Object Oriented programming

Chapter 11

INHERITANCE

AND

POLYMORPHISM

Instructors:

Dr. Rasha Orban

Dr. Mustafa Abdul Salam

INHERITANCE AND POLYMORPHISM

Object oriented programming allows you to derive new classes from existing classes. This is called *inheritance.*

Inheritance is an important and powerful feature in Java for reusing software.

Suppose you are to define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so to avoid redundancy and make the system easy to comprehend and easy to maintain? The answer is to use inheritance.

Superclasses and Subclasses

You use a class to model objects of the same type. Different classes may have some common properties and behaviors, which can be generalized in a class that can be shared by other classes. Inheritance enables you to define a general class and later extend it to more specialized classes. The specialized classes inherit the properties and methods from the general class.

In Java terminology, a class C1 extended from another class C2 is called a *subclass*, *and* C2 is called a *superclass*

A superclass is also referred to as a *parent class, or a base class*, *and a* subclass as a *child class, an extended class, or a derived class*. *A subclass inherits accessible* data fields and methods from its superclass and may also *add new data fields and methods.*

The Circle class extends the GeometricObject class (Listing 11.2) using the following syntax:

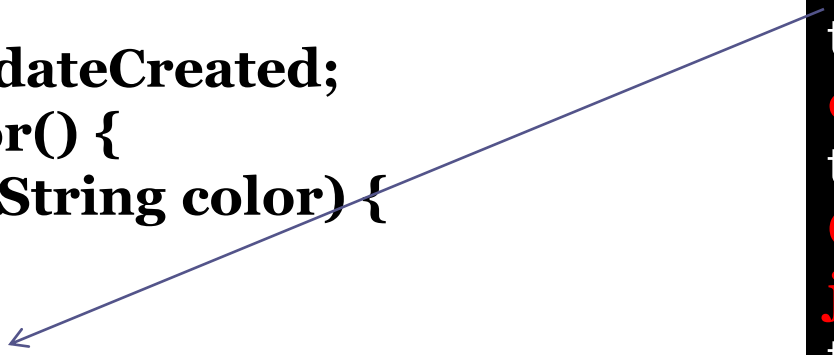
```
public class Circle extends GeometricObject
```

LISTING 11.1 GeometricObject1.java

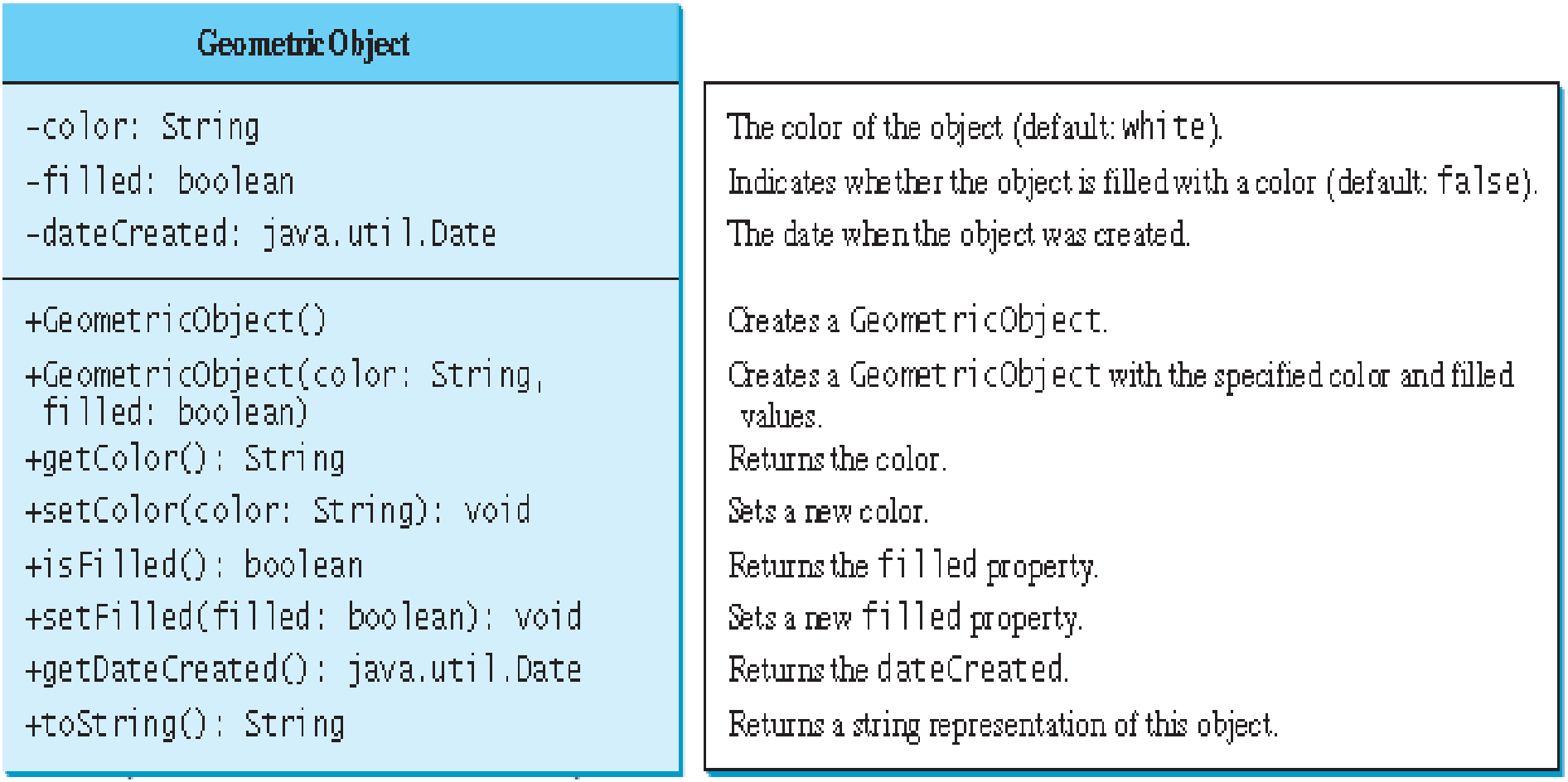
```
1 public class GeometricObject1 {
2 private String color = "white";
3 private boolean filled;
4 private java.util.Date dateCreated;
20 public String getColor() {
25 public void setColor(String color) {
```

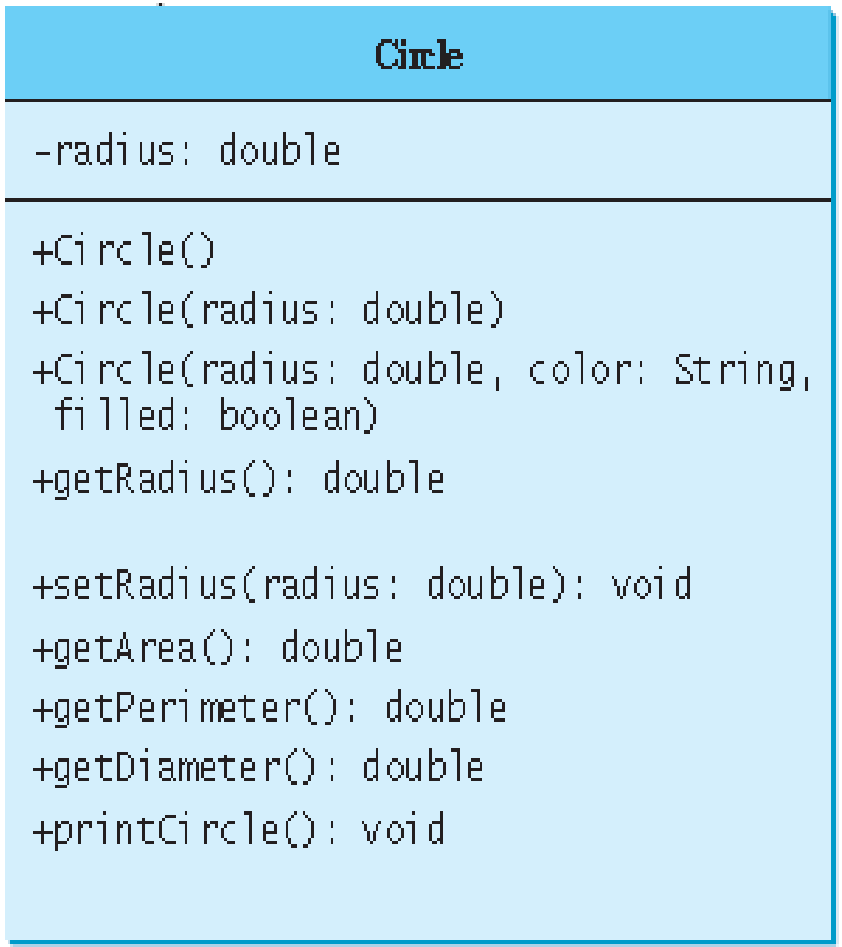
LISTING 11.2 Circle4.java

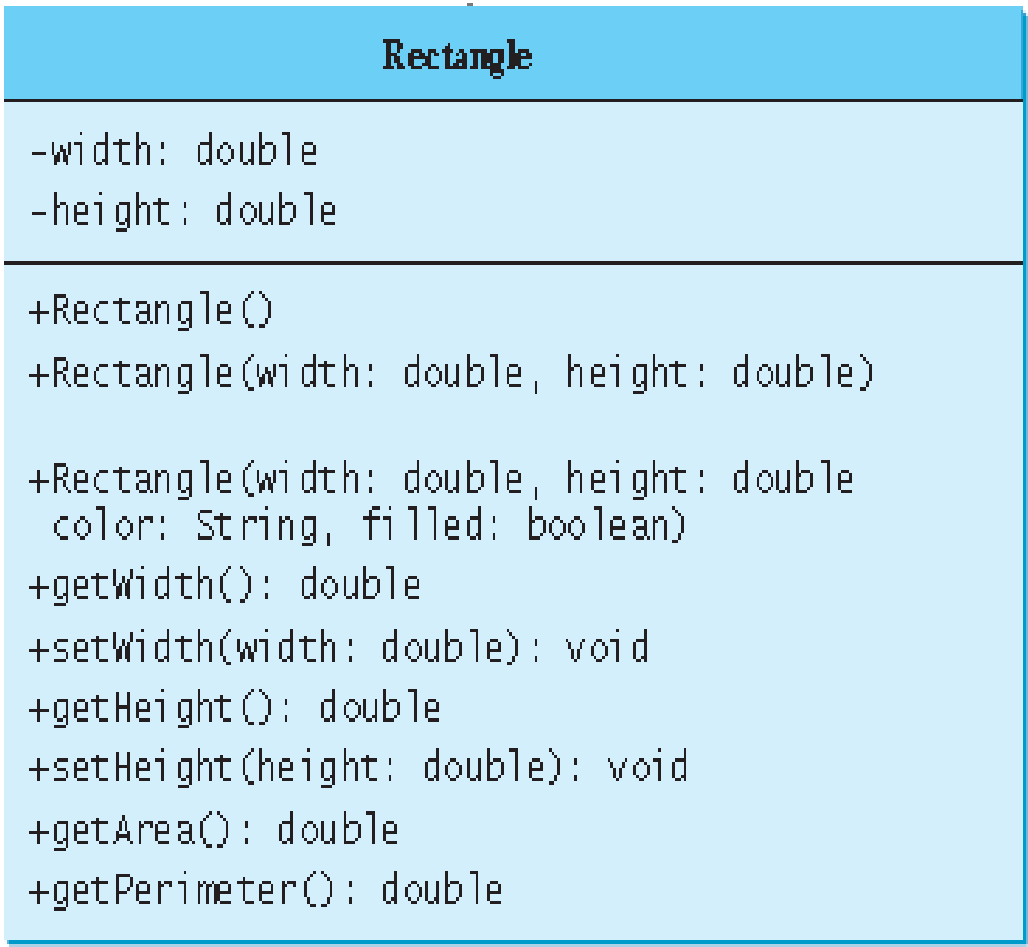
```
1 public class Circle4 extends GeometricObject1 {
2 private double radius;
11 public Circle4(double radius, String color, boolean filled
{
12 this.radius = radius;
13 setColor(color);
14 setFilled(filled);
15 }
```



The keyword extends (line 1) tells the compiler that the Circle class extends the GeometricObject class, thus inheriting the methods getColor, setColor, isFilled, setFilled, and toString.


GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.
Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.


Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+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)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

```
public Circle4(double radius, String
color, boolean filled) {
this.radius = radius;
this.color = color; // Illegal
this.filled = filled; // Illegal
}
```

because the private data fields color and filled in the Geometric-Object class cannot be accessed in any class other than in the GeometricObject class itself. The only way to read and modify color and filled is through their get and set methods.

The following points regarding inheritance are worthwhile to note:

- **a subclass is not a subset of its superclass (more in subclass),**
- **Private data fields in a superclass are not accessible outside the class,**
- **Some programming languages allow you to derive a subclass from several classes. This capability is known as multiple inheritance. Java, however, does not allow multiple inheritance. A Java class may inherit directly from only one superclass. This restriction is known as single inheritance.**

Using the super Keyword

A subclass inherits accessible data fields and methods from its superclass. Does it inherit constructors?

Can superclass constructors be invoked from subclasses?

The this Reference, the use of the keyword this to reference the calling object. The keyword super refers to the superclass of the class in which super appears. It can be used in two ways:

- **To call a superclass constructor.**
- **To call a superclass method.**

•The syntax to call a superclass's constructor is:
super(), or super(parameters);

•The statement super() invokes the no-arg constructor of its superclass,

and the statement super(arguments) invokes the superclass constructor

that matches the arguments.

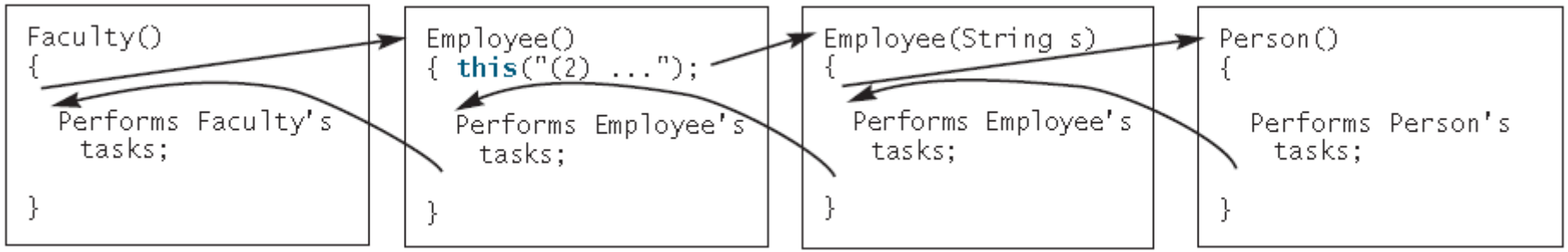
•The statement super() or super(arguments) must appear in the first line

of the subclass constructor; this is the only way to explicitly invoke a

superclass constructor.

Constructor Chaining

When constructing an object of a subclass, the subclass constructor first invokes its superclass constructor before performing its own tasks. If the superclass is derived from another class, the superclass constructor invokes its parent-class constructor before performing its own tasks. This process continues until the last constructor along the inheritance hierarchy is called. This is *constructor chaining*.



Calling Superclass Methods

The keyword super can also be used to reference a method other than the constructor in the superclass. The syntax is like this:

super.method(parameters);

```
public void printCircle() {
System.out.println("The circle is created " +
super. getDateCreated() + " and the radius is " +
radius);
}
```

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. This is referred to as *method overriding*.**
- The toString() method is defined in the GeometricObject class and modified in the Circle class.
- Both methods can be used in the Circle class. To invoke the toString method defined in the GeometricObject class from the Circle class, use super.toString()

- **Can a subclass of Circle access the toString method defined in the GeometricObject class using syntax such as super.super.toString()? No. This is a syntax error.**

Several points are worth noting:

- **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.**

■ **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.**

Overriding vs. Overloading

- You have learned about **overloading** methods in §5.8. Overloading means to define multiple methods with the same name but different signatures.
- **Overriding** means to provide a new implementation for a method in the subclass. The method is already defined in the superclass.
- To override a method, the method must be defined in the subclass using the **same signature and the same return type.**

In (a) below, the method p(double i) in class A overrides the same method defined in class B. In (b), however, the class B has two overloaded methods p(double i) and p(int i). The method p(double i) is inherited from B.

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

(a)

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

(b)

Note that

- Overridden methods are in different classes related by inheritance;
- overloaded methods can be either in the same class or different classes related by inheritance.
- Overridden methods have the same signature and return type;
- overloaded methods have the same name but a different parameter list.

The Object Class and Its toString() Method

Every class in Java is descended from the java.lang.Object class. If no inheritance is specified when a class is defined, the superclass of the class is Object by default. For example,

the following two class definitions are the same:

```
public class ClassName {
  ...
}
```

Equivalent

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

Usually you should override the toString method so that it returns a descriptive string representation of the object. For example, the toString method in the Object class was overridden in the GeometricObject class.

```
public String toString() {
return "created on " + dateCreated + "\ncolor: " +
color + " and filled: " + filled;
}
```

Polymorphism

First let us define two useful terms: subtype and supertype. A class defines a type. A type defined by a subclass is called a *subtype and a type defined by its superclass is called a supertype.*

So, you can say that Circle is a subtype of GeometricObject and GeometricObject is a supertype for Circle.

- **The inheritance relationship enables a subclass to inherit features from its superclass with additional new features.**
- **A subclass is a specialization of its superclass; every instance of a subclass is also an instance of its superclass, but not vice versa. For example, every circle is a geometric object, but not every geometric object is a circle.**
- **Therefore, you can always pass an instance of a subclass to a parameter of its superclass type. Consider the code**

LISTING 11.5 PolymorphismDemo.java

```
1  public class PolymorphismDemo {
2   /** Main method */
3   public static void main(String[] args) {
4   // Display circle and rectangle properties
5   displayObject(new Circle4(1, "red", false));
6   displayObject(new Rectangle1(1, 1, "black", true));
7   }
8
9   /** Display geometric object properties */
10   public static void displayObject(GeometricObject1 object) {
11   System.out.println("Created on " + object.getDateCreated()
+
12   ". Color is " + object.getColor());
13   }
14   }
```

Method displayObject (line 10) takes a parameter of the GeometricObject type. You can invoke displayObject by passing any instance of GeometricObject (e.g., new Circle4(1, "red", false) and new Rectangle1(1, 1, "black", false) in lines 5–6). An object of a subclass can be used wherever its superclass object is used. This is commonly known as *polymorphism* (*from a Greek word meaning "many forms"*). *In simple terms, polymorphism means that a variable of a supertype can refer to a subtype object*.

Dynamic Binding

A method may be defined in a superclass and overridden in its subclass. For example, the toString() method is defined in the Object class and overridden in GeometricObject. Consider the following code:

```
Object o = new GeometricObject();
System.out.println(o.toString());
```

Which toString() method is invoked by o?

To answer this question, we first introduce two terms: declared type and actual type. A variable must be declared a type. The type of a variable is called its *declared type*. *Here o's declared type is Object*. A variable of a reference type can hold a null value or a reference to an instance of the declared type.

The *actual type of the* variable is the actual class for the object referenced by the variable. Here o's actual type is GeometricObject, since o references to an object created using new GeometricObject(). Which toString() method is invoked by o is determined by o's actual type. This is known as *dynamic binding*.

Dynamic binding works as follows: Suppose an object o is an instance of classes C1, C2, Cn-1, and Cn, where C1 is a subclass of C2, C2 is a subclass of C3, and Cn-1 is a subclass of Cn. That is, Cn is the most general class, and C1 is the most specific class. In Java, Cn is the Object class. If o invokes a method p, the JVM searches the implementation for the method p in C1, C2, Cn-1, and Cn, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.

```
1 public class DynamicBindingDemo {
2 public static void main(String[] args) {
3 m(new GraduateStudent());
4 m(new Student());
5 m(new Person());
6 m(new Object());
7 }
8
9 public static void  m(Object x) {
10 System.out.println(x.toString());
11 }
12 }
13
14 class GraduateStudent extends Student {
15 }
16
17 class Student extends Person {
18 public String toString() {
19 return "Student";
20 }
21 }
22
23 class Person extends Object {
24 public String toString() {
25 return "Person";
26 }
27 }
```

```
Student
Student
Person
java.lang.Object@130c19b
```

Casting Objects and the instanceof operator

The statement `Object o = new Student()`, known as *implicit casting*, is legal because an instance of Student is automatically an instance of Object. Suppose you want to assign the object reference o to a variable of the Student type using the following statement:

```
Student b = o;
```

In this case a compile error would occur. Why does the statement `Object o = new Student()` work but `Student b = o` doesn't?

The reason is that a Student object is always an instance of Object, but an Object is not necessarily an instance of Student. Even though you can see that o is really a Student object, the compiler is not clever enough to know it. To tell the compiler that o is a Student object, use an *explicit casting.*

```
Student b = (Student)o; // Explicit casting
```

To ensure that the object is an instance of another object before attempting a casting. This can be accomplished by using the *instanceof operator*. *Consider the following code:*

```
... // Some lines of code
Object myObject = new Circle();
/** Perform casting if myObject is an instance of
Circle */
if (myObject instanceof Circle) {
System.out.println("The circle diameter is " +
((Circle)myObject).getDiameter());
...
}
```

You may be wondering why casting is necessary. Variable **myObject is declared Object.** The *declared type decides which method to match at compile time. Using* **myObject.getDiameter() would cause a compile error, because the Object class does** not have the **getDiameter method. The compiler cannot find a match for myObject.getDiameter(). It is necessary to cast myObject into the Circle type to tell** the compiler that **myObject is also an instance of Circle.**

Why not define **myObject as a Circle type in the first place? To enable generic programming,** it is a good practice to define a variable with a supertype, which can accept a value of any subtype.

The protected Data and Methods

• So far you have used the private and public keywords to specify whether data fields and methods can be accessed from the outside of the class.

• Private members can be accessed only from the inside of the class, and public members can be accessed from any other classes.

• Often it is desirable to allow subclasses to access data fields or methods defined in the superclass, but not allow nonsubclasses to access these data fields and methods. To do so, you can use the protected keyword. A protected data field or method in a superclass can be accessed in its subclasses.

The modifiers private, protected, and public are known as *visibility or accessibility modifiers because they specify how class and class members are accessed. The visibility of*

these modifiers increases in this order:

TABLE 11.2 Data and Methods Visibility

Modifier on members in a class	*Accessed from the same class*	*Accessed from the same package*	*Accessed from a subclass*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
(default)	✓	✓	–	–
private	✓	–	–	–

Preventing Extending and Overriding

- You may occasionally want to prevent classes from being extended. In such cases, use the final modifier to indicate that a class is final and cannot be a parent class.
- The Math class is a final class. The String, StringBuilder, and StringBuffer classes are also final classes. For example, the following class is final and cannot be extended:

```
public final class C {
// Data fields, constructors, and methods omitted
}
```

You also can define a method to be final; a final method cannot be overridden by its subclasses.

For example, the following method is final and cannot be overridden:

```
public class Test {
// Data fields, constructors, and methods omitted
public final void m() {
// Do something
}
}
```

Important Links:

http://www3.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html

http://education-portal.com/academy/topic/introduction-to-programming.html

http://examples.javacodegeeks.com/

http://www.javaworld.com/

http://www.javatpoint.com/

Assignment (5) Programming Exercises:

11.1

11.2

Quiz

11.1 What is the printout of running the class C in (a)? What problem arises in compiling the program in (b)?

```
class A {
  public A() {
    System.out.println(
      "A's no-arg constructor is invoked");
  }
}

class B extends A {
}

public class C {
  public static void main(String[] args) {
    B b = new B();
  }
}
```

(a)

```
class A {
  public A(int x) {
  }
}

class B extends A {
  public B() {
  }
}

public class C {
  public static void main(String[] args) {
    B b = new B();
  }
}
```

(b)

A's no-arg constructor is invoked

The default constructor of B attempts to invoke the default of constructor of A, but class A's default constructor is not defined.

Quiz

11.2 True or false?

1. A subclass is a subset of a superclass.

2. When invoking a constructor from a subclass, its superclass's no-arg constructor is always invoked.

3. You can override a private method defined in a superclass.

4. You can override a static method defined in a superclass.

Quiz

All false.

(1) A subclass is an extension of a superclass and normally contains more details information than its superclass.

(2) If a subclass's constructor explicitly invoke a superclass's constructor, the superclass's no-arg constructor is not invoked.

(3) You can only override accessible instance methods.

(4) You can only override accessible instance methods.

Quiz

11.11 Show the output of following program:

```
1 public class Test {
2 public static void main(String[] args) {
3 A a = new A(3);
4 }
5 }
6
7 class A extends B {
8 public A(int t) {
9 System.out.println("A's constructor is invoked");
10 }
11 }
12
13 class B {
14 public B() {
15 System.out.println("B's constructor is invoked");
16 }
17 }
```

Is the no-arg constructor of Object invoked when new A(3) is invoked?

Quiz

B's constructor is invoked

A's constructor is invoked

The default constructor of Object is invoked, when new A(3) is invoked. The Object's constructor is invoked before any statements in B's constructor are executed.

Quiz

11.3 Identify the problems in the following classes:

```
1 public class Circle {
2 private double radius;
3
4 public Circle(double radius) {
5 radius = radius;
6 }
7
8 public double getRadius() {
9 return radius;
10 }
11
12 public double getArea() {
13 return radius * radius * Math.PI;
14 }
15 }
16
17 class B extends Circle {
18 private double length;
19
20 B(double radius, double
length) {
21 Circle(radius);
22 length = length;
23 }
24
25 /** Override getArea() */
26 public double getArea() {
27 return getArea() * length;
28 }
29 }
```

Quiz

Quiz

The following lines are erroneous:

```
        {
Line 5:  radius = radius;   // Must use this.radius = radius
        }
        class B extends Circle (missing extends)
        {
Line 21: Circle(radius);    // Must use super(radius)
Line 22: length = length;          // Must use this.length =
length
        }
        public double getArea()
        {
Line 27: return getArea()*length;        // super.getArea()
        }
```

Thanks for Attention