Nesneye Yönelik Programlama

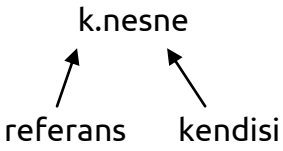
Burak Kıymaz

burakkiymaz.com

2015 - 2016

Nesneye Yönelik Programlama
(Object Oriented Programming)

```
int *a;

a= adres belirtir.
*a = 10    // a=10 değerini atar.
```



referans nesneyi kontrol eden bileşendir.

OOP gerçek hayattaki nesne sınıf olgusunun yazılıma aktarılmış şeklidir. . Örneğin otomobil sınıfı. Sokakta gördüğümüz bütün otomobiller bu sınıfa dahil olabilir.

Ders içerisinde **http://kodcu.com/e-kitap/** adresindeki e-kitap referans alınacaktır.

```
Class Kumanda{
      ...
      Kumanda a1 = new kumanda() // nesne üretildi.
      ...
}
```

new operatörünün olduğu yerde nesne üretildi demektir.

Kumanda	→sınıf
a1	→referans
new kumanda()	→nesne
kumanda()	→kurucu metod

nesnenin kendisi belleğin heap bölgesinde, referans ise stack bölgesinde tutulur. Heap bölgesi stack bölgesine göre daha büyüktür. Fakat stack bölgesi heap e göre daha hızlıdır.

`a1.command();` →verilen 'command' metodunun içerisindeki komutu çalıştırır.

Nesne sayısının bir sınırı yoktur. Ne kadar new yazılırsa o kadar (heap alanı dolana kadar) nesne üretilir.

Sınıf tipindeki değişkenlerin ilk harfi büyük yazılır. Örneğin "String" metodu.

heap, stack, static area Ram üzerinde bulunur.

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

Integer in = new Integer(5);     // sarmalayıcı sınıf (nesnel
                                 mantıkla oluşturulmuş şekli)
```

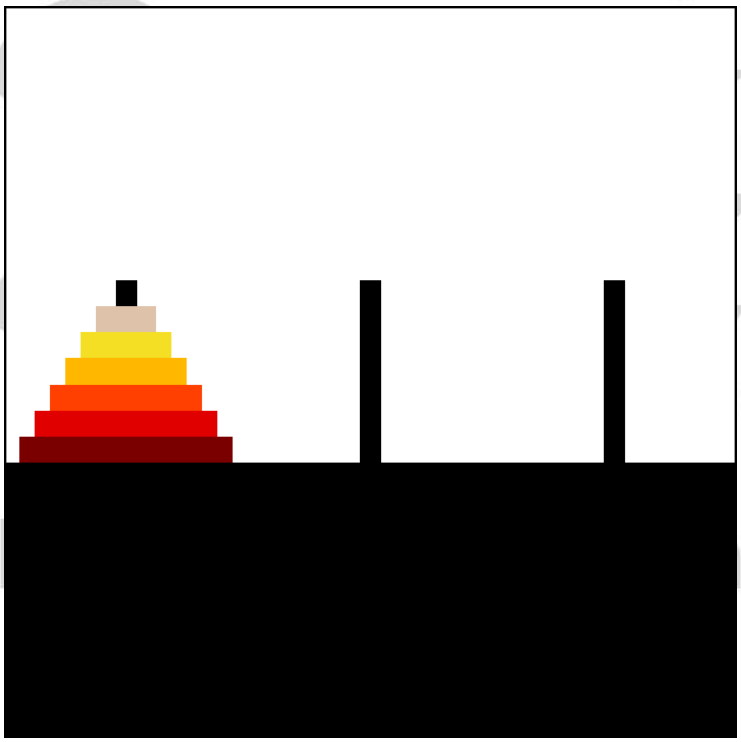
in tamsayı sınıfından bir nesnedir, i bir değişkendir.

Nesne tüm alanlarda geçerli, referans sadece blok içerisinde geçerlidir.

Yerel değişkenlere başlangıç değeri verilmelidir.

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

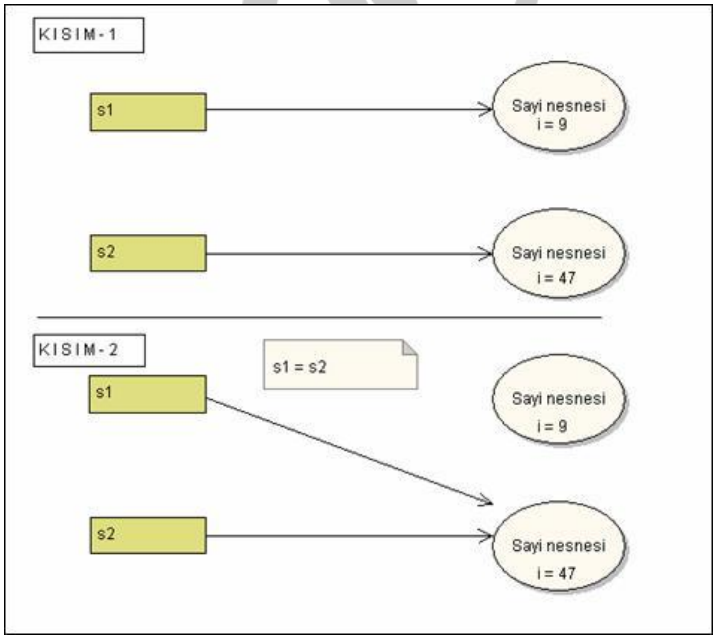
—Ders içerisinde bahsedilen Hanoi-Tower probleminin çözümüne **buradan** ulaşabilirsiniz.



Nesne 29 şubat

```
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 kopyalaniyor.. nesneler degil
        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);  }
 }
```

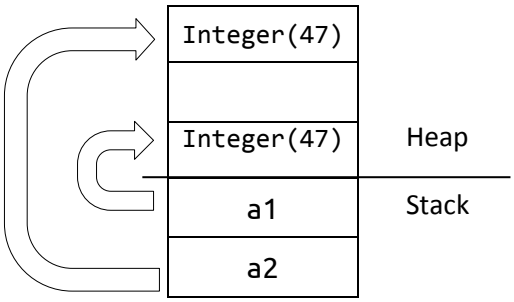
`s1 = s2` komutu verildiği zaman s1 in gösterdiği nesne garbage collector tarafından toplanır.



Kaybolan s1 in nesnesini tekrar oluşturmak için new operatörünü kullanmamız gerekir.

```
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);
      }
}
```

```
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);
  }
}
```

a1 a2 farklı nesneleri gösterdiği için birbirine eşit değildir. Bu yüzden ekran çıktısı:

```
False
True
```

Şeklinde olacaktır.

Kurucu methodlar

Başlangıçta ne yaptırılması isteniyorsa onu yapan method a "kurucu method" denir. Sınıf ismi ile aynı isimde tanımlanır ve parametre alabilir fakat değer döndürmez bir nevi void gibi çalışır.

```
  class KahveFincani {

  public KahveFincani() {
        System.out.println("KahveFincani...");
  }
}

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

```
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 );
  }
}

//Şeklinde gösterildiği gibi kurucu methodlar parametre de alabilir.
```


Döngü her döndüğünde kahve fincanı class ının içerisindeki kurucu method çalışır ve ekrana 5 kere "KahveFincani..." yazar.

Kurucu yazılmadıysa default olarak boş bir kurucu method eklenir.

```
 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
 }
}
```

Overload methodlar:

İsimleri aynı fakat parametreleri farklı methodlardır. Parametrelerin sayısı, sırası ve türü ile birbirlerinden ayırt edilebilir. Döndürdüğü değerin türü ayırt edici bir özellik değildir.

```
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 VarsayilanYapilandiriciVersiyon2 {
   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);
  }
}
```


- ➔ Public aynı paket içerisindeki tüm classlarda nesne üretmeye izin verir.
- ➔ Private aynı class içerisinde nesne üretmeye izin verir.

Bir class içerisinde birden fazla kurucu method tanımlanabilir. Fakat bu kurucu methodların overloading biçimde tanımlanması gerekir.

“this” sözcüğü değerlerin global alanlarına erişmek için referansların yerine geçer.

Nesneye Yönelik Programlama (7 Mart 2016)

`this` sözcüğü nesnenin global alanına erişim sağlamak için kullanılır.

```
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();
  }
}
```

- birinci kullanımı nesnelerin global alanlarına erişim sağlamak içindir.
- ikinci kullanımı ise kurucu metotların yerine `this` kelimeciği kullanılabilir.

```
 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 yapılandırıcılarda
kullanılır
          System.out.println("Tost sayisi="+sayi+ "malzeme =" + malzeme );
  }

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

ilk kullanımda referansı yerine geçmiştir fakat ikinci kullanımda direk nesneyi döndürür.

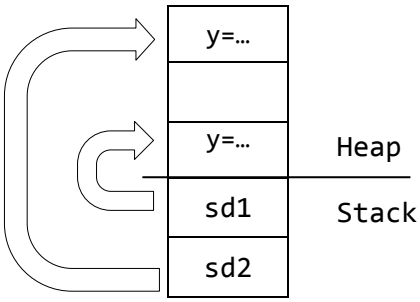
bir üçüncü kullanımı ise kurucular içerisinde geçiş sağlamak için `this` kullanılabilir.

bir fonksiyon içerisinde aynı görevde iki tane `this` kullanılamaz. Birinin kurucu diğerinin ise referans olarak kullanılmış olması gerekmekte.

`this` sadece kurucu methodlar içerisinde yazılır ve yalnız birinci satırda yazılır.

Bir kurucu method içerisinde yalnızca bir defa kullanılabilir.

STATİC AREA



Eğer bir değişken 'static' olarak tanımlanmışsa o nesnenin alanı değildir. Bellekte bir değişken olarak ayrılmıştır. Statik alan nesneden bağımsız alandır. Nesne üretilmese bile x e değer atanabilir.

```
sd1.x = 10; // x=10 ile hiçbir farkı yoktur.
```

Statik method ise nesneden bağımsız method dur. Yani nesne nin ismini kullanmadan bu methoda erişim sağlayabiliriz. Nesne oluşturmadan kullanılabilir.

Normal (Statik olmayan) bir methodun içerisinden statik methodu çağırabiliriz. Fakat statik mothodun içerisinden normal methodlar çağırılamaz.

```
public class StatikTest {

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

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

        hesapla(a,b);
  }
}
```

Başka bir sınıf içerisinden static methodu çağırmak için `sınıfAdı.method` şeklinde yazmak yeterlidir. Fakat normal metholarda çağıracağımız sınıfta bir nesne oluşturup o nesne üzerinden çağırmamız gerekir.

"Önce şunu sormalıyız bir methodun çalışması için nesneye gerek var mı yok mu?

TEMİZLİK İŞLEMLERİ (GARBAGE COLLECTOR)

Finalize methodu:

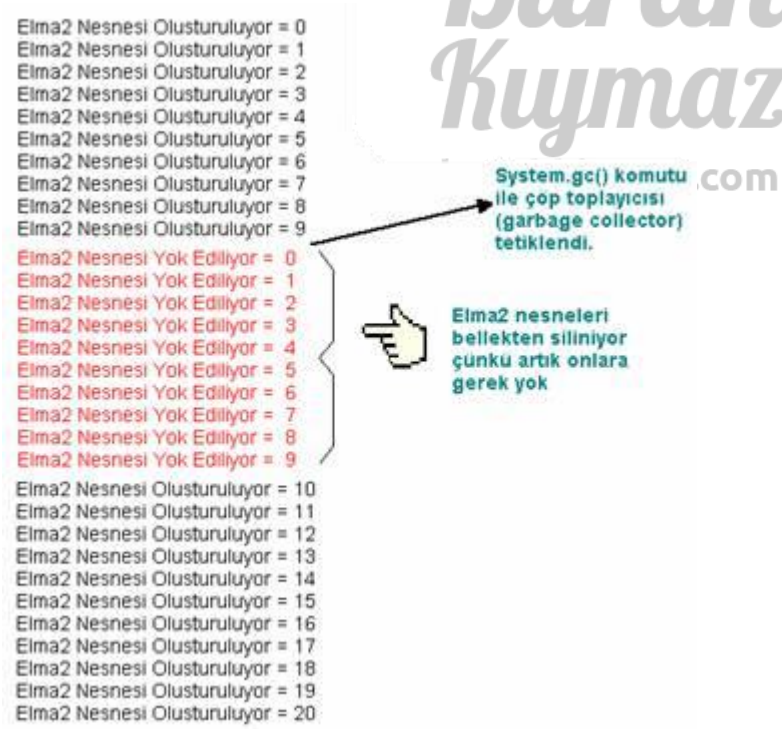
Bu method bellekten silinecek bir değerin bellekten silinmeden önce yapılacak olan işlemleri yapmak için kullanılır. Yani silinecek nesneye son bir dileğin var mı diye sorar.

```
 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);
          }
        }
}
```

```
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
```

```
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.gc() : Garbage Collector tetiklendi. GC tetiklendiği anda tüm referansı olmayan nesneleri bellekten siler. (Silme işlemi gerçekleşmeden önce finalize komutu çalışır.)



Nesneye Yönelik Programlama (14 Mart 2016)

Static alan method içerisinde kullanılmaz. Global alanda tanımlama yapılması gerekmektedir.

Tanımlanan bir veri tipinin ilk değerinin atanması gerekmektedir. Eğer ilk değer atanmazsa bu değişken (örneğin Integer değişken) integer sınırları aralığında otomatik bir değer alacaktır.

```
int i;

i++; // !başlangıç değeri atanmadığı için böyle bir tanımlama yapılamaz.
```

Temel veri tipindeki global alanların static yapılabileceği gibi sınıf tipindeki global alanlar da static yapılabilir.

Static alanlara methodlar aracılığıyla değer atanabilir.

```
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();
  }
}
```


Bir sınıfın içerisinde önce nesne üretme işlemleri gerçekleşir. Bu bir kuraldır. Nesneler üretildikten sonra defter sınıfının kurucusu çalışır. Statik değerlerin ise normal değerlere göre daha çok önceliği vardır.

```
class Kagit {
  public Kagit(int i) {
      System.out.println("Kagit ("+i+") ");
  }
}

public class Defter {
  Kagit k1 = new Kagit(1);        // dikkat
  public Defter(){// bu kısım k1,k2 ve k3 nesneleri üretildikten sonra çalışır.
      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();
  }
}
```

Static değerlerin normal nesnelere göre daha çok önceliği vardır.

Statik alanlara toplu değer atama

```
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(); // burada nesne oluşturuldu.
  }
}
```


Diziler

```
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
```

```
Int liste[] = new int[5];

Liste = new int[15];
```

Ilk satırda liste isminde 5 elemanlı bir dizi tanımlanır. İkinci satırda ise 15 elemanlı bir dizi liste isimli referansa bağlanmaktadır. Liste ismli değişken ikinci kere kullanıldığı için liste ikinci diziyi (15 elemanlı) gösterecek ve ilk satırdaki 5 elemanlı liste ise garbage collector tarafından toplanacaktır.

Paket Erişimleri (bölüm 4)

`import java.io.*` : java/io dizini altındaki tüm .class uzantılı dosyaları import eder.

`import java.io.BufferedReader` : java/io içerisindeki BufferedReader class'ını import eder.

Paket kavramı kütüphane dosyalarının bir araya gelerek oluşturduğu yapıdır.

Jar Paketleme

Jar dosyası, oluşturulan class dosyalarının daha derli toplu görünmesini sağlayan bir sıkıştırma biçimidir.

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

Erişim belirteçleri

Public:

public yazıldığı zaman her taraftan (tüm class lar ve paketler içerisinden) erişim yetkisi sağlanır. Sınıfın, alanların ve methodların önüne yazılabilir.

Friendly:

Sınıf, alan ve methoların önüne yazılabilir. Sadece bulunduğu paket içerisinde erişim sağlayabilir. Java derleyicisi erişim belirteci bulamadığı yerlere otomatik olarak `friendly` erişim belirtecini koyar. Bu yüzden kodlarda çok fazla kullanılmaz.

Protected:

İç içe class lar hariç sadece alanlar ve methodların önüne yazılır. Kalıtım olması gerekir. Kalıtım hiyerarşisi (inheritance) olan sınıflarda işe yarar bir kullanım sunar.

```
class Kaplan extends kedi{

}
```

Kedi classı içerisindeki protected kısımlar Kaplan classı tarafında erişim sağlanabilir. Extends eki bu işe yarar.

Private:

İç içe sınıflar hariç alanlar ve methodların önüne yazılır. Sadece bulunduğu class içerisinde erişim yetkisi sağlayabilir. Diğer class lar içerisinden erişilemez.

```
package tr.edu.kou.gerekli;
  class Robot {
    int calisma_sure = 0;
        String renk = "beyaz";
        int motor_gucu = 120;
        Robot() {
          System.out.println("Robot olusturuluyor");
  }
}
```

```
package tr.edu.kou.gerekli;
class Profesor {
         public void kullan() {
            Robot upuaut = new Robot();
/*
* aynı paket içerisinde tanımlandıkları için friendly olan bir class
* a direk erisim sağlanabilir.
*/
         }
 }
```

```
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();
/*
* Farklı paket içerisinde olduları için friendly bir class a erişim
* sırasında hata alınacaktır.
*/

       }
}
```

Nesneye Yönelik Programlama (20.03.2016)

```
package tr.edu.kou.gerekli;
class Kahve {
       private int siparis_sayisi;

       private Kahve() {    }

       private void kahveHazirla() {
         System.out.println(siparis_sayisi + " adet kahve"
          +" hazirlandi");
       }

       public static Kahve siparisGarson(int sayi) {
           Kahve kahve = new Kahve(); //dikkat
           kahve.siparis_sayisi = sayi ;
           kahve.kahveHazirla();
           return kahve;
       }
}
```

```
 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 kh = Kahve.siparisgarson(5);
```

Komutu çalışır. Çünkü fonsiyon public olarak tanımlanmıştır. Bu fonksiyon static olarak tanımlanıdığı için friendly olarak tanımlansa bile erişim sağlanabilir.

Protected erişim belirteci

```
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";
}
```

```
Package tr.edu.kou.gerekli;İmport 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();
      }
}
```

Burada Hayvan classındaki b elemanına friendly erişim belirteci olduğu için erişilemez. Çünkü hayvan class ı ile Kedi class ı farklı paketler içerisinde. c elemanına ise önündeki erişim belirteci private olduğu için erişilemez.

Kapsüllenme

Nesneye yönelik programlama özelliklerinden birisi kapsüllenmedir; 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.

Makine2.java dosyası:

```
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().

GetSet.java dosyası:

```
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);
     }
}
```

Sınıflar için erişim tablosu:

	Aynı Paket	**Ayrı Paket**	**Ayrı paket-türetilmiş**
`public`	erişebilir	erişebilir	erişebilir
`Protected`	-	-	-
`Friendly`	erişebilir	erişemez	erişemez
`Private`	-	-	-

Static veya static olmayan yordamlar için erişim tablosu:

	Aynı Paket	**Ayrı Paket**	**Ayrı paket-türetilmiş**
public	erişebilir	erişebilir	public
protected	erişebilir	erişemez	erişebilir
friendly	erişebilir	erişemez	erişemez
private	erişemez	erişemez	erişemez

Nesneye Yönelik Programlama (28.03.2016)

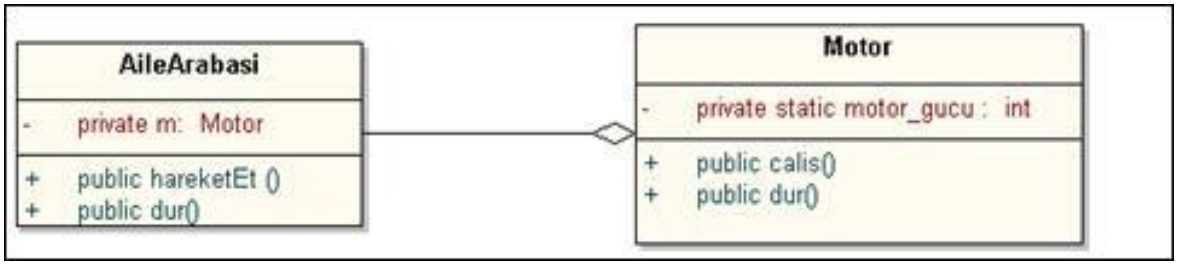
Motor.java Dosyası

```
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") ;
   }
}
```

AileArabasi.java Dosyası

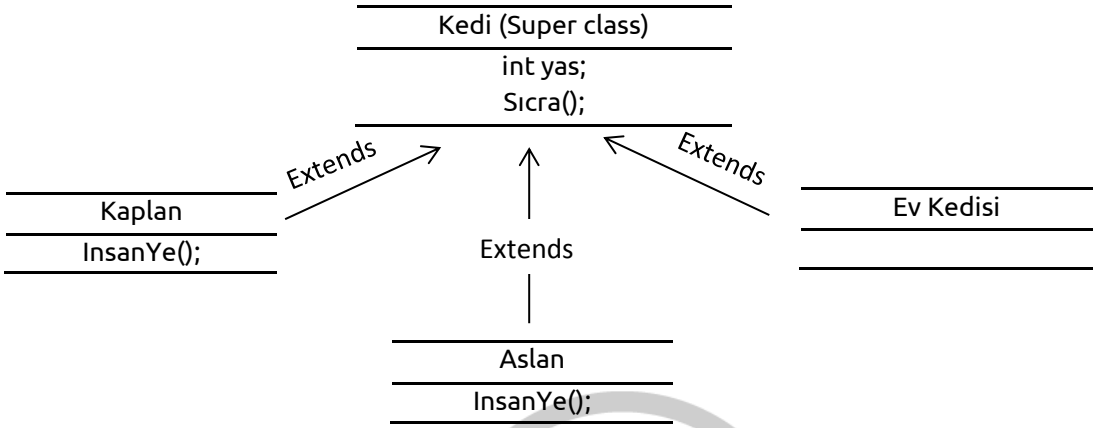
```
 public class AileArabasi {
        private Motor m = new Motor();

        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();
   }
}
```


Burada aile arabası class ının içerisine motor class I eklenerek bu iki class birbirine bağlanmış oldu. Bunu UML Diagramları şeklide ifade edecek olursak:



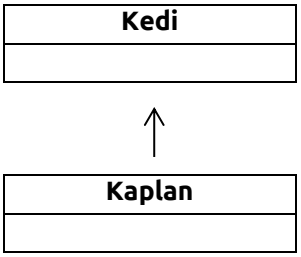
Elmas işaretlerinin olduğu yer, üretilen nesneleri gösterir. Diğer tarafı ise nesnenin üretildiği class ı gösterir.

Kalıtım




```
class kedi{
	int yas;
	sicra();
}

class Kaplan extend kedi{
	insanYe();
	Kaplan a3 = new Kaplan();
}

class Aslan extends kedi{
	insanYe();
	Aslan a2 = new Aslan();
}

class EvKedisi extends kedi{
	kedi EvKedisi a1 = new EvKedisi(); // şeklinde de yazılabilir.
}
```


Kısaca bir sınıftan diğer bir sınıfın türemesidir. Örnekte kaplan sınıfı kedi sınıfından türemiştir.



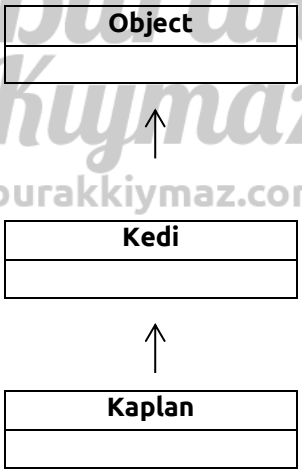
Şekline gösterilir.

Super class da protected olarak tanımalanan alanlar hem tanımlanan class da hem de bu class dan türetilen class larda kullanılır.

KediKaplan.java Dosyası

```
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);
  }
}
```

Gizli kalıtım: bir class oluşturulduğunda otomatik olarak Object sınıfından türetilir.




```
class yeniTur extends Aslan,Kedi // yazımı yanlıştır. Bir sınıf iki sınıftan
                                    türetilemez.
```

evKedisi a1 = new evKedisi();

Kodu çalıştırıldığı zaman ev kedisi class I kedi class ından türetilidiği için önce kedi class ının kurucusu çalıştırılır ve sonrasında ev kedisi classının kurucusu çalışır.

IlkDegerVerme.java Dosyası

```
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();
   }
}
```

Buradaki kodu inceleyecek olursak UcanYarasa class ı Yarasa class ından, Yarasa class ı ise hayvan class ından türetilmiştir. Bu kodu derleyecek olursak en tepede Hayvan classı olmasında dolayı ilk önce Hayvan classının kurucusu çalışacatır. Sonuç çıktı ise aşağıdaki gibi olacaktır:

```
Hayvan Yapilandiricisi
Yarasa Yapilandiricisi
UcanYarasa Yapilandiricisi
```

Nesneye Yönelik Programlama (25.04.2016)

```
class asker{
	int yas;
	int boy;
	selamver();
}

class onbasi extends asker{
	selamver(){
		önce;
	}

}

class albay extends asker{
	selamver(){
		sonra;
	}
}
```

Onabasi ve albay class ları içerisindeki selamver() metodları yukarıdaki metodu ezerler (override)

KURAL!

Method ezerken altsınıftaki metodun erişim belirteci aynı veya daha erişilebilir olmalıdır. Farklı pakette olmaları kuralı değiştirmez. Yani üst sınıftaki metod friendly tanımlamışsa ezecek metodlar private tanımlanamaz.

! OVERRIDE methodlar OVERLOAD methodlarla karıştırılmamalı

```
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);
  }
}
```

Burada overload methodlar kullnılmıştır. işYap fonksiyonu girilen parametreye göre çalışır.

UPCASTING

```
class asker{
	int yas;
	int boy;
	selamver();
}

class onbasi extends asker{
	selamver(){
		önce;
	}
	onbasi a1 = new onbasi();
	asker x = new onbasi();   // Upcasting

}

class albay extends asker{
	selamver(){
		sonra;
	}
	albay a1 = new albay();
	asker x = new albay(); // Upcasting
}
```

```
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.

Final

Sonlandırmak sabitlemek anlamındadır. Global alanlara methodların ve sınıfların önüne final yazılabilir. Bir kere değer alır ve daha sonradan değer atanmasına izin vermez. Yani sabit olarak tanımlanır. class ların önüne final getirildiği zaman o sınıftan asla bir nesne türetilemez.

Blank final:

program içerisinden ilk değer alır sonra değiştirilemez

Polimorfizm (çok biçimlilik):

bir sınıfın nesnesinin başka bir sınıfın nesnesi gibi davranması olayıdır.

bir alt sınıf türetildiği ana sınıftaki tüm özellikleri alır.

```
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 cevirim ! yok !
        hazirOl(e); // yukarı cevirim (upcasting)
       hazirOl(y); // yukarı cevirim (upcasting)
 }
}
```

Geç Bağlama (Late Binding)

Runtime de ne yapacağı belli olan eylemlere late binding (geç bağlama) denir.
Yukarıdaki örnekte hazır ol metodu içerisindeki selam ver komutu hangi nesneye bağlı olduğu belli değildir. Bu runtime da belli olur.

Bu olayın tam tersi de early binding dir. Yani derleme zamanında belli olur.

```
class Hayvan {
  public void avYakala() {
        System.out.println("Hayvan av Yakala");
  }
}

class Kartal extends Hayvan {
  public void avYakala() {
        System.out.println("Kartal av Yakala");
  }
}

class Timsah extends Hayvan{
  public void avYakala() {
        System.out.println("Timsah av Yakala");
  }
}

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!
        }
  }
}
```

Final Ve Geç Bağlama:

bir class a final özelliği verilirse yani method ezme özelliği kaldırılırsa late binding den söz edilemez.

Nesneye Yönelik Programlama (02.05.2016)

Soyut sınıflar(Abstract class)

Kural 1: soyut sınıftan nesne üretilmez.

Kural 2: Abstract class lar içerisinde en az bir tane absract method bulunmalı. Soyut sınıfların gövdesi bulunmaz.

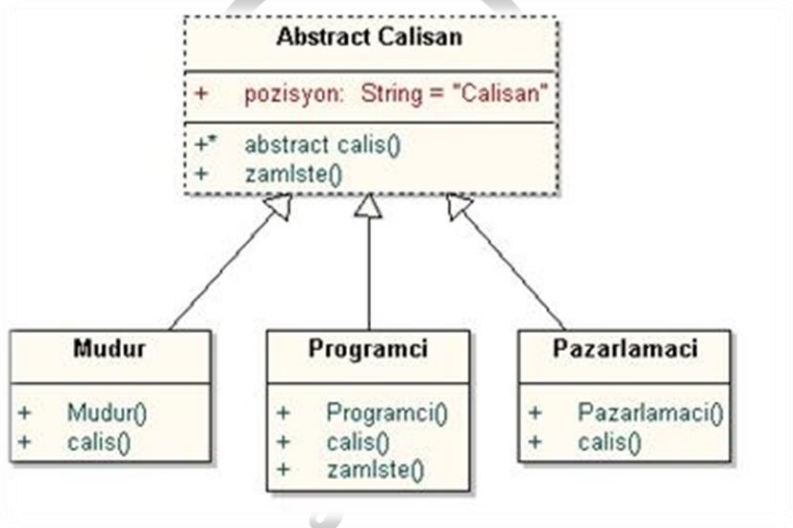
```
abstract void calis(); // gövde bulunmaz.
```

Soyut sınıflarda soyut methodlar olabileceği gibi normal methodlar da tanımlanabilir.

Soyut sınıflarda türetilmiş alt sınıflardan yeni nesneler üretilebilir veya upcasting (yukarı çevrim) yapılabilir.
Bu sayede soyut sınıflarda polimorfizm kullanılması mümkündür.

Soyut sınıftan türetilmiş class lar abstract class içerisindeki abstract metodu kesin olarak ezeceklerdir. Aksi taktirde derleyici hata verecektir.

Global alanlar abstract olamaz. UML Diagramında kesikli çizgilerle gösterilir.



Neden ihtiyaç duyarız?

Eğer bir işlem değişik verilere ihtiyaç duyup aynı işi yapıyorsa, bu işlem soyut (*abstract*) sınıfın içerisinde tanımlanmalıdır.

Nesneye Yönelik Programlama (09.05.2016)

Inteface

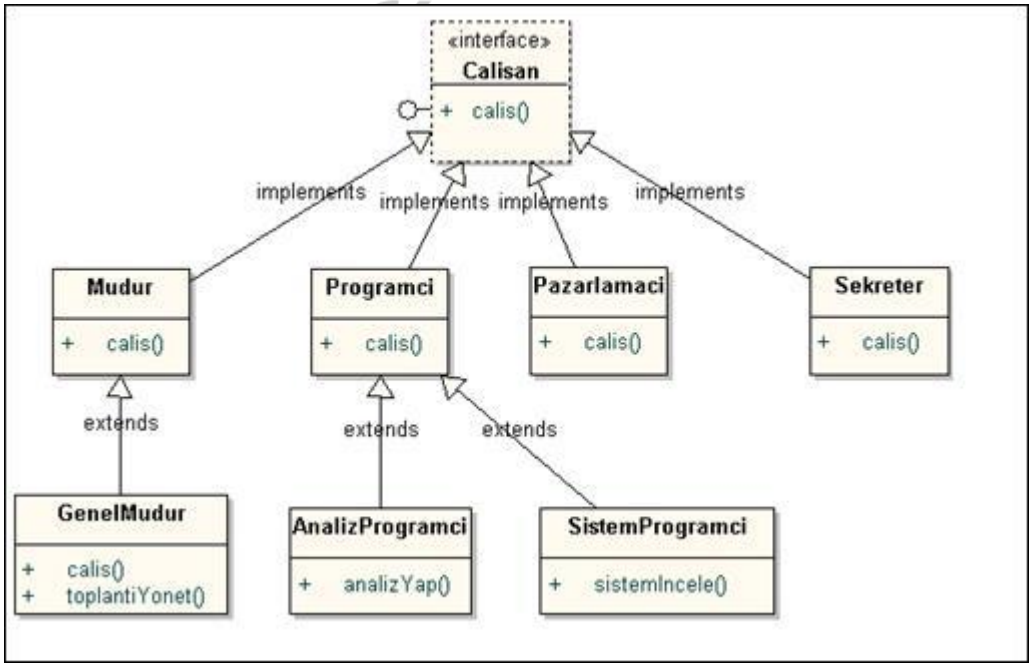
```
Inteface Calisan{
	Public void calis();
}

Class Mudur implements Calisan{
}
```

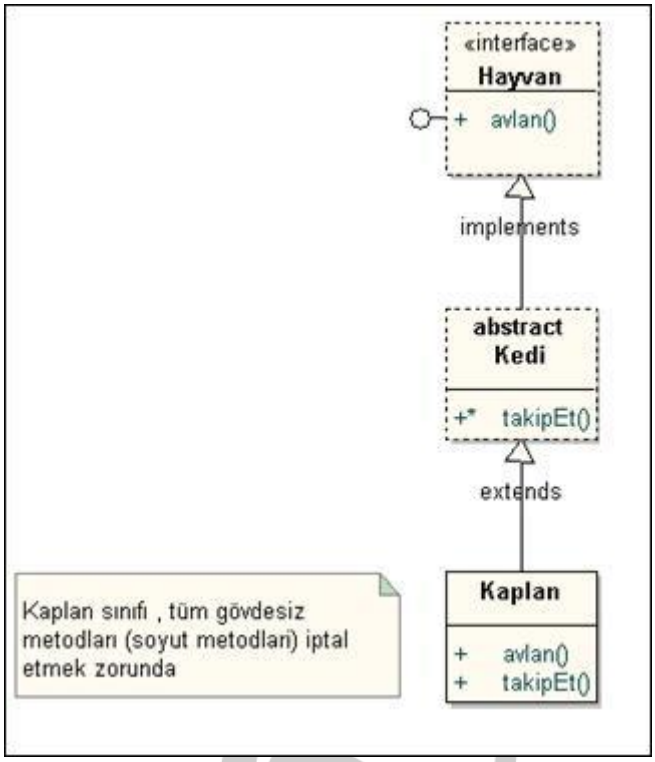
Intefaceler abstract classlar gibi çalışır. Yani çatı görevi görür. UML diagramında kesikli çizgilerle gösterilir. Implements sözcüğüyle başka bir class a bağlanabilir.

Arayüzlerin içerisinde iş yapan metotlar bulunmaz sadece gövdesiz metotlar bulunur. Bu metotlar otomatik olarak public olarak tanımlanır, final ve statik özelliği içerir. Bu classlardan nesne üretilemez. Interfacelere verilecek isimler eylem olmalıdır.

```
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 toplantiYonet() {
	System.out.println("GenelMudur toplanti 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 yaratılamaz
          c[0]=new Programci(); // yukari cevirim (upcasting)
          c[1]=new Pazarlamaci();// yukari cevirim (upcasting)
          c[2]=new Mudur(); //yukari cevirim (upcasting)
          c[3]=new GenelMudur(); //yukari cevirim (upcasting)
          c[4]=new AnalizProgramci();//yukari cevirim
           //(upcasting)

          c[5]=new SistemProgramci();//yukari cevirim
           //(upcasting)

          mesaiBasla(c);
  }
}
```



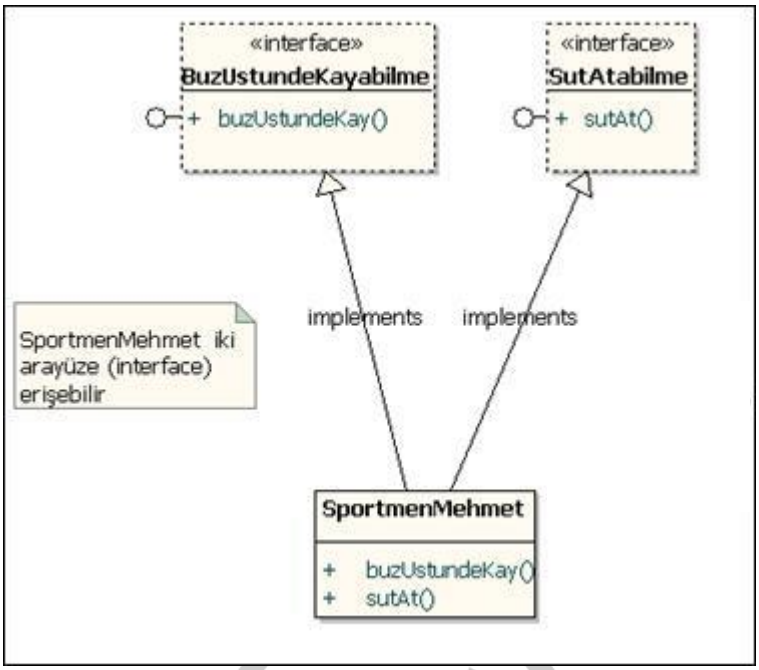


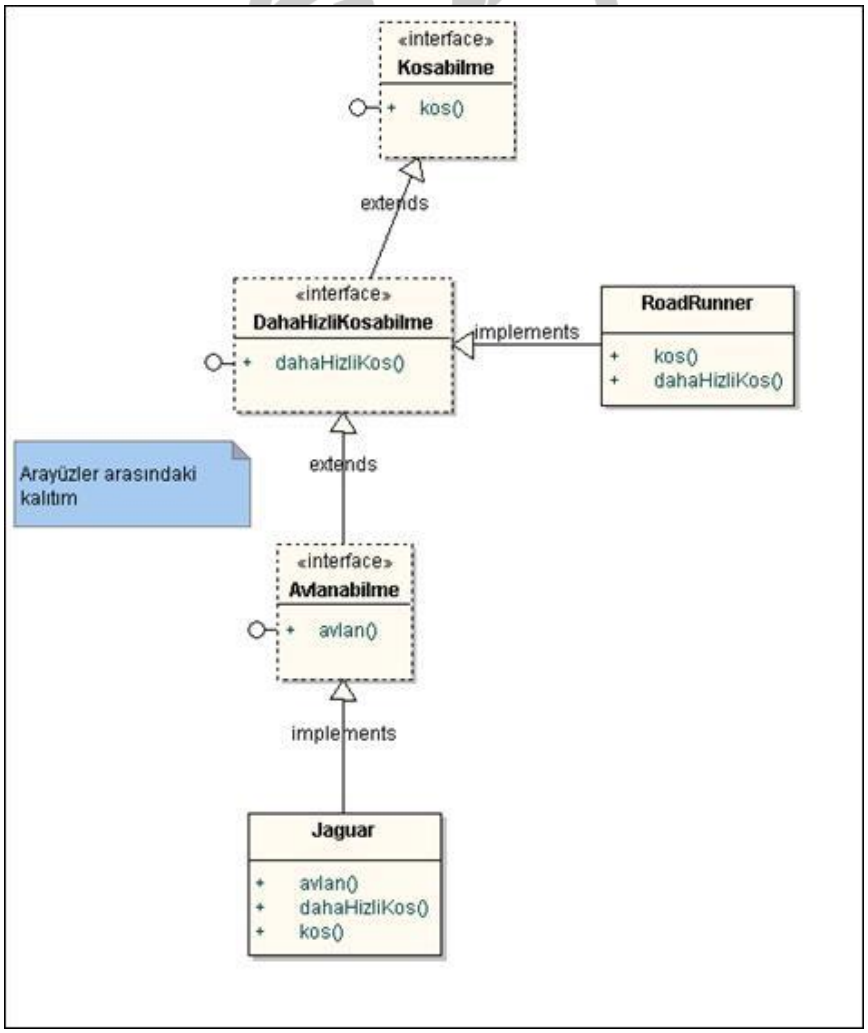
Kaplan sınıfı tüm gövdesiz metotları iptal etmek zorundadır.

```
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 avlaniyor...");
  }

  public void takipEt() { // iptal etti (override)
        System.out.println("Kaplan takip ediyor...");
  }
}
```

Alt sınıf en fazla bir tane sınıftan türetilebilir. Fakat bir alt sınıfa birden fazla interface implement edilebilir. Arayüzler çoklu kalıtımı destekler.

```
class class_ismi implements interface1, interface2,…{
      …
}
```


«interface»
BuzUstundeKayabilme
+ buzUstundeKay()
«interface»
SutAtabilme
+ sutAt()
implements
implements
SportmenMehmet iki arayüze (interface) erişebilir
SportmenMehmet
+ buzUstundeKay()
+ sutAt()


«interface»
Kosabilme
+ kos()
extends
«interface»
DahaHizliKosabilme
+ dahaHizliKos()
implements
RoadRunner
+ kos()
+ dahaHizliKos()
Arayüzler arasındaki kalıtım
extends
«interface»
Avlanabilme
+ avlan()
implements
Jaguar
+ avlan()
+ dahaHizliKos()
+ kos()

```
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");
  }
}
```


Interfaceler arasında extends ile çoklu kalıtım yapılabilir. Bir sınıf bir abstract sınıftan extend edilebilir ve 2 farklı interface den implement edilebilir.

Soyut bir sınıftan üretilen nesne, aynı zamanda implement edilmiş interfacelerin de alt sınıflarıdır.

Nesneye Yönelik Programlama (16.05.2016)

Inner Classes

Başka bir sınıfın içerisinde tanımlanan bu sınıfa dahili üye sınıf denir.

```
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
```

Dahili bir sınıftan nesne tanımlayacaksak iki tane new operatörü bulunur. Bunlardan biri çevreleyici class dan nesne oluşturmuş diğeri ise iç class dan nesne oluşturmaya yarar. Her iki nesne de ht referansına bağlıdır. Dahili sınıftan nesne oluşturmak için çevreleyici sınıftan nesne oluşturmak gerekir.

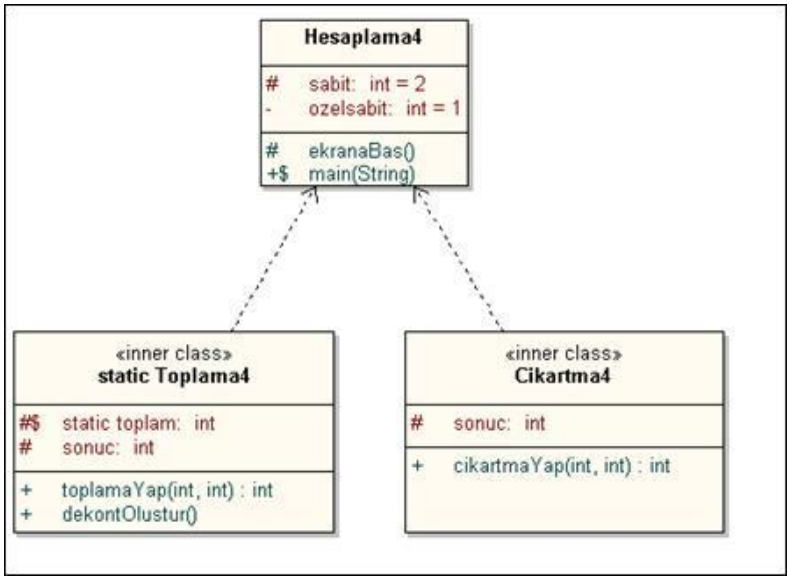
Dahili üye sınıflara tüm erişim belirteçleri (public, protected, friendly, private) atanabilir. Bir dahili sınıf private erişim belirtecine sahip olsa bile çevreleyici sınıflar içerisindeki tüm methodlar tarafından bu class a erişim sağlanabilir.

Dahili Üye Sınıf İle Çevreleyici Arasındaki İlişki

Dahili sınıflar çevreleyici sınıfların tüm alanlarına static olsun olmasın alanlarına ve metotlarına erişim sağlayabilir.

dahili sınıflar static olarak tanımlanımışsa çevreleyici sınıftan nesne üretilmesine gerek yoktur ve çevreleyici sınıfa ait bağlantısını kaybeder. Yani çevreleyici sınıfa ait alanlara ve metotlara erişim sağlayamaz.

UML diagramında class isminin üzerine <<inner class>> yazılır ve çevreleyici class lara kesikli çizgilerler bağlantı yapılır.



çevreleyici sınıfta da dahili sınıfta da kurucu tanımlanabilir ve soyut sınıf olarak tanımlanabilir.

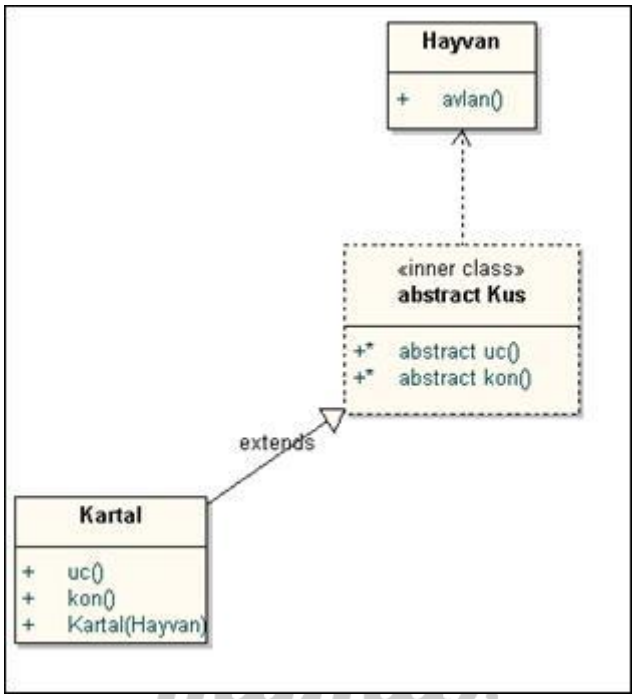
```
public class BuyukA {

  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();
  }
}
```


İç içe Dahili Üye Sınıflar

```
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
```

Dahili sınıflar bağımsız başka bir classı extend edebilir.



Yerel Sınıflar (Local Classes)

Bir metodun içerisinde tanımlanmış class lara denir. Yerel sınıflar yalnızca içerisinde tanımlanan bloğun içerisinde geçerlidir. Tanımlandıkları bloğun dışından erişilemezler. Başka sınıflardan türetilebilir. Yerel sınıfların kurucuları olabilir.

Yerel sınıflar içerisinde bulundukları yordamın sadece final alanlarına ulaşabilirler.

Static veya static olmayan yordamlar içerisinde tanımlanabilir. Yerel sınıflarda erişim belirteci kullanılamaz ve static tanımlanamaz.

İsimsiz Sınıflar

- Diğer dahili sınıf çesitlerinde 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

örnek

```
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
```


Dahili Sınıflara Neden İhtiyaç Duyulmuştur?

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 türetilebilir veya başka arayüzlere erişebilir.