Nesneye Yönelik Programlama Dillerinde Inheritance(Kalıtım)

Encapsulation hakkında bilgi vermiştik. **Encapsulation'un** Java'nın temel yapı taşlarından biri olduğunu ifade etmiştik. İşte **inheritance** yani kalıtım da Java'nın temel ikinci yapıtaşıdır. Bu nedenle **inheritance** kavramını çok iyi öğrenmek gereklidir.
Bir örnekle inheritancenin ne olduğunu açıklayalım:

```
1  public class GeometricObject{
2      private String color="white;
3      private boolean filled;
4      private java.util.Date dateCreated;
5
6      public GeometricObject(){
7          dateCreated=new java.util.Date();
8      }
9      public String getColor(){
10         return color;
11     }
12     public void setColor(String color){
13         this.color=color;
14     }
15     public boolean isFilled(){
16         return filled;
17     }
18     public void setFilled(boolean filled){
19         this.filled=filled;
20     }
21     public java.util.Date getDateCreated(){
22         return dateCreated;
23     }
24     public String toString(){
25         return " created on"+dateCreated+"\ncolor:"+color;
26     }
27 }//end of GeometricObject
28
29
30 public class Circle extends GeometricObject{
31     private double radius;
32     public Circle(){
33     }
34     public Circle(double radius){
35         this.radius=radius;
36     }
37     public double getRadius(){
38         return radius;
39     }
40     public void setRadius(double radius){
41         this.radius=radius;
42     }
43     public double getArea(){
44         return radius*radius*Math.PI;
45     }
46 }//end of Circle
47
48 public class TestCircle{
49     public static void main(String[] args){
50         Circle daire=new Circle();
51         daire.setRadius(32);
52         daire.setColor("blue");
53         daire.setFilled(true);
54
55         System.out.println(daire.getColor()+daire.getFilled());
56     }
57 }//end of TestCircle
```

Soru: Sizce bu program çalışır mı?
Cevap: Çalışır. Ama hata vermesi gerekmez miydi? Çünkü **Circle** sınıfından **setColor(), setFilled(), getColor(), getFilled()** metodları mı var ki, **Circle** nesnesi bunları çağırmış?
Madem bu program çalışıyor o halde bu metodlar **Circle** sınıfının içende yokken nereden geldi?
Dikkat edersek **Circle** sınıfı tanımlanırken;
public class Circle extends GeometricObject
dedik. Yani bir keyword **extends** kullanıldı. Bu işleme **inheritance** denir.
Yani **extends** keyword ile **GeometricObject** sınıfında **public** olarak tanımlanan tüm metodlar **Circle** sınıfının içerisine bir nevi yerleştirilmiş oldu. Yani biz iki kez hem **GeometricObject** sınıfında, hem de **Circle** sınıfında aynı metodları yazmaktan kurtulmuş olduk. Bunu ise **extends** keyword ile sağladık.
Diyelim ki, 10 tane sınıf tanımladık. Bu sınıflar **GeometricObject** içindeki tüm metodlara ihtiyaç duyuyor. On kez aynı metodları yazmaya ne gerek var ki? Inheritance ile sadece **extends** keyword kullanarak **GeometricObject** sınıfındaki ihtiyaç duyulan metodlara erişebilmekteyiz.

Bazı Kavramlar:

GeometricObject sınıfı superclass olarak adlandırılır.
Circle sınıfı subclass olarak adlandırılır.
Bu adlandırma sadece insan dilinde kullanılır, yani kodta superclass, subclass gibi keyword yazılmaz
Ya iyi güzelde bu **inheritance** işlemi nasıl gerçekleşiyor?
Hani bizim bir metodumuz vardı:
Constructor metodu. Yani sınıfı memory'e yükleyen metod.
Bu metod sayesinde **inheritance** gerçekleşiyor:

```
1  public Circle(){ //subclass olan Circle sınıfı
2     //superclass olan GeometricObject constructor'unu çağıran metod
3     super();
4  }
```

Ama önceki örnekte **Circle** sınıfının constructor'u içerisinde **super()** metodu yoktu?
Peki o halde **GeometricObject** sınıfının constructor **super()** metodunu **Circle** sınıfının constructor'u içinde çağırdığına göre ve önceki örneğimizde **super()** metodu kullanılmadığına göre nasıl oldu da **GeometricObject** sınıfını extend ettik? Yani **GeometricObject** sınfındaki metodları **Circle** sınıfında kullanabildik?
Cevap: Biz **super()** metodunu subclass constructor'u içinde yazsakta yazmasakta o otomatik olarak Java Compiler tarafından yazılıyor.
O halde biz niçin biraz önce yazma gereği duyduk?
Cevap: Explicitly olarak yazabildiğimizi kanıtlamaki çin ve bir sınıfın nasıl extend edildiğini açıklamak için yazdık.

Overriding Method

Metodların nasıl overloading edildiğini anlatmıştık. Peki **overriding** etmek nasıl oluyor?
GeometricObject sınıfında yani superclass'taki bir metodu beğenmedik diyelim ve bu nedenle metodtaki kodları değiştirmek istiyoruz. İşte yapılan bu işleme **overriding** denir.
Ama dikkat edelim: Subclass'taki metod ismi parametreler ve metodun dönüş tipi superclass'taki metod ile aynı olacak.
GeometricObject Sınıfında:

```
1  public String toString(){
2      return " created on"+dateCreated+"\ncolor:"+color;
3  }
```

Circle sınıfında yeni bir **toString()** metodu yazalım:

```
1  public String toString(){
2      return " oluşturulma zamanı"+dateCreated+"\rengi:"+color;
3  }
```

extend edildiği zaman **Circle** sınıfında **GeometricObject** sınıfının tüm metodları oluyordu. O halde **Circle** sınıfında yeni yazdığımız **toString()** metodu ile **GeometricObject** sınıfındaki **toString()** metodu çakışmıyor mu? Çünkü aynı ismi kullanıyorlar.
İşte böyle bir durumda **GeometricObject** sınıfındaki **toString()** metodunun üzerine **Circle** sınıfındaki **toString()** metodu yazılıyor. Yani **overriding** ediliyor. **Circle** nesnesi ise **GeometricObject** sınıfındaki **toString()** metoduna değil **Circle** sınıfındaki **toString()** metoduna erişebiliyor.
Peki **GeometricObject** sınıfındaki **toString()** metodunu kullanmak istediğimiz zaman ne yapacağız?
Cevap: super keyword ile. **Circle** sınıfı içerisindeki bir metod içine **super.toString()** ifadesini yazarsak **GeometricObject** sınıfının **toString()** metodu çağrılmış olur.

Object Class

Bazen bir IDE(Eclipse, Netbeans, Intellij) kod yazarken bir sınıftan nesne oluşturduktan sonra **nesneAdi.** dediğimiz zaman hem bu sınıfın metodları hem de başka metodlar gelmektedir. Bu başka metodlar nereden gelmektedir? Örnek olarak benim **Circle** sınıfımda bu metodlar yokken ve ben hiçbir sınıfı **extends** etmemişken? **Cevap:** Java'da yazdığımız her sınıf **default** olarak Object isimli bir sınıfı **extends** eder. Aynı zamanda Java kütüphanesindeki tüm sınıflarda Object sınıfını **extends** eder.
Örneğin, **GeometricObject** sınıfını **extends** eden **Circle** sınıfını düşünelim: **Circle** hem Object hem de **GeometricObject** sınıfını **extends** etmiş anlamına gelmektedir. Bunun anlamı ise şudur: **Circle** hem Object hem de bir **GeometricObject** tir.
Extends etme işleminde dikkat edilmesi gereken nokta şudur: Superclass olacak sınıf ile subclass olacak sınıf arasında mantıksal bir bağ olmalıdır. Örneğin **Tree** isimli bir sınıf olsa ide **Circle** ise bu sınıfı **extends** etmiş olsa idi, bunun anlamı **Circle** aynı zamanda bir Tree'dir olacağı için böyle bir kalıtım saçma olacaktı. Fakat, **Circle GeometricObject** sınıfını **extends** ettiği için **Circle** aynı zamanda bir **GeometricObject'tir** demek mantıklı olmaktadır.
Java'da kod yazarken bu mantığa göre hareket etmemiz gerekmektedir. Yani genelden özele doğru gitmek gereklidir. Buradaki genel **GeometricObject** sınıfı, özel ise **Circle** sınıfı demektir.