilk satırını okumaya çalışmaktır. Yukarıdaki uygulamamızı derlemeye çalışırsak, derleyicinin bize vereceği mesaj aşağıdaki gibi olur.

```
IstisnaOrnek1.java:9: unreported exception  
java.io.FileNotFoundException;  
must be caught or declared to be thrown new FileReader(f));  
^  
IstisnaOrnek1.java:10: unreported exception java.io.IOException;  
must be caught or declared to be thrown  
System.out.println(bf.readLine());  
^  
2 errors
```

Biz diskimizde bulunduğu varsayılan bir dosyaya erişip onun ilk satırını okumaya çalışmaktayız. Çok masum gibi gözüken ama tehlikeli istekler. Peki daha detaylı düşünelim ve oluşabilecek olan istisnaları tahmin etmeye çalışalım.

İlk oluşabilecek olan istisna, o dosyanın yerinde olmayabileceğidir. Bu beklenmeyen bir durum oluşturabilir, başka neler olabilir? Bundan ayrı olarak biz sanki o dosyanın orada olduğundan eminmişiz gibi birde onun ilk satırını okumaya çalışıyoruz, bu isteğimizde istisnaya sebebiyet verebilir çünkü dosya yerinde olsa bile dosyanın ilk satırı olmayabilir. Dikkat ederseniz hep olasılıklar üzerinde durmaktayım ama güçlü olasılıklar. Peki bu uygulamayı derlemenin bir yolu yok mu?

Az önce bahsedildiği gibi bir yordam içerisinde oluşmuş olan istisnayı bir üst bölüme yani o yordamı çağıran yordama fırlatabilir. Eğer bir istisna oluşursa bu anlattıklarımıza göre bir yordamın iki şansı vardır diyebiliriz. Birincisi oluşan bu istisnayı ya yakalayıp gereken işlemleri kendi içerisinde sessizce gerçekleştirebilir veya bu istisna ile ben ne yapacağımı bilmiyorum beni çağıran yordam düşünsün diyip, istisna nesnesini bir üst bölüme fırlatabilir.

Aşağıdaki örnekte, oluşan istisnayı aynı yordamın içerisinde yakalanmaktadır; bu yüzden yordamın hangi istisnayı fırlatabileceğini açıklamasına gerek yoktur. Bir yordamın hangi tür istisnayı nasıl fırlatabileceğini açıklama olayını az sonra göreceğiz ama önce aşağıdaki örneğimizi inceleyelim.

### **Örnek:** *IstisnaOrnek2.java*

```
import java.io.*;  
  
public class IstisnaOrnek2 {  
  
    public void cokCalis() {  
        try {  
            File f = new File("ornek.txt");  
            BufferedReader bf=new BufferedReader(new  
FileReader(f) );  
            System.out.println(bf.readLine());  
        } catch (IOException ex) {  
            System.out.println("Hata Yakalandi =" + ex);  
        }  
    }  
}
```

```
    }

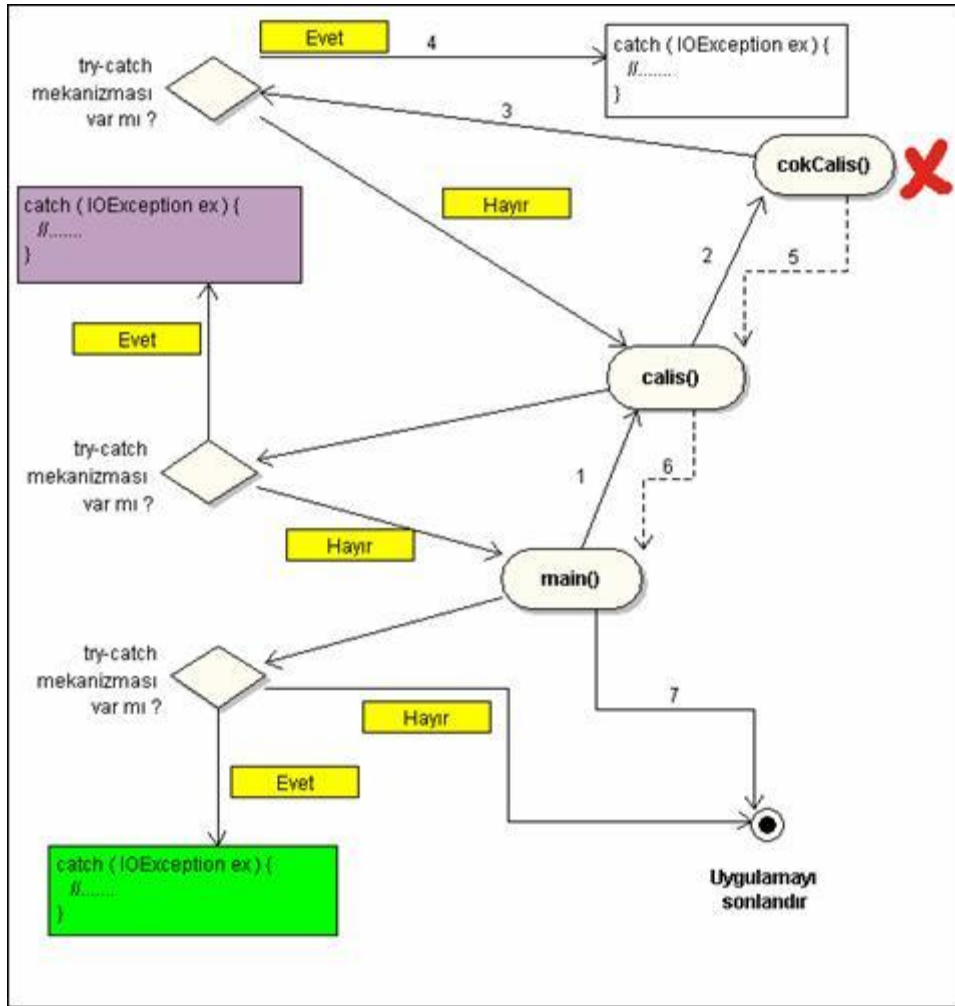
    public void calis() {
        cokCalis();
        System.out.println("calis() yordamı");
    }

    public static void main(String args[]) {
        IstisnaOrnek2 io2 = new IstisnaOrnek2();
        io2.calis();
        System.out.println("main() yordamı");
    }
}
```

Verilen örnekte, dosyaya erişirken veya ondan birşeyler okumak isterken oluşabilecek olan istisnalar; *java.io.IOException* istisna tipini kullanarak yakalanabilir. Zaten *IstisnaOrnek1.java* uygulamasının derlemeye çalışırken alınan hatadan hangi tür istisna tipinin kullanılması gerektiğini de çıkartabiliriz. *java.io.FileNotFoundException* istisna tipini, *java.io.IOException* tipi kullanılarak yakalanabilir bunun nasıl olduğunu biraz sonra göreceğiz.

Yukarıdaki uygulama güzel bir şekilde derlenir çünkü oluşabilecek olan tüm istisnalar için tedbir alınmıştır. Olayların akışını inceliyelim, bu uygulamayı çalıştırdığımız zaman (java IstisnaOrnek2) ilk olarak main() yordamından akışa başlanır. Daha sonra calis() yordamının ve cokCalis() yordamının çağırılması şeklinde akış devam ederken olanlar olur ve cokCalis() yordamının içerisinde istisna oluşur. Çünkü *ornek.txt* diye bir dosya ortalarda yoktur (yok olduğunu varsayın) ama olsun içimiz rahat çünkü try-catch hata yakalama mekanizmamız mevcuttur. Anlattıklarımızı akış diyagramında incelersek....





**Şekil-9.1. İstisna Yakalama Mekanizması – I**

Akış şemasında numaralandırılmış olan okları takip ederseniz olayların gelişimini çok rahat bir şekilde kavrayabilirsiniz. Akış diyagramımızı açıklamaya başlayalım;

1. Öncelikle akış, main() yordamının içerisinde başlar. Bu uygulamamızda main() yordamının içerisinde calis() yordamı çağırılmıştır.
2. calis() yordamının içerisinde cokCalis() yordamı çağırılmıştır.
3. cokCalis() yordamının içerisinde istisna oluşmuştur çünkü uygulamamızın yer aldığı dizinin içerisinde *ornek.txt* dosyası aranmış ve bulunamamıştır. Şimdi kritik an geldi, cokCalis() yordamının içerisinde try-catch mekanizması var mı?
4. Evet, cokCalis() yordamının içerisinde try-catch mekanizması olduğu için, catch bloğuna yazılmış olan kodlar çalışır. Bu uygulamamızda ekrana " *Hata Yakalandı =java.io.FileNotFoundException: ornek.txt (The system cannot find the file specified)* " basılır, yani dosyanın olmayışından dolayı bir istisna olduğu belirtilir. Not: *java.io.IOException* istisna tipi, *java.io.FileNotFoundException* istisna tipini kapsadığından bir sorun yaşanmaz bunun nasıl olduğunu biraz sonra inceleyeceğiz.
5. Bitti mi? Tabii ki hayır, uygulamamız kaldığı yerden devam edecektir. Şimdi sıra calis() yordamının içerisindeki henüz çalıştırılmamış olan kodların çalıştırılmasına. Burada da ekrana "calis() yordamı" basılır.

6. Son olarak akış `main()` yordamına geri döner ve `main()` yordamının içerisinde çalıştırılmamış olan kodlar çalıştırılır ve ekrana "`main()` yordamı" basılır.
7. Ve uygulamamız normal bir şekilde sona erer.

Uygulamamızın toplu olarak ekran çıktısı aşağıdaki gibidir.

```
Hata Yakalandi =java.io.FileNotFoundException: ornek.txt (The
system cannot find
the file specified)
calis() yordamı
main() yordamı
```

Akıllara şöyle bir soru gelebilir, "Eğer *ornek.txt* dosyası gerçekten olsaydı yine de `try-catch` mekanizmasını yerleştirmek zorundaydık". Cevap evet, az önce bahseldiği gibi ortada istisna oluşma tehlikesi varsa bile bu tehlikenin önlemi Java programla dilinde önceden kesin olarak alınmalıdır.

*IstisnaOrnek2.java* uygulamamızda, oluşan istisna aynı yordamın içerisinde yakalanmıştır ve böylece uygulamanın akışı normal bir şekilde devam etmiştir. Peki oluşan bu istisnayı aynı yordamın içerisinde yakalamamak gibi bir lüksümüz olabilir mi? Yani oluşan istisna nesnesini -ki bu örneğimizde oluşan istisnamız *java.io.FileNotFoundException* tipindeydi, bir üst kısma fırlatılabilir mi? Bir üst kısma fırlatmaktan kasıt edilen, istisnanın meydana geldiği yordamı çağıran yordama bu istisna nesnesini fırlatmaktır. "Peki ama niye böyle birşeye ihtiyaç duyalım ki?" diyebilirsiniz. Bunun başlıca sebebi, istisnanın olduğu yordam içerisinde, o istisna nesnesi ile ne yapılabileceğinin bilenememesi olabilir. Bir üst kısımda elimizde daha fazla bilgi olabilir, ve bu bilgi çerçevesinde, elimizdeki istisna nesnesini daha güzel bir şekilde değerlendirip, uygulamanın akışını ona göre yönlendirebiliriz.

**Örnek:** *IstisnaOrnek3.java*

```
import java.io.*;

public class IstisnaOrnek3 {

    public void cokCalis() throws IOException{

        File f = new File("ornek.txt");
        BufferedReader bf= new BufferedReader( new
        FileReader(f) );
        System.out.println(bf.readLine());

    }

    public void calis() {
        try {
            cokCalis();
            System.out.println("calis() yordamı");
        } catch(IOException ex) {
            System.out.println("Hata Yakalandi-calis() =" +
ex);
        }

    }

    public static void main(String args[]) {
        IstisnaOrnek3 io3 = new IstisnaOrnek3();
        io3.calis();
        System.out.println("main() yordamı");
    }
}
```

```

    }
}

```

*IstisnaOrnek3.java* örneğimizde oluşan istisna oluştuğu yordam içerisinde yakalanmamıştır. Peki nasıl olurda derleyici buna kızmaz, cevabı hemen aşağıdadır.

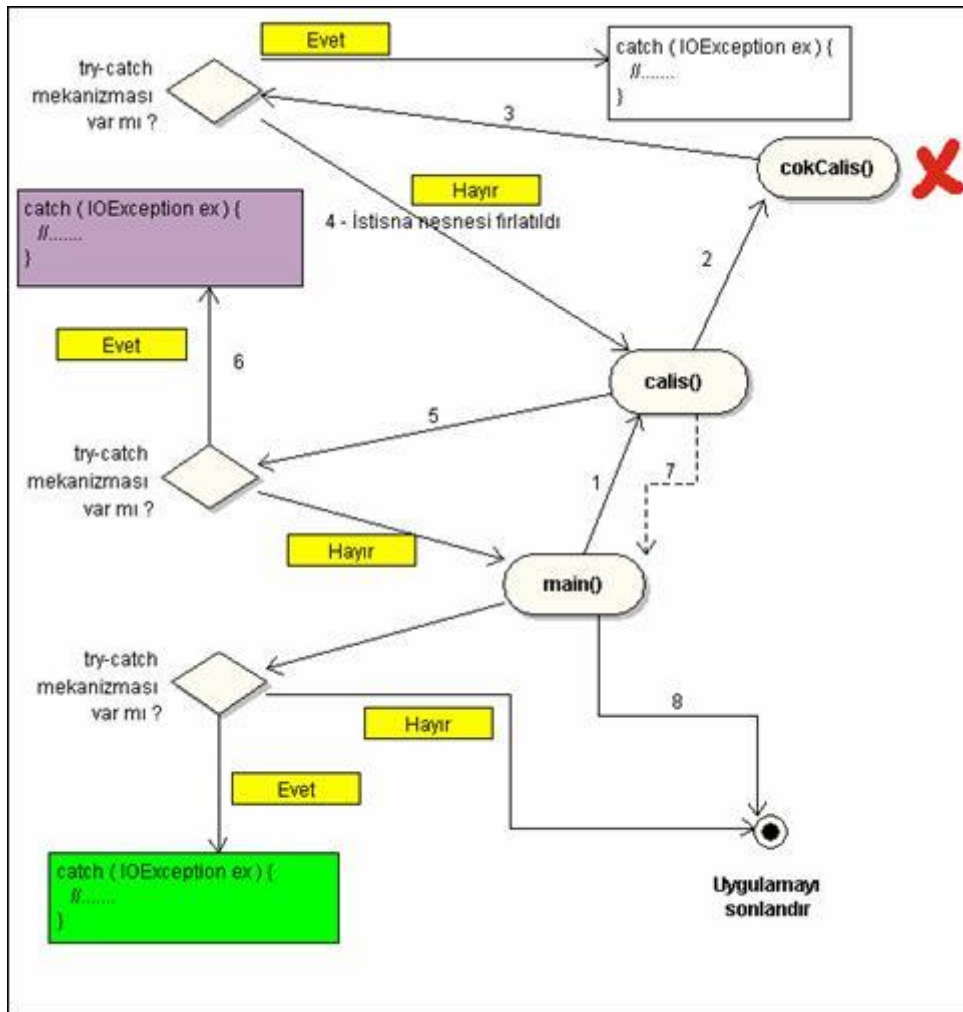
#### Gösterim-8.2:

```

public void cokCalis() throws IOException {
    //..
}

```

Eğer bir istisna oluşursa, istisnanın oluştuğu yordamın yapacağı iki şey vardır demiştik. Birincisi oluşan istisnayı kendi içerisinde try-catch mekanizmasıyla yakalayabilir. İkincisi ise oluşacak olan istisnayı bir üst bölüme (kendisini çağıran yordama) fırlatabilir. Örneğin *cokCalis()* yordamı "throws IOException" diyerek, kendisini çağıran yordamlara şöyle bir mesaj gönderir, "Bakın benim içimde istisnaya yol açabilecek kod var ve eğer istisna oluşursa ben bunu fırlatırım, bu yüzden başınız çaresine bakın". Buraya kadar anlattıklarımızı akış diyagramında incelersek...



Şekil-8.2. İstisna Yakalama Mekanizması - II

Akış şemasında numaralandırılmış olan okları takip ederseniz olayların gelişimini çok rahat bir şekilde kavrayabilirsiniz. Akış diyagramımızı açıklamaya başlayalım;

1. Öncelikle akış, `main()` yordamının içerisinde başlar. Bu uygulamamızda `main()` yordamının içerisinde `calis()` yordamı çağırılmıştır.
2. `calis()` yordamının içerisinde `cokCalis()` yordamı çağırılmıştır.
3. `cokCalis()` yordamının içerisinde istisna oluşmuştur çünkü uygulamamızın yer aldığı dizinin içerisinde `ornek.txt` dosyası aranmış ve bulunamamıştır. Şimdi kritik an geldi, `cokCalis()` yordamının içerisinde `try-catch` mekanizması var mı?
4. Hayır, `cokCalis()` yordamının içerisinde oluşan istisnayı yakalama mekanizması yoktur(`try-catch`) ama `java.io.IOException` tipinde bir hata nesnesi fırlatacağını "`throws IOException`" diyerek belirtmiştir. İstisna oluşmuş ve istisna nesnesi (`java.io.IOException`) bir üst bölüme yani `calis()` yordamına fırlatılmıştır.
5. Artık istisna nesnemiz `calis()` yordamının içerisinde, şimdi sorulması gereken soru "`calis()` yordamının içerisinde hata yakalama mekanizması var mıdır? "
6. `calis()` yordamının içerisinde hata yakalama mekanizması vardır (`try-catch`) bu yüzden `catch` bloğunun içerisindeki kod çalıştırılır ve ekrana "`Hata Yakalandi-calis() =java.io.FileNotFoundException: ornek.txt (The system can not find the file specified)`" basılır, yani dosyanın olmayışından dolayı bir istisna olduğu belirtilir. Dikkat edilirse ekrana "`calis()` yordamı "basılmadı" bunun sebebi istisnanın oluşmasından dolayı akışın catch bloğuna dallanmasıdır. *Not: java.io.IOException istisna tipi, java.io.FileNotFoundException istisna tipini kapsadığından bir sorun yaşanmaz bunun nasıl olduğunu biraz sonra inceleyeceğiz.*
7. Son olarak akış `main()` yordamına geri döner ve `main()` yordamının içerisinde çalıştırılmamış olan kodlar çalıştırılır ve ekrana "`main() yordamı`" basılır.
8. Ve uygulamamız normal bir şekilde sona erer.

Uygulamamızın toplu olarak çıktısı aşağıdaki gibidir.

```
Hata Yakalandi-calis() =java.io.FileNotFoundException: ornek.txt (The
system can not find the file specified)
main() yordamı
```

Bu örneğimizdeki ana fikir, bir istisna kesin olarak oluştuğu yordamın içerisinde yakalanmayabileceğidir. Fırlatma özelliği sayesinde istisna nesnesi (eğer istisna oluşmuş ise) bir üst bölüme yani istisna oluşan yordamı çağırın yordama fırlatılabilir.

Peki bu istisna nesnesi (`java.io.IOException`) `calis()` yordamın yakalanmasaydı ne olurdu? Cevap: O zaman `main()` yordamın yakalanırdı. Nasıl? Hemen gösterelim.

**Örnek:** *IstisnaOrnek4.java*

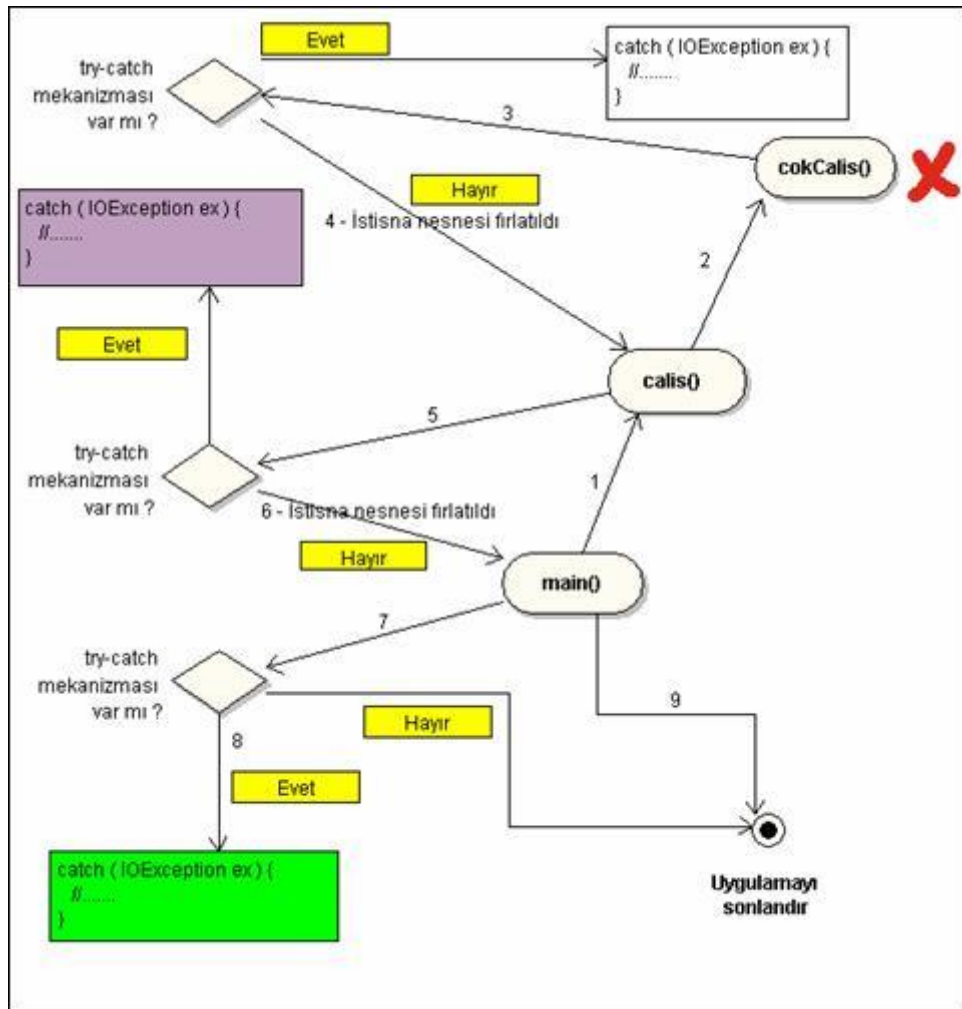
```
import java.io.*;
public class IstisnaOrnek4 {
    public void cokCalis() throws IOException {
        File f = new File("ornek.txt");
        BufferedReader bf = new BufferedReader( new FileReader(
f ) );
        System.out.println(bf.readLine());
    }
    public void calis() throws IOException {
        cokCalis();
    }
}
```

```

        System.out.println("calis() yordamı");
    }
    public static void main(String args[]) {
        try {
            IstisnaOrnek4 io4 = new IstisnaOrnek4();
            io4.calis();
            System.out.println("main() yordamı");
        } catch (IOException ex) {
            System.out.println("Hata Yakalandi-main() =" + ex);
        }
    }
}

```

Bu sefer biraz daha abartıp, oluşan istisna nesnesini son anda main() yordamında yakalıyoruz. Bu örneğimizde hem istisnanın meydana geldiği `cokCalis()` yordamı hem de `calis()` yordamı oluşan istisnayı fırlatmışlardır. Buraya kadar anlattıklarımızı akış diyagramında incelersek...



**Şekil-8.3. İstisna Yakalama Mekanizması - III**

Akış şemasında numaralandırılmış olan okları takip ederseniz olayların gelişimini çok rahat bir şekilde kavrayabilirsiniz. Akış diyagramımızı açıklamaya başlayalım;

1. Öncelikle akış, `main()` yordamının içerisinde başlar. Bu uygulamamızda `main()` yordamının içerisinde `calis()` yordamı çağırılmıştır.
2. `calis()` yordamının içerisinde `cokCalis()` yordamı çağırılmıştır.
3. `cokCalis()` yordamının içerisinde istisna oluşmuştur çünkü uygulamamızın yer aldığı dizinin içerisinde `ornek.txt` dosyası aranmış ve bulunamamıştır. Şimdi kritik an geldi, `cokCalis()` yordamının içerisinde `try-catch` mekanizması var mı?
4. `cokCalis()` yordamının içerisinde oluşan istisnayı yakalama mekanizması yoktur (`try-catch`) ama `java.io.IOException` tipinde bir hata nesnesi fırlatacağını "`throws IOException`" diyerek belirtmiştir. İstisna oluşmuş ve istisna nesnesi (`java.io.IOException`) bir üst bölüme yani `calis()` yordamına fırlatılmıştır.
5. Artık istisna nesnemiz `calis()` yordamının içerisinde, şimdi sorulması gereken soru "`calis()` yordamının içerisinde hata yakalama mekanizması var mıdır? "
6. Cevap hayırdır. `calis()` yordamı da oluşan istisna nesnesini bir üst bölüme yani kendisini çağıran `main()` yordamına fırlatmıştır.
7. İstisna nesnemiz `main()` yordamının içerisine geldi. Sorulması gereken soru "`main()` yordamının içerisinde hata yakalama mekanizması var mıdır? "
8. Cevap evettir. Böylece akış `main()` yordamının içerisindeki `catch` bloğuna dallanır ve `catch` bloğunun içerisindeki kod çalıştırılır.
9. Ve uygulamamız normal bir şekilde sona erer.

Uygulamanın toplu olarak çıktısı aşağıdaki gibidir.

```
Hata Yakalandi-main() =java.io.FileNotFoundException: ornek.txt
(The system cann
ot find the file specified)
```

Oluşan bir istisna nesnesini catch bloğunda yakalamanın ne gibi avantajları olabilir? Bu sorunun cevabına değinmeden evvel olaylara eğer istisna nesnesi **main()** yordamında yakalanmasaydı neler olacağını inceleyerek başlayalım.

**Örnek:** *IstisnaOrnek5.java*

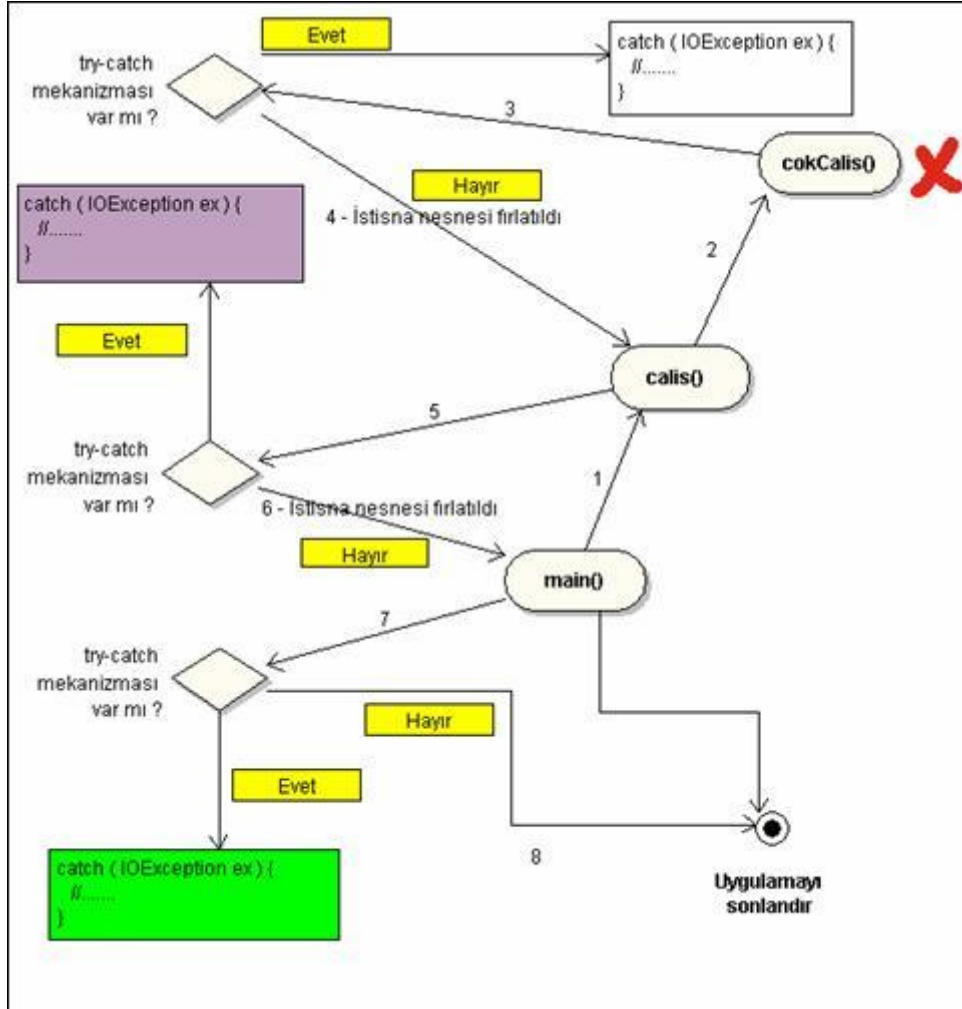
```
import java.io.*;
public class IstisnaOrnek5 {
    public void cokCalis() throws IOException {
        File f = new File("ornek.txt");
        BufferedReader bf = new BufferedReader( new
        FileReader(f));
        System.out.println(bf.readLine());
    }
    public void calis() throws IOException {
        cokCalis();
        System.out.println("calis() yordamı");
    }
    public static void main(String args[]) throws
    IOException {
        IstisnaOrnek5 io5 = new IstisnaOrnek5();
        io5.calis();
        System.out.println("main() yordamı");
    }
}
```

```

}
}

```

Görüldüğü üzere `cokCalis()` yordamının içerisinde oluşan istisna hiçbir yordam içerisinde hata yakalama mekanizması kullanılarak yakalanmamıştır (`try-catch`). Bunun yerine tüm yordamlar bu istisna nesnesini fırlatmayı seçmiştir, buna `main()` yordamı da dahildir. Böyle bir durumda akışın nasıl gerçekleştiğini, akış diyagramında inceleyelim.....



Şekil-8.4. İstisna Yakalama Mekanizması - IV

Akış şemasında numaralandırılmış olan okları takip ederseniz olayların gelişimini çok rahat bir şekilde kavrayabilirsiniz. Akış diyagramımızı açıklamaya başlayalım;

1. Öncelikle akış, `main()` yordamının içerisinde başlar. Bu uygulamamızda `main()` yordamının içerisinde `calis()` yordamı çağırılmıştır.
2. `calis()` yordamının içerisinde `cokCalis()` yordamı çağırılmıştır.
3. `cokCalis()` yordamının içerisinde istisna oluşmuştur çünkü uygulamamızın yer aldığı dizinin içerisinde ornek.txt dosyası aranmış ve bulunamamıştır. Şimdi kritik an geldi, `cokCalis()` yordamının içerisinde `try-catch` mekanizması var mı?
4. `cokCalis()` yordamının içerisinde oluşan istisnayı yakalama mekanizması yoktur (`try-catch`) ama java.io.IOException tipinde bir hata nesnesi fırlatacağını "throws IOException" diyerek



belirtmiştir. İstisna oluşmuş ve istisna nesnesi (java.io.IOException) bir üst bölüme yani `calis()` yordamına fırlatılmıştır.

5. Artık istisna nesnemiz `calis()` yordamının içerisinde, şimdi sorulması gereken soru "`calis()` yordamının içerisinde hata yakalama mekanizması var mıdır? "
6. Cevap hayırdır. `calis()` yordamı da oluşan istisna nesnesini bir üst bölüme yani kendisini çağıran `main()` yordamına fırlatmıştır.
7. İstisna nesnemiz `main()` yordamının içerisine geldi. Sorulması gereken soru " `main` yordamının içerisinde hata yakalama mekanizması var mıdır? "
8. Cevap hayırdır. Peki ne olacak? Çok basit, uygulama doğal olarak sonla-nacaktır.

Uygulamanın toplu olarak çıktısı aşağıdaki gibidir.

```
Exception in thread "main" java.io.FileNotFoundException:
ornek.txt (The system
cannot find the file specified)
    at java.io.FileInputStream.open(Native Method)
    at java.io.FileInputStream.<init>(FileInputStream.java:103)
    at java.io.FileReader.<init>(FileReader.java:51)
    at IstisnaOrnek5.cokCalis(IstisnaOrnek5.java:8)
    at IstisnaOrnek5.calis(IstisnaOrnek5.java:13)
    at IstisnaOrnek5.main(IstisnaOrnek5.java:19)
```

"Hata yakalama mekanizması koyduğumuzda da uygulama sonlanıyordu, şimdide sonlandı bunda ne var ki" diyebilirsiniz. Haklı olabilirsiniz ama önce oluşan bir istisna nesnesi `catch` bloğunda yakalamanın ne gibi avantajları olabilir?

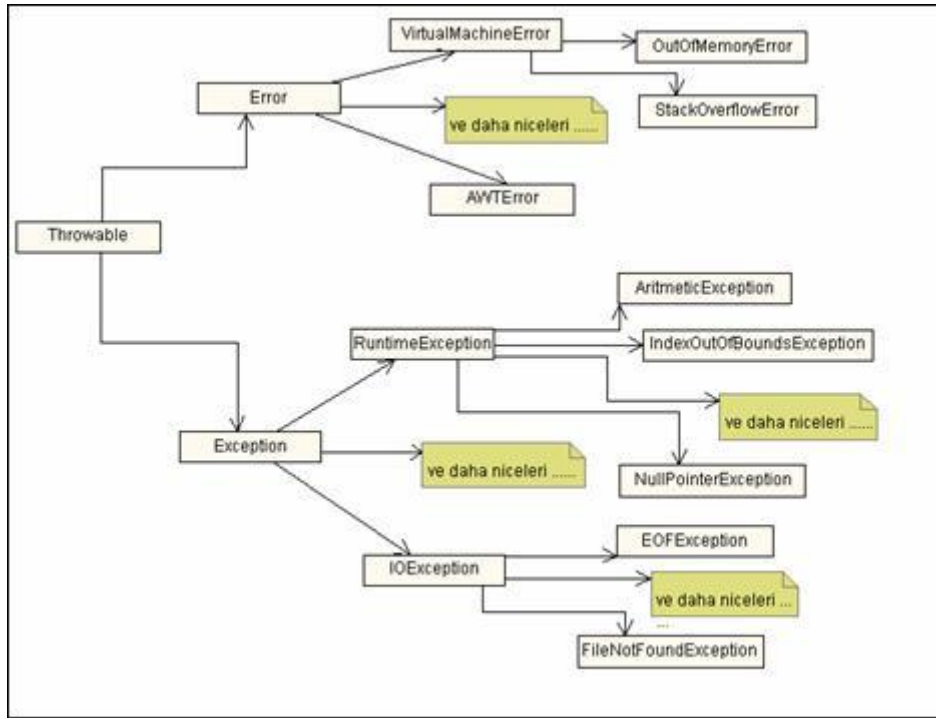
Oluşan bir istisna nesnesini `catch` bloğundan yakalamak, daha doğrusu hata yakalama mekanizması kullanmak uygulamayı yazan kişilere büyük kolaylıklar sağlar. En büyük avantaj oluşan hatayı `catch` bloğunun içerisinde kaydedilirsiniz (*logging*) (dosyaya ama veri tabanına... gibi gibi...) . Örneğin iyi işleyen bir uygulama yazdınız ve bu uygulama yaptığınız tüm -daha doğrusu aklınıza gelen- testlerden geçmiş herşey harika, kendinize güveniniz gelmiş, dünya gözünüze artık bambaşka bir yer gibi geliyor ama bir gün bir bakıyorsunuz ki uygulamanız çalışması durmuş!! ilk yapacağınız şey "bu uygulamayı kim kapattı!" diye etrafa sormak oysaki kimsenin günahı yok, kimse elini uygulamanıza sürmemiştir zaten böyle bir riski kim alabilir ki? Asıl gerçek, uygulamada ters giden birşey olmuş ve uygulama kapanmıştır. İşte tam o anda tutunacağınız tek dal dosyaya veya veri tabanına kayıt ettiğiniz hata mesajlarıdır. Bu bakımdan `catch` bloğunun içerisine oluşan hata ile alakalı ne kadar detaylı bilgi gömerseniz, bu bilgi sizi ileride -eğer hata oluşursa- o kadar yardımcı olacaktır.

*IstisnaOrnek5.java* kötü bir uygulama örneğidir. Oluşabilecek olan bir istisna, hata yakalama mekanizması (`try-catch`) ile sizin öngördüğünüz bir yerde yakalanmalıdır. Bir istisna meydana geldiği zaman uygulama mutlaka sonlanmak zorunda değildir. Eğer bir telafisi var ise bu `catch` bloğunun içerisinde yapılmalı ve uygulama tekrardan ayağa kaldırılmalıdır ama çok ölümcül bir hata ise o zaman hata mesajını kaydetmekten (dosyaya veya veri tabanına.. gibi gibi...) başka yapılacak pek fazla birşey yoktur.

#### 8.1.5. İstisna Tip Hiyerarşisi

Nasıl olurda java.io.IOException istisna tipi, *java.io.FileNotFoundException* istisna tipini kapsayabilir? Kapsamak ne demektir? Kapsamak demek, eğer uygulamanızda *java.io.FileNotFoundException* tipinde bir istisna nesnesi oluşmuşsa (bir istisna oluşmuşsa) bu istisna tipini *java.io.IOException* tipini kullanarak da `catch` bloğunda yakalayabileceğiniz anlamına gelir.





**Şekil-8.5. İstisna Tip Hiyerarşisi**

Yukarıdaki şemamızdan görüleceği üzere, *FileNotFoundException* istisna tipi, *IOException* istisnasının alt kümesi olduğu için, *FileNotFoundException* tipinde bir istisna nesnesini *catch* bloğunun içerisinde *IOException* istisna tipiyle yakalayabiliriz.

*Throwable* istisna nesnesi, tüm istisna nesnelerinin atasıdır. Yukarıdaki şemamıza bakarak istisnaları 3 gruba ayırabiliriz.

- **Error** istisna tipi ölümcül bir hatayı işaretler ve telafisi çok zordur, neredeyse imkansızdır. Örneğin *OutOfMemoryError* (yetersiz bellek) istisnası oluşmuş ise uygulamanın buna müdahale edip düzeltilmesi imkansızdır.
- **RuntimeException** istisna tipleri, eğer uygulama normal seyrinde giderse ortaya çıkmaması gereken istisna tipleridir. Örneğin *ArrayIndexOutOfBoundsException* istisna tipi, bir dizinin olmayan elemanına eriştiğimiz zaman ortaya çıkan bir istisnadır. *RuntimeException* istisna tipleri, kontrolsüz kodlamadan dolayı meydana gelen istisna tipleri diyebiliriz. Biraz sonra bu istisna tipini detaylı biçimde inceleyeceğiz.
- Ve **diğer Exception tipleri**. Bu istisna tipleri çevresel koşullardan dolayı meydana gelebilir. Örneğin erişmeye çalışan dosyanın yerinde olmaması (*FileNotFoundException*) veya network bağlantısının kopması sonucu ortaya çıkabilecek olan istisnalardır ve bu istisnalar için önceden bir tedbir alınması şarttır.

#### 8.1.5.1. Tüm Diğer Exception İstisna Tiplerini Yakalamak

Bir uygulama içerisinde oluşabilecek olan tüm istisna tiplerini yakalamak için aşağıdaki ifadeyi kullanabilirsiniz.

#### **Gösterim-8.3:**

```
catch (Exception ex) {
```

```
//.....  
}
```

Tüm istisnaları yakalamak (*Error*, *RuntimeException* ve diğer *Exception* türleri) için *Throwable* istisna tipini kullanmak iyi fikir değildir. Bunun yerine bu üç gruba ait daha özellikli istisna tiplerinin kullanılmasını önerilir.

#### 8.1.5.2. *RuntimeException* İstisna Tipleri

*DiziErisim.java* uygulama örneğimiz içerisinde istisna oluşma riski olmasına rağmen nasıl oldu da Java buna kızmayarak derledi? Peki ama *IstisnaOrnek1.java* uygulamasını niye derlemedi? Bu soruların cevapları istisna tiplerinin iyi bilinmesi ile ortaya çıkar.

*DiziErisim.java* uygulama örneğinde istisna oluşma riski vardır. Eğer uygulamayı yazan kişi dizinin olmayan bir elemanına erişmeye kalkarsa *ArrayIndexOutOfBoundsException* hatası alacaktır, yani *RuntimeException* (çalışma-anı hatası). Peki bunun sebebi nedir? Bunun sebebi kodu yazan arkadaşın dikkatsizce davranmasıdır. Bu tür hatalar derleme anında (*compile-time*) fark edilemez. Java bu tür hatalar için önceden bir tedbir alınmasını şart koşmaz ama yine de tedbir almakta özgürsünüzdür. Bir dosyaya erişirken oluşacak olan istisnaya karşı bir tedbir alınmasını, Java şart koşar çünkü bu tür hatalar diğer *Exception* istisna tipine girer. Genel olarak karşılaşılan *RuntimeException* istisna türlerine bir bakalım;

- ***ArithmeticException***: Bir sayının sıfıra bölünmesiyle ortaya çıkabilecek olan bir istisna tipidir.

#### Gösterim-8.4:

```
int i = 16 / 0 ; // ArithmeticException ! hata !
```

- ***NullPointerException***: Bir sınıf tipindeki referansı, o sınıfa ait bir nesneye bağlamadan kullanmaya kalkınca alınabilecek bir istisna tipi.

#### Gösterim-8.5:

```
String ad == null;  
// NullPointerException ! hata !  
System.out.println("Ad = " + ad.trim() );
```

Bu hatayı almamak için;

#### Gösterim-8.6:

```
String ad = " Java Kitap Projesi "; // baglama islemi  
System.out.println("Ad = " + ad.trim() ); //dogru
```

- ***NegativeArraySizeException***: Bir diziyi negatif bir sayı vererek oluşturmaya çalışırsak, bu istisna tipi ile karşılaşırız.

#### Gösterim-8.7:

```
// NegativeArraySizeException ! hata !  
int dizi[] = new dizi[ -100 ];
```

- **ArrayIndexOutOfBoundsException**: Bir dizinin olmayan elemanına ulaşmak istendiği zaman karşılaşılan istisna tipidir. Daha detaylı bilgi için *DiziErisim.java* uygulama örneğini inceleyiniz.

- **SecurityException**: Genellikle tarayıcı (*browser*) tarafından fırlatılan bir istisna tipidir. Bu istisnaya neden olabilecek olan sebepler aşağıdaki gibidir;

- Applet içerisinden, yerel (*local*) bir dosyaya erişilmek istendiği zaman.
- Appletin indirildiği sunucuya (*server*) değilde değişik bir sunucuya bağlantı kurulmaya çalışıldığı zaman.
- Applet içerisinde başka bir uygulama başlatmaya çalışıldığı zaman.

*SecurityException* istisna tipi fırlatılır.

Önemli noktayı bir kez daha vurgulayalım, *RuntimeException* ve bu istisna tipine ait alt tipleri yakalamak için, Java derleme anında (*compile-time*) bizlere bir bir zorlama yapmaz.

#### 8.1.6. İstisna Mesajları

Bir istisna nesnesinden bir çok veri elde edebilirsiniz. Örneğin istisna oluşumunun yol haritasını izleyebilirsiniz veya istisna oluşana kadar hangi yordamların çağrıldığını öğrenebilirsiniz.

Bu bilgileri elde etmek için kullanılan *Throwable* sınıfına ait *getMessage()*, *getLocalizedMessage()* ve *toString()* yordamlarının ne iş yaptıklarını örnek uygulama üzerinde inceleyelim.

**Örnek:** *IstisnaMetodlari.java*

```
public class IstisnaMetodlari {
    public void oku() throws Exception {
        throw new Exception("istisna firlatildi"); //
dikkat
    }
    public static void main(String args[]) {
        try {
            IstisnaMetodlari im = new IstisnaMetodlari();
            im.oku();
        } catch (Exception ex) {
            System.out.println("Hata- ex.getMessage() : " +
ex.getMessage() );
            System.out.println("Hata-
ex.getLocalizedMessage() : " +
ex.getLocalizedMessage() );
            System.out.println("Hata- ex.toString() : " + ex
);
        }
    }
}
```

*oku()* yordamının içerisinde bilinçli olarak *Exception* (istisna) nesnesi oluşturulup fırlatılmıştır. Bu istisna sınıfının yapılandırıcısına ise kısa bir not düştüm. *main()* yordamının içerisindeki *catch*

blogunda *Exception* istisna sınıfına ait yordamlar kullanılarak, oluşan istisna hakkında daha fazla bilgi alınabilir. *Exception* sınıfı *Throwable* sınıfından türediği için, *Throwable* sınıfı içerisindeki erişilebilir olan alanlar ve yordamlar otomatik olarak *Exception* sınıfının içerisinde de bulunur. Bu yordamların detaylı açıklaması aşağıdaki gibidir.

#### **String getMessage()**

Oluşan istisnaya ait bilgileri *String* tipinde geri döner. Bu örneğimizde bilgi olarak "istisna fırlatıldı" mesajını yazdık. Mesajın *String* olmasından dolayı bu yordam bize bu bilgiyi *String* tipinde geri döndürecektir. Eğer *Exception* sınıfının yapılandırıcısına birşey gönderilmeseydi; o zaman `null` değeri döndürülürdü.

#### **String getLocalizedMessage()**

Bu yordam, *Exception* sınıfından türetilmiş alt sınıflar tarafından iptal edilebilir (*override*). Biraz sonra kendi istisna sınıflarımızı nasıl oluşturacağımızı gördüğümüzde, bu yordam daha bir anlam taşıyacaktır. Eğer bu yordam alt sınıflar tarafından iptal edilmemiş ise `getMessage()` yordamı ile aynı sonucu döndürür.

#### **String toString()**

Oluşan istisna hakkında kısa bir açıklamayı *String* tipinde geri döner. Eğer istisna sınıfına ait nesne; bir açıklama ile oluşturulmuş ise - `new Exception ("hata fırlatıldı")` - bu açıklamayı da ekrana basar. `toString()` yordamı oluşan istisna ile ilgili bilgiyi belli bir kural içerisinde ekrana basar.

- Oluşan istisna nesnesinin tipini ekrana basar.
- ": " iki nokta üst üste koyar ve bir boşluk bırakır.
- Son olarak `getMasseege()` yordamı çağrılır ve buradan - eğer bilgi varsa- ekrana basılır.

Eğer oluşan istisna sınıfına ait nesne bir açıklama ile oluşturulmamış ise yani direk - `new Exception()` - diyerek oluşturulmuş ise son adımda hiçbirsey basmaz.

Uygulamamızın çıktısı aşağıdaki gibi olur.

```
Hata- ex.getMessage() : istisna fırlatildi
Hata-ex.getLocalizedMessage() : istisna fırlatildi
Hata- ex.toString() : java.lang.Exception: istisna fırlatildi
```

#### **Throwable getCause()**

Java 1.4 ile gelen *Throwable* sınıfına ait bir başka yordam ise `getCause()` yordamıdır. Bu yordam *Throwable* nesnesine bağlı referans geri döner. Buradaki amaç, oluşmuş olan istisnanın -eğer varsa- sebebini daha detaylı bir biçimde yakalamaktır.

#### **Örnek: IstisnaMetodlari2.java**

```
import java.io.*;

public class IstisnaMetodlari2 {

    public void oku() throws Exception {
        throw new Exception("istisna fırlatildi",
```

```

new IOException() ); //
dikkat
}

public static void main(String args[]) {
    try {
        IstisnaMetodlari2 im2 = new IstisnaMetodlari2();
        im2.oku();
    } catch (Exception ex) {
        System.out.println("Hata-ex.getCause():" +
ex.getCause());
    }
}
}

```

Bu örnek için *java.io.IOException* kullanıldığı için `import java.io.*` denilmeliydi. Bu kısa açıklamadan sonra detayları vermeye başlayalım.

`getCause()` yordamın işe yaraması için, istisna sınıfına ait yapılandırıcının içerisine bu istisnaya sebebiyet vermiş olan istisna tipini yerleştirmemiz gerekmektedir. Tekrardan belirtelim bu yordam *Throwable* nesnesine bağlı bir referans geri döndürür.

#### **Gösterim-8.8:**

```

throw new Exception("istisna fırlatildi",new
IOException()); // dikkat

```

Böyle bir ifade nerede işimize yarar ki diyebilirsiniz. Bu olay aslında aynı anda iki tip istisna fırlatabilmenize olanak tanır ve bu çoğu yerde işinize yarayabilir. Eğer istisnanın olduğu yerde alt istisna nesnesi belirtilmemiş ise - `throw new Exception ("istisna fırlatildi")` gibi- `getCause()` yordamı "null" dönecektir.

Uygulamanın çıktısı aşağıdaki gibi olur.

```

Hata- ex.getCause() : java.io.IOException

```

#### **Throwable initCause (Throwable cause)**

Java 1.4 ile gelen bir başka yenilik ise `initCause()` yordamıdır. Bu yordam iki istisna tipini birleştirmeye yarar.

#### **Örnek:** *IstisnaMetodlari3.java*

```

import java.io.*;

public class IstisnaMetodlari3 {

    public void oku() throws Throwable {
        Exception ioEx = new IOException(); // dikkat
        Exception fnfEx = new FileNotFoundException(); //
dikkat
        Throwable th = ioEx.initCause(fnfEx);
        throw th;
    }
}

```

```

    }

    public static void main(String args[]) {
        try {
            IstisnaMetodlari3 im3 = new IstisnaMetodlari3();
            im3.oku();
        } catch (Throwable th) {
            // Throwable th = ex.getCause();
            //Throwable th2 = ex.initCause(th); //hata
            System.out.println("Hata-th.initCause():"+ th );
            System.out.println("Hata-th.getCause():"+
th.getCause() );
        }
    }
}

```

oku() yordamının içerisinde iki ayrı tipte istisna nesnesi oluşturulmuştur. Bu istisna nesneleri birleştirilerek tek bir **Throwable** tipinde nesne oluşturmak mümkündür. Hatanın yakalandığı yerde birleştirilen bu iki istisna tipine ayrı ayrı ulaşılabilir.

Eğer bir istisna **Throwable (Throwable)** veya **Throwable (String, Throwable)** ile oluşturulmuş ise **initCause()** yordamı çağrılmaz.

Uygulamanın çıktısı aşağıdaki gibi olur.

```

Hata - th.initCause() : java.io.IOException
Hata - th.getCause() : java.io.FileNotFoundException

```

#### **printStackTrace() :**

Oluşan bir hatanın yol haritasını **printStackTrace()** yordamı sayesinde görebilirsiniz.

#### **Örnek:** *IstisnaMetodlari4.java*

```

public class IstisnaMetodlari4 {

    public void cokOku() throws Exception {
        System.out.println("cokOku() yordamı cagrildi");
        throw new Exception("istisna olustu"); //dikkat
    }
    public void oku() throws Exception {
        System.out.println("oku() yordamı cagrildi");
        cokOku();
    }
    public static void main(String args[]) {
        try {
            IstisnaMetodlari4 im4 = new IstisnaMetodlari4();
            im4.oku();
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}

```

Yol haritasına, bir istisna oluşmuş ise bunun hangi satırda meydana gelmiş, istisnanın olduğu yordamı hangi yordam çağırmış gibi soruların cevaplarının bulunduğu bir çeşit bilgi kümesi diyebiliriz. `printStackTrace()` yordamı hatayı `System.err` kullanarak kullanıcıya iletir. Bunun ne gibi avantajları var dersanız hemen açıklayalım: Eğer bir uygulamanın çıktısını dosyaya veya buna benzer bir yere yönlendirmiş iseniz `System.out` kullanarak yazılmış ifadeler yine bu dosyalara ve buna benzer yerlere yazılacaktır.

```
$ java Test > a.txt
```

Fakat `System.err` kullanılarak yazılmış bir ifade, uygulama nereye yönlendirilmiş olursa olsun kesin olarak konsola yazılır ve kullanıcının dikkatine sunulur.

#### **`printStackTrace (PrintStream s)`**

`PrintStream` sınıfına ait nesne kullanılarak, oluşan istisnanın yol haritasını konsol yerine başka bir yere bastırmanız mümkündür. Başka bir yer derken, örneğin bir dosya veya ağ (*network*) bağlantısı ile başka bir bilgisayara oluşan bu istisnanın yol haritasını gönderebilirsiniz.

#### **`printStackTrace (PrintWriter s)`**

`PrintWriter` sınıfına ait nesne kullanılarak, oluşan istisnanın yol haritasını konsol yerine başka bir yere bastırmanız mümkündür. Özellikle JSP ve Servlet kullanırken oluşan bir istisnanın yol haritasını HTTP/HTTPS kanalı ile kullanıcılara gösterilebilir.

*IstisnaMetodlari4.java* uygulamamızın çıktısı aşağıdakidir.

```
oku() yordamı cagrildi
cokOku() yordamı cagrildi
java.lang.Exception: istisna olustu
    at IstisnaMetodlari4.cokOku(IstisnaMetodlari4.java:10)
    at IstisnaMetodlari4.oku(IstisnaMetodlari4.java:15)
    at IstisnaMetodlari4.main(IstisnaMetodlari4.java:22)
```

#### **`Throwable fillInStackTrace()`**

Oluşan bir istisnanın yol haritasını `Throwable` nesnesi içerisinde elde etmeniz için `fillInStackTrace()` yordamını kullanmalısınız. Bu olay istisnanın tekrardan fırlatılması söz konusu olduğunda - biraz sonra inceleyeceğiz - faydalı olabilir.

**Örnek:** *IstisnaMetodlari5.java*

```
public class IstisnaMetodlari5 {

    public void cokOku() throws Exception {
        System.out.println("cokOku() yordamı cagrildi");
        throw new Exception("istisna olustu");
    }

    public void oku() throws Exception {
        System.out.println("oku() yordamı cagrildi");
        cokOku();
    }

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

```

        try {
            IstisnaMetodlari5 im5 = new IstisnaMetodlari5();
            im5.oku();
        } catch (Exception ex) {
            Throwable t = ex.fillInStackTrace();
            System.err.println( t.getMessage() );
        }
    }
}

```

Bu method oluşan istisnanın yol haritasına müdahale ederek değiştirir ve değiştirilen bilgiler ışığında yeni bir *Throwable* nesnesi oluşturulur.

Uygulamanın çıktısı aşağıdaki gibidir.

```

oku() yordamı cagrildi
cokOku() yordamı cagrildi
istisna olustu

```

#### **StackTraceElement[] getStackTrace()**

Yine Java 1.4 ile birlikte gelen `getStackTrace()` yordamı, `printStackTrace()` yordamı ile oluşan hata satırlarını *StackTraceElement* tipindeki dizi nesnesine çevirir.

**Örnek:** *IstisnaMetodlari6.java*

```

public class IstisnaMetodlari6 {

    public void cokOku() throws Exception {
        System.out.println("cokOku() yordamı cagrildi");
        throw new Exception("istisna olustu");
    }

    public void oku() throws Exception {
        System.out.println("oku() yordamı cagrildi");
        cokOku();
    }

    public static void main(String args[]) {
        try {
            IstisnaMetodlari6 im6 = new IstisnaMetodlari6();
            im6.oku();
        } catch (Exception ex) {
            StackTraceElement[] ste = ex.getStackTrace(); //
dikkat
            for(int i=0 ;i < ste.length;i++) {
                System.err.println("-->" + ste[i].getFileName()
+" - "+
ste[i].getMethodName() +" - "+
ste[i].getLineNumber() );
            }
        }
    }
}

```



```
}  
}
```

Oluşan istisnanın yol haritası bilgilerine ulaşmak için `getStackTrace()` yordamı kullanılmalıdır. Bu yordam, oluşan istisnaya ait yol bilgilerini bir *StackTraceElement* dizisi şeklinde sunar. `printStackTrace()` yordamının çıktısı göz önüne getirirsek, buradaki ilk satır, *StackTraceElement* dizisinin ilk elemanına denk gelir. Bu dizinin son elemanında, oluşan istisna yol haritasının son satırına denk gelir.

Uygulamamızın çıktısı aşağıdaki gibi olur.

```
oku() yordamı cagrildi  
cokOku() yordamı cagrildi  
--> IstisnaMetodlari6.java - cokOku - 8  
--> IstisnaMetodlari6.java - oku - 13  
--> IstisnaMetodlari6.java - main - 20
```

**setStackTrace (StackTraceElement[] stackTrace)**

Son olarak inceleyeceğimiz yordam yine Java 1.4 ile birlikte gelen `setStackTrace()` yordamıdır. Bu yordam sayesinde oluşan istisnanın yol haritası değiştirilebilir.

**Örnek:** *IstisnaMetodlari7.java*

```
public class IstisnaMetodlari7 {  
  
    public void cokOku() throws Exception {  
        System.out.println("cokOku() yordamı cagrildi");  
        Exception eE = new Exception("istisna olustu-1");  
        // dikkat  
        System.out.println("-----");  
        Exception eE2 = new Exception("olusan istisna-2");  
        // dikkat  
        eE2.setStackTrace( eE.getStackTrace() ); // dikkat  
        throw eE2; // dikkat  
    }  
  
    public void oku() throws Exception {  
        System.out.println("oku() yordamı cagrildi");  
        cokOku();  
    }  
  
    public static void main(String args[]) {  
        try {  
            IstisnaMetodlari7 im7 = new IstisnaMetodlari7();  
            im7.oku();  
        } catch (Exception ex) {  
            StackTraceElement[] ste = ex.getStackTrace(); //  
            dikkat  
            for(int i=0 ;i < ste.length;i++) {  
                System.err.println("-->" + ste[i].getFileName()  

```

```

+" - "+

ste[i].getMethodName()+" - "+

ste[i].getLineNumber() );
    }
}
}
}

```

Bu değişimden sonra `getStackTrace()` veya `printStackTrace()` gibi benzeri yordamlar artık değişen bu yeni yol haritasını basacaklardır.

Uygulamamızın çıktısı aşağıdaki gibi olur.

```

oku() yordamı cagrildi
cokOku() yordamı cagrildi
-----
--> istisnametodlari7.java - cokOku - 6
--> istisnametodlari7.java - oku - 16
--> istisnametodlari7.java - main - 23

```

Yukarıdaki örneğimizde fırlatılan istisnanın çıkış noktası 9. satırda olmasına rağmen, `setStackTrace()` yordamı kullanarak oluşan istisnanın yol haritasında değişiklik yapabildik. Artık fırlatılan istisnanın yeni çıkış noktasını 6. satır olarak gösterilmektedir.

### 8.1.7. Kendi İstisnalarımızı Nasıl Oluşturabiliriz?

Javanın kendi içerisinde tanımlanmış istisna tiplerinin dışında, bizlerde kendimize özgü istisna tiplerini oluşturup kullanabiliriz. Sonuçta istisnalar da birer nesnedir ve kendilerine has durumları ve özellikleri olabilir. İlk istisna sınıfını oluşturalım;

**Örnek:** *BenimHatam.java*

```

public class BenimHatam extends Exception {
    private int id ;

    public BenimHatam() {
    }

    public BenimHatam(String aciklama) {
        super(aciklama); // dikkat
    }

    public BenimHatam(String aciklama , int id) {
        super(aciklama); //dikkat
        this.id = id ;
    }

    public String getLocalizedMessage() { // iptal etme
        (override)

        switch(id) {

```

```

        case 0 : return "onemsiz hatacik" ;
        case 1 : return "hata" ;
        case 2 : return "! onemli hata !" ;
        default: return "tanimsiz hata";
    }
}

public int getId() {
    return id;
}
}

```

İlk istisna sınıfımızın ismi *BenimHatam*. Şimdi olaylara geniş açıdan bakıldığında görülmesi gereken ilk şey, bir sınıfın istisna tipleri arasında yer alabilmesi için *Exception* sınıfından türetilmesi gerektiğidir.

### **Gösterim-8.9:**

```

public class BenimHatam extends Exception {
    //..
}

```

*Exception* sınıfından türetilen bir sınıf artık istisna sınıfları arasında yerini almaya hazırdır. Fakat önce bir kaç önemli noktayı gün ışığına çıkartalım. Öncelikle *BenimHatam* istisna sınıfının yapılandırıcılarına (*constructor*) dikkat etmenizi istiyorum. *BenimHatam* istisna sınıfının iki adet yapılandırıcısı (*constructor*) bulunmaktadır, bunlardan biri *String* tipinde diğeri ise bir *String* birde ilkel *int* tipinde parametre kabul etmektedir. Bu yapılandırıcıların ortak olarak aldıkları parametre tipi *String* tipidir. Niye?

*BenimHatam* istisna sınıfından anlaşılacağı üzere, bu sınıfımızın içerisinde aynı diğer sınıflarımızda oldu gibi yordamlar tanımlayabildik, örneğin *getId()* yordamı. Bu yordam hataya ait *Id* numarasını dönmektedir. “Hangi hata numarası, bu da ne ?” demeyin çünkü bunu yordamlara zenginlik katması için ekledim.

Bu sınıfımızın içerisinde *Throwable* (*Exception* sınıfının da *Throwable* sınıfından türetildiğini unutmayalım) sınıfının bir yordamı olan *getLocalizedMessage()* yordamını iptal ettik (*override*). Yukarıdaki açıklamalardan hatırlayacağınız üzere eğer *getLocalizedMessage()* yordamı iptal edilmez ise *getMessage()* ile aynı açıklamayı dönerdi (bkz: istisna yordamları). Fakat biz burada sırf heyacan olsun diye her numarayı bir açıklama ile eşleştirip geri döndürmekteyiz. Örneğin "sıfır=önemsiz hata", "bir= !önemli hata!" gibi gibi, tabii bunlarda tamamen hayal ürünü olarak yazılmıştır.

İstisna sınıflarının yapılandırıcılarına *String* ifade göndermenin amacı *getMessage()* yordamının yaptığı işi anlamaktan geçer. Tabii sadece istisna sınıfının yapılandırıcısına *String* tipte parametre göndermek ile iş bitmez. Bu gönderilen parametre eğer *super(String)* komutu çağrılırsa amacına ulaşır. Peki amaç nedir? Diyelim ki bir dosya açmak istediniz ama açılmak istenen dosya yerinde değil? Böyle bir durumda *java.io.FileNotFoundException* tipinde bir istisna nesnesi oluşturulup fırlatılacaktır. Oluşan istisna ile (dosyanın bulunamaması) oluşan istisna tipinin ismi (*java.io.FileNotFoundException*) arasında bir ilişki kurabiliyor musunuz? "*FileNotFoundException*" ifadesinin Türkçesi "dosya bulunamadı istisnası" demektir, bingo!. Böyle bir durumda oluşan istisnayı, bu istisna için oluşturulan bakarak aklımızda bir ışık yanabilir. Peki her istisna tipinin ismi bu kadar açıklayıcı mıdır? Örneğin *SQLException*, bu ifadenin Türkçesi "SQL istisnası" demektir. Böyle bir istisna bazı zamanlarda anlamsız gelebilir. Evet SQL istisnası çok güzel ama niye? Ne ters gitti de ben bu hatayı aldım, ek açıklama yok mu diyeceğimiz anlar olmuştur ve olacaktır. Sonuçta ek açıklamalara ihtiyaç olduğu bir gerçektir. İşte bu ek açıklamaları bu istisna sınıflarının yapılandırıcılarına gönderebiliriz. Böylece oluşan istisna nesnesinin ismi ve bizim vereceğimiz ek açıklamalar ile sır perdesini aralayabiliriz. Ek açıklamanın *String* tipinde olmasına herhalde kimsenin bir itirazı yoktur. İşte bu yüzden iyi tasarlanmış bir istisna sınıfının *String* tipinde parametre kabul

eden yapılandırıcıları vardır. Ama ek olarak yukarıdaki örneğimizde olduğu gibi hem *String* hem de ilkel (*primitive*) *int* tipinde parametre kabul eden yapılandırıcılar olabilir. Fazla parametre göz çıkartmaz.

Şimdi ikinci istisna sınıfımız olan *SeninHatan* sınıfını inceleyelim;

**Örnek:** *SeninHatan.java*

```
public class SeninHatan extends Exception {
    public SeninHatan() {
    }
    public SeninHatan(String aciklama) {
        super(aciklama); // dikkat
    }
}
```

*SeninHatan* istisna sınıfı bir öncekine (*BenimHatam*) göre daha sadedir. Şimdi tek eksiğimiz bu istisna sınıflarımızın kullanıldığı bir kobay örnek. Onu da hemen yazalım.

**Örnek:** *Kobay.java*

```
public class Kobay {
    public void cikart(int a,int b) throws BenimHatam,
    SeninHatan{
        if(a == 0) {
            throw new SeninHatan("a parametresi sifir
geldi");
        }
        if(b == 0) {
            throw new SeninHatan("b parametresi sifir
geldi");
        }
        if( (a<0) || (b<0) ) {
            throw new SeninHatan(); // kotu, aciklama yok
        }
        int sonuc = a - b ; // hesaplama islemi

        if(sonuc < 0) {
            throw new BenimHatam("sonuc eksi",2);
        }else if( sonuc == 0) {
            throw new BenimHatam("sonuc sifir",1);
        }
    }
    public static void main(String args[]) {
        System.out.println("-----");
        try {
            Kobay it = new Kobay();
            it.cikart(1,2);
        } catch (BenimHatam ex1) {
            System.out.println( "Hata Olustu-1:"+
ex1.getMessage() );
            System.out.println(ex1.getLocalizedMessage());
            System.out.println(ex1.getId());
        } catch (SeninHatan ex2) {
            System.out.println("Hata Olustu-2:"+ ex2);
        }
    }
}
```

```

        System.out.println("-----");
        try {
            Kobay it = new Kobay();
            it.cikart(1,0);
        } catch (BenimHatam ex1) {
            System.out.println("Hata Olustu-1:"+
ex1.getMessage());
            System.out.println(ex1.getLocalizedMessage());
            System.out.println(ex1.getId());
        } catch (SeninHatan ex2) {
            System.out.println("Hata Olustu-2:"+ ex2);
        }

        System.out.println("-----");
        try {
            Kobay it = new Kobay();
            it.cikart(1,-124);
        } catch (BenimHatam ex1) {
            System.out.println("Hata Olustu-1:"+
ex1.getMessage());
            System.out.println(ex1.getLocalizedMessage());
            System.out.println(ex1.getId());
        } catch (SeninHatan ex2) {
            System.out.println("Hata Olustu-2:"+ ex2);
        }
    }
}

```

Yukarıdaki örnekte üç adet harekete kızılmaktadır. Bunlar sırasıyla:

- Sonucun eksi çıkması durumunda *BenimHatam* tipinde istisna oluşmaktadır.
- Parametrelerden birinin sıfır gönderilmesi durumunda *SeninHatan* tipinde istisna oluşmaktadır
- Parametrelerden birinin eksi gönderilmesi durumunda *SeninHatan* tipinde istisna oluşmaktadır

Eğer *BenimHatam* tipinde bir istisna oluşursa nasıl detaylı bilgi alınacağına lütfen dikkat edin. Aynı şekilde *SeninHatan* tipinde bir istisna oluşursa ekrana sadece `toString()` yordamından geri dönen açıklama gönderilecektir. *SeninHatan* istisnasının fırlatıldığı yerlere dikkat ederseniz, ek açıklamaların ne kadar hayati bir önem taşıdığını göreceksiniz. Uygulamanın çıktısı aşağıdaki gibidir.

```

-----
Hata Olustu-1:sonuc eksi
! onemli hata !
2
-----
Hata Olustu-2:SeninHatan: b parametresi sifir geldi
-----
Hata Olustu-2:SeninHatan

```

*SeninHatan* istisna tipinin nasıl meydana geldiğini gönderilen ek açıklama ile daha iyi kavrayabiliyoruz ama son `try-catch` bloğunda yakalanan *SeninHatan* istisna tipinin sebebi açık değildir. Ortada bir istisna vardır ama bu istisnayı nasıl giderebileceğimiz konusunda bilgi yoktur. Bu uygulama karmaşık

olmadığı için kolaylıkla " burada oluşan istisnanın sebebi parametrenin eksi gönderilmesidir " diyebilirsiniz ama çok daha büyük uygulamalarda bu tür çıkarımlar yapmak zannedildiği kadar kolay olmayabilir.

#### 8.1.8. **finally Bloğu**

Bir işlemin her koşulda - istisna olsun ya da olmasın - kesin olarak yapılmasını istiyorsak finally bloğu kullanmalıyız.

##### **Gösterim-8.10:**

```
try {
    // riskli kod
    // bu kod BenimHatam,SeninHatan
    // OnunHatasi, BizimHatamiz
    // tipinde istisnalar fırlatabilir
} catch (BenimHatam bh) {
    // BenimHatam olursa buraya
} catch (SeninHatan sh) {
    // SeninHatan olursa buraya
} catch (OnunHatasi oh) {
    // OnunHatasi olursa buraya
} catch (BizimHatamiz bizh) {
    // BizimHatamiz olursa buraya
} finally {
    // ne olursa olsun çalışacak kod buraya
}
```

Ne olursa olsun çalışmasını istediğiniz kodu `finally` bloğuna yazabilirsiniz. Bir uygulama üzerinde açıklanmaya çalışılsa.

##### **Örnek:** *FinallyOrnek1.java*

```
public class FinallyOrnek1 {
    public static void a(int deger) throws SeninHatan {
        if (deger < 0) {
            throw new SeninHatan();
        }
    }
    public void hesapla() {
        for (int i = -1; i < 1; i++) {
            try {
                System.out.println("a() çağrılıyor");
                a(i);
            } catch (SeninHatan shEx) {
                System.out.println("SeninHatan olustu : " + shEx);
            } finally {
                System.out.println("finally bloğu çalıştırıldı");
            }
        }
    }
    public static void main(String args[]) {
        FinallyOrnek1 fol = new FinallyOrnek1();
        fol.hesapla();
    }
}
```

```
}  
}
```

Bu uygulamada dikkat edilmesi gereken yer `hesapla()` yordamıdır. `for` döngüsü

-1'den 0'a kadar ilerlemektedir, ilerleyen bu değerler `a()` yordamına parametre olarak gönderilmektedir. `a()` yordamının içerisinde ise gelen parametrenin değeri kontrol edilip eğer bu değer 0 dan küçükse *SeninHatan* istisnası fırlatılmaktadır. Buradaki amaç bir istisnalı birde istisnasız koşulu yakalayarak; her koşulda `finally` bloğuna girildiğini ispatlamaktır. Uygulamanın çıktısı aşağıdaki gibidir.

```
a() cagriliyor  
SeninHatan olustu: SeninHatan  
finally blogu calistirildi  
a() cagriliyor  
finally blogu calistirildi
```

Ayrıca `finally` bloğunun daha bir çok faydası bulunur. Örneğin birşey aramak için geceleyin bir odaya girdiğinizde ilk olarak ne yaparsanız? Genelleme yaparak ışığı yakarsanız diyelim. Aynı şekilde odanın içerisinde arama işlemi bittiğinde ve odaya terk edeceğiniz zaman ne yaparsanız? Açık olan ışığı kapatırsınız değil mi? Sonuçta odadan çıkarken ışığın kapatılması gerekir bunun her zaman olması gereken bir davranış olarak kabul edip aşağıdaki uygulamayı inceleyelim.

**Örnek:** *Oda.java*

```
class BeklenmeyenHata1 extends Exception {  
    public BeklenmeyenHata1(String ekAciklama) {  
        super(ekAciklama);  
    }  
}  
  
class BeklenmeyenHata2 extends Exception {  
    public BeklenmeyenHata2(String ekAciklama) {  
        super(ekAciklama);  
    }  
}  
  
public class Oda {  
    public void isiklariKapat() {  
        System.out.println("isiklar kapatildi");  
    }  
    public void isiklariAc() {  
        System.out.println("isiklar acildi");  
    }  
    public void aramaYap() throws BeklenmeyenHata1,  
        BeklenmeyenHata2 {  
        // istisna fırlatabilecek olan govde  
        //...  
    }  
    public void basla() {  
        try {  
            // riskli kod  
            isiklariAc();  
            aramaYap();  
        }  
    }  
}
```

```

        isiklariKapat(); // dikkat
    } catch (BeklenmeyenHata1 bh1) {
        System.out.println("BeklenmeyenHata1
yakalandi");
        isiklariKapat(); // dikkat
    } catch (BeklenmeyenHata2 bh2) {
        System.out.println("isiklar acildi");
        isiklariKapat(); // dikkat
    }
}
}
public static void main(String args[]) {
    Oda o = new Oda();
    o.basla();
}
}

```

Bu örneğimizde `basla()` yordamında gelişen olaylara dikkat edelim. Karanlık bir odada arama yapmak için ilk önce ışıkları açıyoruz daha sonra aramayı gerçekleştiriyoruz ve en sonunda ışıkları kapatıyoruz. Fakat dikkat edin ışıkların kapanmasını garantilemek için üç ayrı yerde `isiklariKapat()` yordamı çağrılmaktadır, peki ama niye?

`try` bloğunun içerisine yerleştirilen `isiklariKapat()` yordamı, eğer herşey yolunda giderse çağrılacaktır. `BeklenmeyenHata1` istisnasının yakalandığı `catch` bloğundaki `isiklariKapat()` yordamı, eğer `BeklenmeyenHata1` istisnası oluşursa ışıkların kapatılması unutulmasın diye yerleştirilmiştir. Aynı şekilde `BeklenmeyenHata2` istisnasının yakalandığı `catch` bloğundaki `isiklariKapat()` yordamı, eğer `BeklenmeyenHata2` istisnası oluşursa ışıkların kapatılmasını garantilemek amacı için buraya yerleştirilmiştir. Bir işi yapabilmek için aynı kodu üç farklı yere yazmak ne kadar verimlidir olabilir? Daha karmaşık bir yapıda belki de ışıkları söndürmek unutulabilir. İşte böyle bir durumda `finally` bloğu hem verimliliği artırmak hem de çalışması istenen kodun çalışmasını garantilemek amacıyla kullanılabilir. Yukarıdaki uygulama örneğimizin doğru versiyonunu tekrardan yazarsak.

### **Örnek:** *Oda2.java*

```

public class Oda2 {
    public void isiklariKapat() {
        System.out.println("isiklar kapatildi");
    }
    public void isiklariAc() {
        System.out.println("isiklar acildi");
    }
    public void aramaYap() throws BeklenmeyenHata1,
    BeklenmeyenHata2 {
        // istisna fırlatabilecek olan govde
    }
    public void basla() {
        try {
            // riskli kod
            isiklariAc();
            aramaYap();
        } catch (BeklenmeyenHata1 bh1) {
            System.out.println("BeklenmeyenHata1
yakalandi");
        } catch (BeklenmeyenHata2 bh2) {
            System.out.println("isiklar acildi");
        } finally {

```



```

        isiklariKapat(); //dikkat
    }
}
public static void main(String args[]) {
    Oda2 o2 = new Oda2();
    o.basla();
}
}

```

Bu uygulama örneğimizde `isiklariKapat()` yordamı sadece `finally` bloğunun içerisine yazılarak her zaman ve her koşulda çalıştırılması garantili hale getirilmiştir. Artık herhangi bir istisna oluşsun veya oluşmasın ışıklar kesin olarak söndürülecektir.

`finally` bloğunun kesin olarak çağrıldığını aşağıdaki uygulamamızdan da görebiliriz.

### **Örnek:** *FinallyOrnek2.java*

```

public class FinallyOrnek2 {

    public static void main(String args[]) {
        try {
            System.out.println("1- try blogu");
            try {
                System.out.println("2- try blogu");
                throw new Exception();
            } finally {
                System.out.println("2- finally blogu");
            }
        } catch (Exception ex) {
            System.out.println("1- catch blogu");
        } finally {
            System.out.println("1- finally blogu");
        }
    }
}

```

Bu örneğimizde dip taraftaki `try` bloğunun içerisinde bir istisna oluşturulmuştur. Dip taraftaki `try` bloğunun `catch` mekanizması olmadığı için bu oluşan istisna dış taraftaki `catch` mekanizması tarafından yakalanacaktır. Fakat bu yakalanma işleminin hemen öncesinde dipte bulunan `finally` bloğunun içerisindeki kodlar çalıştırılacaktır. Uygulamanın çıktısı aşağıdaki gibidir.

```

1- try blogu
2- try blogu
2- finally blogu
1- catch blogu
1- finally blogu

```

#### **8.1.8.1. return ve finally Bloğu**

`finally` bloğu her zaman çalıştırılır. Örneğin bir yordam hiçbir şey döndürmüyorsa (`void`) ama bu yordamın içerisinde yordamı sessizce terk etmek amacı ile `return` ifadesi kullanılmış ise, `finally` bloğu içerisindeki kodlar bu `return` ifadesi devreye girmeden hemen önce çalıştırılır. Uygulama üzerinde gösterilirse.

**Örnek:** *ReturnOrnek.java*

```
public class ReturnOrnek {
    public void calis(int deger) {
        try {
            System.out.println("calis yordamı cagrildi,
gelen deger: "
+ deger);
            if(deger == 0) {
                return; //yordamı sessizce terk et
            }
            System.out.println("-- calis yordamı normal bir
sekilde bitti--");
        } catch (Exception ex) {
            System.out.println("catch blogu icerisinde");
        } finally {
            System.out.println("finally blogu cagrildi");
            System.out.println("-----");
        }
    }
    public static void main(String args[]) {
        ReturnOrnek ro = new ReturnOrnek();
        ro.calis(1);
        ro.calis(0); //dikkat
    }
}
```

calis() yordamına gönderilen parametre eğer sıfırsa, bu yordam çalışmasını sona erdiliyor fakat finally bloğu içerisindeki kodlar bu durumda bile çalıştırılmaktadır. Dikkat edilmesi gereken bir başka nokta ise calis() yordamının bilerek birşey döndürmemesidir -void- olmasıdır. Çünkü eğer calis() yordamı birşey -ör: *String* tipi- döndüreceğini söyleseydi, geri döndürme (*return*) işlemini finally bloğunun içerisinde yapması gerekirdi, aksi takdirde derleme anında (*compile-time*) uyarı alınırdı. Yani bir yordamın içerisinde try - finally blok sistemi tanımlanmış ise try bloğunda return ile bir değer geri döndürülmesine izin verilmez. Uygulamanın çıktısı aşağıdaki gibidir.

```
calis yordamı cagrildi, gelen deger: 1
-- calis yordamı normal bir sekilde bitti--
finally blogu cagrildi
-----
calis yordamı cagrildi, gelen deger: 0
finally blogu cagrildi
-----
```

**8.1.8.2. Dikkat System.exit();**

Eğer *System* sınıfının statik bir yordamı olan *exit()* çağrılırsa finally bloğuna hiç girilmez. *System.exit()* yordamı uygulamanın içerisinde çalıştığı JVM'i (*Java virtual machine*) kapatır. Anlatılanları bir uygulama üzerinde incelersek.

**Örnek:** *SystemExitOrnek.java*

```
public class SystemExitOrnek {
```

```

    public void calis(int deger) {
        try {
            System.out.println("calis yordamı cagrildi,
gelen deger: "
+ deger);
            if(deger == 0) {
                System.exit(-1); // JVM'i kapat
            }
            System.out.println("-- calis yordamı normal bir
sekilde bitti--");
        } catch (Exception ex) {
            System.out.println("catch blogu icerisinde");
        } finally {
            System.out.println("finally blogu cagrildi");
            System.out.println("-----");
        }
    }
    public static void main(String args[]) {
        SystemExitOrnek seo = new SystemExitOrnek();
        seo.calis(1);
        seo.calis(0); // dikkat
    }
}

```

Bu örneğimizin bir öncekine göre tek farkı return yerine System.exit() komutunun yazılmış olmasıdır. System.exit() komutu, uygulamanın içerisinde çalıştığı JVM'i kapatır. exit() yordamına gönderilen eksi bir değer JVM'in anormal bir sonlanmış yapacağını ifade eder. Bu çok ağır bir cezalandırmadır. Normalde uygulamanın bu şekilde sonlandırılması pek tercih edilmemektedir ancak tek başına çalışan (*standalone*) uygulamalarda kullanıcının yanlış parametre girmesi sonucu kullanılabilir.

#### 8.1.9. İstisnanın Tekrardan Fırlatılması

Oluşan bir istisnayı catch bloğunda yakaladıktan sonra tekrardan bir üst kısma fırlatmanız mümkündür. Genel gösterim aşağıdaki gibidir.

##### **Gösterim-8.11:**

```

    try {
        // riskli kod
    } catch (Exception ex){
        System.out.println("istisna yakalandi: " + ex);
        throw ex; // dikkat
    }

```

Oluşan bir istisnayı bir üst kısma fırlatırken istisna nesnesinin içerisindeki bilgiler saklı kalır. Bir uygulama üzerinde incelersek.

##### **Örnek:** *TekrarFirlatimOrnek1.java*

```

    public class TekrarFirlatimOrnek1 {
        public void cokCalis() throws Exception {
            try {

```

```

        throw new Exception("oylesine bir istisna"); //
        istisnanin olusumu
    } catch(Exception ex) {
        System.out.println("cokCalis() istisna
yakalandi: " + ex);
        throw ex; //dikkat
    }
}
public void calis() throws Exception {
    try {
        cokCalis();
    } catch(Exception ex) {
        System.out.println("calis() istisna yakalandi: "
+ ex);
        throw ex; //dikkat
    }
}
public void basla() {
    try {
        calis();
    } catch(Exception ex) {
        ex.printStackTrace(); // bilgi alimi
    }
}
public static void main(String args[]) {
    TekrarFirlatimOrnek1 tfol = new
TekrarFirlatimOrnek1();
    tfol.basla();
}
}

```

Yukarıdaki örneğimizde istisna `cokCalis()` yordamının içerisinde oluşmaktadır. Oluşan bu istisna `catch` mekanizması sayesinde yakalandıktan sonra tekrardan bir üst kısma fırlatılmaktadır. `calis()` yordamının içerisinde de aynı şekilde fırlatılan istisna `catch` mekanizması sayesinde yakalanıp tekrardan bir üst kısma fırlatılmaktadır. `basla()` yordamına kadar gelen istisna nesnesi burada yakalanıp içerisinde saklı bulunan bilgiler `printStackTrace()` yordamıyla gün ışığına çıkarılmaktadır. Uygulamanın çıktısı aşağıdaki gibidir.

```

cokCalis() istisna yakalandi: java.lang.Exception:
                                oylesine bir istisna
calis() istisna yakalandi: java.lang.Exception:
                                oylesine bir istisna
java.lang.Exception: oylesine bir istisna
    at TekrarFirlatimOrnek1.cokCalis(TekrarFirlatimOrnek1.java:7)
    at TekrarFirlatimOrnek1.calis(TekrarFirlatimOrnek1.java:17)
    at TekrarFirlatimOrnek1.basla(TekrarFirlatimOrnek1.java:29)
    at TekrarFirlatimOrnek1.main(TekrarFirlatimOrnek1.java:38)

```

Dikkat edilirse oluşan istisnaya ait bilgiler `basla()` yordamının içerisinde ekrana basılmasına karşın, orijinalliğini hiç kaybetmedi. Orijinallikten kasıt edilen istisnanın gerçekten nerede oluştuğu bilgisidir. Oluşan bir istisnayı yakalayıp yeniden fırlatmadan evvel, onun içerisindeki bilgilere müdahale etmeniz mümkündür. Şöyle ki...

**Örnek:** *TekrarFirlatimOrnek2.java*

```

public class TekrarFirlatimOrnek2 {
    public void cokCalis() throws Exception {
        try {
            throw new Exception("oylesine bir istisna");
        } catch(Exception ex) {
            System.out.println("cokCalis() istisna
yakalandi: " + ex);
            throw ex; //dikkat
        }
    }
    public void calis() throws Throwable {
        try {
            cokCalis();
        } catch(Exception ex) {
            System.out.println("calis() istisna yakalandi: "
+ ex);
            throw ex.fillInStackTrace(); //dikkat
        }
    }
    public void basla() {
        try {
            calis();
        } catch(Throwable th) {
            th.printStackTrace(); //döküm
        }
    }
    public static void main(String args[]) {
        TekrarFirlatimOrnek2 tfo2 = new
TekrarFirlatimOrnek2();
        tfo2.basla();
    }
}

```

Bu örneğimizde istisnanın orijinal oluşma yeri `cokCalis()` yordamıdır ama `calis()` yordamı içerisinde, istisna nesnesinin içindeki bilgilere `fillInStackTrace()` müdahale edilip değiştirmektedir. Uygulamanın çıktısı aşağıdaki gibidir.

```

cokCalis() istisna yakalandi: java.lang.Exception:
                                oylesine bir istisna
calis() istisna yakalandi: java.lang.Exception:
                                oylesine bir istisna
java.lang.Exception: oylesine bir istisna
    at TekrarFirlatimOrnek2.calis(TekrarFirlatimOrnek2.java:20)
    at TekrarFirlatimOrnek2.basla(TekrarFirlatimOrnek2.java:28)
    at TekrarFirlatimOrnek2.main(TekrarFirlatimOrnek2.java:36)

```

Artık istisnanın oluşma yeri olarak 20. satırı yani `fillInStackTrace()` yordamının devreye girdiği yer gösterilmektedir. Böylece oluşan istisnanın içerisindeki bilgilere müdahale etmiş bulunmaktayız. `calis()` yordamında niye *Throwable* tipinde bir istisna fırlatıldığına gelince, bunun sebebi `fillInStackTrace()` yordamının *Throwable* tipinde bir istisna nesnesi geri döndürmesidir. Bu sebepten dolayı `basla()` yordamının içerisindeki `catch` bloğunda *Exception* istisna tipi yerine *Throwable* tipi belirtilmiştir. Eğer bu `catch` bloğunda *Exception* tipi belirtilseydi derleme anında (compile-time) Throwable yakalanmalı diye hata alınırdı. Şekil-8.5'e dikkat ederseniz *Throwable* istisna

tipi en üste bulunmaktadır. Bunun anlamı eğer bir *Throwable* tipinde istisna fırlatılmış ise bunu kesin olarak `catch` bloğunun içerisinde *Throwable* tipi belirterek yakalayabileceğimizeyiz.

**Örnek:** Rutbe.java

```
public class Rutbe {
    public static void main(String args[]) {
        try {
            throw new Throwable();
        } catch ( Exception ex ) {
            System.out.println(" istisna yakalandi: " + ex);
        }
    }
}
```

Yukarıdaki örnek derlenmeye (*compile*) çalışılırsa; aşağıdaki hata mesajı ile karşılaşılır:

```
Rutbe.java:7: unreported exception java.lang.Throwable; must be
caught or declared to be thrown
        throw new Throwable();
            ^
1 error
```

Bunun anlamı, *Throwable* tipindeki fırlatılmış bir istisna nesnesini `catch` bloğunun içerisinde *Exception* tipi belirtilerek yakalanamayacağıdır.

#### 8.1.10. ***printStackTrace()* ve Hata Mesajlarının Kısaltılması**

Java 1.4 ile beraber gelen bir başka özellik ise *Throwable* sınıfının yapılandırıcısına bir başka istisna tipini parametre olarak gönderebiliyor olmamızdır. Bu özellikten daha evvel bahsetmiştik, esas ilginç olan bu özelliğin fazla kullanılmasıyla aynı hata mesajlarının tekrarlamasıdır. Java tekrarlayan bu hata mesajları için bir kısaltma kullanır.

**Örnek:** Kisaltma.java

```
class YuksekSeviyeliIstisna extends Exception {
    YuksekSeviyeliIstisna(Throwable cause) {
        super(cause);
    }
}

class OrtaSeviyeliIstisna extends Exception {
    OrtaSeviyeliIstisna(Throwable cause) {
        super(cause);
    }
}

class DusukSeviyeliIstisna extends Exception {
}

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

```

        a();
    } catch(YuksekJSeviyeliIstisna e) {
        e.printStackTrace();
    }
}
static void a() throws YuksekJSeviyeliIstisna {
    try {
        b();
    } catch(OrtaSeviyeliIstisna e) {
        throw new YuksekJSeviyeliIstisna(e);
    }
}
static void b() throws OrtaSeviyeliIstisna {
    c();
}
static void c() throws OrtaSeviyeliIstisna {
    try {
        d();
    } catch(DusukSeviyeliIstisna e) {
        throw new OrtaSeviyeliIstisna(e);
    }
}
static void d() throws DusukSeviyeliIstisna {
    e();
}
static void e() throws DusukSeviyeliIstisna {
    throw new DusukSeviyeliIstisna(); //baslangic
}
}

```

Yukarıdaki örneğimizde üç adet istisna tipi bulunmaktadır. e() yordamının içerisinde başlayan istisnalar zinciri main() yordamının içerisinde son bulmaktadır. Buradaki olay oluşan bir istisnayı diğerine ekleyerek aynı tip hata mesajları elde etmektir. Uygulamanın çıktısı aşağıdaki gibidir.

```

YuksekJSeviyeliIstisna: OrtaSeviyeliIstisna: DusukSeviyeliIstisna
  at Kisaltma.a(Kisaltma.java:29)
  at Kisaltma.main(Kisaltma.java:20)
Caused by: OrtaSeviyeliIstisna: DusukSeviyeliIstisna
  at Kisaltma.c(Kisaltma.java:39)
  at Kisaltma.b(Kisaltma.java:33)
  at Kisaltma.a(Kisaltma.java:27)
  ... 1 more
Caused by: DusukSeviyeliIstisna
  at Kisaltma.e(Kisaltma.java:46)
  at Kisaltma.d(Kisaltma.java:43)
  at Kisaltma.c(Kisaltma.java:37)
  ... 3 more

```

Uygulamanın çıktısından da anlaşılacağı üzere, tekrar eden kısmın kaç kere tekrar ettiği bilgisi de verilmektedir. Mesela:

```

  at Kisaltma.c(Kisaltma.java:39)
  at Kisaltma.b(Kisaltma.java:33)

```

```
at Kisaltma.a(Kisaltma.java:27)
```

Yukarıdaki kısım 1 kere tekrar etmiştir

```
at Kisaltma.e(Kisaltma.java:46)
at Kisaltma.d(Kisaltma.java:43)
at Kisaltma.c(Kisaltma.java:37)
```

Bu kısım ise 3 kere tekrar etmiştir.

### 8.1.11. İlginç Gelişme

Oluşan bir istisna her zaman fırlatılmayabilir. Aşağıdaki uygulamamızı inceleyelim

**Örnek:** *FirlatimOrnek1.java*

```
public class FirlatimOrnek1 {

    public void basla(int a, int b) throws Exception {
        int sonuc = 0;
        try {
            sonuc = a / b;
        } catch(Exception ex) {
            System.out.println("basla() istisna yakalandi");
            throw ex;
        } finally {
            System.out.println("sonuc: " + sonuc);
        }
    }

    public static void main(String args[]) {
        try {
            FirlatimOrnek1 fol = new FirlatimOrnek1();
            fol.basla(1,0);
        } catch(Exception ex) {
            System.out.println("main() istisna yakalandi");
        }
    }
}
```

Yukarıdaki örneğimizde akışın nasıl olmasını bekleriz ? İlkel (*primitive*) `int` tipinde bir sayının sıfıra bölünmesi sonucu *ArithmeticException* tipinde bir istisna oluşur aynı bizim bu örneğimizde olduğu gibi. Daha sonra ekrana `finally` bloğunun içerisinde tanımlanmış ifade yazılır ve en son olarak istisna nesnesi bir üst kısma fırlatılır. Herşey beklendiği gibi gitmekte! Uygulamamızın çıktısı aşağıdaki gibidir.

```
basla() istisna yakalandi
sonuc: 0
main() istisna yakalandi
```



Önce `basla()` yordamının içerisinde yakalanan istisna, `finally` bloğunun çalıştırılmasından sonra bir üst kısma fırlatılabilmektedir. Fırlatılan bu istisna `main()` yordamı içerisinde yakalanmaktadır.

Peki ya `basla()` yordamı bir değer döndürseydi olaylar nasıl değişirdi?

**Örnek:** *FirlatimOrnek2.java*

```
public class FirlatimOrnek2 {
    public int basla(int a, int b) throws Exception {
        int sonuc = 0;
        try {
            sonuc = a / b;
        } catch (Exception ex) {
            System.out.println("basla() istisna yakalandi");
            throw ex;
        } finally {
            System.out.println("sonuc: " + sonuc);
            return sonuc; // dikkat
        }
    }
    public static void main(String args[]) {
        try {
            FirlatimOrnek2 fo2 = new FirlatimOrnek2();
            fo2.basla(1,0);
        } catch (Exception ex) {
            System.out.println("main() istisna yakalandi");
        }
    }
}
```

Uygulamamızın çıktısı nasıl olacaktır? Bir önceki uygulama ile aynı mı?

```
basla() istisna yakalandi
sonuc: 0
```

Oluşan istisna, `basla()` yordamında yakalanmıştır ama daha sonra ortaldan kaybolmuştur. Aslında bu olay hata gibi algılanabilir ve haklı bir algılamadır. Fakat olaylara birde Java tarafından bakarsak anlayış gösterilebilir. Bir yordamın bir seferde sadece tek bir şey döndürme hakkı vardır. Ya bir değer döndürebilir veya bir istisna fırlatabilir, sonuçta fırlatılan bir istisna da değer niteliği taşır. Bu uygulamamızda `basla()` yordamı `int` tipinde değer döndüreceğini söylediği ve `finally` bloğu kullanıldığı için, oluşan bir istisnanın tekrardan fırlatılması olanaksızdır. Bu işin bir çözümü var mı? Düşünelim... Bir yordam bir değer döndürse bile eğer bir istisna oluşursa bu oluşan istisnayı öncelikli olarak nasıl fırlatabilir? Böyle bir ikilem ile er ya da geç karşı karşıya kalınacaktır. Aşağıdaki gibi bir çözüm iş görecektir.

**Örnek:** *FirlatimOrnek3.java*

```
public class FirlatimOrnek3 {
    public int basla(int a, int b) throws Exception {
        int sonuc = 0;
        Exception globalEx = null;
        try {
```

```

        sonuc = a / b;
    } catch(Exception ex) {
        System.out.println("basla() istisna yakalandi");
        globalEx = ex; // aktarim
    } finally {
        System.out.println("sonuc: "+ sonuc);
        if(globalEx != null) { // eger istisna olusmus ise
            throw globalEx; // tekrardan firlatim
        }
        return sonuc; // degeri geri dondur
    }
}
public static void main(String args[]) {
    try {
        FirlatimOrnek3 fo3 = new FirlatimOrnek3();
        fo3.basla(1,1);
        fo3.basla(1,0);
    } catch(Exception ex) {
        System.out.println("main() istisna yakalandi");
    }
}
}

```

Yukarıdaki örneğimizde, eğer bir istisna oluşmuş ise *Exception* tipinde tanımlanan `globalEx` alanına, `catch` bloğu içerisinde değer aktarılmaktadır. `finally` bloğunun içerisinde `globalEx` alanına bir istisna nesnesinin bağlı olup olmadığı kontrol edilmektedir. Eğer `globalEx`, `null` değerinden farklıysa, bu `catch` bloğunda bir istisna nesnesine bağlandığı anlamına gelir yani bir istisnanın oluştuğunu ifade eder. Eğer `globalEx` `null` değerine eşitse sorun yok demektir. Böylece istisna oluşmuş ise `finally` bloğunda istisna fırlatılır, değilse de yordam normal dönmesi gereken değeri geri döndürür. Uygulamamızın çıktısı aşağıdaki gibidir.

```

sonuc: 1
basla() istisna yakalandi
sonuc: 0
main() istisna yakalandi

```

### 8.1.12. İptal Etme (Override) ve İstisnalar

İptal etme (*override*) konusunu 5. bölümde incelemiştik. Bir sınıftan türetilen bir alt sınıfın içerisinde, üst (ana) sınıfa ait bir yordamı iptal edebilmesi için bir çok şart aranmaktaydı, bunlar sırasıyla, iptal eden yordamın, iptal edilen yordam ile aynı parametrelere, aynı isme ve üst sınıfa ait yordamın erişim belirleyicisinden daha erişilebilir veya aynı erişim belirleyicisine sahip olması gerekirdi. Buraya kadar anlattıklarımızda hemfikirsek esas soruyu sorabiliriz; İptal edilme (*override*) ile istisnalar arasında bir bağlantı olabilir mi? Bu konu için bir başlık ayrıldığına göre herhalde bir bağlantı var ama nasıl? Bir uygulama üzerinde incelersek.

#### Örnek: AB.java

```

import java.io.*;
class A {
    public void basla() throws FileNotFoundException,
    EOFException {
        //...
    }
}

```

```

    }
}
public class AB extends A {
    public void basla() throws IOException {
        //...
    }
}

```

*AB.java* uygulamasını derlemeye (*compile*) çalıştığımız zaman aşağıdaki hata mesajını alırız.

```

AB.java:12: basla() in AB cannot override basla() in A;
overridden method does not throw java.io.IOException
    public void basla() throws IOException {
                ^
1 error

```

Bu hata mesajının anlamı nedir? *AB* sınıfının içerisindeki *basla()* yordamının, *A* sınıfının içerisindeki *basla()* yordamını iptal edemediği çok açıktır, bunun sebebi erişim belirleyiciler olabilir mi? Hayır olamaz çünkü hem iptal eden hem de edilen yordam aynı erişim belirleyicisine sahip (public erişim belirleyicisine). Hımm peki sorun nerede? Sorun istisna tiplerinde. *A* sınıfına ait *basla()* yordamı iki adet istisna nesnesi fırlatıyor (*FileNotFoundException* ve *EOFException*) ama bunu iptal etmeye çalışan *AB* sınıfına ait *basla()* yordamı sadece bir tane istisna nesnesi fırlatıyor (*IOException*), sorun bu olabilir mi?

İptal edememe sorununu anlamak için Şekil-8.5.'deki yapıyı incelemek gerekir. Bu şeklimizden görüleceği üzere *FileNotFoundException* ve *EOFException* istisna tipleri, *IOException* istisna tipinden türetilmişlerdir. Kötü haberi hemen verelim, iptal ederken (*override*) artık yeni bir kuralımız daha oldu, şöyle ki: iptal edilen yordamının (*A* sınıfının içerisindeki *basla()* yordamı) fırlatacağı istisna tipi, iptal eden yordamın (*AB* sınıfı içerisindeki *basla()* yordamı) fırlatacağı istisna tiplerini kapsamalıdır. Aşağıdaki uygulamamız bu kuralı doğru bir şekilde yerine getirmektedir.

**Örnek:** *CD.java*

```

import java.io.*;
class C {
    public void basla() throws IOException {
        //...
    }
}
public class CD extends C {
    public void basla() throws FileNotFoundException,
    EOFException {
        //...
    }
}

```

İşte doğru bir iptal etme (*override*) örneği. *C* sınıfının *basla()* yordamı (iptal edilen) sadece bir adet istisna fırlatmaktadır (*IOException*) fakat *CD* sınıfının *basla()* yordamı (iptal eden) iki adet istisna fırlatmaktadır (*FileNotFoundException* ve *EOFException*). Buradan çıkarılacak sonuç doğru bir iptal etme işlemi için fırlatılan istisna sayısı değil, tiplerinin önemli olduğudur. Şekil-8.5.'e bir kez daha bakılırsa, *IOException* istisna tipinin, *FileNotFoundException* ve *EOFException* istisna tiplerini kapsadığını görürsünüz; yani, *FileNotFoundException* ve *EOFException* tipinde istisna tipleri fırlatılırsa bu istisnaları

*IOException* tipi ile `catch` bloğunda yakalanabilir ama bunun tam tersi olanaksızdır. Daha ilginç bir örnek verelim.

**Örnek:** *EF.java*

```
import java.io.*;
class E {
    public void basla() throws IOException {
        //...
    }
}
public class EF extends E {
    public void basla() {
        //...
    }
}
```

İptal edilen yordam *IOException* tipinde bir istisna fırlatmasına karşın, iptal eden yordamın hiç bir istisna fırlatmama lüksü vardır. İptal eden yordamın hiç bir istisna fırlatmaması bir soruna yol açmaz. Niye iptal edilen yordamın daha kapsamlı bir istisna fırlatması gerekir? Bu sorunun cevabını daha kapsamlı bir örnek üzerinde inceleyelim.

**Örnek:** *Sekreter.java*

```
import java.io.*;
class Calisan {
    public void calis(int deger) throws IOException {
        System.out.println("Calisan calisiyor "+ deger);
    }
}

public class Sekreter extends Calisan {

    public void calis(int deger) throws
    FileNotFoundException,
    EOFException {

        System.out.println("Calisan calisiyor "+ deger);
        if(deger == 0) {
            throw new FileNotFoundException("Dosyayi
            bulamadim");
        } else if(deger == 1) {
            throw new EOFException("Dosyanin sonuna
            geldim");
        }
    }

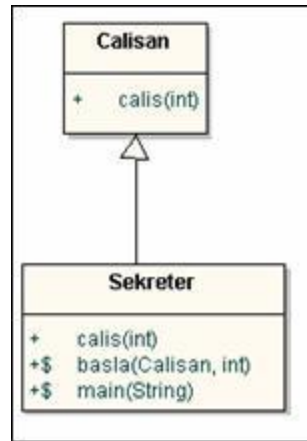
    public static void basla(Calisan c, int deger) {
        try {
            c.calis(deger);
        } catch (IOException ex) {
            System.out.println("Istisna olustu: "+ ex);
        }
    }
}
```

```

public static void main(String args[]) {
    Sekreter s1 = new Sekreter();
    Sekreter s2 = new Sekreter();
    Sekreter s3 = new Sekreter();
    basla(s1,2); // sorunsuz
    basla(s1,1); // EOFException
    basla(s3,0); // FileNotFoundException
}
}

```

Bu örneğimizde *Sekreter* sınıfı *Calisan* sınıfından türemiştir. Ayrıca *Sekreter* sınıfının *calis()* yordamı, kendisinin ana sınıfı olan *Calisan* sınıfının *calis()* yordamını iptal etmiştir (*override*). *Calisan* sınıfına ait *calis()* yordamının fırlatacağı istisna daha kapsamlı olmasındaki sebep yukarı çevrimlerde (upcasting) sorun yaşanmaması içindir. Şimdi *basla()* yordamına dikkat edelim. Bu yordam *Calisan* tipinde parametre kabul etmektedir; yani, *main()* yordamının içerisinde oluşturulan *Sekreter* nesneleri *basla()* yordamına parametre olarak gönderilebilir çünkü arada kalıtım (*inheritance*) ilişkisi vardır.



Şekil-8.6. İptal etme (*override*) ve İstisnalar

Fakat bu gönderilme esnasında bir daralma (yukarı çevirim) söz konusudur, *Sekreter* nesneleri *heap* alanında dururken onlara bağlı olan referansların tipi *Calisan* tipindedir. Burada bir ayrıntı saklıdır, bu ayrıntı şöyledir: *c.calis()* komutu çağrıldığı zaman *Calisan* sınıfının *basla()* yordamına ait **etiketin** altında *Sekreter* sınıfında tanımlı olan *basla()* yordamına ait kodlar çalıştırılır. Bu uygulamamızda kullanılan etiket aşağıdadır.

#### Gösterim-8.12:

```

public void calis(int deger) throws IOException { //
etiket

```

Çalıştırılacak gövde aşağıdadır.

#### Gösterim-8.13:

```

System.out.println("Calisan calisiyor "+ deger);
if(deger == 0) {
    throw new FileNotFoundException("Dosyayı
bulamadım");

```

```
} else if(deger == 1) {  
    throw new EOFException("Dosyanin sonuna geldim");  
}
```

Bu yüzden iptal edilen yordamın olabilecek en kapsamlı istisnayı fırlatması gerekir -ki yukarı çevirim işlemlerinde (*upcasting*) iptal eden yordamların gövdelerinden fırlatılabilecek olan istisnalara karşı aciz kalınmasın. Olabilecek en kapsamlı istisna tipi bu uygulama örneğimizde *IOException* istisna tipindedir çünkü bu istisna tipi hem *FileNotFoundException* istisna tipini hem de *EOFException* kapsamaktadır (bkz:Şekil-85.).

Uygulamamızın çıktısı aşağıdaki gibidir.

```
Calisan calisiyor 2  
Calisan calisiyor 1  
Istisna olustu: java.io.EOFException: Dosyanin sonuna geldim  
Calisan calisiyor 0  
Istisna olustu: java.io.FileNotFoundException: Dosyayi bulamadim
```

### 8.1.13. İstisnaların Sıralanması

Bir istisna *catch* bloğunda veya *catch* bloklarında yakalanırken, istisnaların hiyererşik yapılarına dikkat edilmelidir.

**Örnek:** *IstisnaSiralamasi.java*

```
class IstisnaBir extends Exception {  
}  
  
class IstisnaIki extends IstisnaBir {  
}  
  
public class IstisnaSiralamasi {  
    public static void main(String args[]) {  
        try {  
            throw new IstisnaIki(); // dikkat  
        } catch (IstisnaIki is2) {  
            System.out.println("istisna yakalandi  
IstisnaIki: " );  
        } catch (IstisnaBir is1) {  
            System.out.println("istisna yakalandi  
IstisnaBir: " );  
        }  
    }  
}
```

Bu örneğimizde kendimize özgü iki adet istisna tipi vardır. *IstisnaIki* sınıfı, *IstisnaBir* sınıfından türetilmiştir. Bunun anlamı eğer *IstisnaIki* tipinde bir istisna fırlatılırsa bunun *IstisnaBir* tipiyle *catch* bloğunda yakalanabileceğidir. Yukarıdaki örneğimizde *IstisnaIki* tipinde bir istisna fırlatılmaktadır, fırlatılan bu istisna ilk *catch* bloğunda yakalanmaktadır. Bir istisna bir kere yakalandı mı artık diğer *catch* bloklarının bu istisnayı bir daha tekrardan yakalama şansları yoktur (tekrardan fırlatılmadıkları varsayılarak). Yani bir istisna bir kerede ancak bir *catch* bloğu tarafından yakalanabilir. Uygulamanın çıktısı aşağıdaki gibidir.

```
istisna yakalandi IstisnaIki:
```

Az önce bahsettiğimiz gibi eğer *IstisnaIki* tipinde bir istisna fırlatılırsa bu *IstisnaBir* tipiyle catch bloğunda yakalanabilir.

**Örnek:** *IstisnaSiralamasi2.java*

```
public class IstisnaSiralamasi2 {  
    public static void main(String args[]) {  
        try {  
            throw new IstisnaIki(); // dikkat  
        } catch (IstisnaBir is1) {  
            System.out.println("istisna yakalandi  
IstisnaBir: " );  
        }  
    }  
}
```

Yukarıdaki örneğimiz doğrudur. Uygulamamızın çıktısı aşağıdaki gibidir.

```
istisna yakalandi IstisnaBir:
```

Eğer *IstisnaIki* tipinde bir istisna fırlatılırsa ve bu ilk etapda *IstisnaBir* tipiyle catch bloğunda ve ikinci etapda ise *IstisnaIki* tipiyle catch bloğunda yakalanmaya çalışırsa ilginç bir olay meydana gelir.

**Örnek:** *IstisnaSiralamasi3.java*

```
public class IstisnaSiralamasi3 {  
    public static void main(String args[]) {  
        try {  
            throw new IstisnaIki(); // dikkat  
        } catch (IstisnaBir is1) {  
            System.out.println("istisna yakalandi  
IstisnaBir: " );  
        } catch (IstisnaIki is2) {  
            System.out.println("istisna yakalandi  
IstisnaIki: " );  
        }  
    }  
}
```

Yukarıdaki örneğimizi derlemeye (*compile*) çalıştığımız zaman aşağıdaki hata mesajını alırız.

```
IstisnaSiralamasi3.java:9: exception IstisnaIki has already been  
caught  
    } catch (IstisnaIki is2) {  
    ^  
1 error
```

Bu hata mesajının anlamı, ikinci `catch` bloğunun boşu boşuna konulduğu yönündedir çünkü zaten ilk `catch` bloğu, bu istisnayı yakalayabilir, bu yüzden ikinci `catch` bloğuna

**Bu dökümanın her hakkı saklıdır.**

© 2004

## **Java 24 Bölüm 14: 7 Yıl Sonra Applet**

Yıllar önce ben üniversitede öğrenciyken (sanırım 1996 veya 1997 yılıydı) değerli bir sınıf arkadaşımı ziyarete gitmiştim. Kendisi bilgisayar teknolojilerine son derece ilgili birisiydi ve bu anlamda ortak pek çok yönümüz vardı. O yıllarda ikimizde, özellikle görsel programlamaya yönelik yazılım geliştirme ortamlarına ilgi duyuyorduk. O günkü ziyaretimde, dostumun elinde o güne kadar gördüğüm en kalın kitap duruyordu. Sanırım o zamanlar gözüme çok büyük gözüküyordu. Öyleki o güne dek hiç 900 sayfalık bir bilgisayar kitabı görmemişim. Oysaki şimdi o 900 sayfalık kitapları arar oldum. En son çalıştığım bilgisayar kitabı 1500 sayfaya yakın olunca, insan ister istemez özlüyor.

Neyse sözün kısası, arkadaşımın elinde tuttuğu kitap, ingilizce bir bilgisayar kitabıydı ve Java diye bir şeyden bahsediyordu. Oha falan oldum yani der gibi arkadaşımın gözlerine baktım. Çünkü ilk aklıma gelen StarWars serisindeki Java olmuştu. Hemen ne demek istediğimi anladı ve anlatmaya başladı. Java'nın yeni bir programlama dili olduğunu, C++'ın syntax'ına çok benzer yer yer aynı yazımları kullandığını ancak işin içinde platform bağımsızlığın yer aldığını söyledi. O zamanlar bende pek çok kişi gibi platform bağımsız kısmına geldiğinde, hafif bir tebessümle hadi canım demiştim. Çok geçmeden bana kitabın ilk kaynak uygulamsından geliştirdiği kodu gösterdi. Burada komik bir çizgi karakter (kırmızı burunlu) bir internet explorer penceresinde bir oraya bir oraya taklalar atıyordu. Bu nedir diye sorduğumda bana bunun bir Applet olduğunu ve browser'ın üzerinde dinamik olarak yerel makinede çalıştığını söyledi. O zamanlar elbetteki browser üzerinde çalışan dinamik uygulamalara hiç aşına değildim.

Java dilini öğrenmeye başladığımda, günün birinde bu değerli arkadaşımı hatırlayacağımı ve kulaklarını çınlatacağımı biliyordum. Artık o zamanlar söyledikleri şimdi kulağıma daha teknik olarak geliyor. Eeee ne demişler "geç olsun da güç olmasın". İşe appletlerin ne olduğunu kavramak ile başlamam gerekiyordu. Daha sonraki kahve molalarımdaya ise appletleri kullanıcı ile dinamik olarak etkileşime sokmaya çalışacaktım. Ama önce teknik bilgi ve basit kodlara ihtiyacım vardı. Tabiki appletin basit bir tanımından sonra.

Bir applet, istemci uygulamada yada başka bir deyişle yerel makinede, Java Virtual Machine'e sahip herhangi bir tarayıcıda (browser) derlenerek çalıştırılan dinamik bir java programcısından başka bir şey değildir. Applet'leri normal java programları yazar gibi java dosyaları olarak yazar ve javac aracı ile class olarak byte-code'a çeviririz. Tek fark, bu program parçalarının, tarayıcıdan talep edilmeleri halinde, tarayıcının sahip olduğu JVM sayesinde derlenerek bu tarayıcının yer aldığı yerel makinede dinamik olarak çalışacak olmalarıdır. Dolayısıyla normal java byte kodları gibi, bu kodlarda çalıştırıldıklarında derlenirler. Ancak çalışma sistemleri, içerdikleri olay yapıları konsol veya görsel arabirime sahip java uygulamalarından biraz daha farklıdır. Herşeyden önce, tarayıcıda çalıştıkları için, belirli bir alan içerisinde çizilebilirler yada kullanılabilirler. Bununla birlikte dinamik çalışmaya müsait oldukları için aşağıdaki olayları gerçekleştirmelerine, yerel makinelerin güvenliği açısından izin



verilmez.

| Applet'lere Özgü Kısıtlamalar  |
|--|
| Yerel makineden (çalıştıkları makine) dosya kopyalayamazlar.                         |
| Dosya silemezler.  |
| Dosya açamazlar veya oluşturamazlar.   |
| İndirildikleri sunucudan başka bir sunucu ile herhangi bir ağ bağlantısı kuramazlar. |
| İndirildikleri bilgisyarda başka programları çalıştıramazlar.                        |
| Dosya sistemine erişemezler veya okuyamazlar.  |

Applet'lerin çalışması ile ilgili olarak en dikkat çekici nokta, çağırıldıkları sunucudan istemci bilgisayarın tarayıcısına indirilmeleridir. Nitekim, bu işlemin gerçekleştirilmesi için, applet'e ait class dosyasının bir şekilde html kodu içerisine gömülmesi gerekecektir. Bunun nasıl yapıldığını görmek için öncelikle bir applet geliştirmek gerektiği kanısındayım. Ne kadar basit olursa olsun en azından nasıl çalıştığını görmem gerekiyor. Kaynaklarımı inceledikten sonra, aşağıdaki gibi bir örnek java dosyasını oluşturdum.

```
import java.awt.*;
import java.applet.Applet;

public class IlkApplet extends Applet
{
    public void Paint(Graphics g)
    {
        g.drawString("Yihuuu",50,50);
    }
}
```

Burada oluşturduğum java dosyasını javac ile derlediğimde herhangi bir sorun ile karşılaşmadım. Peki ama kodum ne yapıyordu? Herşeyden önce ilk dikkatimi çeken, kullanılmak üzere eklediğim awt ve applet paketleriydi. Awt paketini ileride detaylı incelemeyi düşünüyordum zaten. Ancak yinede ön bilgiye ihtiyacım vardı. Awt paketi içerisinde, java ile kullanabileceğimiz görsel arayüzlere ait nesneler için bir çok sınıf bulunuyordu. Applet'lerde sonuç itibariyle, tarayıcı penceresinde çalışacaklarından, kullanıcılar ile görsel iletişim sağlamamıza yarayacak buton, textbox gibi nesneler içerebilirdi. İşte bu amaçla awt paketi vardı. Gerçi kullandığımız bir nesne yok gibi gözükabilir ancak, Graphics sınıfı awt paketi içerisinde yer alan ve appletin çalıştığı alan içerisine bir şeyler çizmek için (örnekte olduğu gibi yazı yazmak için mesela) kullanılan bir sınıftır.

Diğer önemli bir kavramda, sınıfın Applet sınıfından türetilmiş olmasıydı. Bu, yazılan java sınıfının bir applet olarak değerlendirileceğini belirtmekteydi. Dolayısıyla applet sınıfından bir takım özellikleri kalıtsal olarak alacağımız kesindi. Gelelim, Paint metoduna. İşte işin en can alıcı noktası burasıydı. Bu metod, tarayıcı penceresinde,

appletin çalıştığı alana birşeyler çizmek için kullanılıyordu. Daha doğrusu applet, sınırları ile birlikte tarayıcı penceresinde çizilmeye başladığında çalışıyordu. Artık, değerli dostumun tarihi java kitabındaki kırmızı burunlu kahramanın nasıl taklalar attığını daha iyi anlamaya başlamıştım. O zamanlar çizgi filim gibi gelmişti. Ancak şimdi gerçeğin ta kendisi karşımdaydı. Peki şimdi ne olacak? Bir şekilde yazdığım appleti test etmem gerekiyor. İlk aklıma gelen ancak denemek istemediğim şeyi deneyerek işe başladım. Şöyleki,

```
C:\ Command Prompt

E:\JavaSamples\Appletler>javac IlkApplet.java

E:\JavaSamples\Appletler>java IlkApplet
Exception in thread "main" java.lang.NoSuchMethodError: main

E:\JavaSamples\Appletler>
```

Böyle birşeyin başıma geleceği kesindi diyebilirim. Elbetteki appletin çalışma sistemine bakıldığında farklı şekilde uygulanmaları gerekirdi. Her şeyden önce, bu applet bir tarayıcıya indirilecek ve oradaki JVM tarafından derlenecekti. Bunu test etmenin iki yolu vardı. Birincisi bir applet tagı ile bu sınıfı bir html sayfasına koymak yada Applet Viewer aracını kullanmaktı. İlk önce applet tagını aşağıdaki gibi denedim. Bunun için applet sınıfım ile aynı klasörde olan bir html sayfası hazırladım.

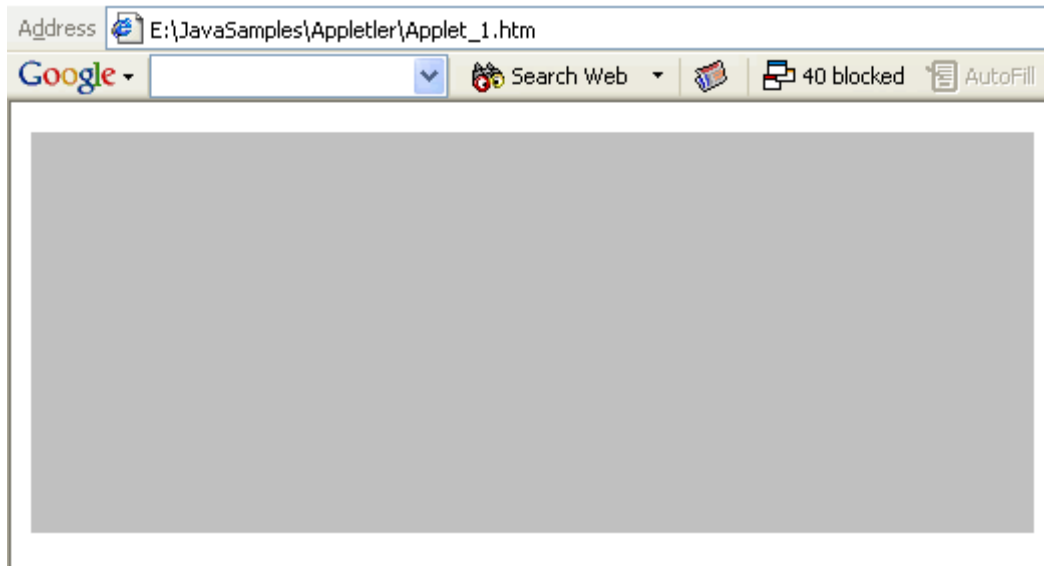
```
<html>

<head>
<meta http-equiv="Content-Language" content="tr">
<meta http-equiv="Content-Type" content="text/html; charset=windows-1252">
<title>New Page 1</title>
</head>

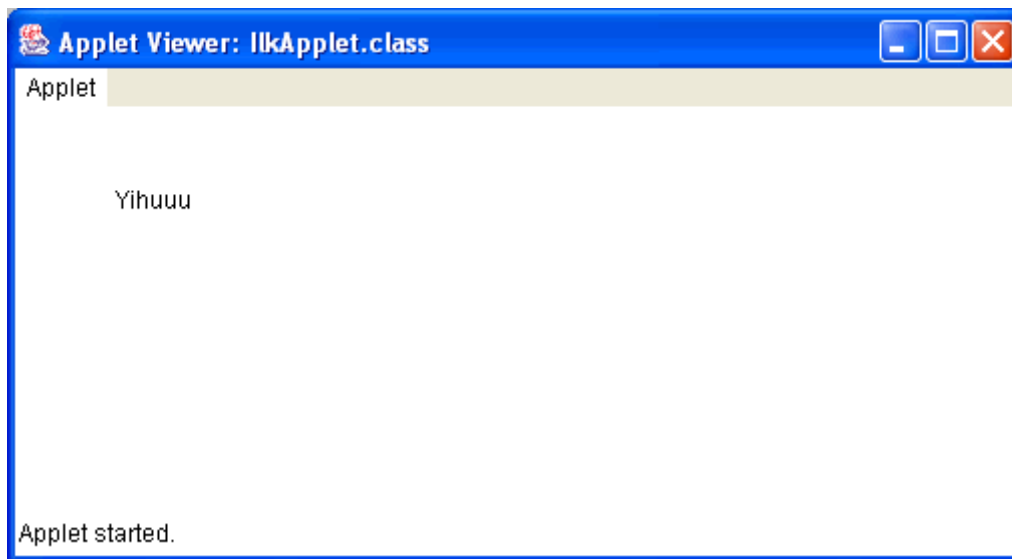
<body>
<APPLET CODE="IlkApplet.class" WIDTH="500" HEIGHT="200">
</APPLET>
</body>

</html>
```

Applet tagı içinde en önemli kısım CODE anahtar kelimesinin olduğu kısım idi. Burada, belirtilen 500 piksel genişlik ve 200 piksel yüksekliğindeki alanda hangi applet sınıfının çalıştırılacağını belirtiyorduk. Şimdi oluşturduğum bu html sayfasını tarayıcıda açtım. Ancak hiç beklemediğim aşağıdaki sonucu elde ettim.



500 piksel'e 200 piksellik bir alan açılmıştı. Ancak yazmak istediğim yazıyı görememiştim. Bunun tek nedeni olabilirdi. JVM, ya sınıf dosyasını derlememişti yada appet sınıfını tarayıcı penceresine indirememiştim. Tabi bir diğer ihtimalde tarayıcının özellikle Microsoft Internet Explorer olduğu için, JVM desteğinin kaldırılmış olabileceğiydi. Aklıma ilk gelen en güncel java plug-in indirmek oldu. Ancak daha öncesinde en azından yazdığım appletin doğru olup olmadığından emin olmalıydım. Neyseki, java'nın appletviewer aracı yardımına yetiştii. Komut satırında aşağıdaki satır ile appletimin çalışmasının sonucunu gördüm. Applet Viewer programı, yazılmış olan appletlerin tarayıcı penceresine ihtiyaç duyulmadan çalıştırılabilmelerini sağlıyordu.

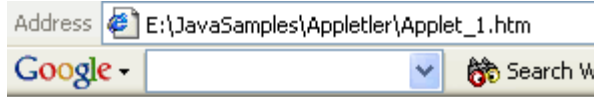


Evet appletim çalışmıştı. Applet viewer bir applet'i test etmek için ideal bir yoldu.

Ama kafam halen daha internet explorer tarayıcısında neden çalışmadığındaydı. Hemen internete girdim ve java plug-in için en güncel sürümü aradım.

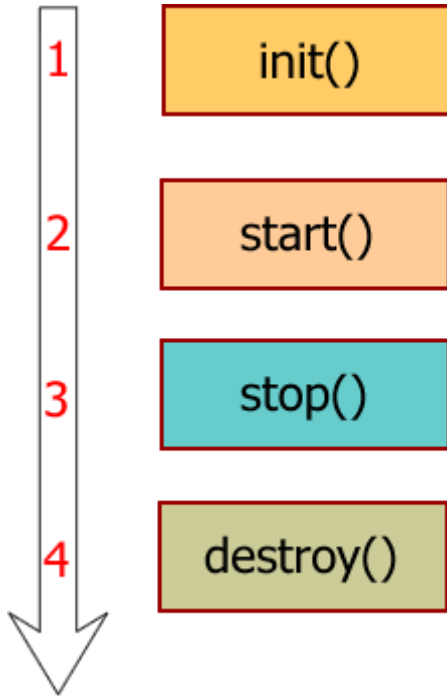
<http://java.sun.com/products/plugin/reference/codesamples/index.html>

Bu adreste örnek java appletleri vardı. En son sürüme ait olanlardan bir tanesini çalıştırmak istediğimde, JVM için gerekli sürümü yüklemek isteyip istemediğimi sordu. Tabiki bunu istiyordum. Hemen yükledim. Hemen derken yüklemek biraz zaman aldı tabiki ama sonuca herşey yoluna girdi. Bu işlemin sonucunda html sayfamı tarayıcıdan tekrar çalıştırdığımda aşağıdaki sonucu elde ettim.



Yihuuu

Appletim html sayfasından da çalışmıştı. Harika. Bu appletin ardından daha gelişmiş bir applet yazmam gerektiğini düşünüyordum ki karşıma appletlerin çalıştırıldığında gerçekleşen olayların bir listesi geliverdi. Bir applet çalıştırıldığında aslında aşağıdaki olaylar gerçekleştiriliyordu.



Görüldüğü gibi bir appletin çalışması sırasında işleyen 4 temel olay var. Bu metodlardan ilki init metodu, applet tarayıcı bilgisayara indirildiğinde çalıştırılmaktadır. Start metodundaki kod satırları ise applet çalışmaya başladığında tetiklenir. Stop metodunda yer alan kodlar, appletin bulunduğu sayfadan başka bir sayfaya atlandığında dolayısıyla applet kapatıldığında çalıştırılır. Destroy metodundaki kodlar ise, tarayıcı penceresi kapatıldığı sırada çalıştırılır. Elbette birde paint metodumuz var. Bu metod ile, appletin içerisinde tarayıcı penceresinde belirlenen alanlarda birşeyler çizdirmek için kullanacağımız kodlar yer alır. Diğer yandan,

kullanıcı ile etkileşim halinde olan appletlerde, kullanıcının tepkisine göre applet üzerinde yapılacak yeni çizimler repaint isimli metodlar içerisinde gerçekleştirilir.

Şimdi bana bu metodların bir appletin çalışması sırasında nerelerde devreye girdiğini gösterecek bir örnek gerekiyordu. Hemde appleti biraz daha geliştirmiş olurdum. Bu amaçla kaynaklarımdan edindiğim bilgiler ışığında aşağıdaki gibi bir java applet sınıfı oluşturdum.

```
import java.awt.*;
import java.applet.Applet;

public class IlkApplet extends Applet
{
    public void init()
    {
        setBackground(Color.yellow);
        System.out.println("Applet yüklendi...");
    }

    public void paint(Graphics g)
    {
        g.drawString("paint",50,50);
    }

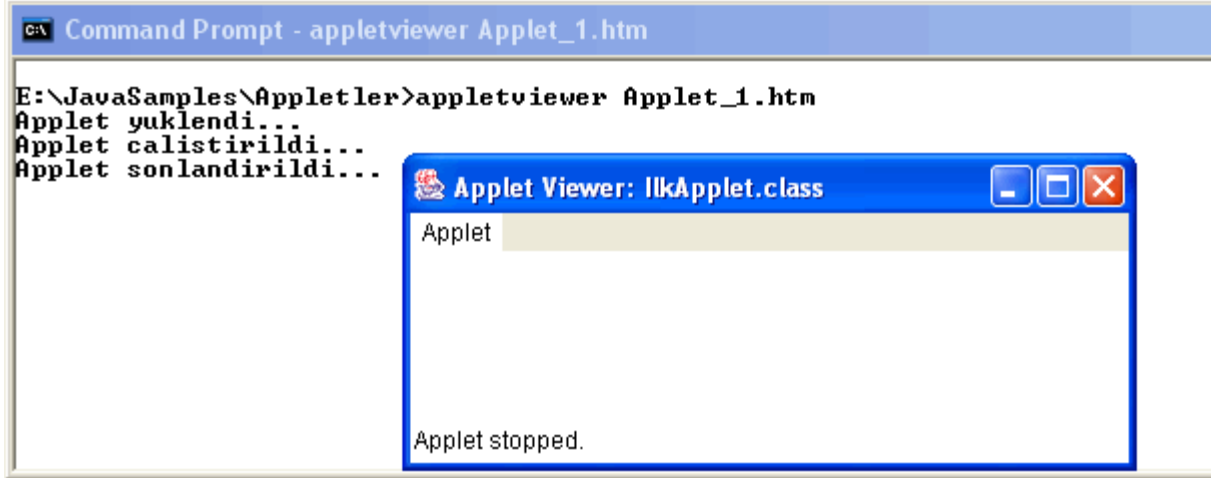
    public void start()
    {
        System.out.println("Applet çalıştırıldı...");
    }
    public void stop()
    {
        System.out.println("Applet sonlandırıldı...");
    }
}
```

Bu applet sınıfını derleyip appletviewer ile çalıştırdığımda ilk olarak aşağıdaki görüntüyü elde ettim.



Görüldüğü gibi ilk önce init metodu devreye girdi. Ardından applet'in start metodunda

yer alan kodlar çalıştırıldı ve sonrasında ise paint metodundaki kodlar devreye girdi. Çalışan bu applet'i Applet Viewer'ın Applet menüsünden stop komutu ile durdurduğumda ise aşağıdaki ekran görüntüsünü elde ettim.



Bu kez applet'in stop metodundaki kodlar devreye girmişti ve applet'in çalışmasında sona ermişti. Applet'lerin çalışma sistemini anladıktan, yaşam süresi boyunca çalıştıracağı metodları ve gerçekleşen olayları inceledikten ve bir buçuk satırlık applet kodu yazdıktan sonra, daha işe yarar bir örnek görmek istiyordum. Hatta yazmak istiyordum. Ancak işe yaramasa bile beni etkileyebilecek bir örnek bulmanın daha iyi olacağı kanısına vardım. Bu amaçla Sun'ın sitesinden örnek appletlere baktım.

<http://java.sun.com/products/plugin/1.5.0/demos/plugin/applets/MoleculeViewer/example2.html>

Bu adreste yer alan applet'i güzelce bir inceledim. Applet'te yapılması gereken, mouse'a basılı tutup şekli herhangi bir yöne doğru sürüklemeye çalışmaktı. Açıkçası bu applet'e bakınca ve şu an java dilinde bulunduğum yeri düşününce kendi kendime şöyle dedim. "ÇOOOOK ÇALIŞMAM LAZIMMM. ÇOOOOKK!!!". Neyseki önümüzdeki hafta boyunca, java appletlerinde awt sınıfına ait GUI nesnelerini kullanarak kullanıcılar ile nasıl dinamik etkileşime geçileceğini öğrenmeye çalışacağım. Artık dinlenmenin tam zamanı. Kahvemde bitmiş zaten.

Burak Selim ŞENYURT

[selim@bsenyurt.com](mailto:selim@bsenyurt.com)

Yazar: Burak Selim Şenyurt

## Java 24 Bölüm 15: Appletler ile Görsel Tasarım (Hiç Olmasa Bir Başlangıç)

Geçen hafta boyunca Applet'lerin büyüğü dünyasını daha çok keşfedebilmek için uğraştım durdum. Nitekim Applet'leri daha etkin bir şekilde kullanabileceğimi ve Applet'lerin çok daha fazlasını sunduğunu biliyordum. Örneğin, kullanıcılar ile dinamik etkileşime geçilmesini sağlayacak tarzda applet'ler üzerinde çalışabiliyordum. Bu konuda aklıma ilk gelen, bilgi giriş formu ekranı oldu. İlk başta nereden başlamam gerektiğini bilmiyordum. Applet'lerin çalışma mimarisinden haberim vardı. Ancak, görsel öğeleri bu applet'ler üzerinde nasıl oluşturabilirdim? Dahada önemlisi, bu görsel nesnelerin, kullanıcı aktivitelerine cevap vermesini nasıl

programlayabilirdim? Bir başka deyişle, görsel programlamanın en önemli yapıtaşlarından birisi olan olay-güdümlü programlamayı nasıl gerçekleştirebilirdim? Tüm bu soruların cevaplarını bulmak maksadıyla, hafta boyunca araştırmalarımı sürdürdüm.

Kilit nokta, Java dilinin Awt isimli (Abstract Windows Toolkit) paketi idi. Awt hem applet' ler için hemde ileride incelemeyi düşündüğüm normal GUI (Graphical User Interface) ler için ortak nesne modellerini kapsülleyen bir paketti. İlk okuduğumda bu paketin, Voltran' ın parçalarından birisi olduğunu zannetmişim. Ancak sağladığı imkanlar ile, Voltran' ın değil gövdesi tüm benliğini oluşturabilirdim. İşin geyiği bir yana, Awt paketi, java ile geliştirilen herhangi bir GUI uygulaması için gerekli olan tüm görsel bileşenleri sağlamaktaydı. Hatta bu bileşenlerin her GUI uygulamasında aynı tipte görünmesinde imkan tanıyordu.

Özellikle Visual Studio.Net gibi görsel geliştirme ortamlarında program arayüzlerini (interface programlama değil, görsel tasarım anlamında) tasarlamak son derece kolay. Ama ister .net platformunda olsun ister Java platformunda, nesne yönelimli dillerin doğası gereği tüm görsel bileşenlerde aslında birer sınıf modelinin örneklemelerinden başka bir şey değiller. Dolayısıyla Awt paketi içindeki görsel bileşenlerinde birer sınıf modeli olduğunu belirtmekte yarar var. Örneğin, sayfalarda gösterebileceğim butonlar Button sınıfına ait nesne örnekleri olacak. Sadece metin bilgisi taşıyan okuma amaçlı Label bileşenleri, Label sınıfına ait olacak. Yada Checkbox, TextField, TextArea, List, Image kontrolleri vs...

Artık bir noktadan başlamam gerektiğini düşünüyordum. Kahvemden bir yudum aldım ve ilk önce nasıl bir form tasarlamak istediğime karar verdim. Bunu kağıt üzerinde çizmek kolaydı ancak dijital ortama aktarmak zordu. Tasviri tam yapmak için, Vs.Net editörünü kullandım ve aşağıdaki gibi bir formu, applet olarak tasarlamaya karar verdim.

Ad

Soyad

Adres

Label

CheckboxGroup

Cinsiyet ☐ Erkek ☐ Kadın

Hobi ☐ Internet ☐ Sinema

☐ Müzik ☐ Tiyatro

Checkbox

Button Yaz

Bu formu oluşturmak için hangi sınıfları kullanmam gerektiğinde, JSDK'dan baktım. Şu ana kadar her şey açık ve netti. Şimdi sıra kodlama kısmına gelmişti. Aslında ne yapmam gerektiğini açıkça tahmin edebiliyordum. Applet, bileşenleri üzerinde barındıracak yer olduğuna göre, Applet' i oluştururken, başka bir deyişle applet' i çalışır hale getirirken bu nesneleri yükleyebilirdim. Ancak bundan önce bu bileşen nesnelerini oluşturmam ve daha sonra bir şekilde Applet'e eklemem

gerekliyordu. Felsefe işte bu kadar basitti. Felsefenin asıl karışacağı noktanın, bu bileşen nesnelerinin olaylara cevap verebilecek şekilde kodlanmasında oluşacağını da hissediyordum. Ancak ilk etapta, öncelikle applet'i tasarlamam ve en azından bileşenleri applet üzerinde sorunsuz bir şekilde göstermem gerektiği kanısındaydım. Hemen kolları sıvadım ve aşağıdaki applet sınıfını oluşturdum.

```
import java.awt.*;
import java.applet.Applet;

public class Gui_1 extends Applet
{
    public void init()
    {
        setBackground(Color.white);

        Label lbAd=new Label("Ad ");
        Label lbSoyad=new Label("Soyad ");
        Label lbAdres=new Label("Adres ");
        Label lbCinsiyet=new Label("Cinsiyet ");
        Label lbHobi=new Label("Hobi ");

        TextField txtAd=new TextField();
        TextField txtSoyad=new TextField();

        TextArea txtAdres=new TextArea(2,5);

        CheckboxGroup cbCinsiyet=new CheckboxGroup();

        Checkbox cbInternet=new Checkbox("Internet");
        Checkbox cbMuzik=new Checkbox("Muzik");
        Checkbox cbSinema=new Checkbox("Sinema");
        Checkbox cbTiyatro=new Checkbox("Tiyatro");

        Button btnYaz=new Button("Yaz");

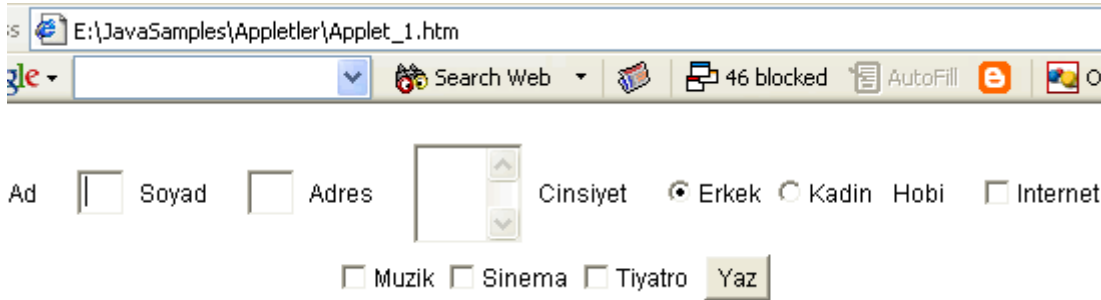
        add(lbAd);
        add(txtAd);
        add(lbSoyad);
        add(txtSoyad);
        add(lbAdres);
        add(txtAdres);
        add(lbCinsiyet);
        add(new Checkbox("Erkek",cbCinsiyet,true));
        add(new Checkbox("Kadin",cbCinsiyet,false));
        add(lbHobi);
        add(cbInternet);
        add(cbMuzik);
        add(cbSinema);
        add(cbTiyatro);
        add(btnYaz);
    }
}
```

Bu uzun kod dosyasında temel olarak yapılan işlemler çok basitti. Herşeyden önce



nesneleri oluşturabileceğim en uygun yer init metoduydu. Burada her bir görsel bileşen nesnesini teker teker new operatörü ile oluşturdum. Label bileşenleri için yapıcılara Label'ın başlık metnini gönderdim. Aynı işlemi, CheckBox bileşenleri ve Button bileşeni içinde yaptım. Böylece ekrandaki Label, Checkbox ve Button bileşenlerinin başlıklarının ne olacağını otomatik olarak belirlemiş oldum. Textfield bileşenleri içinde parametreler girilebilir. Özellikle Textfield'ın boyutunu belirlemek için. Textarea bileşeni için ise, iki parametre girdim. İlki satır sayısını ikincisi de sütun sayısını göstermekte. Burada tek özel oluşum radyo buton dediğim CheckboxGroup bileşenine ait. Bu bileşene, yine Checkbox bileşenleri ekleniyor. Yani bir CheckboxGroup bileşeni birden fazla Checkbox kontrolünü aynı isim altında gruplamak ve böylece bu bileşenlerden sadece birisinin seçili olmasını garanti etmek amacıyla kullanılmakta.

Bu bileşen örneklerinin oluşturulmasından sonra tek yaptığım add metodunu kullanarak bunları Applet'imin üzerine eklemek oldu. Sonuç mu? Aslında göstermeye çekiniyorum. Çünkü hiçte umduğum gibi değil. Java bytecode dosyasının class olarak derledikten sonra bir html sayfasına <Applet tagını kullanarak ekledim. İşte sonuç;



Ahhhh!!! Ahhh. Nerede güzelim Visual Studio.Net, Dreamweaver, Frontpage....İyi güzel bileşenleri oluşturmuştum, applet'ede başarılı bir şekilde eklemiştim. Ama ya görsel tasarımın zerafeti ne olacak. Tam bu noktada işte çakılıp kalmıştım. Kaynaklarıma baktım benim yaptığım bu estetik abidesini onlarda başarmışlardı. Kaynaklarımı biraz daha araştırdıktan sonra aslında bu tasarım ve sayfaya yerleştirme işinin bazı kitaplarda bölüm olarak işlendiğini gördüm. Ancak şu an için bana en azından ızgaralanmış bir görümü hazırlayabileceğim bir teknik gerekiyordu. Sonunda buldum ama... Gerçi bulana kadar bir kaç fincan kahveyide bitirdim. İşin sırrı Olinde değil, tabiki sağdakindeydi. Yani Layout tekniği. Uyguladığım teknik GridLayout tekniği. Tek yapmam gereken tüm kontrolleri eklemekten önce, Applet üzerinde bir GridLayout yani ızgara belirlemektir. Bunu gerçekleştirmek için Applet'in init metodunun en başına aşağıdaki kod satırını ekledim.

```
setLayout(new GridLayout(15,2));
```

Bu satır ile Applet'in web sayfasında kaplayacağı alanı, 15 satır ve 2 sütuna bölmüştüm. Artık GUI bileşenleri sırasıyla yerleşecekti. Ancak yinede sonuç istediğim gibi olmamıştı.

Address E:\JavaSamples\Appletler\Applet\_1.htm

Ad

Soyad

Adres

Cinsiyet

☒ Erkek

☐ Kadın

Hobi

☐ Internet

☐ Muzik

☐ Sinema

☐ Tiyatro

Hiç yoktan iyidir diyerek devam etmek zorundaydım. Ancak Layout ayarlamaları ile ilgili olarak başka bir kahve molasında daha derin bir araştırma yapmayıda kafama koymuştum. Olay yerinden uzaklaşırken, en azından applet üzerinde dinamik olarak görsel bileşenlerin nasıl eklendiğini anlamış ve bir kaç bileşenide öğrenmiştim. Asıl merak ettiğim, butona basıldığında olmasını istediklerimi nasıl yazacağımıydı? Bunun için, C# dilinde özellikle görsel programlamada delegeler ile yakın ilişkide olan event'lar kullanılıyordu. Java dilindede durum çok farklı değildi ancak anlaşılması daha kolaydı. Java dilindede, kullanıcı tepkilerini ele alabilmek için delegasyon mantığı kullanılıyordu. Bu modelin en önemli yanı, görsel bileşenlerinin kullanıcı tepkilerini algılayabilmelerini istediğimiz Applet sınıfına, ActionListener arayüzünü uygulamamız gerekliliği idi. Kolları sıvadım ve ilk olarak, en basit haliyle, Button bileşenime tıklandığında meydana gelecek kodları hazırladım.

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class Gui_1 extends Applet implements ActionListener
{
    TextField txtAd;
    TextField txtSoyad;
    Button btnYaz;
```

```

public void init()
{
    setBackground(Color.white);

    Label lbAd=new Label("Ad ");
    Label lbSoyad=new Label("Soyad ");
    Label lbAdres=new Label("Adres ");
    Label lbCinsiyet=new Label("Cinsiyet ");
    Label lbHobi=new Label("Hobi ");

    txtAd=new TextField();
    txtSoyad=new TextField();

    TextArea txtAdres=new TextArea(2,5);

    CheckboxGroup cbCinsiyet=new CheckboxGroup();

    Checkbox cbInternet=new Checkbox("Internet");
    Checkbox cbMuzik=new Checkbox("Muzik");
    Checkbox cbSinema=new Checkbox("Sinema");
    Checkbox cbTiyatro=new Checkbox("Tiyatro");

    btnYaz=new Button("Yaz");

    setLayout(new GridLayout(15,2));

    add(lbAd);
    add(txtAd);
    add(lbSoyad);
    add(txtSoyad);
    add(lbAdres);
    add(txtAdres);
    add(lbCinsiyet);
    add(new Checkbox("Erkek",cbCinsiyet,true));
    add(new Checkbox("Kadin",cbCinsiyet,false));
    add(lbHobi);
    add(cbInternet);
    add(cbMuzik);
    add(cbSinema);
    add(cbTiyatro);
    add(btnYaz);

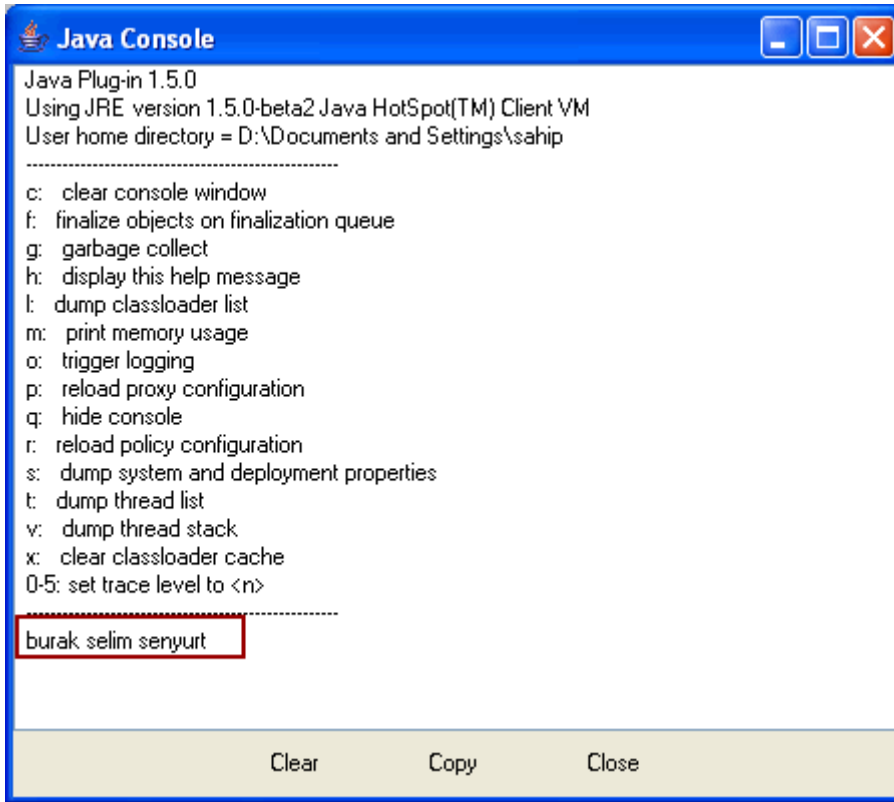
    btnYaz.addActionListener(this);
}

public void actionPerformed(ActionEvent e)
{
    if(e.getSource()==btnYaz)
    {
        System.out.println(txtAd.getText()+" "+txtSoyad.getText());
    }
}
}

```

Burada kullanılan tekniği anlamanın çok önemli olduğunu düşünüyorum. Öncelikle Applet'in üzerindeki nesnelerin olaylarını dinleyecek şekilde programlamalıydım. Bunu gerçekleştirebilmek için, sınıfa ActionListener arayüzünü uyguladım. Ardından hangi bileşen için bir olay dinlemesinin gerçekleştirilmesini istiyorsam, o nesne için addActionListener metodunu this parametresi ile kullandım. Böylece, bu örnekteki Button bileşenine bu applet içinde yapılacak tıklamalar ele alınabilecek ve kodlanabilecekti. Başka bir deyişle, nesneye ait olası olayların dinlemeye alınmasını sağlamıştım. Bu işlemin ardından elbetteki, olay meydana geldiğinde çalıştırılacak kodları yazacağım bir metod gerekliydi. İşte buda, Button bileşenlerine yapılan tıklamaları dinleyen actionPerformed metodu. Bu metod(ActionEvent isimli bir parametre alıyor. Bu parametre sayesinde, hangi buton bileşenine tıklandığını dinleyebilirim. İşte böylece buton bileşenine basıldığında işleyecek olan satırları burada yazmış oldum. Şimdi bu sistemi deneme vakti geldi.

Tarayıcıda sayfamı çalıştırdım txtAd ve txtSoyad kontrollerine isim ve soyisim bilgilerimi girdim buton' a tıkladım ve ekranın sağ alt köşesindeki System Tray'da yer alan JVM kahve sembolünden, Open Console diyerek, console penceresine geçtim. Sonuç olarak tıklama olayım algılanmış ve olaya karşılık gelen kod satırları çalıştırılmıştı.



Şimdi aklıma takılan başka bir nokta vardı. Eğer applet'imde iki button bileşeni olsaydı. Her biri için ayrı ayrı olay dinleyicilerimi yazacaktım? Nitekim, actionPerformed metodunun yapısı buna müsait değildi. Bu amaçla, applet' e TextField ve TextArea kontrollerinin içeriğini temizleyecek yeni bir Button bileşeni ekledim. Kodun son hali aşağıdaki gibi oldu.

```
import java.awt.*;  
import java.applet.Applet;
```

```

import java.awt.event.*;

public class Gui_1 extends Applet implements ActionListener
{
    TextField txtAd;
    TextField txtSoyad;
    TextArea txtAdres;

    Button btnYaz;
    Button btnSil;

    public void init()
    {
        setBackground(Color.white);

        Label lbAd=new Label("Ad ");
        Label lbSoyad=new Label("Soyad ");
        Label lbAdres=new Label("Adres ");
        Label lbCinsiyet=new Label("Cinsiyet ");
        Label lbHobi=new Label("Hobi ");

        txtAd=new TextField();
        txtSoyad=new TextField();

        txtAdres=new TextArea(2,5);

        CheckboxGroup cbCinsiyet=new CheckboxGroup();

        Checkbox cbInternet=new Checkbox("Internet");
        Checkbox cbMuzik=new Checkbox("Muzik");
        Checkbox cbSinema=new Checkbox("Sinema");
        Checkbox cbTiyatro=new Checkbox("Tiyatro");

        btnYaz=new Button("Yaz");
        btnSil=new Button("Sil");

        setLayout(new GridLayout(16,2));

        add(lbAd);
        add(txtAd);
        add(lbSoyad);
        add(txtSoyad);
        add(lbAdres);
        add(txtAdres);
        add(lbCinsiyet);
        add(new Checkbox("Erkek",cbCinsiyet,true));
        add(new Checkbox("Kadin",cbCinsiyet,false));
        add(lbHobi);
        add(cbInternet);
        add(cbMuzik);
        add(cbSinema);
        add(cbTiyatro);
        add(btnYaz);
        add(btnSil);
    }
}

```

```

        btnYaz.addActionListener(this);
        btnSil.addActionListener(this);
    }

    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource()==btnYaz)
        {
            System.out.println(txtAd.getText()+" "+txtSoyad.getText());
        }
        else if(e.getSource()==btnSil)
        {
            txtAd.setText("");
            txtSoyad.setText("");
            txtAdres.setText("");
            repaint();
        }
    }
}

```

İlk hali;

Ad

Soyad

Adres

Cinsiyet

☒ Erkek

☐ Kadın

Hobi

☐ İnternet

☐ Muzik

☐ Sinema

☐ Tiyatro

Yaz

Sil

Sil başlıklı Button bileşenine tıklandıktan sonraki hali.

Ad

Soyad

Adres

Cinsiyet  
☒ Erkek  
☐ Kadın

Hobi  
☐ İnternet  
☐ Muzik  
☐ Sinema  
☐ Tiyatro

Her iki buton bileşenide, aynı olay dinleyicisi içerisinde ele alınmıştı. actionPerformed dinleyicisinde, hangi butona tıklandığı,(ActionEvent parametresinin getSource metodu ile tespit edilmekteydi. Elbetteki awt paketinde yer alan diğer görsel bileşenler için sadece bu olay dinleyicisi söz konusu değildi. İşin aslı, nesneler üzerindeki kullanıcı etkilerine göre dinleyiciler tanımlanmıştı. Örneğin, CheckBox bileşenine tıklanması bu bileşenin durumunun değişmesi anlamına gelmekteydi ki bu durumda, ItemListener dinleyici metodu bu etkiyi ele alacaktı. Diğer nesneler içinde benzer durumlar söz konusu. Ama ne zaman? Bir dahaki kahve kokusunda.

Burak Selim ŞENYURT

[selim@bsenyurt.com](mailto:selim@bsenyurt.com)

Yazar: Burak Selim Şenyurt

## Java 24 Bölüm 17: Layout

Geçtiğimiz hafta hayatımın en mutlu günlerinden birisini, değerli bir arkadaşımın ofisinde, bilgisayarının başında JBuilder kullanarak geçirdim. Bu hafta, o günü mumla aradığımı söylemek isterim. JBuilder ile daha önce hiç uygulama geliştirmemiş olmama rağmen, kolayca adapte olmuştum. Elbetteki benim için en büyük rahatlığı, Applet gibi görsel uygulamaların ekran tasarımlarının son derece kolay bir şekilde yapılabilmesiydi. Bu hafta yine sevimsiz notepad editorüm ile başbaşayım. Aslında amacım arkadaşımın bilgisayarında JBuilder ile başka çalışmalarda yapmaktı. Fakat

kendisi tatile çıktığı için banada notepad ile bir kahve molası geçirmek kaldı.

Özellikle notepad editorünü kullanarak applet tasarlamının en zor yanlarından birisi, applet üzerindeki bileşenlerin yerleşim şekillerinin ayarlanmasının zorluğudur. Bu hafta ne yapıp edip, bu fobiyi yenmeye karar verdim ve java dilinde Layout kavramını incelemeye başladım. Layout' lar applet üzerine yerleştirilecek bileşenlerin belli bir nizamda olmasını sağlamaktadırlar. Java paketiyle gelen Layout sınıfları 5 adettir.

| Java Layouts  |
|---------------|
| GridLayout    |
| BorderLayout  |
| FlowLayout    |
| CardLayout    |
| GridBagLayout |

Öncelikle işe en kolay olanından başladım. FlowLayout. Layout sınıflarını anlamanın en iyi yolu elbette onları bir örnek üzerinde uygulamakla mümkün olabilirdi. Bu amaçla çok basit olarak aşağıdaki gibi bir java örneği geliştirdim.

```
import java.awt.*;
import java.applet.Applet;

public class Layouts extends Applet
{
    TextField tf1;
    TextField tf2;
    Button bt1;
    Label lb1;
    Label lb2;

    public void init()
    {
        setLayout(new FlowLayout(FlowLayout.CENTER,15,30));

        lb1=new Label("Username");
        tf1=new TextField(25);
        lb2=new Label("Password");
        tf2=new TextField(25);
        bt1=new Button(" OK ");

        add(lb1);
        add(tf1);
        add(lb2);
        add(tf2);
        add(bt1);
    }
}
```



Burada tek yaptığım, applet üzerine yerleşecek bileşenlerin FlowLayout 'a göre düzenlenmesiydi. Applet' in FlowLayout' u uygulayacağını belirtmek için,

```
setLayout(new FlowLayout(FlowLayout.CENTER,15,30));
```

kod satırını kullandım. Burada, bileşenlerin Applet üzerinde bulundukları satırda ortalanarak yerleştirilecekleri ve yatay olarak aralarında 15 piksel, dikey olarak ise 30 piksel boşluk olacağı belirtiliyor. Peki ama bu yerleşim nasıl oluyor? Nitekim kaynaklarda, FlowLayout' un bileşenleri sola ve aşağı doğru hizaladığı söyleniyordu. Denemekten başka çarem olmadığını düşünerek hemen işe koyuldum ve basit bir html sayfasında applet tagımı aşağıdaki gibi ekledim.

```
<APPLET CODE="Layouts.class" width="300" height="500">  
</APPLET>
```

Şimdi bu html sayfasını çalıştırdığımda aşağıdaki gibi bir görüntü elde ettim.



Doğruyu söylemek gerekirse kel alaka bir tasarım olmuştu. FlowLayout için diğer iki durum, yerleşim şeklinin sola dayalı olmasını belirten FlowLayout.LEFT ve sağa dayalı olmasını belirten FlowLayout.RIGHT seçenekleriydi. Önce LEFT durumunu inceledim ve aşağıdaki ekran görüntüsünü elde ettim.



Sanki durumu anlamaya başlamış gibiydim. Bu kez RIGHT durumunu denedim ve

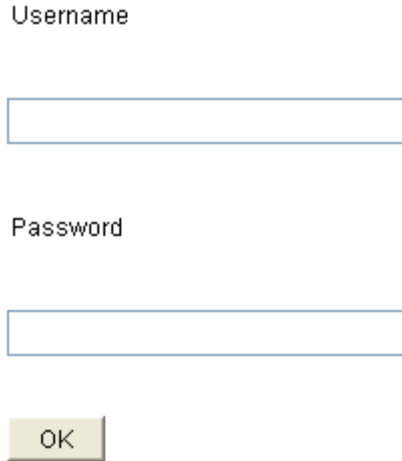


aşağıdaki sonucu elde ettim.

Görünen o ki, düzgün bir tasarım yapmak istiyorsam bunu öncelikle kafamda yapmalı ve Applet' in boyutlarını çok tutarlı belirtmeliydim. Ancak bu şekilde düzgün bir tasarım elde edebilirdim. Şu an için deneme yanılma yöntemini kullanmaktan başka bir şey aklıma gelmiyor açıkçası. Örneğin, en uzun Label olan Username ile TextField' ların boyu düşünüldüğünde Applet tagını aşağıdaki gibi düzenlemek daha mantıklı geliyordu.

```
<APPLET CODE="Layouts.class" width="250" height="500">
</APPLET>
```

Java kodunda da, FlowLayout dizilimini FlowLayout.LEFT olarak belirlediğimde, daha düzenli bir ekran görüntüsü elde ettim. Gerçi bu varsayımsal yaklaşım ile kullanılan teknik pek hoşuma gitmemiştii ama en azından buz dağının üst kısmını biraz olsun yontmayı başaramıştım.



Bu örnekten sonra, arkadaşımı bir kat daha özledim desem yalan olmaz. Heleki JBuilder uygulamasını. Oradaki tasarım rahatlığı gerçekten muhteşemdi. Bu sırada aklıma başka bir şey geldi. Acaba, bir Layout düzeneğini, bir Panel bileşenine uygulayabilir miydim? Eğer böyle bir şey söz konusu olursa, görsel tasarımı biraz daha kolaylaştırabilirdim. Bu amaçla aşağıdaki gibi bir örnek geliştirdim.

```
import java.awt.*;
import java.applet.Applet;

public class Layouts extends Applet
{
```

```

TextField tf1;
TextField tf2;
Button bt1;
Label lb1;
Label lb2;
Panel p1;

public void init()
{
    p1=new Panel();
    p1.setBackground(Color.yellow);
    p1.setLayout(new FlowLayout(FlowLayout.LEFT));

    lb1=new Label("Username");
    tf1=new TextField(25);
    lb2=new Label("Password");
    tf2=new TextField(25);
    bt1=new Button(" OK ");

    p1.add(lb1);
    p1.add(tf1);
    p1.add(lb2);
    p1.add(tf2);
    p1.add(bt1);

    add(p1);
}
}

```

İlk olarak, bir Panel bileşeni oluşturdum ve bu bileşen üzerine yerleştireceğim diğer bileşenlerin FlowLayout düzeneğine göre konumlandırılmalarını sağlamak için Panel bileşenine,

```
p1.setLayout(new FlowLayout(FlowLayout.LEFT));
```

satırındaki setLayout metodunu uyguladım. Böylece, Panel bileşeni üzerine yerleşecek bileşenler, FlowLayout düzeneğine göre, Layout' un solundan hizalanacak şekilde konumlanacaklardı. Bileşenleri Panel' e eklemek için, Panel sınıfına ait add metodunu kullandım. Tabi bu işlemlerden sonra Panel bileşeninde, Applet' e add metodu ile eklemeyi unutmadım. Bu adımlardan sonra, Java dosyasını derleyip applet' i içeren html sayfasını ilk çalıştırdığımda aşağıdaki sonucu elde ettim.

Böyle olacağı belliydi zaten. Applet tagında width özelliğini arttırmam gerekiyordu. Bu değeri 600 olarak belirledim. Şimdi elde ettiğim sonuç çok daha iyiydi.

Layout sınıfları bitmek bilmiyordu. Sırada BorderLayout sınıfı vardı. Bu sınıfın en ilginç yanı, Applet ekranının, NBA basketbol takımlarının liglerinde gruplanışlarına benzer bir yapıda ayrıştırılıyor olmasıydı. Doğu Grubu, Merkez Grubu, Batı Grubu, Kuzey Grubu ve Güney Grubu. Yani, applet üzerine ekleyeceğim bileşenleri, bu gruplara yerleştirmem gerekiyordu. Bunu görsel olarak anlayabilmek için, kaynaklarımı araştırdım ve yukarıdaki örneğe bu kez BorderLayout düzeneğini uyguladım.

```
import java.awt.*;
import java.applet.Applet;

public class Layouts extends Applet
{
    TextField tf1;
    TextField tf2;
    Button bt1;
    Label lb1;
    Label lb2;

    public void init()
    {
        setLayout(new BorderLayout());

        lb1=new Label("Username");
        tf1=new TextField(25);
        lb2=new Label("Password");
        tf2=new TextField(25);
        bt1=new Button(" OK ");

        add("North",lb1);
        add("Center",tf1);
        add("West",lb2);
        add("South",tf2);
        add("East",bt1);
    }
}
```

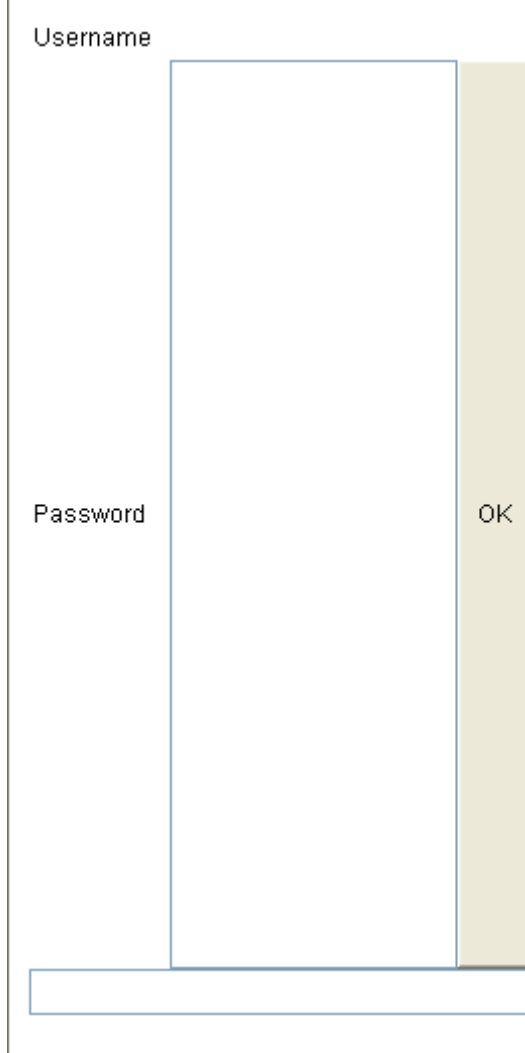
Applet üzerindeki bileşenlerin BorderLayout düzeneğine göre yerleştirileceğini,

```
setLayout(new BorderLayout());
```

satırı belirtiyordu. BorderLayout, Applet' i 5 yön bölgesine böldüğü için, her bileşeni Applet' e eklerken add metoduna ilk parametre olarak, bileşenin hangi bölgeye yerleştirileceğinin belirtilmesi gerekiyordu. Bu işlemde aşağıdaki satırla gerçekleştirmektedir.

```
add("North",lb1);
add("Center",tf1);
add("West",lb2);
add("South",tf2);
add("East",bt1);
```

Örneğin, tf2 isimli TextField bileşeni Applet'in South (Güney) bölgesine yerleşecekti. Artık Applet' i kullandığım html sayfasını çalıştırıp sonuçları görmek istiyordum. Ama sonuçları göstermeye çekiniyorum açıkçası. BorderLayout' un Applet ekranına uygulanması sonucu aşağıdaki gibi muhteşem ötesi bir tasarım elde ettim. Tarihi bir başarı olduğunu belirtmek isterim.



İşin kötü yanı, belli bir bölgeye sadece tek bir bileşen yerleştirebiliyor olmasıydı. BorderLayout düzeneğinin nerede kullanılacağını düşünürken, diğer Layout' ları incelemenin daha uygun olacağını farkettim. Nitekim, BorderLayout her ne kadar yukarıdaki ekran tasarımı için uygun olmasada elbetteki kullanıldığı bir takım yerler olabilirdi. Layout' lar ile uğraşmak gerçekten insana sıkıcı geliyor. Artık JBuilder gibi görsel tasarım araçlarının o kadar paraya gerçekten deydığını söyleyebilirim. Hemde gönül rahatlığıyla. Bu düşünceler eşliğinde kendime yeni bir fincan kahve aldıktan sonra bir diğer Layout sınıfı olan GridLayout' u incelemeye başladım. GridLayout ile nispeten biraz daha güzel ve kolay form tasarımları oluşturabileceğimi düşünüyordum. Yapmam gereken son derece basitti. Aynı örneği GridLayout sınıfı ile geliştirmek.

```
import java.awt.*;  
import java.applet.Applet;
```

```

public class Layouts extends Applet
{
    TextField tf1;
    TextField tf2;
    Button bt1;
    Label lb1;
    Label lb2;
    Panel p1;
    Panel p2;

    public void init()
    {
        setLayout(new GridLayout(2,1));

        p1=new Panel();
        p1.setBackground(Color.orange);
        p1.setLayout(new FlowLayout(FlowLayout.LEFT));

        p2=new Panel();
        p2.setBackground(Color.blue);

        lb1=new Label("Username");
        tf1=new TextField(25);
        lb2=new Label("Password");
        tf2=new TextField(25);
        bt1=new Button(" OK ");

        p1.add(lb1);
        p1.add(tf1);
        p1.add(lb2);
        p1.add(tf2);
        p1.add(bt1);

        add(p1);
        add(p2);
    }
}

```

Bu kez, Applet'e iki Panel bileşeni ekledim. Bu iki Panel bileşeninde 2 satır ve 1 sütunluk bir GridLayout düzeneği içinde Applet' e ekledim. GridLayout nesneleri, temel olarak iki parametre almakta. İlk parametre, satır sayısını belirtirken, ikinci parametrede doğal olarak sütun sayısını belirtiyor. Bununla birlikte, GridLayout sınıfının 4 parametre alan bir diğer yapıcısı da mevcut. Bu yapıcının aldığı son iki parametre ise, Grid hücrelerine yerleştirilecek bileşenlerin arasındaki yatay ve dikey uzaklıkları piksel olarak belirtmekte. Bu örneği geliştirdikten sonra Applet tagını aşağıdaki gibi düzenledim.

```

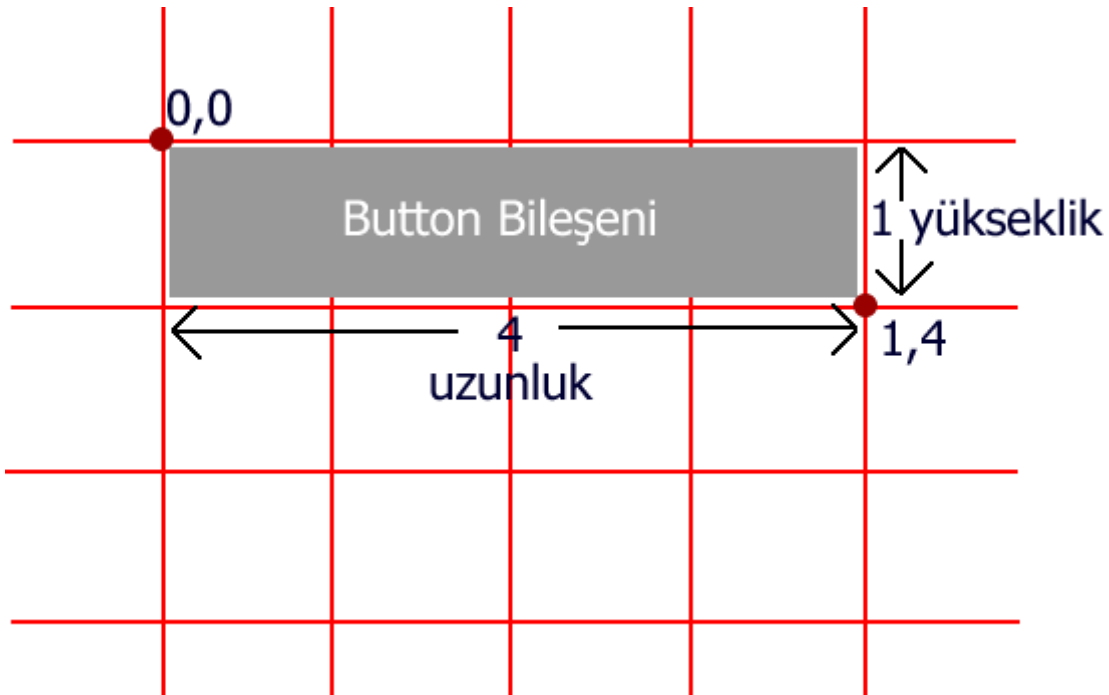
<APPLET CODE="Layouts.class" width="610" height="100">
</APPLET>

```

Sonuç ise, FB (FB lilerin şampiyonluklarında tebrik ederim bu arada) renkleriyle aşağıdaki gibi oluştu. BJK taraftarı olmama rağmen, tasarımı böyle

renklendirebildiğim için çok mutlu oldum. Demek ki, Layout' ları bir arada kullanarak ve işin içine Panel bileşenlerini katarak daha sağlam tasarımlar oluşturulabilirdi. Notepad ile dahi olsa. Ama bununla kim uğraşırdıki güzelim JBuilder varken. (Elbette ben uğraşırdım sanırım.)

GridLayout sınıfında geçtim derken karşıma daha karmaşık bir Layout sınıfı çıkıverdi. GridBagLayout. Bunu anlayabilmek için, Bilim Teknik dergisinde her ay çıkan soruları çözer gibi kağıt kaleme sarılıp çalışmam gerektiğini itiraf etmeliyim. GridBagLayout sınıfı ile, Applet üzerinde çok karmaşık ekran tasarımlarını yapabilmek mümkün. Ancak bu işi başarabilmek için bir o kadar karışık bir teknik kullanmak gerekiyor. Burada önemli olan nokta, Applet üzerine yerleşecek bileşenlerin X,Y koordinatları ile uzunluk ve yükseklik bilgileri. Nitekim, GridBagLayout tekniğinde, Applet karelere bölünüyor ve bileşenler bu kareler üzerine yerleştiriliyor. Şimdi burada hakketten ele kağıt kalem almak ve çizmek lazım. Ancak dijital bir ortamda olduğumuzu düşünürsek bu iş için, gelişmiş grafik programlarının da kullanabilirim. İşte bu amaçla bu Layout sınıfını kullanarak, Applet üzerinde konumlandırılacak bir Button nesnesini göz önüne alarak basit bir çizim oluşturdum.



Temel olarak yapacağım işlem buydu. Applet ekranını karelere bölmek. Örneğin bir Button yerleştirmek istiyorum. Önceden bu Button bileşenin X, Y koordinatlarını, uzunluk ve yükseklik bilgilerini, yukarıdaki gibi bir şekli baz alarak bilmem gerekiyor. Olayın esprisi buydu işte. Bu nedenle de GridBagLayout diğer Layout sınıflarına göre daha karmaşık Applet tasarımları oluşturmamıza izin veriyordu. Çünkü bir ekran pozisyonlaması için gereki 4 önemli unsuru baz alıyordu. X,Y Koordinatları, uzunluk ve yükseklik. Peki bu zahmetli tasarımı kodlamada nasıl gerçekleştirebilirdim. Bu

amaçla epeyce bir saatimi, sırf GridBagLayout' un nasıl kullanıldığını anlamaya harcadım. Sonuç olarak aşağıdaki gibi bir örnek geliştirdim.

```
import java.awt.*;
import java.applet.Applet;

public class Layouts extends Applet
{
    TextField tf1;
    TextField tf2;
    Button bt1;
    Label lb1;
    Label lb2;
    GridBagLayout gbl;
    GridBagConstraints gbc;

    public void init()
    {

        gbl=new GridBagLayout();
        gbc=new GridBagConstraints();

        setLayout(gbl);
        gbc.insets=new Insets(2,2,2,2);
        gbc.fill=GridBagConstraints.BOTH;

        gbc.gridx=0;
        gbc.gridy=0;
        gbc.gridwidth=5;
        gbc.gridheight=1;
        lb1=new Label("Username");
        gbl.setConstraints(lb1,gbc);
        add(lb1);

        gbc.gridx=6;
        gbc.gridy=0;
        gbc.gridwidth=5;
        gbc.gridheight=1;
        tf1=new TextField(25);
        gbl.setConstraints(tf1,gbc);
        add(tf1);

        gbc.gridx=0;
        gbc.gridy=1;
        gbc.gridwidth=5;
        gbc.gridheight=1;
        lb2=new Label("Password");
        gbl.setConstraints(lb2,gbc);
        add(lb2);

        gbc.gridx=6;
        gbc.gridy=1;
        gbc.gridwidth=5;
        gbc.gridheight=1;
```



```

tf2=new TextField(25);
gbl.setConstraints(tf2,gbc);
add(tf2);

gbc.gridx=0;
gbc.gridy=3;
gbc.gridwidth=5;
gbc.gridheight=1;
bt1=new Button(" OK ");
gbl.setConstraints(bt1,gbc);
add(bt1);
}
}

```

Kodları yazdıktan sonra önce Applet' in nasıl çalıştığına baktım. Sonuç hiçte fena değildi.



Nasıl olmuşuda böylesine güzel bir sonuç elde edebilmişim. Herşeyden önce, GridBagLayout sınıfını oluşturdum. Ancak burada GridBagConstraints isimli başka bir sınıf daha vardı. Bu sınıf, her bir bileşen için gerekli X, Y koordinatları ile, uzunluk ve yükseklik ayarlamalarını taşıyacak nesneler örneklendirmek amacıyla kullanılmaktaydı. Dolayısıyla bir bileşeni, GridBagLayout sınıfı nesnesine eklerken setConstraints metodu ikinci parametre olarak, GridBagConstraints sınıfına ait nesne örneğini almaktaydı. Böylece, setConstraints metodunun ilk parametresinde belirtilen bileşen, Applet üzerinde belirtilen X, Y koordinatlarına, belirtilen yükseklik ve uzunlukta çiziliyordu. Burada uzunluk ve yükseklik piksel bazında değil hüce bazındadır. Örneğin aşağıdaki satırları göz önüne alalım.

```

gbc.gridx=6;
gbc.gridy=1;
gbc.gridwidth=5;
gbc.gridheight=1;
tf2=new TextField(25);
gbl.setConstraints(tf2,gbc);
add(tf2);

```

Burada TextField bileşeni, X=6, Y=1 koordinatlarına yerleştirilmiştir. Uzunluğu 5 hüce kadar, yüksekliği ise 1 hüce kadardır. Daha sonra, bu bileşen GridBagLayout nesnesine, setConstraints metodu ile eklenmiştir. Artık, bu bileşeni Applet' e add metodu ile eklediğimizde, GridBagLayout konumları esas alınacaktır.

```

gbc.insets=new Insets(2,2,2,2);
gbc.fill=GridBagConstraints.BOTH;

```

Satırlarına gelince. Bu satırlarda, GridBagConstraint nesnesinin bir takım özellikleri belirlenmektedir. Bunlardan birisi insets özelliğidir. Insets ile, hücreler içindeki bileşenlerin hücrenin kenarlarına olan uzaklığı belirtilmektedir. Bir başka deyişle boşluk miktarı. Fill özelliği ileyse, GridBagLayout' un genişlemesi durumunda, hücre içi bileşenlerin her yönde eşit şekilde genişlemesi belirtilmiştir.

Bu zor ama güçlü Layout' tan sonra, incelemeyi unuttuğum bir Layout daha olduğunu farkettim. CardLayout. İlk başta iskambil desteleri ile bir alakası olabilir mi diye düşündüm. Gerçektende öyleymiş. CardLayout' ta, birbirlerinin üstüne binen katmanlar söz konusu. İşin güzel yanı ise, CardLayout' lar ile, birbirinden farklı katmanların tasarlanabilmesi ve çalışma zamanında bu katmanlardan sadece birisinin görünür olması. Dolayısıyla bu katmanlar arasında gezinmek için, olay güdümlü programlama tekniklerini kullanmak gerekiyor. Nasıl mı? İşte örnek.

```
import java.awt.*;
import java.applet.Applet;
import java.awt.event.*;

public class Layouts extends Applet implements ActionListener
{
    TextField tf1;
    TextField tf2;
    Button bt1;
    Button bt2;
    Label lb1;
    Label lb2;
    Label lb3;
    Panel p1;
    Panel p2;
    Panel p3;
    Panel p4;
    CardLayout iskambil;

    public void init()
    {

        iskambil=new CardLayout();

        p1=new Panel();
        p2=new Panel();
        p2.setBackground(Color.blue);
        p3=new Panel();
        p3.setBackground(Color.green);
        p4=new Panel();
        iskambil=new CardLayout();
        p4.setLayout(iskambil);

        lb1=new Label("Username");
        p1.add(lb1);

        tf1=new TextField(25);
        p1.add(tf1);
```

```

lb2=new Label("Password");
p1.add(lb2);

tf2=new TextField(25);
p1.add(tf2);

bt1=new Button("Onceki");
bt1.addActionListener(this);
bt2=new Button("Sonraki");
bt2.addActionListener(this);

p2.add(bt1);
p2.add(bt2);

p4.add("Giris",p1);
p4.add("Diger",p3);

add(p4);
add(p2);
}
public void actionPerformed(ActionEvent e)
{
    if(e.getActionCommand()=="Onceki")
    {
        iskambil.previous(p4);
    }
    else if(e.getActionCommand()=="Sonraki")
    {
        iskambil.next(p4);
    }
}
}

```

Ve sonuç. CardLayout daha güzel bir tasarım sundu. Tek bir applet sayfasında üst üste duran paneller arasında gezebilme imkanı.

Username  Password

---

Artık Layout' lara veda etme vakti geldi. Gelecek kahve molalarında umarım JBuilder ile daha fazla ilgilenme fırsatı bulabilirim. Katetmem gereken daha çok kilometre taşı var. Network programlama, veritabanları, web servisleri, swing bileşenleri, windows uygulamaları vs...

Burak Selim ŞENYURT

[selim@bsenyurt.com](mailto:selim@bsenyurt.com)

Yazar: Burak Selim Şenyurt

## Java 24 Bölüm 18: Pencereleler

Bir kaç haftadır Java dilini popüler yapan Applet' ler ile uğraşıyorum. Buna karşın günümüz dünyasının bir programlama dilinden bekledikleri arasında mutlaka windows uygulamalarının var olması gerektiğini düşünüyorum. Sonuç olarak Applet' ler her ne kadar çok başarılı olsalarda, zaman zaman windows uygulamaları geliştirmemizde gerekiyor. Bir windows uygulamasının belkide en temel özelliği mutlaka bir Form (Nam-ı diğer pencere diyebiliriz) ekranına sahip olması. Peki java dilinde, windows uygulamaları oluşturmak için nasıl bir yol izlemem gerekir. İşte bu hafta boyunca, java dili ile bağımsız olarak çalışabilen pencereleri incelemeye çalıştım.

Sun' in Java platformu, Microsoft' un ciddi rakiplerinden birisi. Belkide tek ciddi rakibi. Ancak bu rekabet zaman zaman biraz komik olaylarada neden olmuyor değil. Örneğin, yaptığım araştırmalarda gördüm ki, Windows uygulamalarında Form kavramı, java dilinde Frame olarak adlandırılıyor. Bu kısa politik düşüncelerden sonra, artık ilk form ekranımı, pardon düzeltiyorum; ilk frame ekranımı tasarlamam gerektiğine karar verdim. Bu amacımı gerçekleştirebilmek amacıyla aşağıdaki çok kısa uygulamayı yazdım.

```
import java.awt.*;

public class IlkPencere
{
    public static void main(String args[])
    {
        Frame pencere=new Frame("ILK PENCEREM");
        pencere.setLocation(0,0);
        pencere.setBackground(Color.red);
        pencere.setVisible(true);
    }
}
```

Yazdığım bu java dosyasının derledikten sonra çalıştırdım. Karşımda beni bekleyen güzel bir pencere olacağı düşüncesindeydim. Gerçektende muazzam bir pencere oluşturmayı başarmıştım :)



Doğruyu söylemek gerekirse daha büyük bir frame olacağını düşünmüştüm. Bunun üzerine yazmış olduğum kod satırlarını incelemeye başladım. İlk olarak awt.window paketinde yer alan Frame sınıfından bir nesne örneği oluşturmuştum. Bunu yaparkende, yapıcı metoda string tipte bir parametre gönderdim. Bu parametre Frame penceresinin başlığı (Title) olacaktı.

```
Frame pencere=new Frame("ILK PENCEREM");
```

Daha sonra, Frame' in ekran üzerindeki konumunu belirledim. Bunun içinde setLocation metoduna X ve Y koordinatlarını 0 olarak verdim. Böylece, Frame penceresi ekranın sol üst köşesinde konumlanacaktı.

```
pencere.setLocation(0,0);
```

setBackground metodu ile Frame penceresinin arka plan rengini kırmızı olarak belirledim.

```
pencere.setBackground(Color.red);
```

Frame sınıfının en önemli metodu ise setVisible idi. Bu metod, oluşturulan Frame penceresinin gösterilmesini sağlıyordu. Bunun için parametre olarak metoda true değerini vermek yeterliydi.

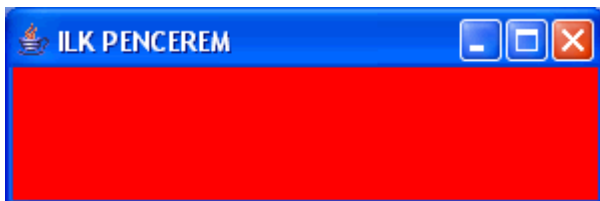
```
pencere.setVisible(true);
```

Buraya kadar herşey sorunsuz gözüküyordu. Ancak Frame' in neden böyle görüldüğünü tam olarak anlayamamıştım. Kaynaklarımı gözden geçirdiğimde, setSize isimli metodu kullanmadığımı farkettim. Bu metod ile Frame' in başlangıç boyutlarını belirleyebiliyordum. Şimdi tek yapmam gereken uygulama koduna setSize metodunu ilave etmek olacaktı. Lakin ufak bir sorun vardı. O da, Frame penceresini X butonuna basıp kapatamıyor oluşuydu. Programdan çıkamıyordum. Bunun tek bir nedeni olabilirdi o da, X butonu ile kapatma işlemi için gerekli olan olay dinleyecisinin ilgili olay metodunu çağtırmayıyordu.

Frame sınıfının olaylarına sonradan zaten bakacaktım. Ancak bu pencereyi bir şekilde kapatıp, kodumu düzenlemek istiyordum. Yaklaşık bir yarım saat kadar sırf bu pencerenin nasıl kapatılacağını araştırdım. Nitekim windows' un ALT+F4 tuş kombinasyonu dahi işe yaramıyordu. Sonunda komut satırından CTRL+C tuş kombinasyonuna basmam gerektiğini öğrendim. Bu tuş kombinasyonu sayesinde açık olan uygulama kapatılabiliyordu. Artık uygulama kodlarımı düzenleyebilir ve Frame penceresinin istediğim boyutlarda oluşturulmasını sağlayabilirdim. Bu amaçla kodlarıma aşağıdaki satırı ekledim. Burada ilk parametre Frame penceresinin genişliğini (width), ikinci parametres ise yüksekliğini (height) belirtmekteydi.

```
pencere.setSize(300,100);
```

Uygulamayı bu haliyle derleyip çalıştırdığımda 300 piksel genişliğinde ve 100 piksel yüksekliğinde bir Frame penceresi elde ettim. Artık hem Title görünüyordu, hemde Frame penceresi daha makul boyutlardaydı.



Kaynaklardan Frame ile ilgili olarak kullanabileceğim diğer teknikleride araştırmaya başladım. Örneğin, X butonunun aksine, Minimize ve Maksimize butonları çalışıyor dolayısıyla Frame penceresi minimize edilebiliyor yada maksimize olabiliyordu. Derken aklıma, bu Frame' in Maksimize edilmek istendiğinde, belirli yükseklik ve genişliğin üstüne çıkmamasını nasıl sağlayabileceğim sorusu geldi. Bunun için `setMaximizedBounds()` isimli bir metod buldum. Bu Frame sınıfına ait metoda `Rectangle` sınıfı türünden bir nesne parametre olarak aktarılabilirdi. Bu `Rectangle` nesnesi, bir dörtgen şeklini boyutları ve konumları ile bildirebildiğinden, `setMaximizedBounds` metodu sayesinde, Frame penceresi belirtilen `Rectangle` nesnesinin boyutları kadar büyüyecekti. Hemen bu durumu analiz etmek amacıyla uygulama kodlarını aşağıdaki gibi geliştirdim.

```
Rectangle r=new Rectangle(500,500);
pencere.setMaximizedBounds(r);
```

Burada `Rectangle` sınıfından nesne örneğini oluştururken, parametre olarak genişlik ve yüksekliği bildirdim. İlk parametre `Rectangle` nesnesinin genişliğini, ikinci parametre ise yüksekliğini belirtmekteydi. Daha sonra, `setMaximizedBounds` metoduna, bu `Rectangle` nesnesini parametre olarak verdim. Uygulamayı tekrar derleyip çalıştırdığımda ve Maksimize butonuna bastığımda, Frame' in 500 piksel X 500 piksel boyutlarına geldiğini gördüm. Normal şartlar altında bu metodu kullanmasaydım, Frame tüm ekranı kaplayacak şekilde boyutlandırılacaktı.

Frame pencereleri ile ilgili aklıma takılan bir diğer nokta ise, X butonu ile pencereyi kapatamayışımı. Bunu kendim programlamam gerekiyordu. Bir başka deyişle, olay metodunu yazmalıydım. Kaynaklarımı araştırdığımda, Java Frame sınıfının aşağıdaki window olay metodlarına cevap verebildiğini öğrendim.

| Frame için Window Olayları |  |                |
|----------------------------|--|----------------|
| windowOpened               | Pencere ilk kez gösterildiğinde çalışan olay.                              | Window         |
| windowClosing              | Pencere kullanıcı tarafından kapatılırken gerçekleşen olay.                |                |
| windowClosed               | Pencere kapatıldıktan sonra çalışan olay.                                  |                |
| windowIconified            | Pencere minimize edildiğinde gerçekleşen olay.                             |                |
| windowDeiconified          | Minimize olan bir Pencere normal haline döndüğünde gerçekleşen olay.       |                |
| windowActivated            | Pencereye odaklanıldığı (Focus) yani aktifleştirildiği zaman çalışan olay. |                |
| windowDeactivated          | Pencereden ayrıldığında çalışan olay.                                      | Window Arayüzü |
| windowLostFocus            | Focus (odak) pencereden uzaklaştığında çalışan olay.                       |                |

|                    |  |                                 |
|--------------------|--|---------------------------------|
| windowGainedFocus  | Odak (Focus) pencereye geldiğinde çalışan olay.  |                                 |
| windowStateChanged | Pencerenin durumu değiştiğinde (minimize edildiğinde, maksimize edildiğinde vb.) çalışan olay. | WindowStateListener Arayüzünden |

İlk olarak denemek istediğim, pencerenin X butonu ile kapatılabilmesiydi. Öncelikle, windowClosing metodunu uygulamam gerekiyordu. Bunu gerçekleştirebilmek için, WindowListener arayüzünü sınıfa uygulamalıyım. Böylece, WindowListener arayüzünden uyguladığım windowClosing metodunda yazabilir ve X butonu ile pencerenin kapatılması sırasında oluşacak olayı kodlayabilirdim. Bu amaçla sınıf kodlarını aşağıdaki gibi geliştirdim.

```
import java.awt.*;
import java.awt.event.*;

public class IlkPencere implements WindowListener
{
    public static void main(String args[])
    {
        IlkPencere p=new IlkPencere();
        Frame pencere=new Frame("ILK PENCEREM");
        pencere.setLocation(0,0);
        pencere.setBackground(Color.red);
        pencere.setSize(300,100);
        Rectangle r=new Rectangle(500,500);
        pencere.setMaximizedBounds(r);
        pencere.addWindowListener(p);
        pencere.setVisible(true);
    }

    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }
}
```

Programı bu haliyle derlediğimde aşağıdaki hata mesajını aldım.

```

C:\ Command Prompt

E:\JavaSamples\GUI>javac IlkPencere.java
IlkPencere.java:4: IlkPencere should be declared abstract; it does not defi
ndowOpened(java.awt.event.WindowEvent) in IlkPencere
public class IlkPencere implements WindowListener
^
1 error
E:\JavaSamples\GUI>_

```

Anladığım kadarı ile WindowListener arayüzündeki tüm window olay metodlarını sınıf içerisinde kullanmasamda bildirmeliydim. Bu amaçla sınıfa aşağıdaki metodları da

ekledim.

```
public void windowOpened(WindowEvent e)
{
}
public void windowClosed(WindowEvent e)
{
}
public void windowIconified(WindowEvent e)
{
}
public void windowDeiconified(WindowEvent e)
{
}
public void windowActivated(WindowEvent e)
{
}
public void windowDeactivated(WindowEvent e)
{
}
```

Uygulama başarılı bir şekilde derlendikten sonra, hemen X butonu ile kapatılıp kapatılamadığını denemedim. Sonuç başarılıydı. Elbetteki bir pencere bu haliyle çok yavan durmaktaydı. Bu pencereye kontroller eklemek gerekiyordu. Normal bir Applet'e kontroller nasıl ekleniyorsa buradada aynı kurallar geçerliydi. Bu kez bir Applet'e kontrol eklemek yerine bir Frame nesnesine kontrol ekleyecektim. Bu amaçla uygulamayı biraz daha düzenlemeye ve ilginç hale getirmeye karar verdim. Amacım Frame içindeki bir button yardımıyla başka bir frame penceresinin açılabilmesini sağlamaktı. Bu amaçla aşağıdaki örneği oluşturdum.

```
import java.awt.*;
import java.awt.event.*;

public class IlkPencere implements WindowListener, ActionListener
{
    public Frame p1;
    public Button btnIkinciPencere;
    public Button btnKapat;
    public int X;
    public int Y;

    public void PencereAyarla(String baslik,int genislik,int yukseklik,int konumX, int
konumY,Color
arkaPlanrengi)
    {
        X=konumX;
        Y=konumY;

        p1=new Frame(baslik);
        p1.setLocation(konumX,konumY);
        p1.setBackground(arkaPlanrengi);
        p1.setSize(genislik,yukseklik);
        p1.setLayout(new FlowLayout());
    }
}
```



```

        p1.addWindowListener(this);

        btnIkinciPencere= new Button("Ikinci Pencere");
        btnKapat=new Button("Kapat");

        p1.add(btnIkinciPencere);
        p1.add(btnKapat);

        btnKapat.addActionListener(this);
        btnIkinciPencere.addActionListener(this);

        p1.setVisible(true);
    }

    public static void main(String args[])
    {
        IlkPencere p=new IlkPencere();
        p.PencereAyarla("ANA PENCERE",250,100,0,0,Color.white);
    }

    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource()==btnKapat)
        {
            p1.setVisible(false);
        }
        else if(e.getSource()==btnIkinciPencere)
        {
            IlkPencere p=new IlkPencere();
            X=X+50;
            Y=Y+50;
            p.PencereAyarla("ANA PENCERE",100,100,X,Y,Color.red);
        }
    }

    public void windowClosing(WindowEvent e)
    {
        System.exit(0);
    }

    public void windowOpened(WindowEvent e)
    {
    }

    public void windowClosed(WindowEvent e)
    {
    }

    public void windowIconified(WindowEvent e)
    {
    }

    public void windowDeiconified(WindowEvent e)
    {
    }

    public void windowActivated(WindowEvent e)
    {
    }

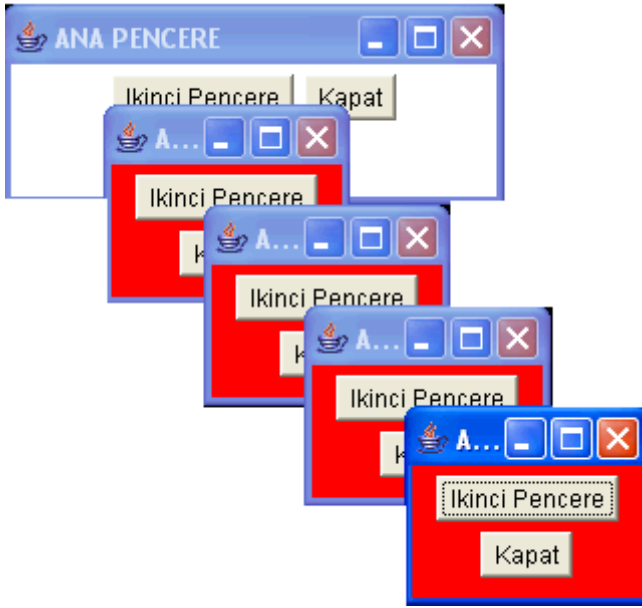
    public void windowDeactivated(WindowEvent e)

```

```
{  
}  
}
```

Bu uzayıp giden kodlar çok işe yaramıyor. Ancak şu ana kadar GUI 'ler ile ilgili bilgilerimi tekrar etmemede yardımcı oldu. Bu uygulama çalıştığında ilk olarak belirtilen boyutlarda, konumda, başlıkta ve art alan renginde bir ana pencere oluşturuyor. Bu pencere üzerine, FlowLayout sınıfının öngördüğü Layout düzenine göre yerleşen iki Button bileşenim var. İkinci Pencere başlıklı button bileşenine tıklandığında yeni bir pencere oluşturuluyor. Kapat button bileşeni ise, bu pencereyi kapatıyor. Bu kapatma işleminde setVisible(false) metodunu kullandım. Böylece sonradan açılan pencereler aslında gizleniyordu.

Uygulamayı bu haliyle derleyip çalıştırdığımda aşağıdaki gibi bir görüntü oluştu. Her yeni pencere bir öncekinin konumunun 50 birim sağına ve altına konumlandırılıyor. Elbette X butonuna basıldığında System.exit(0) metodu o an çalışan prosesi sonlandırdığı için tüm pencereler kapanmaktaydı. Mesela ilk pencerede Kapat başlıklı butona basınca komut satırı açık kalacak şekilde pencere ortadan kayboluyor. Yani görünmez oluyor. Ancak proses çalışmaya devam ediyor. Sanırım neden işe yaramaz bir program olduğu ortada. Olsun en azından el cimmastipi yapmış oldum.



Şu anada kadar yaptıklarım ile geliştirdiğim bu pencere uygulamalarında önemli bir sorun var aslında. Bu uygulamaları çalıştırabilmek için komut satırında ilgili sınıfı java yorumlayıcısı ile açmam gerekiyor. Diğer taraftan uygulama çalışırken, komut satırı açık kalıyor. Oysaki normal bir exe dosyası gibi bu uygulamanın tek başına çalışabilmesi çok daha yerinde olur. İşte bunu gerçekleştirmek için kaynaklarda 3 yoldan bahsedildiğini öğrendim. En basit olanı üçüncü parti yazılımlar ile bu işi gerçekleştirmek. Örneğin halen daha özlemimi çektiğim değerli arkadaşımın bilgisayarımda yer alan JBuilder gibi.

Diğer iki yol ise bizim manuel olarak kullanabileceğimiz teknikler içeriyor. Bunlardan birisi Dos ortamından kalma bat(batch) uzantılı dosyalar içerisine uygulamayı çalıştıracak kod satırını yazmak. Diğeri ise, GUI uygulamasına atı tüm sınıfları ve gerekli dosyaları içeren bir JAR paketi oluşturmak. Açıkçası JAR paketini oluşturmak bana daha mantıklı göründü. Ancak bir JAR paketini oluşturmadan önce, bu JAR

paketi için versiyon numarası, ana sınıf gibi bilgileri içeren bir manifesto dosyası hazırlamam gerektiğini öğrendim. Bu manifesto dosyası, mf uzantılı olmakla birlikte, aslında .net assembly' larındaki manifesto bilgilerinin tutulduğu yapıya benzer bir içeriğe sahip. Çok basit olarak geliştirdiğim java uygulaması için aşağıdaki bilgileri içeren bir manifesto dökümanı hazırladım.

```
Manifest-Version: 1.0
Main-Class: IlkPencere
Created-By: 1.4.1 (Sun Microsystems Inc.)
```

Bu dosyayı Manifesto.mf ile kaydettikten sonra aşağıdaki komut ile, Jar dosyasını oluşturdum.

```
Command Prompt

E:\JavaSamples\GUI>jar cvfm Pencere.jar Manifesto.mf IlkPencere.class
added manifest
adding: IlkPencere.class(in = 2256) (out= 1133)(deflated 49%)

E:\JavaSamples\GUI>_
```



Artık Pencere.jar dosyasına çift tıkladığımda GUI uygulamasının, normal bir windows uygulaması gibi çalıştığını gördüm. Bu sorunu çözmem son derece önemli idi. Artık windows tabanlı GUI' lerin nasıl oluşturulduğunu, window olaylarına nasıl cevap verdiğini biliyordum. Dahası bu pencereler üzerine awt bileşenlerinin nasıl ekleneceğini ve herşeyden önemlisi bu GUI uygulamasının çift tıklamalı versiyonunun Jar dosyası olarak nasıl hazırlanabileceğini biliyordum. Artık tüm bu bildiklerimi birleştirerek daha işe yarar bir uygulama yapabileceğim kanısındaydım. Bunun için aklıma basit bir hesap makinesi uygulması yazmak geldi. Ama çok basit. Sadece 2 operand değeri için 4 işlem yapacaktı. Lakin burada önemli olan, Frame' in tasarlanması ve Frame üzerindeki bileşenlerin olaylara tepki vermesinin sağlanmasıydı. Hemen kolları sıvadım ve uygulamayı geliştirmeye başladım. Sonuçta hem pratik yapmış oldum hemde GUI bilgilerimi tekrar etmiş. Sonuçta aşağıdaki küçük programcık ortaya çıktı.

```
import java.awt.*;
import java.awt.event.*;

/* HesapMakinesi sınıfında window olaylarına ve Button olaylarına izin verebilmek
için, WindowListener ve ActionListener arayüzlerinin uygulanması gerekir. */
public class HesapMakinesi implements WindowListener,ActionListener
{
```

```

    /* Frame sınıfına ait nesne tanımlanıyor ve bu Frame üzerindeki awt bileşenleri
    tanımlanıyor.*/
    public Frame f;
    public Button btnHesapla;
    public Label lbSayi1;
    public Label lbSayi2;
    public Label lbIslem;
    public TextField tfSayi1;
    public TextField tfSayi2;
    public Choice lstIslem;

    /* iki sayı değerini ve işlem sonucunu tutacak double tipinden değişkenler
    tanımlanıyor.*/
    public double sayi1,sayi2,sonuc;

    /* Olustur metodunda, penceremiz ve üzerindeki bileşenler oluşturuluyor.*/
    public void Olustur()
    {
        f=new Frame("Hesap Makinesi"); // Başlığı (Title) Hesap Makinesi olan bir
        Frame nesnesi oluşturuluyor.
        f.setLayout(new FlowLayout()); // Frame üzerindeki bileşenler FlowLayout
        tekniğine göre dizilecekler.
        Color c=new Color(248,221,139); /* Color tipinden bir nesne R (Red), G
        (Green), B(Blue) formatında oluşturuluyor.*/
        f.setBackground(c); // Pencerenin arka plan rengi c isimli Color nesnesine
        göre belirleniyor.

        /* TextField bileşenleri 10 karakter uzunluğunda oluşturuluyor.*/
        tfSayi1=new TextField(10);
        tfSayi2=new TextField(10);

        /* Label bileşenleri başlıkları ile oluşturuluyor.*/
        lbSayi1=new Label("Sayı 1");
        lbSayi2=new Label("Sayı 2");
        lbIslem=new Label("ISLEMIN SONUCU...");

        /* Button bileşeni oluşturuluyor ve bu bileşen için olay dinleyicisi ekleniyor.*/
        btnHesapla=new Button("Hesapla");
        btnHesapla.addActionListener(this);

        /* Choice (başka bir deyişle ComboBox) bileşeni oluşturuluyor. Listedeki
        elemanlar addItem metodu ile ekleniyor.*/
        lstIslem=new Choice();
        lstIslem.addItem("TOPLA");
        lstIslem.addItem("CIKART");
        lstIslem.addItem("BOL");
        lstIslem.addItem("CARP");

        /* Bileşenler sırasıyla Frame bileşenine yani pencereye add metodu ile
        ekleniyor. */
        f.add(lbSayi1);
        f.add(tfSayi1);
        f.add(lstIslem);
        f.add(lbSayi2);
    }

```

```

        f.add(tfSayi2);
        f.add(btnHesapla);
        f.add(lbIslem);
        f.pack(); /* pack metodu ile pencerenin yüksekliği ve genişliği, içerdiği
bileşenlerin kapladığı alana göre otomatik olarak ayarlanıyor.*/
        f.addWindowListener(this); // Frame bileşeni için window olay dinleyicisi
ekleniyor.

        f.setVisible(true); // Frame bileşeni (pencere) gösteriliyor.
    }

    /* IslemYap metodunda 4 işlem gerçekleştiriliyor. */
    public void IslemYap()
    {
        sayi1=Double.parseDouble(tfSayi1.getText()); /* TextField bileşenlerinin
string içeriği Double sınıfının parseDouble metodu ile double tipine dönüştürülerek
değişkene atanıyor.*/
        sayi2=Double.parseDouble(tfSayi2.getText());

        /* if koşullarında Choice bileşeninde seçili olan item getSelectedItem()
metodu ile alınıyor ve uygun olan işlemler yapılıyor.*/
        if(lstIslem.getSelectedItem()=="TOPLA")
        {
            sonuc=sayi1+sayi2;
            lbIslem.setText(sayi1+"+"+sayi2+"="+sonuc);
        }
        else if(lstIslem.getSelectedItem()=="CARP")
        {
            sonuc=sayi1*sayi2;
            lbIslem.setText(sayi1+"x"+sayi2+"="+sonuc);
        }
        else if(lstIslem.getSelectedItem()=="CIKART")
        {
            sonuc=sayi1-sayi2;
            lbIslem.setText(sayi1+"-"+sayi2+"="+sonuc);
        }
        else if(lstIslem.getSelectedItem()=="BOL")
        {
            sonuc=sayi1/sayi2;
            lbIslem.setText(sayi1+"/"+sayi2+"="+sonuc);
        }
    }

    public static void main(String args[])
    {
        HesapMakinesi m=new HesapMakinesi();
        m.Olustur();
    }
    /* actionPerformed olayı meydana geldiğinde bu metod çalışıyor.*/
    public void actionPerformed(ActionEvent e)
    {
        if(e.getSource()==btnHesapla) // Eğer olayın kaynağı Button bileşeni ise, yani
Button' a tıklandıysa.
        {

```

```

        IslemYap();
    }
}

/* Kullanıcı X buton ile pencereyi kapatmak istediğinde bu olay metodu çalışıyor.
*/
public void windowClosing(WindowEvent e)
{
    System.exit(0); /* Güncel olan proses sonlandırılıyor. Dolayısıyla uygulama
sona eriyor. */
}
public void windowOpened(WindowEvent e)
{
}
}
public void windowClosed(WindowEvent e)
{
}
}
public void windowIconified(WindowEvent e)
{
}
}
public void windowDeiconified(WindowEvent e)
{
}
}
public void windowActivated(WindowEvent e)
{
}
}
public void windowDeactivated(WindowEvent e)
{
}
}
}

```

Programı derledikten sonra Manifesto dosyasını (ManifestoHesapMakinesi.mf) aşağıdaki gibi düzenledikten sonra, JAR Paketini de hazırladım.

```

Manifest-Version: 1.0
Main-Class: HesapMakinesi
Created-By: 1.4.1 (Sun Microsystems Inc.)

```

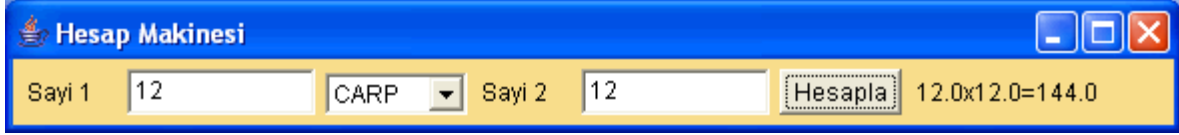
```

C:\> Command Prompt

E:\JavaSamples\GUI>jar cvfm HM.jar ManifestoHesapMakinesi.mf HesapMakinesi.
added manifest
adding: HesapMakinesi.class(in = 3458) (out= 1668)(deflated 51%)
E:\JavaSamples\GUI>_

```

Paketi çift tıkladığımda, basit hesap makinesi uygulamam kullanılmaya hazırды.



Artık GUI'lerde iyice ilerlemeye başladığımı hissediyordum. Bununla birlikte, 2 boyutlu grafik çizimleri, animasyon hazırlamak, resim işlemek, ses işlemek, Swing bileşenleri, Menu'ler vs... gibi henüz bilmediğim daha pek çok konu vardı. Ancak hem kahvem hemde pilim bitmişti. Sanırım önümüzdeki günlerde, bu konulara eğileceğim.

Burak Selim ŞENYURT

[selim@bsenyurt.com](mailto:selim@bsenyurt.com)

Yazar: Burak Selim Şenyurt

## Grafik Çizimi

### Graphics Class'ı

Ekranın belli bir yerine şekil, yazı veya image koymak için Graphics class'ı kullanılır. Bu class Component'teki

```
public void paint(Graphics g);
```

şeklindeki methodundan elde edilerek kullanılabilir. Java değişik işletim sistemlerinde ve aynı işletim sisteminin değişik versiyonlarında çalışabildiğinden, Graphics class'ı da abstract'tır. paint() method'undan aslında her işletim sistemi için ayrı olarak yazılmış Graphics class'ının bir subclass'ı elde edilir. Ancak programcıyı bu hiç ilgilendirmez, çünkü bütün subclass'lar Graphics class'ındaki bütün methodları imlement etmişlerdir. Bazı dillerde 'device context' diye de adlandırılan, belli bir dikdörtgende çizim yapılmasını sağlayan class Graphics'dir. Bu dikdörtgen ekranda mutlak bir yeri değil, görelî bir bölümü temsil etmektedir. Bir pencere içerisine çizim yapıldığında, pencerenin ekranın neresinde olduğu bilinmez. Nokta olarak x=100 ve y=200 verildiğinde ekranda pencerenin başlangıç konumundan 100'e 200 uzaklıkta bir nokta kastedilmektedir. Sadece pencere için değil, bütün component'lerin paint() method'larında elde edilen Graphics nesneleri o component'e ait alana göre bir koordinat sistemine sahiptir. P(0,0) noktası o component'in başlangıç noktasıdır, ekranın ilk noktası değil. Zaten bir component'in kendi dışındaki bir alana çizmesi mümkün değildir. Koordinatları alanından fazla verme veya negatif değerler verme gibi durumlarda çizimin sadece component içerisinde kalan kısmı yapılır, dışındaki kısım clip edilir (kırılır).

### Graphics Class'ındaki Çizim Method'ları

Graphics class'ı çizim yapmak içine çeşitli methodlar içermektedir. Çizim methodlarını içeren başka bir class zaten yoktur. En çok kullanılanları şunlardır.

```
drawLine()  
drawRect()  
drawArc()  
drawImage()
```

Bazı drawXxx() methodlarının fillXxx() karşılıkları vardır. drawXxx()'ler boş bir şekil çizerken fillXxx() methodları içi dolu bir şekil çizer.

### Color (Renk) Belirleme

Çizim rengi, bir şekli çizmeden önce setColor() methoduyla belirlenir. Background rengini belirlemek için de setBackground() methodu bulunmakdadır. Bu iki method da parametre olarak Color adlı class'ın instance'larını alırlar. Color, ARGB (Alpha, Red, Green, Blue) sistemiyle çalışır. Alpha değeri çizimin opaque veya transparent olmasını belirleyen sayıdır. 0 transparent, 0xFF da opaque demektir. Her renk RGB bileşenlerinin kombinasyonu ile üretilir. Örneğin yellow (sarı), red ve green bileşenlerinin tam, blue bileşeninin sıfır olması durumunda üretilir.

Color class'ının constructor'larının ve ARGB sisteminin karmaşıklığı nedeniyle, çok kullanılan renkler bu class içerisinde sabit (final static) olarak verilmiştir. Bu nedenler

```
g.setColor(new Color(0xFF,0xFF,0x00));
```

yerine

```
g.setColor(Color.yellow);
```

kullanılabilir.

### Yazı 'Çizmek'

Graphics class'ındaki drawString() methodu ekrana bir yazı 'çizmek' için kullanılır. Adından da anlaşılacağı gibi yazı herhangi bir şekil gibi 'çizilir'.

Bir yazının font'unu belirlemek için setFont() methodu kullanılır. Bu methodun kabul ettiği parametre tipi Font class'ıdır ve bu class isim, style ve size değerlerini tutar. Style düz, italik veya bold olmasını belirler. Belirtilen isimde bir font yoksa default font kullanılır.

## Java 24 Bölüm 22: Java ile Grafik Çizim

Geçtiğimiz hafta boyunca, Java dili ile fazla ilgilenemedim. Nitekim vaktimin büyük çoğunluğunu Whidbey' i incelemekle geçirmiştım. Aslında yazın bu sıcak dönemlerinde, beni şöyle rahatalecek, fazla terleymeyecek çalışmalar yapmak istiyordum. Whidbey beni bir nebze olsa rahatlatırsa, klimanın verdiği ferahlığı sağlayamamıştı. Bana biraz eğlenceli ve eğlenceli olduğu kadarda işe yarayacak bir konu gerekiyordu. Sonunda, Java programlama dili ile, grafiksel çizimlerin nasıl yapıldığını araştırmaya karar verdim. Zor olmayan, sıkıcı olmayan hatta zaman zaman işe yarar bir şekil oluşturabilmek için eski matematik bilgilerimi hatırlamama yol açan bu konu benim için yeteri kadar eğlenceli ve güzeldi.

Elbette, bir programlama dili ne kadar güçlü olursa olsun, sağladığı grafiksel kütüphanelerin kabiliyetleri, sıradan bir tasarım programının yerini tutamazdı. Ancak insan durup düşündüğünde, bu tip grafik programlarının oluşturulmasında java, C# gibi dillerin kullanılabileceğini kolaylıkla anlayabilir. Sonuç olarak, bir grafik programında mouse ile, toolbar' dan seçtiğimiz bir şekli kolayca oluşturabiliriz. Mouse ile sürükleme bir olaydır. Seçilen şekle göre ekranda bir vektör grafiğın oluşmasında, dilin sağladığı grafik kütüphaneler ile mümkün olabilir. Olayı dahada sofistike düşündüğümde, C# ile veya Java ile yazılmış, haritacılık, şehir planlama gibi programların olduğunu da biliyordum. Hatta böyle bir programı iş başındaykenden inceleme fırsatı bulmuştım.

Sonuçta, eğlenceli olan grafik nesneleri aslında büyük çaplı projelerde temel yapı taşları olarak rol alabilirlerdi. Kendimi bu düşünceler eşliğinde gaza getirdikten

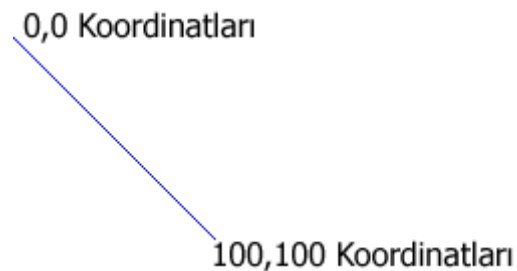


sonra, şöyle ağız tadıyla web sayfasına bir çizik atayım dedim. Yapacağım işlemler son derece kolaydı. Bir applet tasarlayacak ve bu sayede bir web sayfasına, çeşitli grafik metodları yardımıyla vektör şekiller çizebilecektim. Bunun için aşağıdaki gibi bir java kaynak kod dosyasını oluşturdum.

```
import java.awt.*;
import java.applet.Applet;

public class Grafikler extends Applet
{
    public void paint(Graphics g)
    {
        g.setColor(Color.BLUE);
        g.drawLine(0,0,100,100);
    }
}
```

Ekrana bir çizgi çizmek için, Applet' in paint metodunu kullandım. Burada, çizgiyi Graphics sınıfının, drawLine metodu ile oluşturdum. Metod 4 parametre alıyordu. İlk ikisi x ve y koordinatlarını, son ikisi ise, çizginin bittiği yerin x ve y koordinatlarını veriyordu. Ayrıca, çizgiyi mavi renkte boyamak istediğimden, Graphics nesnesine setColor metodu ile Color numaralandırıcısından BLUE değerini atadım. Sonuç aşağıdaki gibiydi.

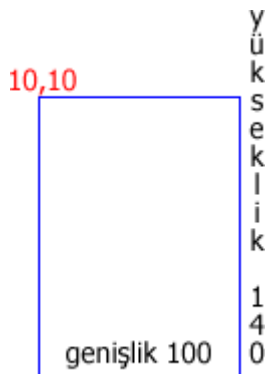


Elbette Graphics sınıfından çizebileceğim şekiller sadece basit bir çizgiden ibaret olamazdı. Daha pek çok şekil vardı. İçi dolu olanlar veya içi boş olanlar gibi. O halde yapmam gereken ortadaydı. Çizebileceğim her şekli deneyecektim ve sonuçlarını görecektim. Bu aynı zamanda benim için bir rehber olacaktı. Çizim kodu ve altında şeklin görüntüsü. Hemen parmakları sıvadım ve klavyemin tuşlarını aşındırmaya başladım. Öncelikle temel şekillerle işe başlama taraftarıyım. Yani dörtgenlerden.

```
import java.awt.*;
import java.applet.Applet;

public class Grafikler extends Applet
{
    public void paint(Graphics g)
    {
        g.setColor(Color.BLUE);
        g.setFont(new Font("Verdana",Font.BOLD,12));
        g.drawString("ICI BOS DORTGEN",0,165);
    }
}
```

```
g.drawRect(10,10,100,140);  
}  
}
```



**ICI BOS DORTGEN**

drawRect metodu 4 parametre almaktaydı. İlk ikisi, dörtgenin yerleştirileceği X ve Y koordinatlarını belirtirken, sonraki iki parametrede genişlik ve yüksekliği belirtmekteydi. Bir dörtgenin içinin dolu olmasını veya kenarlarının yuvarlatılmış olmasını isteyebilirdik. Bunun içinde değişik metodlar vardı. Örneğin fill ile başlayan metodlar içi dolu şekiller oluştururken, draw ile başlayanlar içi boş şekiller oluşturmaktaydı. Hemen, diğer olası dörtgen şekillerinide çizmeye çalıştım ve aşağıdaki örnek kodları oluşturdum.

```
g.fillRect(10,10,100,140);
```



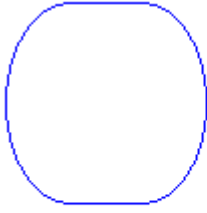
**ICI DOLU DORTGEN**

```
g.fillRoundRect(10,10,100,100,16,16);
```



**ICI DOLU VE KENARLARI YUVARLATILMIS DORTGEN**

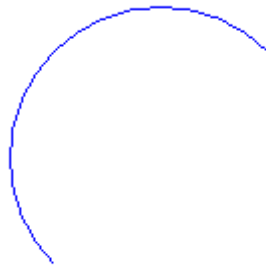
```
g.drawRoundRect(10,10,100,100,64,128);
```



#### ICI BOS VE KENARLARI YUVARLATILMIS DORTGEN

Round tipteki dörtgenlerin kenarları yuvarlatılırken son iki parametre kullanılmakta. Bunlardan ilki, yayın genişliğinin piksel cinsinden değerini belirtirken ikinciside yayın yüksekliğini belirtiyor. Dörtgenlerden sonra sıra yaylara gelmişti. Lisans eğitimim sırasında, yaylarla epeyce haşırneşir olmuştum. Ancak genelde yayları elle kağıtlara çizmek yerine, yaylar ile ilgili teoremlerin ispatlarının yapılması ile uğraşmıştım. Sırf bu nedenle, ilk ve orta okulda çok iyi olan resim çizme kabiliyetimin yok olduğunu söyleyebilirim. Nitekim, artık şekil çizmeye çalıştığımda o şekle matematiksel bir gözle bakıyor ve olabilecek çeşitli ispatları ve teoremleri hatırlıyorum. Neyse, konuyu ve kafayı fazla bulandırmadan yay çizme işlemine girişsek hiç fena olmaz sanırım.

```
g.drawArc(10,10,150,150,45,180);
```



YAYLAR...

```
g.fillArc(10,10,150,150,45,45);
```



YAYLAR...

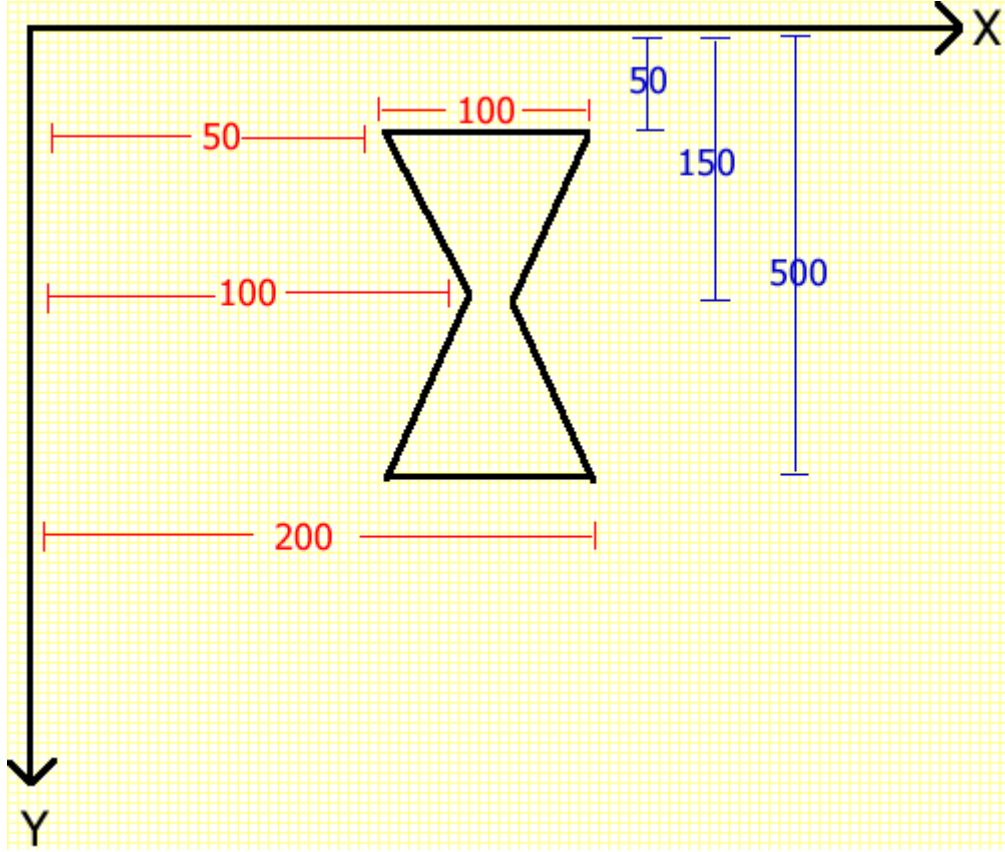
Yayların çizimindeki en önemli nokta son iki parametreydi. Bu parametrelerden ilki, başlangıç açısını derece cinsinden belirtirken, ikinci parametre yayın oluşturacağı açıyı derece cinsinden belirtmekteydi. Tabiki, içi dolu yay aslında bir daire diliminden başka bir şey olmamaktaydı. Ancak daireleri çizmek içinde başka metodlar vardı. Daireler oval şekillerin çizildiği fillOval yada drawOval metodları ile elde edilebilirdi. Daire olması için, genişlik ve yükseklik değerlerinin eşit olması yeterliydi. Bu şekilleride aşağıdaki kod satırları ile test ettim.

```
g.setColor(Color.BLACK);  
g.setFont(new Font("Verdana",Font.BOLD,12));  
g.drawString("YAYLAR...",0,165);  
g.fillOval(70,100,80,18);  
g.drawOval(30,30,75,75);
```



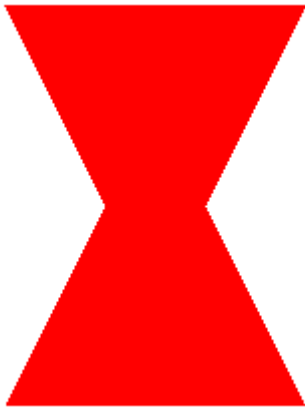
**YAYLAR...**

Sırada daha komplike bir şekil olan poligonlar vardı. Bir poligon çizebilmek için, drawPolygon yada fillPolygon metodlarından birisini kullanabilirdim. Aralarındaki tek fark birisinin içinin dolu ötekisinin ise boş oluşuydu. Hangileri tahmin edin bakalım :) Elbette bir poligon oluşturabilmek için bir takım verilere ihtiyacım vardı. Herşeyden önce poligonların köşe sayıları belli değildi. Dolayısıyla bu köşelerin x ve y koordinatlarını belirleyecek iki integer diziye ihtiyacım olacaktı. Bu dizilerden birisi, x koordinatlarını diğeri ise y koordinatlarını taşımalıydı. İşte bu anda, kalem kağıda sarıldım ve acaba bir kum saatinin sembolik resmini çizebilir miyim diye düşünmeye başladım. Oturup ciddi ciddi, kağıt üzerinde, bir kum saatinin iki boyutlu görüntüsüne ait köşe koordinatlarını, ekranın ordinat sistemine göre çıkarmaya çalıştım. İşte kağıttaki çalışmamın güzelim Fireworks grafik programı ile şematize edilişi.



Sonuç olarak aşağıdaki kodlar bana izleyen şekli verdi. Ehhh işte. Biraz da olsa kum saatini andırıyor.

```
int xKoordinatlari[]={50,200,150,200,50,100};  
int yKoordinatlari[]={50,50,150,250,250,150};  
g.fillPolygon(xKoordinatlari,yKoordinatlari,6);
```



Artık tüm şekilleri öğrendiğime göre, bunları harmanlayıp beni daha çok neşelendirecek bir uygulama yazabilirdim. Örneğin, hepimizin kağıda çizmekte hiç zorlanmayacağı, hatta patatesler ve bir kaç kibrit çöpü ile birlikte yapabileceği bir çöp adamı, Java' daki grafik metodlarını kullanarak çizmek ne kadar zor olabilirdi ki? Doğruyu söylemek gerekirse o basit çöp adamı oluşturabilmek için bir süre

çalışmam gerekti. Kağıt üzerinde matematiksel olarak koordinatların belirlenmesi, hangi şeklin nereye geleceği derken saat sabahın üçü oluvermişti bile. Ancak bir kaç dakikalık bu eziyet sonrasında aşağıdaki program kodları sayesinde oldukça ilginç bir eser ortaya çıkartabilmişim. Belki Da Vinci kadar iyi değildi. Ancak yinede modern sanatın bir görüntüsüydü.

```
import java.awt.*;
import java.applet.Applet;

public class Grafikler extends Applet
{
    public void paint(Graphics g)
    {
        g.setColor(Color.RED);
        g.setFont(new Font("Verdana",Font.BOLD,12));
        g.drawArc(20,10,20,20,45,75);
        g.drawArc(50,10,20,20,45,75);

        g.setColor(Color.blue);
        g.drawLine(20,1,20,6);
        g.drawLine(25,1,25,7);
        g.drawLine(30,1,30,7);
        g.drawLine(35,1,35,8);
        g.drawLine(40,1,40,8);
        g.drawLine(45,1,45,9);
        g.drawLine(50,1,50,8);
        g.drawLine(55,1,55,8);
        g.drawLine(60,1,60,7);
        g.drawLine(65,1,65,7);
        g.drawLine(70,1,70,6);

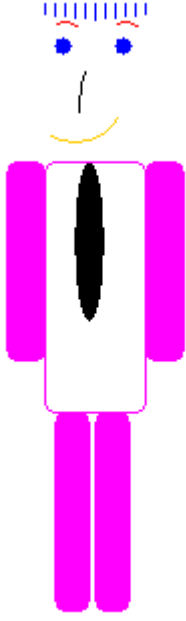
        g.fillOval(25,18,9,9);
        g.fillOval(55,18,9,9);

        g.setColor(Color.BLACK);
        g.drawArc(37,30,15,45,130,60);

        g.setColor(Color.orange);
        g.drawArc(10,20,50,50,-120,90);

        g.setColor(Color.black);
        g.fillOval(35,80,15,80);

        g.setColor(Color.MAGENTA);
        g.fillRoundRect(1,80,20,100,10,10);
        g.drawRoundRect(20,80,50,125,10,10);
        g.fillRoundRect(70,80,20,100,10,10);
        g.fillRoundRect(25,205,18,100,10,10);
        g.fillRoundRect(45,205,18,100,10,10);
    }
}
```



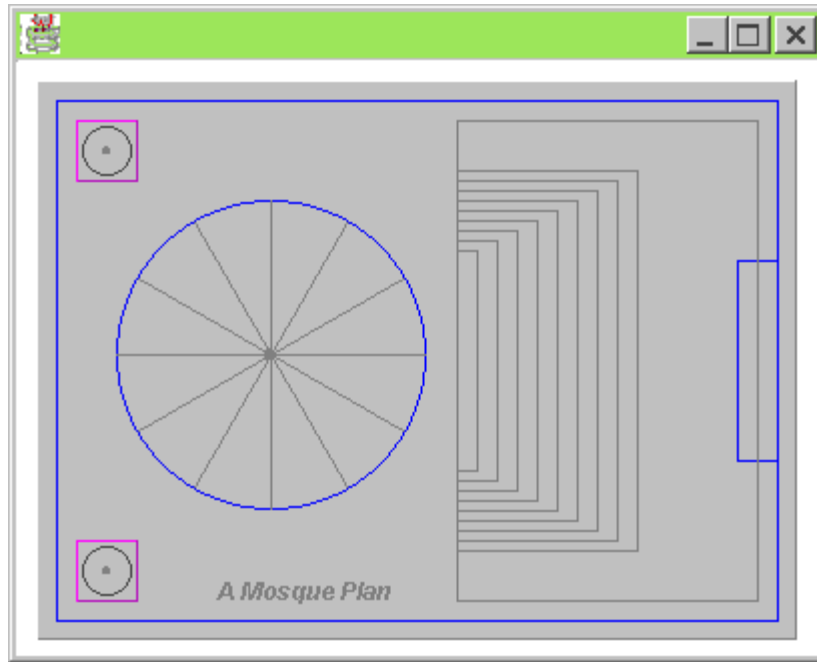
Şimdi gelde, şu grafik programlarını, harita programlarını nasıl yazıyorlar düşün. Düşünmeden edemedim. Allah bu tip grafik tabanlı yazılımları Java veya C# gibi nesne yönelimli diller ile geliştirilenlere sabır versin. Bende bir kaç have finacanı ile ancak yukarıdaki kadar garip bir adam elde edebildim.

Burak Selim ŞENYURT

[selim@bsenyurt.com](mailto:selim@bsenyurt.com)

Yazar: Burak Selim Şenyurt

**Graphics Kullanım Örneği**



Graphics class'ına örnek olarak üzerine bir caminin krokisini çizen bir Canvas yapalım. Örnekte bütün çizimleri paint() methodunun içerisine yapılmayıp, her bölüm ayrı bir method'da yapılmıştır. Her method'da renkler yeniden belirlenmiştir. Aksi takdirde çizim bir önceki method'da belirlenen renge göre yapılırdı. En üste duracak çizim (örneğin yazı) en son çizilmiştir. Yoksa daha sonra çizilen bir şeklin altında kalabilirdi. drawArc method'u çizeceği yay ilişkin açıyı derece cinsinden alır. Math class'ındaki trigonometric methodlarsa açıyı radyan (pi) cinsinden alırlar.

```
import java.awt.*;
```

```
public class GraphicsSample
```

```
    extends Canvas
```

```
{
```

```
    public void paint(Graphics g){
```

```
        paintGround(g,0,0,400,300);
```

```
        paintGate(g,360,100,20,100);
```

```
        paintDome(g,50,70,77);
```

```
        paintStairs(g,220,150,100,5);
```

```
        paintMinaret(g,30,30,30,30);
```

```
        paintMinaret(g,30,240,30,30);
```

```
        paintCourt(g,220,30,150,240);
```

```
        paintLabel(g,"A Mosque Plan",100,270);
```

```
}
```

```
    private void paintLabel(Graphics g,String string,int x,int y){
```

```
        g.setColor(Color.darkGray);
```

```
        Font font=new Font("Arial",Font.ITALIC|Font.BOLD,12);
```

```
        g.setFont(font);
```

```
        g.drawString(string,x,y);
```



```

    }

    private void paintGround(Graphics g,int x,int y,int width,int
height){
        g.setColor(Color.lightGray);
        g.fill3DRect(x+10,y+10,width-20,height-20,true);
        g.setColor(Color.blue);
        g.drawRect(x+20,y+20,width-40,height-40);
    }

    private void paintGate(Graphics g,int x,int y,int width,int
height){
        g.setColor(Color.blue);
        g.drawRect(x,y,width,height);
    }

    private void paintDome(Graphics g,int x,int y,int radius){
        int xcenter=x+radius;
        int ycenter=y+radius;
        g.setColor(Color.blue);
        g.drawArc(x,y,radius*2,radius*2,0,360);
        g.setColor(Color.gray);
        g.fillArc(xcenter-3,ycenter-3,6,6,0,360);
        for(int angle=0;angle<360;angle+=30){
            paintAngle(g,xcenter,ycenter,angle,radius);
        }
    }

    private void paintStairs(Graphics g,int xcenter,int ycenter,int
ybase,int length){
        for(int number=1;number<10;number++){
            paintStair(g,xcenter,ycenter,ybase,length,number);
        }
    }

    private void paintStair(Graphics g,int xcenter,int ycenter,int
ybase,
        int length,int number)
    {
        int xstart=xcenter;
        int ystart=ycenter-ybase/2-length*number;
        int width=2*length*number;
        int height=ybase+2*length*number;
    }

```

```

        g.drawRect(xstart,ystart,width,height);
    }

    private void paintAngle(Graphics g,int xcenter,int ycenter,int
angle,int radius){
        double radian=(Math.PI*angle)/180;
        int xend=xcenter+(int) (radius*Math.cos(radian)) ;
        int yend=ycenter-(int) (radius*Math.sin(radian)) ;
        g.drawLine(xcenter,ycenter,xend,yend);
    }

    private void paintMinaret(Graphics g,int x,int y,int width,int
height){
        g.setColor(Color.magenta);
        g.draw3DRect(x,y,width,height,true);
        g.setColor(Color.black);
        g.drawArc(x+3,y+3,width-6,height-6,0,360);
        g.setColor(Color.gray);
        g.fillArc(x+width/2-2,y+height/2-2,4,4,0,360);
    }

    private void paintCourt(Graphics g,int x,int y,int width,int
height){
        g.setColor(Color.gray);
        g.drawRect(x,y,width,height);
    }

    public static void main(String[] args){
        Frame frame=new Frame();
        frame.setBounds(100,100,410,330);
        frame.setLayout(new BorderLayout());
        GraphicsSample sample=new GraphicsSample();
        frame.add(sample,BorderLayout.CENTER);
        frame.setVisible(true);
    }
}

```