

COP 3330: Object-Oriented Programming Summer 2011

Classes In Java – Part 3 Abstract Classes and Interfaces

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



Abstract Classes

- In the inheritance hierarchy, classes become more specific and concrete with each new subclass. If you move from a subclass back up to a superclass, the classes become more general and less specific.
- Class design should ensure that a superclass contains common features of its subclasses.
- Sometimes a superclass is so abstract that it cannot have any specific instances. Such a class is called an **abstract class**.

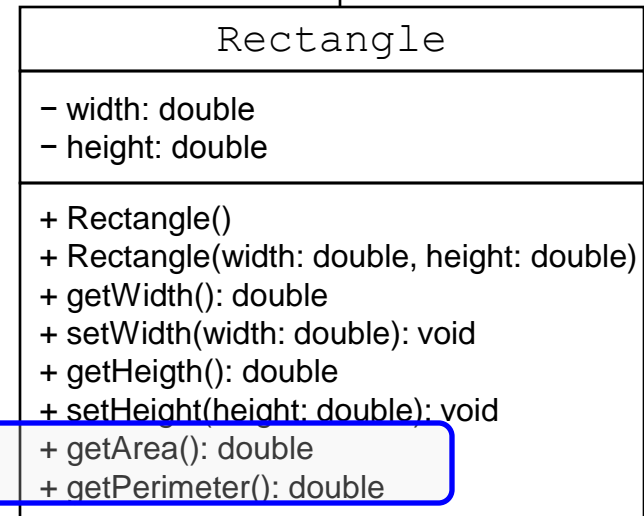
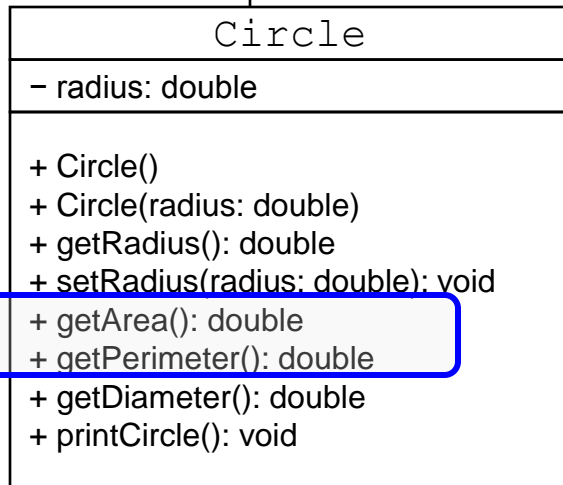
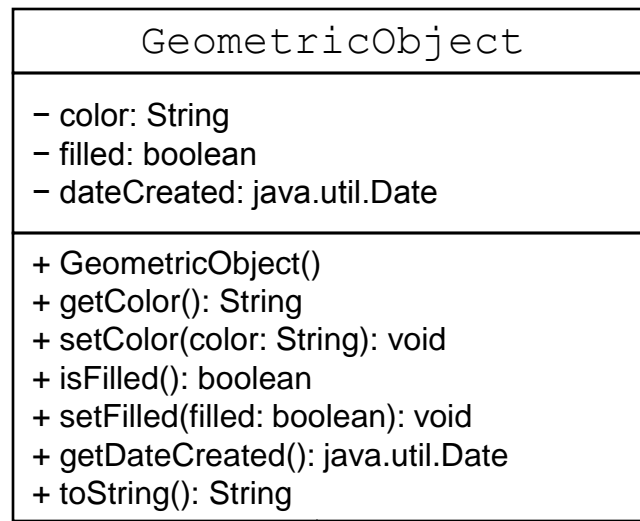


Abstract Classes

- Recall our `GeometricObject` class that was the superclass for `Circle` and `Rectangle`.
- The `GeometricObject` class models common features of geometric objects.
- Both the `Circle` and `Rectangle` classes contain the `getArea()` and `getPerimeter()` methods for computing the area and perimeter of a circle and a rectangle.



Original UML classes



The original Rectangle class

Rectangle.java

```
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
```



The original Circle class

```
//end setRadius 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 " + getDa  
        " and the radius is " + radius);  
}
```



Abstract Classes

- However, our earlier design was somewhat lacking in that you can compute the area and perimeter of all geometric objects, hence these are not properties of circles or rectangles, but of geometric objects.
- A better design would be to declare the `getArea()` and `getPerimeter()` methods in the `GeometricObject` class. But there is a problem doing this. What is the problem?



Abstract Classes

- The problem is that in the `GeometricObject` class we can't provide an implementation for these methods because their implementation depends on the specific type of geometric object.
 - To compute the area of a circle we need to use the expression `perimeter = 2πr`, but for a rectangle the expression is `perimeter = 2 (height+width)`.
- In order to define these methods in the `GeometricObject` class, they need to be defined as **abstract methods**.



Abstract Classes

- An abstract method is specified using the `abstract` modifier in the method header.

- Example:

```
public abstract double getArea();
```

- Similarly, an abstract class is denoted using the `abstract` modifier in the class header.

- Example:

- ```
public abstract class GeometricObject {
 . . .
}
```

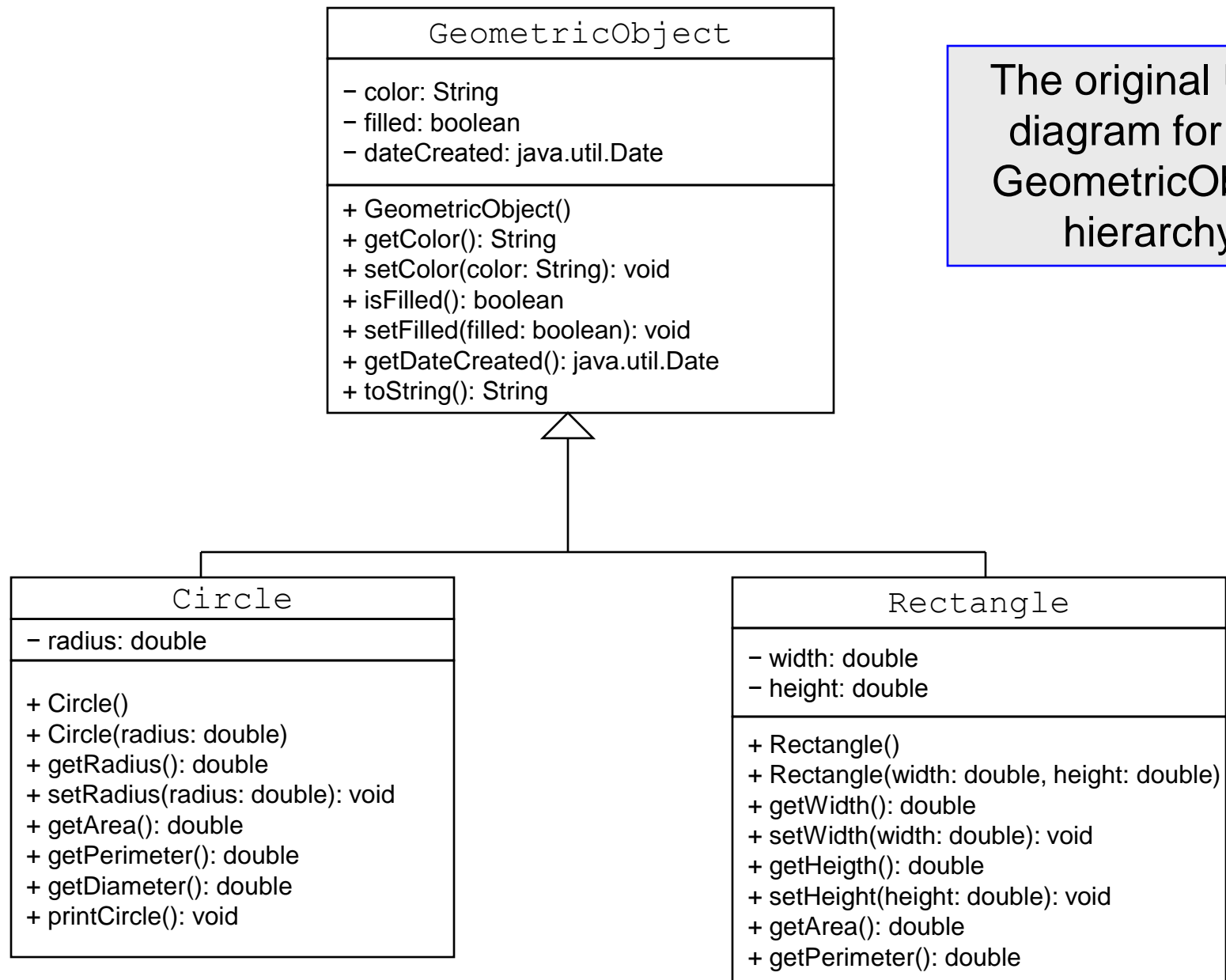
- In UML notation, the names of abstract classes and their methods are italicized (see pages 9-10 of Introduction to UML notes).



# Abstract Classes

- Abstract classes are like regular classes with data fields and methods, but you cannot create instances of abstract classes using the `new` operator.
- An abstract method is a method signature without implementation. Its implementation is provided by the subclasses.
- **Any class that contains an abstract method must be declared abstract.**
- The constructor of an abstract class is declared protected, because it is used only by subclasses. When you create an instance of a concrete subclass, the subclass's parent class constructor is invoked to initialize data fields in the parent class.
- Let's now reconsider the `GeometricObject` case and redesign it using abstract classes and methods.





The original UML diagram for the GeometricObject hierarchy



Abstract class (italics)

*GeometricObject*

- color: String  
- filled: boolean  
- dateCreated: java.util.Date

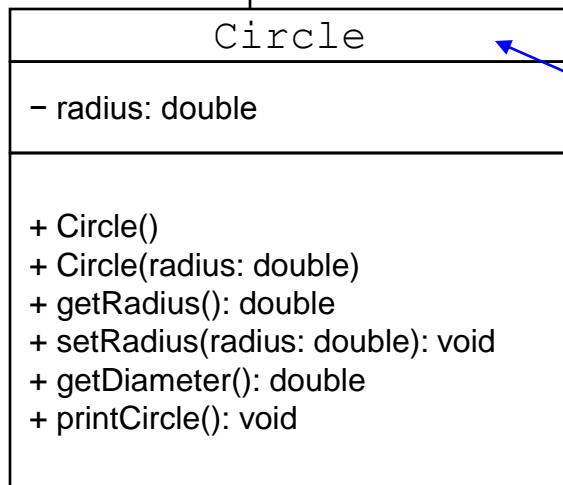
Constructor denoted as protected

# GeometricObject()  
+ getColor(): String  
+ setColor(color: String): void  
+ isFilled(): boolean  
+ setFilled(filled: boolean): void  
+ getDateCreated(): java.util.Date  
+ toString(): String

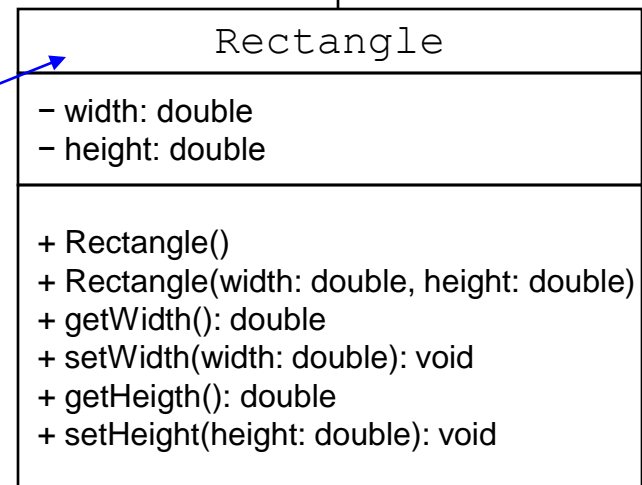
Abstract methods (italics)

+ *getArea(): double*  
+ *getPerimeter(): double*

The new UML diagram for the GeometricObject hierarchy using an abstract class and methods



Methods  
getArea and  
getPerimeter  
are overridden in  
Circle and  
Rectangle.  
Overridden  
methods are  
commonly  
generally omitted  
in the UML for  
subclasses.



# Abstract Classes

- Now let's re-write the `GeometricObject` class using abstract methods which will convert it into an abstract class.
- What changes will we need to make to the `Circle` and `Rectangle` classes?
- **Answer:** Absolutely nothing! We will need to provide implementations for the `getArea()` and `getPerimeter()` methods in both classes, but they already have them (thus they override the ones defined in the `GeometricObject` class!)



```
/* AbstractGeometricObject Class - Classes In Java
 * used to illustrate abstract classes/inheritance in OO/Java
 * MJL June 7, 2011
 * No known bugs
 */
public abstract class AbstractGeometricObject {
 private String color = "white";
 private boolean filled;
 private java.util.Date dateCreated;

 /* Construct a default geometric object */
 protected AbstractGeometricObject() {
 dateCreated = new java.util.Date();
 } //end default constructor
```

Declare class as abstract

Denote constructor as protected  
(Go to page 23 for more  
information on this modifier.)



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

Declare abstract method `getArea()`

```
/* abstract method for returning the area of a geometric object */
public abstract double getArea();
```

Note: No braces!

```
/* abstract method for returning the perimeter of a geometric object */
public abstract double getPerimeter();
```

Declare abstract method `getPerimeter()`

```
} //end class GeometricObject
```

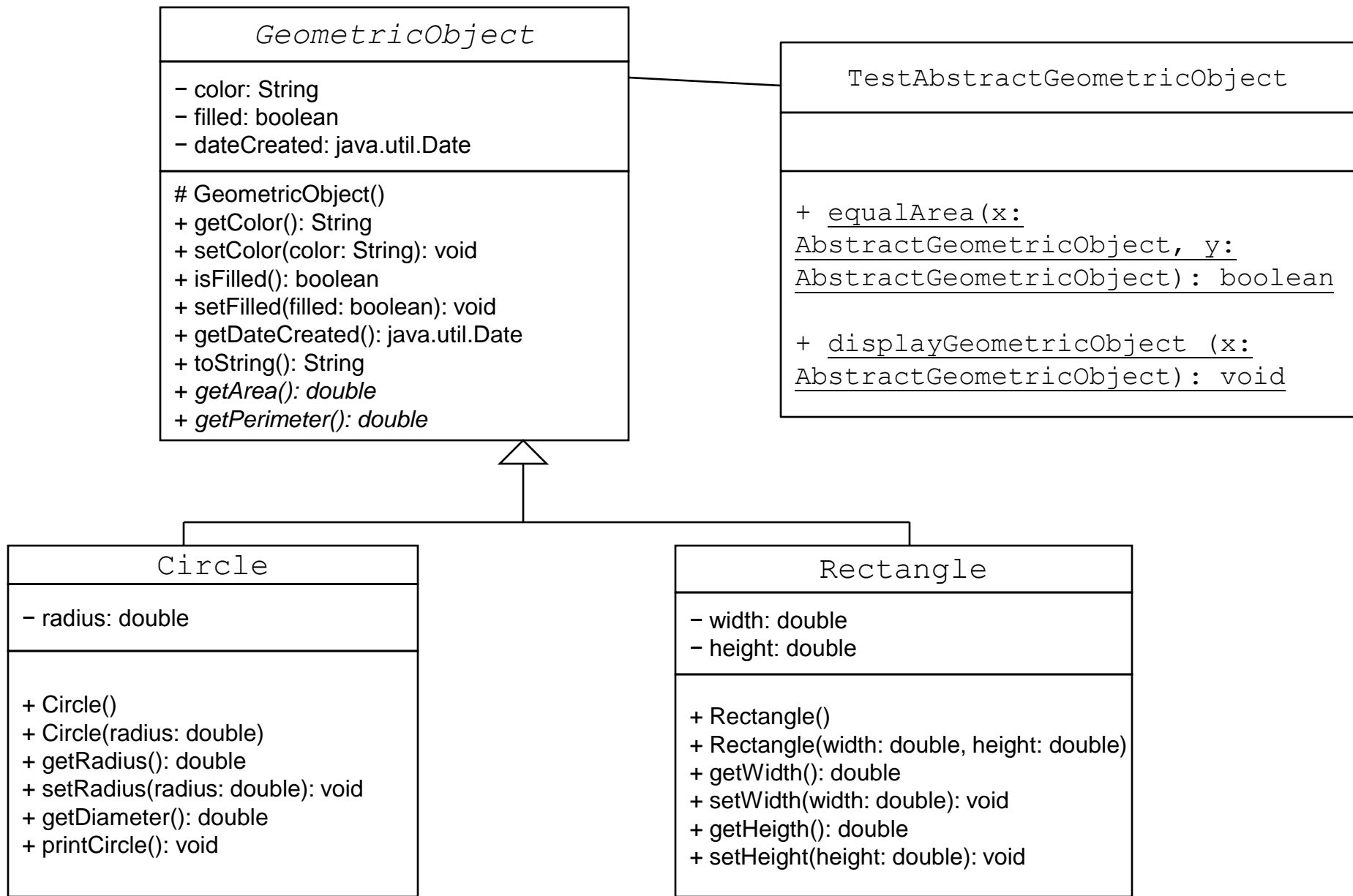


# Abstract Classes

- Now let's write a class to test the abstract version of the `GeometricObject` class.
- We'll use the example to illustrate not only how abstract classes work, but also illustrate the advantages of using abstract classes.
- Notice that if the methods `getArea` and `getPerimeter` were only defined in the `Circle` and `Rectangle` classes and not in the `GeometricObject` class, we would not be able to define the `equalArea` and `displayObject` methods shown in this example.







```
public class TestAbstractGeometricObject {

 /** A method for comparing the areas of two geometric objects */
 public static boolean equalArea(AbstractGeometricObject object1,
 AbstractGeometricObject object2) {
 return object1.getArea() == object2.getArea();
 } //end method equalArea
 /** A method for displaying a geometric object */

 public static void displayGeometricObject(AbstractGeometricObject object) {
 System.out.println();
 System.out.println("The area is " + object.getArea());
 System.out.println("The perimeter is " + object.getPerimeter());
 } //end method displayGeometricObject

 /** Main method */
 public static void main(String[] args) {
 // Declare and initialize three geometric objects
 System.out.println("\nCreating a circle object with radius of 5.");
 AbstractGeometricObject geoObject1 = new Circle(5); //implicit casting
 System.out.println("Creating a rectangle with width 5 and height 3.");
 AbstractGeometricObject geoObject2 = new Rectangle(5, 3); //implicit casting
 System.out.println("Creating a rectangle with width 3 and height 5.");
 AbstractGeometricObject geoObject3 = new Rectangle(3, 5); //implicit casting
```



```
System.out.println("\nDoes geoObject1 have the same area as geoObject2? " +
 equalArea(geoObject1, geoObject2));
System.out.println("\nDoes geoObject2 have the same area as geoObject3? " +
 equalArea(geoObject2, geoObject3));
// Display circle characteristics
System.out.print("\nFor the circle (geoObject1):");
displayGeometricObject(geoObject1);
//Display rectangle 1 characteristics
System.out.print("\nFor the first rectangle (geoObject2):");
displayGeometricObject(geoObject2);
//Display rectangle 2 characteristics
System.out.print("\nFor the second rectangle (geoObject3):");
displayGeometricObject(geoObject3);
} //end main method

} //end class TestAbstractGeometricObject
```



```
Creating a circle object with radius of 5.
Creating a rectangle with width 5 and height 3.
Creating a rectangle with width 3 and height 5.

Does geoObject1 have the same area as geoObject2? false

Does geoObject2 have the same area as geoObject3? true

For the circle (geoObject1):
The area is 78.53981633974483
The perimeter is 31.41592653589793

For the first rectangle (geoObject2):
The area is 15.0
The perimeter is 16.0

For the second rectangle (geoObject3):
The area is 15.0
The perimeter is 16.0
```



# Interesting Points On Abstract Classes

- An abstract method cannot be contained in a nonabstract class.
- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must also be declared abstract. In other words, in a nonabstract subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.
- Abstract methods cannot be static.
- An abstract class cannot be instantiated using the `new` operator, but you can still define its constructors, which are invoked implicitly by the constructors of its subclasses. For example, the constructors of `GeometricObject` are invoked in the `Circle` and `Rectangle` classes.



# Interesting Points On Abstract Classes

- A class that contains any abstract methods, must be declared as abstract. However, it is possible to declare an abstract class that contains no abstract methods! In this case, you cannot create instances of the class using the `new` operator. This class would be used as a base class for defining a new subclass only.
- A subclass can be abstract even if its superclass is concrete. For example, the `Object` class is concrete, but its subclasses, such as `GeometricObject`, may be abstract.
- A subclass can override a method from its superclass to declare it abstract. This is very unusual, but is useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be declared abstract.



# More On Accessibility Modifiers

- So far this semester, we've mostly used `public` and `private` accessibility modifiers for our class variables and methods. With the introduction of abstract classes, we now need to expand this to include the `protected` modifier.
- A `protected` data or `protected` method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in different packages.
- Packages are simply a way to organize files into different directories, usually based on functionality, usability, or category. Packaging is a way to avoid class name collisions when the same class name is used.
- The accessibility modifiers are summarized in the table on the next page.



# More On Accessibility Modifiers

| Modifier on members in a class | Access from the same class | Access from the same package | Access from a subclass | Access from a different package |
|--------------------------------|----------------------------|------------------------------|------------------------|---------------------------------|
| <code>public</code>            | yes                        | yes                          | yes                    | yes                             |
| <code>protected</code>         | yes                        | yes                          | yes                    | no                              |
| (default)                      | yes                        | yes                          | no                     | no                              |
| <code>private</code>           | yes                        | no                           | no                     | no                              |





## package p1:

```
public class C1{
 public int x;
 protected int y;
 int z;
 private int u;

 protected void m() { }
}
```

```
public class C2{
 C1 o = new C1();
 can access o.x;
 can access o.y;
 can access o.z;
 cannot access o.u;

 can invoke o.m();
}
```



```
public class C3
 extends C1{
 can access x;
 can access y;
 can access z;
 cannot access u;

 can invoke m();
}
```

## package p2:

```
public class C4
 extends C1{
 can access x;
 can access y;
 cannot access z;
 cannot access u;

 can invoke m();
}
```

```
public class C5{
 C1 o = new C1();
 can access o.x;
 cannot access o.y;
 cannot access o.z;
 cannot access o.u;

 cannot invoke o.m();
}
```



## More On Accessibility Modifiers

- Use the `private` modifier to hide members of a class completely so that they cannot be accessed directly from outside the class.
- Use no modifiers (default case) in order to allow the members of the class to be accessed directly from any class within the same package but not from other packages.
- Use the `protected` modifier to enable the members of the class to be accessed by the subclasses in any package or classes in the same package.
- Use the `public` modifier to enable the members of the class to be accessed by any class.



# More On Accessibility Modifiers

- A class can be used in two ways: for creating instances of the class, and for creating subclasses by extending the class.
- Make the members `private` if they are not intended for use from outside the class.
- Make the members `public` if they are intended for the users outside of the class.
- Make the fields or methods `protected` if they are intended for the extenders of the class, but not the users of the class.
- The `private` and `protected` modifiers can be used only for members of the class. The `public` modifier and the default modifier (i.e. no modifier) can be used on members of the class as well as on the class itself. A class with no modifier (i.e., not a public class) is not accessible by classes from other packages.



## More On Accessibility Modifiers

- One additional note – a subclass may override a `protected` method in its superclass and change its visibility to `public`. However, a subclass cannot weaken the accessibility of a method defined in the superclass.
- For example, if a method is defined as `public` in the superclass, it must be defined as `public` in the subclass.



# Preventing Extending and Overriding

- Sometimes you may want to prevent classes from being extended. In such cases, use the `final` modifier to indicate that a class is final and thus cannot be a parent class.
  - In Java, the `Math` class and the `String` class (among others) are defined as final meaning that you cannot extend the class.
- You can also define a method to be final, a final method cannot be overridden by its subclasses.



# Interfaces

- An **interface** is a class-like construct that contains only constants and abstract methods. (It is class-like because it is not technically a class but rather a partial template of what must be in a class that implements the interface.)
- In many ways an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects.
- An interface is a Java type and can be used as such.
- The Java syntax for declaring an interface is:

```
modifier interface InterfaceName {
 // constant declarations
 // method signatures
}
```



# Interfaces

- An interface cannot specify any method implementations. All methods of an interface must be declared `public abstract`.
- All of the variables of an interface must be declared `public, final` and `static`.
- All of the methods of an interface must be `public`.
  - Since all data fields are `public final static` and all methods are `public abstract` in an interface, Java allows these modifiers to be omitted in an interface declaration. Therefore, the following declarations are equivalent.

```
public interface T {
 public static final int K = 1;
 public abstract void m();
}
```

equivalent

```
public interface T {
 int K = 1;
 void m();
}
```



# Interfaces

- An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class.
- As with an abstract class, you cannot create an instance from an interface using the `new` operator, but in most cases you can use an interface more or less the same way that you use an abstract class.
  - For example, you can use an interface as a data type for a variable, as the result of casting, and so on.





# Interfaces

- The relationship that exists between a class that uses an interface and the interface itself is known as **interface inheritance**.
- Interface inheritance and class inheritance are essentially the same thing, although in Java a class can use class inheritance from only one class (its superclass), but can use interface inheritance from several interfaces.
- Typically both forms are simply referred to as inheritance.



# Interfaces

- Lets' create a simple interface:

```
// an interface to describe how to eat
public interface Edible {
 public abstract String howToEat();
}
```

- The Edible interface can now be used anytime you want to specify how something should be eaten.
- Using an interface is accomplished by letting the class for an object `implement` the interface.



```
public class TestEdible {
 public static void main(String[] args) {
 Object[] objects = {new Apple(), new Orange(), new Banana()};
 for (int i = 0; i < objects.length; i++)
 if (objects[i] instanceof Edible)
 System.out.println(((Edible) objects[i]).howToEat());
 }
}

abstract class Fruit implements Edible {
 // Data fields, constructors, and methods omitted here
}

class Apple extends Fruit {
 public String howToEat() {
 return "Apple: Make an apple pie.";
 }
}

class Orange extends Fruit {
 public String howToEat() {
 return "Orange: Make orange juice.";
 }
}

class Banana extends Fruit {
 public String howToEat() {
 return "Banana: First peel it, then take a bite.";
 }
}
```

Use the keyword `implements` in order for a class to utilize an interface.

Fruit does not implement the `howToEat` method, it must be denoted as `abstract`.





```

public class TestEdible {
 public static void main(String[] args) {
 Object[] objects = {new Apple(), new Orange(), new Banana()};
 for (int i = 0; i < objects.length; i++)
 if (objects[i] instanceof Edible)
 System.out.println(((Edible)objects[i]).howToEat());
 }
}

```

<terminated> TestEdible [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 7, 2011 1:59:19 PM)

```

Apple: Make an apple pie.
Orange: Make orange juice.
Banana: First peel it, then take a bite.

```



# The Comparable Interface

- Suppose you want to design a generic method to find the larger of two objects. The objects could be students, circles, rectangles, etc.
- Since compare methods are different for different types of objects, you need to define a generic compare method to determine the order of the two objects. Then you can tailor the method to compare students, circles, rectangles, etc.
- For example, you could use student ID as the key for comparing students, radius as the key for comparing circles, and area as the key for comparing rectangles.
- We'll use an interface to define a generic `compareTo` method, as shown on the next page.



# The Comparable Interface

```
// Interface for comparing object - as defined in java.lang
package java.lang;

public interface Comparable {
 public int compareTo(Object obj);
}
```

- The `compareTo` method determines the order of this object with the specified object `obj`, and returns a negative integer, zero, or a positive integer if this object is less than, equal to, or greater than the specified object `obj`.



# The Comparable Interface

- Many classes in the Java library (e.g., `String` and `Date`) implement `Comparable` to define a natural order for the objects. If you look at the source code of these classes, you will see the following definitions:

```
public class String extends
 Object implements Comparable {
 //class body omitted
}
```

```
public class Date extends
 Object implements Comparable {
 //class body omitted
}
```



# The Comparable Interface

- Thus, strings are comparable, and so are dates. Let `s` be a `String` object and `d` be a `Date` object. All of the following expressions are true:

```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```





# The Comparable Interface

- Now we can declare a generic max method for finding the larger of two objects.
- Let's write two different versions and see which one is better.

```
TestEdible.java Edible.java MaxVersion1.java MaxVersion2.java

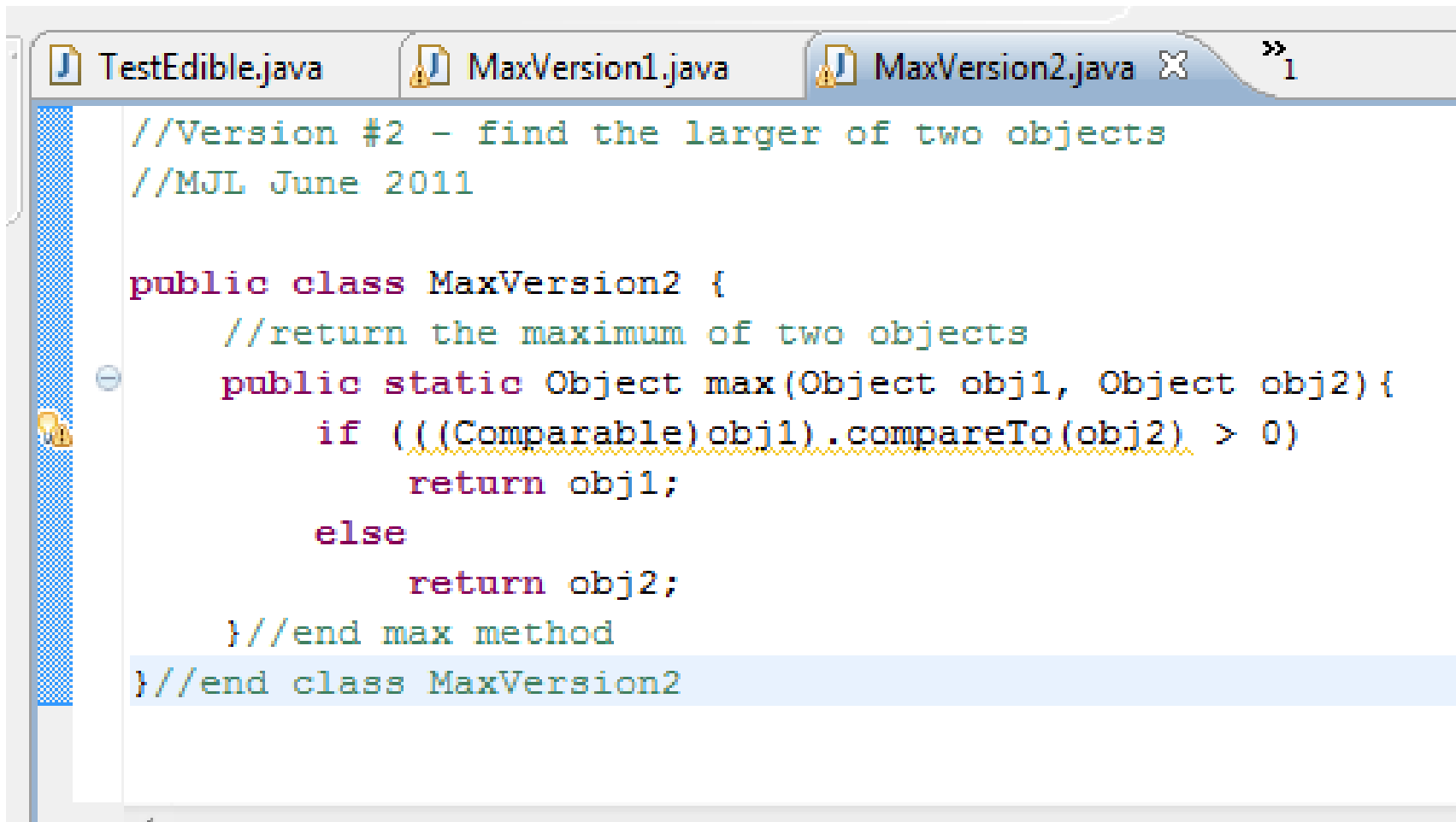
//Version #1 - find the larger of two objects
//MJL June 2011

public class MaxVersion1 {
 //return the maximum of two objects
 public static Comparable max(Comparable obj1, Comparable obj2){
 if (obj1.compareTo(obj2) > 0)
 return obj1;
 else
 return obj2;
 } //end max method
} //end class MaxVersion1
```



# The Comparable Interface

- Here's version #2:



```
//Version #2 - find the larger of two objects
//MJL June 2011

public class MaxVersion2 {
 //return the maximum of two objects
 public static Object max(Object obj1, Object obj2){
 if (((Comparable)obj1).compareTo(obj2) > 0)
 return obj1;
 else
 return obj2;
 } //end max method
} //end class MaxVersion2
```



# The Comparable Interface

- Which of the two versions of `Max` is better?
- Version #1 is simpler than the version #2. In version #2, `obj1` is declared as `Object`, and `(Comparable)obj1` tells the compiler to cast `obj1` into `Comparable` so that the `compareTo` method can be invoked from `obj1`. However, no casting is needed in version #1, since `obj1` is declared as `Comparable`.
- The `max` method in version #1 is more robust than the one in version #2. You must invoke the `max` method with two comparable objects.
  - Suppose you were to invoke `max` with two noncomparable objects such as: `Max.max(anyObject1, anyObject2)`; The compiler would detect an error using version #1, because `anyObject1` is not an instance of `Comparable`. In version #2 however, the compiler will not detect an error but a run-time error (`ClassCastException`) would occur because `anyObject1` is not an instance of `Comparable` and cannot be cast into `Comparable`.



# The Comparable Interface

- So, version #1 is the better solution to our problem.
- We'll assume that this is the version that will be included in our Comparable interface.
- Since strings and dates are comparable, you could use the max method to find the larger of two instances of String or Date as shown in the examples below:

```
String s1 = "abcedf";
String s2 = "abcdee";
String s3 = (String)Max.max(s1, s2);
```

The return value from the max method is of the Comparable type, so you need to use explicit casting.

```
Date d1 = new Date();
Date d2 = new Date();
Date d = (Date)Max.max(d1, d2);
```



# The Comparable Interface

- As things stand right now, we cannot use the `max` method to find the larger of two instances of `Rectangle`. Why?
- Because the `Rectangle` class does not implement the `Comparable` interface (it only extended the `GeometricObject` class).
- So, let's declare a new `Rectangle` class that implements the `Comparable` interface. This is shown on the next page.



# The ComparableRectangle Class

ComparableRectangle.java ✕

```
//ComparableRectangle class extends Rectangle class and implements
//the Comparable interface
//MJL June 2011
public class ComparableRectangle extends Rectangle implements Comparable {
 //constructor method
 public ComparableRectangle(double width, double height) {
 super(width, height);
 } //end constructor

 //Implement the compareTo() method defined in Comparable
 public int compareTo(Object o) {
 if (this.getArea() > ((ComparableRectangle)o).getArea())
 return 1;
 else if (this.getArea() < ((ComparableRectangle)o).getArea())
 return -1;
 else
 return 0;
 } //end implementation of compareTo()
} //end class ComparableRectangle
```



```
/* TestComparableRectangle Class
 * a driver class to test the comparable interface used in
 * the ComparableRectangle class
 * MJL June 2011
 */

public class TestComparableRectangle {
 public static void main(String[] args) {
 System.out.println("\nCreating two comparabale rectangle objects.");
 ComparableRectangle rectangle1 = new ComparableRectangle(4,6);
 ComparableRectangle rectangle2 = new ComparableRectangle(4,5);
 Comparable whichone = MaxVersion1.max(rectangle1, rectangle2);
 if (whichone.equals(rectangle1))
 System.out.println("Rectangle 1 is larger than Rectangle 2.");
 else System.out.println("Rectangle 2 is larger than Rectangle 1.");
 System.out.println("\nCreating two more comparable rectangle objects.");
 ComparableRectangle rectangle3 = new ComparableRectangle(3,3);
 ComparableRectangle rectangle4 = new ComparableRectangle(5,3);
 switch (rectangle3.compareTo(rectangle4)) {
 case -1: System.out.println("Rectangle 3 is smaller than Rectangle 4.");
 break;
 case 0: System.out.println("Rectangle 3 and Rectangle 4 are the same size.");
 break;
 case 1: System.out.println("Rectangle 3 is larger than Rectangle 4.");
 break;
 default: System.out.println("ERROR...");
 }
 }
 }
}
```

Modification of the TestCircleRectangle Class to use ComparableRectangle objects



```
ComparableRectangle.java TestComparableRectangle.java GeometricObject.java Rectangle.java
System.out.println("\nCreating two comparabale rectangle objects.");
ComparableRectangle rectangle1 = new ComparableRectangle(4,6);
ComparableRectangle rectangle2 = new ComparableRectangle(4,5);
Comparable whichone = MaxVersion1.max(rectangle1, rectangle2);
if (whichone.equals(rectangle1))
 System.out.println("Rectangle 1 is larger than Rectangle 2.");
else System.out.println("Rectangle 2 is larger than Rectangle 1.");
System.out.println("\nCreating two more comparable rectangle objects.");
ComparableRectangle rectangle3 = new ComparableRectangle(3,3);
ComparableRectangle rectangle4 = new ComparableRectangle(5,3);
switch (rectangle3.compareTo(rectangle4)) {
 case -1: System.out.println("Rectangle 3 is smaller than Rectangle 4.");
 break;
```

Problems Javadoc Declaration Console

<terminated> TestComparableRectangle [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (Jun 7, 2011 2:36:22 PM)

```
Creating two comparabale rectangle objects.
Rectangle 1 is larger than Rectangle 2.

Creating two more comparable rectangle objects.
Rectangle 3 is smaller than Rectangle 4.
```

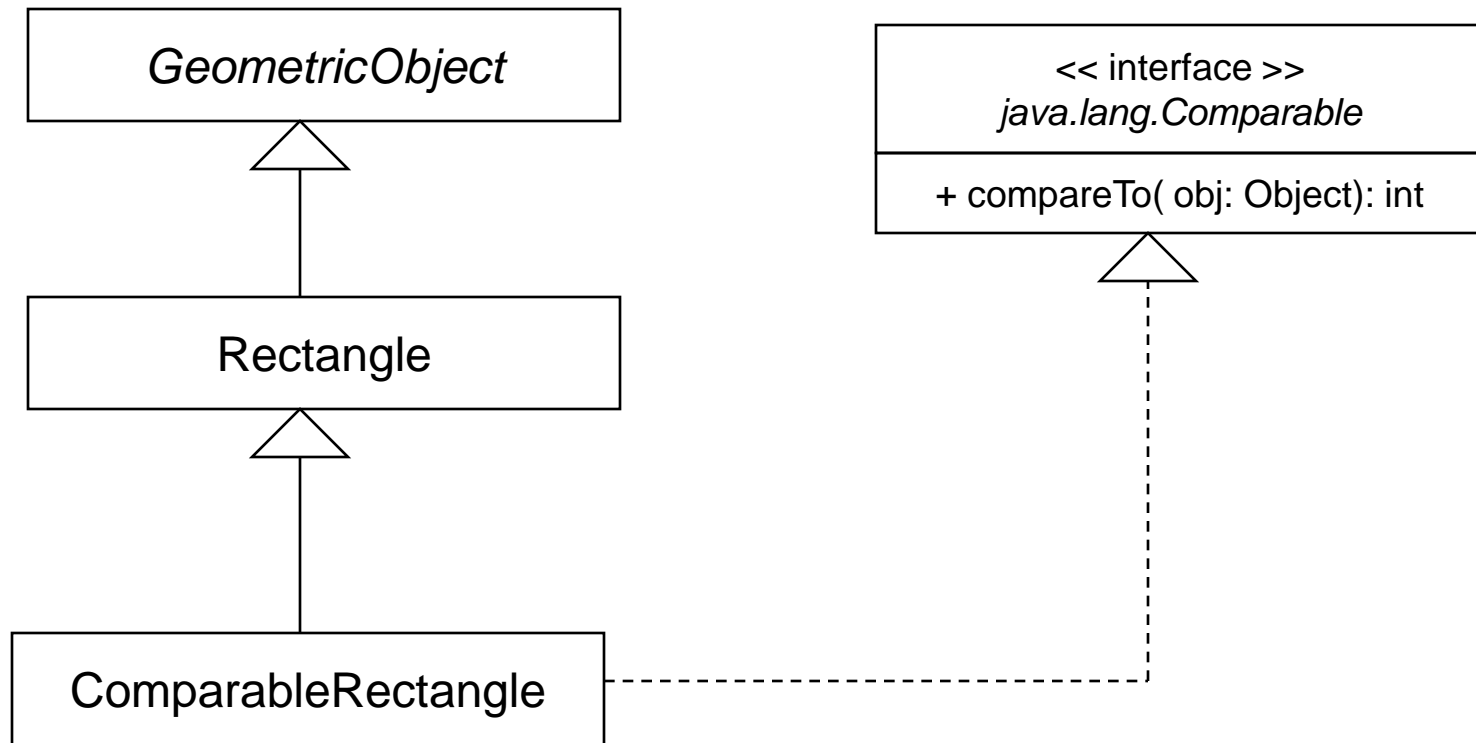
Using MaxVersion1 class and max method

Using compareTo method in ComparableRectangle class





# UML Showing Interface



# The Comparable Interface

- An interface provides another form of generic programming.
- It would be difficult to use a generic `max` method to find the maximum of the objects without using an interface in our example. Why?
- Because it would be necessary to inherit the `Comparable` and another class, such as `Rectangle`, at the same time, which involves multiple inheritance, which is not allowed in Java.



# The Comparable Interface

- The `Object` class contains the `equals` method, which is intended for the subclasses of the `Object` class to override in order to compare whether the contents of the objects are the same.
- Suppose that the `Object` class contains the `compareTo` method, as defined in the `Comparable` interface: the new `max` method can be used to compare a list of any objects.



# The Comparable Interface

- Whether a `compareTo` method should be included in the `Object` class is debatable. Since the `compareTo` method is not defined in the `Object` class, the `Comparable` interface is created in Java to enable objects to be compared if they are instances of the `Comparable` interface.
- It is strongly recommended that `compareTo` should be consistent with `equals`. That is, for two objects `obj1` and `obj2`, `obj1.compareTo(obj2) == 0`, iff `obj1.equals(obj2)` is true.



# Interfaces vs. Abstract Classes

- An interface can be used in the same way as an abstract class, but declaring an interface is different from describing an abstract class. The table below summarizes the differences.

|                | Variables                                              | Constructors                                                                                                                                       | Methods                                                           |
|----------------|--------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------|
| Abstract class | No restrictions                                        | Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the <code>new</code> operator. | No restrictions                                                   |
| Interface      | All variables must be <code>public static final</code> | No constructors. An interface cannot be instantiated using the <code>new</code> operator.                                                          | All methods must be <code>public abstract</code> instance methods |



# Interfaces vs. Abstract Classes

- Java allows only single inheritance for class extension, but multiple extensions for interfaces.
  - For example,

```
public class NewClass extends BaseClass
 implements Interface1, Interface2, . . . ,
 InterfaceN {
 . . .
}
```



# Interfaces vs. Abstract Classes

- An interface can inherit other interfaces using the `extends` keyword. Such an interface is called a **subinterface**.

– For example,

```
public interface NewInterface extends
 Interface1, . . ., InterfaceN {
 //constants and abstract methods
 . . .
}
```

- A class implementing `NewInterface` must implement the abstract methods defined in `NewInterface`, `Interface1`, ... and `InterfaceN`.



# Interfaces vs. Abstract Classes

- An interface can extend other interfaces but not classes.
- A class can extend its superclass and implement multiple interfaces.
- All classes share a single root, the `Object` class, but there is no single root for interfaces.
- Like a class, an interface also defines a type. A variable of an interface type can reference any instance of the class that implements the interface. If a class implements an interface, the interface is like a superclass for the class.



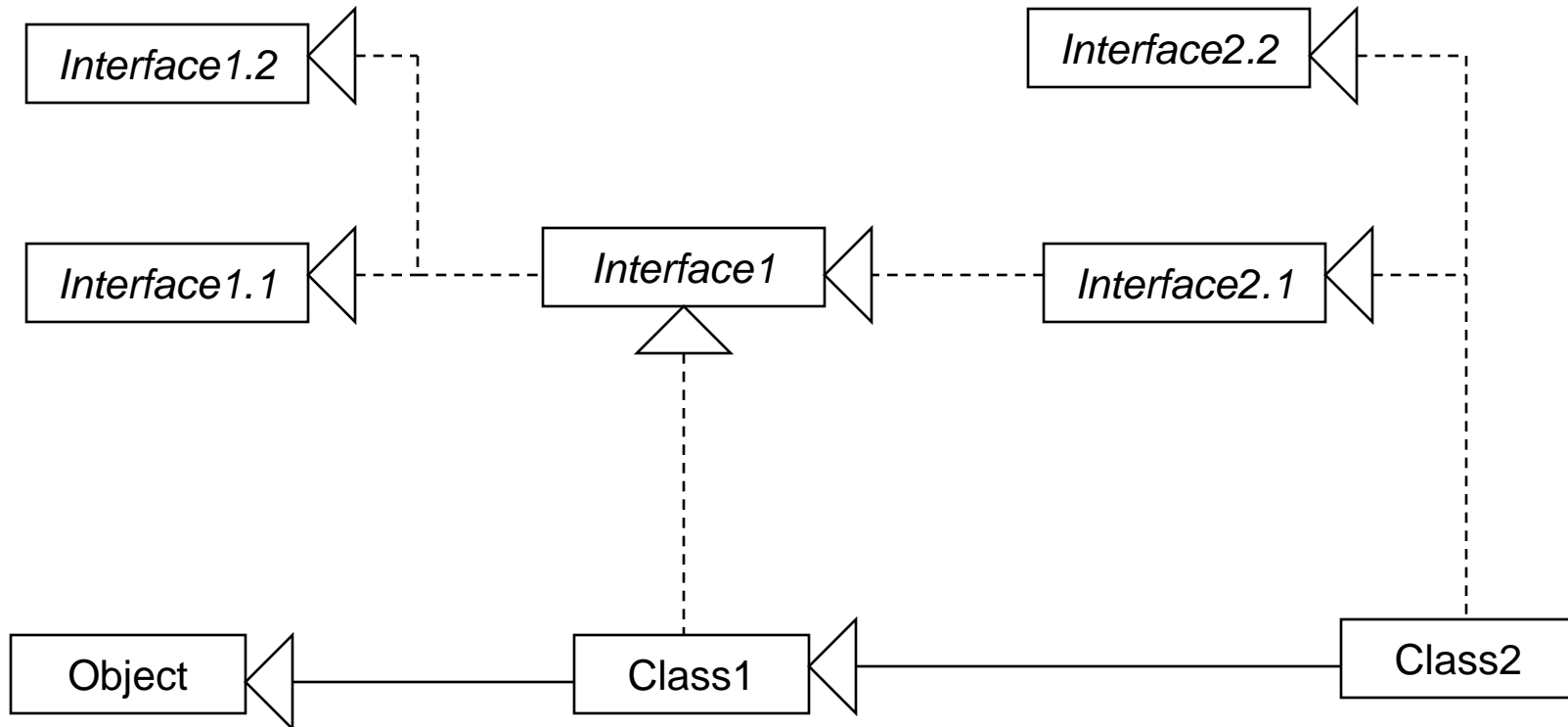


# Interfaces vs. Abstract Classes

- You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa.
  - For example, suppose that `c` is an instance of `Class2` (see the diagram on the next page). Then `c` is also an instance of `Object`, `Class1`, `Interface1`, `Interface1.1`, `Interface1.2`, `Interface2.1` and `Interface2.2`.
- Class names are nouns, interface names can be nouns or adjectives.



# Interfaces vs. Abstract Classes



Class1 implements Interface1. Interface1 extends Interface1.1 and Interface1.2. Class2 extends Class1 and implements Interface2.1 and Interface2.2



# Interfaces vs. Abstract Classes

- Abstract methods and interfaces can both be used to specify common behavior of objects. How do you decide whether to use an interface or a class?
- In general, a strong *is-a* relationship that clearly describes a parent-child relationship should be modeled using classes.
- A weak *is-a* relationship, also known as an *is-kind-of* relationship, indicates that an object possesses a certain property. A weak *is-a* relationship can be modeled using interfaces.
- In general, interfaces are preferred over classes because an interface can represent a common supertype for unrelated classes.

