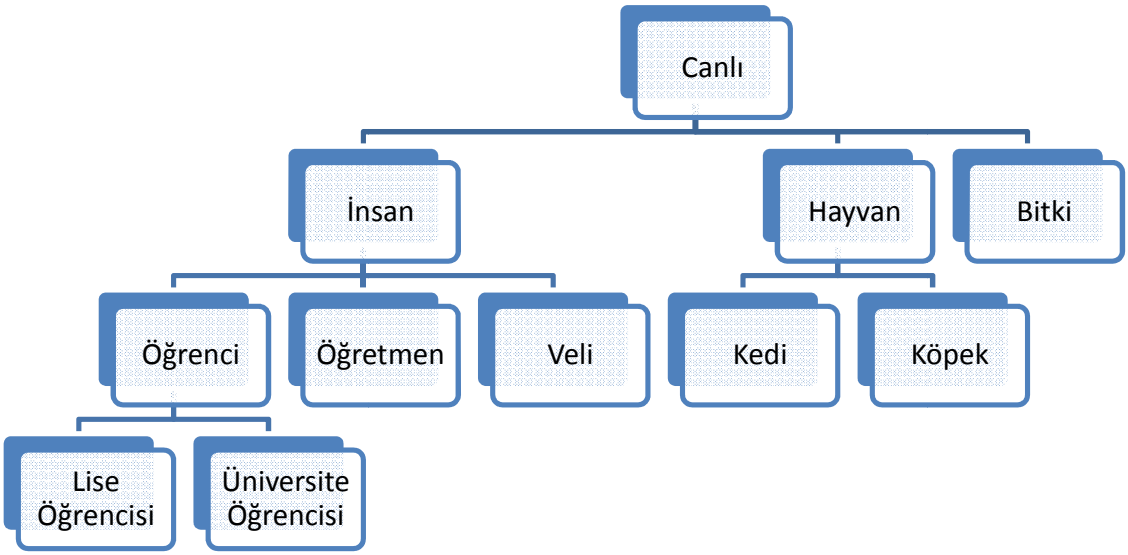
Miras - Inheritance

Nesneye Dayalı Programlamada Miras Modeli

Miras kelimesi nesneye dayalı programlama ile gerçek dünyadaki anlamı arasında farklılık gösterebilmektedir. Örneğin miras kelimesini aileden gelen şeklinde düşünebilirsiniz ancak nesneye dayalı programlama ve Java' da tam olarak böyle değildir.

Örneğin öğrenci aynı zamanda bir insandır, insan da aynı zamanda bir canlıdır demeniz mirası ifade eder. Bunu bir ağaç yapısında aşağıdaki gibi örneklendirelim.



Burada dikkat ederseniz örneklerin bir üstü hep var olanı getirir. Bu ne demektir ? Aslında bunu sihirli bir kelimeyle çözebiliriz. Bu kelimemiz "**aynı zamanda**".

Aynı zamanda kelimesi model açısından bizim doğru tasarım yapmamızı sağlayacaktır. Örneğin Öğrenci bir üstü olan insan ile nasıl ilişkili ? Bunu hemen soralım öğrenci **aynı zamanda** insan mı ? Bunun cevabı evet ise tasarımımız doğru demektir.

Şimdi hatalı bir tasarım yapalım. Bir öğrenci bilgi sistemi için şube diye bir sınıfımız olsun ve bu sınıfı insan sınıfından miras alalım. Şimdi sorumuzu soralım: şube **aynı zamanda** insan mıdır ? Cevabımız tabiki hayır olacaktır. Peki bu nasıl bir hatadır ? Aslına bakarsanız teknik olarak mümkün ve ne çalışma zamanı nede derleme anında herhangi bir hata almayacağımız bu durum uygulama tasarımımızda ciddi bir model hatasıdır. Özellikle ileriki bölümlerde göreceğimiz polymorphism yani çok biçimlilik konusunda nesnelerin yanlış algılanması gibi hatalara neden olabilir.

Şimdi sorularımızı az önceki model ile örneklendirelim.

Sınıf	Anahtar Kelime	Üst Sınıf	Yanıt	Model Sonucu
Öğrenci	aynı zamanda	insan mdır ?	Evet	**Doğru**
Öğretmen	aynı zamanda	insan mıdır ?	Evet	**Doğru**
Öğretmen	aynı zamanda	canlı mıdır ?	Evet	**Doğru**
Kedi	aynı zamanda	hayvan mıdır ?	Evet	**Doğru**
Kedi	aynı zamanda	insan mıdır ?	Hayır	**Hatalı**
Kedi	aynı zamanda	canlı mıdır ?	Evet	**Doğru**
Bitki	aynı zamanda	insan mıdır ?	Evet	**Doğru**
Lise Öğrencisi	aynı zamanda	öğrenci midir ?	Evet	**Doğru**
Lise Öğrencisi	aynı zamanda	öğretmen midir ?	Hayır	**Hatalı**

Yukarıdaki örneklerde bazı durumlar doğru tasarlanmış ancak bazı durumlarda model hatası oluşmuştur. Örneğin “kedi aynı zamanda insan mıdır ?” sorusu dizaynımız içerisinde mantık hatası oluşturacaktır.

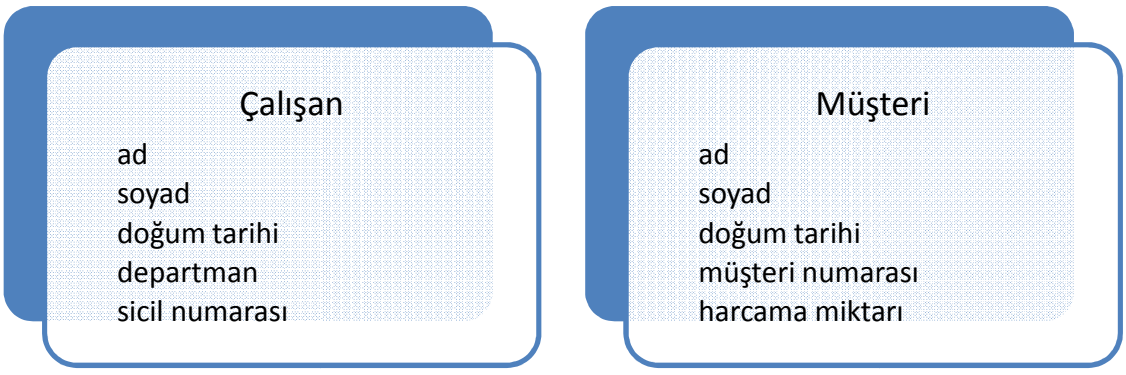
Bunun dışında dikkat etmemiz gereken bir başka durum ise “öğretmen aynı zamanda canlı mıdır ?” sorusunun evet yanıtı dönmesi. Öğretmen sınıfı direkt olarak canlı yı miras almıyor ancak miras aldığı insan sınıfı canlı sınıfını miras aldığı için öğretmen de aynı zamanda bir canlı olmakta yani canlı sınıfını miras almış sayılmaktadır.

Model Hatası ve Doğru Modelleme Örneği

Şimdi gerçek dünyaya uygun bir örnek yapalım buna göre bir müşteri ilişkileri yönetimi yazılımı hazırlıyoruz ve bu yazılım içerisinde müşteri ve çalışan isminde iki adet sınıflandırmamız bulunsun. Şimdi bunu iki şekilde çözümleyelim.

Hatalı Yöntem

Uygulamamda çalışan ve müşteri sınıflarını oluşturup ihtiyaç olan alanları içerisine ekliyorum

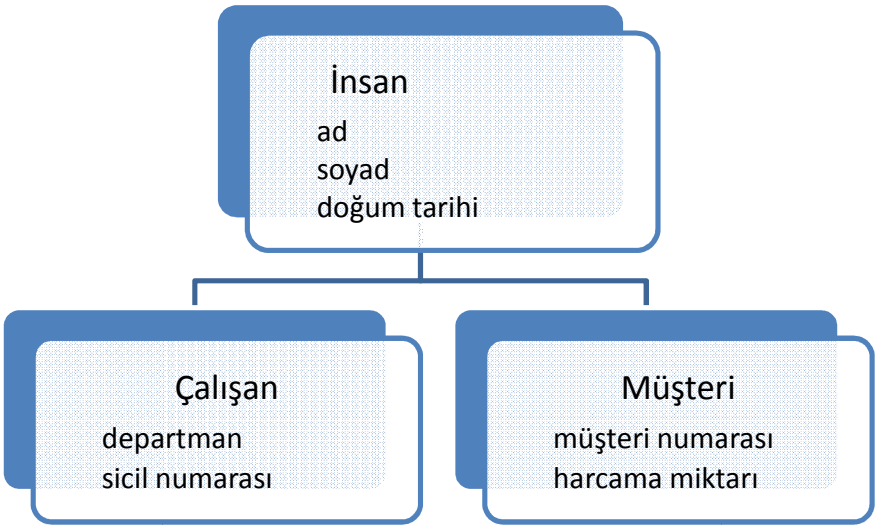


Bu modele göre çalışan ve sicil içerisinde bazı ortak bilgilerim var. Bu bilgiler ad, soyad ve doğum tarihi gibi insana özgü bilgiler olup aslında her iki sınıflandırmanında insan olmasından kaynaklanan bir durum.

Yani aslında her insanda olan özellikler. Bu durumda modellememizi insan etrafında yapmak çok daha doğru olacaktır.

Doğru Yöntem

Az önceki örneğin doğru modeli aslında aşağıdaki gibi olmalıdır.



Buna göre sınıflandırmada ortak özelliği insan olarak modelleyip insana ait bilgileri bu insan sınıfı içerisine ekledik. Bu doğru olandı ve bize bir çok avantaj sağladı.

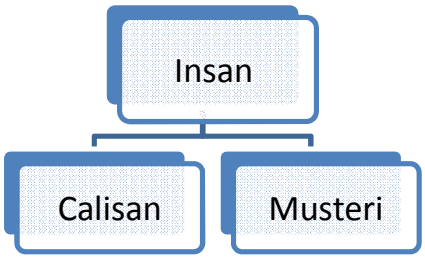
Örnek vermek gerekirse uygulama içerisinde TC kimlik bilgisi istenecek olsaydı bunu sadece insan sınıfına eklemem yeterli olacaktı aksi durumda hem çalışan hemde müşteri sınıflarına ayrı ayrı eklememiz gerekirdi.

Başka bir avantaj durumu ise ileride göreceğimiz polymorphism yani çok biçimlilik. Eğer çalışan ve müşteriye insan olarak bakmak isteseydik hatalı modelde bu mümkün olmayacaktı. Şimdi ise hem müşteri hemde çalışan benim için birer insandır ve ben bunlara bu açıdan bakıp işlem yapma şansına sahibim.

Sub-class ve Super-class

Miras ile gelen iki kelime vardır. Bunlar sub-class yani alt sınıf ve super-class yani üst sınıf. Sub-class bir sınıf için türeyen sınıflar ifade eder. Super-class ise türenen yani üst sınıfı belirtir.

Örneklendirmek gerekirse aşağıdaki hiyerarşide Insan' ın sub-class ları Calisan ve Musteri' dir. Calisan' in super-class' ı ise Insan' dır.



extends Kelimesi ve Miras Uygulama

Miras yani inheritance kod içerisinde uygulanmak istenirse **extends** kelimesi kullanılmalıdır. Az önceki örneği uygulamaya döküp aşağıdaki gibi örneklendirebiliriz.

Öncelikle Insan sınıfını oluşturuyoruz.

```
public class Insan {
	String ad;
	String soyad;
	Integer dogumTarihi;

	public Integer yasBilgisiGetir(){
		Integer simdikiTarih = Calendar.getInstance().get(Calendar.YEAR);
		return simdikiTarih - dogumTarihi;
	}
}
```

Şimdi Insan sınıfını miras alan Calisan ve Musteri sınıflarını oluşturuyoruz.

```
public class Calisan extends Insan{
	String departman;
	Integer sicilNumarasi;
}
```

```
public class Musteri extends Insan{
	Integer musteriNumarasi;
	Double harcamaMiktari;
}
```

Bu durumda miraz özelliğini kullanmış olduk. Peki bu ne işimize yarayacak ? Şimdi aşağıdaki örnekte Bir çalışan oluşturup bunun özelliklerini kullanalım.

```
public class Ornek {

	public static void main(String[] args) {
		Calisan cls = new Calisan();
		cls.ad = "Melih"; //insan olduğu için
		cls.soyad = "Sakarya"; //insan olduğu için
		cls.dogumTarihi = 1981; //insan olduğu için
		cls.sicilNumarasi = 1234; //calışan olduğu için
```

```
            cls.departman = "Bilgi İşlem"; //çalışan olduğu için
            int yas = cls.yasBilgisiGetir(); //çalışan olduğu için

      }
}
```

Bu örneğe göre Calisan sınıfı tipinde bir cls nesnesi oluşturduk ve özelliklerinin atamasını yaptık. Örnekteki atamalarda ad, soyad, doğum tarihi gibi özellikler Calisan sınıfı içerisinde yer almamasına karşın Calisan tarafından miras alınan Insan sınıfı içisinde yer aldığından Calisan icerisinde varmış gibi davrandırabiliyoruz. Aynı zamanda yasBilgisiGetir gibi metodların da kullanımı sağlanabiliyor. Bu durum Musteri sınıfı içinde geçerli olacaktır. Bu kullanım tarzı bize Inheritance yani Miras sayesinde sağlanan bir özelliktir.

Dikkat : **Java' da sadece tek bir sınıfın miras alınmasına izin verilebilir.** Ancak üst üste sınıflar bir birlerinini miras alabilirler. Örneğin Yazilimci-Calisan-Insan seklinde bir hiyerarşiye sahip olabilirler.

Metod Ezme (Method Override)

Eğer miras aldığınız sınıftaki metod sizin için artık geçerliliğini yitirmişse bu metodu tekrar yazabilirsiniz. Buna metod override yeni metod ezme denir. Kural olarak aynı isimde ve tipte bir sınıf yazdığınızda üst sınıfı artık ezmiş olmanız anlamına gelir.

Bunu aşağıdaki gibi örneklendirelim. Öncelikle Insan ve onu miras alan Calisan isminde iki adet sınıf yaratıyoruz. Bu sınıflar içerisinde bilgileriYazdir metodları bulunmakta ve Calisan sınıfındaki metod Insan' daki bilgileriYazdir metodunu ezmektedir.

```
public class Insan {
      String ad;
      String soyad;
      Integer dogumTarihi;

      public void bilgileriYazdir() {
            System.out.println("*** Insan Bilgileri ***");
            System.out.println("Ad : " + ad);
            System.out.println("Soyad : " + soyad);
            System.out.println("Doğum Tarihi : " + dogumTarihi);
      }
}
```

```
public class Calisan extends Insan{
      String departman;
      Integer sicilNumarasi;

      @Override
      public void bilgileriYazdir(){
            System.out.println("*** Calisan Bilgileri ***");
            System.out.println("Ad : " + ad);
            System.out.println("Soyad : " + soyad);
```

```
            System.out.println("Doğum Tarihi : " + dogumTarihi);
            System.out.println("Departman : " + departman);
            System.out.println("Sicil Numarası : " + sicilNumarasi);
      }
}
```

Şimdi bunun aşağıdaki gibi kullanımını örneklendirelim.

```
public class Ornek {

      public static void main(String[] args) {
            Calisan cls = new Calisan();
            cls.ad = "Melih";
            cls.soyad = "Sakarya";
            cls.dogumTarihi = 1981;
            cls.sicilNumarasi = 1234;
            cls.departman = "Bilgi İşlem";

            cls.bilgileriYazdir();
      }
}
```

Buna göre çıktımız aşağıdaki gibi olacaktır.

```
*** Calisan Bilgileri ***
Ad : Melih
Soyad : Sakarya
Doğum Tarihi : 1981
Departman : Bilgi İşlem
Sicil Numarası : 1234
```

Yani sadece Calisan sınıfı içerisindeki bilgileriYazdir metodu çalışacaktır. Bir üstteki yani Insan sınıfındaki metodumuz ise override edilmiş olacaktır.

Not : **@Override** annotation' ı ile ezilen metodlar kontrol edilebilir. Bu sayede üst sınıflarda olabilecek olası değişikliklere karşı bir kontrol metkanizması oluşur. Örneğin üst sınıftaki metod parametreleri değişirse ezilme işlemi yapılan sınıfta derleme anında aşağıdaki gibi bir hata oluşup override işleminin hatalı olduğu belirtilecektir.

The method bilgileriYazdir() of type Calisan must override or implement a supertype method

Miras ile Constructor çalışma şekli

Eğer miras kullandığınız sınıflarda constructor yani kurucu metod kullanımı varsa constructor' lar en üstten yani süper den itibaren çalışmaya başlar.

Bunu aşağıdaki gibi örneklendirelim

```
public class Insan {
	public Insan() {
		System.out.println("Insan() constructor...");
	}

}
```

```
public class Calisan extends Insan {
	public Calisan() {
		System.out.println("Calisan() constructor...");
	}

}
```

Şimdi aşağıdaki gibi bir Calisan nesnesi yaratalım.

```
public class Ornek {

	public static void main(String[] args) {
		new Calisan();
	}
}
```

Buna göre çıktımız aşağıdaki gibi olacaktır. Yani önce Insan sonra Calisan constructor çalışacaktır.

```
Insan() constructor...
Calisan() constructor...
```

Not : Eğer birden fazla miras olsaydı az önce bahettiğimiz gibi çalışma daha üstten başlıyor olacaktır.

super kelimesi

Sub-class ve super-class kelimesinde gördüğümüz super kelimesinin üst sınıfı ifade ettiğini belirtmiştik. Aslında **super** kelimesi Java içerisinde ayrı bir keyword olarak ta yer almaktadır. super kelimesinin kullanım amacı miras alınan sınıftaki özelliklere ulaşmaktır.

super kelimesini iki amaçla kullanabiliriz. Birincisi üst sınıftaki ezilen metodlara ulaşmak olabilir. Bunu örneklendirecek olursak az önce yaptığımız override örneğinde bilgileriYazdir metodunda ad soyad yazdırma işlemlerini alt ve üst sınıflarda iki defa yazmıştık. Yani aynı işlemi iki defa gerçekleştirdir. Bu yazılım geliştirme içeriside re-usability yani tekrar kullanılabilirlik kavramına aykırıdır. Bunun yerine aşağıdaki kod bloğundakşi gibi Calisan içerisindeki bilgileriYazdir metodunda ezdiğimiz metodun eski halinide kullanma şansına sahip olabilirdik.

```
public class Insan {
	String ad;
	String soyad;
	Integer dogumTarihi;
```

```
	public void bilgileriYazdir() {
		System.out.println("*** Insan Bilgileri ***");
		System.out.println("Ad : " + ad);
		System.out.println("Soyad : " + soyad);
		System.out.println("Doğum Tarihi : " + dogumTarihi);
	}
}
```

```
public class Calisan extends Insan{
	String departman;
	Integer sicilNumarasi;

	@Override
	public void bilgileriYazdir(){
		System.out.println("*** Calisan Bilgileri ***");
		super.bilgileriYazdir();
		System.out.println("Departman : " + departman);
		System.out.println("Sicil Numarası : " + sicilNumarasi);
	}
}
```

Buna göre ad soyad bilgilerini tekrar tekrar yazdırmak yerine ezdiğimiz metodun eski halini `super.bilgileriYazdir();` şeklinde kullanarak işlem tekrarını engellemiş olduk. Sonuç olarak çıktımız aşağıdaki gibi olacaktır. Bu durumda her iki metodta çalıştırılmış olacaktır.

```
*** Calisan Bilgileri ***
*** Insan Bilgileri ***
Ad : Melih
Soyad : Sakarya
Doğum Tarihi : 1981
Departman : Bilgi İşlem
Sicil Numarası : 1234
```

super kelimesinin ikinci kullanımı ise constructor yani kurucu metod çağırımlarıdır. Daha önceki örneklerde bulunan constructor çalışma prensiplerinde gördüğümüz üzere çağırımlar en üst yani süper den başlayarak aşağıya doğru gitmektedir. Peki şimdi aşağıdaki gibi bir örnekte durum nasıl olacaktır ?

```
public class Insan {
	public Insan() {
		System.out.println("Insan() constructor...");
	}

	public Insan(String ad) {
		System.out.println("Insan(String ad) constructor...");
	}
}
```

```
public class Calisan extends Insan {
	public Calisan() {
```

```
            System.out.println("Calisan() constructor...");
      }

      public Calisan(String ad) {
            System.out.println("Calisan(String ad) constructor...");
      }
}
```

Şimdi aşağıdaki örnekteki gibi bir Calisan nesnesi oluşturalım ve durumu görelim ?

```
public class Ornek {

      public static void main(String[] args) {
            new Calisan("Melih");
      }
}
```

Bu kod örneğini çalıştırdığımda çıktım aşağıdaki gibi oldu.

```
Insan() constructor...
Calisan(String ad) constructor...
```

Burada genelde yanılgıyı oluşturan durum üst sınıfın parametresiz metodunun çağırılmasıdır. Aslında ben ad bilgini göndermeme rağmen Insan parametresiz, Calisan ise ad parametreli constructor metodunu çağırmış oldu. Peki bu durumda Insan içerisindeki parametreli metodu nasıl çağırabilirim ? Bunu yine super kelimesini kullanarak aşağıdaki gibi gerçekleştirebiliriz.

```
public class Calisan extends Insan {
      public Calisan() {
            System.out.println("Calisan() constructor...");
      }

      public Calisan(String ad) {
            super(ad);
            System.out.println("Calisan(String ad) constructor...");
      }
}
```

Bu kod örneğine göreyse çıktımız aşağıdaki gibi yani her iki çağırımda parametreli olacak şekilde gerçekleştirilecektir.

```
Insan(String ad) constructor...
Calisan(String ad) constructor...
```

Not : Constructor çağırımları aynı **this** örneğindeki gibi bir metod içerisinde ilk satırda olmak zorundadır.

Bir başka örnekte ise aşağıdaki gibi bir çağırım olursa derleme anında hata ile karşılaşılacaktır.

```
public class Insan {
	public Insan(String ad) {
		System.out.println("Insan(String ad) constructor...");
	}
}
```

```
public class Calisan extends Insan {
	public Calisan() {
		System.out.println("Calisan() constructor...");
	}
}
```

Bu hatanın sebebi Calisan nesnesi oluşturulurken Insan sınıfında constructor parametresi beklenmesidir. Bu durumda Calisan sınıfının aşağıdaki gibi olması beklenmektedir.

```
public class Calisan extends Insan {
	public Calisan() {
		super("Ad parametresi");
		System.out.println("Calisan() constructor...");
	}
}
```

Miras ve Metod Override engelleme

Bir sınıfın miras alınmasını istemiyorsak final yapmalıyız. Bu durumda derleme anında hata alınacaktır. Örneğin aşağıdaki sınıf miras alınamaz.

```
public final class Insan {

}
```

Bir metodun override edilmesini yani ezilmesini istemiyorsak yine final yapabiliriz. Örneğin aşağıdaki sınıf içerisindeki bilgileriYazdir metodu miras alınan sınıflarda ezilemeyecektir.

```
public class Insan {

	public final void bilgileriYazdir() {
		System.out.println("*** Insan Bilgileri ***");
	}

}
```

Object Sınıfı

Java' primitive yani ilkel tip dışındaki nesnelerin ortak özelliklerinden biri de tamamının Object tipinde olması yani Object sınıfından türemeleridir. Peki bu durum nasıl oluşuyor ? Eğer Java' da herhangi bir sınıfı miras almıyorsanız gizli bir miras durumu vardır. Yani siz desenizde demeseniz de extends Object tanımı sizin için otomatik olarak ekleniyor.

Bu duruma göre aşağıdaki iki sınıf tanımı arasında bir fark bulunmamaktadır.

```
public class Insan {

}
```

```
public class Insan extends Object {

}
```

Eğer bu sınıflardan bir nesne üretecek olursak her ikisinde de Object sınıfına ait aşağıdaki metodlar kullanılıyor olacaktır.

```
public class Ornek {

	public static void main(String[] args) {
		Insan i1 = new Insan();
		System.out.println(i1.toString());
		System.out.println(i1.hashCode());

	}
}
```

Object sınıfı için yapılan gizli mirasın nedenlerinden biri polymorphism kullanımıdır. Bunun birlikte bizim için kolaylaştırılmış bazı metodlar içermektedir. Bu metodları aşağıdaki gibi listeleyebiliriz.

equals	Nesne için karşılaştırma yapmak için kullanılır.
hashCode	Nesnenin üretilen hashCode değerini döndürür.
toString	Nesnenin yazdırma anında veya bilgi almak için String şeklinde değerini döndürür.
getClass	Nesnenin asıl sınıf tipini döndürür.
notify	Bekletilen Thread in tekrar çalışmasını sağlar.
notifyAll	Bekletilen tüm Thread leri tekrardan aktif hale getirir.
wait	Thread leri bekletmeyi sağlar.
clone	Nesnenin bellekte bir kopyasının oluşmasını sağlar.
finalize	Nesne garbage collecter tarafından temizlenmeden hemen önce çalıştırılan metodtur.

Bu metodlar içerisinde özellikle equals, hashCode, toString metodlarının sınıflar içerisinde override edilmesi yani yeniden yazılması beklenir.