Nesneye Yönelik Programlama

Class Kavramı
Package'ler ve Derleme Birimleri
Encapsulation (Kapsülleme)
Inheritance (Kalıtım)
Interface'ler ve Abstract Class'lar
Çok Biçimlilik (Polymorphism)

Class'lar (Sınıflar)

Class,Property ve Method Kavramları

Yazılım geliştirmenin iki temel bileşeni vardır : **veri** ve **işlev.** Her tür program belli veriler üzerinde belli işlever gerçekleştirir. Nesne, belli bir anlam bütünlüğü içeren veriler ve gerektiğinde o verilerle çalışan işlemlerden oluşan yapıdır. Dikdörgenler prizması diye bir nesne olduğunu düşünelim. Bununla ilgili en, boy ve yükseklik gibi veriler vardır. Ayrıca alanını ve hacmini hesaplama gibi işlemler bulunmaktadır. Java'da veri ve işlem içeren her birime "**class**" (sınıf) denir. Class'ların içerdiği veriye "**property**" (özellik), işlemlere de "**method**" (yöntem) denir. Dikdörtgenler prizması'nı belirten bir nesne yapalım.

```
public class RectangularPrism{
  float width;
  float height;
  float depth;
  float getVolume(){
    return width*height*depth;
  }
  float getArea(){
    return 2*(width*height+width*depth+height*depth);
  }

}
```

Burada yaptığımız, dikdörtgenler prizması diye bir nesne olduğunu, bu nesnenin *width*, *height*, *depth* gibi özellikleri olduğunu, bu nesneyle alan ve hacim hesaplaması yapabileceğimizi söylemekten ibarettir. Ayrıca her property'nin veri tipini de belirttik. Dikdörgenler prizmasının kenarların kesirli sayılarla belirtilebilir.

Instance

Yukarıda bir class'ı tanımladık ('definition' yaptık). Bu class bütün dikdörtgenler prizmalarında olabilecek özellikler ('property'ler) ve yöntemler ('method'lar) içeriyor. Henüz ortada belli bir prizma yok. Belli bir pizmanın "yaratılması" gerekli ki biz hacmini ve alanını bulalım. Class tanımına uygun olarak yaratılan nesnelere **instance** (örnek) denir. Dikdörtgenler prizmasına bir örnek verilmesi, başka bir deyişle bir dikdörtgenler prizması yaratılması lazım. Buna nesneye yönelik programlamadaki terminolojiyi kullanırsak, RectangularPrism class'ından bir instance yaratılması gerekiyor. Bir prizma yaratalım, boyutlarını 10,20 ve 30 olarak verelim. Alanını ve hacmini hesaplayıp ekrana yazalım.

```
    public class RectangularPrismTest{
      public static void main(String[] args){
        RectangularPrism prism=new RectangularPrism();
```

```
        prism.width=10.0f;
        prism.height=20.0f;
        prism.depth=30.0f;
        float volume=prism.getVolume();
        System.out.println("Volume : "+volume);
        float area=prism.getArea();
        System.out.println("Area : "+area);
      }
    }
```

Burada RectangularPrism class'ından bir instance yarattık. İsmi de 'prism'. Bu isim rastgele seçilmiş bir isimdir, kısaca 'p' de diyebilirdik. Ancak bir kez prism dedikten sonra aşağıda aynı ismi kullanmak zorundayız.

Property Intialization

Bir nesne yaratılırken 'property'lerinin değerleri **default** (varsayılan) değerlerine atanır. Property'ler de birer nesneyse (primitive type değilse) '**null**' değerini alır. Bir değer olarak 'null' hiç bir değeri yok demektir. Bir nesneyi yaratıp kullanan kişi bir method'u çağırmak için gerekli olan property'lerin değerlerini önceden vermezse hata durumu oluşur. Örneğin, prism değerinin 'height' property'si verilmezse hacim hesaplanmaya çalışılırsa ne olacak? Nasıl bir hacim değeri üretilecek? Böyle sorunlarla karşılaşmamak için property'lerin '**initialize**' edilmesi gerekebilir. Property'ler belirlenirken

```
        float width=1.0f;

        float height=1.0f;

        float depth=1.0f;
```

şeklinde bir initialization yapılabilir. Böylelikle bir değerin verilmesi unutulsa bile belli bir değer default alınacağından sorun çıkmaz. Height değeri verilmezse hacim 1 çıkar. Alan da 2*(1*1+1*1+1*1)=6 çıkar.

Constructor

Bir nesnenin yaratılma aşamasında bazı işlemlerin yapılması, bazı değerlerin atanması gerekebilir. Bir nesne yaratılırken çağrılan method benzeri yapıya '**constructor**' denir. Constructor aslında bir method değildir. Ama class'la aynı ismi taşıyan, dönüş değeri olmayan bir method olarak da düşünülebilir. Prizmanın boyutlarının daha instance yaratılırken verilmesi bu şekilde sağlanabilir.

```
public class PrismWithConstructor{
  float width;
  float height;
  float depth;
  public PrismWithConstructor(float w,float h,float d){
    width=w;
    height=h;
    depth=d;
  }
  float getVolume(){
    return width*height*depth;
  }
  float getArea(){
    return 2*(width*height+width*depth+height*depth);
  }

}
```

Bu class'ı kullanan bir application yapalım.

```
public class PrismWithConstructorTest{
  public static void main(String[] args){
    PrismWithConstructor prism=new PrismWithConstructor(10.0f,20.0f,30.0f);
    float volume=prism.getVolume();
    System.out.println("Volume : "+volume);
  }
}
```

Method Signature

Bir method'un ismiyle birlikte sırasıyla aldığı parametre tipleri, o method'un '**signature**'unu oluşturur. Siz bir method çağırdığınızda, öncelikle o nesnede o isimde bir method olup olmadığına bakılır. Eğer varsa, çağırdığınız parametreler'in tipleriyle birlikte eşleştirme yapılır ve uygun bir method varsa derleme işlemi başarılı olur. Aksi takdirde kod derlenemez.

Bir nesnenin bir methodunu çağırdığınızda compiler kızıyorsa ve siz method ismini doğru yazdığınıza eminseniz, bilinki verdiğiniz paramtereler ya eksiktir, ya fazladır, ya sırası yanlıştır ya da parametre tipleri yanlıştır.

Overloading (Çeşitleme)

Bir method'la aynı isimde başka bir method yazılabilir. Buna **overloading** (çeşitleme) denir. Bu işlem, benzer işlemleri farklı parameteler'le yapan iki method için yapılır. Bu şekilde, bir çok method ismini öğrenmekten kurtulmuş oluruz. Ancak, overloading işleminde yeni method'un signature'u mutlaka farklı olmalıdır ki bir method çağırdığınıza hangisini çağırdığınız anlaşılabilsin. Zaten aynı isimde aynı parametreleri aynı sırada alan bir ikinci method yazmanın hiç bir anlamı yoktur. Aynı işi yapıyorlarsa, bir tanesine hiç gerek yok. Farklı işler yapıyorlarsa farklı bir isimde olmalıdır.

Bir method'un **return type**'ı signature'a dahil değildir. Yani aynı isimde aynı parametre tiplerini aynı sırada belirten fakat iki ayrı return type olan iki ayrı method olamaz. Java'da bir method'un döndürdüğü değerin fazla bir önemi yoktur. Sadece o method'u çağıran için gerekliyse bir anlamı vardır. Hatta bir değer döndüren method, döndürdüğü değer kullanılmadan da çağrıabilir. Bazı method'lar belki lazım olur diye belli bir değeri döndürmektedirler.

Birden Fazla Constructor

Bir class'ın farklı parametreler alan birden fazla constructor'u olabilir.
Yani constrcutor'lar da method'lar gibi 'overload' edilebilirler.

```
public class PrismWithMultipleConstructor{
  float width;
  float height;
  float depth;
  public PrismWithMultipleConstructor(float w,float h){
    width=w;
    height=h;
    depth=1.0f;
  }
  public PrismWithMultipleConstructor(float w,float h,float d){
    width=w;
    height=h;
    depth=d;
```

```
  }
  float getVolume(){
    return width*height*depth;
  }

}
```

Bir constructor başka bir constructor'u çağırabilir. Bunun için 'this' operatörü kullanılmalıdır.

```
        public PrismWithMultipleConstructor(float w,float h){
                this(w,h,1.0f);
        }
        public PrismWithMultipleConstructor(float w,float h,float d){
                width=w;
                height=h;
                depth=d;
        }
```

Burada iki constrcutor'dan biri diğerini çağırıyor. Kendisine gelen değerleri aynen öbürüne devrederken bir değeri kendisi belirliyor. Bu yöntemin yararı, bir değişiklik yapıldığında sadece tek bir constructor'un değiştirilmesinin yeterli olmasıdır.

Default & Parameter'siz Constructor'lar

Aslında, programcı hiç bir tane yazmasa dahi, her class'ın bir constructor'u vardır. Sanki

```
public class MyClass{
        public MyClass(){
        }
}
```

şeklinde bir constructor verilmiş gibi kabul edilir. Programcı hiç bir şey yapmadığında sağlanan **default constructor** hiç parametre almaz ve hiç bir işlem yapmaz. Hiç parametre almayan ama bazı initilization işlemleri yapan contructor'lar da yazılabilir. O zaman **parameterless constructor** (parametresiz kurucu) olur. Birim küpü 2, 2, 2 boyutlarında kabul ettiğimizi varsayalım.

```
public class PrismWithParameterlesConstructor{
  float width;
  float height;
  float depth;
  public PrismWithParameterlesConstructor(){
    width=2.0f;
    height=2.0f;
    depth=2.0f;
  }
  public PrismWithParameterlesConstructor(float w,float h,float d){
    width=w;
    height=h;
    depth=d;
  }
  float getVolume(){
    return width*height*depth;
  }
```

```
  float getArea(){
    return 2*(width*height+width*depth+height*depth);
  }

}
```

Görüldüğü gibi class'ın iki constructor'u var. Birisi parametreleri dışardan alıyor, diğeri içerde belirliyor. İki constructor'u da test edelim. Birisi default contructor'la birim küp yaratsın, diğeri boyutları vererek.

```
public class PrismWithParameterlesConstructorTest{
  public static void main(String[] args){
    PrismWithParameterlesConstructor prism1=
      new PrismWithParameterlesConstructor();
    PrismWithParameterlesConstructor prism2=
      new PrismWithParameterlesConstructor(10.0f,20.0f,30.0f);
    float volume1=prism1.getVolume();
    float volume2=prism2.getVolume();
    System.out.println("Volume 1: "+volume1);
    System.out.println("Volume 2: "+volume2);
  }
}                                  ,
```

Package'ler ve Derleme Birimleri

Package

Birbiriyle işlev veya kullanım açısından ilişkili class'lar '**package**' adı verilen bir birim altında birleştirilir. Bir class'ın ait olduğu package'i belirtmek için '**package**' keyword'ü kullanılır.

```
package mypackage;

public class MyClass

      // ..

}
```

Bir class'ın diğer bir class'la aynı package'te olması, ona bazı avantajlar sağlar.

Aslında bütün class'lar bir package'e ait olmak zorundadır. Hiç bir isim verilmediğinde o class isimsiz default package'de kabul edilir. Package'i belirtilmemiş bütün classlar aynı package'in class'ları sayılır. Bazı durumlarda sadece örnek olsun diye yapılıp hatalı (derlenemez durumda) bırakılan class, yine package'siz olan başka bir class'ın derlenmesini engelleyebilir.

Import

Bir class'ın kendisiyle aynı package'de olmayan class'ı kullanabilmesi için, class tanımından önce **import** etmesi gerekir. Bu sadece kullanılacan class'ların tam olarak hangi pakette olduğunu söylemek anlamına gelir. Bunun için '**import**' keyword'ü kullanılır.

```
package myanotherpackage;
```

```
import mypackage.MyClass;


public class MyAnotherClass
        // ..
}
```

Burada sadece kullanılacağı belirtilmektedir. Yoksa, import edilen class'ın import eden class'ın üzerine '*include*' edilmesi (eklenmesi) sözkonusu değildir.

'java.lang' package'indeki class'ları kullanmak için import ifadesine gerek yoktur. Bu package olmadan hiç bir Java programı yapılamayacağı için, default olarak import edilmiş sayılır.

'*' Operatörüyle Import

Bir package'deki bütün class'ları import etmek için '*' operatörü kullanılabilir.

```
package myanotherpackage;


import mypackage.*;


public class MyAnotherClass
        // ..
}
```

Bu işlem, altalta çok sayıda import ifadesi yazmaktan daha kolaydır.

'*' operatörü kullanılıdığında bir çok class'ın gereksiz yere import edildiği gibi bir izlenim doğmakla birlikte, import işlemi aslında bir belirtmeden öteye gerçek bir 'içine alma' işlemi anlamına gelmediği için önemli bir performans kaybına yol açmaz. Sadece bazı programcılar okunurluğu arttırmak için böyle bir yola gitmektedirler.

Fully-Qualified Name (Tam Nitelendirilmiş İsim)

Aslında her class package ismiyle birlikte adlandırılır. Örneğin, 'mypackage' isimli bir package'de 'MyClass' isimli bir class'ın tam nitelendirilmiş ismi 'mypackge.MyClass' dır. Import ifadesi, sadece programcıyı her class ismini böyle uzun yazmaktan kurtarmaktadır. 'java.awt' package'inde olan Button adlı class'ın bir instance'ını 'import java.awt.*' diyerek kullanırsak

```
        Button b=new Button();
```

şeklinde yaratabiliriz. Hiç 'import' kullanmayarak aynı ifadeyi

```
        java.awt.Button b=new java.awt.Button();
```

şeklinde de yazabiliriz. Bu şekilde paket ismiyle birlikte verilen class isimlerine **fully-qualified name** denir.

Alt Package'ler

Bazı durumlarda çok fazla class içermemesi için bir package alt birimlere bölünebilir. Örneğin, görsel bileşenler içeren 'java.awt' package'ın image'larla ilşkili olan 'java.awt.image' şeklinde bir alt package'i bulunmaktadır. Aslında alt package'in teknik anlamda üst package'in 'içinde' olması sözkonusu değildir. Alt package isimlendirmesi iki 'ayrı' package'i sadece anlamca ilişkilendirmektedir.

Bir class'ta üst package'i import etmek, otomatikman alt package'lerin de import edileceği anlamına gelmez. Örneğin, 'java.awt.*' ifadesi 'java.awt.event.*' ifadesini içermez. Bu alt package'den herhangi bir class kullanılacaksa ayrıca import edilmesi gerekir.

Standart Package'ler

Java'nın kendi API'ı (Application Programming Interface - Uygulama Programlama Arayüzü'ü) bir çok class'dan oluştuğu için bir çok package altında gruplandırılmıştır. Bunların çoğu 'java.' ve 'javax.' la başlar. 'javax' öneki daha çok, sonradan extension (ekleme) package'ler için kullanılır. Önceden 'extension' olan package'ler daha sonra standart API'a dahil olabilmektedir.

Java'nın standart package'leri benzer ve ilişkili class'lardan oluşur. Her package'de bulunan class'ların ezberlenmesine gerek yoktur. Sadece her package'de ne tür class'ların bulunduğu öğrenilmesi yeterlidir. Örneğin, arayüzle ilgili class'lar 'java.awt'dedir. Görsel bir bileşen olarak Button bu package'tedir. Yine görsel bir bileşen olan text alanı'nın (TextArea'nın) da bu package'de olması gayet doğaldır.

Naming Convention (İsimlendirme Geleneği)

Package isimlerinin rastgele verilmesi, bir sistemde farklı kurumlara ait package'leri bulunması durumunda '**name collision**' (isim çarpışması) adı verilen bir sorun yaratabilmektedir. Aynı package ismini başka bir kurum da kullandıysa ne olacak? Örneğin siz package'in adını 'myapplets' koydunuz. Sistemde aynı isimde başka bir kuruma ait bir package daha var tesadüfen. Java önce hanginizinkini bulursa onu yükler. Sizinki önce bulunuyorsa onların package'i doğru çalışmaz, onlarınki önceyse sizinki.

Bu sorunlarla karşılaşmamak için package isimlerinde standart bir yöntem uygulanır. Bir kurum, sahip olduğu İnternet domain ismini tersten yazarak package isimlerinin önüne ekler. Örneğin 'godoro.com'a sahip olan kurum veya kişiler, chat appletlerini 'com.godoro.chat' diye bir package'e koyabilirler. ChatApplet isimli biri class için fully-qualified class name 'com.godoro.chat.ChatApplet' şeklinde olur. Java'yı ortaya çıkaran 'Sun' firması da kendi package'lerini 'com.sun.' şeklinde başlatmaktadır. ÖZel kuruluşlar **com.**, ticari olmaytan kurumlar da **org.** ile başlarlar.

Aynı Package'teki Class'lar

Aynı package'deki class'lar birbirlerinin private olmayan property ve methodlarına, onlar public olmasala bile ulaşırlar. Ayrıca, aynı package'deki class'ların birbirlerini 'import' etmeleri gerekmez.

Derleme Birimi

Class'lar '**.java**' uzantılı dosyalarda bulunabilirler. Bu dosya aslında bir **compilation unit** (derleme birimi)dir (). Bir dosyada birden fazla class alt alta tanımlanabilir. Ancak en fazla bir tanesi 'public' olabilir. Ve dosyanın (derleme biriminin) adı bu public class'ın adıyla aynı olmalıdır. Bu class derleme biriminin asıl class'ıdır ve adı değiştirildiğinde dosyanın da adı değiştirilmelidir. Tek dosyada bulunan class'lar derlendiğinde her biri için ayrı

.class üretilir. Yani derlendikten sonra class'lar birbirinden ayrılırlar. Bir derleme biriminde sadece bir tane class bulundurmak, zorunlu olmasa da bir gelenek halini almıştır.

Encapsulation

Encapsulation Kavramı

Her class çeşitli üyelerden (property ve method'lardan) oluşur. Hemen hemen bütün class'lar başka bir class tarafından kullanılmak amacıyla üretildiğinden, bazı üyeler sadece dışarıdan kullanılmak üzere hazırlanmıştır. Buna karşı bazı property ve method'lar, diğerlerine yardımcı olmak, sadece onlar tarafından class'ın iç işlerinde kullanılmak için yazılırlar. Belli bir class'ı kullanan class'ın bunları görmesi veya bilmesi gerekmez. Hatta bazı durumlarda property ve method'ların sadece ufak bir kısmı dışarısı tarafından kullanılır. Bir insanın belli bir class'a baktığında üyelerden hangisi kendisinin işine yarar, hangisi sadece başka methodlarda kullanılmak için yapılmıştır ve class'a özel'dir anlaması çok zor olabilir. Bazı durumlarda da güvenlik ve sağlamlık sağlamak amacıyla bir class'ı kullanan class'ın bazı property'leri değiştirmesi ve bazı methodlara erişmesi engellenmek istenebilir. Bazı property ve methodların ait olduğu class'ın dışında erişimini sınırlama özelliğine '**encapsulation**' (kapsülleme) denir. Encapsulation sağlamak için **private**, **public** ve **proteced** sözcükleri kullanılır. Bunlara **access modifier** (erişim değiştirici) denir. Hiç birini kullanmak da bu bağlamda belli bir tercih anlamına gelir.

Default Access

Hiç bir access modifier kullanılazsa, yani bir property ve methodun başında private, protected ve public keyword'lerinden hiç biri kullanılmazsa bu üye '**package private**' olur. Yani ona sadece aynı package'deki diğer class'lar erişebilir, diğer package'lerdekiler erişemez.

Public Access

Sistemdeki bütün class'ların erişebilmesini sağlamak için '**public**' erişim değiştiricisi kullanılır. Herkesin erişmesinde sakınca bulunmayan üyeleri için kullanılır. Genellikle bu değiştirici özellikle başkaları erişsin diye yazılan property ve method'lar için kullanılır. Bir nesnede ne kadar az public değişken varsa dışarıya o kadar sade görünür ve o kadar kolay kullanılır. O nesneyi kullananlar gereksiz bir çok üye arasında dolaşmak zorunda kalmazlar.

Private Access

Bir property veya method'un sadece tanımlandığı class'dan erişilebilmesini, başka bir deyişle, o class dışındaki bütün class'lardan (aynı package'de olsalar bile) erişiminin yasaklanmasını sağlamak için '**private**' keyword'ü kullanılır.

Bir değişkenin dışardan görülebilmesini ancak değiştirilemesini veya değiştirme işleminin class'ın kontrolünde yapılmasını sağlamak için standart bir yöntem izlenir. Bir property'yi read-only (salt okunur) yapmak için, o property private yapılır. Ona erişmek için bir tane public method yazılır. Bu şekilde dışarından property'nin value'si dolaylı olarak öğrenilmiş olur.

```
public class MyDiscreetClass{
  private int value=3;
  public int getValue(){
    return value;
  }
}
```

Bu class'tan yaratılmış bir nesne'yi kullanan örnek yapalım.

```
public class MyDiscreteUserClass{
  public static void main(){
    MyDiscreetClass d=new MyDiscreetClass();
    int v=d.getValue();
    System.out.println("Value : "+v);
  }
}
```

Buradaki value private olduğu için

```
int v=d.value;
```

yazarsak derleyici hata verir. Property orada olduğu halde artık erişilmez hale gelmiştir. Ama değeri öğrenmek için getValue() methodu kullanılabildiği için bir programın işleyişini bozmaz. Biz bu değeri değiştirmek istersek :

```
d.value=5;
```

derleyici yine hata verecektir. Bu şekilde class kendi property'sini korumuş oluyor.

Bir değişkenin hem okunması hem de değiştirilmesi istenip sadece bunun kontrollü bir şekilde yapılmasını istersek yine property'yi private tutarız ve hem erişebilmek için hem de değiştirebilmek için iki ayrı method yazarız. Bir değerin eksi olmamasını sağlayan bir class yapalım :

```
public class MyControlledClass{
  private int value=3;
  public int getValue(){
    return value;
  }
  public void setValue(int v){
    if(v>0){
      value=v;
    }else{
      value=-v;
    }
  }
}
```

Bu değeri değiştirip sonrsa yeni değeri alan bir program yazalım.

```
public class MyControlledUserClass{
  public static void main(){
    MyControlledClass d=new MyControlledClass();
    d.setValue(-7);
    int v=d.getValue();
    System.out.println("Value : "+v);
  }
}
```

Bu program, kendisinin değiştirmeye (kullanılan class açısından bakarsak, 'bozmaya') çalıştığı değerin hala sağlam olduğunu ortaya koymaktadır.

Protected Access

Bazı durumlarda bir property veya method'un dışarıya kapalı olmasını ancak o class'ı extend edenlere açık olması istenebilir. Bu durumda '**protected**' erişim değiştiricisi kullanılır. Bir class diğerini extend ederse, onun bütün public property'lerin erişebilir hale gelir. Zaten

bunlara her class'ın erişebilmektedir. Hiç bir private property extend eden class'a geçmez. Bunlar diğer bütün class'lara da kapalıdır. Ancak protected property ve methodlar dışarıya kapalı oldukları halde, extend edilen class bunlara erişim hakkına sahiptir. Bu keyword, belli bir property veya method'un sadece onu extend eden class'ları ilgilendirmesi durumunda kullanılır.

Erişim değiştirici olarak protected verilmiş property veya method'lara o class'la aynı package içerisindeki bütün class'lar tarafından da erişilebilir.

Erişim Düzeyleri Sıralaması

Encapsulation düzeyleri için

```
public > protected > 'default' > private
```

şeklinde bir sıralama yapılır. Büyük olan düzey küçük olanların haklarını da kapsar. Bir property'nin public olup private özelliği içermemesi, yani tanımlandığı class'tan erişilemez ama onun dışındaki bütün class'lardan erişilmesi gibi bir durum söz konusu olamaz.

Inheritance (Kalıtım)

Inheritance Kavramı

Bir class'ın başka bir class'daki property ve method'lara sahip olmasına **inheritance** denir.. Belli bir class, daha genel bir kavramı ifade eden class'ın üyelerine sahip olarak, onları tekrar tanımlamak zorunda kalmaz. Bir class'ın diğerindeki özellikleri miras olarak alması için kullanılan keyword '**extends**' dir. Anlam olarak "Bu class şu class'ı 'genişletir' yani ondaki property ve methodları alır ve yenilerini ekler" demektir.

Inheritance, bazı durumlarda başka birinin yazdığı class'a bazı eklemeler yaparak belli bir işlem için kullanılır hale getirmek için kullanılabilir. Böylelikle sadece o class'ta olmayan property ve methodları eklemek suretiyle çok kısa sürede gelişmiş bir class sahibi olabilir. Bazen de ortak bir çok özellik içeren iki nesnede de aynen bulunan property ve methodları tanımlamaktan kurtulmak için kullanılabilir.

'Extends' Kullanımı

Diyelim ki bize bir küp ve küre'yle hacim, yüzey alanı, kütle ve ağırlık gibi hesaplamalar yapmamız gerekiyor. Bir çok özellik bu iki nesnede ortaktır. İkisinde de bir özkütle vardır. Hacim hesaplama teknikleri değişebilir ama kütle hesap teknikleri değişmez. Her iki cisim için de kütle, özkütle ile hacmin çarpımıdır. Bu iki cisim için SimpleSphere ve SimpleCube gibi iki class yapacaksak, ortak özellikleri içeren SimpleBody diye bir class yapmak akıllıca olur. Şimdilik sadece SimpleSphere'e ihtiyacımız varsa dahi, sonradan SimpleCube veya başka bir cismin gerekebileceğini düşünerek bir base class yapmak da faydalı olabilir.

```
public class SimpleBody{
  private double density=1.0;
  public SimpleBody(){
  }
  public SimpleBody(double d){
    setDensity(d);
  }
  public void setDensity(double d){
    density=d;
  }
  public double getDensity(){
```

```
        return density;
      }
    }
```

Görüldüğü gibi, cismin yoğunluğunu belirten bir property ve bunula ilgili iki method var. Bu class'ı 'extend' eden SimpleSphere adlı bir class yapalım.

```
public class SimpleSphere extends Body{
  public double radius=1.0;
  public SimpleSphere(double d,double r){
    super(d);
    radius=r;
  }
  public double getVolume(){
    return (3.14 * radius * radius * radius )/3;
  }
}
```

Yazdığımız SimpleSphere class'ın radius property'si dışında, density property'si de vardır. Şöyle bir kod parçası geçerlidir :

```
SimpleSphere s=new SimpleSphere();

s.setRadius(3.6);

s.setDensity(1.2);
```

Extend eden class'a **subclass**, extend edilen class'a da **super class** (veya base class) denir.

Super Class'ın Constructor'unun Kullanımı

Bir class'ı extend eden class'ın constructor'u extend ettiği class'ın constructor'unu '**super()**' şeklinde çağırabilir. Bu şekilde super class'a veri aktarılmış olur. SimpleSphere class'ına bir constructor ekleyelim :

```
public SimpleSphere(double d,double r){

        super(d);

        setRadius(r);

}
```

Bu şekildeki constructor'u kullanmak istersek

```
SimpleSphere sphere=new SimpleSphere(3.6,1.2);
```

biçiminde bir kod yazmak yeterli olacaktır.

Bir constructor'ın diğerini çağırması durumunda super ifadesi kesinlikle ilk satırda belirtilmelidir. Herhangi bir kod parçası yazıldıktan sonra super class'ın bir constructor'u çağrılamaz.

Super class'ın constructor'u açık olarak çağrılmadıkça, parametresiz olan super() constructor'un çağrıldığı varsayılır. Super constructor'un böyle bir constructor'u yoksa mutlaka parametreli bir constructor çağrılmalıdır. Aksi takdirde derleyici hata verir.

Super Class'larda Protected Property'ler

Base class'da 'private' olan property'lere subclass'tan erişilemez. Erişilmesi için 'public' yapılması durumunda da ilgili üye herkese açık hale gelecektir. Sadece extend eden class'ın

erişimini sağlamak için 'protected' erişim değiştiricisi kullanılır. SimpleBody class'ında 'density' property'si protected olmalıdır.

Overriding (Ezme)

Bir subclass super class'ının bir methodunun davranışını yeniden yazabilir, yani super'dekini ezebilir. Buna **overriding** denir. Bunun için aynı üst class'ın method'unun signature'uyla aynı olmalıdır. Bu işlem bazen methodun tamamen değiştirme, bazen de sadece bir ekleme yapma şeklinde yapılır. SimpleSphere class'ında, SimpleBody class'ında density property'sini belirleyen setDensity() method'u, negatif değer kontrolü yapacak şekilde değiştirmek için

```
public void setDensity(double d){
        if(d>=0){
                density=d;
        }else{
                density=0.0;
        }

}
```

biçimde bir kod yeterlidir. SimpleSphere class'ında setDensity() methodu çağrıldığında bu yeni method geçerli olacaktır. Elbette bu değişiklik SimpleBody class'ı ve bu class'ı extend eder diğer class'ları bağlamaz. Onlar hala üsteki methodu çağırırlar.

Override eden method, override edilen methodu çağırabilir. Ancak ikisinin de adı da parametreleri de aynı olduğu için super anahtar sözcüğü kullanılır.

```
public void setDensity(double d){
        if(d>=0){
                super.setDensity(d);
        }else{
                super.setDensity(0.0);
        }
}
```

Super class'da olan bir method'un ismiyle aynı farklı parametreler alan bir method yazlıabilir. Ama bu durumda overriding değil overloading yapılmış olur. Yani var olan bir method'u değiştirmek yerine yeni bir method eklenmiş olur. İlk method çağrıldığında yine eskisi gibi çalışır.

Multiple Inheritance (Çoklu Kalıtım)

Java'da '**multiple inheritance**', yani bir class'ın iki veya daha çok class'ı extend etmesi, onların property ve method'larını miras alması mümkün değildir. C++ gibi diğer nesneye yönelik dillerde bu mümkündür ama Java'da kafa karışıklığına, çalışma zorluğuna, birbirine girmiş yapılar yaratmaya neden olduğu için izin verilmemiştir. Java'daki **single inheritance** (tekli inheritance)da hangi class'ın hangisini extend ettiği bir ağaç yapısı şeklinde net bir şeklide görülmektedir. Aslında bu yazılım tasarımı açısından da gereklidir. Örneğin bir

hayvan hem memeli hem de kuş olmaz. Hem balık hem memeli olmaz. Hatta yarasa uçsa bile, balina yüzse bile onlar birer memelidir. Yarasanın kuşlar gibi uçtuğunu görüp onu hem memeli hem kuş saymak ne kadar yanlışsa, bir class'ı iki ayrı class'ın özelliklerine birden sahip olması gerektiğini düşünmek o derece yanlıştır. Eğer tasarımınız bunu zorunlu kılıyorsa ya yanlış tasarım yapmışsınızdır ya yanlış çözümleme yapmışsınızdır. Çoklu kalıtımın gerekli gibi göründüğü durumlarda başka çözüm yolları devreye girer. Bunları programcı ve sistem tasarımcıları Java'da bilgi ve deneyimlerini arttırdıkça öğrenebilirler. Örneğin iki class'ı extend etmek yerine birini extend etmek, diğerini bir property olarak tanımlamak ve kullanmak mantıklı bir çözüm olabilir..

Interface'ler ve Abstract Class'lar

Interface Kavramı

Java'da **interface**, bir class'ta olması gereken method ve property'leri tanımlayan yapıdır. Kendisi normal bir class değildir, sadece neyin yapılacağını göstermekte, ancak nasıl yapılacağını göstermemektedir. Örnek olarak, bir madde'de ne gibi işlevler ve özellikler olması gerektiğini belirten bir interface yazalım.

```
public interface Matter{
  public double getVolume();
  public double getMass();
}
```

Bu ifade, 'bir maddenin yoğunluğu, hacmi ve kütlesi olur' demenin Java'daki yoludur. Ancak görüldüğü gibi maddenin yoğunluğu nasıl hesaplanır, kütlesi nasıl verilir, hiç bir şekilde belirtmemektedir. Sadece ne olması gerktiğini söylemektedir.

Interface'in Implementation'u

Bir class'ın interface'deki bütün method'ları içerdiğini, gerçekleştirdiğini belirtmesine **implementation** denir ve '**implements**' keyword'üyle kullanılır.

```
public class CubeMatter implements Matter{
  public double density=1.0;
  public double edge=1.0;
  public double getDensity(){
    return density;
  }
  public double getVolume(){
    return edge*edge*edge;
  }
  public double getMass(){
      return density*edge*edge*edge;
  }
}
```

Burada "Küp diye bir nesnemiz var ve o bir maddedir, yani bir maddede olabilecek bütün nitelikler onda da bulunur." demiş olduk ve bunların nasıl hesaplandığını gösterdik. Aynı interface'i implement eden başka bir class'da yapılabilir.

```
public class SphereMatter implements Matter{
  public double density=1.0;
  public double radius=1.0;
  public double getDensity(){
    return density;
  }
  public double getVolume(){
```

```
      return (3.14 * radius * radius * radius )/3;
    }
    public double getMass(){
      return density*(3.14 * radius * radius * radius )/3;
    }
  }
```

Görüldüğü gib bu class'da aynı interface implement ediyor. Ancak hacim ve kütle hesaplaması çok değişik.

Absract Class'lar

Bazı methodlarını implement etmiş, bazılarının imlementation'unun kendisini extend eden class'a bırakmış olan class'a **abstract class** denir. Bu tip class'lar en çok, iki class'ın ortak methodlarından bazılarının implementation'u da aynı olması durumunda kullanılır. Örneğin bir küp için de kütle, hacim ve yoğunluğun çarpımına eşittir, bir küre için de. Ancak yaptığımız iki nesnede de bu ortak özelliği kullanamıyor. Her birisi kütle hesaplamasını kendisi yapılıyor. Matter interface'ini implement eden ne kadar class varsa bu işlem o kadar tekrarlanacak demektir. Bu 'ortak' işlemi bir kere yapıp, hep onun kullanılmasını sağlamak mümkündür.

Ancak çözülmesi gereken bir sorun var : Her class'ın kütlesi aynı şekilde hesaplanıyor bunu hesaplamak için gerekli hacim değerini bilmiyoruz ve her class kendisine göre değişik bir şeklide hesaplıyor. Bu sorun **abstract method** kullanarak çözülebilir.

```
  abstract public class Body{
    public double density=1.0;
    public Body(double d){
      density=d;
    }
    public double getDensity(){
      return density;
    }
    public double getMass(){
      return density*getVolume();
    }
    abstract public double getVolume();
  }
```

Burada abstract bir Body class'ı yarattık. Bu class, hem Cube'ün hem de Sphere'in ortak özelliklerini içermektedir. Zaten density property'si ikisinde de ortaktır. Bu şekilde inherintace yoluyla Cube ve Sphere'a geçebilir. Burada getVolume() method'u abstract bırakılmıştır. Yani implement edilmemiştir. Çünkü volume hesaplamak için her nesne farklı bir method kullanmaktadır. Ancak getMass() implement edilmiştir. Çünkü kütle yoğunlukla hacmin çarpımıdır. Burada object-oriented porgramlama dışındaki bir teknikle asla yapılamayacak bir işlem bulunmaktadır. Bir method (getMass()) henüz yazılmamış bir mehodu (getVolume()'u) kullanarak bir işlem yapabilmektedir. Bu şekilde her cisim için ayrı ayrı hesap yapmaktan kurtulmuş olduk. Şimdi küp ve küre için yeni class'lar yapalım :

```
  public class CubeBody extends Body{
    public double edge=1.0;
    public CubeBody(double d,double e){
      super(d);
      edge=e;
    }
    public double getVolume(){
      return edge*edge*edge;
    }
  }
```

Görüldüğü gibi bu class kütle hesabını yapan getMass() diye bir method'u yazmak zorunda kalmadı. Çünkü ona miras yoluyla sahip olmuştur. Aşağıdaki gibi kod yazılabilir :

```
CubeBody cube=new CubeBody(2.5,10.2);

double mass=cube.getMass();

System.out.println("Mass : "+mass);
```

Aynı şekilde yeni bir küre class'ı yazalım.

```
public class SphereBody extends Body{
  public double radius=1.0;
  public SphereBody(double d,double r){
    super(d);
    radius=r;
  }
  public double getVolume(){
    return (3.14 * radius * radius * radius )/3;
  }
}
```

Tıpkı küp gibi küre de artık kütle hesabını, hiç bir ek kodlama yapmadan yapabilmektedir.

Hem Extend Hem Implement Etmek

Bir class sadece bir class'ı extend edebilir. Ancak aynı anda bir class'ı extend edip bir veya daha fazla interface'i implement edebilir. Çünkü 'extends' keywordünün aksine, 'implements' demek yoluyla bir class'ın aldığı hiç bir şey yoktur. Sadece bazı method'ları implement etmeyi taahüt etmektedir. Bir Cube class'ı hem Body classını extend edebilir hem de Matter inetrface'ni implement edebilir.

```
public class Cube extends Body implements Matter{

    // ...

}
```

Base class'ın bir interface'i imlement etmesi durumda subclass'da etmiş sayılır. Yani

```
public class Body implements Matter{

    // ...

}
```

dersek

```
public class Cube extends Body{

    // ...

}
```

dememiz yeterlidir. Cube class'ı Matter'i implement etmiş sayılır. Yani implementation'u da miras yoluyla almış demektir.

Interface'in Kullanım Özellikleri

Interface'ler bütün methodları absract olan bir absract class gibi düşünülebilirler. Ancak class'lardan ayrılan başka özellikleri vardır. Ayrıca söylensin veya söylenmesin bütün

method ve property'ler public sayılır ve başların proteced veya private gibi anahtar sözcükler alamazlar. Interface'lerde bütün propertyler **public**, **final** ve **static'**tir. O yüzden interface'lerde tanımlanan property'ler bir veya daha fazla class'da kullanılan sabitler için kullanılır.

```
public interface MathConstants{
  double PI=3.14;
}

public class Circle implements MathConstants{
  private double radius=1.0;
  public getCircumference(){
    return 2*PI*radius;
  }
}
```

görüldüğü Circle classında tanımlanmadığı halde PI değişkeni, sadece MathConstants interface'ini implement ettiği için erişilebilir durumdadır. Bu interface'i implement eden Cylinder adlı bir class'da aynı şekilde PI'a erişebilir. Bu anlamda PI bir sabittir. Ama PI property'si sadece implement edilen class'lar tarafından görülmekte olduğundan global değildir. Zaten Java'da **global değişken** ve **sabit** yoktur. Onlara ihtiyaç duyulan yerlerde interface kullanılır. Ancak yine de değişken (dah doğrusu sabit) bir yere (interface) ait olduğundan global olmasından doğacak sakıncalardan kurtulunabilir. Örneğin Math.PI ve GreekLetters.PI yazılabilir, ikisi farklı değerler taşıyabilir.

Polymorphism (Çok Biçimlilik)

Polymorphism (Çok Biçimlilik)

Nesneye yönelik programlamada **polymorphism** , nesnelerin içeride farklı çalışmalarına rağmen dışarıdan aynı biçimde görünmelerine denir. Bu şekilde, bir grup nesneyi kullanan class'lar implementation'la ilgili detayları bilmek zorunda kalmazlar, içerideki değişikliklerden etkilenmeden çalışmaya devam ederler. Aynı class'ı extend eden veya aynı interface implement eden class'lar standart bir şekilde erişilebilme özelliklerine sahip olurlar.

Polymorphism Olmadan Kodlama

Bir miktar kübün toplam kütlesini hesaplamak istediğimizi düşünelim. Küp class'ının ana hatları şöyle olsun :

```
public class Cube{
    // ...
    public double getMass(){
        // ...
    }
}
```

Bu class'tan yaratılmış instance'ların kütlelerini toplayan bir method ve program yazalım.

```
public class CubesMassCalculator{
  public static double calculateTotalMass(Cube[] cubes){
    double total=0.0;
    for(int i=0;i<cubes.length;i++){
      total+=cubes[i].getMass();
```

```
        }
        return total;
      }
      public static void main(String[] args){
        Cube[] cubes=new Cube[3];
        cubes[0]=new Cube(2.2,10.5);
        cubes[1]=new Cube(3.1,6.1);
        cubes[2]=new Cube(5.6,1.2);
        cubes[3]=new Cube(9.3,6.1);
        cubes[4]=new Cube(4.0,3.7);
        double total=calculateTotalMass(cubes);
        System.out.println("Total Mass : "+total);
      }
    }
```

Aynı programı küreler içinde yazabiliriz. Sphere class'ımız da aynı biçimde tanımlanmış olsun :

```
    public class Sphere{
        // ...
        public double getMass(){
                // ...
        }
    }
```

Ancak calculateTotalMass() method'unu şu biçimin de tanımlamalıyız :

```
        public static double calculateTotalMass(Sphere[] spheres){
                // ..
        }
```

Peki ya bir nesne dizimiz değişik cisimlerden oluşuyorsa? Örneğin üç tane kübümüz iki tane küremiz varsa ne olacak? Onun için şöyle bir method daha yazmalıyız.

```
    public static double calculateTotalMass(Cube[]
  cubes,Sphere[] spheres){
            // ..
    }
```

Ya dikdörgenler prizması ve piramit gibi cisimlerimiz de varsa ne yapacağız? Her biri için ayrı toplamlar alan bir sürü method yazmaktan başka çaremiz kalmayacak. İşte polymorphism burada devreye giriyor.

Ploymophism'den Yararlanma

Eğer iki class'da ortak bir method varsa, bu ikisinin de temel alacağı Body şeklinde bir class yapmak gereklidir :

```
class Body{
        // ...
        public double getMass(){
                // ...
```

```
        }
}
class Cube extends Body{
        // ...
}
class Sphere extends Body{
        // ...
}
```

Çeşitli tipte cisimlerden olişan bir dizinin toplam kütlesini hesaplayan bir class yapalım.

```
public class BodiesMassCalculator{
  public static double calculateTotalMass(Body[] bodies){
    double total=0.0;
    for(int i=0;i&amp;amp;bodies.length;i++){
      total+=bodies[i].getMass();
    }
    return total;<noframes></noframes>
  }
  public static void main(String[] args){
    Body[] bodies=new Body[3];
    bodies[0]=new Cube(2.2,10.5);
    bodies[1]=new Sphere(3.1,6.1);
    bodies[2]=new Cube(5.6,1.2);
    bodies[3]=new Cube(9.3,6.1);
    bodies[4]=new Sphere(4.0,3.7);
    double total=calculateTotalMass(bodies);
    System.out.println("Total Mass : "+total);
  }
}
```

Görüldüğü gibi tek method çeşitli cisimlerin kütlelerini toplayabiliyor. Her cismin farklı bir hacim ve kütle hesaplama şekli olduğu halde. Bunun için tek gerekli şart Cube ve Sphere class'larının Body class'ını extend etmeleri ve Body class'ında getMass() şeklinde bir method olmasıdır. Pyramid diye Body'yi extend eden bir class yapıp, dizinin dördüncü elamanı olarak verirseniz calculateTotalMass() methodu yine toplam kütleyi doğru olarak hesaplayacaktır. Çünkü o zaten Sphere'ı da Cube'ü de tanımamaktadır. Sadece Body class'ını tanımaktadır ki, bir cismin kütlesi olması, onun kütlesini toplama dahil etmesi için yeterlidir.

Interface ve Abstract Class'larla Polymorphism

Polymorphism özelliğini kullanmak için, class'ların belli bir concrete base class'a sahip olmaları gerekmez. Base class'ları abstract olabilir. Hatta ortak kullanılan method'un kendisi bizzat abstract olabilir. Bu biçimde interface'lerin de kullanılması mümkündür. Aynı interface'i implement eden her class aynı type'da kabul edilir. Yukarıdaki örnekteki class'lar

```
abstract class Body{
  // ...
  abstract public double getMass();
}
class Cube extends Body{
  // ...
```

```
  public double getMass(){
          // ...
  }
}
class Sphere extends Body{
  // ...
  public double getMass(){
          // ...
  }
}
```

şeklinde de tanımlanmış olsalar aynı şekilde kullanılabilirdi. Body'nin bir class değil de bir interface olarak tasarlanması durumunda da aynı şey geçerlidir.

```
interface class Body{
  public double getMass();
}
class Cube implements Body{
  // ...
  public double getMass(){
          // ...
  }
}
class Sphere implements Body{
  // ...
  public double getMass(){
          // ...
  }
}
```