6 CLASS DESIGN

Objectives

► Define *inheritance*, *polymorphism*, *overloading*, *overriding*, and *virtual method invocation*,

►Use the access modifiers *protected* and “*package-friendly*”

►Describe the concept of constructor and method overloading,

►Describe the complete object construction and initialization operation

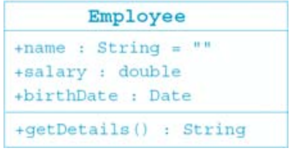
Objectives

►In a Java program, identify the following:

- ▪Overloaded methods and constructors
- ▪The use of ***this*** to call overloaded constructors
- ▪Overridden methods
- ▪Invocation of ***super*** class methods
- ▪Parent class constructors
- ▪Invocation of parent class constructors

Subclassing

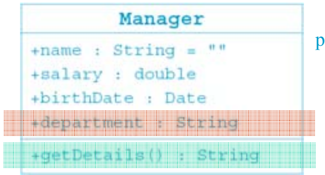
The `Employee` class



```
public class Employee {
      public String name=“”;
      public double salary;
      public Date birthDate ;
      public String getDetails(){...}
}
```

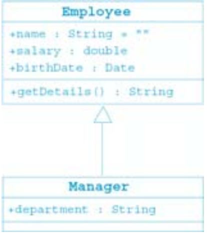
Subclassing

The `Manager` class



```
public class Manager {
      public String name=“”;
      public double salary;
      public Date birthDate ;
      public String department;
      public String getDetails(){...}
}
```

Subclassing



```
public class Employee {
      public String name=“”;
      public double salary;
      public Date birthDate ;
      public String getDetails(){...}
}
```

```
public class Manager extends Employee{
      public String department ;
}
```

Single Inheritance

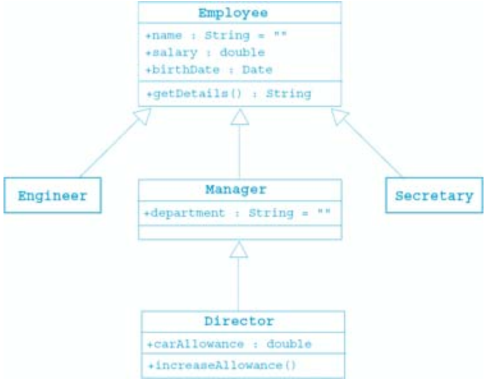
► When a class inherits from only one class, it is called *single inheritance*.

► Single inheritance makes code more reliable.

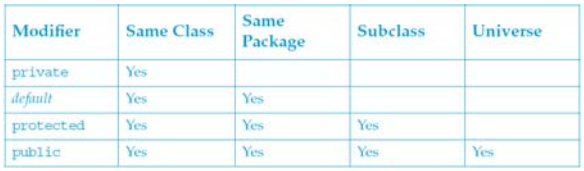
► *Interfaces* provide the benefits of multiple inheritance without drawbacks.

► Syntax of a Java Class:

```
<modifier> class <name> [extends <superclass>] {
        <declarations>*
}
```



Access Control



Modifier	Same Class	Same Package	Subclass	Universe
private	Yes			
default	Yes	Yes		
protected	Yes	Yes	Yes	
public	Yes	Yes	Yes	Yes

Overriding Methods

► A subclass can modify behavior inherited from a parent class.

► Subclass can create a method in a subclass with a different functionality than the parent's method but with the same

- Name
- Return type
- Argument list

The `super` Keyword

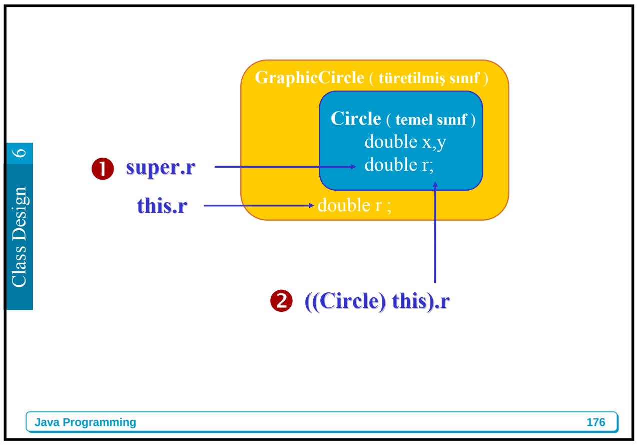
► `super` is used in a class to refer to its superclass.

► `super` is used to refer to the member variables of superclass.

► Superclass behavior is invoked as if the object was part of the superclass.

► Behavior invoked does not have to be in the superclass; it can be further up in the hierarchy.

```
public class GraphicCircle extends Circle {
   Color outline, fill;
   float r;  // New variable.  Resolution in dots-per-inch.
   public GraphicCircle(double x, double y, double rad, Color o, Color f){
      super(x, y, rad); outline = o; fill = f;
   }
   public void setResolution(float resolution) { r = resolution; }
   public void draw(DrawWindow dw) {
       dw.drawCircle(x, y, r, outline, fill); }
}
```



GraphicCircle (türetilmiş sınıf)

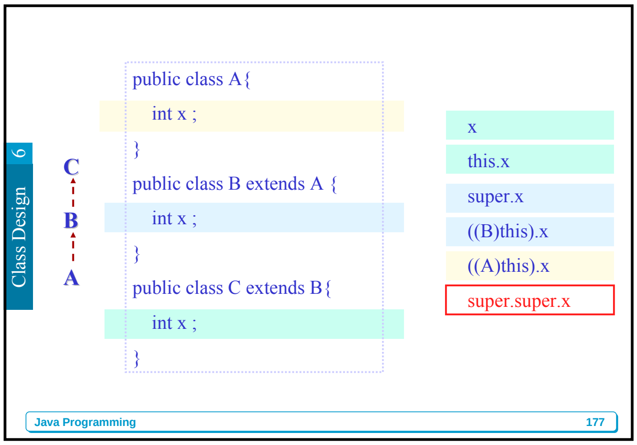
Circle (temel sınıf)
double x,y
double r;

double r ;

❶ super.r

this.r

❷ ((Circle) this).r



C
B
A

```
public class A{
    int x ;
}
public class B extends A {
    int x ;
}
public class C extends B{
    int x ;
}
```

x

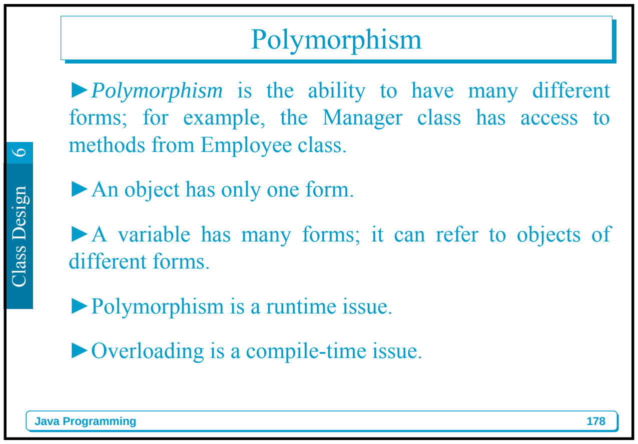
this.x

super.x

((B)this).x

((A)this).x

super.super.x



Polymorphism

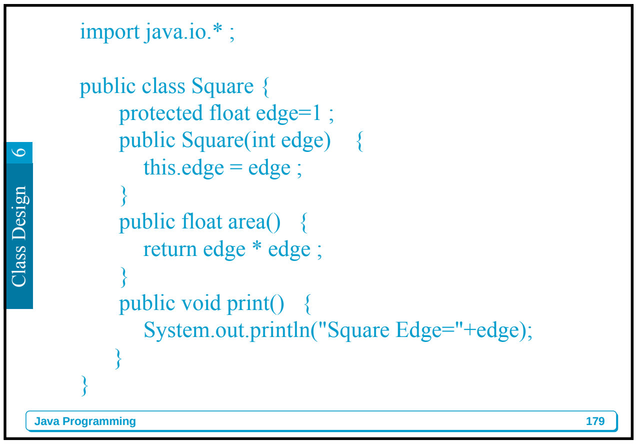
► *Polymorphism* is the ability to have many different forms; for example, the Manager class has access to methods from Employee class.

► An object has only one form.

► A variable has many forms; it can refer to objects of different forms.

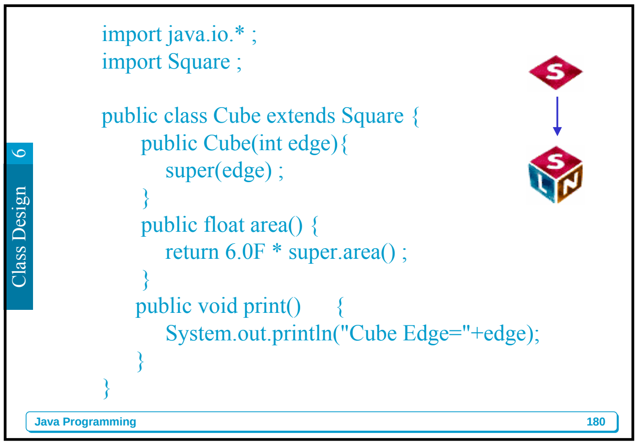
► Polymorphism is a runtime issue.

► Overloading is a compile-time issue.



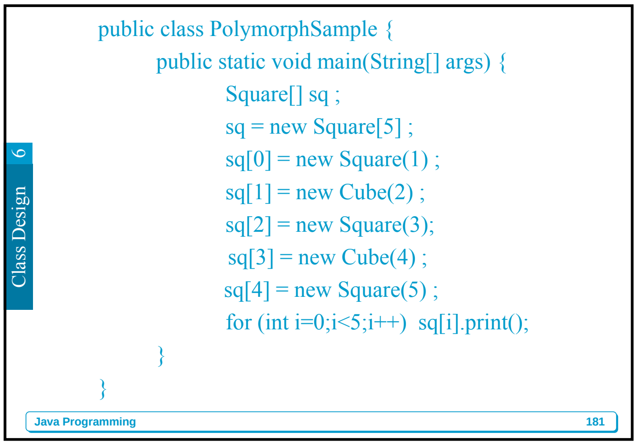
```
import java.io.* ;

public class Square {
      protected float edge=1 ;
      public Square(int edge)    {
          this.edge = edge ;
      }
      public float area()   {
          return edge * edge ;
      }
      public void print()   {
          System.out.println("Square Edge="+edge);
      }
}
```



```
import java.io.* ;
import Square ;

public class Cube extends Square {
      public Cube(int edge){
          super(edge) ;
      }
      public float area() {
          return 6.0F * super.area() ;
      }
     public void print()      {
          System.out.println("Cube Edge="+edge);
      }
}
```



```
public class PolymorphSample {
         public static void main(String[] args) {
                    Square[] sq ;
                    sq = new Square[5] ;
                    sq[0] = new Square(1) ;
                    sq[1] = new Cube(2) ;
                    sq[2] = new Square(3);
                     sq[3] = new Cube(4) ;
                    sq[4] = new Square(5) ;
                    for (int i=0;i<5;i++)  sq[i].print();
         }
}
```

```
public class A {
  public int i = 1;
  public int f() { return i;}
}
public class B extends A {
 public int i = 2;
 public int f(){ return -i;}
}
```

```
public class override_test {
    public static void main(String args[]) {
        B b = new B();
        System.out.println(b.i);
        System.out.println(b.f());
        A a = (A) b;
        System.out.println(a.i);
        System.out.println(a.f());
    }
}
```

polymorphism

Virtual Method Invocation

- ► Compile-time and run-time type

```
Square S = new Square(1.0) ;
Cube   C = new Cube(1.0) ;
S.area();
C.area();
```

compile-time type

- ► Virtual method invocation:

```
Square q = new Cube(1.0) ;
q.area();
```

run-time type

```
Employee e = new Manager() //legal
e.department = "Finance" //illegal


Employee [] staff = new Employee[1024];
staff[0] = new Manager();
staff[1] = new Employee();
```

Rules About Overridden Methods

- ►Must have a return type that is identical to the method it overrides
- ►Cannot be less accessible than the method it overrides
- ►Cannot throw more exceptions than the method it overrides

```
public class Parent {
     public void doSomething() {}
}
public class Child extends Parent {
     private void doSomething() {}
}
public class UseBoth {
     public void doOtherThing() {
          Parent p1 = new Parent();
          Parent p2 = new Child();
          p1. doSomething();
          p2. doSomething();
     }
}
```

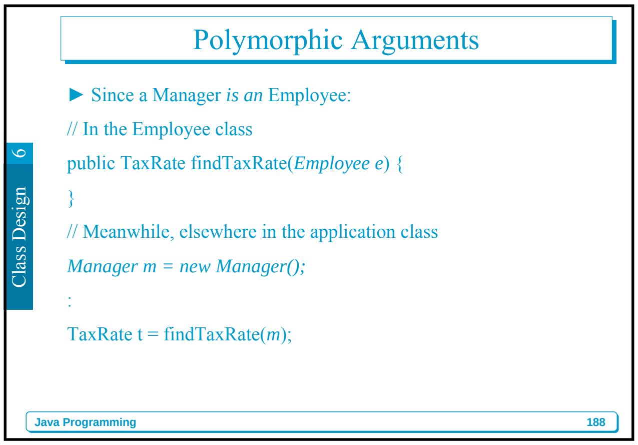
Heterogeneous Collections

- ►Collections with a common class are called *homogenous* collections.

```
 MyDate[] dates = new MyDate[2] ;
dates[0] = new MyDate(22,12,1964)  ;
dates[1] = new MyDate(22,7,1964)  ;
```

- ►Collections with dissimilar objects is a *heterogeneous* collection:

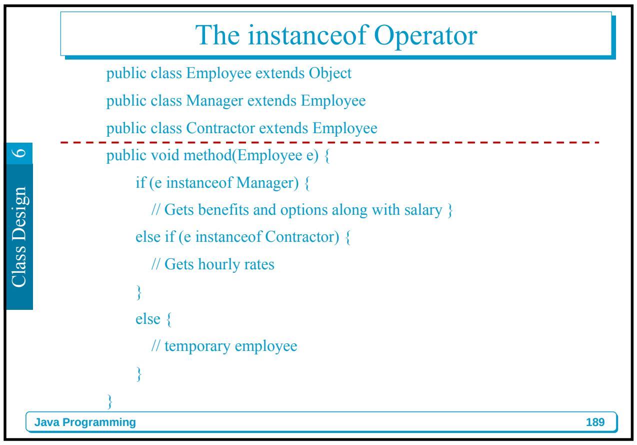
```
Employee[] staff = new Employee[1024] ;
staff[0] = new Manager() ;
staff[1] = new Employee() ;
staff[2] = new Engineer(),
```



Polymorphic Arguments

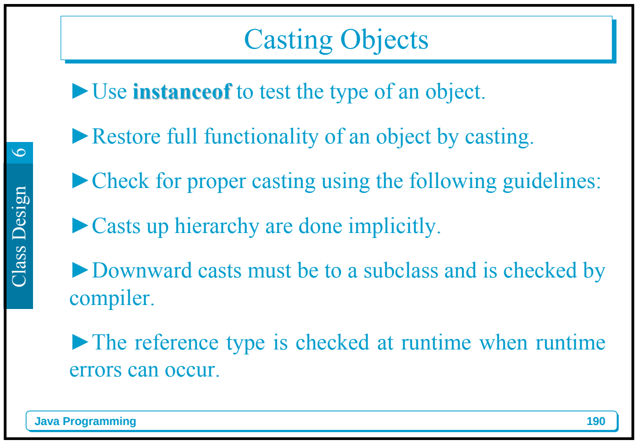
► Since a Manager *is an* Employee:

```
// In the Employee class
public TaxRate findTaxRate(Employee e) {
}
// Meanwhile, elsewhere in the application class
Manager m = new Manager();
:
TaxRate t = findTaxRate(m);
```



The instanceof Operator

```
public class Employee extends Object
public class Manager extends Employee
public class Contractor extends Employee
public void method(Employee e) {
      if (e instanceof Manager) {
         // Gets benefits and options along with salary }
      else if (e instanceof Contractor) {
         // Gets hourly rates
      }
      else {
         // temporary employee
      }
}
```



Casting Objects

►Use **instanceof** to test the type of an object.

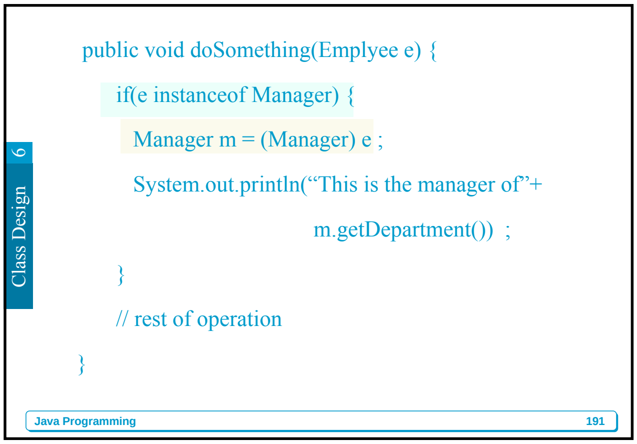
►Restore full functionality of an object by casting.

►Check for proper casting using the following guidelines:

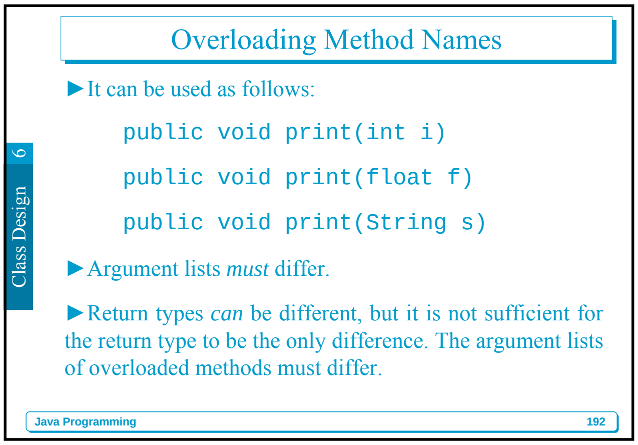
►Casts up hierarchy are done implicitly.

►Downward casts must be to a subclass and is checked by compiler.

►The reference type is checked at runtime when runtime errors can occur.



```
public void doSomething(Emplyee e) {
     if(e instanceof Manager) {
        Manager m = (Manager) e ;
        System.out.println("This is the manager of"+
                                   m.getDepartment())  ;
     }
     // rest of operation
}
```



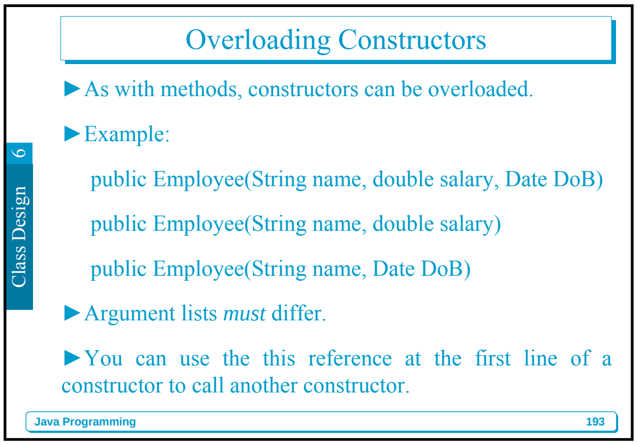
Overloading Method Names

►It can be used as follows:

```
public void print(int i)
public void print(float f)
public void print(String s)
```

►Argument lists *must* differ.

►Return types *can* be different, but it is not sufficient for the return type to be the only difference. The argument lists of overloaded methods must differ.



Overloading Constructors

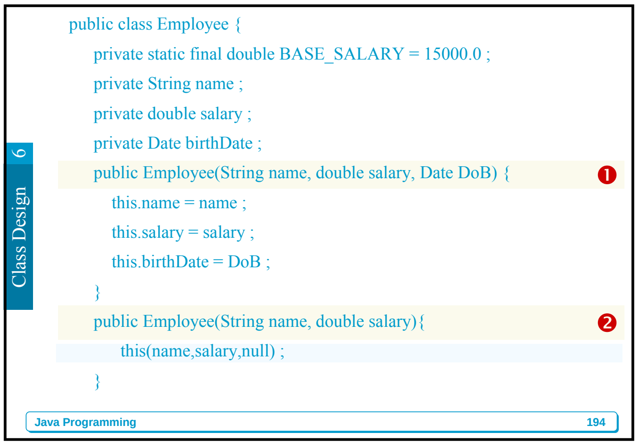
►As with methods, constructors can be overloaded.

►Example:

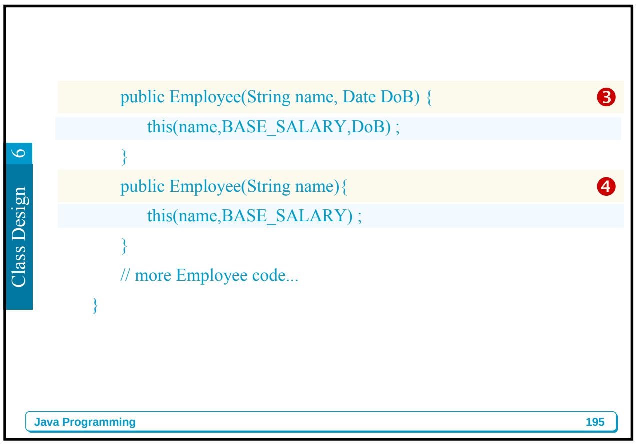
```
public Employee(String name, double salary, Date DoB)
public Employee(String name, double salary)
public Employee(String name, Date DoB)
```

►Argument lists *must* differ.

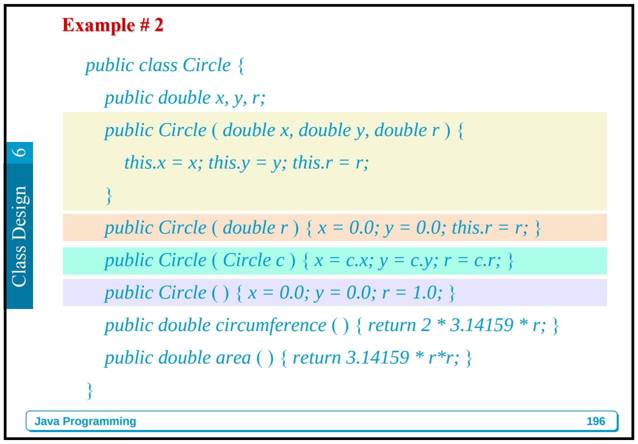
►You can use the this reference at the first line of a constructor to call another constructor.



```
public class Employee {
    private static final double BASE_SALARY = 15000.0 ;
    private String name ;
    private double salary ;
    private Date birthDate ;
    public Employee(String name, double salary, Date DoB) {      ❶
        this.name = name ;
        this.salary = salary ;
        this.birthDate = DoB ;
    }
    public Employee(String name, double salary){                 ❷
        this(name,salary,null) ;
    }
```

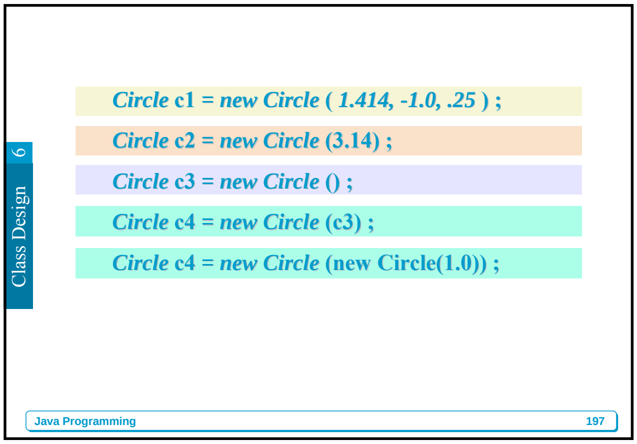


```
        public Employee(String name, Date DoB) {                 ❸
            this(name,BASE_SALARY,DoB) ;
        }
        public Employee(String name){                            ❹
            this(name,BASE_SALARY) ;
        }
        // more Employee code...
    }
```

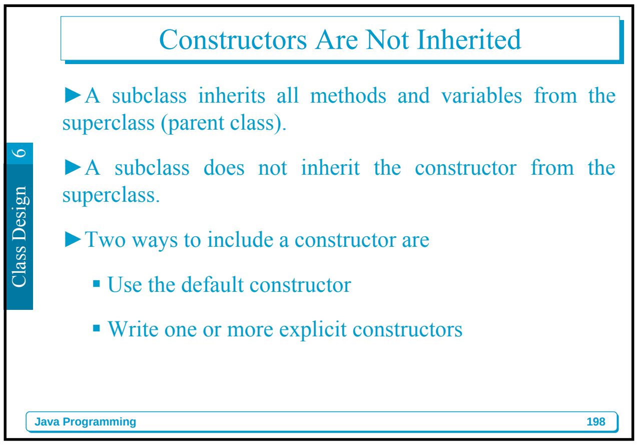


Example # 2

```
public class Circle {
    public double x, y, r;
    public Circle ( double x, double y, double r ) {
        this.x = x; this.y = y; this.r = r;
    }
    public Circle ( double r ) { x = 0.0; y = 0.0; this.r = r; }
    public Circle ( Circle c ) { x = c.x; y = c.y; r = c.r; }
    public Circle ( ) { x = 0.0; y = 0.0; r = 1.0; }
    public double circumference ( ) { return 2 * 3.14159 * r; }
    public double area ( ) { return 3.14159 * r*r; }
}
```



```
Circle c1 = new Circle ( 1.414, -1.0, .25 ) ;
Circle c2 = new Circle (3.14) ;
Circle c3 = new Circle () ;
Circle c4 = new Circle (c3) ;
Circle c4 = new Circle (new Circle(1.0)) ;
```



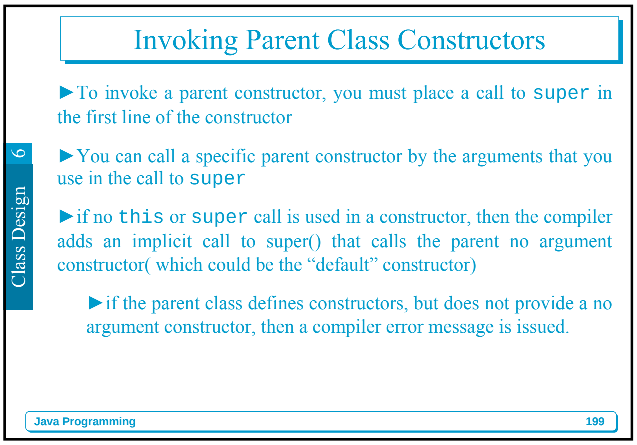
Constructors Are Not Inherited

►A subclass inherits all methods and variables from the superclass (parent class).

►A subclass does not inherit the constructor from the superclass.

►Two ways to include a constructor are

- Use the default constructor
- Write one or more explicit constructors



Invoking Parent Class Constructors

►To invoke a parent constructor, you must place a call to `super` in the first line of the constructor

►You can call a specific parent constructor by the arguments that you use in the call to `super`

►if no `this` or `super` call is used in a constructor, then the compiler adds an implicit call to super() that calls the parent no argument constructor(which could be the "default" constructor)

> ►if the parent class defines constructors, but does not provide a no argument constructor, then a compiler error message is issued.

```
public class Manager extends Employee {
    private String department ;
    public Manager(String name, double salary, String dept) {
        super(name,salary) ;
        department = dept ;
    }
    public Manager(String name, String dept){
        super(name) ;
        department = dept ;
    }
    public Manager(String dept) {
        department = dept ;                              super() ;
    }
}
```

Eğer türetilmiş sınıfta bir kurucu fonksiyon tanımlı değil ise derleyici bir tane yaratır. Yaratılan bu kurucu fonksiyon temel sınıfın kurucu fonksiyonunu çağırır :

```
class A {
    int i;
    public A() {
     i = 3;
   }
}
class B extends A {
    // Default constructor: public B() { super(); }
}
```

The Object Class

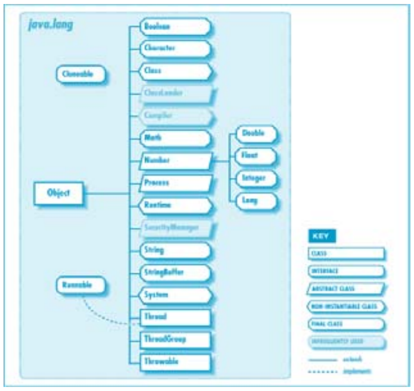
►The Object class is the root of all classes in Java

►A class declaration with no extends clause, implicitly uses “extends Object”

```
public class Employee {
   ...
}
```

is equivalent to:

```
public class Employee extends Object {
     ...
}
```



The class Class

Classes in Java source code are represented at run-time by instances of the `java.lang.Class` class. There's a Class object for every class you use; this Class object is responsible for producing instances for its class

Classes in the Java language have a run-time representation. There is a class named Class, instances of which contain run-time class definitions. If you're handed an object, you can find out what class it belongs to. In a C or C++ program, you may be handed a pointer to an object, but if you don't know what type of object it is, you have no way to find out. In the Java language, finding out based on the run-time type information is straightforward.

```
String myString = “Try!" ;
Class c = myString.getClass();
or
Class c = String.class;
```

```
String s = “Relations between IMF and Turkey";
Class strClass = s.getClass();
System.out.println( strClass.getName() );
// prints "java.lang.String"
String s2 = (String) strClass.newInstance();
```

```
try{
    Class c = Class.forName("java.lang.String") ;
    Object o ;
    String s ;

    o = (Object) c.newInstance();
    if(o instanceof String){
      s = (String) o ;
      System.out.println(s ) ;
    }
  }
  catch(Exception e){
    System.out.println("something is wrong.") ;
  }
```

The == Operator Compared With equals

►The == operator determines if two references are identical to each other (that is, refer to the same object).

►The `equals` method determines if objects are “equal” but not necessarily identical.

►The `Object` implementation of the equals method uses the == operator.

►User classes can override the `equals` method to implement a domain-specific test for equality.

►Note: You should override the `hashCode` method if you override the `equals` method.

```
public class MyDate {
    private int day ;
    private int month ;
    private int year ;
    public MyDate(int day, int month, int year){
       this.day = day ; this.month = month ; this.year = year ;
    }
    public boolean equals(Object o) {
       boolean result = false ;
       if( (o != null) && (o instanceof MyDate) ){
         MyDate d = (MyDate) o ;
        if( (day == d.day) && (month == d.month) && (year == d.year) )
           result = true ;
       }
       return result ;
    }
```

```
    public int hashCode() {
        return (
                        (new Integer(day).hashCode())
                      ^ (new Integer(day).hashCode())
                      ^ (new Integer(day).hashCode())
               ) ;
    }
}
```

```
public class TestEquals {
   public static void main(String[] args) {
        MyDate date1 = new MyDate(13, 3, 1976) ;
        MyDate date2 = new MyDate(13, 3, 1976) ;

        if( date1 == date2 )
         System.out.println("date1 is identical to date2") ;
        else
         System.out.println("date1 is not identical to date2") ;

        if( date1.equals(date2) )
         System.out.println("date1 is equal to date2") ;
        else
         System.out.println("date1 is not equal to date2") ;
```

```
System.out.println("set date2 to date1") ;
        date2 = date1 ;
        if( date1 == date2 )
         System.out.println("date1 is identical to date2") ;
        else
         System.out.println("date1 is not identical to date2") ;
   }
}
```

The toString Method

►Converts an object to a `String`.

►Used during string concatenation.

►Override this method to provide information about a user-defined object in readable format.

►Primitive types are converted to a `String` using the wrapper class's `toString` static method.

```
String one = String.valueOf( 1 );
String two = String.valueOf( 2.0f );
String notTrue = String.valueOf( false );

String date = String.valueOf( new Date() );
System.out.println( date );
// Wed Jul 11 12:46:16 GMT+03:00 2001


date = null;
System.out.println( date );
// null
```

Wrapper Classes

- Look at primitive data elements as Objects

Primitive Data Type	Wrapper Class
boolean	Boolean
byte	Byte
char	Character
short	Short
int	Integer
long	Long
float	Float
double	Double

```
int pInt = 500 ;
Integer wInt = new Integer(pInt) ;
int p2 = wInt.intValue();
```

```
public class StringTest {
  public static void main(String[] args)  {
        String s = "123";
        Integer wInt = new Integer(Integer.parseInt(s)) ;
        System.out.println(  wInt )                   ;
        System.out.println(  wInt.intValue() )        ;
        System.out.println(  wInt.floatValue() )      ;
                System.out.println(  wInt.toString() )  ;
          }
}
```

```
public class StringTest{
  public static void main(String[] args) {
        String s = "-123.45";
        Double wDouble = new Double(Double.parseDouble(s));
        System.out.println(  wDouble ) ;
        System.out.println(  wDouble.intValue() )       ;
        System.out.println(  wDouble.toString() )       ;
  }
}
```

6# Class Design

►Exercise-1: "Creating Subclasses of Bank Accounts"

►Exercise-2: "Creating Customer Accounts"

Sun microsystems HANDS-ON LAB