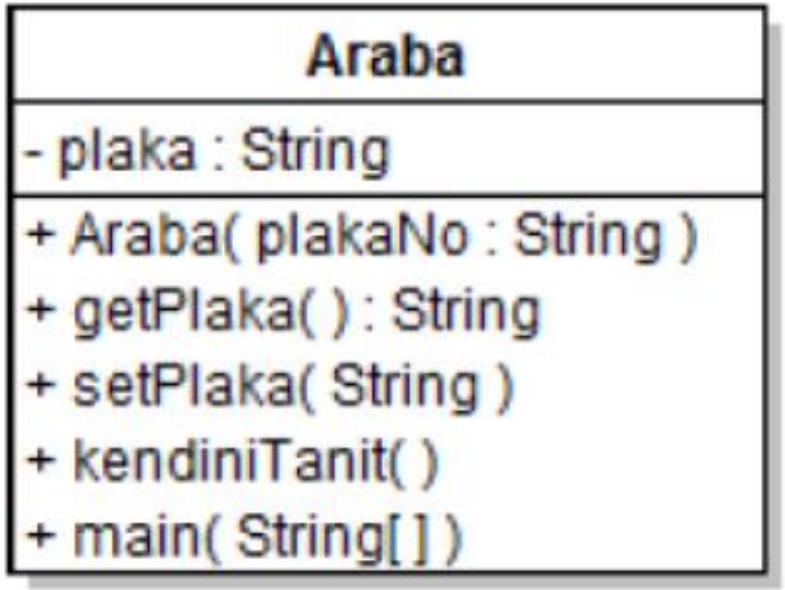
YMT 112 Algoritma ve Programlama

www.kriptrium.com/programlama

İçerik

- Nesne Yönelimli Programlama
  - Kapsülleme
  - Çok biçimlilik
  - Kalıtım




```
public class Araba {
   private String plaka;

   public Araba( String plakaNo ) {
      plaka = plakaNo;
   }
   public String getPlaka( ) {
      return plaka;
   }
   public void setPlaka( String plaka ) {
      this.plaka = plaka;
   }
   public void kendiniTanit( ) {
      System.out.println( "Plakam: " + getPlaka() );
   }

   public static void main( String[] args ) {
      Araba birAraba;
      birAraba = new Araba( "34 RA 440" );
      birAraba.kendiniTanit( );
   }
}
```

KALITIM

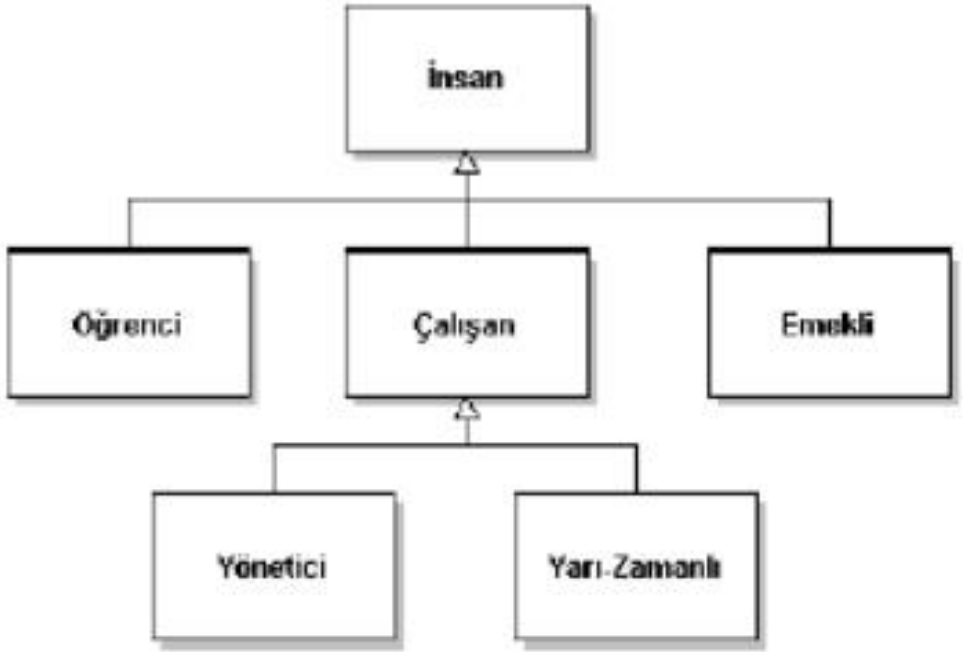
- Kalıtım benzetmesi: Bir çocuk, ebeveyninden bazı genetik özellikleri alır.
- NYP: Mevcut bir sınıftan yeni bir sınıf türetmenin yoludur.
- Gösterim:

Kalıtım (inheritance)

- Ok yönüne dikkat!
- A:
  - Ebeveyn sınıf (parent)
  - Üst sınıf (super)
  - Temel sınıf (base)
- B:
  - Çocuk sınıf (child)
  - Alt sınıf (sub)
  - Türetilmiş sınıf (derived)
- Kalıtımın işleyişi:
  - Kalıtım yolu ile üst sınıftan alt sınıfa hem üye alanlar hem de üye metotlar aktarılır
    - private üyeler dahil, ancak alt sınıf onlara doğrudan ulaşamaz.
  - Protected üyeler ve kalıtım:
    - Alt sınıflar tarafından erişilir, diğer sınıflar tarafından erişilemez.

KALITIM

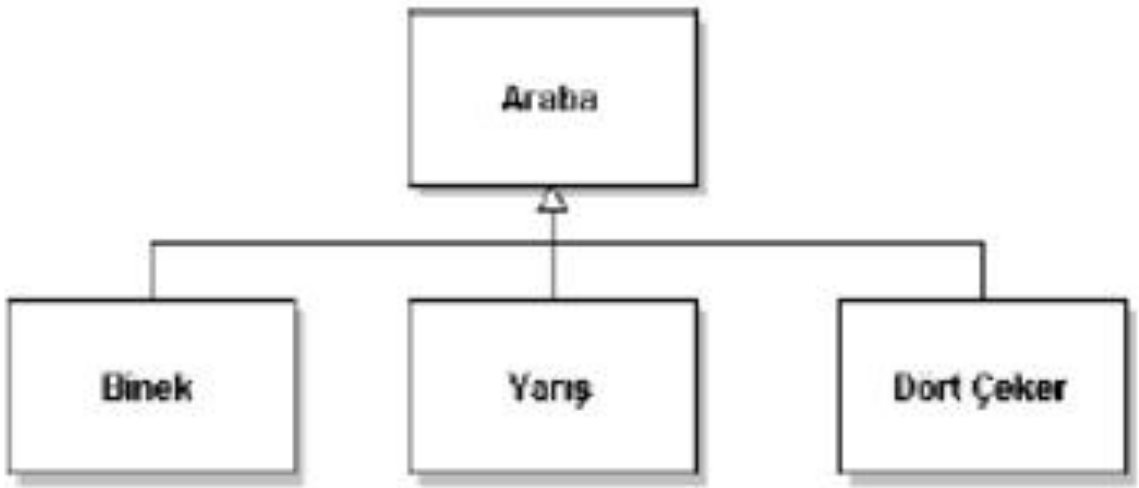
- Kalıtım kuralları:
  - Miras alma adlandırmasının uygunsuzluğu: Alt sınıf herhangi bir üyeyi miras almamayı seçemez.
  - Ancak kalıtımla geçen metotların gövdesi değiştirilebilir.
    - Yeniden tanımlama: Overriding.
    - Final olarak tanımlanan metotlar yeniden tanımlanamaz.
  - Alt sınıfta yeni üye alanlar ve üye metotlar tanımlanabilir.
  - Alt sınıflardan da yeni alt sınıflar türetilebilir. Oluşan ağaç yapısına kalıtım hiyerarşisi veya kalıtım ağacı denir.

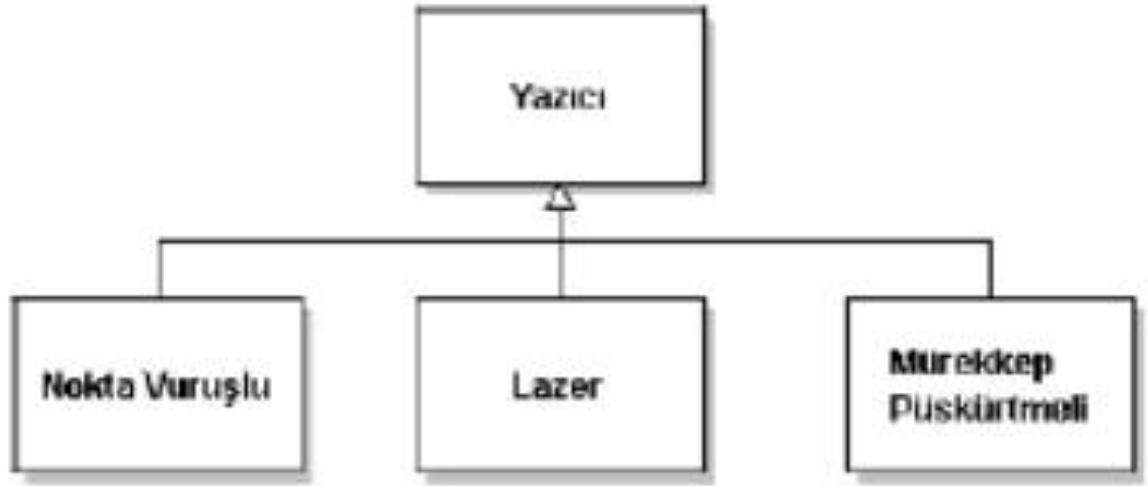


- Kalıtım ağacını çok derin tutmak doğru değildir (Kırılgan üst sınıf sorunu: Bina temelinin çürümesi gibi).

KALITIM

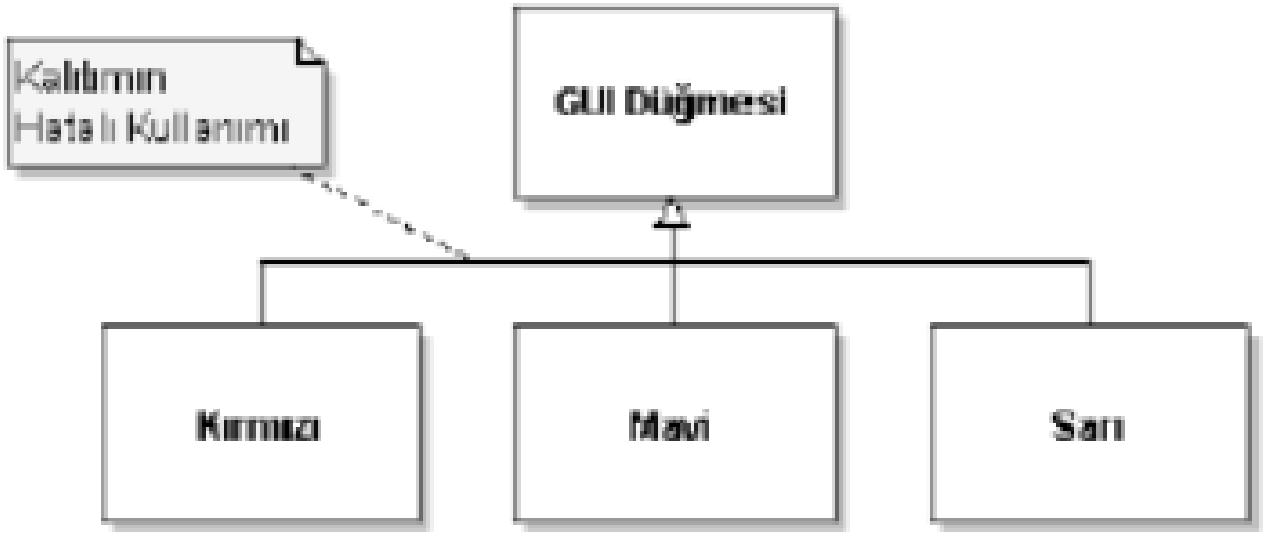
- Kalıtımın etkileri:
  - Genelleşme – özelleşme ilişkisi (generalization – specialization).
    - Alt sınıf, üst sınıfın daha özelleşmiş, daha yetenekli bir türüdür.
  - Yerine geçebilme ilişkisi (substitutability).
    - Alt sınıftan bir nesne, üst sınıftan bir nesnenin beklendiği herhangi bir bağlamda kullanılabilir.
    - Bu nedenle IS-A ilişkisi olarak da adlandırılır.

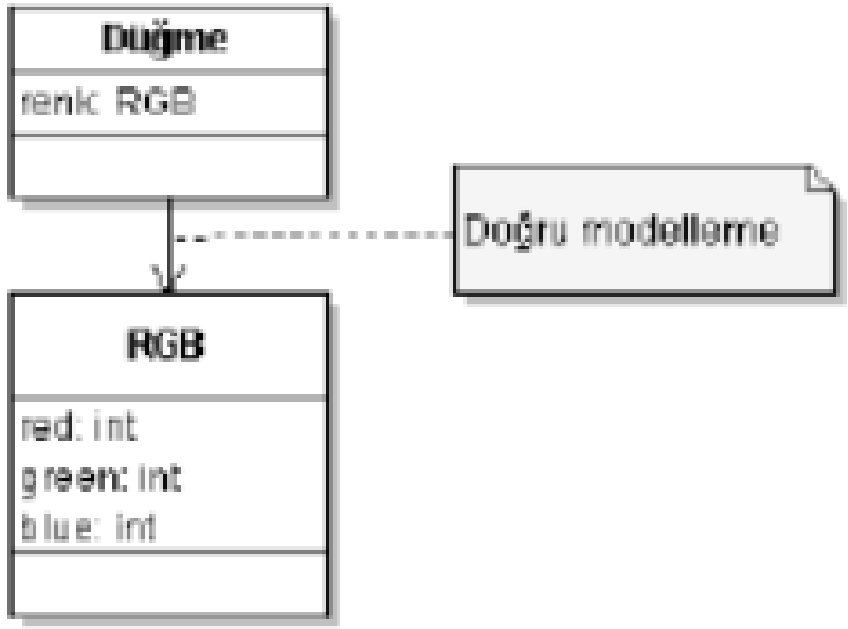




KALITIM

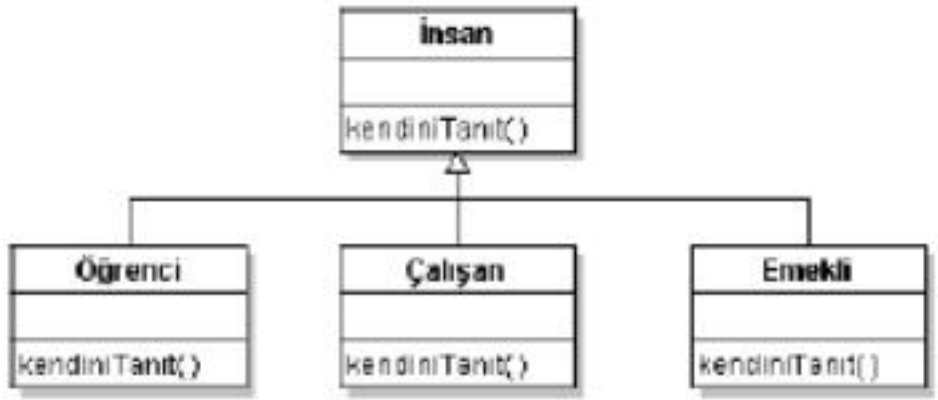
- Kalıtımın yanlış kullanımı:





ÇOK BİÇİMLİLİK (POLYMORPHISM) ve YENİDEN TANIMLAMA (OVERRIDING)

- İstersek kalıtımla geçen metotların gövdesini değiştirebileceğimizi öğrendik.
  - Bu işleme yeniden tanımlama (overriding) adı verildiğini gördük.
- Üst sınıftan bir nesnenin beklendiği her yerde alt sınıftan bir nesneyi de kullanabileceğimizi gördük.
- Bu iki özellik bir araya geldiğinde, ilgi çekici bir çalışma biçimi ortaya çıkar.

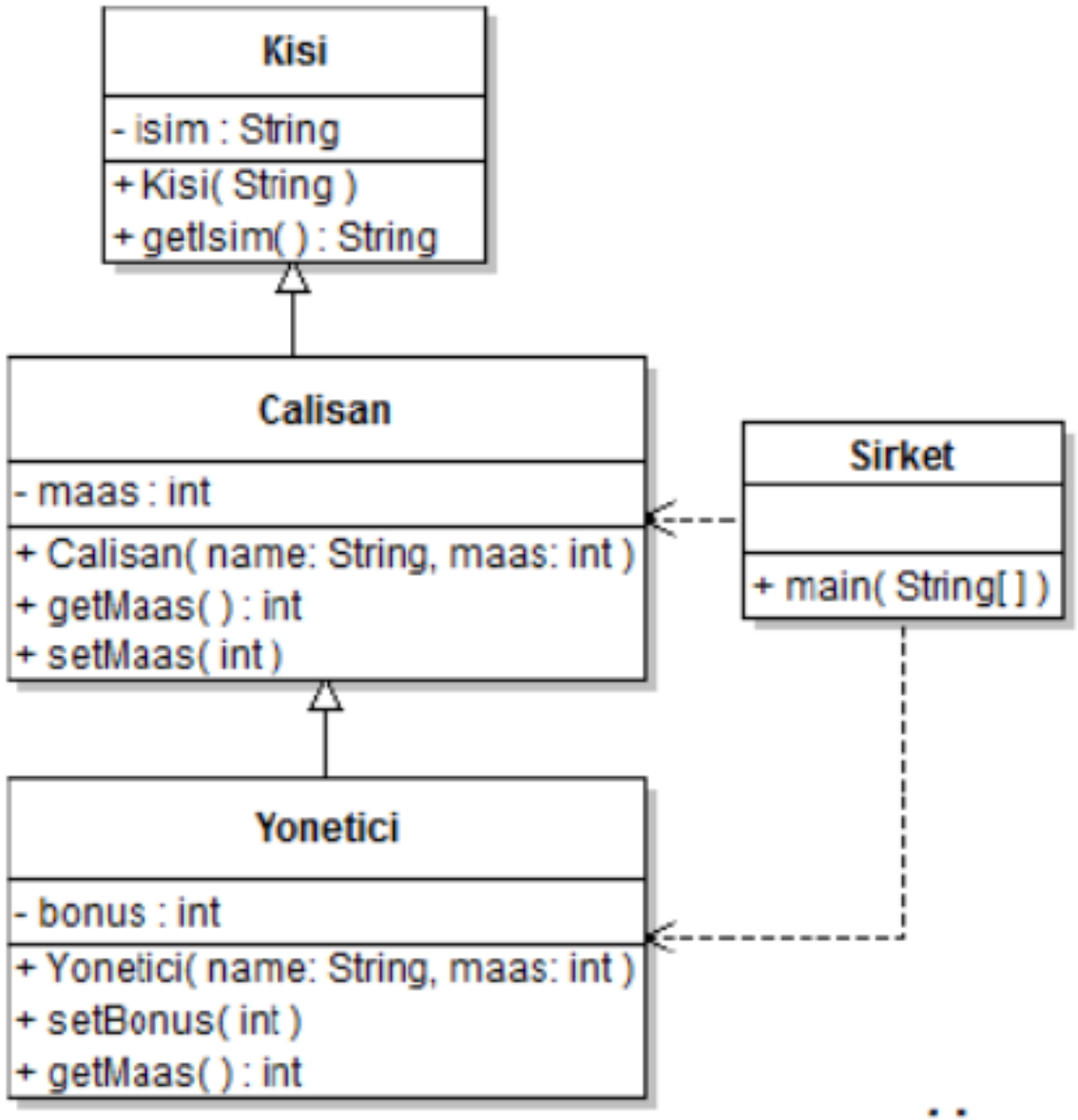


- Örnek alan modeli soldadır.
  - kendiniTanıt() metodu alt sınıflarda yeniden tanımlanmıştır.

- İnsan türünden bir dizi düşünelim, elemanları İnsan ve alt sınıflarından karışık nesneler olsun. Dizinin tüm elemanlarına kendini tanıt dediğimizde ne olacak?
  - Çalışma anında doğru sınıfın metodu seçilir.
  - Bu çalışma biçimine de çok biçimlilik (polymorphism) denir.
- Peki, üst sınıfın altta yeniden tanımladığımız bir metoduna eski yani üst sınıftaki hali ile erişmek istediğimizde ne yapacağız?
  - Bu durumda da super işaretçisi ile üst sınıfa erişebiliriz!

ÇOK BİÇİMLİLİK (POLYMORPHISM) ve YENİDEN TANIMLAMA (OVERRIDING)

- Örnek kalıtım ağacı: Kişi – Çalışan – Yönetici
  - Ve bunları kullanan sınıf: Şirket
  - UML sınıf şeması:



ÇOK BİÇİMLİLİK (POLYMORPHISM) ve YENİDEN TANIMLAMA (OVERRIDING)

- Kaynak kodlar:

```
package ders04;
public class Kisi {
        private String isim;
        public Kisi( String name ) { this.isim = name; }
        public String getIsim( ) { return isim; }
}
```

```
package ders04;
public class Calisan extends Kisi {
        private int maas;
        public Calisan( String name, int maas ) {
                super( name );
                this.maas = maas;
        }
        public int getMaas( ) { return maas; }
        public void setMaas( int salary ) { this.maas = salary; }
}
```

ÇOK BİÇİMLİLİK (POLYMORPHISM) ve YENİDEN TANIMLAMA (OVERRIDING)

- Kaynak kodlar (devam):

```
package ders04;

public class Yonetici extends Calisan {
        private int bonus;

        public Yonetici( String name, int maas ) {
                super( name, maas );
                bonus = 0;
        }

        public void setBonus( int bonus ) {
                this.bonus = bonus;
        }

        public int getMaas( ) {
                return super.getMaas( ) + bonus;
        }

}
```

DİKKAT: super.super olmaz!!!

ÇOK BİÇİMLİLİK (POLYMORPHISM) ve YENİDEN TANIMLAMA (OVERRIDING)

- Kaynak kodlar (devam):

```
package ders04;
public class Sirket {
   public static void main(String[] args) {
            Calisan[] calisanlar = new Calisan[3];;
            Yonetici mudur = new Yonetici( "Oktay Sinanoğlu", 10000 );
            mudur.setBonus( 2500 );
            calisanlar[0] = mudur;
            calisanlar[1] = new Calisan( "Attila İlhan", 7500 );
            calisanlar[2] = new Calisan( "Ümit Zileli", 6000 );
            for( Calisan calisan : calisanlar )
                        System.out.println( calisan.getIsim() + " " +
                                    calisan.getMaas( ) );
   }
}
```

ADAŞ METOTLAR / ÇOKLU ANLAM YÜKLEME (OVERLOADING)

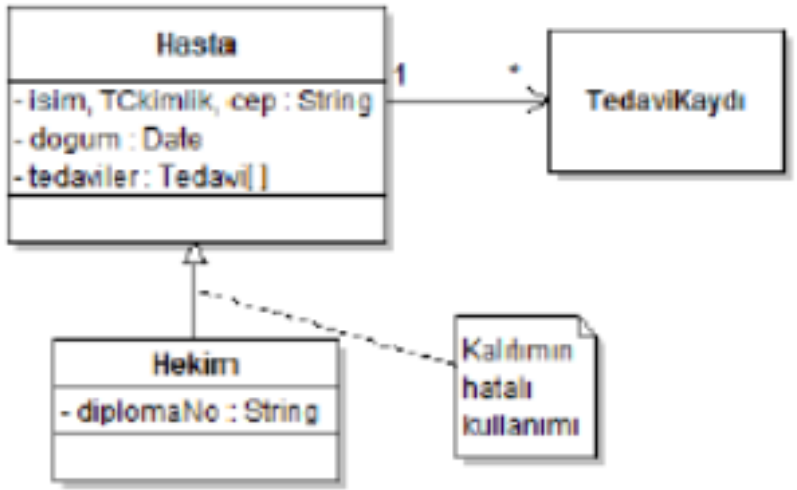
- Bir sınıfın aynı adlı ancak farklı imzalı metotlara sahip olabileceğini gördük.
- Böyle metotlara adaş metotlar, bu işleme ise çoklu anlam yükleme (overloading) adı verilir.
- Örnek: Çok biçimlilik konusu örneğindeki Yönetici sınıfına bir yapılandırıcı daha ekleyelim:
  - Yonetici(String name, int maas, int bonus)

```
public Yonetici( String name, int maas, int bonus ) {
        super( name, maas );
        this.bonus = bonus;
}
```

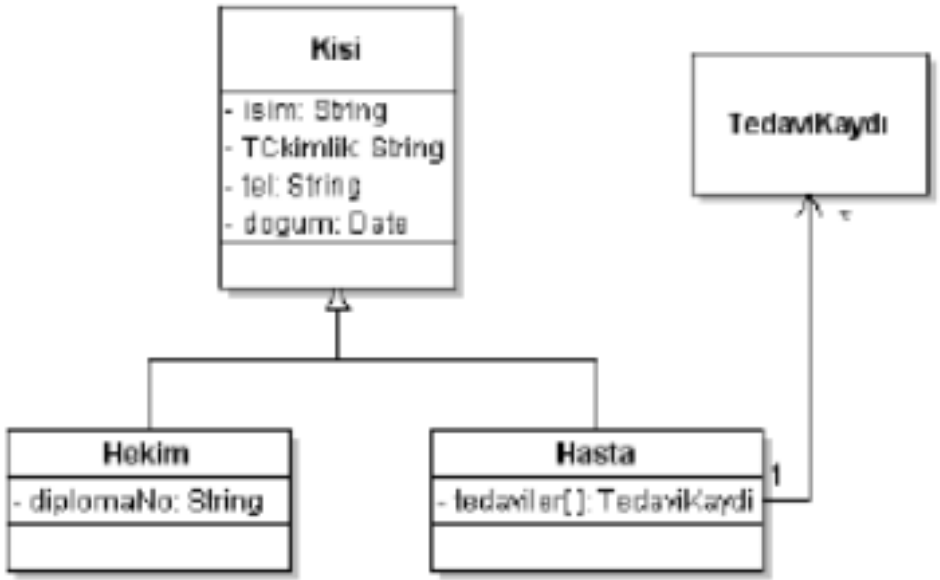
  - Böylece yapılandırıcıya çoklu anlam yüklemiş olduk.
  - Bu kez de bu yapılandırıcıyı kullanacak kişi, maaş ile bonus'u birbirine karıştırmamalı.
- DİKKAT: Çoklu anlam yüklemenin kalıtımla bir ilgisi yoktur. Kalıtım olmadan da adaş metotlar oluşturulabilir, ancak kalıtım olmadan çok biçimlilik ve yeniden tanımlama mümkün değildir.

KALITIM

- Gereksinim:
  - Hastaların isimleri, TC kimlik no.ları, doğum tarihleri ve cep telefonları saklanmalıdır. Bu bilgiler diş hekimleri için de saklanmalıdır. Hekimlerin diploma numaralarının saklanması ise kanun gereği zorunludur. Hangi hastanın hangi tarihte hangi hekim tarafından hangi tedaviye tabi tutulduğu sistemden sorgulanabilmelidir.
- Kalıtımın yanlış kullanımı:



Kalıtımın doğru kullanımı:



- Hekimlerin tedavi kaydının tutulması gerekmemektedir. Yanlış kullanımda her hekim aynı zamanda bir hasta olduğu için, tedavi kaydı bilgisini de alır.

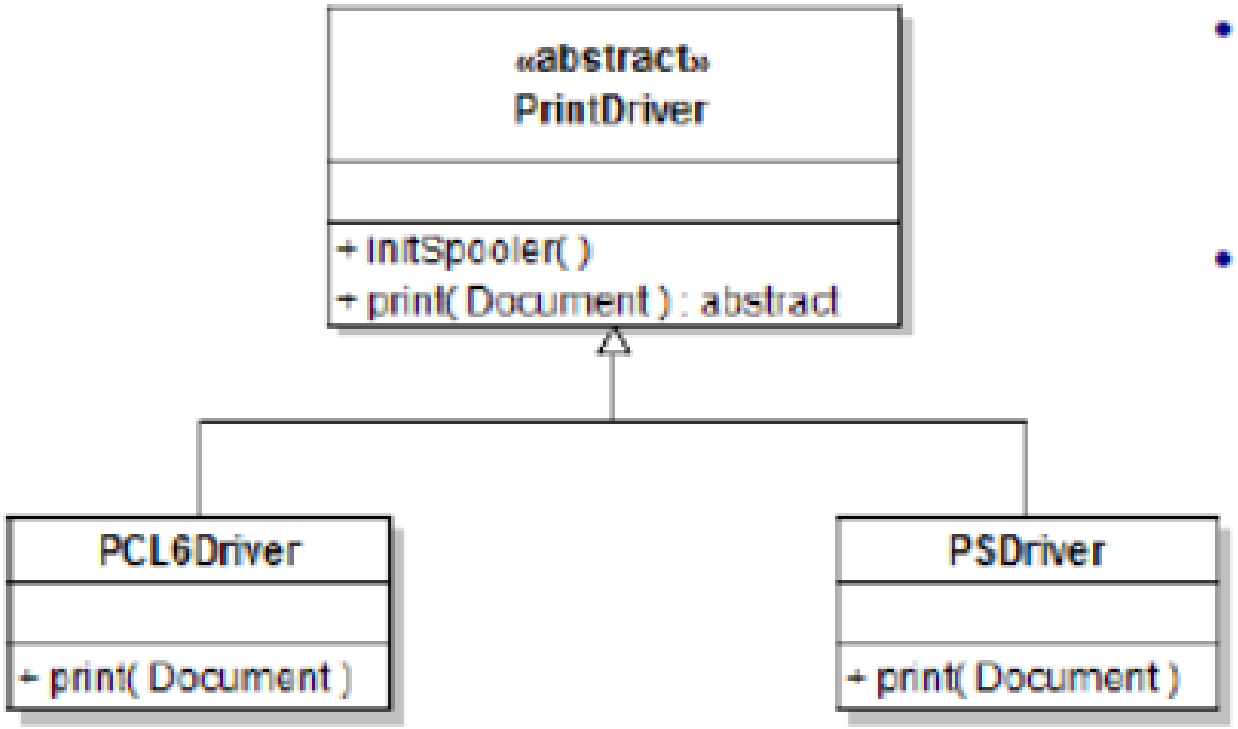
SOYUT SINIFLAR

- Soyut sınıflar, kendilerinden kalıtım ile yeni normal alt sınıflar oluşturmak suretiyle kullanılan, bir çeşit şablon niteliğinde olan sınıflardır.
  - Şimdiye kadar kodladığımız normal sınıflara İngilizce *concrete* de denir.
  - Eğer bir sınıfı soyut yapmak istiyorsak, onu **`abstract`** anahtar kelimesi ile tanımlarız.
- Soyut sınıflardan nesne oluşturulamaz.
- Ancak soyut sınıfın normal alt sınıflarından nesneler oluşturulabilir.
- Soyut sınıflar da normal sınıflar gibi üye alanlar içerebilir.
- Soyut sınıfın metotları soyut veya normal olabilir:
  - Soyut metotların abstract anahtar kelimesi de kullanılarak sadece imzası tanımlanır, gövdeleri tanımlanmaz.
  - Bir soyut sınıfta soyut ve normal metotlar bir arada olabilir.
- Soyut üst sınıflardaki soyut metotların gövdeleri, normal alt sınıflarda mutlaka yeniden tanımlanmalıdır.
  - Aksi halde o alt sınıflar da soyut olarak tanımlanmalıdır.

SOYUT SINIFLAR

- Ne zaman soyut sınıflara gereksinim duyulur:
  - Bir sınıf hiyerarşisinde yukarı çıkıldıkça sınıflar genelleşir. Sınıf o kadar genelleşmiş ve kelime anlamıyla soyutlaşmıştır ki, nesnelere o açıdan bakmak gerekmez.
  - Soyut sınıfları bir şablon, bir kalıp gibi kullanabileceğimizden söz açmıştık. Bu durumda:
    - Bir sınıf grubunda bazı metotların mutlaka olmasını şart koşuyorsanız, bu metotları bir soyut üst sınıfta tanımlar ve söz konusu sınıfları ile bu soyut sınıf arasında kalıtım ilişkisi kurarsınız.
- Soyut sınıfların adı sağa yatık olarak yazılır ancak gösterimde sorun çıkarsa <<STEREOTYPE>> gösterimi.
  - <<...>>: Bir sembol anlamı dışında kullanılmışsa.

SOYUT SINIFLAR



- Üst sınıfta bir yazdırma işleminin olması gerektiği biliniyor ama bu işin nasıl yapılacağı bilinmiyor.
- Bu nedenle PrintDriver nesneleri bir işimize yaramaz.
  - Sadece yazdırma biriktiricisinin (spooler) nasıl ilklendirileceğinin kodunu yazma yükümlülüğünü üzerimizden alır.

- Yazdırma işlemin nasıl yapılacağı alt sınıflarda tanımlanmıştır.
- Tasarımımız, PCL6 ve PS tiplerinden ve hatta ortaya çıkacak yeni yazdırma tiplerinden birden fazla sürücünün bir bilgisayarda kurulu olmasına ve hepsine ortak bir PrintDriver sınıfı üzerinden erişilmesine izin verir.

SOYUT SINIFLAR

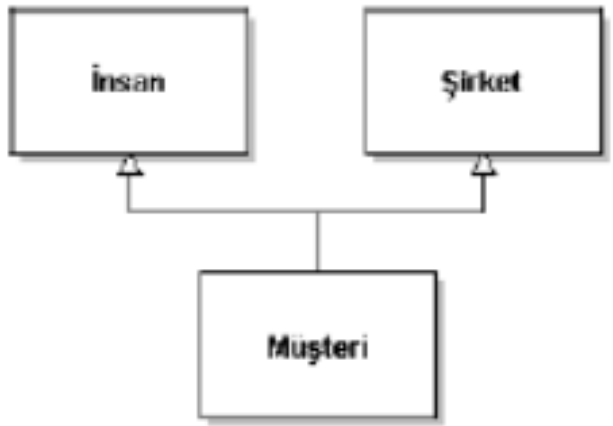
- Kaynak kodlar:

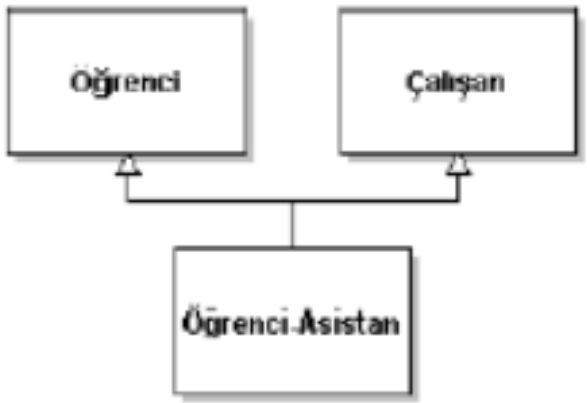
```
package ooc06;
public abstract class PrintDriver {
    public void initSpooler( ) {
       /* necessary codes*/
    }
    public abstract void print( Document doc );
}
```

```
package ooc06;
public class PCL6Driver extends PrintDriver {
    public void print(Document doc) {
       //necessary code is inserted here
    }
}
```

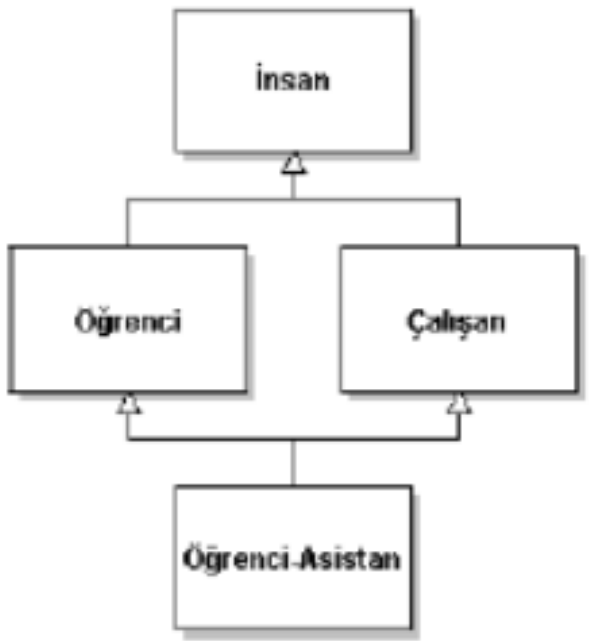
ÇOKLU KALITIM

- Bir sınıfın birden fazla üst sınıftan kalıtım yolu ile türetilmesi.
- Alt sınıfın, birden fazla üst sınıfın özelliğini taşıması anlamına gelir.



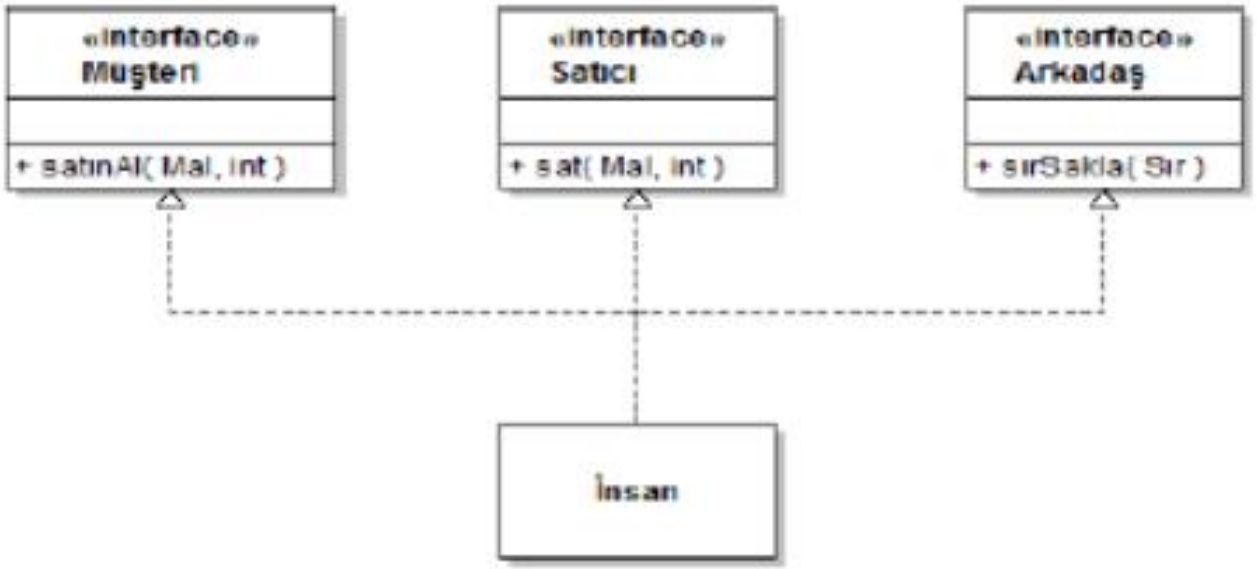


- Çoklu kalıtım ile ilgili sorunlar:
  - Kalıtım çevrimi (Diamond problem): Orta düzeyde çokbiçimlilik varsa alt düzeyde çokbiçimli metodun hangi sürümü çalışacak?
  - Her dil desteklemez. Ör: Java, C#



ARAYÜZLER (INTERFACE)

- Üye alanları olmayan ve tüm metotları soyut olan bir soyut sınıf gibi görülebilir.
  - Eğer isterseniz "public final static" üye alan ekleyebilirsiniz.
- Bir ad altında derlenmiş metotlar topluluğudur.
- Bir örnek üzerinden UML gösterimi:



- Bir sınıf, gerçeklediği arayüzdeki tanımlı tüm metotların gövdelerini tanımlamak zorundadır.

- Kodlama:

```
public interface Müşteri {
   public void satınAl( Mal mal, int adet );
}
public class İnsan implements Müşteri,
                Satıcı, Arkadaş {
   public void satinAl( Mal mal, int adet ) {
                //ilgili kodlar
   }
   public void sat ( Mal mal, int adet ) {
                //ilgili kodlar
   }
  public void sırSakla( Sır birSır ) {
                //ilgili kodlar
   }
}
```

ARAYÜZLER (INTERFACE)

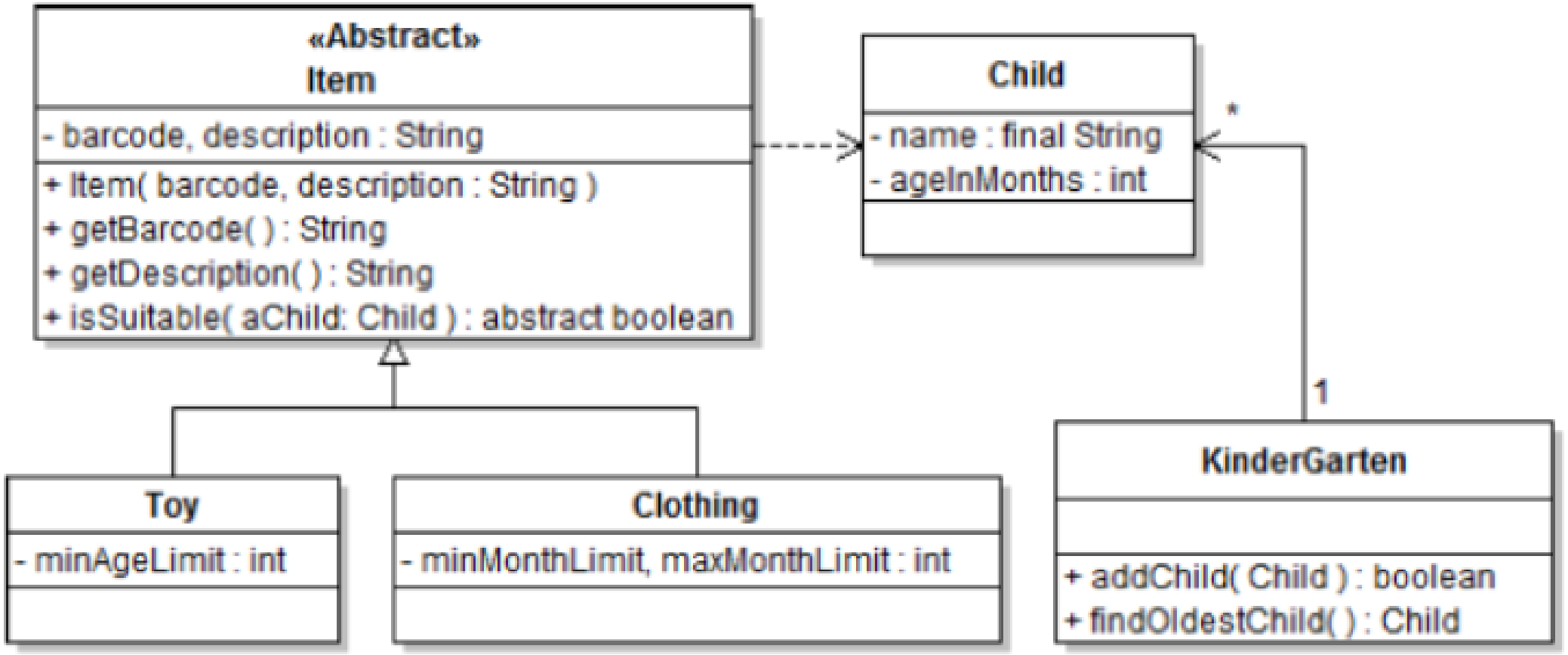
- Arayüzler neye yarayabilir?
  - Nesnenin sorumluluklarını gruplamaya.
  - Nesneye birden fazla bakış açısı kazandırmaya:
    - Farklı tür nesneler aynı nesneyi sadece kendilerini ilgilendiren açılardan ele alabilir.
    - Farklı tür nesneler aynı nesneye farklı yetkilerle ulaşabilir.
  - Kalıtımın yerine kullanılabilme:
    - Çünkü kalıtım "ağır sıklet" bir ilişkidir. Bu yüzden sadece çok gerektiğinde kullanılması önerilir.
  - Çoklu kalıtımın yerine kullanılabilme.

ARAYÜZLER (INTERFACE)

- Arayüzler ile ilgili kurallar:
  - Bir sınıf, gerçeklediği arayüzdeki tanımlı tüm metotların gövdelerini tanımlamak zorundadır.
  - Arayüzlerde normal üye alanlar tanımlanamaz, sadece "public final static" üye alanlar tanımlanabilir.
  - Arayüzlerde sadece public metotlar tanımlanabilir.
  - Arayüzülerin kurucusu olmaz.
  - Bir sınıf birden fazla arayüz gerçekleyebilir.

SOYUT BİR SINIF TASARLAMAK VE KODLAMAK

- Çocuk malzemelerini düşünün:
  - Her malzeme her yaştan çocuğa uygun değildir.
    - Oyuncakların ay türünden olmak üzere bir minimum yaş sınırı vardır.
    - Giysilerin ise yıl türünden minimum ve maksimum yaş sınırları vardır.
  - Bu durumu nasıl modellemeli?



SOYUT BİR SINIF TASARLAMAK VE KODLAMAK

- Item (malzeme) sınıfının kaynak kodu:

```
package ooc06;
public abstract class Item {
    private String barcode, description;
    public Item(String barcode, String description) {
        this.barcode = barcode;
        this.description = description;
    }
    public String getBarcode() {
        return barcode;
    }
    public String getDescription() {
        return description;
    }
    public abstract boolean isSuitable(Child aChild);
}
```

- Bir malzemenin uygunluğunun belirlenmesi için kullanılması gereken mantık farklı olduğu için, isSuitable (uygunMu) metodunu burada soyut tanımladık.
- Ancak her tür malzeme için ortak olan işlemleri bu soyut üst sınıfta kodladık ki bunları alt sınıflarda boş yere aynen tekrarlamak zorunda kalmayalım.

SOYUT BİR SINIF TASARLAMAK VE KODLAMAK

- Soyut olmayan alt sınıfların kaynak kodları:

```
package ooc06;
public class Clothing extends Item {
    private int minMonthLimit, maxMonthLimit;

    public Clothing(String barcode, String description,
                    int minMonthLimit, int maxMonthLimit ) {
        super(barcode, description);
        this.minMonthLimit = minMonthLimit;
        this.maxMonthLimit = maxMonthLimit;
    }
    public boolean isSuitable(Child aChild) {
        if( aChild.getAgeInMonths() >= minMonthLimit
                && aChild.getAgeInMonths() <= maxMonthLimit )
            return true;
        return false;
    }
}
```

SOYUT BİR SINIF TASARLAMAK VE KODLAMAK

- Soyut olmayan alt sınıfların kaynak kodları:

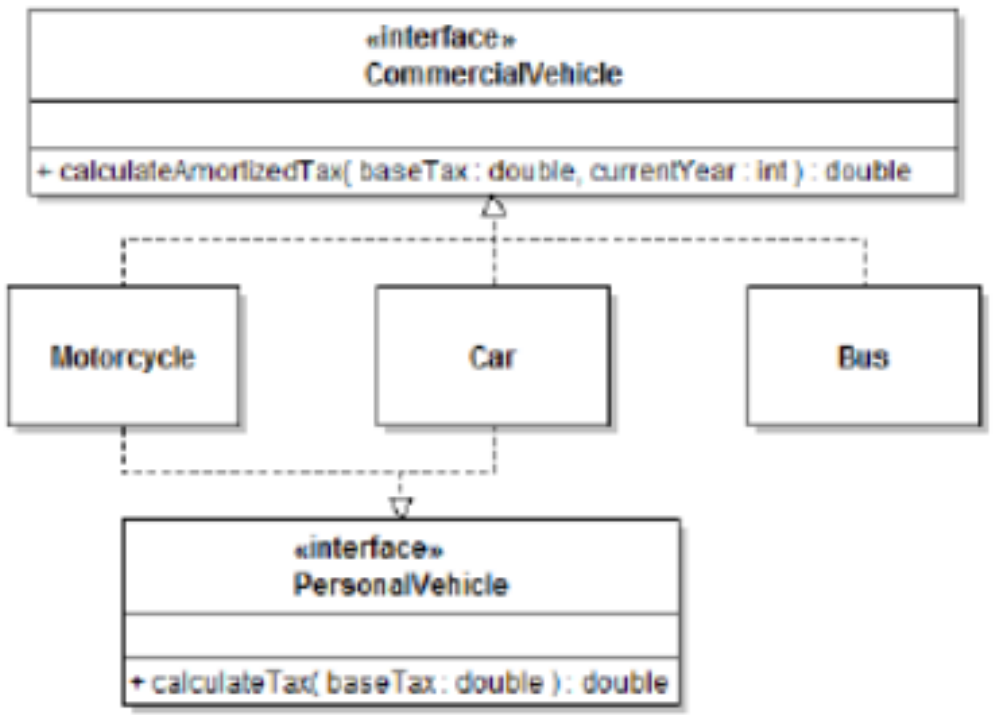
```
package ooc06;
public class Toy extends Item {
    private int minAgeLimit;

    public Toy(String barcode, String description, int minAgeLimit) {
        super(barcode, description);
        this.minAgeLimit = minAgeLimit;
    }
    public boolean isSuitable(Child aChild) {
        if( aChild.getAgeInMonths()/12 >= minAgeLimit )
            return true;
        return false;
    }
}
```

- Kindergarten (AnaOkulu) sınıfının kaynak kodunu UML sınıfında verildiği kadarıyla kodlayıp yapılan tasarımı yeni özelliklerle geliştirme işini alıştırma olarak yapabilirsiniz.

BİR ARAYÜZ TASARLAMAK VE KODLAMAK

- Araçların vergilendirilmesi ile ilgili olarak şu gereksinimler verilmiştir:
  - Ticari ve şahsi araçlar farklı şekilde vergilendirilir.
  - Motosikletler, arabalar ve otobüsler ticari araç olarak kayıt edilebilir.
  - Sadece motosikletler ve arabalar şahsi araç olarak kayıt edilebilir.
  - Sadece ticari araçların vergilerinden amortisman düşülebilir.
  - Ticari veya şahsi olmalarından bağımsız olarak farklı tür araçların vergilendirilmesi farklıdır.
- Bu gereksinimleri nasıl modelleyebiliriz?



- Not: Eğer farklı tür araçların vergilendirilmesi benzer olsaydı, arayüz yerine önceki örnekteki gibi soyut üst sınıf kullanımı daha doğru olurdu.

BİR ARAYÜZ TASARLAMAK VE KODLAMAK

- Arayüzlerin kodlanması:

```
package ooc07;
public interface CommercialVehicle {
    public double calculateAmortizedTax( double baseTax, int currentYear );
}
```

```
package ooc07;
public interface PersonalVehicle {
    public double calculateTax( double baseTax );
}
```

BİR ARAYÜZ TASARLAMAK VE KODLAMAK

- Araba sınıfının kodlanması

```
package ooc07;
public class Car implements CommercialVehicle, PersonalVehicle {
    private int modelYear;
    private double engineVolume;
    public Car(int modelYear, double engineVolume) {
        this.modelYear = modelYear;
        this.engineVolume = engineVolume;
    }
    public double calculateTax( double baseTax ) {
        return baseTax * engineVolume;
    }
    public double calculateAmortizedTax( double baseTax, int currentYear ) {
        //Tax can be reduced %10 for each year as amortization
        int age = currentYear - modelYear;
        if( age < 10 )
            return baseTax * engineVolume * (1-age*0.10);
        return baseTax * engineVolume * 0.10;
    }
    public int getModelYear() { return modelYear; }
    public double getEngineVolume() { return engineVolume; }
}
```

BİR ARAYÜZ TASARLAMAK VE KODLAMAK

- Otobüs sınıfının kodlanması

```
package ooc07;
public class Bus implements CommercialVehicle {
    private int modelYear;
    private double tonnage;
    public Bus(int modelYear, double tonnage) {
           this.modelYear = modelYear; this.tonnage = tonnage;
    }
    public double calculateAmortizedTax( double baseTax, int currentYear ) {
           double ratioT, ratioA;
           if( tonnage < 1.0 )
               ratioT = 1.0;
           else if( tonnage < 5.0 )
               ratioT = 1.2;
           else if( tonnage < 10.0 )
               ratioT = 1.4;
           else
               ratioT = 1.6;
           ratioA = (currentYear - modelYear) * 0.05;
           if( ratioA > 2.0 )
               ratioA = 2.0;
           return baseTax * ratioT * ratioA;
    }
    public int getModelYear() { return modelYear; }
    public double getEngineVolume() { return tonnage; }
}
```

ARAYÜZLER İLE SOYUT SINIFLAR ARASINDA TERCİH YAPMAK

- Eğer farklı tür araçların vergilendirilmesi benzer olsaydı, yani aynı formülde farklı katsayılar kullanılarak hesaplanabilseydi (parametrize edilebilseydi) arayüzler yerine iki soyut üst sınıf kullanımı daha doğru olurdu.
- Benzer şekilde, ticari ve şahsi araçların vergilendirilmesi parametrize edilebilseydi, sadece bir soyut üst sınıf tanımlayıp uygun metot parametrelerinin seçimi daha doğru olurdu.
- Bu durumlar alıştırma olarak sizlere bırakılmıştır.

Kalıtım (Inheritance)

- Nesne Yönelimli Programlama dillerinde kalıtım olgusu, bir sınıfta (class) tanımlanmış değişkenlerin ve/veya metotların (fonksiyon, procedure) yeniden tanımlanmasına gerek olmaksızın yeni bir sınıfa taşınabilmesidir.
- Bunun için yapılan iş, bir sınıftan bir alt-sınıf (subclass) türetmektir.
- Türetilen alt-sınıf, üst-sınıfta tanımlı olan bütün değişkenlere ve metotlara sahip olur. Bu özeliğe kalıtım özeliği (inheritance) denir.

Kalıtım (Inheritance)

- Programcı, yeni alt-sınıfları tanımlarken, üst-sınıftan (superclass) kalıtsal olarak geleceklere ek olarak, kendisine gerekli olan başka değişken ve metotları da tanımlayabilir.
- Bu yolla, bir kez kurulmuş olan sınıfın tekrar tekrar kullanılması olanaklı olur. Böylece, programlar daha kısa olur, programın yazılma zamanı azalır ve gerektiğinde değiştirilmesi ve onarılması (debug) kolay olur.
- Alt-sınıf türetme hiyerarşik bir yapıda olur. Bir alt-sınıfın türetildiği sınıf, o alt-sınıfın üst-sınıfıdır. Java'da bir alt-sınıfın ancak bir tane üst-sınıfı olabilir (C++ 'dakinden farklı olduğuna dikkat ediniz). Ama bir sınıftan birden çok alt-sınıf türetilebilir.
- Üst-sınıfa ata (parent), alt-sınıfa da oğul (child) denir.

```
class A {
  int i, j;
  void showij() {
    System.out.println("i and j: " + i + " " + j);
  }
}

class B extends A {
  int k;
  void showk() {
    System.out.println("k: " + k);
  }
  void sum() {
    System.out.println("i+j+k: " + (i+j+k));
  }
}
```

```
class BasitInheritance {
  public static void main(String args[]) {
    A ustOb = new A();      // A ‘ya ait nesne
    B altOb = new B();      // B ’ye ait nesne
    ustOb.i = 15;
    ustOb.j = 25;
    System.out.println(" ustOb nesnesinin öğeleri: ");
    ustOb.showij();   //ustOb ‘nin öğelerini yazar
    System.out.println();
    altOb.i = 3;
    altOb.j = 5;
    altOb.k = 7;
    System.out.println("altOb nesnesinin öğeleri: ");
    altOb.showij();   //üst-sınıftaki değişkenleri yazar
    altOb.showk();    //alt-sınıftaki değişkeni yazar
    System.out.println();
    System.out.println("üst ve alt sınıftaki değişkenler toplanıyor… ");
    System.out.println("(i + j + k) = ");
    altOb.sum();
  }
}
```

```
class A {
  int i;              // erişim kısıtı yok
  private int j;      // erişim kısıtlanıyor
  void setij(int x, int y) {
    i = x;
    j = y;
  }
}
class B extends A {
  int toplam;
  void sum() {
    total = i + j; // HATA!, alt-sınıftaki bu
kod j ye erişemez
  }   }
```

```
class DegerVer {
  public static void main(String args[]) {
    B subOb = new B();
    subOb.setij(15, 20);
    subOb.sum();
    System.out.println("Toplam : " +
                                  subOb.toplam);
  }
}
```

super() metodu

- Bir alt-sınıf ne zaman üst-sınıfına erişmek isterse super anahtar sözcüğünü kullanabilir.
- super ‘in kullanımı iki türlü olur.
  - üst-sınıfa ait nesne yaratmak içindir.
  - üst-sınıfın öğelerine erişmek içindir.

```
class Kutu {
double en;
double boy;
double yukseklik;
 Kutu(double e, double b, double y)
{
      en = e;
      boy = b;
      yukseklik = y;
}
}
```

```
class AltKutu extends Kutu {
  double agr;
AltKutu(double e, double b,
                double y, double a) {
     super(e, b, y);
     agr = a;
   }
}
```

```
class Kutu {
  private double en;
  private double boy;
  private double yukseklik;
  Kutu(Kutu ob) {
    Width  = ob.en;
    boy    = ob.boy;
    yukseklik  = ob.yukseklik;
  }
  Kutu(double e, double b, double y) {
    en  = e;
    boy = b;
    yukseklik  = y;
  }
  Kutu() {
    en  = -1;        // geçici değer
    boy = -1;        // geçici değer
    yukseklik  = -1;  // geçici değer
  }
  Kutu(double uzunluk) {
    en = boy = yukseklik = uzunluk;
  }
  double hacim() {
    return en * boy * yukseklik;
  }
}
```

```
class AltKutu extends Kutu {
  double agr; // kutu'nun ağırlığı
  AltKutu(AltKutu ob) {
    super(ob);
    agr = ob.agr;
  }

  AltKutu(double e, double b,
                double y, double a) {
    super(e, b, y); agr = a;
  }

  AltKutu() {
     super();
     agr = -1;
   }

   AltKutu(double uzunluk, double a) {
     super(uzunluk);
     agr = a;
   }
 }
```

```
class ColorKutu extends Kutu {
  int color;                // Kutu'ın rengi

  ColorKutu(double e, double b, double y, int c) {
    en = e;
    boy = b;
    yukseklik = y;
    color = c;
  }
}
```

```
class A {
  int i, j;
  A(int a, int b) {
    i = a;
    j = b;
  }
  void show() {
    System.out.println("i and j: " + i + " " + j);
  }
}

class B extends A {
  int k;
  B(int a, int b, int c) {
    super(a, b);
    k = c;
  }
  void show() {
    System.out.println("k: " + k);
  }
}

class Override {
  public static void main(String args[]) {
    B subOb = new B(1, 2, 3);
    subOb.show();
  }
}
```

Interfaces(Arayüzler)

- Kalıtım OOP bir dilde olmazsa olmazlardandır.
- Bir nesne tanımlandıktan sonra bu nesneden başka bir nesne daha türetebilirsiniz.
- Fakat ya iki ya da daha çok nesnenin bir araya gelip bir nesne oluşturmasını istendiğinde java çoklu kalıtımı desteklemez.
- Bir nesneyi yalnızca bir tane nesneden türetebilirsiniz.
- Bu aşamada Interfaces (Arayüzler) kullanılabilir.
- Java'da arayüzler **interface** anahtar sözcüğü ile tanımlanmakta ve **implements** anahtar sözcüğü ile de uygulanmaktadır.

```
public class Main
{
    interface IKendiniTanit
    {
        void IsminiYaz();
    }
    interface IKendiniDetayliTanit extends IKendiniTanit
    {
        void NumaraniYaz();
        void SoyadiniYaz();
    }
    interface MaasIslemleri
    {
        double MaasHesapla();
    }
    abstract static class KISI implements IKendiniDetayliTanit, MaasIslemleri
    {
        public String Isim="";
        public String Soyad="";
        public String Numara="";
        public double TabanMaas = 3000.0;
        public double MaasKatsayi=0.0;
        //
        KISI(double MaasKatsayim, String Ismim, String Soyadim, String Numaram)
        {
            this.Isim = Ismim;
            this.Soyad = Soyadim;
            this.Numara = Numaram;
            this.MaasKatsayi = MaasKatsayim;
        }
```

```
    public void NumaraniYaz()
    {
        System.out.println(this.Numara);
    }
    public void SoyadiniYaz()
    {
        System.out.println(this.Soyad);
    }
    public void IsminiYaz()
    {
        System.out.println(this.Isim);
    }
    public void TumBilgileriDok()
    {
        System.out.println("isim: "+this.Isim+", soyad: "+this.Soyad+", numara: "+this.Numara);
    }
}
static class GenelMudur extends KISI
{
    GenelMudur(String Ismim, String Soyadim, String Numaram)
    {
        super(2.5, Ismim, Soyadim, Numaram);
    }
    public double MaasHesapla()
    {
        return(this.TabanMaas*this.MaasKatsayi + 2000.0);
    }
}
```

```
static class Mudur extends KISI
{
    Mudur(String Ismim, String Soyadim, String Numaram)
    {
        super(1.5, Ismim, Soyadim, Numaram);
    }
    public double MaasHesapla()
    {
        return(this.TabanMaas*this.MaasKatsayi + 1000.0);
    }
}
static class Muhendis extends KISI
{
    Muhendis(String Ismim, String Soyadim, String Numaram)
    {
        super(1.0, Ismim, Soyadim, Numaram);
    }
    public double MaasHesapla()
    {
        return(this.TabanMaas*this.MaasKatsayi + 500.0);
    }
}
public static void main(String[] args)
{
    GenelMudur GenelMudurum = new GenelMudur("Cem","Kefeli","1111111");
    Mudur Mudurum = new Mudur("Onur","Kelebek","2222222");
    Muhendis Muhendisim = new Muhendis("Meltem","Uyanir","5555555");
    GenelMudurum.TumBilgileriDok();
    System.out.println("GenelMudur maasi:"+GenelMudurum.MaasHesapla());
    Mudurum.TumBilgileriDok();
    System.out.println("Mudur maasi:"+Mudurum.MaasHesapla());
    Muhendisim.TumBilgileriDok();
    System.out.println("Muhendis maasi:"+Muhendisim.MaasHesapla());
}
```

Soyut Sınıflar (Abstract Classes)

- Bazı durumlarda, yapılacak işlere uyan somut bir üst-sınıf ve ona ait somut metotlar tanımlamak mümkün olmayabilir.
- Böyle durumlarda, soyut bir üst-sınıf ve ona ait soyut metotlar tanımlamak sorunu kolayca çözebilir.
- Soyut metot (abstract method) adı ve parametreleri olduğu halde gövdesi olmayan bir metottur.
- Dolayısıyla belirli bir iş yapmaz. O, alt-sınıflarda örtülür (overriding). Sözdizimi şöyledir:
- abstract [erişim_belirtkesi] [veri_tipi] metot_adı(parametre_listesi);

Soyut Sınıflar (Abstract Classes)

- Her soyut sınıfın en az bir tane soyut metodu olmalıdır.
- Bu nedenle, soyut bir sınıfa ait nesne yaratılamaz (instantiate edilemez).
- Tersine olarak, içinde soyut bir metot olan her sınıf soyut bir sınıf olur.
- Soyut sınıfın gövdesinde constructor metodu, soyut metot(lar) ve gerekiyorsa somut metotlar yer alabilir. Sözdizimi şöyledir:

```
[Erişim_belirtkesi] abstract class sınıf_adı {

        // soyut sınıf içinde en az bir soyut bir metot yer almalıdır
        // soyut sınıf içinde somut metot olabilir
        // constructor metodu tanımlanabilir
}
```

```
abstract class A {
    abstract void soyMet();
    void somMet() {
        System.out.println("Ben
            somut bir metodum.");
    }
}
class B extends A {
  void soyMet() {
    System.out.println("Sen soyut bir
                        metotsun.");
  }
}
```

```
class UygulamaPrg {
  public static void main(String args[]) {
    B b = new B();

    b.soyMet();
    b.somMet();
  }
}
```

```
abstract class Sekil {
  double boyut1;
  double boyut2;

  Sekil(double a, double b) {
    boyut1 = a;
    boyut2 = b;
  }

  // soyut metot
  abstract double alan();
}
```

```
class Dikdortgen extends Sekil {
  Dikdortgen(double a, double b) {
    super(a, b);
  }

   // Dikdörtgen alanı için override
  double alan() {
    System.out.println("Dikdortgenin alanı .");
    return boyut1 * boyut2;
  }
}
```

```
class Ucgen extends Sekil {
  Ucgen(double a, double b) {
    super(a, b);
  }

        // Üçgen alanı için override
  double alan() {
    System.out.println("Ucgenin alanı .");
    return boyut1 * boyut2 / 2;
  }
}
```

```
class SoyutAlanBul {
  public static void main(String args[]) {
   // Sekil f = new Sekil(10, 10);        // deyim geçersizdir
  // soyut sınıfa ait nesne yaratılamaz
    Dikdortgen r = new Dikdortgen(9, 5);
    Ucgen t = new Ucgen(10, 8);

    Sekil ref; // referans değişkeni tanımlıyor; nesne işaret etmiyor
                              // deyim geçerlidir

    ref = r;   // alt-sınıfa ait bir nesneyi işaret ediyor; deyim geçerlidir
    System.out.println("Alan = " + ref.alan());

    ref = t;   // alt-sınıfa ait bir nesneyi işaret ediyor; deyim geçerlidir
    System.out.println("Alan = " + ref.alan());
  }
}
```

Final Anahtar Sözcüğü

- Final anahtar sözcüğünün üç farklı kullanılışı vardır.
  - Sabit tanımlamak için.
  - Bir metodun örtülmesini (overriding) önlemek için.
  - Kalıtımı (inheritance) önlemek için.

Final ile Sabit Tanımlama

- Program boyunca değeri sabit kalacak bir değişken tanımlamak için değişkenin önüne final anahtar sözcüğü konulur.
- Final'in bu işlevi, öteki programlama dillerinde sabit (constant) tanımlama işlevine denktir.
- Java'da final damgalı değişkenlerin adları büyük harflerle yazılır.
- Bu kural, kaynak programa bakanın sabitleri hemen görebilmesine yardım eder:
  - Final int KATSAYI = 15;
  - Final double FIYAT = 385;
  - Final STRİNG = "Amkara başkenttir";

Final ile Bir Metodun örtülmesini (overriding) Önleme

- Overriding özelliği java’da polymorphism olgusunu yaratan çok kullanışlı bir özeliktir.
- Ancak, bazı durumlarda bir sınıfta tanımlı metodun alt sınıflarda değiştirilmemesi gerekebilir.
- Böyle durumlarda, değişmemesi istenen metodun tanımında, önüne final nitelemi getirilir.
- Final damgalı metotlar alt-sınıflarda örtülemez (overriding).

```
class A {
  final void deneme() {
    System.out.println("Bu bir final metot’tur.");
  }
}


class B extends A {
  void deneme() { // HATA! Override edilemez.
    System.out.println("Geçersidir!");
  }
}
```

Final ile Kalıtımı (inheritance) Önleme

- Bazı durumlarda bir sınıfın hiçbir değişikliğe uğratılmadan korunması gerekir.
- Bu durumlarda, o sınıfın hiçbir alt-sınıfının yaratılmaması gerekir. Bunu yapmak için, sınıf tanımında final anahtar sözcüğü kullanılır.
- Final anahtar sözcüğü ile damgalı bir sınıfın metotlar da kendiliğinden final anahtar sözcüğü alırlar.
- Bir sınıf aynı anda hem final hem abstract olamaz. Çünkü, soyut sınıfın (abstract class) alt-sınıfı yaratılamaz ise, onun hiçbir işlevi olamaz.
- Aşağıdaki örnek final nitelemesinin sınıf için nasıl kullanıldığını göstermektedir.

```
final class A {
        // ...
}

// bu alt sınıf yaratılamaz.
class B extends A {         // HATA!
                // ...
}
```