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 indirgeyecektir. Uygulamalarımızda daha evvelden yazılmış ve doğruluğu kanıtlanmış olan sınıfları tekrardan kullanmanın iki yöntemi bulunur. (yorum ekle)
Birinci yöntem komposizyon'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 komposizyon yöntemini çokça kullandık. (yorum ekle)
İkinci yöntem ise kalıtımdır (*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. (yorum ekle)

5.1. Komposizyon

Komposizyon yönetimini daha önceki örneklerde kullanıldı. Şimdi bu yöntemin detaylarını hep beraber inceleyelim. (yorum ekle)

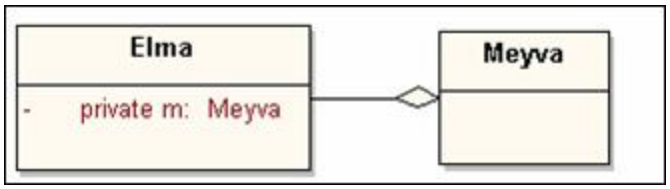
Gösterim-5.1:

```
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; (yorum ekle)



Şekil-5.1. Komposizyon-I

Başka bir örnek verilirse,

Örnek-5.1: *Motor.java* (yorum ekle)

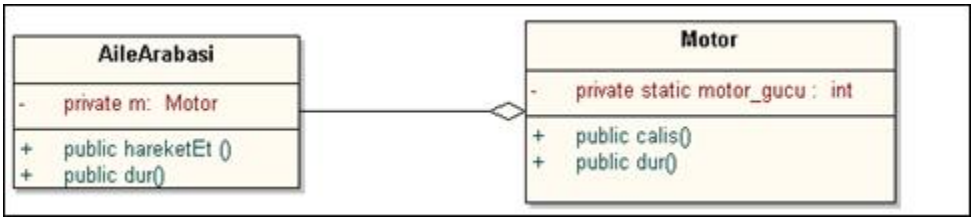
```
public class Motor {
     private static int motor_gucu = 3600;

   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* (yorum ekle)

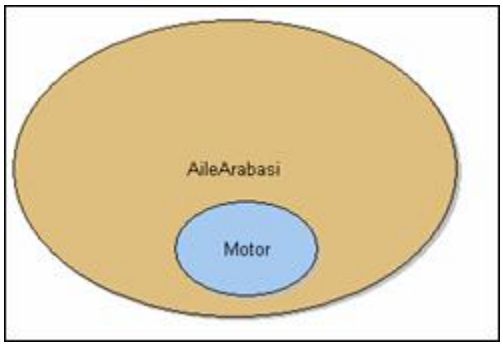
```
public class AileArabasi {
     private Motor m = new Motor();

     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();
   }
}
```

AileArabası sınıfının içerisine, *Motor* tipinde global bir alan yerleştirilerek, bu iki sınıf birbirine bağlanmış oldu. *AileArabası* sınıfının `hareketEt()` ve `dur()` metotlarında, önce *Motor* sınıfına ait yordamlar (methods) direk olarak çağrıldı. Bu ilişki UML diyagramında incelenirse: (yorum ekle)



Şekil-5.2. Komposizyon-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. (yorum ekle)



Şekil-5.3. Kuş Bakışı Görünüş

AileArabasi sınıfı çalıştırılırsa, ekrana görülen sonucun aşağıdaki gibi olması gerekir: (yorum ekle)

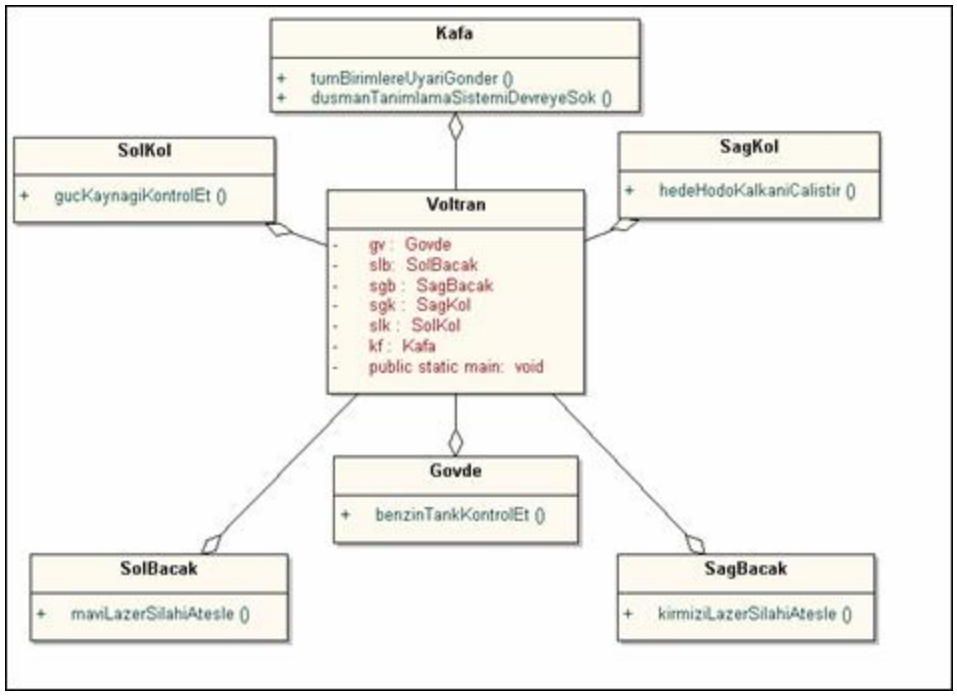
```
Motor Calisiyor
Aile Arabasi Calisti
Motor Durdu
Aile Arabasi Durdu
```

Komposizyon 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, (yorum ekle)

Örnek-5.3: *Voltran.java* (yorum ekle)

```
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: (yorum ekle)



Şekil-5.4. Komposizyon - III

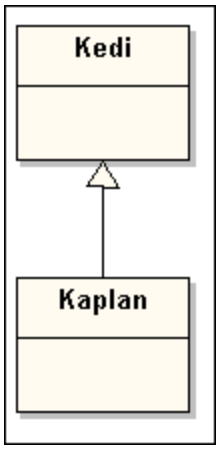
5.2. **Kalıtım**

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). (yorum ekle)
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. (yorum ekle)

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. (yorum ekle)
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; (yorumekle)



Şekil-5.5. Kalıtım İlişkisi-I

Kedi ve *Kaplan* sınıflarımızı biraz daha geliştirelim,

Örnek-5.4: *KediKaplan.java* (yorum ekle)

```
 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. (yorum ekle)

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. (yorum ekle)

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. (yorum ekle)

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. (yorum ekle)

Örnek-5.5: *YeniBirSinif.java* (yorum ekle)

```
 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@82f0dbYeniBirSinif.toString()
YeniBirSinif@92d342ybs1.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. (yorum ekle)

Gösterim-5.4:

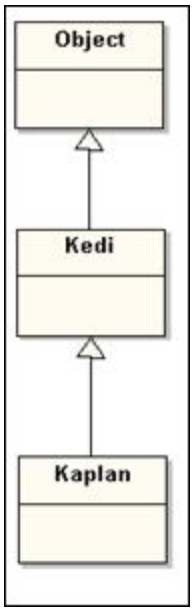
```
public class YeniBirSinif extends Object {
```

Bu sayede *Object* nesnesine ait erişebilir yordamları kullanabiliriz. *Object* nesnesine ait yordamlar aşağıdaki gibidir: (yorum ekle)

- **`clone()`:** Bu nesnenin (`this`) aynısını klonlar ve yeni nesneyi döndürür. (yorum ekle)
- **`equals(Object obj)`:** `obj` referansına bağlı olan nesnenin, kendisine (`this`) eşit olup olmadığı kontrolü yapan yordam. (yorum ekle)

- **`finalize()`:** Çöp toplayıcısı tarafından silinmeden önce çalıştırılan yordam. (yorum ekle)
- **`getClass()`:** Bu nesnenin (`this`) çalışma anındaki sınıf bilgilerini *Class* nesnesi şeklinde geri döner. (yorum ekle)
- **`hashCode()`:** Bu nesnenin (`this`) hash kodunu geri döner.
- **`notify()`:** Bu nesnenin (`this`), monitöründe olan tek bir iş parçacığını (*thread*) uyandırır. (ilerleyen bölümlerde inceleyeceğiz) (yorum ekle)
- **`notifyAll()`:** Bu nesnenin (`this`), monitöründe olan tüm iş parçacıklarını (thread) uyandırır. (ilerleyen bölümlerde incelenecektir) (yorum ekle)
- **`toString()`:** Bu nesnenin (`this`), *String* tipindeki ifadesini geri döner. (yorum ekle)
- **`wait()`:** O andaki iş parçacığının (*thread*) beklemesini sağlar; Bu bekleme `notify()` veya `notifyAll()` yordamları sayesinde sona erer. (yorum ekle)
- **`wait (long zamanAsimi)`:** O andaki iş parçacığının (*thread*), belirtilen süre kadar beklemesini sağlar (`zamanAsimi`); bu bekleme `notify()` veya `notifyAll()` yordamları sayesinde de sona erdirilebilir. (yorum ekle)
- **`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. (yorum ekle)

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. (yorum ekle)
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 sınıftantüretilmiştir. *Object* sınıfı, Java programlama dilinde, sınıf hiyerarşinin en tepesinde bulunur. (yorum ekle)



Şekil-5.6. Gizli Kalıtım

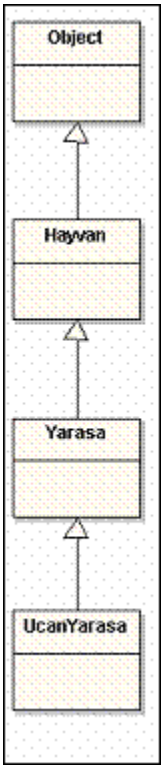
Çoklu kalıtım (*multiple inheritance*), bazı konularda faydalı olmasının yanında birçok sorun oluşturmaktadır. Örneğin iki ana sınıf düşünün, bunların aynı isimde değişik işlemler yapan yordamları bulunsun. Bu olay türetilen sınıfın içerisinde birçok probleme yol açacaktır. Bu ve bunun gibi sebeplerden dolayı Java programlama dilinde çoklu kalıtım yoktur. Bu sebeplerin detaylarını ilerleyen bölümlerde inceleyeceğiz. (yorum ekle)
Java programlama dilinde çoklu kalıtımın faydalarından yararlanmak için Arayüzler (*Interface*) ve dahili sınıflar (*inner class*) kullanılır. Bu konular yine ilerleyen bölümlerde inceleyeceğiz. (yorum ekle)

5.2.2. **Kalıtım ve İlk Değer Alma Sırası**

Tek bir sınıf içerisinde ilk değerlerin nasıl alındığı 3. bölümde incelenmişti İşin içerisine 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. (yorum ekle)
Verilen örnekte, *UcanYarasa* nesnesi oluşmadan evvel, *UcanYarasa* sınıfının ana sınıfı olan *Yarasa* nesnesi oluşturulmaya çalışılacaktır. Fakat *Yarasa* sınıfıda *Hayvan* sınıfından türetildiği için daha öncesinde *Hayvan* sınıfına ait olan yapılandırıcı çalıştırılacaktır. Bu zincirleme giden olayın en başında ise *Object* sınıfı vardır. (yorum ekle)
Örnek-5.6: *IlkDegerVermeSirasi.java* (yorum ekle)

```
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; (yorum ekle)

```
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 çağırmak gereklidir. Şöyle ki; (yorum ekle)

Örnek-5.7: *IlkDegerVermeSirasiParametreli.java* (yorum ekle)

```
 class Insan {
  public Insan(int par) {
      System.out.println("Insan Yapilandiricisi " + par);
  }
}

class ZekiInsan extends Insan {
  public ZekiInsan(int par) {
      super(par+1); //dikkat
      System.out.println("ZekiInsan Yapilandiricisi " +
par);
 }
}

class Hacker extends ZekiInsan{
 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. (yorum ekle)

```
Insan Yapilandiricisi 7ZekiInsan Yapilandiricisi 6Hacker 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. (yorum ekle)

Örnek-5.8: *IlkDegerVermeSirasiParametreliAmaHatali.java* (yorum ekle)

```
 class Insan2 {
  public Insan2(int par) {
      System.out.println("Insan2 Yapilandiricisi " + par);
  }
}

class ZekiInsan2 extends Insan2 {
  public ZekiInsan2(int par) {
      System.out.println("ZekiInsan2 Yapilandiricisi " +
par);
      super(par+1);         // 2. satira yaziliyor ! hata !
  }
}

class Hacker2 extends ZekiInsan2 {
  public Hacker2(int par) {
      System.out.println("Hacker2 Yapilandiricisi " + par);
      System.out.println(".......selam..........");
      super(par+1);         // 3. satira yaziliyor ! hata !
  }

  public static void main(String args[]) {
      Hacker2 hck2 = new Hacker2(5);
  }
}
```

IlkDegerVermeSirasiParametreli.java örneğini derlenirse:

Gösterim-5.5:

```
javac IlkDegerVermeSirasiParametreliAmaHatali.java
```

Aşağıdaki derleme-anı (*compile-time*) hatası ile karşılaşılır: (yorum ekle)

```
IlkDegerVermeSirasiParametreliAmaHatali.java:11: cannot
resolve symbol
symbol : constructor Insan2 ()
location: class Insan2
    public ZekiInsan2(int par) {
                               ^
IlkDegerVermeSirasiParametreliAmaHatali.java:14: call to
super must be first statement in constructor
        Super(par+1);              // 2. satira yaziliyor !
hata !
            ^
```

```
IlkDegerVermeSirasiParametreliAmaHatali.java:21: cannot
resolve symbol
symbol : constructor ZekiInsan2 ()
location: class ZekiInsan2
    public Hacker2(int par) {
                           ^
IlkDegerVermeSirasiParametreliAmaHatali.java:25: call to
super must be first statement in constructor
        super(par+1);              // 3. satira yaziliyor !
hata !
             ^
4 errors
```

5.3. **Komposizyon mu? Kalıtım mı?**

Yeni oluşturduğunuz sınıfın içerisinde, daha evvelden yazılmış sınıfların özelliklerinden faydalanmak istiyorsanız bunun iki yolu olduğunu belirtmiştik; Komposizyon ve kalıtım. Peki hangi yöntemi ne zaman tercih etmeliyiz? Komposizyon, daha evvelden yazılmış sınıfların özelliklerini kullanmak için temiz bir yöntemdir. (yorum ekle)

Örnek-5.9: *Araba.java* (yorum ekle)

```
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. (yorum ekle)
Örnekleri çoğaltmak mümkündür; *UçanYarasa*, *Yarasa* ve *Hayvan* arasındaki ilişki açıklanısa, (yorum ekle)

- *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. (yorum ekle)

Örnek-5.10: *KitapEvi.java* (yorum ekle)

```
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; (yorum ekle)

```
Kitap - sayfaSayisiOgren()
Kitap - fiyatOgren()
Kitap - yazarIsmiOgren()
```

Roman sınıfının içerisinde `sayfaSayisiOgren()`, `fiyatOgren()`, `yazarIsmiOgren()` 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. (yorum ekle)

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, (yorum ekle)

Örnek-5.11: *KitapEvi2.java* (yorum ekle)

```
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 sınıfdan türeyen yeni sınıfın içerisine (Roman2) yazmış olduk. Peki bu durumda uygulamanın ekrana basacağı sonuç nasıl olur? Uygulamayı derleyip, çalıştırınca, ekrana basılan sonuç aşağıdaki gibidir; (yorum ekle)

```
Roman2 - sayfaSayisiOgren()
Roman2 - fiyatOgren()
Kitap2 - yazarIsmiOgren()
```

Roman2 sınıfının içerisinde, ana sınıfa ait yordamların aynılarını tanımladıktan sonra, *Roman2* sınıfının `sayfaSayisiOgren()` ve `fiyatOgren()` yordamlarını çağrınca, 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. (yorum ekle)

Ana sınıfa ait yordamları iptal ederken dikkat edilmesi gereken önemli hususlardan biri erişim belirleyicilerini iyi ayarlamaktır. Konuyu hatalı bir örnek üzerinde gösterirsek; (yorum ekle)

Örnek-5.12: *Telefonlar.java* (yorum ekle)

```
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 Telefon; attempting to assign weaker
access privileges; was protectedprivate 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. (yorum ekle)

En erişilebilir erişim belirleyicisinden, en erişilemez erişim belirleyicisine doğru sıralarsak; (yorum ekle)

- **`public`**: Her yerden erişilmeyi sağlayan erişim belirleyicisi. (yorum ekle)
- **`protected`**: Aynı paket içerisinden ve bu sınıfdan türemiş alt sınıflar tarafından erişilmeyi sağlayan erişim belirleyicisi. (yorum ekle)
- **`friendly`**: Yalnızca aynı paket içerisinden erişilmeyi sağlayan erişim belirleyicisi. (yorum ekle)
- **`private`**: Yalnızca kendi sınıfı içerisinden erişilmeyi sağlayan, başka her yerden erişimi kesen erişim belirleyicisi. (yorum ekle)

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. (yorum ekle)

Örnek-5.13: *Hesap.java* (yorum ekle)

```
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 içerisindeler- *Bilgisayar* sınıfı, *HesapMakinesi* sınıfına erişemeyeceğinden dolayı (çünkü *HesapMakinesi* sınıfı `friendly` erişim belirleyicisine sahip) kalıtım kavramı söz konusu bile olmazdı. (yorum ekle)

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. (yorum ekle)
İki ayrı sınıfı farklı paketlere kopyaladık, özellikle *HesapMakinesi* sınıfını `public` sınıf yapmalıyız, yoksa değişik paketlerdeki sınıflar tarafından erişilemez, dolayısıyla kendisinden türetilme yapılamaz. (yorum ekle)
HesapMakinesi sınıfını *tr.edu.kou.math*, *Bilgisayar* sınıfını ise *tr.edu.kou.util* paketinin içerisine yerleştirelim; (yorum ekle)

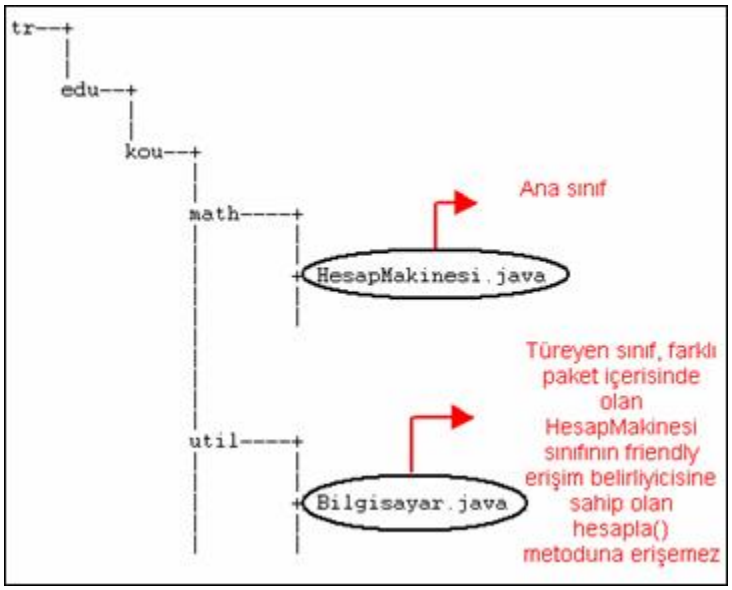
Örnek-5.14: *HesapMakinesi.java* (yorum ekle)

```
package tr.edu.kou.math;
public class HesapMakinesi {
  void hesapla(double a , double b) {
      System.out.println("HesapMakinesi.hesapla()");
  }
}
```

Örnek-5.15: *Bilgisayar.java* (yorum ekle)

```
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: (yorum ekle)

```
 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. (yorum ekle)

Örnek-5.16: *CalisanMudur.java* (yorum ekle)

```
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: (yorum ekle)

```
Calisan.isYap()
```

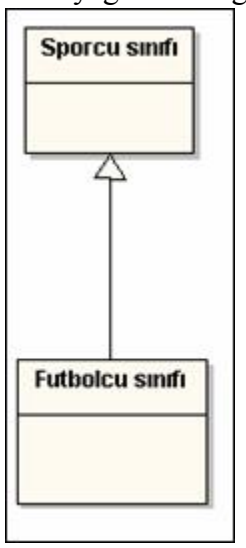
5.5. **Yukarı Çevirim** (***Upcasting***)

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. (yorumekle)
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: (yorum ekle)

Örnek-5.17: *Spor.java* (yorum ekle)

```
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; (yorum ekle)



Şekil-5.9. Yukarı Çevirim

Sporcu.java örneğimizde, türeyen sınıf (*Futbolcu*) türetildiği sınafa (*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. (yorum ekle)
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. (yorum ekle)

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. (yorum ekle)

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; (yorum ekle)

- Derleme anında değerlerini bilebildiğimiz `final` global alanlar. (yorum ekle)
- Çalışma anında değerlerini bilebildiğimiz `final` global alanlar. (yorum ekle)

Örnek-5.18: *FinalOrnek.java* (yorum ekle)

```
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. (yorum ekle)
Final özelliğinin etkisi, ilkel tipteki alanlar ve sınıf tipindeki alanlar farklıdır. Yukarıdaki örneğimizi incelerseniz, `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: (yorum ekle)

```
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. (yorum ekle)
Örnek-5.19: *FinalParametre.java* (yorum ekle)

```
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 alarak 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. (yorum ekle)

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. (yorum ekle)

Örnek-5.20: *BosFinal.java* (yorum ekle)

```
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. (yorum ekle)

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. (yorum ekle)

Örnek-5.22: *FinalMetod.java* (yorum ekle)

```
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: (yorum ekle)

```
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. (yorum ekle)

Örnek-5.23: *SivilPolis.java* (yorum ekle)

```
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. (yorum ekle)

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 komposizyon yöntemi ile kullanabilirler. (yorum ekle)

Örnek-5.24: *Tv.java* (yorum ekle)

```
final class Televizyon {
  public void kanalBul() {
  }
}

/*
class SuperTelevizyon extends Televizyon{  // Hatali
}
*/

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. (yorum ekle)

Örnek-5.25: *Bocekcik.java* (yorum ekle)

```
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 verildistatic Bocekcik.x2 ilk deger
verildiBocekcik - basla..a = 10, b = 0Bocekcik.k ilk deger verildik = 18b =
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…). (yorum ekle)

Daha sonra statik olan global alanlara ilk değerleri verilmeye 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. (yorum ekle)

```
 static Bocek.x1 ilk deger verildistatic Bocekcik.x2 ilk deger verildi
```

Sırada `main()` yordamının çağrılmasına gelmiştir. Ekrana çıkan üçüncü satır aşağıdaki gibidir; (yorum ekle)

```
 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; (yorum ekle)

```
a = 10, b = 0
```

Son olarak *Bocekcik* sınıfının içerisindeki statik olmayan (*non-static*) alanlara ilk değerleri verilir ve *Bocecik* sınıfının yapılandırıcısı çağrılır. (yorum ekle)

```
Bocekcik.k ilk deger verildik = 18b = 17
```

ve mutlu son …