Az Kod Çok İş için Bilinmesi Gerekenler

Sınıfların Tekrar Tekrar Kullanılması

Bir amaca hizmet eden sınıflarımızı oluşturduk. Bu sınıfları projenin başka yerlerinde veya farklı projelerde kullanmak işlerimizi kolaylaştıracak, zaman kazandıracak ve de hata almamızı engelleyecektir.

Bunu yapmak için iki yöntem mevcuttur.

1. Kompozisyon: Daha önceden yazılmış sınıfı yeni sınıf içinde kullanmak.
2. Kalıtım (Inheritance) : Yeni oluşturacağımız sınıfı daha önceden yazılmış sınıftan türetmek. Böylece yeni sınıf eski sınıfın özelliklerine sahip olacağı gibi, kendine ait ek özellikleri de içerebilir.
   İki sınıf arasında kalıtım uygulanacaksa sorulması gereke soru şudur:
   Yeni sınıf bir eski sınıf mıdır?
   Kaplan bir Kedi midir?(Kaplan, Kedi sınıfından türetilebilir.)
   Samsung bir CepTelefonu mudur?(Samsung, CepTelefonu sınıfından türetilebilir.
   Köpek Balığı > Balık > Hayvan Kalıtım Uygulanabilir mi?
   Evet, köpek balığı bir balıktır, balık bir hayvandır. Ayrıca dolaylı olarak köpek balığı bir hayvandır diyebiliriz ve kalıtıma uygundur.

Kompozisyon Kavramı

Aslında daha önceki örneklerde bilmeyerek bu kavramı uyguladık. Bir sınıf içinde String bir değişken tanımladığımızda daha önceden yazılmış olan bir sınıfı yeni oluşturduğumuz sınıfta kullanarak az kod çok iş yapmış olduk.

Aşağıda farklı bir kompozisyon kavramı örneği vardır. Bunu uygularken iki sınıfın arasında is - a ilişkisi olmadığını ama birbirlerinin parçası olduğunu görmenizi istiyorum.

```
class Motor{

    private String isim;
    private int beygirGucu;

    public Motor(){
        isim = "Honda";
        beygirGucu = 135;
        System.out.println(isim +" isimli ve "+beygirGucu+" beygirgüçlü motor oluşturuldu");
    }

    public void calistir(){
        System.out.println("Motor çalıştırıldı");
    }

    public void durdur(){
        System.out.println("Motor durduruldu");
    }



}

class Araba{

    private Motor arabaMotoru;
    private String renk;
    private String marka;
    private int uretimYili;

    public Araba(){
        arabaMotoru=new Motor();
        renk="Kırmızı";
        marka="Toyota";
        uretimYili = 2019;
        System.out.println(renk+" renkli "+uretimYili+" model "+ marka +" marka araba
oluşturuldu");
    }

    public void hareketeGec(){
        System.out.println("Araba harekete geçiyor");
    }

    public void durmayaBasla(){
        System.out.println("Araba durmaya başladı");
    }

}
```

Kalıtım Inheritance

Kalıtım yolu ile de daha önceden yazılmış kodları kullanarak yeni sınıflar oluşturabiliriz.

Kalıtım ile bir sınıf türetildiği sınıfın private alanları dışındaki tüm değişken ve metotlarına sahip olmuş olur.

İki sınıfı birbirinden türetmeli miyim konusunda kararsızsanız şunu sormalısınız. Acaba yeni sınıf bir türetilen sınıf mı? Acaba Kaplan bir Kedi Mi? Cevap evet ise kaplan sınıfını kedi sınıfından türetebilirsiniz.

Türetme işlemi extends anahtar kelimesi ile gerçekleşir.
class Kaplan extends Kedi{} böylece kaplan sınıfı kedi sınıfındaki private olmayan değişken ve metotlara erişebilir.

Aslında oluşturduğumuz her sınıf otomatik olarak bir sınıftan türetilir. Javada her sınıf Object dediğimiz bir üst sınıftan gizli bir şekilde kalıtılır.

Oluşturduğumuz her sınıf Object sınıfının metotlarına sahip olur. Bu metotlara şunları örnek verebiliriz:

1. clone: Nesneyi kopyalar.
2. equals(Object obj) : obj ye bağlı olan nesnenin kendisine eşit olup olmadığını söyler.
3. finalize() : Çöp toplayıcı tarafından silinmeden önce çalıştırılır.
4. hashCode(): Nesnenin hash kodunu verir. hasCode degerini objeler icin ID degeri olarak dusunebiliriz. Obje değiştikçe hashcode da değişir.
5. toString(): İlgili nesneyi string olarak ifade etmek için kullanılır.

Java dilinde çoklu kalıtıma izin verilmez. Her sınıf en fazla bir sınıftan türetilebilir.

Kalıtım demek bir sınıfın başka bir sınıfı kopyalaması demek değildir. Kalıtım bir sınıfın türetildiği sınıfın alanlarına erişebilmesi ve gerekiyorsa kendi alanlarını oluşturabilmesidir.
KopekBalik > Balik > Hayvan > Object bu yapıda bir kopek balığı nesnesi oluşturulmadan önce mutlaka önce object sonra hayvan sonra balik sınıfı nesneleri oluşturulur. En son kopek balığının kurucu metotu çalıstırılır.

NOT:
Alt sınıfa ait bir nesne üst sınıfa ait bir nesne olarak gösterilebilir.
Hayvan h=new KopekBaligi();

super()--> bu ifade üst sınıfa ait ve parametre almayan constructor'ı çağırır.

Üst sınıfa ait bir metotun alt sınıflarca tekrar yazılmasına **method overriding** denir. Üst sınıftaki bir metot alt sınıfta isim, parametre sayısı, parametre turleri ve geri dönüş tipi aynı olacak şekilde tanımlanır. Alt sınıf isterse bu metotun içeriğini değiştirebilir.

Polymorpizm

Bir alt sınıf aslında üst sınıfın özelleştirilmiş halidir.Alt sınıf üst sınıfın tüm özelliklerini ve kendine özgü alanlarını barındırır.
class Kopek extends Hayvan örneğinde Kopek sınıfının her nesnesi aslında Hayvan sınıfı nesnesidir, tam tersi doğru değildir.

Her dikdortgen bir geometrik şekildir ama her geometrik şekil bir dikdortgen değildir diyebiliriz.

Bundan dolayı siz üst sınıf tipine alt sınıf referansını bağlayabilirsiniz.

```
Hayvan h=new Kopek();
Kopek k=new Hayvan();//yanlış ifade
GeometrikSekil gs=new Dikdortgen();
```

Şöyle bir örnek yapalım;

```
GeometrikSekil gs1 = new GeometrikSekil();
Kare k1=new Kare();
Dikdortgen d1=new Dikdortgen();

alanHesapla(gs1);
alanHesapla(k1);
alanHesapla(d1);

public static void alanHesapla(Geometrik sekil){
    System.out.println(sekil.alanBul());
}

public static void alanHesapla(Kare sekil){
    System.out.println(sekil.alanBul());
}

public static void alanHesapla(Dikdortgen sekil){
    System.out.println(sekil.alanBul());
}
```

Herhangi bir sorun olmadan yukarıdaki kod çalışacaktır. Alt sınıf nesnesi mesela k1veyad1, üst sınıfın nesnesinin kullanıldığı her yerde kullanılabilir.(alanHesapla metotunun parametresine bakın) Bu duruma **çok biçimlilik polymorphism** denir.
Kısaca polymorphism üst sınıf değişkeninin alt sınıf nesnelerini referans edebilmesidir.

alanHesaplaya yolladıgımız k1 ve d1ye otomatik olarak tip dönüştürme uygulandı ki buna **upCasting** yani üst sınıfa dönüştür diyoruz.
Aslında burada gizli başka bir kavram daha var **Geç Bağlama (Late binding)**. Bir nesne örneğinin hangi nesneye bağlandığı ve hangi nesneye ait olduğunun çalışma anında belirlenmesidir.(Runtime)
k1.alanHesapla() --> bu ise derleme zamanında belirlenir(Compile time)

Binding Çeşitleri

Static veya Early Binding veya Erken Bağlama

static ve final metotlar compile time da bind edilir. Çünkü bu metotlar override edilemezler. Bundan dolayı daha kodu yazarken hangisinin çalıştırılacağı bellidir.

Dynamic Late Binding Geç Bağlama

Her sınıf gizli bir kalıtımla Object sınıfından türetilebilir demiştik. Ve object sınıfının toString metotu bulunmaktaydı. Oluşturacağımız tüm sınıflar bu metotu override ederek kendi isteklerine göre değiştirebilirdi. Peki:

```
Object obj=new GeometrikSekil();
System.out.println(obj.toString());
```

bu ifade de Object sınıfının mı yoksa GeometrikSekil sınıfının mı toString metotu çağrılacak ? Gelin bu sorunu inceleyelim...

```
Object obj=new GeometrikSekil();
Object obj;
obj=new Object();
```

Her değişkenin mutlaka bir veri türü olmalıdır. Buna declared type denir.
Değişken adımız obj, declared type : Object

Bu değişken türü referans tipli olduğu için ya null ya da declare edilmiş veri türünden olan bir nesnenin referansını saklar. Ayrıca declare edilmiş sınıfın alt sınıflarının referanslarını da saklayabilir.

Geometrik şekil Object sınıfının alt sınıfı olduğu için sorun çıkmayacaktır. Ve obj değişkenin declared tipi Object bile olsa aslında GeometrikSekil türündeki nesneyi gösterir. Bundan dolayı ;
Değişken adımız obj, actual type : GeometrikSekil

obj.toString() çağrıldığında hangi sınıftaki metotun tetikleneceği obj nesnesinin actual type'ına göre belirlenir. Yani obj.tostring çağrıldığında GeometrikSekil sınıfının toString metotu tetiklenir.

Bu duruma **dinamik bağlama veya dynamic binding** denir.Kısacası kalıtım zincirinde bir metotun farklı sınıflarca kullanılması durumunda JVM'nin çalısma anında hangisini çalıstıracağına karar vermesi olayıdır.

Birden fazla üst sınıf olduğunda cagrılan metot en alt sınıftan en üst sınıfa doğru aranır. Bulunmazsa bir üst sınıfa bakılır.

Farkları

- Static binding compile zamanı, dinamik binding çalışma zamanında olur
- Static bağlama static ve final metotlarda olur çünkü bunlar değiştirilemez. Dinamik binding çalışma anında değişkenin nesneye bağlandığı zaman override edilmiş metotlar sayesinde olur.

Casting İşlemleri ve Instanceof

Aşağıdaki sınıflarda Kare Dikdortgen sınıfından, Dikdortgen de GeometrikSekil sınıfından extend edilmiştir.

```
GeometrikSekil gs2=new Kare(5);//implicit casting, dolaylı olarak çevrim yapılır
   GeometrikSekil gs3=new Dikdortgen(5,10);//implicit casting
   //bu ifadeler doğrudur çünkü her kare ve her dikdortgen bir geometrik şekildir.

   Dikdortgen d2=new Kare(6);//implicit casting
   //bu ifade de doğrudur çünkü her kare bir dikdortgendir

   //Kare k3 = new GeometrikSekil(5,10);//bu ifade yanlıstır çünkü her  geometrik
şekil kare olmayabilir

   GeometrikSekil gs4=new Kare(5);

   Kare k4= gs4;//her ne kadar gs4 kare nesne referansını tutsa da bu ifade hata
verir.
   //Compiler bunu anlayacak kadar zeki değil. k4'e geometrik şekil atadığımızı
sanıyor.
   //Bu ifadeyi çalışır yapmak için bunu explicit casting ile belirtmemiz gerekir.

   Kare k5= (Kare) gs4;//buna da downcasting denir
```

Eğer polimorfizm olmasaydı şöyle bir kod yazmak zorunda kalacaktık:

```
 public static void alaniGoster(GeometrikSekil g){

        //polymorpizm olmasaydı bu şekilde yapardık
        if(g instanceof Kare){
            Kare gecici = (Kare) g;
            gecici.alanHesapla();
            System.out.println("kare sınıfındaki alan hesapla çalıstı");
        }else if(g instanceof Dikdortgen){
            Dikdortgen gecici = (Dikdortgen) g;
            gecici.alanHesapla();
            System.out.println("dikdortgen sınıfındaki alan hesapla çalıstı");
        }else {
            GeometrikSekil gecici = g;
            gecici.alanHesapla();
            System.out.println("geometrik şekildeki alan hesapla çalıstı");
        }
        //polimorfizm olduğu için
        g.alanHesapla();

    }
```

Bazı Object metotları 1

Equals Metotu

İki nesnenin birbiri ile eşit olup olmadığının kontrolünü yapar. Varsayılan halinde ogrenci.equals(ogrenci2) dediğimizde ogrenci ve ogrenci2 referanslarının aynı alanı göstermesi gereklidir.

Wrapper sınıflar ve String sınıfları istisnadır. Burada referansların aynı nesneyi tutmasına değil değerlere bakılır.

```
String isim = new String("isim");
String isim2= new String("isim");
```

İsim ve isim2 farklı nesneleri tutsa da equals metodu true değer verecektir.

Çoğu zaman bu istediğimiz bir durum değildir. Bundan dolayı iki nesneyi hangi alana göre kıyaslamak ve eşitliğini kontrol etmek istiyorsak bu metotu override etmeliyiz.

```
public boolean equals(Object o) {
if (o instanceof Daire)
return yaricap == ((Daire)o).yaricap;
else
return this == o;
}
```

Peki bir sınıf içinde bu metota Object yerine o sınıfın referansını yollarsak ne olur?

```
public class Test {
  public static void main(String[] args) {
    Object circle1 = new Circle();
    Object circle2 = new Circle();
    System.out.println(circle1.equals(circle2));
  }
}
```

```
class Circle {
  double radius;

  public boolean equals(Circle circle) {
    return this.radius == circle.radius;
  }
}
```

```
class Circle {
  double radius;

  public boolean equals(Object circle) {
    return this.radius ==
      ((Circle)circle).radius;
  }
}
```

Equals metotu override ediliyorsa hashcode metotu da override edilmeli. Bu konudan collections kısmında bahsedilecektir.

Bazı Object Metotları 2

Finalize metotu

Çalışmakta olan bir programda, bir nesnenin artık kullanılmadığına ve silinmesi gerektiğine karar veren mekanizmaya çöp toplayıcı demiştik. Finalize metotu bir nesne silinmeden önce tetiklenen metottur.

Şunu unutmamak gerekir: Çok fazla çöp nesne üretilmiyorsa çöp toplayıcı devreye girmeyebilir. System.gc() ile tetikleyebiliriz ama bunun dışında garbage collector ın ne zaman devreye gireceğini bilemeyiz.

```
for(int i = 0 ; i< 100; i++){

    Ogrenci a=new Ogrenci(i);
}

//bu kodda a referansına ogrenci nesneleri bağlanmaktadır. Ama her seferinde a değişkenine farklı nesneler bağlandıgı için diğer nesneler artık çöp olarak kalmaktadır.
```

Acaba çöp toplayıcı tetiklenecek mi? finalize metotu harekete geçecek mi?

System.gc() ile elle tetikleme yapılabilir.

Final Nedir ?

Final kelime anlamı olarak son demektir ve javada da final kullanarak değişken metot ve sınıf oluşturabiliriz.

Oluşturulan final değişkenlere bir kere değer atandıktan sonra bu değer değiştirilemez. Bundan dolayı sabitlerimizi belirlerken final olarak belirtiriz.

```
final int SAYFA_BASINA_GONDERI_SAYISI = 5;
```

Bu değişkene sonradan baska bir değer atanamaz.

Bir sınıfı final olarak tanımladığınızda o sınıftan başka sınıflar türetilemez. Şu ana kadar kullandığımız Math sınıfı bu şekildedir. Ondan dolayı bu sınıftan baska bir sınıf türetemeyiz.

Bir sınıf içindeki bir metotu final olarak belirtebilirsiniz. Final olarak belirtilen metotlar alt sınıflar tarafından override edilemez.

final tasarımsal istekleri sağladığı gibi performans olarak da verimlilik sağlar.

bir metotu final yaparak o metotun alt sınıflarca override edilmesinin önüne geçeriz. Ondan dolayı late binding durumu da söz konusu olmaz. Eğer sınıfı final yaparsak performans çok daha artacaktır.

String ve Ozellikleri

String is Immutable

String sınıfı özel bir sınıftır. Referans tipli olmasına rağmen new demeden kullanabiliriz. Kendine özgü yardımcı metotları bulunur ve string nesneleri değiştirilemez yani immutable'dır.

```
String isim = "emre";
isim ="yeni emre";
```

isim stringini yeni değerle değiştirdik. Hani değişmezdi ? Gelin bunu açıklayalım.

Bir sınıfın değişmez olması bir kere değer atadıktan değişemez olmasıdır. Yukarıdaki örnekte isim değişkenine değer atandıktan sonra aslında gelen yeni değer olan yeni emre için yeni bir nesne oluşturulur. Bundan dolayı string immutable bir sınıftır.

Daha detaylı anlatım videoda bulunmaktadır.

String isim ="emre" değeri string poolda tutulur. String pool heap alanı içinde sadece string sabitleri tutan özel bir alandır. Buraya emre değeri yazıldıktan sonra bunu değiştiremeyiz.
isim.concate(" altunbilek") dediğimizde ve sout(isim) dediğimizde yine emre değerini görürüz.

String tamAd= isim.concate(" altunbilek") sout(tamAd) dersek artık ekranda emre altunbilek değerini görürürz. Yeni bir nesne oluşturulmustur.

String poolda aynı değerli iki string tutulmaz.
String ad="emre";
String ad1 = "emre"; ad ve ad1 aynı değeri gösteren iki farklı referanstır.

ad.concate("altunbilek") heap alanında emre altunbilek değerine sahip yeni bir alan oluşturur.

String isim=new String("emre") bu ifade ise heap alanında bir string nesnesinin oluşturulmasına neden olur. Biz bu değeri string pool alanına tasımak istiyorsak String isim=new String("emre").intern() metotunu çağırırız. Böylece emre değeri hem heap hem de string pool da olur

Sonrasında String ad = "emre" denirse ikisi de aynı alanı gösterdiği için s1 ==s2 true değerini verir.

String Sınıfı Metotları

Önemli string sınıfı metotları şunlardır.

String isim = "emre";

- isim.length() -> kaç karakter olduğunu döndürür. 4
- isim.charAt(index) -> index=0 için ilk elemanın ne olduğunu döndürür. e
- isim.concat(" altunbilek") --> parametre olarak verilen ifade ile isim değişkenini birleştirir ve YENI bir string döndürür. + ile de yapılabilir.
- isim.toUpperCase()-->tüm karakterleri büyük harfe dönüştürülmüş YENI bir string döndürür.

  isim.toLowerCase()-->tüm karakterleri büyük harfe dönüştürülmüş YENI bir string döndürür.
- isim.trim()-->iki taraftaki boşlukları siler ve YENİ bir string döndürür.

Bu metotların kullanılması için mutlaka bir string nesnesi gerektiği için bunlara instance metot denir.

isim.equals(isim2) --> içerik olarak isim ve isim2 stringlerini karşılaştırır.

isim.equalsIgnoreCase(isim)-->Büyük küçük harfe duyarlılığı önemsemeden iki değeri karşılaştırır. emre ve EmrE için true değer döndürür.

isim.compareTo(isim2) --> iki stringi alfabetik olarak kıyaslar.

isim.startWith(deger) --> isim stringi deger ile mi başlıyor ona bakar

isim.endsWith(değer) --> isim stringi değer ile mi bitiyor ona bakar.

isim.contains(deger) --> isim stringi içinde değer var mı ona bakar.

isim.substring(0,5)--> 0 ile 5 arasındaki karakterleri içeren string döndürür.

isim.index(ifade) --> isim stringinde ifade stringinin geçtiği ilk yerin indeksini döndürür. Bulamazsa -1 döndürür.

isim.lastIndexOf(ifade)--> isim stringinde ifade stringinin son geçtiği yerin indeksini döndürür.

index ve lastIndexOf metotları parametre olarak char değer de alabilir.

int sayi = Integer.parseInt("1") metin olan 1'i int olan 1 e dönüştürür.
Tamsayı bir ifadeyi stringe dönüştürmek için sayi.toString() metotu yeterlidr

Arraylist Sınıfı ve Metotları

Neden ArrayList Sınıfı ?

Nesneleri tutmak için bir dizi tanımlayabiliriz. Ama dizilerin boyutları sabittir ve değiştirilemez. Java da boyutu değişebilen yapılar için Arraylist sınıfı bulunmaktadır.

Daha fazla ayrıntı ve diğer liste çeşitleri koleksiyonlar kısmında anlatılacaktır.

ArrayList<Ogrenci> ogrenciler = new ArrayList<Ogrenci>(); diyerek içinde öğrenci nesneleri tutacağımız bir arraylist oluşturmuş oluruz.

ArrayList Sınıfının Metotları

ogrenciler.size() --> arraylistte tutulan eleman sayısını verir.

ogrenciler.add(emre) --> ogrenci nesnesi olan emre referansını ekler.

ogrenciler.contains() --> bir elemanın listede olup olmadığını kontrol eder.

ogrenciler.remove(emre) --> elemanı listeden siler.

ogrenciler.get(i) --> i. indeksteki elemanı okur.

ogrenciler.set(index, hasan);--> verilen indekse hasanı yerleştirir.

Elinizde bir dizi varsa bunu listenize aktarabilirsiniz.

Arraylist<String> isimListesi = new Arraylist<>(Arrays.asList(dizi));

isim tutan dizi yapısı isimlistesine aktarıldı. new Arraylist<> dedikten sonra veri türünü belirtmek zorunda değiliz.

Bu listeyi Collections.sort(isimListesi) diyerek sıralayabiliriz.