COP 3330: Object-Oriented Programming Summer 2008

Classes In Java – Part 3

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop3330/sum2008

School of Electrical Engineering and Computer Science
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.

```
public class Rectangle extends GeometricObject {
  private double width;
  private double height;
  public Rectangle() {
  }
  public Rectangle(double width, double height) {
    this.width = width;
    this.height = height;
  }
  /** Return width */
  public double getWidth() {
    return width;
  }
  /** Set a new width */
  public void setWidth(double width) {
    this.width = width;
  }
  /** Return height */
  public double getHeight() {
    return height;
  }
  /** Set a new height */
  public void setHeight(double height) {
    this.height = height;
  }
  /** Return area */
  public double getArea() {
    return width * height;
  }
  /** Return perimeter */
  public double getPerimeter() {
    return 2 * (width + height);
  }
}
```

The original Rectangle class

```
public class Circle extends GeometricObject {
  private double radius;
  public Circle() {
  }
  public Circle(double radius) {
    this.radius = radius;
  }
  /** Return radius */
  public double getRadius() {
    return radius;
  }
  /** Set a new radius */
  public void setRadius(double radius) {
    this.radius = radius;
  }
  /** Return diameter */
  public double getDiameter() {
    return 2 * radius;
  }
  /** Return area */
  public double getArea() {
    return radius * radius * Math.PI;
  }
  /** Return perimeter */
  public double getPerimeter() {
    return 2 * radius * Math.PI;
  }
  /* Print the circle info */
  public void printCircle() {
    System.out.println("The circle is created " + getDateCreated() +
      " and the radius is " + radius);
  }
}
```

The original Circle class

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

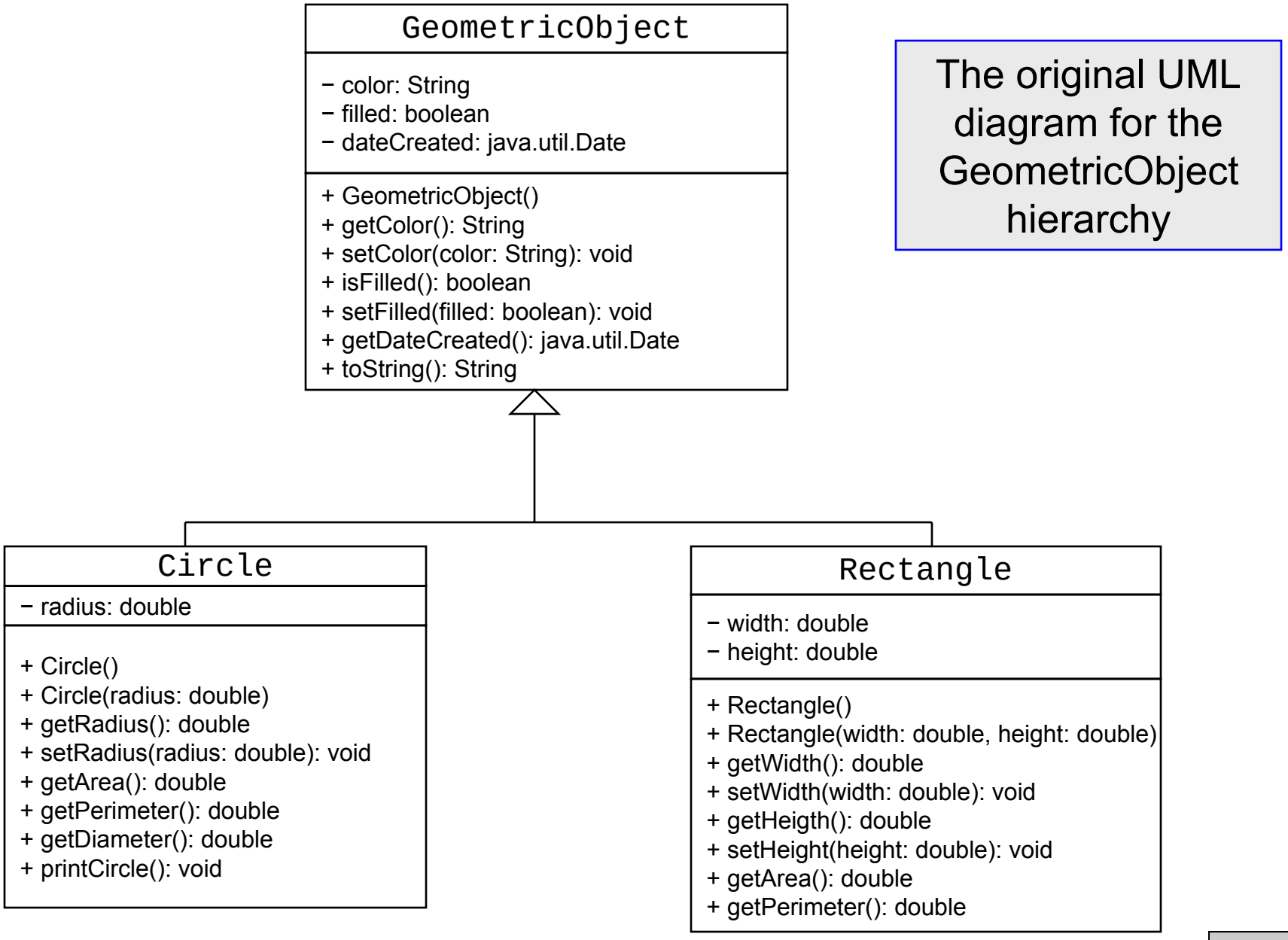
- An abstract method is specified using the `abstract` modifier in the method header.
  - Example:

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

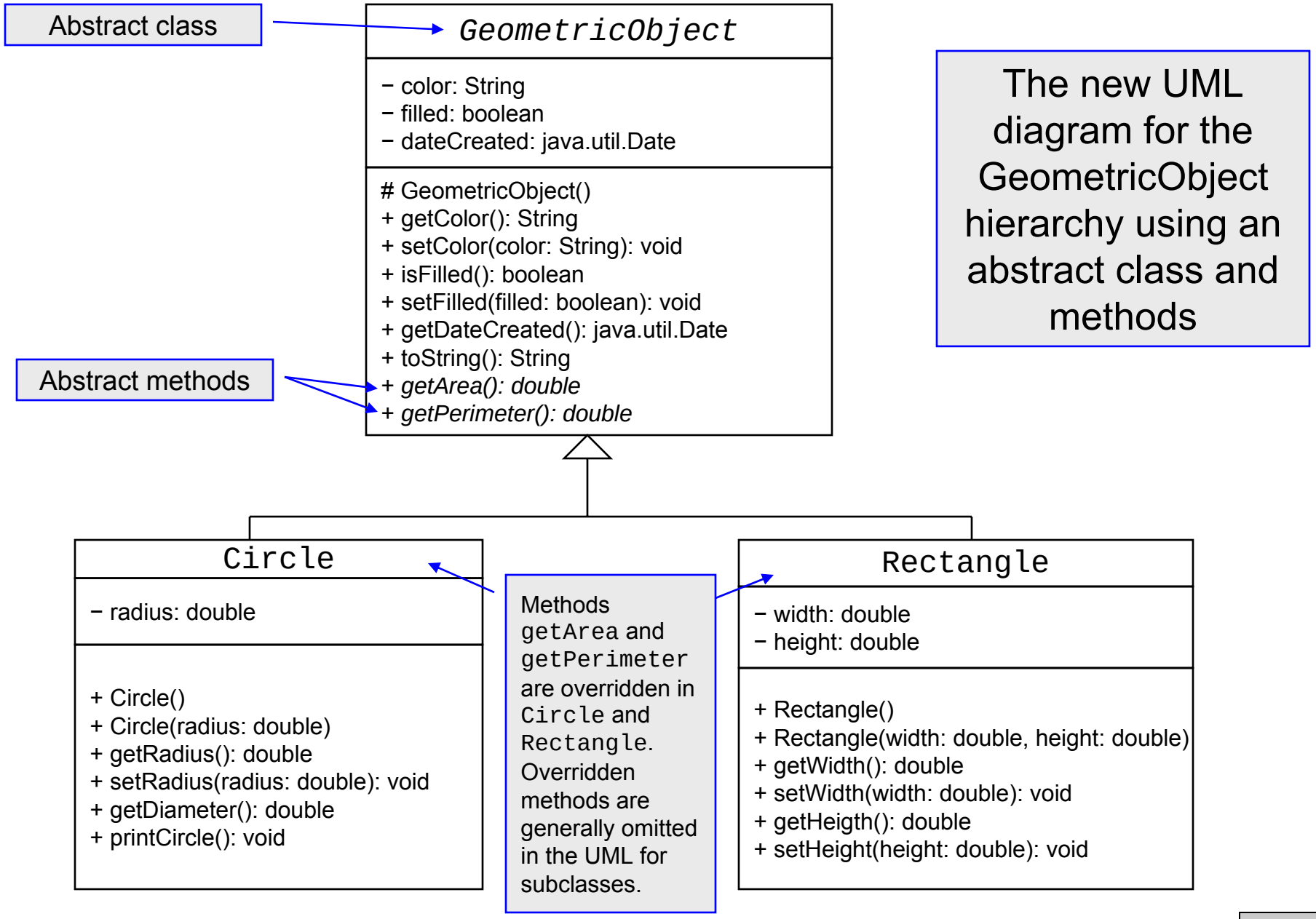
- Similarly, an abstract class is denoted using the `abstract` modifier in the class header.
  - Example:
  - `public abstract class GeometricObject { . . . }`
- In UML notation, the names of abstract classes and their methods are italicized (see pages 9-10 of Introduction to Object-Oriented Programming - Part 3 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
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
+ getArea(): double
+ getPerimeter(): double
Abstract methods
The new UML diagram for the GeometricObject hierarchy using an abstract class and methods
Circle
− radius: double
+ Circle()
+ Circle(radius: double)
+ getRadius(): double
+ setRadius(radius: double): void
+ getDiameter(): double
+ printCircle(): void
Methods getArea and getPerimeter are overridden in Circle and Rectangle. Overridden methods are generally omitted in the UML for subclasses.
Rectangle
− width: double
− height: double
+ Rectangle()
+ Rectangle(width: double, height: double)
+ getWidth(): double
+ setWidth(width: double): void
+ getHeigth(): double
+ setHeight(height: double): void

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!)

```
public abstract class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

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

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

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

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

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

Declare class as abstract

Denote constructor as protected

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

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

  /** Abstract method getArea */
  public abstract double getArea();

  /** Abstract method getPerimeter */
  public abstract double getPerimeter();
}
```

Declare abstract method getArea()

Declare abstract method getPerimeter()

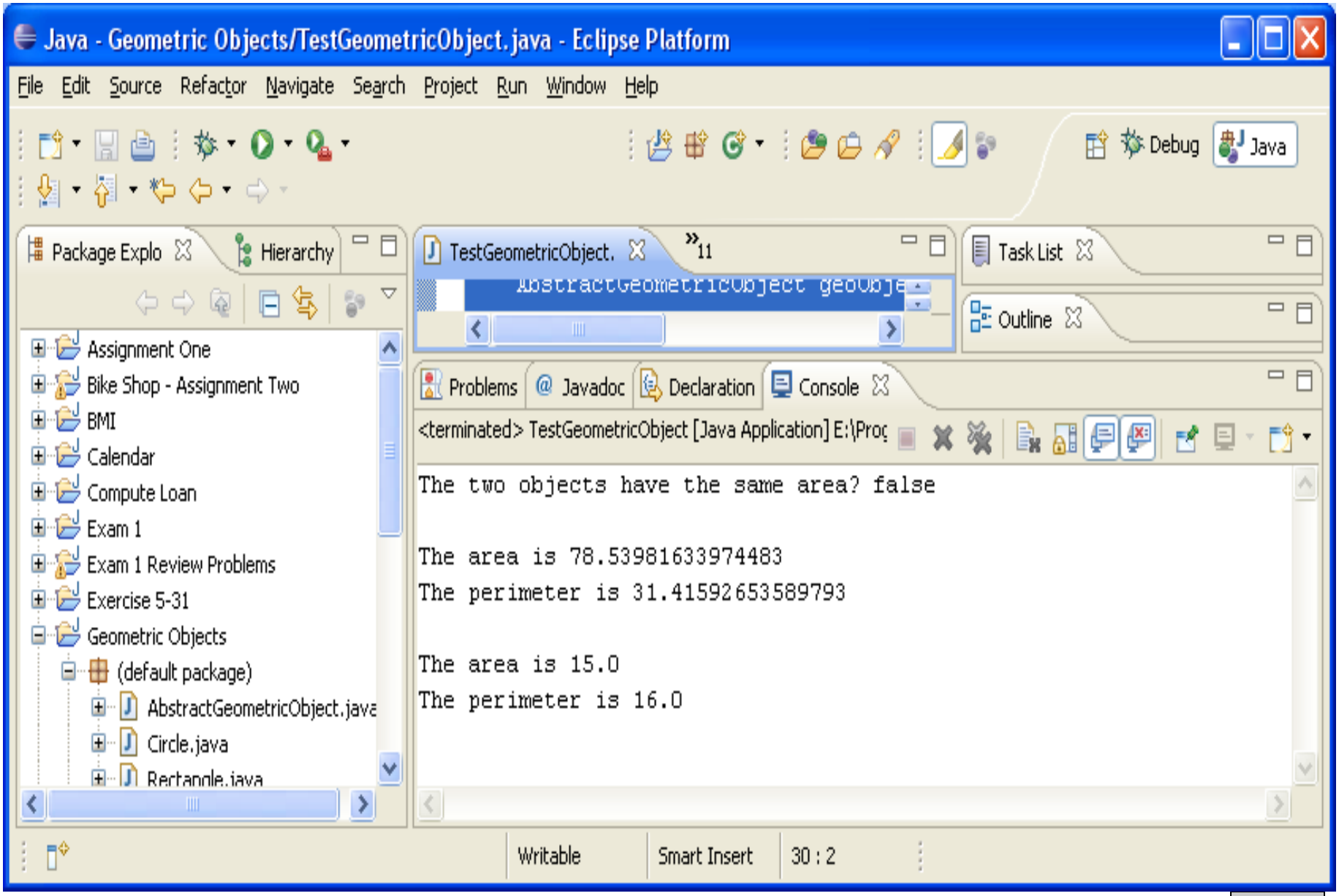
Abstract Classes

- 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` where 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 TestGeometricObject {
  /** Main method */
  public static void main(String[] args) {
    // Declare and initialize two geometric objects
    GeometricObject geoObject1 = new Circle(5); //implicit casting
    GeometricObject geoObject2 = new Rectangle(5, 3); //implicit casting

    System.out.println("The two objects have the same area? " +
      equalArea(geoObject1, geoObject2));
    // Display circle
    displayGeometricObject(geoObject1);

    // Display rectangle
    displayGeometricObject(geoObject2);
  }
  /** A method for comparing the areas of two geometric objects */
  public static boolean equalArea(GeometricObject object1,
      GeometricObject object2) {
    return object1.getArea() == object2.getArea();
  }
  /** A method for displaying a geometric object */
  public static void displayGeometricObject(GeometricObject object) {
    System.out.println();
    System.out.println("The area is " + object.getArea());
    System.out.println("The perimeter is " + object.getPerimeter());
  }
}
```



Java - Geometric Objects/TestGeometricObject.java - Eclipse Platform
File Edit Source Refactor Navigate Search Project Run Window Help
Debug
Java
Package Explo
Hierarchy
TestGeometricObject.
Task List
Outline
Assignment One
Bike Shop - Assignment Two
BMI
Calendar
Compute Loan
Exam 1
Exam 1 Review Problems
Exercise 5-31
Geometric Objects
(default package)
AbstractGeometricObject.java
Circle.java
Problems
Javadoc
Declaration
Console
<terminated> TestGeometricObject [Java Application] E:\Proc
The two objects have the same area? false
The area is 78.53981633974483
The perimeter is 31.41592653589793
The area is 15.0
The perimeter is 16.0
Writable
Smart Insert
30 : 2

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 in the constructors of its subclasses. For example, the constructors of `GeometricObject` are invoked in the `Circle` and `Rectangle` classes.

Interesting Points On Abstract Classes

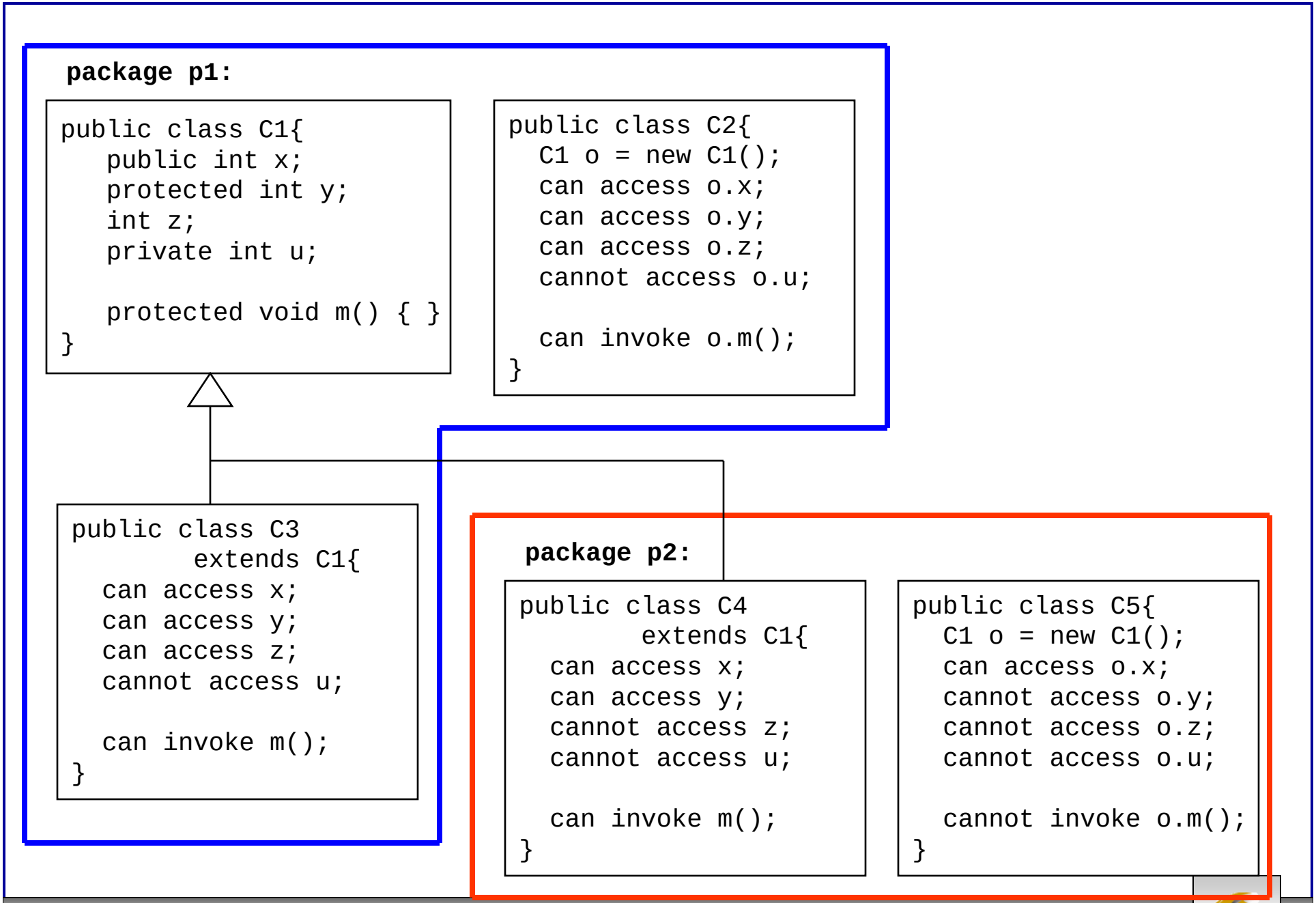
- A class that contains any abstract methods, must be declared as abstract. However, it is possible to declare an abstract class that contains no abstract methods! In this case, you cannot create instances of the class using the `new` operator. This class would be used as a base class for defining a 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.
- 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
`public`	yes	yes	yes	yes
`protected`	yes	yes	yes	no
`(default)`	yes	yes	no	no
`private`	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 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.