Java ile Nesne Merkezli ve Fonksiyonel Programlama

1. Bölüm
Nesne-Merkezli Programlamaya Giriş: Sınıflar, Nesneler ve Tekrar Kullanım

Akın Kaldıroğlu

www.javaturk.org

Kasım 2015

Küçük Ama Önemli Bir Konu

- Bu dosya ve beraberindeki tüm, dosya, kod, vb. eğitim malzemelerinin tüm hakları Akın Kaldıroğlu'na aittir.
- Bu eğitim malzemelerini kişisel bilgilenme ve gelişiminiz amacıyla kullanabilirsiniz ve isteyenleri http://www.javaturk.org adresine yönlendirip, bu malzemelerin en güncel hallerini almalarını sağlayabilirsiniz.
- Yukarıda bahsedilen amaç dışında, bu eğitim malzemelerinin, ticari olsun/olmasın herhangi bir şekilde, toplu bir eğitim faaliyetinde kullanılması, bu amaca yönelik olsun/olmasın basılması, dağıtılması, gerçek ya da sanal/Internet ortamlarında yayınlanması yasaktır. Böyle bir ihtiyaç halinde lütfen benimle, akin@javaturk.org adresinden iletişime geçin.
- Bu ve benzeri eğitim malzemelerine katkıda bulunmak ya da düzeltme ve eleştirilerinizi bana iletmek isterseniz çok sevinirim.
- Bol Java'lı günler dilerim.

İçerik

- Bu bölüm, nesne-merkezli programlamanın en temel kavramlarına bir giriştir:
  - Soyutlama (abstraction)
  - Sınıf ve nesne (class and object)
  - Sınıfın bileşenleri: Üye değişkenler ve metotlar (member variables and methods), kurucular (constructors)
  - Nesne başlatma
  - Sarmalama (encapsulation), bilgi saklama (information hiding) ve erişim niteleyiciler (access modifiers)
  - Bileşim (composition) ve kalıtım (inheritance)
  - java.lang.Object sınıfı ve metotları

Soyutlama

Soyutlama

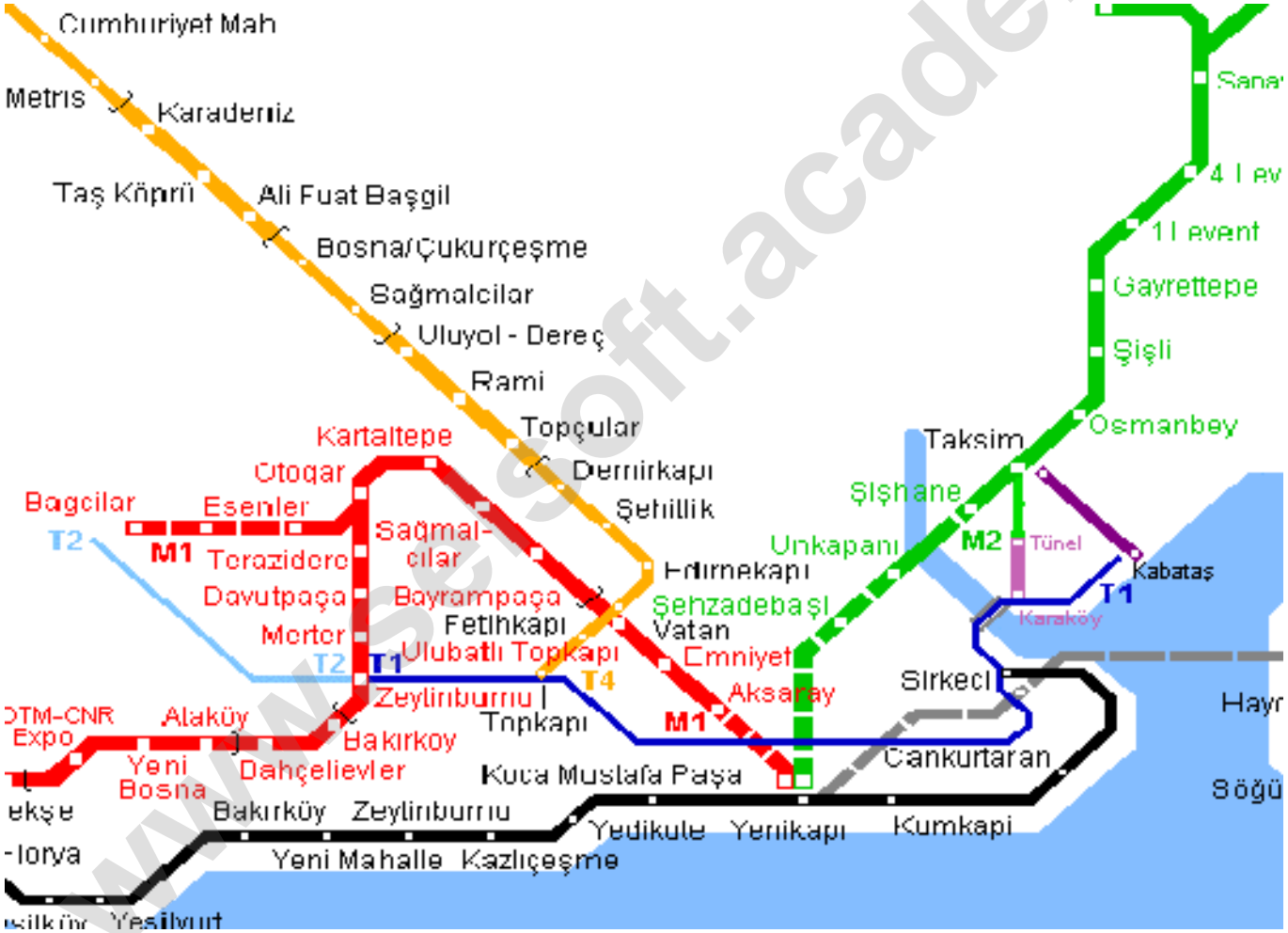
- **Soyutlama**, bir şeyin, sahip olunan bakış açısı itibariyle, en önemli özelliklerini ön plana çıkarırken, önemli olmayan özelliklerini bastırmaktır, görmezden gelmektir:
  - Önemli olan özellikler, genel olarak o şeyi diğer şeylerden ayırt eden unsurlardır ya da ana, asli özelliklerdir,
  - Ayırt edici olmayanlara ise ikincil ya da arizi özellikler denir.

- Zihnimiz karşılaştığı her nesneyi tek tek algılamak yerine, nesneleri, karekteristik özellikleriyle algılar, sonra da ya zihinde var olan kavramsal bir kategoriyle örtüştürür ya da böyle bir kategori yoksa, bu nesneden yola çıkarak yeni bir kategori oluşturur.

Neden Soyutlama?

- Çünkü zihnimiz bu haliyle soyutlama yapmadan bir nesneyi tüm yönleriyle kavrayamaz.
- Dolayısıyla soyutlama, bir indirgemedir, basitleştirmedir ya da genellemedir.
- Soyutlamalar, bir bebeğin dünyayı algılama şeklinde apaçık görülür:
  - Köpek: 4 ayaklı, havlayan ve kuyruğunu sallayan canlı

İstanbul Metro Haritası



"Ne"lik ve "Nasıl"lık

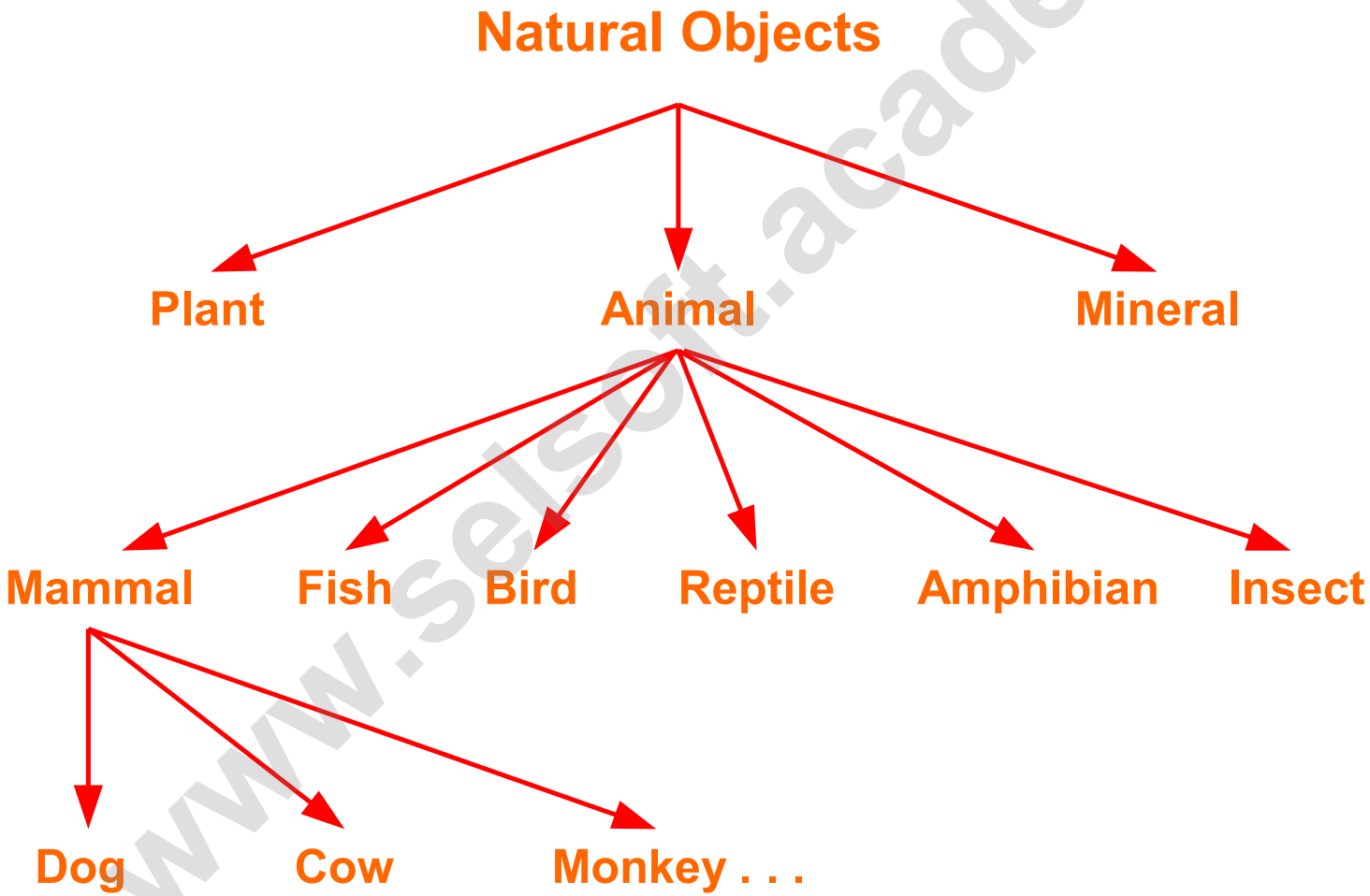
- Soyutlamalar, bir şeyin "ne"liği üzerine yoğunlaşır, "nasıl"lık üzerine eğilmez.

- Yani temel özellikleri, o özelliklerin oluşumundan, nasıl meydana geldiğinden bağımsız olarak ele alır:
  - Araba'yı algılamak için, motor yapısını bilmeye gerek yoktur,
  - Ya da bir insanla ortaklık kurmak için onun DNA dizilerini bilmeye de gerek yoktur.

Sınıflandırma

- Zihin, sonsuz sayıda soyutlama ile uğraşır,
- Ve bu soyutlamalarla elde edilen genellemeler birer kategori oluşturur,
- Soyutlamayı yönlendiren ilgi alanları ya da kriterler çerçevesinde kategori oluşturmaya **sınıflandırma** (**classification**) denir.
- Böylece nesneler, olgular, duygular vs. hepsi belli sınıflara ait hale gelirler.
  - Sınıflar, kavramsal genellemeleri, nesneler ise sınıfların gerçeklikte var olan örneklerini oluşturur.

Sınıflandırma Örneği



Sınıf ve Nesne (Kavramsal)

Sınıf ve Nesne

- Konuşmalarımızda "Köpek dediğin sadık olmalı." ya da "Köpekler çok sadık hayvanlardır." diyorsak, komşunun köpeğinden ya da sokakta az önce karşılaştığımız köpekten değil de kavramsal olarak *köpek sınıfından* bahsediyoruz demektir ve söylediklerimiz, var olmuş ve olacak bütün köpekler için geçerlidir.
- "Komşunun köpeği çok sadık." dediğimizde ise köpek sınıfının bir örneği ya da nesnesi olan, soyut olmayıp somut olan bir canlıdan bahsediyoruz demektir.
- İlk durumda "köpek" bir sınıfı, "komşunun köpeği”nde ise köpek bir nesneyi (object ya da instance) temsil eder.

Nesne (Object)

- **Nesne**, insan zihninin algıladığı herhangi bir kavramsal ya da fiziksel şeydir:
  - Öğrenciler, derslere devam ediyorlar.
  - Öğretmen, sınıfta öğrencileri dinliyor.
  - Dersler yarın başlıyor.

- Nesnelerin özellikleri vardır ve bu özellikler, nesnelerin **durumlarını** *(state)* ve **davranışlarını** *(behavior)* ifade eder:
  - Sarı boyalı sınıfta öğrenci şiir okuyor.
  - Kırmızı top suya yuvarlandı.

Sınıf I

- **Sınıf**, benzer nesnelerin kategorisidir.
- Sınıf, nesneler için bir kalıptır, şablondur, yani kendisinden üretilecek olan nesnelerin sahip olacağı özellikler ile davranışları tarif eder.
- Sınıf, nesnelerinin özellikleri değişik tiplerde değişkenlerle ya da bir başka deyişle veri yapılarıyla (data structures), davranışlarını ise metotlarla (method) (fonksiyon (function)/prosedür (procedure)) ifade eder.
- Nesnenin özelliklerinin bütününe **durum** (**state**), metotların bütününe de **arayüz** (**interface**) denir.

Sınıf II

- Böylece, aynı sınıftan üretilen nesneler aynı tipte olurlar, yani:
  - Aynı özelliklere sahiptir ama özelliklerin değerleri değişebilir,
  - Aynı davranışlara sahiptir,
    - Davranışlar genelde duruma bağlı olduğundan, farklı durumdaki nesnelerin davranışları da farklı olur.

Yazılımın Nesnesi

- Yazılımın nesnesi ise gerçek dünyadaki, kavramsal ya da fiziksel, bir nesneyi temsil etmek üzere, onun özelliklerini ve davranışlarını ifade eden yapıdır:
  - Yazılımın nesnesi, temsil ettiği gerçek dünyadaki nesnenin durumunu, sınıfında tanımlanan değişkenlerle, davranışlarını da metotlarla yerine getirir.
  - Yani nesne, sınıfının ifade ettiği soyut yapının hayat bulmuş, gerçekleşmiş halidir.

Durum

- Nesnelerin durumu ile daha çok durağan (static), görülen ve hissedilen, özellikleri kastedilir ve programlama dillerinde farklı tiplerde bir grup değişken ile ifade edilir.
- Nesnenin durumunu oluşturan her bir ayrık bilgiye ise **özellik** (**attribute**) denir:
  - Öğrenci
    - No, isim, soy isim, doğum tarihi, cinsiyet, adres, bölüm, aldığı dersler, vs.
  - Ders
    - No, isim, bölüm, veren kişi, kredi sayısı, vs.

Davranış

- Nesneler davranırlar, hareket ederler, belli işleri yerine getirirler.
- Yazılım nesnelerinin davranışlarına, yerine getirdiği **sorumluluk (responsibility)**, verdiği **hizmet (service)** ya da aldığı **mesaj** (**message**) olarak bakmak, işimizi kolaylaştırır:
  - Öğrenci
    - Kayıt olur
    - Ders alır
    - Sınava girer, vs.
  - Ders
    - Öğrencinin kaydolmasına/bırakmasına izin verir
    - Ön şart dersleri hakkında bilgi verir, vs.

Mesajlaşma

- Bir nesne şu 4 şekilden biri ile bir sorumluluk yerine getirir:
  - Nesne, kendi durumu hakkında bilgi verir,
  - Nesne, kendi durumunu değiştirir,
  - Nesne, bir işi ya da faaliyeti yerine getirir ve zorunlu olmamakla birlikte bu faaliyet sonucunda bize birşeyler geri verir ,
  - Nesne, bizim ondan istediğimizi, bir başka nesneye havale eder:
    - Havale edilen nesne bu sefer yukarıdaki 3 durumdan birisiyle istenileni yerine getirir.
- Nesnelerin birbirlerinden bir sorumluluk/hizmet yerine getirmesini istemeye **mesajlaşma** (**messaging**) denir.

Nesne-Merkezli Yazılım

- Nesne-merkezli yazılım sistemi ise, birbirleriyle mesajlaşan ve bu şekilde iş süreçlerini yerine getiren bir grup nesneden başka birşey değildir.

- Nesneler, yazılım sisteminin yerine getireceği sorumlulukları paylaşırlar öyle ki her bir nesne, temsil ettiği kavramla ilgili sorumlulukları yerine getirir.

Sınıf ve Nesne (Mekanizma)

Sınıf

- Sınıf, kendisinden üretilecek nesnelerin kalıbıdır - şablonudur.
  - Aynı sınıftan üretilen nesnelerin tipi, aynıdır.
- Sınıf, nesnelerinin özelliklerini (attributes) ve davranışlarını (behavior) tanımlar.
  - Nesnelerin özellikleri, değişkenlerle (variables),
  - Nesnelerin davranışları ise metotlarla (methods) tanımlanır.
- Nesnenin özelliklerinin bütününe **durum** (**state**), metotların bütününe de **arayüz** (**interface**) denir.

Java'da Sınıf Tanımlama I (Tekrar)

- Java'da sınıf tanımlamak için **class** anahtar kelimesi kullanılır:

```
<niteleyici>* class <SınıfİsmI>{
        <özellik>*
        <kurucu>*
        <metot>*
}
```

- Sınıfın tanımı, Java'da en geniş blok olan sınıf blokuyla yapılır.
- Sınıfın, sıfır ya da daha fazla niteleyicisi (modifier) olabilir.
- Sınıfın, geçerli ve anlamlı bir ismi olmalıdır.

Java'da Sınıf Tanımlama II (Tekrar)

- Kurucu ya da yapılandırıcı (constructor), nesne yaratırken çağrılan özel bir metotdur.

- Zorunlu olmamakla birlikte sınıfın öğeleri, fiziksel olarak sınıf içinde, özellikler, kurucular ve metotlar olarak sıralanır.
  - Özellikler, farklı tiplerde olan nesne değişkenleridir (instance variables).
  - Metotlar ise nesnelerin sorumluluklarını yerine getiren nesne fonksiyonlarıdır (instance methods).

Araba

- Bir "Araba" soyutlaması yapın.
  - Soyutlamada bulunması gereken davranışlar nelerdir?
    - Gitmek
    - Durmak
    - Hızlanmak
    - vs.
  - Soyutlamada bulunması gereken durum bilgileri nelerdir?
    - Marka
    - Model
    - Yıl
    - Hız

```
public class Car {
     String make;
     String model;
     String year;
     int speed;
     int distance;

      public Car(String make, String model, String year, int speed, int distance) {
          this.make = make;
          this.model = model;
          this.year = year;
          this.speed = speed;
          this.distance = distance;
     }

     public void go(int newDistance){
          distance += newDistance;
     }

     public void accelerate(int newSpeed){
          speed = newSpeed;
     }

     public void stop(){
          speed = 0;
     }

     public String getInfo(){
          return "Car Info: " + year + " " + make +...;
     }
}
```

Car.java ve Test1.java

Değişkenlerin Rolleri (Tekrar)

- Java'da değişkenler, basit olsun referans olsun, fonksiyonellik ya da rol açısından üçe ayrılırlar:
  - **Nesne değişkenleri** (**instance** (**object**) **variables**): Nesnenin durumunu oluşturan değişkenlerdir.
  - **Sınıf değişkenleri** (**class variables**): Nesnelerin ortak durumunu ifade eden değişkenlerdir. Değerleri nesneden nesneye değişmez.
  - **Yerel değişkenler** (**local variables**): Geçici değişkenlerdir.
- İlk ikisine **üye değişkenleri** (**member variables**), veri üyeleri (data members) ya da alanlar (fields) denir ve sınıf blokunda tanımlanır.
- Bu bölümde önce nesne ve sınıf değişkenlerini ele alacağız.

Nesne Değişkenleri

- Nesne değişkenleri (instance variables ya da fields), fonksiyonel olarak, nesnenin özelliklerini ifade ederler.
  - Nesne değişkenleri, yapısal olarak referans değişkeni olabildiği gibi basit değişken de olabilir.
  - Nesne değişkenleri, sınıfın içinde ama metot ya da başlatma bloğu (initializer block) gibi herhangi bir alt blok dışında, herhangi bir yerde tanıtılmalıdır,
    - Genelde sınıfın en başında tanıtılırlar.
  - Nesne değişkenleri ilk değerlerini tanıtılırken alabildikleri gibi daha sonra bir metot içinde, genelde kurucu metotta da alabilirler.
  - Tanımlanacak nesne değişkeni sayısında bir kısıtlama yoktur.

Nesne Yaratmak I

- Nesne yaratmak dört adımda gerçekleşir:
  - **Tanıtım (Declaration)**: Önce yaratılacak nesneyi gösterecek referans değişkeni tanıtılır.
  - **Yaratma (Instantiation)**: İkinci adımda **new** anahtar kelimesi kullanılarak nesne yaratılır. **new** operatörü, nesne yaratıldığında, onu gösteren bir referansı geriye döndürür.
  - **Başlangıç durumuna getirme (Initialization)**: Kurucu çağrısı yapılarak, nesne, başlangıç durumuna getirilir.
  - **Atama (Assignment)**: Başlangıç durumuna getirilen nesnenin referansı, kendi tipinden bir referans değişkenine atanır.

```
Car car = new Car();
```

Özelliklere Erişim

- Yaratılan nesnenin özelliklerine ve davranışlarına nesnenin referansı yoluyla erişilir.
- Erişim "." notasyonuyla olur:

```
reference.attribute
reference.method()
```

```
car.speed = 60;
System.out.println(car.speed);
car.go(100)
```

Nesne ve Referans

- Nesne ve referans, farklı kavramlardır.
  - İkisinin de tipi vardır.
  - Nesne, sınıftan türetilen ve bir duruma ve bir grup davranışa sahip olup, heap isimli bellek alanında yaşayan yapıdır.
  - Referans ise, nesneye ulaşmamızı sağlayan ve bellekte bulunan bir değişkendir.
- Bir referans, zamanın farklı anlarında, kendi tipinden farklı nesneleri gösterebilir.
  - Fakat bir anda sadece tek bir nesneyi gösterir.
- Bir nesneye birden fazla referans olabilir.
- Bazen referans olmadan nesne, bazen de nesne olmadan referans vardır.

Car.java ve Test2.java

Metot

- Java'da bir metotun beş ana parçası vardır:
  - **İsim**: Anlamlı ve genelde emir kipinde bir isim.
  - **Parametreler** (**parameters**): Dışarıdan geçilen parametreler.
  - **Dönüş değeri tipi** (**return type**): Çağrıldığı ortama döndüreceği değerin tipi.
  - **Kod** (**body**, **implementation**): İfadelerden oluşan çalışan kısım.
  - **Niteleyiciler** (**modifiers**): Farklı amaçlar için değişik niteleyici anahtar kelimeleri kullanılabilir.

```
<niteleyici>* <dönüş tipi> <isim>(<Parametre>*){
        <kod>*
}
```

Gerçek ve Formal Parametreler

- Gerçek parametre (actual parameter), metot çağrıldığında ona geçilen parametrelere denir.
  - Gerçek parametrelere daha sıklıkla *argüman* da denir.
- Formal parametre (formal parameter), metot tanımında olan parametrelere denir.
- Karşılık gelen gerçek parametreler ile formal parametreler tip bakımından uyumlu olmalıdır.
  - Otomatik yükseltmeler çevirme (cast) operatörüne ihtiyaç duymadan yapılır.
  - Dönüştürülebilen gerçek parametreler için çevirme (cast) operatörü kullanılmalıdır.

Calculator.java

Overloading

- Bir isim, bir sınıftaki birden fazla metotta kullanılabilir.
  - Bu duruma **çoklu kullanım** (**overloading**) denir.
- Overload edilen metotların parametre listesi, sayı ve/veya tip bakımından farklı olmalıdır.
  - Bir sınıfta imzası aynı olan iki tane metot olamaz.
- Overloading, genelde, aynı işi farklı parametrelerle yapan metotlar için kullanılır.
  - System.out.println() metotları

CalculatorOverloaded.java

Kurucu I

- Nesne oluşturulurken çağrılan özel metota **kurucu**/**yapılandırıcı** (**constructor**) denir.
- Kurucu metotlar, bazı açılardan özel metotlardır ve sadece nesne oluşturuken çağrılırlar.
- Kurucular, **new** anahtar kelimesiyle kullanılırlar.
  - Java'da kurucu çağrısı yapmadan oluşturulabilen sadece 2 tip vardır: String ve dizi (array)
  - Diğer her türlü nesne, ancak ve ancak kurucu ile oluşturulur.
- Kurucu metotları ile nesnenin durumunun ilk halini alması sağlanır.
- Bu amaçla kurucu metotlar parametre tanımlayabilir ve bu durumda onlara değer geçilir.

Kurucu II

- Kurucunun ismi, içinde tanımlandığı sınıfın ismiyle aynı olmalıdır.
- Kurucunun dönüş tipi, dolayısıyla da dönüş değeri yoktur.

- Kurucular overload edilebilirler.
  - Bu durum, farklı nesne oluşturma şekillerine karşılık gelir.

- Hiçbir argüman almayan kurucuya **varsayılan kurucu** (**default constructor** ya da **no-arg constructor**) denir.

- Argüman alan kuruculara **akıllı kurucu** (**smart constructor**) denir.

Car.java ve Test3.java

Uygulama

- Daha önce oluşturduğunuz Circle (ya da Daire) isimli sınıfa iki tane kurucu koyun:
  - Varsayılan kurucu yarı çapı 10 olan bir Circle nesnesi oluşturur.
  - Argüman alan kurucu ise geçilen değeri, yarıçapa atar.

- Sonra CircleTest (ya da DaireTest) sınıfında, bu iki kurucu ile oluşturulan nesnelerinizin alan ve çevrelerini hesaplayıp ekrana basın.

Kurucuların Birbirlerini Çağırmaları

- Kurucular, bazen yapacakları işi, diğer kurucuların yardımıyla yapabilirler.
- Bu durumda kurucuların birbirlerini çağırmaları gerekir.
- Bu ise "**this**" anahtar kelimesi ile yapılır.

```
public TreeWithThis(String newType, float newHeight) {
    type = newType;
    height = newHeight;
}
public TreeWithThis(String newType) {
    type = newType;
    height = 1.0f;
}
public TreeWithThis(float newHeight) {
    type = "Pine";
    height = newHeight;
}
```

TreeWithThis.java

- **this** ile, aynı metotlarda olduğu gibi, imzası uyan kurucu metot çağrılır.

- *this* çağrısı, bir kurucuda ilk çalışan satır olmalıdır. Neden?

```
public TreeWithThis(String newType, float newHeight) {
    type = newType;
    height = newHeight;
}
public TreeWithThis(String newType) {
    this(newType, 1.0f);
}
public TreeWithThis(float newHeight) {
    this("Pine", newHeight);
}
```

Uygulama

- Daha önce oluşturduğunuz Circle (ya da Daire) isimli sınıftaki varsayılan kurucuya yapacağıız değişiklikle, onun this(...) çağrısı ile, argüman alan kurucuyu çağırarak nesne oluşturmasını sağlayın.

this Anahtar Kelimesi

- this, anahtar kelime olarak genel olarak şu 2 yerde kullanılır:
- Kurucu ya da bir nesne metotunda, aynı isimde bir yerel değişken olduğunda, nesne değişkenine ulaşmak için.
  - Bu durum, genelde aynı isim, hem nesne hem de yerel değişkende kullanıldığında söz konusu olur.
  - Böyle bir durum yoksa this kullanmaya da gerek yoktur.

```
public class Tree{
    String type;
    float height;
    public Tree(String type, float height){
        this.type = type;
        this.height = height;
    }
}
```

this Anahtar Kelimesi II

- Herhangi bir sebeple, üzerinde metotun üzerinde çağrıldığı nesneye ulaşmak için.

```
public class Tree{
    String type;
    float height;

    public Tree grow(){
        height++;
        return this;
    }
}
```

Uygulama

- Daha önce oluşturduğunuz Circle (ya da Daire) isimli sınıftaki kuruculara geçilen parametreleri, nesne değişkenleriyle aynı isimde yapın ve kurucu içindeki atamlarda nesne değişkenlerine ulaşmak için "this" referansını kullanın.
- Benzer şeyi set metotları için de yapın.

Parametre Geçme

- Java'da parametreler, değerleri ile geçilirler (pass-by-value) .
  - Bir metoda bir basit değişken geçilirken, parametrenin değeri, gerçek değişkenin değeri olarak belirlenir,
  - Benzer şekilde geçilen bir nesne ise, bu durumda o nesnenin referansının değeri, parametre değeri olarak belirlenir. (Aslında "nesne geçme" ifadesi doğru değildir, çünkü hiç bir zaman nesneye doğrudan ulaşamayız, sadece referansına ulaşabiliriz. Dolayısıyla geçilen nesne değil, referansıdır.) Referansın değeri ise zaten nesnenin adresidir.
- Dolayısıyla, her halükarda geçilen şey, gerçek argümanın değeridir.

ObjectPassing.java

static

static Anahtar Kelimesi

- Her nesne bellekte, kendi durumunu ifade eden değişken kümesine sahiptir ve bu kümede bulunan değişkenlerin değerleri, diğer nesnelerdekilerden bağımsız olarak değiştirilebilir.
  - Üye veriler (data members)
- Bazen nesnelerin öyle özellikleri olur ki değeri nesneden nesneye değişmez, bütün nesneler için aynıdır.
  - Bu durumda o bilgiyi, her nesnede ayrı ayrı saklanacak şekilde nesne değişkeni olarak tanımlamak uygun değildir.
- Bu şekilde, aldığı değeri nesneye bağlı olmayan değişkenler "**static**" anahtar kelimesiyle nitelendirilirler.

static Değişkenler

- Statik değişkenler, nesneden bağımsız olduklarından nesnelerin değil, nesnelerin sınıflarının parçasıdırlar.
- Bu yüzden, **static** olarak nitelendirilen değişkenlere **sınıf değişkenleri** (**class variables**) de denir.
- Statik değişkenlerin sadece bir kopyası vardır, o da sınıftadır.

- Sınıf değişkenlerine hem sınıf hem de nesneler üzerinden erişilebilinir.
  - Uygun olan sınıf üzerinden erişmektir; çünkü nesne üzerinden erişildiğinde, nesne değişkeni izlenimi vermektedir.

static Metotlar

- Sınıflar, statik değişkenler gibi statik metotlara da sahip olabilirler,
  - Bunlara da statik metotlar denir.

- Statik metotlar da static değişkenler gibi sınıfın bir parçasındırlar,
  - Hem sınıf hem de nesne üzerinde çağrılabilirler.
    - Uygun olan sınıf üzerinden erişmektir.

StaticDemo.java

Ne Zaman **static**? I

- Statik özellikler, sınıfın bir parçası olduklarından, çağrılmaları için bir nesneye ihtiyaç yoktur.

- Dolayısıyla eğer bir bilgi bir sınıftan oluşturulan nesnelerin durumunun bir parçası olup, değeri nesneden nesneye değişmiyorsa, bir başka değişle, değeri tüm nesneler için her zaman aynı olacaksa, bu değişken static yapılır.

- Benzer şekilde bir metot eğer bir sınıfın statik olan değişkenlerini kullanıyor, nesnelerin değişkenlerini kullanmıyor ise o metot da static yapılır.

Ne Zaman **static**? II

- Zaten statik metotlar doğrudan ancak statik değişkenlere ulaşabilir, nesne değişkenlerine, nesnesiz olarak ulaşamaz.

- Çünkü statik metotlar bir nesneye ihtiyaç duymazlar ve bir nesne üzerinde çağrılsalar bile sınıf üzerinde çalışırlar ve sadece sınıf değişkenlerine ulaşırlar.

- Statik metotlar için this de yoktur.

Ne Zaman **static** Değil? I

- Statik kullanımı, tamamen marjinal ve sıra dışı bir durumdur.
  - Aslolan daima nesnedir, yani nesne değişkenleri ve metotlarıdır.
- Çünkü nesneler, problemimizi modellemeye yararlar.
- Statik değişken ve metotlar ise bu modelde çok özel durumlarda ortaya çıkarlar ve kullanımları ancak bu özel durumlara has olmalıdır.
- Nesne oluşturmanın gereksiz olduğu durumların çözümü static değildir.
  - Bu durumun çözümü gereksiz nesne oluşturmamaktır.

Ne Zaman **static** Değil? II

- Bir sınıftaki tüm değişkenleri ve dolayısıyla da metotları static yapmanın sebebi olsa olsa o sınıftan nesne oluşturmanın teorik ve pratik olarak anlamlı olmamasıdır.
  - java.lang.Math sınıfında var olan E ve PI alanları ile tüm metotlar statiktir, çünkü Math sınıfının nesnelerinin olması teorik açıdan muhaldir. Pratik açıdan da zaten sınıfı, muhtemel tek nesne olarak görülebilir.
  - Benzer şekilde java.lang.System sınıfı da statik alan ve metotlara sahiptir.
- İş alanını modellemeye bir katkısı olmayan utility sınıflarından, çoğu zaman bu sınıfların nesnelerini oluşturmadan, statik metotlarla, hizmet alırız.

Main Metot

- Java'da pek çok sınıfınız olsa bile en az bir tanesi **main** isimli özel bir metota sahip olmalıdır.
- main metota sahip olan sınıf JVM'e geçilerek çalıştırılabilir.
  - main metot, sistemin çalışmaya başladığı yerdir.
  - Nerede tanımlandığının pek önemi yoktur.
- Dolayısıyla diğer sınıfların nesneleri bu metotta oluşturulur ve üzerindeki metotlar burada çağrılarak sistem çalışmaya başlar.
- main metotun *arayüzü (interface)* aşağıdaki gibidir:

```
public static void main(String[] args)
```

Uygulama

- Bir sınıftan kaç tane nesne oluşturulduğunu nasıl bulursunuz?
- Bir sınıf yapın ve bu sınıfın herhangi bir kurucusunu çağırarak oluşturulan tüm nesneleri sayın.
- Bu sayıyı tutacak değişkenin nesne mi yoksa sınıf değişkeni mi olması gerekir?

final

final Anahtar Kelimesi

- **final** anahtar kelimesi ile daha önce bir basit değişkenin nasıl bir sabite haline getirilebileceğini görmüştük.

  - **final** kullanılarak tanıtılan basit değişkenlerin değeri değiştirilemez.

```
final int i = 5;
i = 8;          // Compile-time error.
```

final Referans

- Nesneler doğrudan final yapılamaz, ancak nesnelerin alanları final yapılabilir.
  - Bu şekilde durumu değişmeyen nesne elde edilir.
- Referansın final olmasının anlamı, basit değişkenlere göre biraz farklıdır.
- final referanslar, gösterdikleri nesneden başka bir nesneyi gösteremezler.

```
final Car c = new Car();
c = new Car();     // Compile-time error.
```

final Değişkenler

- final olan değişkenin, basit olsun referans olsun, tanıtıldığı yerde bir ilk değer almasının zorunlu olduğunu belirtmiştik.

- Bu durumun iki istisnası vardır:
  - Tanıtıldıktan sonra, ilk erişimde bir ilk değer vermek
  - Kurucu metotta bir ilk değer atamak
  - Başlatma blokunda bir ilk değer atamak (ileride gelecek)

- Yani final olan değişkenlere kurucuda ya da ilk ulaşımda bir ilk değer atarsanız, tanıtıldığı yerde atama zorunluluğu kalkar.

FinalCar.java

Başlatma (Initialization)

İlk Değer Atama

- Java'da üye değişkenler (member variables) için ilk değer verme aşağıda belirtilen 5 yoldan herhangi birisiyle yapılabilir:
  - Tanımlama cümleleri (definition statements)
  - Kurucular (constrcutors)
  - Metot çağrıları
  - Nesne (ilk değer atama) başlatma blokları (initialization blocks)
  - Statik (ilk değer atama) başlatma blokları (static initialization blocks)

InitializersDemo.java

Başlama Sırası I

- Bir sınıfta, pek çok sınıf ve nesne değişkeni, ilk değer atama blokları ve kurucular olduğunu göz önüne alındığında, bu değişkenlerin oluşturulmaları ve kurucuların çağrılması hangi sırada olur?

- InitializationOrder.java

Başlama Sırası II

- Bir sınıfa ilk defa ulaşıldığında önce o sınıfın *.class* dosyası JVM'e yüklenir.
  - Bir sınıfa ulaşmanın yolları ise şunlardır:
    - Statik bir değişkenine ulaşmak
    - Statik bir metotunu çağırmak
    - Nesnesini oluşturmak için kurucu çağrısı yapmak
- Daha sonra sınıfın statik değişkenleri başlatılır.
  - Sınıfın bir nesnesi oluşturulmasa bile, sınıfa ulaşıldığında statik değişkenleri yüklenir ve ilk değer ataması yapılır.
  - Değişkenlerin başlatılmasına, varsa statik başlatma blokları da dahildir.

Başlama Sırası III

- Sonra eğer sınıfın bir nesnesi oluşturuluyorsa, nesne değişkenleri de başlatılır.
  - Değişkenlerin başlatılmasına, varsa nesne başlatma blokları da dahildir.
- Daha sonra kurucu çağrısı yapılır.
- Her yeni nesne için bu işlemler, yani nesne değişkenlerinin başlatılması ve kurucu çağrısı tekrarlanır.
- Sınıf değişkenleri, ne kadar nesne oluşturulursa oluşturulsun, sadece ve sadece bir defa başlatılır.
- Fakat nesne oluşturulurken, her halukarda, sınıf değişkenleri nesne değişkenlerinden önce başlatılır.

Başlama Sırası IV

- Dolayısıyla başlama sırası
  - Sınıf değişkenleri (statik başlatma blokları dahil)
  - Nesne oluşturuluyorsa
    - Nesne değişkenleri (nesne başlatma blokları dahil)
    - Kurucu çağrısı

  şeklindedir.

- Birden fazla sınıf ve nesne değişkeni olduğu durumda başlatma sırası, fiziksel sırayla belirlenir, önce gelen önce başlatılır.

Uygulama

- Daha önce yaptığınız University uygulaması üzerinde başlama sırasını tahmin edin.
  - Kuruculara gerekeli print ifadelerini yazarak başlama sırasını gözlemleyin.

null I

- **null**, bir anahtar kelimedir, sadece referans değişkenlerine atanabilir ve referansın hiç bir nesneyi göstermediğini ifade eder.
  - **null**'ın tipi yoktur, her referans tipe atanabilir ya da çevrilebilir (cast).
  - Yani referans vardır ama bellekteki hiç bir nesneye işaret etmiyordur.
- Bu şekildeki referanslara "null pointer" denir.
- Bellekteki hiçbir nesneyi göstermeyen refereranslar üzerinden herhangi bir erişim daima "NullPointerException"*a* sebep olur.
  - Çünkü erişilecek değişkenlere ve metotlara sahip bir nesne yoktur.

null II

- Bir referans, iki halde **null** olur:
  - Referans sadece tanımlanıp da herhangi bir nesne ataması yapılmadığında,
  - Referansa özel olarak **null** atandığında.
- Neden bir referansa **null** atanır?
  - Referans ile gösterdiği nesne arasındaki ilişki kesildiğinde ve nihayetinde bir nesneyi gösteren hiç bir referans kalmadığında, o nesne Çöp Toplayıcı (Garbage Collector) tarafından toplanıp işgal ettiği alan da belleğe geri kazandırılır.

```
Car myCar;                    // null reference
myCar.make = "Mercedes"  // NullPointerException
myCar = new Car();       // Not a null reference anymore
myCar = null;                 // null reference
```

Organizasyon

Kod Organizasyonu: Paketler

- Java'da oluşturulan sınıfları (ve arayüzleri), mantıksal açıdan kategorize etmek, bu kategoriler arasında erişim kuralları koymak ve isim uzayı oluşturup çakışmaları önlemek amacıyla paketler vardır.
- Paket "**package**" anahtar kelimesiyle oluşturulur.
- Her sınıf tanımından önce yazılacak bir package cümlesiyle o pakete ait olur.

```
package myPackage;

ya da

package org.javaturk.oofp.ch01.car;
```

package I

- Bir Java kaynak kodunda ancak bir tane paket ifadesi olabilir.
  - Her sınıf, arayüz ya da enumın sadece bir tane paketi olabilir.
  - Bir kaynak dosyasında birden fazla sınıf/arayüz varsa hepsi aynı pakete dahil olur.
- package ifadesi bir Java kaynak kodunda çalışan ilk cümle olmalıdır.
  - package ifadesinden önce boşluk ve yorum satırları olabilir.
- Paket yapısı, .class dosyaları için geçerli ve gereklidir, .java kaynak kodları için değil!
- Fakat genelde kaynak kodları da paketleriyle düzenlenir.

package II

- İç içe paketlerle paket hiyerarşisi oluşturulabilir.
- Böylece bir kökten (root) başlayarak dallanan alt paketlerle, çok sayıda sınıf ve arayüzden oluşan yapılar, anlamlı kategorizasyona ve buna karşılık gelen fiziksel organizasyona sahip olur.
- Paketler, işletim sistemindeki dizinlere (directory) karşılık gelir.
  - Dolayısıyla aynı paketteki yapılar aynı dizinde bulunurlar.

package III

- Paketin bir diğer amacı da bir isim uzayı (namespace) oluşturarak, muhtemel çakışmaları önlemektir.
- Bir projede yazılan bir kaç tane farklı Printer ya da Date sınıfının, birbileriyle ya da satın alınan bileşenlerdeki aynı isimdeki sınıflarla karışmasını önlemenin yolu, her bir Printer ya da Date sınıfının apayrı pakete sahip olmasıdır.
- Bu amaçla, Internet domain adresini tersinden kullanarak paketler oluşturduğunda çakışma olmayacaktır.

```
com.myCompany.myProject.mySubPackage
org.javaturk.oop
org.javaturk.advancedJava
org.javaturk.javaee
org.javaturk.dp
```

Tam İsim

- Bir pakete sahip olan bir sınıfın tam ismi (fully qualified name) artık "**paket.sınıfİsmi**" olarak değişmiştir.
- JVM'e (java) geçilirken artık tam ismiyle geçilmesi gereklidir.
- Daha önce de belirtildiği gibi bu durum .class dosyaları yani derlenmiş sınıflar için geçerlidir, .java dosyalarındaki kaynak kodlar için geçerli değildir.

```
package shipping.domain;          // Company.java
javac Company.java                // Compiling
java shipping.domain.Company      // Running

package shipping.gui;             // MainMenu.java
javac MainMenu.java               // Compiling
java shipping.gui.MainMenu        // Running
```

Paketler Arası Erişim

- Aynı paketlerdeki yapılar birbirlerine doğrudan erişebilirler.
- Farklı paketlerdeki yapıların birbirlerine doğrudan erişebilmeleri, ancak tam isim kullanmakla gerçekleşir:

```
package org.javaturk.oop.ch08.packaging.packageX;

public class ClassX {}
```

```
package org.javaturk.oop.ch08.packaging.packageA;

public class ClassA {
     org.javaturk.oop.ch08.packaging.packageX.ClassX x;

     public ClassA(org.javaturk.oop.ch08.packaging.packageX.ClassX x) {
          this.x = x;
     }
}
```

import I

- Tam isim kullanarak erişimin sıkıntılı olduğu açıktır:
  - Bu durumda farklı paketlerdeki yapılar birbirlerini **import** ederler.
- **import** cümlesi, Java kaynak kodunda varsa **package** cümlesinden sonra gelir.

```
package org.javaturk.oop.ch08.packaging.packageB;

import org.javaturk.oop.ch08.packaging.packageX.ClassX;

public class ClassB {
    ClassX x;
    public ClassB(ClassX x) {
        this.x = x;
    }
}
```

import II

- Bir kaynak kodda birden fazla yapı import edilebilir.
- Bu durumda yapılar tek tek import edilebileceği gibi, "*" ile toptan import edilebilir.
  - "*" ile alt paketler import edilemez.

```
package org.javaturk.oop.ch08.packaging.packageB;

import org.javaturk.oop.ch08.packaging.packageX.ClassX;
import org.javaturk.oop.ch08.packaging.packageX.ClassZ;
// ya da
import org.javaturk.oop.ch08.packaging.packageX.*;

public class ClassB {
    ClassX x;
    ClassZ z;
    public ClassB(ClassX x, ClassZ z) {
        this.x = x;
        this.z = z;
    }
}
```

Paketler ve Dizinler

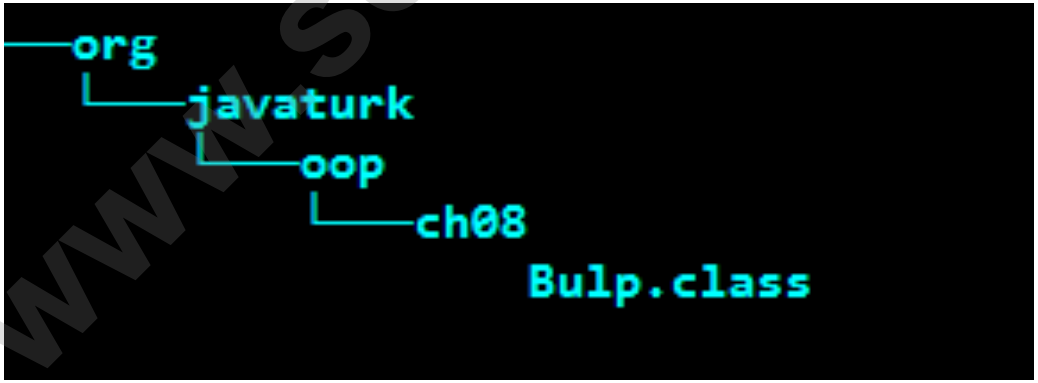
- Paketler, işletim sistemindeki dizinlere karşılık gelir.
- Dolayısıyla bir yapının paketinin org.javaturk.oop.ch02 olması örneğin Windows işletim sistemine bir yerde fiziksel olarak org\javaturk\oop\ch02\ şeklinde bir dizin yapısının bulunduğunu gösterir.
- Belirtilen pakete sahip olan sınıfın .class dosyasının fiziksel olarak, pakete karşılık gelen dizinde olması şarttır.
- Bu şart .java kaynak kodu için geçerli değildir.
- Dolayısıyla paket aslında bir çalışma zamanı yapısıdır ve .class dosyalarını ilgilendirir.
  - .java kaynak kodları tamamen farklı bir dizin yapısında olabilir.

Bir Dizine Derlemek

- Java kaynak kodları derlenirken "**-d**" seçeneği ile bir dizin geçilirse, javac, paket yapısına uygun olarak .class dosyalarını belirtilen dizine koyacaktır.

```
package org.javaturk.oop.ch08;
...
public class Bulp{...}
```

```
C:> javac -d C:\classes Bulp.java
```



Uygulama

- Aynı paket yapısını Eclipse üzerinde kurun ve SelamTest'i çalıştırın.

- Projenin özelliklerine giderek CLASSPATH ayarlarını gözlemleyin.

Java Paketleri

- Java'da pek çok pakete sahiptir:
  - java.lang
  - java.util
  - java.io
- Bu paketlerden **java.lang** her Java kaynak koduna daima otomatik olarak import edilir.

- Java'nın paketleri ve içerikleri, **Java API**'sini oluşturur.

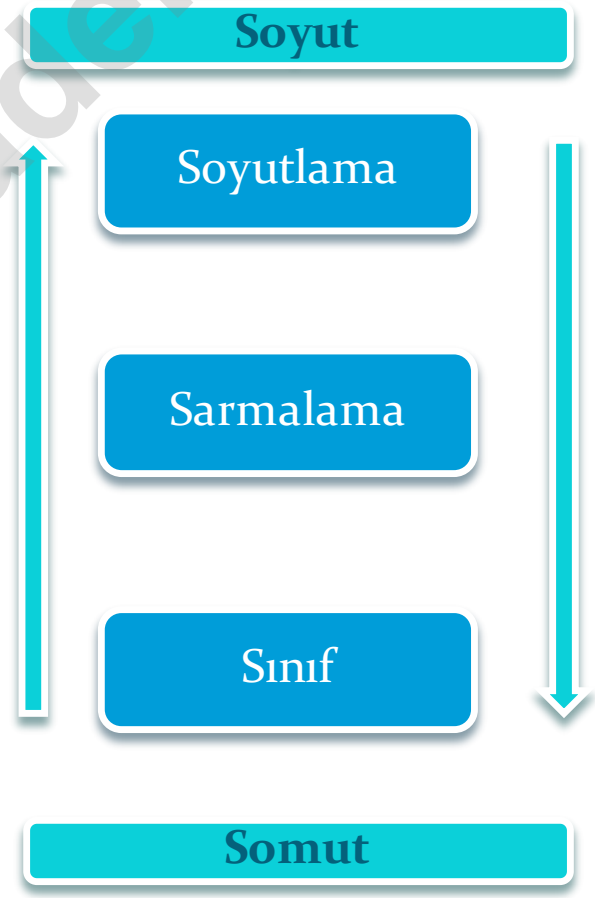
Sarmalama (Encapsulation)

Sarmalama (Encapsulation) I

- Verinin, o veri üzerinde çalışacak metotlarla birlikte bir yapı altında toplanmasına **sarmalama** ya da **encapsulation** denir.
  - Sarmalama, soyutlamayı tasarlanmış halidir.
- Programlama dillerinde eskiden bu yana, soyut veri yapısı (abstract data type) ya da kullanıcı tarafından oluşturulmuş veri tipi (user-defined data type) adları altında var olan bir kavramdır.
- Fakat bu yapılar, sarmalama örneği olmakla birlikte nesne-merkezli dillerin "nesne" kavramından farklı olarak daha teknik bir çıkış noktasına sahiptirler.

Sarmalama (Encapsulation) II

- Nesne-merkezli dillerde en temel sarmalama mekanizması sınıftır (class).
  - Sınıflar, soyutlamaların gerçekleştirildiği yapılardır.
- Nesneler ise soyutlamaların gerçek örnekleri, sınıfların nesneleridir.
- Sarmalama ile bir sınıftan üretilecek olan nesnelerin veri yapıları ile davranışları, mantıksal bir birim haline getirilir.



Bilgi Saklama (Information Hiding) I

- Bu yapıya sarmalama ya da encapsulation denmesinin sebebi, bu teknik ile sarmalanan yapının iç mekanizmasının dışarıdan saklanmasıdır.
- Bu durum, bilgi saklama (information hiding) ile ifade edilir ve bazılarınca sarmalamanın içinde bir teknik olarak görülürken, bazılarınca sarmalamaya ek, tamamlayıcı bir teknik olarak ele alınır.
  - Bazen gerçekleştirme saklama (implementation hiding) ile birlikte anılır.
- Sarmalama, bilgi saklama prensibiyle birlikte, bir sınıfın iç yapısını dışarıdan saklarken, arayüzü üzerinden dışarıya yani istemcilerine hizmet vermesini sağlar.

Bilgi Saklama (Information Hiding) II

- Bağımlılığı düşük ve iç tutarlılığı (lowly-coupled and highly-cohesive) yüksek sınıflar oluşturmak için nesne merkezli dillerde bilgi ve gerçekleştirme saklamayı (information and implementation hiding) sağlayan *erişim niteleyiciler* (*access modifiers*) vardır.

- Bu yapılar sayesinde sınıflar ve sınıfların iç yapıları, dışarıdan doğrudan erişimden saklanabilir.

- Law of Demeter
  http://en.wikipedia.org/wiki/Law_of_Demeter

Erişim Niteleyiciler (Access Modifiers)

- İki tür erişim niteleyici vardır:
  - Sınıf erişim niteleyicileri
  - Üye erişim niteleyicileri

- Erişim niteleyiciler için aşağıdaki anahtar kelimeler mevcuttur:
  - **public**
  - **private**
  - **protected**

Sınıf Erişim Niteleyicileri

- Sınıflar için 2 seviyeli erişim söz konusudur.
- Bu iki seviye public anahtar kelimesinin var oluduğu ve olmadığı 2 durumla belirlenir:
  - **public**: Public sınıflara her yerden erişilebilir.
    - public sınıflar, kendi ismine sahip .java kaynak dosyalarında bulunmalıdırlar.
  - Varsayılan (public kelimesi yok): Bu durumda sınıfa sadece içinde bulunduğu paketten ulaşılabilinir.
    - Bu erişime paket erişimi (package accessibility), varsayılan erişim (default accessibility) ya da arkadaşça erişim (friendly access) denir.

ClassA.java, ClassXX.java ve ClassZZ.java

Üye Erişim Niteleyicileri I

- Üyelere erişim için 4 seviye vardır:
  - **public**
  - **protected**
  - Varsayılan (default), hiç bir niteleyicinin kullanılmadığı durumdur.
  - **private**
- **public** olan üyelere her yerden erişilir.
- **private** olan üyelere, sadece içinde bulunduğu sınıftan erişilir, dışarıdan erişime tamamen kapalıdır.
- Varsayılan halde erişim sadece paket içindeki sınıflara açıktır.
- **protected**, devralan alt sınıflarca erişime açıktır.

Üye Erişim Niteleyicileri II

Niteleyici	Kendisi	Paketi	Çocukları (Farklı Pakette)	Herkes (Farklı Pakette)
public	+	+	+	+
protected	+	+	+	-
- (varsayılan)	+	+	-	-
private	+	-	-	-

ClassP.java ve ClassA.java

public, protected, Varsayılan ve private

- **public**: Genel olarak metotlar public yapılır. Çünkü metotlar nesnenin arayüzüdür.
  - static ve final olan değişkenler de genelde public olurlar.
- **protected**: Başkasının ulaşamayıp, sadece devralan sınıfların ulaşabilecekleri protected yapılır.
  - protected, farklı paketteki devralmayan sınıflara private'dır.
- Varsayılan: Sadece aynı paketteki lerin ulaşabileceği şeyler, varsayılan erişime tabi olur.
- **private**: Bütün üye verilerle sadece iç çalışma için oluşturulmuş metotlar private yapılır.

EncapsulatedElevator.java

Uygulama

- Sadece bir nesnesi olan sınıflara "singleton" denir. Bu durumda bütün bilgi ve hizmet sadece, singleton olan sınıfın var olan tek nesnesi üzerinden verilir. Bunun için de diğer nesnelerin tek olan bir nesneye ulaşmaları gereklidir.

- Bir sınıftan sadece bir nesne oluşturulabilmesi için gerekli yapıyı kurun ve daha fazla oluşturulmasını engelleyin. Sonra da oluşturulan bu nesneye genel bir erişim noktası sağlayın.

- Singleton sınıf yapmak yerine statik metotları olan bir sınıf yapmayı düşünür müsünüz? Tartışın.

public Erişim ve API

- **API**, **Application Programming Interface**'in kısaltmasıdır.
- API, bir sınıfın ya da sistemin (bileşen, çerçeve vs.) *public* olan üyelerine denir.
  - Miras amacıyla protected olan üyeler de API'ye dahildirler.
- API, bir yazılım yapısının arayüzüdür, tek iletişim noktasıdır.
- Sınıflar ya da sistemler, ancak arayüzleri yani API'leri ile ulaşılır ve hizmet alınır.

Arayüz ve Gerçekleştirme

- İyi bir soyutlama, sadece basit bir kavramı ya da fiziksel nesneyi soyutlamalıdır.
  - O şey ile ilgili her türlü gerekli bilgiyi bilmeli ve sorumluluğu yerine getirmeli, o şeyle ilgili olmayan hiçbir bilgiyi bilmemeli ve sorumluluğu yerine getirmemelidir.
- Soyutlamaların karmaşıklaşmaya başladığı her noktada yeni soyutlamalar oluşturulmalıdır.
- Karmaşık bir sistemin hiç bir parçası, bir diğer parçanın iç yapısına bağımlı olmamalıdır.
- Nesneler birbirlerinin arayüzlerine bağlı olmalılar, veri yapıları ile gerçekleştirmelerine bağlı olmamalıdırlar.
- **Program to an interface, not an implementation.**

API ve Java API'si

Java API I

- **API, Application Programming Interface**'in kısaltmasıdır.
- Java API'si, Java'nın sanal makinasının ve standart kütüphanelerinin arayüzüdür:
  - public ve
  - protected üyeleridir.
- Windows için JDK kurulumundaki jre dizinindeki lib dizinindeki **rt.jar** dosyasındaki yapıların arayüzlerini içerir.
  - Sadece JRE kurulmuşsa lib dizininde bulunur.
- Mac için JDK 1.6'da "**classes.zip**" 1.7'de **rt.jar**'dır.
- Java SE Documentation olarak http://www.oracle.com/technetwork/java/javase/downloads/index.html adresinden indirilebilir.

Java API II

- Java API'sinde var olan pek çok sınıf ve benzeri yapılar, tekerleği tekrardan keşfetmeden programlama yapmamızı sağlarlar.

- İyi bir Java'cı, Java API'sini sık ve etkin bir şekilde kullanır.

JDK Kaynak Kodu

- Java SE'nin standart uygulaması olan JDK açık kaynak kodlu olduğu için istenirse ilgili sayfadan o da indirilebilir.

- http://www.oracle.com/technetwork/java/javase/downloads/index.html

Uygulama

- Java SE'nin API'sini Internet'ten indirip açın.
- Java API'sinde var olan paketleri ve içeriklerini inceleyin.
- Java API'sinde aşağıdaki sınıfları inceleyin:
  - java.lang.System
  - java.lang.Math
  - java.lang.String

Uygulama I

- Java API'sinde bulunan bir sınıfı kullanarak, bir metinde geçen tüm kelimeleri bulun, ya da İngilizce olarak:
- ***Find all of the tokens in a string using a class found in Java API***.
- Bunun için Java API'sinden bu iş için ***java.util*** paketinden uygun bir sınıf bulup, API'sini kullanarak uygun kurucu ve metotların neler olduklarını belirleyin.
  a. Daha sonra bu kurucu ve metotları çağırarak verilen bir String nesnesinin kelimelerini tek tek bulup ekrana yazın.
  b. Daha sonra ayıraç olarak 'a' harfini kullanarak String nesnesini parçalayın.
  c. String nesnesini, ayıraç olan 'a' harfini de basacak şekilde parçalayın.

Uygulama II

d. Daha sonra aynı şeyi ***java.lang*** paketindeki ***String*** sınıfı ile yapın.

Tekrar Kullanım (Reusability)

Tekrar Kullanım (Reusability)

- **Tekrar kullanım** (**reusability**), var olan yazılım yapılarından yararlanarak, lego bloklarını kullanır gibi, yeni yazılım sistemleri geliştirmektir.
  - Tekrar kullanım, Yazılım Mühendisliği'nin nirvanasıdır.

- Ama yazılımların, soyut, aşırı karmaşık ve değişime zorunlu doğası, bir yazılım yapısının, kendisi için geliştirildiği sistemden başka bir yerde kullanılabilmesini son derece zorlaştırmaktadır.

- Yine de nesne merkezli diller, en temel seviyede tekrar kullanımı amaçlayan mekanizmalara sahiptirler.

Farklı Seviyelerde Tekrar Kullanım

- Yazılımda, çok farklı şeyler tekrar kullanıma konu olabilir:
  - Metotlar, tekrar kullanımın en basit ve sık uygulandığı yapılardır.
  - Sınıfların tekrar kullanımı daha geniş olmakla birlikte daha zordur.
- Daha karmaşık olan bileşenler (components) ve çerçeveler (frameworks) ile tekrar kullanım çok daha yüksek seviyede elde edilir ama başarılması bir o kadar da zordur.
- Ayrıca iş süreçleri analizi, mimari yaklaşımlar, tasarım, test yapıları vs. hep tekrar kullanıma konudurlar.
  - Tasarım kalıpları, çok tipik tasarım tekrar kullanımına örnektir.

Yeni Bir Sınıf

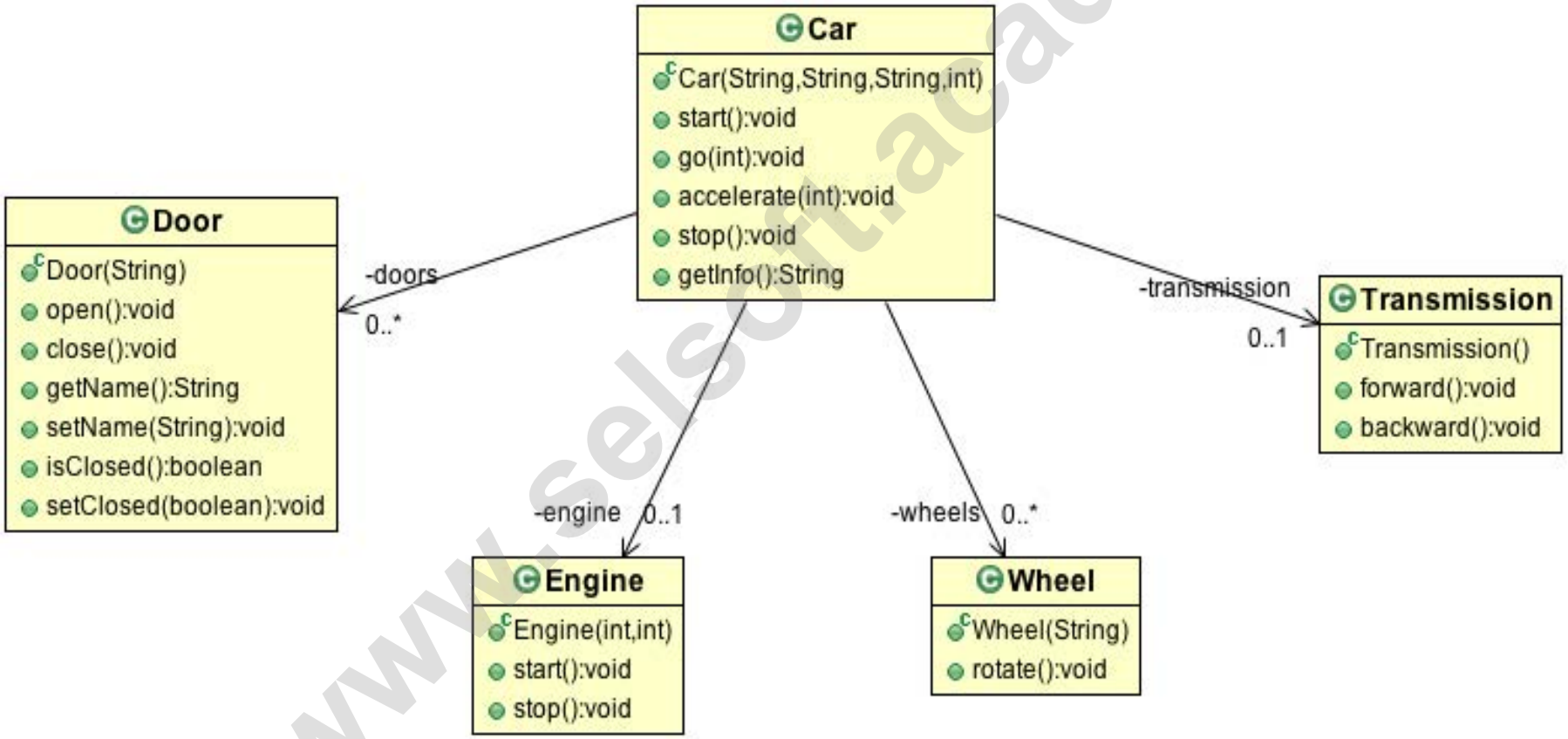
- Yeni bir sınıfa ihtiyaç duyulduğunda alternatifler şunlardır:
  - Pazardan bir tane satın almak,
  - Sıfırdan yazmak,
  - Var olan sınıflardan yararlanarak bileşik (composite) bir sınıf oluşturmak,
  - Var olan bir sınıftan devralarak bir alt sınıf (sub-class) oluşturmak.
- İdeal ve aynı zamanda en az muhtemel olan ilk seçenektir.
- İlk başta kolay gözüküp de uzun vadede en sıkıntılı olan ikinci seçenektir.
- 3. ve 4. seçenekler ise sırasıyla tekrar kullanımın, bileşik nesne oluşturma (composition) ve kalıtım (inheritance) şekilleridir.

Bileşim (Composition)

Bileşik Nesneler

- **Nesne birleştirme** (**object composition**), birden fazla nesneyi bir araya getirilerek daha karmaşık nesneler oluşturmaktır.
  - **Bileşik nesneler** (**composite object**) genel olarak, başka sınıfların nesnelerini, nesne değişkeni olarak kendinde barındıran yapılardır.
- Bu ilişki **sahip olma** (**has-a**) ilişkisi olarak ifade edilir.
  - Unutulmamalıdır ki sahip olma, referanslar üzerinden gerçekleşmektedir.
- Nesne birleştirme ile nesneler arasında bir **ilişki** (**association**) ve aynı zamanda bir **bağımlılık** (**coupling**) oluşturulur.
  - Bu durum, nesneler arasındaki en yaygın ilişki kalıbıdır.

Car as A Composite Object



```
public class Car {
   private String make;
   private String model;
   private String year;
   private int distance;
   private int speed;
   public Engine engine;
   private Transmission tx;
   private Door[] doors;
   private Wheel[] wheels;

   public Car(..., int doorCount,...){
    engine = new Engine(...);
    tx = new Transmission(...);
    doors = new Door[doorCount];
    wheels = new Wheel[4];
    ...
   }

   public void start(){
     engine.start();
   }
 ...
}
```

```
public class Engine{
   private String make;
   private int cc;
   private int horsePower;
   private int rpm;
   ...
}
```

```
public class Wheel{
   private int size;
   ...
}
```

```
public class Door {
   private boolean closed;
   ...
}
```

```
public class Transmission {
   private boolean manual;
   ...
}
```

CarTest.java

Bileşik Nesne ve Bileşenler

- Bileşik nesneler, bileşenlerinden hizmet alırlar:
  - Bileşik nesne, kendisinden isteneni, bileşenlerinden hizmet alarak yerine getirir.
  - Buna, **yönlendirme** (**delegation**) denir.

- Bileşik nesnelerin arayüzleri, bileşenlerinden bağımsızdır.
  - Bileşik nesne, bileşenlerinin arayüzlerini toplayıp, farklı bir arayüzle kulanıma açar.

Bileşenlerin Yaratılması

- Bileşik nesnelerde, bileşenlerin ne zaman oluşturulacağı karar verilmesi gereken bir konudur:
  - Tanıtıldığı yerde ya da kuruculardа oluşturulabilir.
  - Bir başka yerde oluşturulup bileşik nesneye geçilebilir.

- İlk durum genel olarak daha sıkı (composition), ikinci durum ise daha gevşek bir ilişkiyi (aggregation) ifade edebilir.

Uygulama

- Selesi, ön ve arka tekerleği ile pedal takımı olan bir bisiklet sınıfı oluşturun.
  - Bisiklet sınıfı, belirtilen sınıflardan nesne değişkenlerine sahip olmalı ve kendisinden istenen hizmetlere, nesne değişkenleri yardımıyla cevap vermelidir.

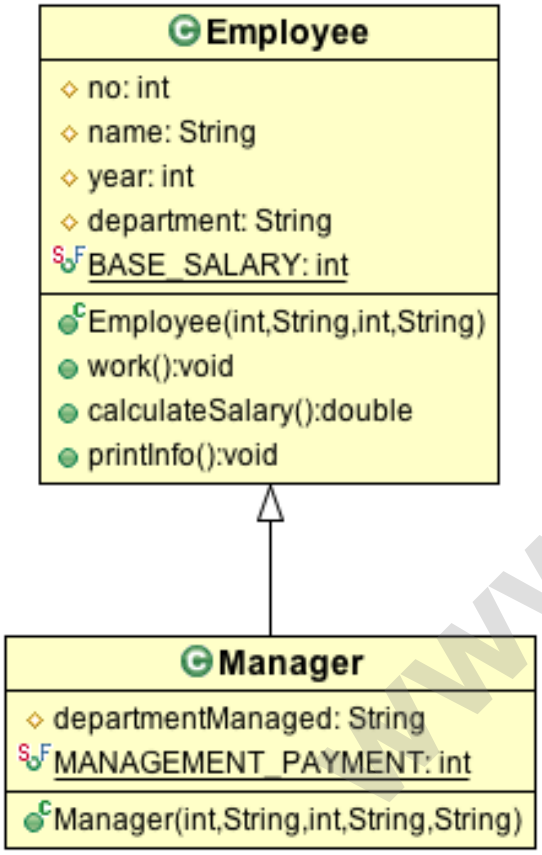
Miras - Kalıtım (Inheritance)

Miras I

- **Miras** ya da **kalıtım** (**inheritance**), aralarında yapısal benzerlik bulunan nesneleri ifade etmekte kullanılan, en yaygın ikinci tekrar kullanım kalıbıdır.
- Miras, **is-a** (**olma**) ya da **is-like-a** (**gibi olma**) ilişkisidir.
- Kendisinden miras alınan sınıfa **ebeveyn** (**parent/base**), miras alan sınıfa ise **çocuk** (**child/sub-class**) sınıf denir.
- Türetilen ya da çocuk sınıf, ebeveyninden, miras olarak alınabilecek üye değişkenler ile üye metotları devralır.
- Dolayısıyla, çocuk sınıflar, ebeveynlerine, durum ve davranış açısından benzerler.

Miras II

- Miras yapısını kurmak için Java'da **extends** anahtar kelimesi kullanılır:




```
public class Employee{
    protected int no;
    protected String name;
    protected int year;
    protected String department;
    ...
}

public class Manager extends Employee{
    protected String departmentManaged;
    ...
}
```

Miras İlişkisi - I

- Bu ilişki aşağıdaki okuma şekillerine imkan verir:
  - Her Manager/Bütün Managerlar aynı zamanda bir Employee'dir.
  - Her Manager/Bütün Managerlar bir Employee gibidirler.

- Çocuk sınıf, ebeveyninde **private** olan yapıları devralamaz, ancak **protected** , **public** ya da aynı pakette ise varsayılan olanları devralır.
- **protected** olan üye değişkenler halen dış dünyaya kapalıdırlar ama genelde **public** olan metotlar her halükarda devralınırlar.

Miras İlişkisi - II

- Miras ilişkisi ile tüm devralınabilecek olan ebeveyn üyeleri, çocuk sınıflar tarafından devralınır:
  - Nesne değişkenleri ve metotları
  - Sınıf değişkenleri ve metotları

- Ebeveynin kurucuları ise çocukları tarafından devralınmaz.

Üye Erişim Niteleyicileri II (Tekrar)

Niteleyici	Kendisi	Paketi	Çocukları (Farklı Pakette)	Herkes (Farklı Pakette)
public	+	+	+	+
protected	+	+	+	-
- (varsayılan)	+	+	-	-
private	+	-	-	-

InheritanceExample.java

- Önce, aynı paketteki ParentClass1 sınıfından devralmayı sonra da otherPakage paketindeki ParentClass2 sınıfından devralmayı deneyin, protected olan üyelerin davranışını gözlemleyin.

Başlatma (Initialization)

Mirasta Başlatma

- Hiyerarşide altta bulunan her sınıf, ebeveynindeki bir kurucuyu çağırmak zorundadır.
  - Bu da "her çocuk sınıfın nesnesinin içinde, gizli de olsa bir ebeveyn nesnesi var" anlamına gelmektedir.
- Bir sınıfın, ebeveynindeki bir kurucuyu çağırması **super()** ile olur.
- super() yoluyla yapılan kurucu çağrıları hiyerarşinin en tepesindeki sınıfa kadar devam eder.
- Dolayısıyla en önce hiyerarşinin en tepesindeki sınıfın kurucusu çağrılır ve nesnesi oluşur.
- Bunun dışında başlama sırasında değişen bir şey yoktur.

Başlama Sırası

- Dolayısıyla başlama sırası, sınıf hiyerarşisindeki en yukarıdaki sınıftan aşağıya doğru olur. Her sınıftaki başlama sırası ise
  - Sınıf değişkenleri (statik başlatma blokları dahil)
  - Nesne oluşturuluyorsa
    - Nesne değişkenleri (nesne başlatma blokları dahil)
    - Kurucu çağrısı

  şeklindedir.
- Birden fazla sınıf ve nesne değişkeni olduğu durumda başlatma sırası, fiziksel sırayla belirlenir, önce gelen önce başlatılır.

InitializationOrder.java

Kurucular (Constructors)

- Miras söz konusu olduğunda kurucularla ilgili iki önemli nokta söz konusudur:
  - Kurucular devralınmazlar.
    - Dolayısıyla her türetilen sınıf kendi kurucusunu tanımlamak zorundadır.
  - Hiyerarşide altta bulunan her sınıf, ebeveynindeki bir kurucuyu çağırmak zorundadır.
    - Bu da “her çocuk sınıfın nesnesinin içinde, gizli de olsa bir ebeveyn nesnesi var” anlamına gelmektedir.

super() Çağrısı - I

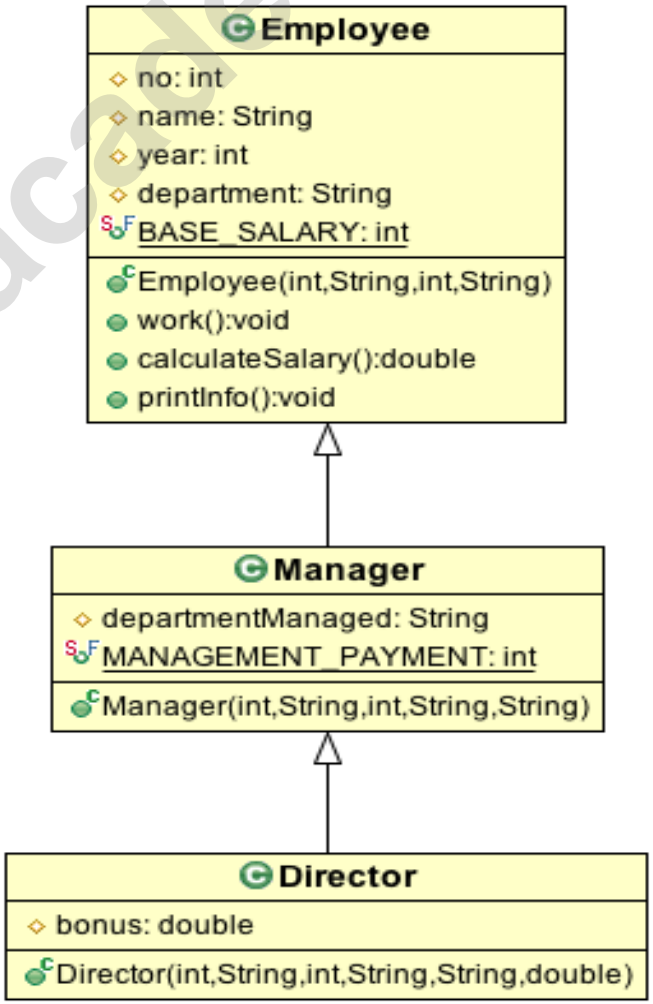
- Bir sınıfın, ebeveynindeki bir kurucuyu çağırması **super()** ile olur.
- super() çağrısı parametre geçmezse, ebeveyndeki argümansız kurucu çağrılmış olur.
- super() çağrısı tabi olarak parametre geçebilir, bu durumda ebeveyndeki bir akıllı kurucu çağrılmış olur.
  - Tipik olarak, ebeveynin tanımladığı durum bilgisi çocuk nesne oluşturulurken kurucusuna geçilir ve bu kurucu da bu durum bilgisini super() ile ebeveynindeki akıllı bir kurucuya geçer ki atamalar ebeveynin kurucusunda yapılsın.
  - Çocuk nesnenin kurucusuna geçilen ve ona has olan durum bilgisi ise ebeveyne geçilmez.

super() Çağrısı - II

- super() çağrısı içinde bulunduğu kurucuda ilk çalışan kod olmalıdır.
  - Dolayısıyla, çocuk nesne oluşmadan önce, içindeki gizli olan ebeveyn nesne oluşmalıdır.

Employee, Manager ve Director.java

- Employee, Manager ve Director sınıflarının kurucularına dikkat edin.
- org.javaturk.oop.ch09.factories.factory1.Test



InitializationOrder.java

Genişletme ve Yerine Geçebilme

Genişletme

- Çocuk sınıflar, ebeveynlerinden miras olarak devraldıkları yapılara ekleme yapabilirler:
  - Çocuk sınıflar, genel olarak, ebeveynlerinde olmayan, yeni üye değişkenlere ve yeni metotlara sahip olurlar.
  - **extends** anahtar kelimesi zaten bu genişletmeyi ifade etmektedir.

- Bu durumda türetilen çocuk sınıf, yeni üye değişkenlerle daha zengin bir yapıya, yeni metotlarla da daha geniş bir arayüze sahip olur.

Yerine Geçebilme I

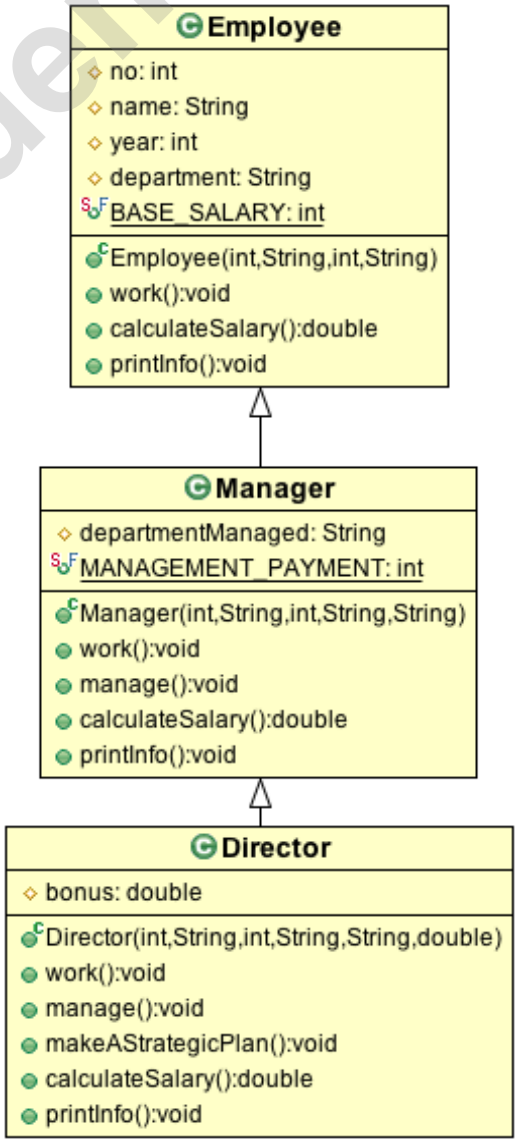
- Miras ilişkisinde çocuk sınıfın arayüzü, en azından ebeveyninin ara yüzüdür.
  - Çocuk sınıflar, ebeveynlerinden devraldıkları arayüze eklemeler yaparak daha geniş bir ara yüze sahip olup, daha çok iş yapar hale gelseler bile, ebeveynlerinin arayüzünü desteklemeye devam ederler.
  - Bu durum, üye değişkenler için de böyledir, yani ebeveynde erişilen her değişkene, çocuklarda da erişilir.
    - Ama prensip olarak değişkenlerin **protected** olduğunu ve dışarıdan ulaşılamadığını varsayıyoruz.
- Dolayısıyla, çocuk sınıflar, ebeveynlerinin sağladığı her özelliği, değişken ya da arayüz, sağlamak zorundadırlar.

Yerine Geçebilme II

- Bu durum, yerine geçebilme (**substitutability**) olarak ifade edilir ve hiyerarşide aşağıdan yukarıya doğru çalışır.
  - Yani, ebeveynin olduğu her yerde, ebeveynin çocuklarından birisi olabilir.
    - Her Manager aynı zamanda bir Employee'dir.
    - Yani, patron, "bana bir çalışan çağırın" dediğinde, ona bir Manager gelirse, patronun isteği yerine gelmiş olur.
    - Ya da patron, tüm çalışanlar toplansın dediğinde, Manager "beni çağırmıyor" diyemez.

Genelleştirme-Özelleştirme

- Miras ilişkisi, bir **genelleştirme-özelleştirme** (**generalization-specialization**) ya da **genel-özel** (**generic-specific**) ilişkisidir.
- Yani hiyerarşide yukarı çıkıldıkça daha genel nesneler, aşağı inildikçe, o nesnelerin daha özel halleri bulunur.
- Ama yerine geçebilme özelliği her zaman geçerlidir:
  - Her Director aynı zamanda hem bir Manager hem de bir Employee'dir.



Employee, Manager ve Director.java

- org.javaturk.oop.ch09.factories.factory2.Test

Uygulama

- Bir üniversitedeki öğrencileri, aralarındaki miras ilişkisini göz önüne alarak modelleyin.

- Hangi durumlarda genişletme söz konusudur tartışın.

Overriding (Ezme)

Overriding – Ezme - I

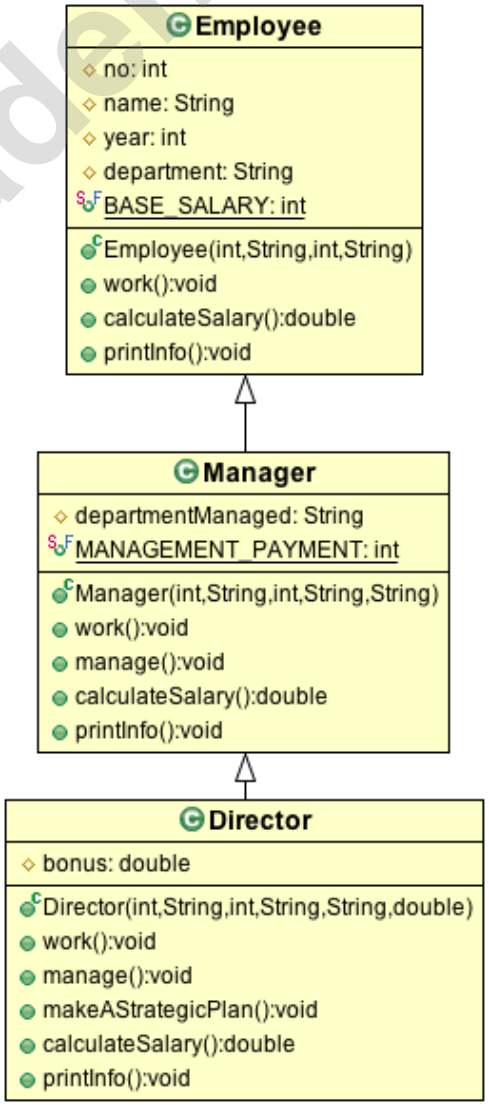
- Nesneler, ebeveynlerinden devraldıkları metotların arayüzlerini değiştirmeden, kodunu değiştirebilirler.
- Buna **overriding** ya da **ezme** denir.
- Yani çocuk nesneler, ebeveynlerindeki sorumluluğu, farklı bir şekilde yerine getirmeyi tercih edebilirler.
- Overriding ile aynı sorumluluk farklı şekillerde yerine getirilir:
  - Sorumluluk aynıdır, çünkü arayüz (interface) aynıdır, ama sorumluluğu yerine getirme şekli (implementation) farklıdır.
- Bu şekilde override edilen metotlara **polymorphic** (**çok şekilli**) metotlar denir.

Overriding – Ezme - II

- Bu şekilde override edilen metotlara **polymorphic** (**çok şekilli**) metotlar denir.
- Çünkü sorumluluk bir tanedir çünkü arayüz bir tanedir ve ebeveynde tanımlanır.
- Ama sorumluluğu yerine getirme yani metot birden fazladır.
- Bu yüzden override edilebilen metotlara polymorphic denir.

Employee, Manager ve Director.java

Employee üzerinde tanımlanan *printInfo()* ve *calculateSalary()* metotlarının *Manager* ve *Director* için override edildiğini gözlemleyin.



Overriding – Ezme - III

- Override, sadece nesne metotları için geçerlidir.
  - Nesne metotlarını aynı arayüzle alt sınıflarda tekrar tanımlarsanız, onları override etmiş (ezmiş) olursunuz.
  - Üye değişkenleri aynı isimle alt sınıflarda tekrar tanımlarsanız, ebeveyndekileri devralmamış, sadece saklamış olursunuz.
    - Çünkü overriding değişkenler için tanımlı değildir.
- Statik metotlar da override edilemezler.
- Polymorphic davranış sadece nesne metotları için geçerlidir, üye değişkenler ve statik metotlar polymorphic değildirler.
- Polymorphismi ileride ele alacağız.

Bir Nokta!

```
public class SubClass extends ParentClass{
    private int i;
    void f(){}
}

class ParentClass{
    public int i;
    public void f(){}
}
```

- Yukarıdaki kod derleme hatası verecektir?
- Neden?

Daha Kısıtlayıcı Olarak Override

- Override ederken, devralınan metodu daha kısıtlayıcı bir erişim belirteciyle tanımlayamazsınız.

- Aksi taktirde, ebeveyn üzerinden ulaşılan bir metodun, çocuk nesneler üzerinden ulaşılamaması söz konusu olurdu!

Override Ederken Alt Tip Parametre

- Java SE'nin 1.5 sürümünden itibaren, override ederken, devralınan metodun parametreleri, alt tipleriyle yer değiştirebilir.

Özet

- Bu bölümde, Java ile Nesne Merkezli Programlama'ya Giirş eğitiminin nesne ile ilgili konuları özetlendi.

Ödevler

Ödevler