COP 3330: Object-Oriented Programming Summer 2011

Classes In Java – Part 1
Inheritance

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop3330/sum2011

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida

Classes In Java

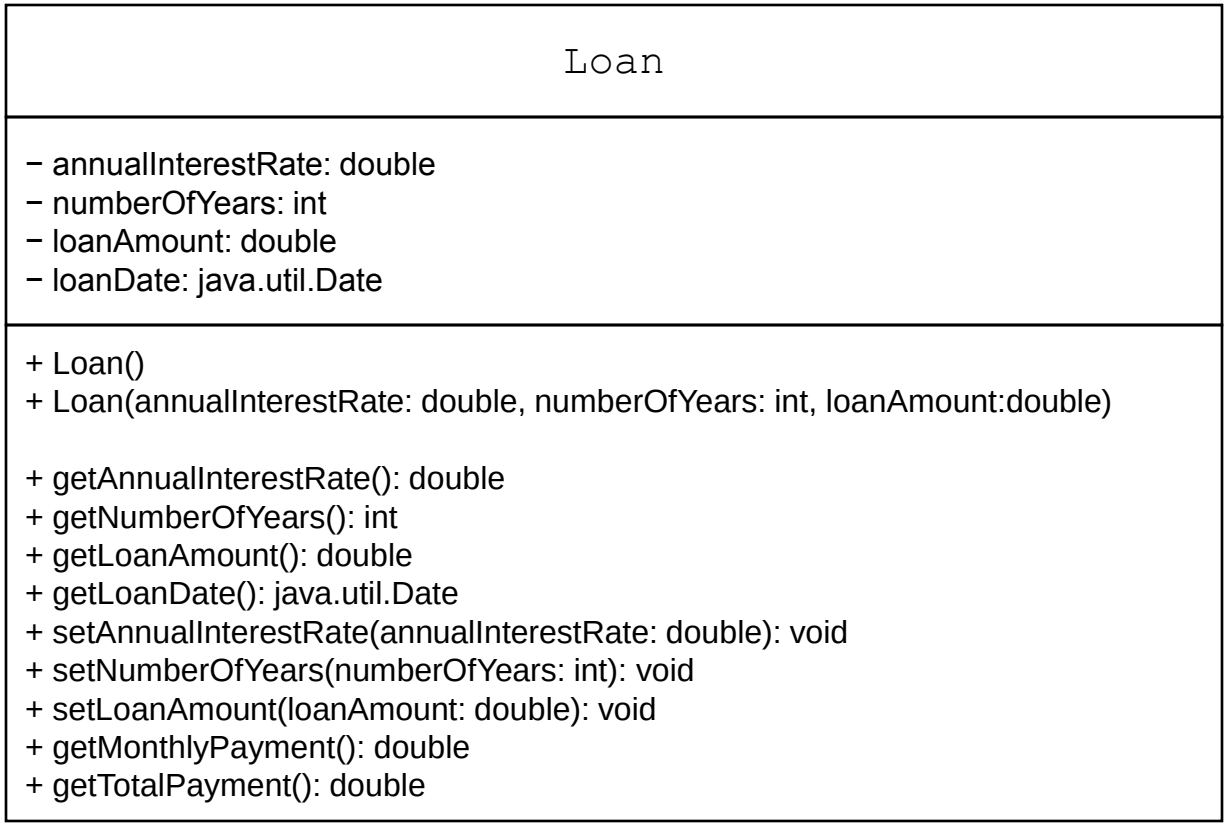
- A class is a group of objects that share common state and behavior. A class is an abstraction or description of an object.
- An object, on the other hand, is a concrete entity that exists in space and time.
- OO languages use classes to define the state and behavior associated with objects and to provide a means to create the objects that make up a program.
- Thus, the class acts as a blueprint or template from which objects can be created (instantiated in OO terms).
  - While `car` is a class, my car and your car are two instances of the class.

Classes In Java

- We've already seen the UML notation for describing a class, and we will continue to use and expand on this notation as we delve deeper into classes in Java.
- One thing that you want to get straight now is that developing a class and using a class are two distinct tasks.
- Developing a class, ultimately, requires that you know all of the inner details of the class and how it works.
- Using a class does not require that you know anything about how the class actually works, only in how you can utilize the class to solve the problem at hand.

Classes In Java

- As an example of this concept, consider the UML diagram of a Loan class as shown below:



Classes In Java

- Now, let's write a class that uses the Loan class without ever worrying about how the loan class is actually implemented.
- In other words, we can use this class without knowing anything more about the class than the methods (and variables) that are available to us outside of the class. The implementation details of this, for example, how the monthly payment is calculated, are not important for us to be able to utilize the class.
- If you want to run the test class, I've put the `Loan.java` source code file on the course web site.

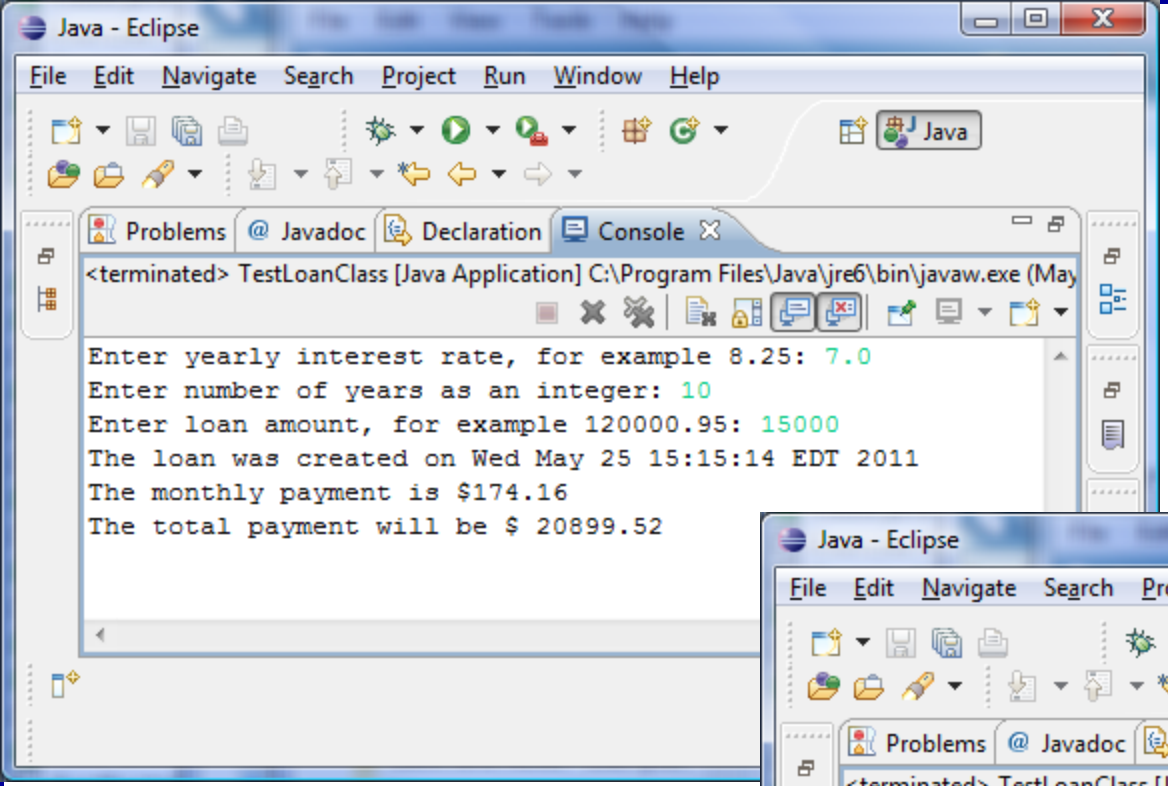
Rectangle.java | Thing.java | Loan.java | *TestLoanClass.java | WindChill.java

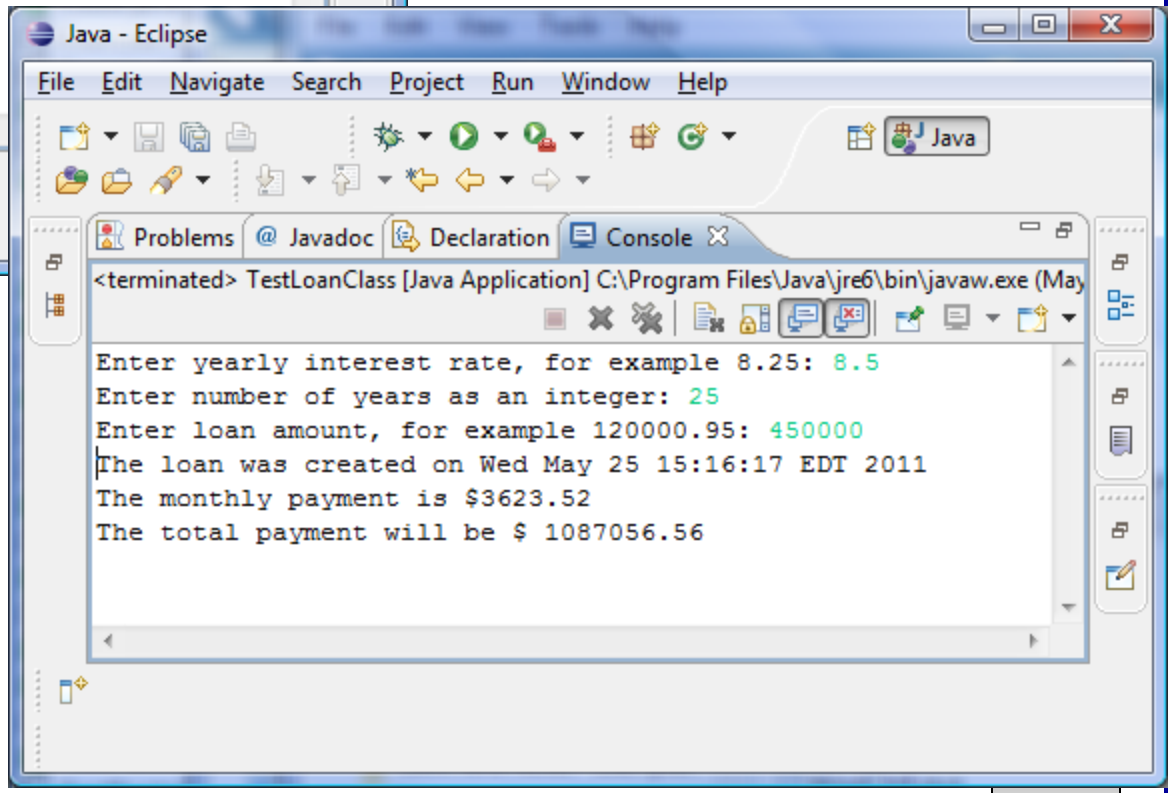
```
// a class to test the Loan class

import java.util.Scanner;

public class TestLoanClass {
  /** Main method */
  public static void main(String[] args) {
    // Create a Scanner
    Scanner input = new Scanner(System.in);
    // Enter yearly interest rate
    System.out.print(
      "Enter yearly interest rate, for example 8.25: ");
    double annualInterestRate = input.nextDouble();
    // Enter number of years
    System.out.print("Enter number of years as an integer: ");
    int numberOfYears = input.nextInt();
    // Enter loan amount
    System.out.print("Enter loan amount, for example 120000.95: ");
    double loanAmount =  input.nextDouble();
    // Create Loan object
    Loan loan =
      new Loan(annualInterestRate, numberOfYears, loanAmount);
    // Format to keep two digits after the decimal point
    double monthlyPayment =
      (int)(loan.getMonthlyPayment() * 100) / 100.00;
    double totalPayment =
      (int)(loan.getTotalPayment() * 100) / 100.00;
    // Display results
    System.out.println("The loan was created on " +
      loan.getLoanDate().toString() + "\nThe monthly payment is $" +
      monthlyPayment + "\nThe total payment will be $ " + totalPayment);
  }//end main method
}//end TestLoanClass
```

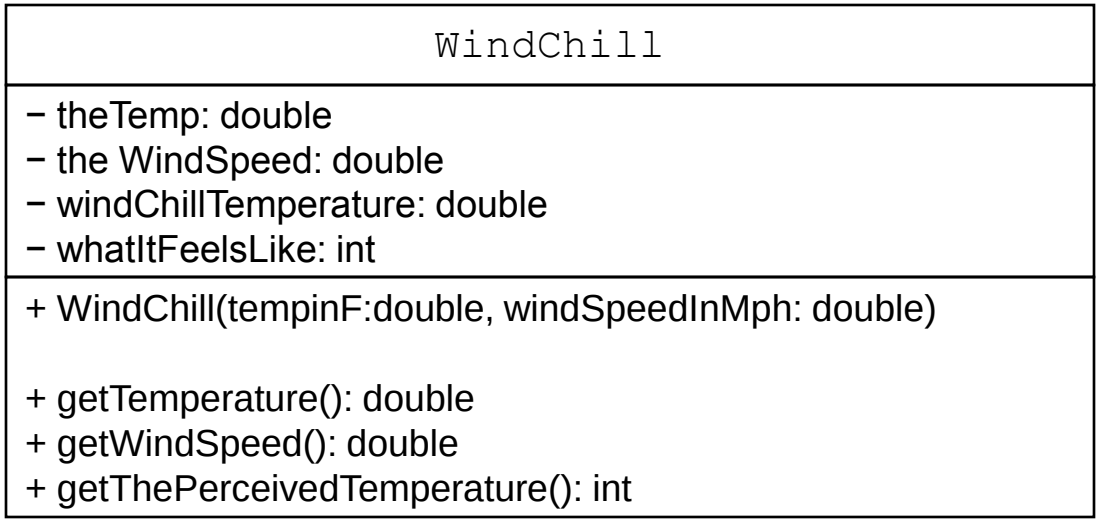

Two Different Executions of the `TestLoanClass` Application





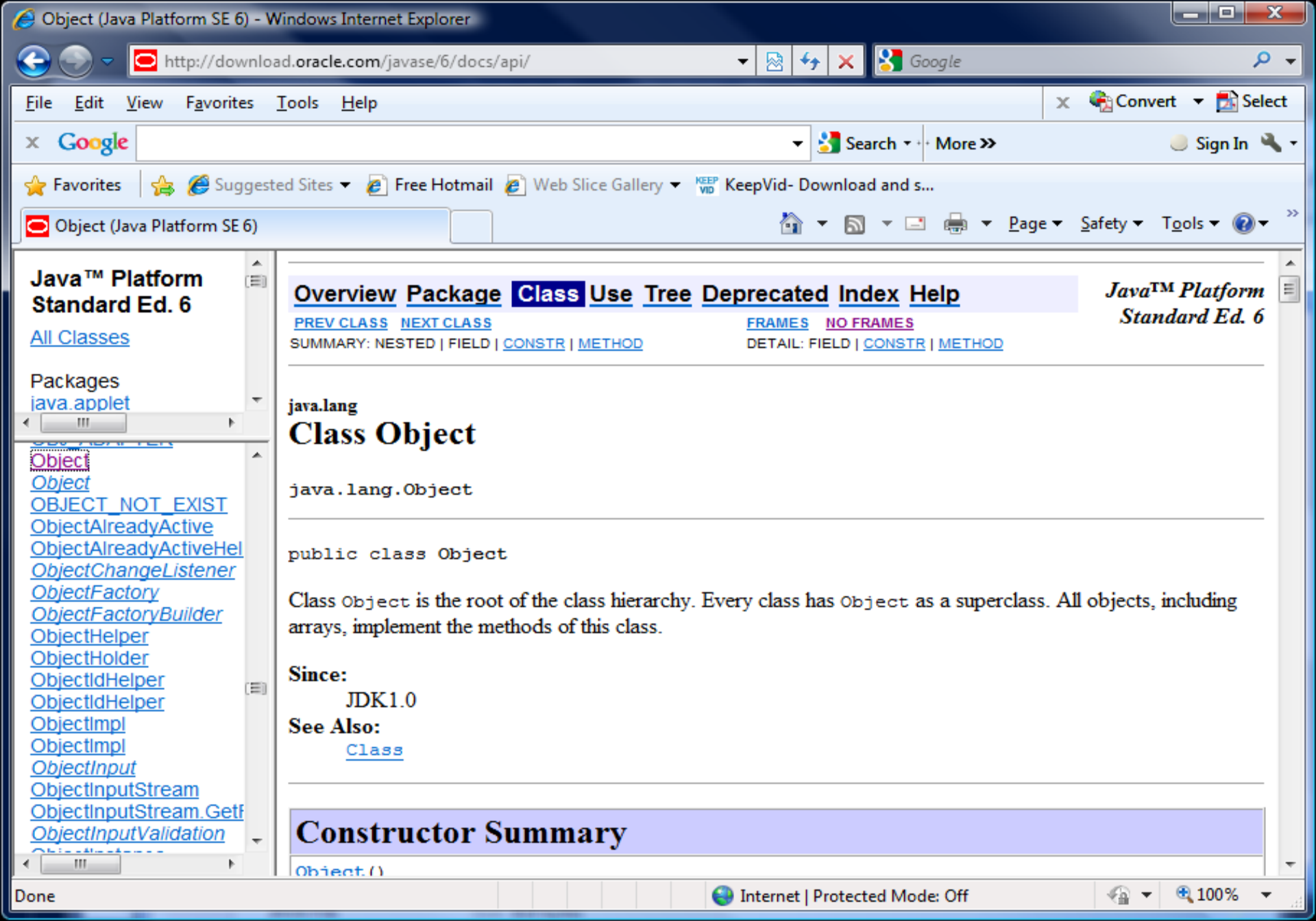
Classes In Java

- To fully illustrate this point, I've created a class named WindChill as defined in the UML diagram shown below. I've implemented this class and placed the `WindChill.class` file on the course code page. (PRACTICE PROBLEM #2) – I want you to write a small test program that will use this class to determine the wind chill for various conditions that you will input. We'll see how you did next class.



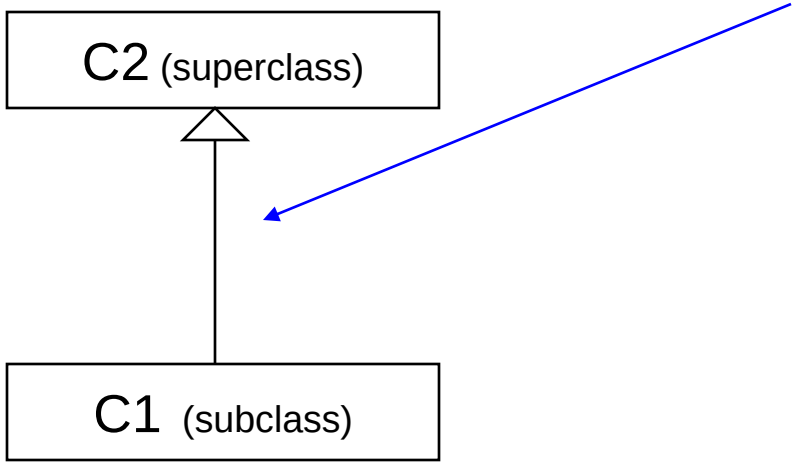
Inheritance Revisited

- OO languages allow you to derive new classes from existing classes via inheritance.
- Inheritance is a powerful component of OO languages that allow the software developer to reuse software.
- Without knowing it (perhaps), you have already been using inheritance when you developed your first programs for this course. This is because every class in Java is inherited from an existing class, either explicitly or implicitly. All of the classes you've constructed so far, as well as all of those in the notes have implicitly extended (inherited from) the `java.lang.Object` class.


Object (Java Platform SE 6) - Windows Internet Explorer
http://download.oracle.com/javase/6/docs/api/
Google
File Edit View Favorites Tools Help
Convert Select
Search More
Sign In
Favorites Suggested Sites Free Hotmail Web Slice Gallery KeepVid- Download and s...
Object (Java Platform SE 6)
Page Safety Tools
Java™ Platform Standard Ed. 6
All Classes
Packages
java.applet
Object
Object
OBJECT_NOT_EXIST
ObjectAlreadyActive
ObjectAlreadyActiveHel
ObjectChangeListener
ObjectFactory
ObjectFactoryBuilder
ObjectHelper
ObjectHolder
ObjectIdHelper
ObjectIdHelper
ObjectImpl
ObjectImpl
ObjectInput
ObjectInputStream
ObjectInputStream.GetF
ObjectInputValidation
Overview Package Class Use Tree Deprecated Index Help
Java™ Platform Standard Ed. 6
PREV CLASS NEXT CLASS
FRAMES NO FRAMES
SUMMARY: NESTED | FIELD | CONSTR | METHOD
DETAIL: FIELD | CONSTR | METHOD
java.lang
Class Object
java.lang.Object
public class Object
Class Object is the root of the class hierarchy. Every class has Object as a superclass. All objects, including arrays, implement the methods of this class.
Since:
JDK1.0
See Also:
Class
Constructor Summary
Object()
Done
Internet | Protected Mode: Off
100%

Inheritance Revisited

- In Java terminology, a class C1 extended from another class C2 is called a subclass, and C2 is called the superclass.
- A superclass can also be called a parent class or a base class, and a subclass may be referred to as a child class, extended class, or a derived class.



When two classes are related by inheritance, the `is-a` relationship will apply to the classes.

The `is-a` relationship holds between two classes when one class is a specialized instance of the second.

Five Forms Of Inheritance

Form of Inheritance	Description
Specification	The superclass defines behavior that is implemented in the subclass but not in the superclass; this provides a way to guarantee that the subclass implements the same behavior. (In short, the superclass defines what the subclass must do, but does not specify how it is to be done.)
Specialization	The subclass is a specialized form of the superclass but satisfies the specifications of the superclass in all relevant aspects. (In short, there is an is-a relationship between the subclass and the superclass.)
Extension	The subclass adds new functionality to the parent class but does not change any inherited behavior.
Limitation	The subclass restricts the use of some behavior inherited from the superclass. Typically, the inherited behavior that is limited is set as a no-operation in the subclass, i.e., the operation still exists but has no effect on the state of the object on which it is invoked.
Combination (Multiple Inheritance)	The subclass inherits features from more than one superclass. This is not implemented in Java, although through the use of interfaces there are ways around this limitation in Java.

Inheritance In Java

- Suppose we are given the task of designing some classes to model geometric objects like circles, square, and rectangles.
- These geometric objects share many common properties and behaviors. They can be drawn in a certain color and be either filled or unfilled.
- What is the best way to design this set of classes? (HINT: use inheritance!)
- Using a top-down approach, meaning let's think in general terms first and then move to the more specific (specialization).

Inheritance In Java

- We'll consider the general case of geometric objects first, and define a class `GeometricObject`, that will be used to model all geometric objects.

GeometricObject
− color: String //color of the object (default: white) − filled: boolean //filled or not filled – (default: false) − dateCreated: java.util.Date //date of creation
+ GeometricObject() //constructor + getColor(): String //returns the color + setColor(color: String): void //sets a new color + isFilled(): boolean //returns the filled property + setFilled(filled: boolean): void //sets a new filled property + getDateCreated(): java.util.Date //returns the dateCreated + toString(): String // returns a string representation of the object

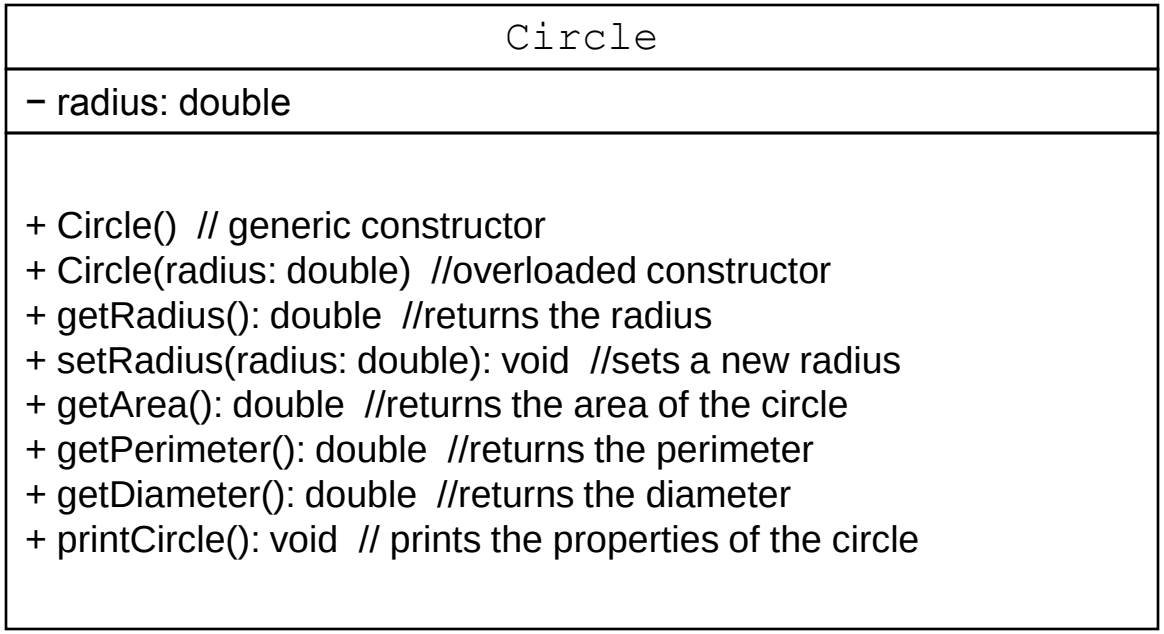
UML class diagram for
GeometricObject class

Inheritance In Java

- Now if we consider a circle, we realize that it is just a special case of a geometric object. Hence, while it has some special properties of its own (those that make it a circle), it also shares certain properties with all other geometric objects.
- Thus, using inheritance allows us to view a circle as simply a special case of the more general geometric object. In this way, we can allow the circle objects to share its common properties and methods with other geometric objects.
- It makes sense to define a `Circle` class than extends the `GeometricObject` class.

Inheritance In Java

- Here is our definition for the `Circle` class. Notice at this point, that all we have is the `Circle` class and we have not yet shown that it extends or is related to the `GeometricObject` class. We'll do that in a minute.



UML class diagram for Circle subclass

Inheritance In Java

- Now if we consider a rectangle object, we realize that, just like the circle object, it is just a special case of a geometric object.
- Thus, we'll define a rectangle class that will inherit from the geometric object class in exactly the same way that the circle class will inherit from the geometric object class.
- Just as it did for the circle objects, it makes sense to define a `Rectangle` class than extends the `GeometricObject` class.

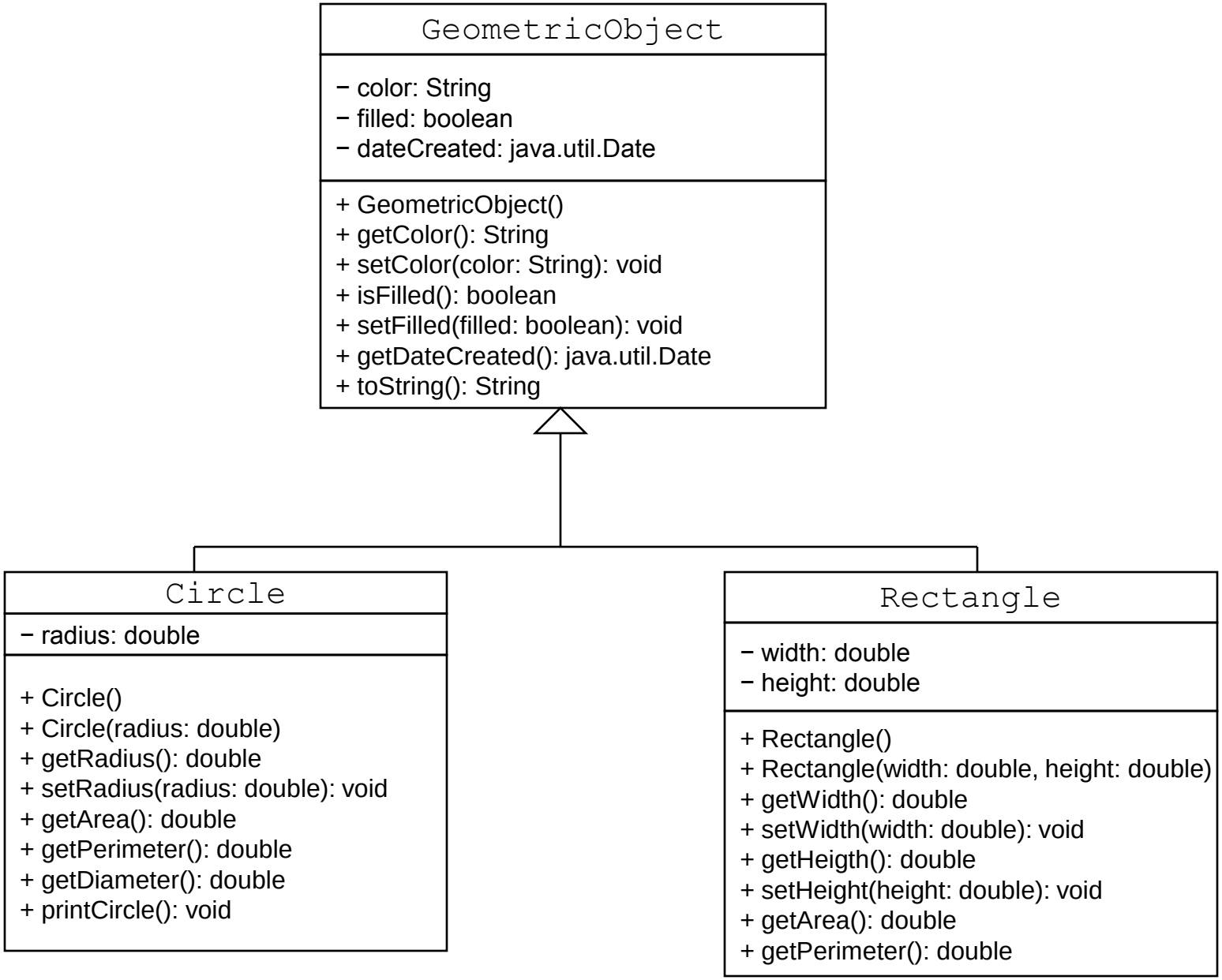
Inheritance In Java

- Here is our definition for the `Rectangle` class. Notice at this point, that all we have is the `Rectangle` class and we have not yet shown that it extends or is related to the `GeometricObject` class. We'll do that next.

Rectangle
− width: double − height: double

+ Rectangle() // generic constructor
+ Rectangle(width: double, height: double) //overloaded constructor
+ getWidth(): double //returns the width
+ setWidth(width: double): void //sets a new width
+ getHeight(): double //returns the height
+ setHeight(height: double): void //sets a new height
+ getArea(): double //returns the area
+ getPerimeter(): double //returns the perimeter

UML class diagram for Rectangle subclass


GeometricObject
− color: String
− filled: boolean
− dateCreated: java.util.Date
+ GeometricObject()
+ getColor(): String
+ setColor(color: String): void
+ isFilled(): boolean
+ setFilled(filled: boolean): void
+ getDateCreated(): java.util.Date
+ toString(): String
Circle
− radius: double
+ Circle()
+ Circle(radius: double)
+ getRadius(): double
+ setRadius(radius: double): void
+ getArea(): double
+ getPerimeter(): double
+ getDiameter(): double
+ printCircle(): void
Rectangle
− width: double
− height: double
+ Rectangle()
+ Rectangle(width: double, height: double)
+ getWidth(): double
+ setWidth(width: double): void
+ getHeigth(): double
+ setHeight(height: double): void
+ getArea(): double
+ getPerimeter(): double

TestLoanClass.java | *GeometricObject.jav | Circle.java | Rectangle.java | TestCircleRectangle. | 18

```
/* GeometricObject Class - Classes In Java
 * used to illustrate inheritance in OO/Java
 *     MJL May 25, 2011
 * No known bugs
 */
public class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  /* Construct a default geometric object */
  public GeometricObject() {
    //uncomment following line for constructor chaining display
    System.out.println("In GeometricObject default constructor method");
    dateCreated = new java.util.Date();
  }//end default constructor

  /* Return color */
  public String getColor() {
    return color;
  }//end getColor method

  /* Set a new color */
  public void setColor(String color) {
    this.color = color;
  }//end setColor method

  /* Return filled. Since filled is boolean,
     so, the get method name is isFilled */
  public boolean isFilled() {
    return filled;
  }//end isFilled method
```

GeometricObject class

Java - Classes In Java/src/GeometricObject.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

`GeometricObject` class (continued)

*GeometricObject.jav | Circle.java | Rectangle.java | TestCircleRectangle. | 19

```
  /* Set a new filled */
  public void setFilled(boolean filled) {
    this.filled = filled;
  }//end setFilled method

  /* Get dateCreated */
  public java.util.Date getDateCreated() {
    return dateCreated;
  }//end dateCreated method

  /* Return a string representation of this object */
  public String toString() {
    return "created on " + dateCreated + "\ncolor: " + color +
      " and filled: " + filled;
  }//end toString method
}//end class GeometricObject
```

Writable | Smart Insert | 1

*GeometricObject.jav | Circle.java | Rectangle.java | TestCircleRectangle. | 19

```
/* Circle Class - Classes in Java
 * Extends GeometricObject class - used in inheritance example
 *
 * MJL May 25, 2011
 * No known bugs
*/

public class Circle extends GeometricObject {
  private double radius;

  /* default constructor */
  public Circle() {
    //uncomment following line for constructor chaining display
    System.out.println("In default Circle constructor.");
  }//end default constructor

  /* radius specific constructor */
  public Circle(double radius) {
    //uncomment following line for constructor chaining display
    System.out.println("In radius specific Circle constructor.");
    this.radius = radius;
  }//end radius specific constructor

  /* Return radius */
  public double getRadius() {
    return radius;
  }//end getRadius method
```

Circle class

```
  /* Set a new radius */
  public void setRadius(double radius) {
    this.radius = radius;
  }//end setRadisu method

  /* Return area */
  public double getArea() {
    return radius * radius * Math.PI;
  }//end getArea method

  /* Return diameter */
  public double getDiameter() {
    return 2 * radius;
  }//end getDiameter method

  /* Return perimeter */
  public double getPerimeter() {
    return 2 * radius * Math.PI;
  }//end getPerimeter method

  /* Print the circle info */
  public void printCircle() {
    System.out.println("The circle is created " + getDateCreated() +
      " and the radius is " + radius);
  }//end printCircle method
}//end Circle class
```

`Circle` class
(continued)

*GeometricObject.jav | Circle.java | Rectangle.java | TestCircleRectangle. | 19

```
/* class Rectangle - Classes in Java
 * Extends GeometricObject class - inheritance example
 *
 * MJL May 25, 2011
 * No known bugs
 */

public class Rectangle extends GeometricObject {
  private double width;
  private double height;

  /* default constructor */
  public Rectangle() {
    //uncomment following line for constructor chaining display
    System.out.println("In default Rectangle constructor.");
  }//end default constructor

  /* length and width specific constructor */
  public Rectangle(double width, double height) {
    //uncomment following line for constructor chaining display
    System.out.println("In length and width specific Rectangle constructor.");
    this.width = width;
    this.height = height;
  }//end length and width specific constructor

  /* Return width */
  public double getWidth() {
    return width;
  }//end getWidth method
```

`Rectangle` class

*GeometricObject.jav | Circle.java | Rectangle.java | TestCircl

```
   /* Set a new width */
   public void setWidth(double width) {
      this.width = width;
   }//end setWidth method

   /* Return height */
   public double getHeight() {
      return height;
   }//end getHeight method

   /* Set a new height */
   public void setHeight(double height) {
      this.height = height;
   }//end setHeight method

   /* Return area */
   public double getArea() {
      return width * height;
   }//end getArea method

   /* Return perimeter */
   public double getPerimeter() {
      return 2 * (width + height);
   }//end getPerimeter method

}//end Rectangle class
```

`Rectangle` class
(continued)

*GeometricObject.jav | Circle.java | Rectangle.java | TestCircleRectangle. | 19

```
/* TestCircleRectangle Class
 * a driver class to test the inheritance hierarchy developed
 * using the GeometricObject superclass
 *
 * MJL May 25, 2011
 * No known bugs
 */

public class TestCircleRectangle {
  public static void main(String[] args) {
    Circle circle = new Circle(1);
    System.out.println("A circle " + circle.toString());
    System.out.println("The radius is " + circle.getRadius());
    System.out.println("The area is " + circle.getArea());
    System.out.println("The diameter is " + circle.getDiameter());
    System.out.println();

    Rectangle rectangle = new Rectangle(2, 4);
    System.out.println("A rectangle " + rectangle.toString());
    System.out.println("The area is " + rectangle.getArea());
    System.out.println("The perimeter is " + rectangle.getPerimeter());
    System.out.println();

    Rectangle rectangle2 = new Rectangle(6, 9);
    rectangle2.setColor("blue");
    rectangle2.setFilled(true);
    System.out.println("A rectangle " + rectangle2.toString());
    System.out.println("The area is " + rectangle2.getArea());
    System.out.println("The perimeter is " + rectangle2.getPerimeter());
  }//end main method
}//end class TestCircleRectangle
```

`TestCircleRectangle` class – A driver class

Java - Classes In Java/src/TestCircleRectangle.java - Eclipse

File Edit Source Refactor Navigate Search Project Run Window Help

Java

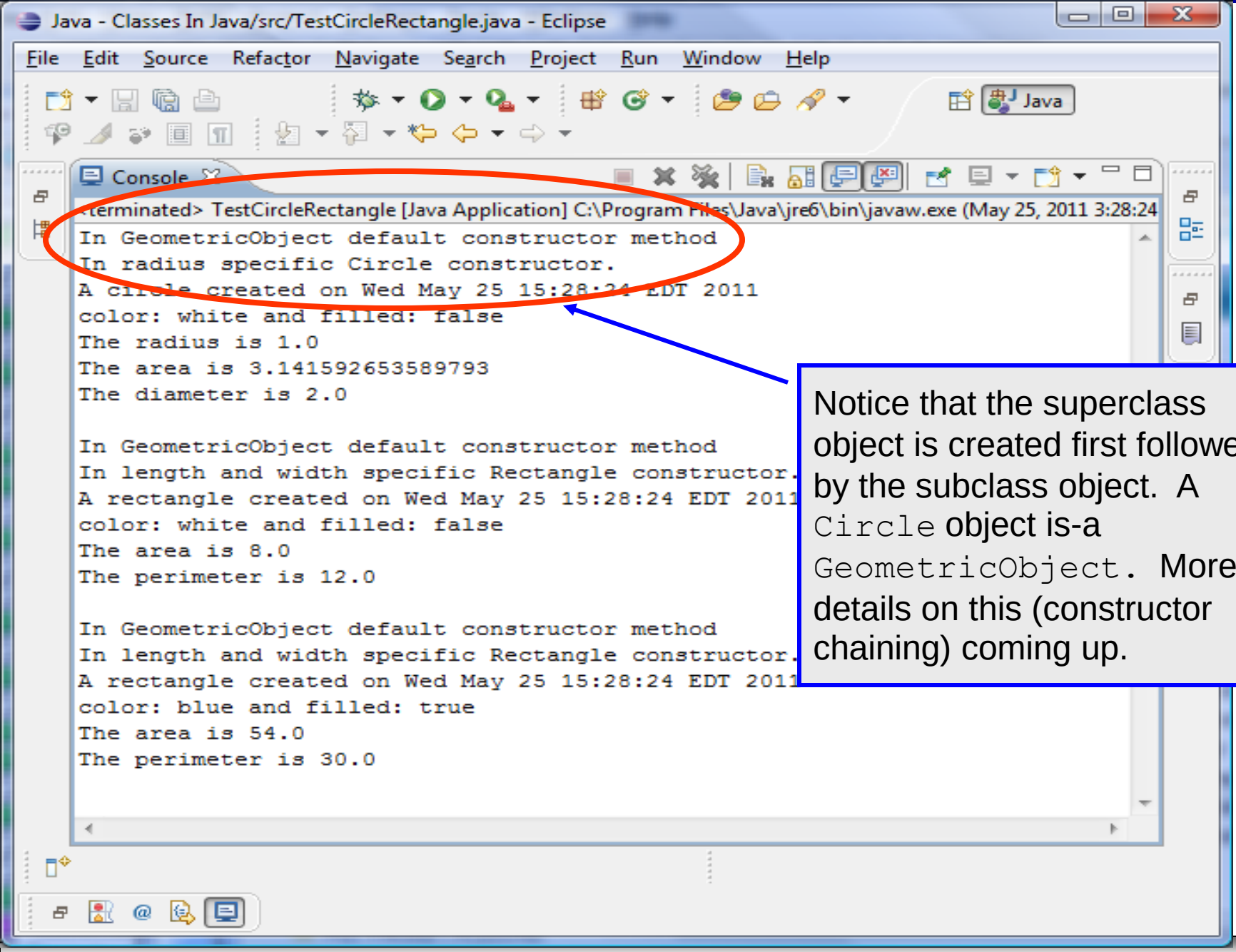
Console

<terminated> TestCircleRectangle [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (May 25, 2011 3:28:24

```
In GeometricObject default constructor method
In radius specific Circle constructor.
A circle created on Wed May 25 15:28:24 EDT 2011
color: white and filled: false
The radius is 1.0
The area is 3.141592653589793
The diameter is 2.0

In GeometricObject default constructor method
In length and width specific Rectangle constructor.
A rectangle created on Wed May 25 15:28:24 EDT 2011
color: white and filled: false
The area is 8.0
The perimeter is 12.0

In GeometricObject default constructor method
In length and width specific Rectangle constructor.
A rectangle created on Wed May 25 15:28:24 EDT 2011
color: blue and filled: true
The area is 54.0
The perimeter is 30.0
```

Notice that the superclass object is created first followed by the subclass object. A `Circle` object is-a `GeometricObject`. More details on this (constructor chaining) coming up.

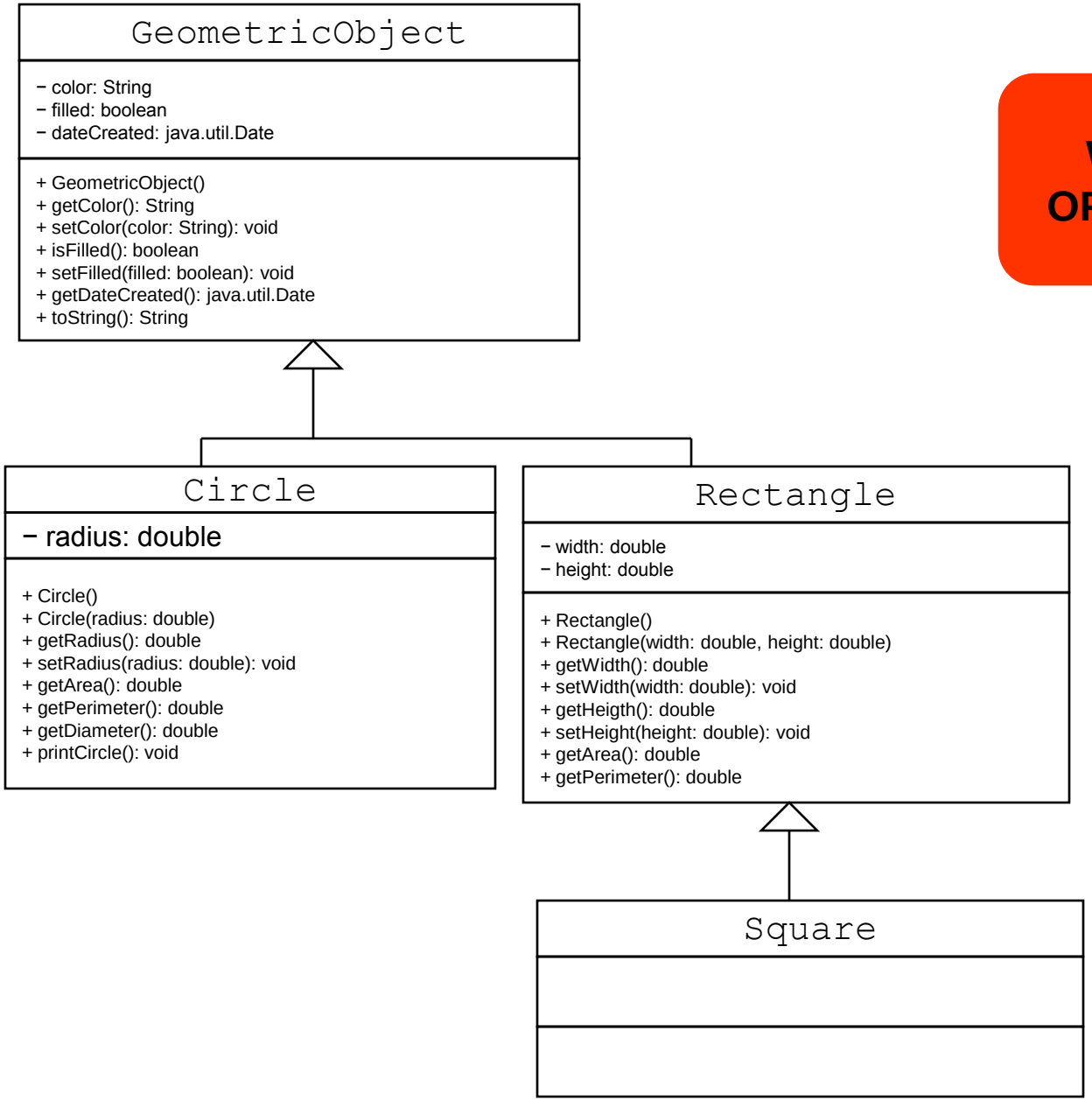
More Details On Inheritance In Java

- Contrary to the conventional interpretation , a subclass is not a subset of its superclass. In fact, a subclass typically contains more information and functions than its superclass. This is because the subclass needs the variables and/or methods that define the special properties and/or behaviors of the specialized objects that are member of the subclass.
- Remember that every instance of an object in a subclass is also (first and foremost) an instance of an object of its superclass as well.

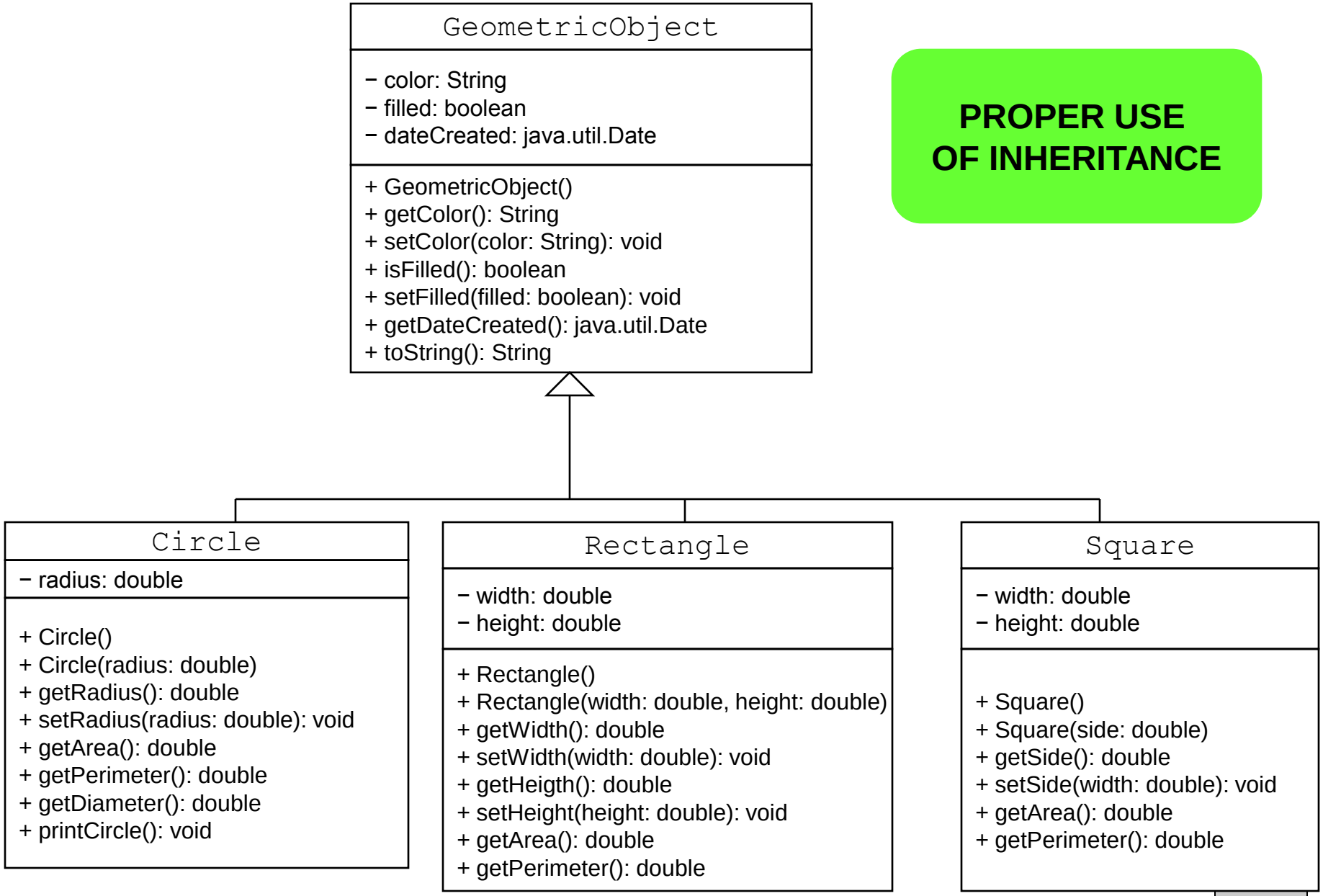
More Details On Inheritance In Java

- Not all `is-a` relationships should be modeled using inheritance. For example, a square `is-a` rectangle, but you should not declare a `Square` class to extend the `Rectangle` class. Why?

- Because there is nothing to extend (or supplement) from a rectangle to a square. In other words, a square object has no additional properties or behaviors that would in any way differentiate it from a rectangle.
  - Note: you might however, create a Square class that extends the GeometricObject class, if you really wanted to view rectangles and squares differently.

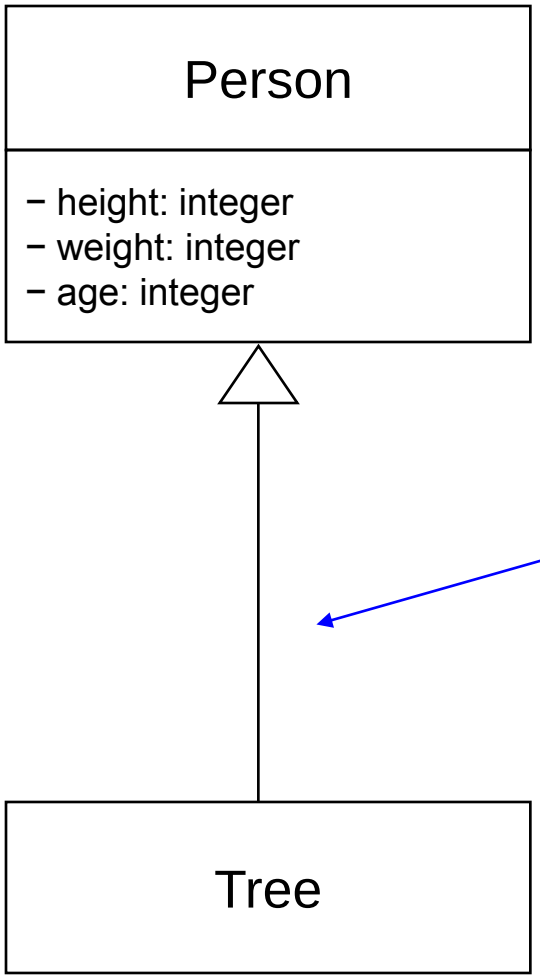


WRONG USE OF INHERITANCE


GeometricObject
− color: String
− filled: boolean
− dateCreated: java.util.Date
+ GeometricObject()
+ getColor(): String
+ setColor(color: String): void
+ isFilled(): boolean
+ setFilled(filled: boolean): void
+ getDateCreated(): java.util.Date
+ toString(): String
PROPER USE OF INHERITANCE
Circle
− radius: double
+ Circle()
+ Circle(radius: double)
+ getRadius(): double
+ setRadius(radius: double): void
+ getArea(): double
+ getPerimeter(): double
+ getDiameter(): double
+ printCircle(): void
Rectangle
− width: double
− height: double
+ Rectangle()
+ Rectangle(width: double, height: double)
+ getWidth(): double
+ setWidth(width: double): void
+ getHeigth(): double
+ setHeight(height: double): void
+ getArea(): double
+ getPerimeter(): double
Square
− width: double
− height: double
+ Square()
+ Square(side: double)
+ getSide(): double
+ setSide(width: double): void
+ getArea(): double
+ getPerimeter(): double

More Details On Inheritance In Java

- Inheritance can be used to model the `is-a` relationship between two classes of objects. Do not blindly extend a class just for the sake of reusing methods.
- For example, it makes no sense for a Tree class to extend a Person class, even though they might share common properties such as height, weight, and age, etc. This would seriously detract from the readabilty and maintainability of the software.
- A subclass and its superclass should have the `is-a` relationship.



WRONG USE OF INHERITANCE

Wrong use of inheritance because a tree is not a person. The two classes do not have an `is-a` relationship, so this hierarchy makes no logical sense

Using the `super` Keyword

- A subclass inherits accessible data fields and methods from its superclass, but it does not inherit constructors.
- However, superclass constructors are accessible to the subclass through the use of the keyword **`super`**.
- The keyword `super` refers to the superclass in which it which it appears. It can be used in two ways:
  1. To invoke (call) a superclass constructor.
  2. To invoke a superclass method.

Invoking Superclass Constructors

- The syntax to invoke a superclass constructor is:

  `super()`, or `super(parameters)`

- The first case invokes the no-arg constructor of its superclass, and the second case invokes the superclass constructor that matches the argument list.

*** IMPORTANT ***

The statement `super()` or `super(parameters)` must appear as the first line of the subclass constructor – no exceptions! This is the only way to invoke a superclass constructor.

Invoking Superclass Constructors

- A constructor can invoke an overloaded constructor or its superclass's constructor. If neither of them is invoked explicitly, the compiler puts super() as the first statement in the constructor.

```
public A() {
}
```

equivalent

```
public A() {
  super();
}
```

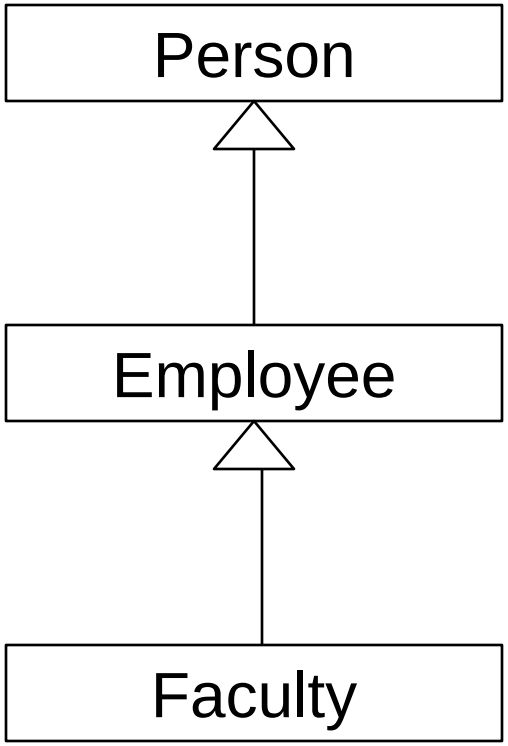
```
public A(double d) {
   //some statements
}
```

equivalent

```
public A(double d) {
   super();
   //some statements
}
```

- In any case, constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain. A superclass's constructor is called before the subclass's constructor. This is called constructor chaining.

Constructor Chaining Example



Consider this hierarchy, which implies that an Employee `is-a` Person and a Faculty `is-a` Employee (and thus `is-a` Person)

Constructor Chaining Example

Circle.java | Rectangle.java | TestCircleRectangle.

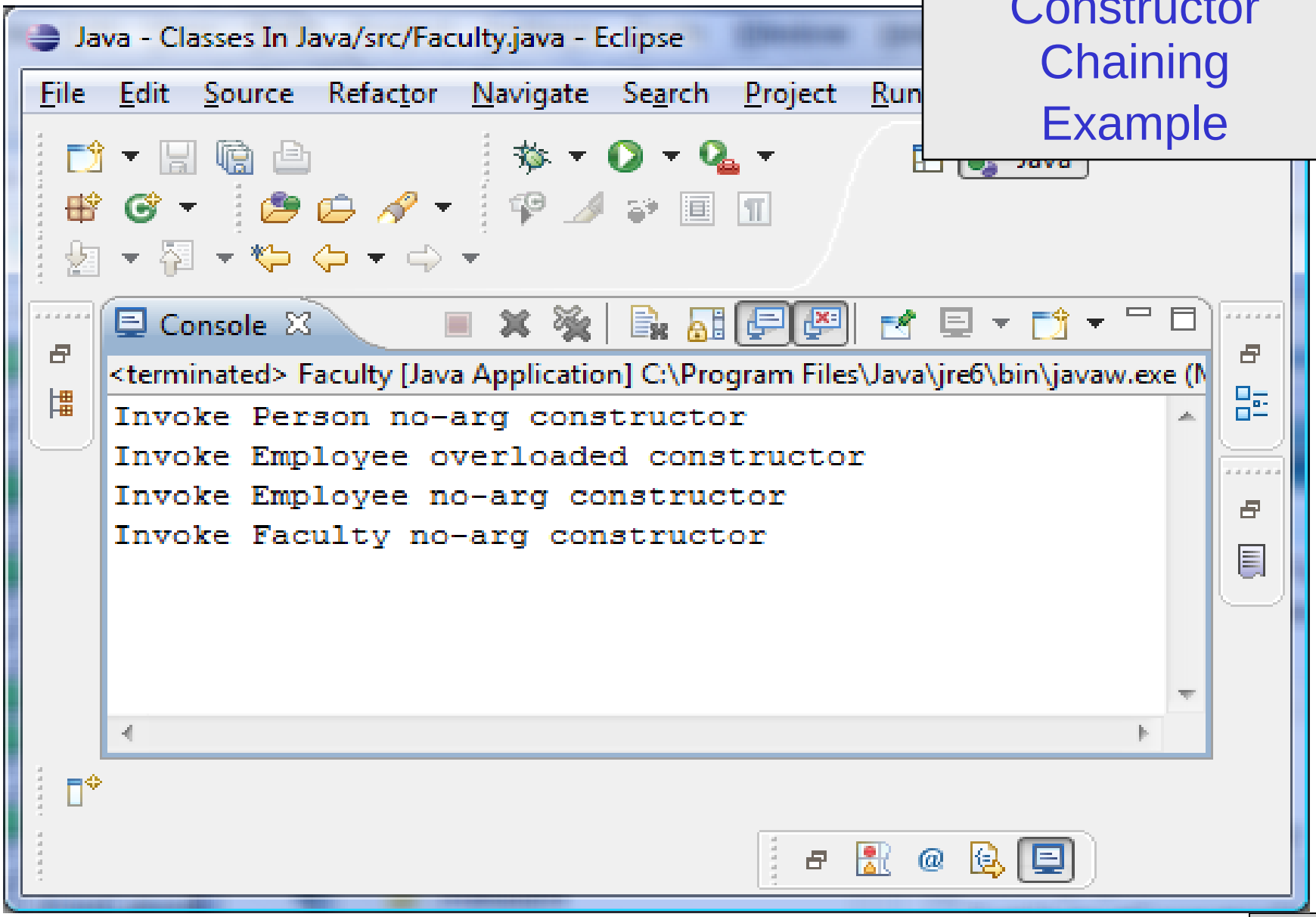
```
/* Constructor chaining example - Summer 201
public class Faculty extends Employee {
  public static void main (String[] args) {
     new Faculty();
  }
  public Faculty() {
    System.out.println("Invoke Faculty no-arg constructor");
  }
}

class Employee extends Person {
  public Employee() {
       this("Invoke Employee overloaded constructor");
       System.out.println("Invoke Employee no-arg constructor");
  }
  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("Invoke Person no-arg constructor");
  }
}
```

Execution begins here

Constructor Chaining Example



Invoking Superclass Constructors

- If a class is designed to be extended, it is better to provide a no-arg constructor to avoid programming errors. Consider the following case…What is the output of this program?

```
public class Apple extends Fruit {
}
class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit constructor is invoked");
  }
}
```

- Since no constructor is explicitly defined in Apple, Apple's default no-arg constructor is declared implicitly. Since Apple is a subclass of Fruit, Apple's default constructor automatically invokes Fruit's no-arg constructor. However, Fruit does not have a no-arg constructor since it has an explicit constructor defined. Therefore, the program cannot be compiled.

Invoking Superclass Methods

- The keyword `super` is also used to reference a method other than the constructor in the superclass.
- The syntax is: `super.method(parameters);`
- As an example, suppose in our earlier example (persons, employees, and faculty) that the `Employee` class contained a public method `getSalary()`. If you wanted to obtain the salary of a faculty person, you could do so within the `Faculty` class with a statement such as:

```
facultySalary = super.getSalary();
```

- Note: You can use `super.p()` to invoke the method `p()` defined in the superclass. However, suppose `A` extends `B` and `B` extends `C` and method `p()` is defined in `C`. It is not possible from within `A` to invoke `super.super.p();` This is not allowed in Java.

Overriding Methods

- A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass (i.e., to provide the specialized behavior peculiar to the subclass). This is referred to as method overriding.
- As an example, consider the `toString` method in the `GeometricObject` class of our earlier example. This method returns the string representation for a geometric object. Suppose that within the `Circle` class we want to provide an implementation of the `toString` method to specialize the output for a circle object. (Do this yourself for practice!)

```
//override the toString method defined in GeometricObject
public String toString() {
   return super.toString() + "\nradius is: " + radius;
}
```


Some Additional Issues On Overriding Methods

- Private data fields in a superclass are not accessible outside the class, Therefore, they cannot be used directly in a subclass. They can, however, be accessed/mutated through public accessor/mutator methods if defined in the superclass.
- An instance method can be overridden only if it is accessible. Thus, a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

Some Additional Issues On Overriding Methods

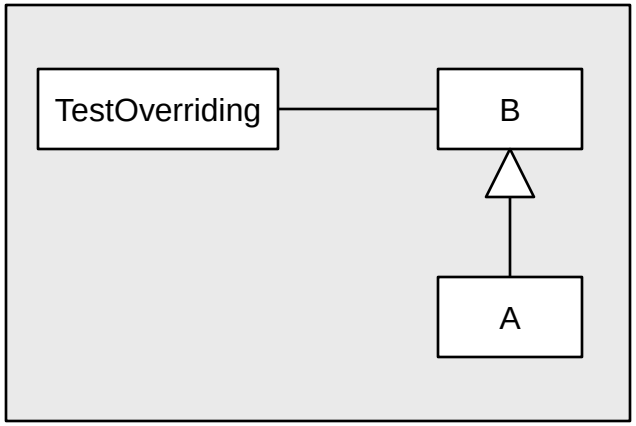
- Like an instance method, a static method can be inherited. However, a static method cannot be overridden. If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden. The hidden static methods can be invoked using the syntax `SuperClassName.staticMethodName`.

- Do not confuse the terms overridding and overloading when applied to methods. Overloading a method is a way to provide more than one method with the same name but with different signatures to distinguish them. To override a method, the method must be defined in the subclass using the same signature and same return type as in its superclass.

Difference between Overriding and Overloading

```
public class TestOverriding {
    public static void main (String[] args) {
        A a = new A();
        a.p(10);
    }
}//end TestOverriding
class B {
  public void p (int i) {
  }
}//end B
class A extends B {
  //this method overrides the method in B
  public void p (int i) {
      System.out.println(i);
  }
}//end A
```

Example of method overridding



Difference between Overriding and Overloading

```
public class TestOverloading {
    public static void main (String[] args) {
        A2 a = new A2();
        a.p(10);
    }
}//end TestOverloading
class B2 {
  public void p (int i) {
  }
}//end B2
class A2 extends B2 {
  //this method overloads the method in B2
  public void p (double i) {
        System.out.println(i);
  }
}//end A2
```

Example of method overloading