

# Nesneye Y6nelik Yazılım M6uhendislięi (376)

---

*Dr. 6ęr. 6yesi Ahmet Arif AYDIN*

# *Functional Decomposition (işlevsel ayrıştırma)*

---

- ❖ Karmaşık bir problemi işlevsel ve çözülebilir alt aşamalara ayrıştırıp çözme işlemidir (*process of taking a complex process and breaking it down into its smaller, simpler parts*)

## Kütüphane Otomasyonu

1. *Kitap bilgisini al*
2. *Veritabanında mevcut değilse kayıt oluştur.*
3. *Mevcut olan kitapları listele*
4. *Emanet alınan kitap listesini yazdır*
5. *Sistemden çık*

# *Functional Decomposition: Problemler*

---

## 1. Bütün detaylar ana program tarafından bilinmesi gerekir.

- ❖ ana program temelli tasarım yapılır

## 2. Değişim isteklerine uygun değildir

- ❖ İyi bir modüler bir yapı bulunmadığından küçük değişiklikler programın tamamına etkisi olabilir
- ❖ Hatalar kod değişikliklerinden kaynaklanıyor (Many bugs originate with changes to the code)

# *Object-Oriented Design (Nesne Tabanlı Tasarım)*

---

❖ **Functional decomposition ile ortaya çıkan problemler nesne tabanlı programlamanın**

- **abstraction** (*soyutlama*)
- **encapsulation** (*kapsülleme*)
- **information hiding** (*bilgi gizleme*)
- **polymorphism** (*çok biçimlilik*)
- **modularity** (*modülerlik*)

**kavramlarının etkin olarak kullanılması ile ortadan kaldırılır!**

# *Object-Oriented Design: (Nesne tabanlı tasarım)*

- ❖ Karmaşık bir problemi bir birinden bağımsız nesleri oluşturarak çözme işlemidir
- ❖ Her bir nesne gerçekleştireceği işlemi gerçekleştirmekten sorumludur.
- ❖ Nesneler özel bir sorumluluğu gerçekleştirmek için tasarlanır.

Kütüphane Otomasyonu



Arama (Search)

Ekleme (Insert)

Güncelleme (Update)

Raporlar (Reports)

Ödünç Kitap

Admin (Yönetim)

Veritabanı

# Object-Oriented Design: Functional Decomposition

---

- ❖ **abstraction** (*soyutlama*), **encapsulation** (*kapsülleme*), **information hiding** (*bilgi gizleme*), **polymorphism** (çok biçimlilik), **modularity** (*modülerlik*)

Kavramları etkin biçimde kullanıldığında yazılan kod

- ❖ **highly cohesive and loosely coupled** olacaktır!
- ❖ Weak cohesion (bir çok işlemi gerçekleştiriyor: sırala görüntüle, db ekle, sil , güncelle,.....)
- ❖ Tight coupling (bir çok bağlantısı var. Bir parçada yapılan değişiklik bütün programı etkiliyor)

# *Object-Oriented Paradigm: Abstraction nedir?*

---

- ❖ Abstraction(soyutlama) aşağıdaki biçimlerde tanımlanabilir:
  - ❖ Kompleks bir yapının detaylarının gizlenerek basit bir arayüzle kullanıcıya sunulmasıdır.
  - ❖ Bir varlığın belirli amaçları gerçekleştirmek, bir görevi veya problemi çözmek için sağlamış olduğu tanımlamadır (set of concepts that some entity provides you in order for you to achieve a task or solve a problem )

# *Object-Oriented Paradigm: Abstraction Örnekleri*

---

- ➔ Assembly dili makine dili için bir *abstraction* dır.
- ➔ Yüksek seviyeli programlama dilleri assembly dili için bir *abstraction* dır.
- ➔ Bir sınıf içerisinde tanımlanan **değişkenler** hafıza hücreleri için bir soyutlamadır.
- ➔ Bir **sınıfa** erişmek için oluşturulan **nesneler**
- ➔ Bir amaç için oluşturulan bir sınıf (class, method, fonksiyon )
- ➔ Bir fonksiyona erişmek için kullanılan fonksiyon çağrısı



# Object-Oriented Paradigm: Abstraction

---

```
import java.util.List;
import java.util.LinkedList;

public class Student{
    private String isim;
    public int yaş;

    public Student(String isim){
        this.isim = isim;
    }

    public void çıktıal() {
        System.out.print(isim + " says \"");
    }

    public String formatlıcıktı() {
        return String.format("%14s : %s",
                                getClass().getName(),
                                name);
    }
}
```

```
public static void main(String[] args) {
    Student öğrenci1= new Student("iu");
    öğrenci1.cıktıal
}
```

# *Access Specifiers (Bağlantı Tanımlayıcıları)*

---

## ❖ **public**

- ➔ Bir sınıfa erişim sağlayabilen nesnelerin kullanımına açık

## ❖ **private**

- ➔ sadece tanımlanan sınıfın içindeki methodlar erişebilir.
- ➔ dışarıya kapalı

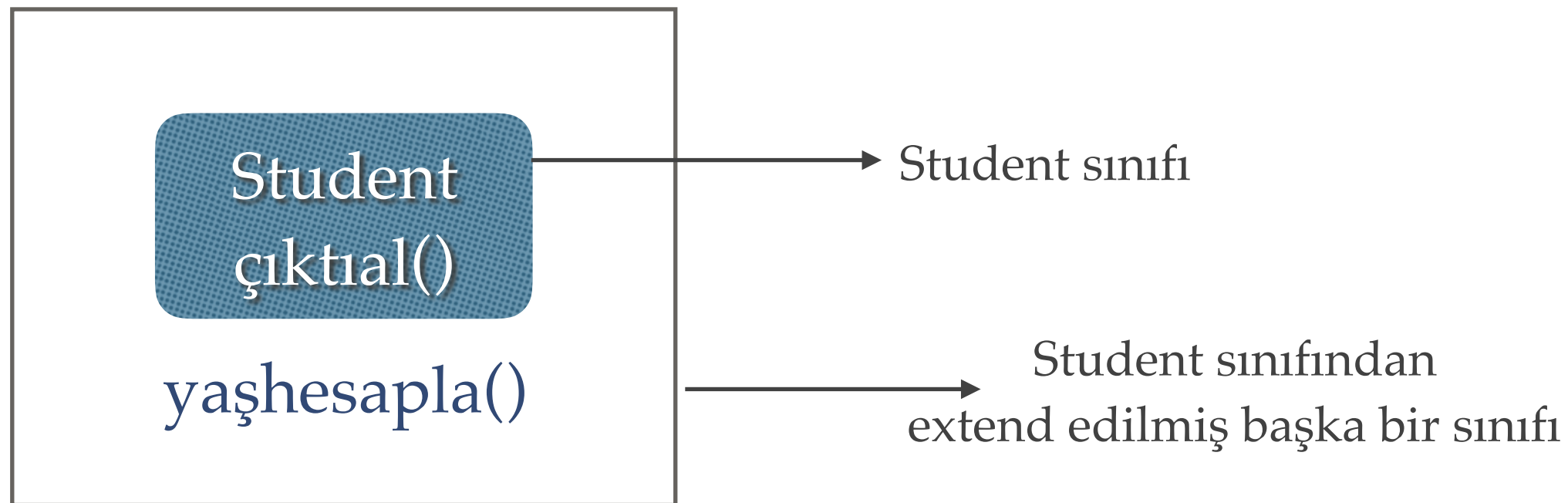
## ❖ **protected**

- ➔ private gibi davranır
- ➔ yalnız kalıtım yoluyla erişilebilir

# Object-Oriented Paradigm: Inheritance (kalıtım)

---

- ❖ Bir sınıfın başka bir sınıftan özelliklerini ve metodlarını kalıtsal olarak devralmasıdır (*process of acquiring properties of another class*)
- ❖ **extends** kelimesi kullanılır.
- ❖ Kalıtım yoluyla devralınan özelliklere yenileri eklenip oluşturulan yeni sınıf genişletilebilir.



# Object-Oriented Paradigm: Inheritance (kalıtım)

---

```
class Hesaplama {
    int z;
    public void toplama(int x, int y) {
        z = x + y;
        System.out.println("Toplam:"+z);
    }
    public void cikarma(int x, int y) {
        z = x - y;
        System.out.println("Fark:"+z);
    }
}

public class BenimHesap extends Hesaplama
{
    public void carpma(int x, int y) {
        z = x * y;
        System.out.println("Çarpım:"+z);
    }
}
```

```
public static void main(String args[]) {
    int a = 20, b = 10;
    BenimHesap demo = new BenimHesap();
    demo.toplama(a, b);
    demo.cikarma(a, b);
    demo.carpma(a, b);
}
```

# Object-Oriented Paradigm: Inheritance (kalıtım)

---

```
import java.util.List;
import java.util.LinkedList;

public class Student{
    private String isim;
    public int yaş;

    public Student(String isim){
        this.isim = isim;
    }

    public void çıktıal() {
        System.out.print(isim + "
says \"");
    }

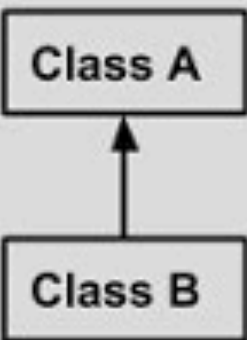
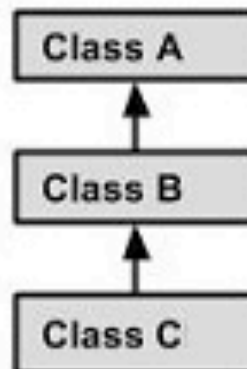
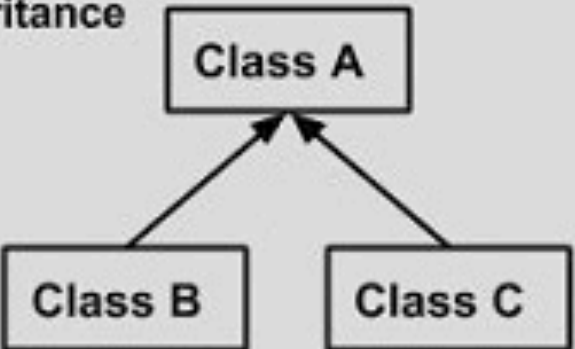
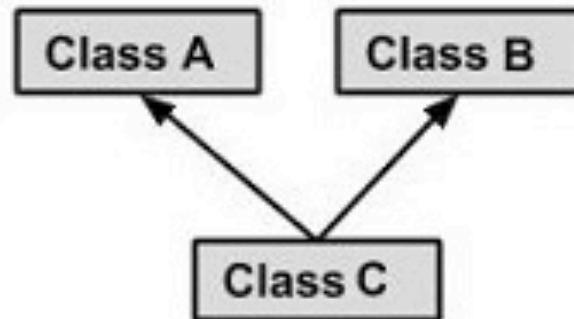
    public String formatlıcıktı() {
        return String.format(
            "%14s : %s",
            getClass().getName(), name);
    }
}
```

```
public class MastersStudent extends Student {

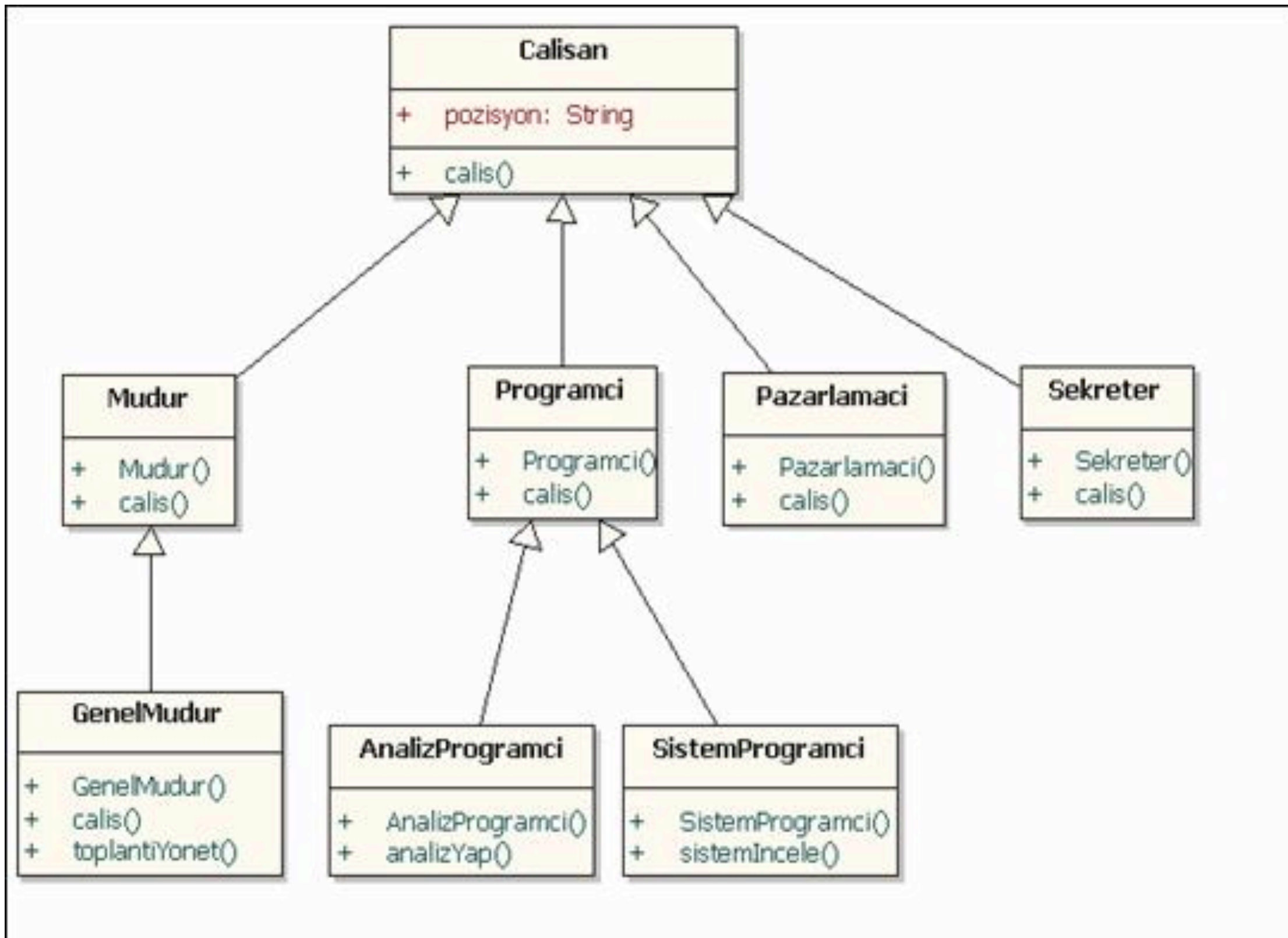
    public MastersStudent(String isim) {
        super(isim);
    }

    public void çıktıal() {
        super.çıktıal();
        System.out.println("*****");
    }
}
```

# Object-Oriented Paradigm: Inheritance (kalıtım)

Single Inheritance	 <pre>graph BT; B[Class B] --&gt; A[Class A]</pre>	<pre>public class A {     ..... } public class B extends A {     ..... }</pre>
Multi Level Inheritance	 <pre>graph BT; C[Class C] --&gt; B[Class B]; B --&gt; A[Class A]</pre>	<pre>public class A { .....} public class B extends A {.....} public class C extends B {.....}</pre>
Hierarchical Inheritance	 <pre>graph BT; B[Class B] --&gt; A[Class A]; C[Class C] --&gt; A</pre>	<pre>public class A { .....} public class B extends A {.....} public class C extends A {.....}</pre>
Multiple Inheritance	 <pre>graph BT; C[Class C] --&gt; A[Class A]; C --&gt; B[Class B]</pre>	<pre>public class A { .....} public class B {.....} public class C extends A,B {     ..... } // Java does not support mutiple Inheritance</pre>

## Object-Oriented Paradigm: Inheritance (kalıtım)



# *Object-Oriented Paradigm: Information Hiding*

---

- ❖ Bir metodun veya nesnenin detaylarının gizlenmesi işlemidir
  - ❖ *The process of hiding the details of an object or function*
  - ❖ *Mechanism for restricting access to some of the object's components.*



# Object-Oriented Paradigm: Information Hiding

---

- ❖ Bir metodun veya nesnenin detaylarının gizlenmesi işlemidir
  - ❖ *The process of hiding the details of an object or function*
  - ❖ *Mechanism for restricting access to some of the object's components.*

```
abstract public class çalışan {  
    abstract public void maaşhesapla();  
}
```

```
public class yönetici extends çalışan {  
    private int maaş;  
    public int katsayı;  
  
    private void katsayı(){  
        this.katsayı=25;  
    }  
    public int katsayıgönder(){  
        katsayı();  
        return katsayı;  
    }  
}
```

# *Object-Oriented Paradigm: Encapsulation*

---

- ❖ **Kapsülleme tasarım detaylarını gizlemek için kullanılan teknik veya programlama dili seviyesindeki mekanizmalardır**
  - ❖ *a set of language-level mechanisms or design techniques that hide implementation details of a class, module, or subsystem from other classes, modules, and subsystems*
  - ❖ a mechanism of wrapping the data (variables) and code acting on the data (methods) together as a single unit.
- ❖ Kapsüllemeyi gerçekleştirmek için *information hiding* de kullanılır

# *Object-Oriented Paradigm: Encapsulation*

---

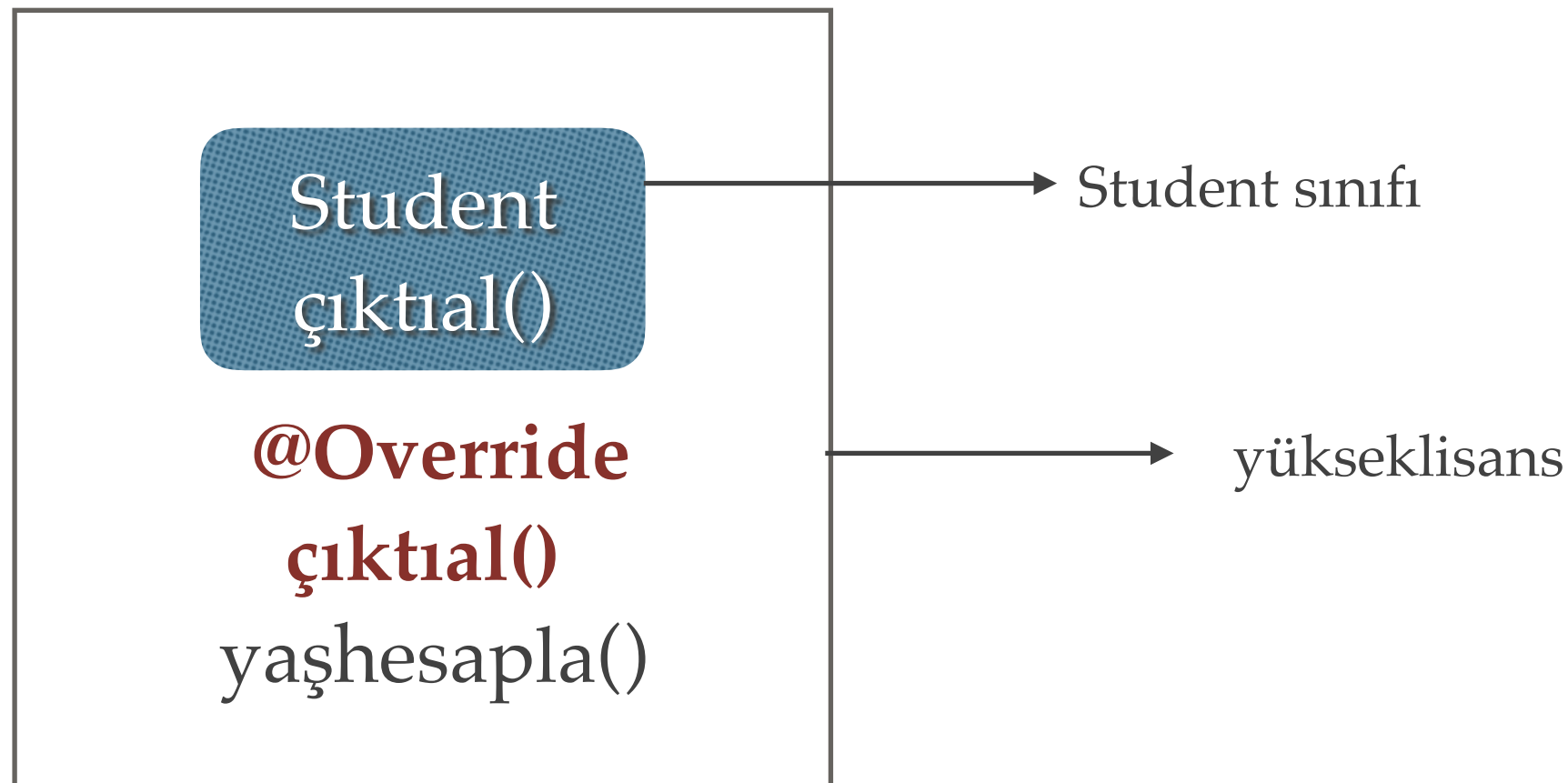
```
public class Student{
    private String name;

    public String getName(){
        return name;
    }
    public void setName(String name){
        this.name=name
    }
    public void yazdır(){
        .....
    }
}
```

```
class Test{
    public static void main(String[] args){
        Student s=new Student();
        s.setName("kemal");
        System.out.println(s.getName());
    }
}
```

# Object-Oriented Paradigm: Polymorphism

- ❖ Bir sınıfın başka bir sınıftan metodlarını kalıtsal olarak devralıp kendine özel bir biçimde tekrar yazma işlemidir.
- ❖ *ability of an object to take on many forms*
- ❖ *to allow an entity such as a **variable**, a **function**, or an **object** to have more than one form (<http://searchmicroservices.techtarget.com/definition/object>)*



# Object-Oriented Paradigm: Polymorphism

---

```
public interface çalışan {  
    public void rapor();  
    public void maaş();  
    public void katsayı();  
}
```

```
public class işçi implements çalışan {  
    @Override  
    public void rapor() {  
        System.out.println("isci rapor");  
    }  
    @Override  
    public void maaş() {  
        System.out.println("isci maaş");  
    }  
    @Override  
    public void katsayı() {  
        System.out.println("isci katsayı");  
    }  
}
```

```
public class kadroluişçi extends işçi {  
    @Override  
    public void maaş() {  
        System.out.println("kadrolu isci maaş");  
    }  
    @Override  
    public void katsayı() {  
        System.out.println("kadrolu isci katsayı");  
    }  
}
```