Nesneye Yönelik Programlama Dillerinde Polymorphism(Çok Biçimlilik)

Polymorphism kelimesi, biyolojide bir organizmanın veya türlerin birden fazla farklı formlara veya aşamalara sahip olması anlamına gelen bir prensibi ifade eder. Bu prensip nesne tabanlı programlama dillerine uyarlanmıştır. Bir sınıfın subclass'ı superclass'tan farklı olarak kendi eşsiz davranışlarını oluşturabilirken aynı zamanda superclass'ın da bazı fonksiyonlarını paylaşabilmektedir. Örneğin; **Circle** sınıfı kendine has özelliklere sahipken, aynı zamanda, superclass olan **GeometricObject** sınıfının bazı özelliklerine de kalıtım yoluyla sahip olmuştur.

Subtype: Subclass'ın tipi yani ismi
Supertype: superclass'ın tipi yani ismi

Örnek: **Circle** bir subtype, **GeometricObject** ise bir supertype
Her **Circle** nesnesi aynı zamanda **GeometricObject** nesnesidir diyebiliyorken, her **GeometricObject** nesnesi **Circle** nesnesidir diyemiyoruz.
Her aslan bir hayvandır, fakat her hayvan bir aslan değildir.
Bütün bunları niçin söylüyoruz?
Bir örnekle açıklayalım:

```
public class Polymorphism{
    public static void main(String[] args){
        m(new GraduateStudent());
        m(new Student());
        m(new Person());
        m(new Object());
    }
    public static void m(Object x){
        System.out.println(x.toString());
    }
}//end of Polymorphism

public class GraduateStudent extends Student{
}
public class Student extends Person{
    public String toString(){
        return "Student";
    }
}
public class Person extends Object{
    public String toString(){
        return "Person";
    }
}
```

Program Çıktısı:
Student
Student
Person
java.lang.Object @130c19b

m metoduna dikkat edersek m(**Object** x) şeklinde tanımlanmış, bunun anlamı şudur: m metodu parametre olarak **Object** nesnelerini alır. Her sınıf **Object** superclass'ını **extends** ettiğine göre tüm nesneleri bu metoda geçirebiliriz. Eğer m(**Person** x) demiş olsaydık, sadece **Person** sınıfını ve varsa **Person** sınıfının subclass'larını geçirebilirdik.
Diyelim ki, metodumuza sadece **GeometricObject** olan nesneler geçirmek istiyoruz. Çünkü metodumuzda yazdığımız kodlar **GeometricObject** tarzı nesneler için kullanılmaktadır. Gidipte bu metoda Hayvan türünden bir nesne geçirirsek çok mantıksız olur.
Bir başka örnek:
Java **JDK** kütüphanesinde Number sınıfı, Double, Integer sınıfları tarafından extend edilen bir sınıftır.

```
public int m(Number a){
    int sayi=(Integer) a*12;
    return sayi;
}
```

Eğer bu metoda aşağıdaki gibi daire nesnesini göndermek istersek hata meydana gelir.

```
Circle daire=new Circle(12.5);
m(daire);
```

Çünkü m metodu sadece Number nesnelerini veya Number sınıfını extend sınıfların nesnelerini parametre olarak alır.

```
Integer sayi=new Integer(12);
m(sayi);//Bu ifade doğrudur. Çünkü Java JDK'da Integer sınıfı Number sınıfını extend eder
Double sayi=new Double(11.32);
m(sayi);//Bu ifade doğrudur. Çünkü Java JDK'da Double sınıfı Number sınıfını extend eder
```

Sonuç: Görüldüğü gibi Polymorphism bir nesnenin birden çok biçime, özelliklere sahip olabilmesini sağlar. Yani **Circle** sınıfından yaratılmış bir nesne yeri geldiği zaman **Circle** gibi davranırken, yeri geldiğinde **GeometricObject** şeklinde davranabilmektedir.

Casting İşlemi

```
Object o=new Student(); //Doğru
Student b= (Student)o; //Doğru
Student b=o; //Yanlış
```

o nesnesi bir **Object** sınıfı nesnesidir. Fakat memory'de tuttuğu adres **Student** sınıfıdır.
b nesnesi ise **Student** sınıfı türünden olan atama işlemi gerçekleşmeden önce memory'de hiçbir yere point etmeyen yani bir adres tutmayan referans değişkendir(nesnedir).
b nesnesi kendi sınıfından yaratılan yerin adresini tutabilir veya subclass'lardan yaratılan yerin adresini tutabilir. Superclass'ından yani **Object** sıfından yaratılan yerin adresini tutamaz. Bu nedenle

```
Student b=o;
```

işlemi yanlış olmaktadır. İllaki tutmasını istiyorsak casting operatorü şarttır. Bu nedenle

```
Student b= (Student)o;
```

ifadesi doğru olmuştur. Tabi casting işlemi kafaya göre yapılmamaktadır:

```
Object o=new Object();
Student b=(Student)o; //Yanlış
```

Çünkü o nesnesi subclass'ından yaratılan yerin adresini değil, kendi sınıfından yaratılan yerin adresini tutuyor.

instanceOf Operatorü:

```
Object myObject=new Circle();
if(myObject instanceOf Circle){
    System.out.println(" The circle diameter is "  +((Circle)myObject).getArea());
}
```

If'in içinde deniliyor ki, eğer myObject referans değişkeni yani nesnesi **Circle** sınıfından yaratılan yerin adresini tutuyorsa, ki bu örneğimizde tutuyor, **true** değerini dönder ve **if** içindeki kodları çalıştır, tutmuyorsa **false** dönder.

Bir sınıfın extend edilmesini önlemek

final keyword ile yapılır

```
public final Circle{

}
```

Böyle tanımlandığı zaman hiçbir sınıf **Circle** sınıfını **extends** edemez. Java **JDK** kütüphanesindeki Math sınıfı böyle tanımlanmıştır.
Ayrıca bir metodun **overriding** edilmesini önlemek istiyorsak yine **final** keywordu kullanılmalıdır.

```
public final void m(){
}
```

protected ve default visible modifiers

Modifiers	Aynı Sınıftan Erişim	Aynı Paketten Erişim	Alt Sınıftan Erişim	Farklı Paketten Erişim
public	✓	✓	✓	✓
protected	✓	✓	✓	-
(default)	✓	✓	-	-
private	✓	-	-	-

Örnek:

```
public class A{
    private int sayi=12;
    int sayi2=23; //default
    protected int sayi3=45;
    public int sayi4=28;
}
```