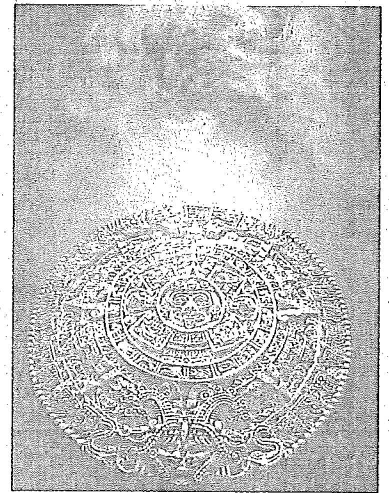
rovided
nethods

:r2 s);

CHAPTER 11

INHERITANCE AND POLYMORPHISM



Objectives

- To define a subclass from a superclass through inheritance (§11.2).
- To invoke the superclass's constructors and methods using the `super` keyword (§11.3).
- To override instance methods in the subclass (§11.4).
- To distinguish differences between overriding and overloading (§11.5).
- To explore the `toString()` method in the `Object` class (§11.6).
- To discover polymorphism and dynamic binding (§§11.7–11.8).
- To describe casting and explain why explicit downcasting is necessary (§11.9).
- To explore the `equals` method in the `Object` class (§11.10).
- To store, retrieve, and manipulate objects in an `ArrayList` (§11.11).
- To construct an array list from an array, to sort and shuffle a list, and to obtain max and min element from a list (§11.12).
- To implement a `Stack` class using `ArrayList` (§11.13).
- To enable data and methods in a superclass accessible from subclasses using the `protected` visibility modifier (§11.14).
- To prevent class extending and method overriding using the `final` modifier (§11.15).

11.1 Introduction

Object-oriented programming allows you to define new classes from existing classes. This is called inheritance.

inheritance

As discussed earlier in the book, the procedural paradigm focuses on designing methods and the object-oriented paradigm couples data and methods together into objects. Software design using the object-oriented paradigm focuses on objects and operations on objects. The object-oriented approach combines the power of the procedural paradigm with an added dimension that integrates data with operations into objects.

why inheritance?

Inheritance is an important and powerful feature for reusing software. Suppose you need 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 as to avoid redundancy and make the system easy to comprehend and easy to maintain? The answer is to use inheritance.

11.2 Superclasses and Subclasses

Inheritance enables you to define a general class (i.e., a superclass) and later extend it to more specialized classes (i.e., 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. You can define a specialized class that extends the generalized class. The specialized classes inherit the properties and methods from the general class.

VideoNote
Geometric class hierarchy

Consider geometric objects. Suppose you want to design the classes to model geometric objects such as circles and rectangles. Geometric objects have many common properties and behaviors. They can be drawn in a certain color and be filled or unfilled. Thus a general class `GeometricObject` can be used to model all geometric objects. This class contains the properties `color` and `filled` and their appropriate getter and setter methods. Assume that this class also contains the `dateCreated` property and the `getDateCreated()` and `toString()` methods. The `toString()` method returns a string representation of the object. Since a circle is a special type of geometric object, it shares common properties and methods with other geometric objects. Thus it makes sense to define the `Circle` class that extends the `GeometricObject` class. Likewise, `Rectangle` can also be defined as a subclass of `GeometricObject`. Figure 11.1 shows the relationship among these classes. A triangular arrow pointing to the superclass is used to denote the inheritance relationship between the two classes involved.

subclass
superclass

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 inherits all accessible data fields and methods from the `GeometricObject` class. In addition, it has a new data field, `radius`, and its associated getter and setter methods. The `Circle` class also contains the `getArea()`, `getPerimeter()`, and `getDiameter()` methods for returning the area, perimeter, and diameter of the circle.

The `Rectangle` class inherits all accessible data fields and methods from the `GeometricObject` class. In addition, it has the data fields `width` and `height` and their associated getter and setter methods. It also contains the `getArea()` and `getPerimeter()` methods for returning the area and perimeter of the rectangle.

The `GeometricObject`, `Circle`, and `Rectangle` classes are shown in Listings 11.1, 11.2, and 11.3.

avoid naming conflicts

Note

To avoid a naming conflict with the improved `GeometricObject`, `Circle`, and `Rectangle` classes introduced in Chapter 13, we'll name these classes

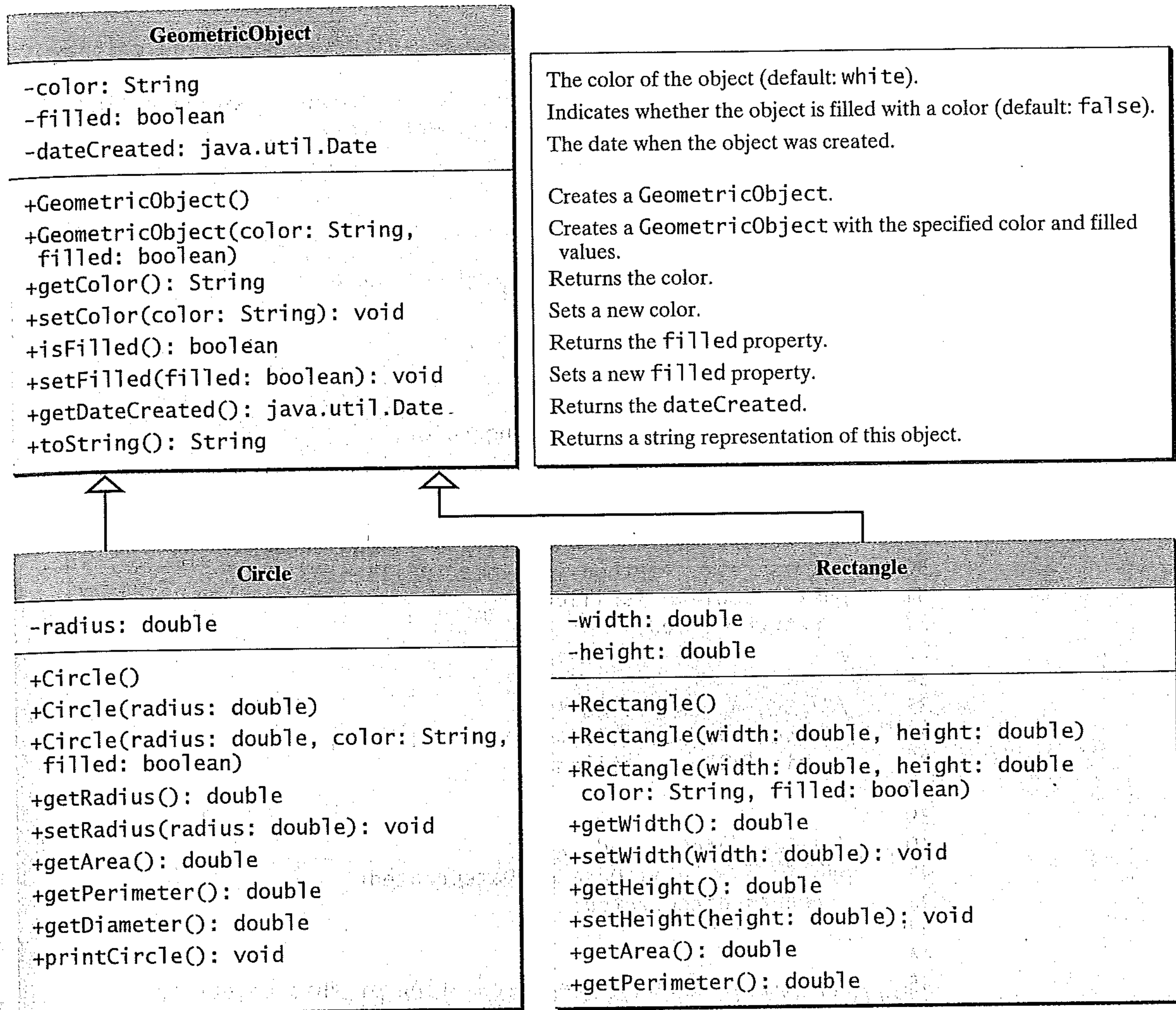


FIGURE 11.1 The **GeometricObject** class is the superclass for **Circle** and **Rectangle**.

SimpleGeometricObject, **CircleFromSimpleGeometricObject**, and **RectangleFromSimpleGeometricObject** in this chapter. For simplicity, we will still refer to them in the text as **GeometricObject**, **Circle**, and **Rectangle** classes. The best way to avoid naming conflicts is to place these classes in different packages. However, for simplicity and consistency, all classes in this book are placed in the default package.

LISTING 11.1 SimpleGeometricObject.java

```
1 public class SimpleGeometricObject {
2   private String color = "white";              // data fields
3   private boolean filled;
4   private java.util.Date dateCreated;
5
6   /** Construct a default geometric object */
7   public SimpleGeometricObject() {               // constructor
8     dateCreated = new java.util.Date();          // date constructed
9   }
```

```
10
11    /** Construct a geometric object with the specified color
12      *  and filled value */
13    public SimpleGeometricObject(String color, boolean filled) {
14      dateCreated = new java.util.Date();
15      this.color = color;
16      this.filled = filled;
17    }
18
19    /** Return color */
20    public String getColor() {
21      return color;
22    }
23
24    /** Set a new color */
25    public void setColor(String color) {
26      this.color = color;
27    }
28
29    /** Return filled. Since filled is boolean,
30       its getter method is named isFilled */
31    public boolean isFilled() {
32      return filled;
33    }
34
35    /** Set a new filled */
36    public void setFilled(boolean filled) {
37      this.filled = filled;
38    }
39
40    /** Get dateCreated */
41    public java.util.Date getDateCreated() {
42      return dateCreated;
43    }
44
45    /** Return a string representation of this object */
46    public String toString() {
47      return "created on " + dateCreated + "\ncolor: " + color +
48        " and filled: " + filled;
49    }
50  }
```

LISTING 11.2 CircleFromSimpleGeometricObject.java

```
 1  public class CircleFromSimpleGeometricObject
 2      extends SimpleGeometricObject {                    // extends superclass
 3    private double radius;                               // data fields
 4
 5    public CircleFromSimpleGeometricObject() {           // constructor
 6    }
 7
 8    public CircleFromSimpleGeometricObject(double radius) {
 9      this.radius = radius;
10    }
11
12    public CircleFromSimpleGeometricObject(double radius,
13        String color, boolean filled) {
14      this.radius = radius;
15      setColor(color);
16      setFilled(filled);
```

```
17     }
18
19     /** Return radius */
20     public double getRadius() {
21       return radius;
22     }
23
24     /** Set a new radius */
25     public void setRadius(double radius) {
26       this.radius = radius;
27     }
28
29     /** Return area */
30     public double getArea() {
31       return radius * radius * Math.PI;
32     }
33
34     /** Return diameter */
35     public double getDiameter() {
36       return 2 * radius;
37     }
38
39     /** Return perimeter */
40     public double getPerimeter() {
41       return 2 * radius * Math.PI;
42     }
43
44     /** Print the circle info */
45     public void printCircle() {
46       System.out.println("The circle is created " + getDateCreated() +
47         " and the radius is " + radius);
48     }
49   }
```

methods

The **Circle** class (Listing 11.2) extends the **GeometricObject** class (Listing 11.1) using the following syntax:

```
      Subclass                  Superclass

public class Circle extends GeometricObject
```

The keyword **extends** (lines 1–2) tells the compiler that the **Circle** class extends the **GeometricObject** class, thus inheriting the methods **getColor**, **setColor**, **isFilled**, **setFilled**, and **toString**.

The overloaded constructor **Circle(double radius, String color, boolean filled)** is implemented by invoking the **setColor** and **setFilled** methods to set the **color** and **filled** properties (lines 12–17). These two public methods are defined in the superclass **GeometricObject** and are inherited in **Circle**, so they can be used in the **Circle** class.

private member in superclass

You might attempt to use the data fields **color** and **filled** directly in the constructor as follows:

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

This is wrong, because the private data fields **color** and **filled** in the **GeometricObject** 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 getter and setter methods.

The **Rectangle** class (Listing 11.3) extends the **GeometricObject** class (Listing 11.1) using the following syntax:

```
            Subclass               Superclass

public class Rectangle extends GeometricObject
```

The keyword **extends** (lines 1–2) tells the compiler that the **Rectangle** class extends the **GeometricObject** class, thus inheriting the methods **getColor**, **setColor**, **isFilled**, **setFilled**, and **toString**.

LISTING 11.3 RectangleFromSimpleGeometricObject.java

```
 1 public class RectangleFromSimpleGeometricObject            // extends superclass
 2     extends SimpleGeometricObject {
 3   private double width;                                    // data fields
 4   private double height;
 5
 6   public RectangleFromSimpleGeometricObject() {            // constructor
 7   }
 8
 9   public RectangleFromSimpleGeometricObject(
10       double width, double height) {
11     this.width = width;
12     this.height = height;
13   }
14
15   public RectangleFromSimpleGeometricObject(
16       double width, double height, String color, boolean filled) {
17     this.width = width;
18     this.height = height;
19     setColor(color);
20     setFilled(filled);
21   }
22
23   /** Return width */
24   public double getWidth() {                               // methods
25     return width;
26   }
27
28   /** Set a new width */
29   public void setWidth(double width) {
30     this.width = width;
31   }
32
33   /** Return height */
34   public double getHeight() {
35     return height;
36   }
37
38   /** Set a new height */
39   public void setHeight(double height) {
40     this.height = height;
41   }
```

```
42
43     /** Return area */
44     public double getArea() {
45       return width * height;
46     }
47
48     /** Return perimeter */
49     public double getPerimeter() {
50       return 2 * (width + height);
51     }
52   }
```

The code in Listing 11.4 creates objects of `Circle` and `Rectangle` and invokes the methods on these objects. The `toString()` method is inherited from the `GeometricObject` class and is invoked from a `Circle` object (line 5) and a `Rectangle` object (line 13).

LISTING 11.4 TestCircleRectangle.java

```
 1 public class TestCircleRectangle {
 2   public static void main(String[] args) {
 3      CircleFromSimpleGeometricObject circle =
 4       new CircleFromSimpleGeometricObject(1);              // Circle object
 5     System.out.println("A circle " + circle.toString());   // invoke toString
 6     System.out.println("The color is " + circle.getColor()); // invoke getColor
 7     System.out.println("The radius is " + circle.getRadius());
 8     System.out.println("The area is " + circle.getArea());
 9     System.out.println("The diameter is " + circle.getDiameter());
10
11     RectangleFromSimpleGeometricObject rectangle =
12       new RectangleFromSimpleGeometricObject(2, 4);        // Rectangle object
13     System.out.println("\nA rectangle " + rectangle.toString()); // invoke toString
14     System.out.println("The area is " + rectangle.getArea());
15     System.out.println("The perimeter is " +
16       rectangle.getPerimeter());
17   }
18 }
```

```
A circle created on Thu Feb 10 19:54:25 EST 2011
color: white and filled: false
The color is white
The radius is 1.0
The area is 3.141592653589793
The diameter is 2.0
A rectangle created on Thu Feb 10 19:54:25 EST 2011
color: white and filled: false
The area is 8.0
The perimeter is 12.0
```

Note the following points regarding inheritance:

- Contrary to the conventional interpretation, a subclass is not a subset of its superclass. In fact, a subclass usually contains more information and methods than its superclass. *(more in subclass)*
- 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 accessors/mutators if defined in the superclass. *(private data fields)*

nonextensible is-a

- Not all is-a relationships should be modeled using inheritance. For example, a square is a rectangle, but you should not extend a **Square** class from a **Rectangle** class, because the **width** and **height** properties are not appropriate for a square. Instead, you should define a **Square** class to extend the **GeometricObject** class and define the **side** property for the side of a square.

no blind extension

- Inheritance is used to model the is-a relationship. 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 share common properties such as height and weight. A subclass and its superclass must have the is-a relationship.

multiple inheritance

single inheritance

- 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*. If you use the **extends** keyword to define a subclass, it allows only one parent class. Nevertheless, multiple inheritance can be achieved through interfaces, which will be introduced in Section 13.4.

11.1 True or false? A subclass is a subset of a superclass.

11.2 What keyword do you use to define a subclass?

11.3 What is single inheritance? What is multiple inheritance? Does Java support multiple inheritance?

11.3 Using the **super** Keyword

The keyword **super** *refers to the superclass and can be used to invoke the superclass's methods and constructors.*

A subclass inherits accessible data fields and methods from its superclass. Does it inherit constructors? Can the superclass's constructors be invoked from a subclass? This section addresses these questions and their ramifications.

Section 9.14, The **this** Reference, introduced 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.

11.3.1 Calling Superclass Constructors

A constructor is used to construct an instance of a class. Unlike properties and methods, the constructors of a superclass are not inherited by a subclass. They can only be invoked from the constructors of the subclasses using the keyword **super**.

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 be the first statement of the subclass's constructor; this is the only way to explicitly invoke a superclass constructor. For example, the constructor in lines 12–17 in Listing 11.2 can be replaced by the following code:

```
public CircleFromSimpleGeometricObject(
    double radius, String color, boolean filled) {
```

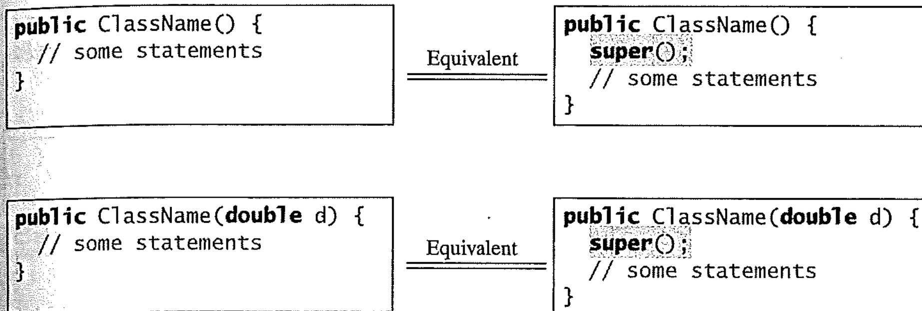
```
    super(color, filled);
    this.radius = radius;
  }
```

Caution

You must use the keyword **super** to call the superclass constructor, and the call must be the first statement in the constructor. Invoking a superclass constructor's name in a subclass causes a syntax error.

11.3.2 Constructor Chaining

A constructor may invoke an overloaded constructor or its superclass constructor. If neither is invoked explicitly, the compiler automatically puts **super()** as the first statement in the constructor. For example:



In any case, constructing an instance of a class invokes the constructors of all the superclasses along the inheritance chain. 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 called *constructor chaining*.

constructor chaining

Consider the following code:

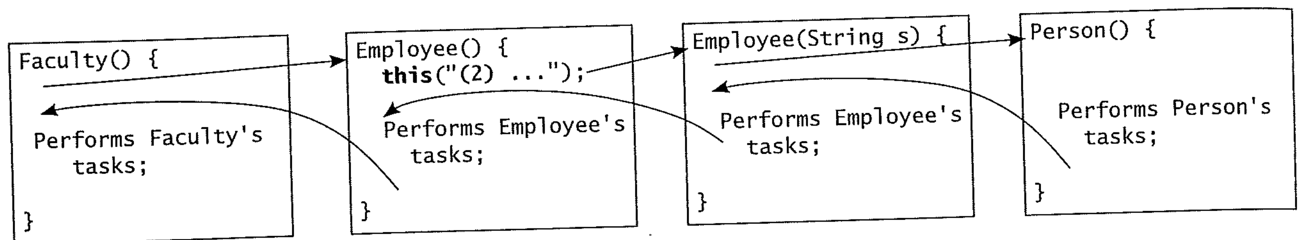
```
 1  public class Faculty extends Employee {
 2    public static void main(String[] args) {
 3      new Faculty();
 4    }
 5
 6    public Faculty() {
 7      System.out.println("(4) Performs Faculty's tasks");
 8    }
 9  }
10
11  class Employee extends Person {
12    public Employee() {
13      this("(2) Invoke Employee's overloaded constructor");
14      System.out.println("(3) Performs Employee's tasks ");
15    }
16
17    public Employee(String s) {
18      System.out.println(s);
19    }
20  }
21
22  class Person {
```

invoke overloaded constructor

```
23    public Person() {
24      System.out.println("(1) Performs Person's tasks");
25    }
26  }
```

```
(1) Performs Person's tasks
(2) Invoke Employee's overloaded constructor
(3) Performs Employee's tasks
(4) Performs Faculty's tasks
```

The program produces the preceding output. Why? Let us discuss the reason. In line 3, **new Faculty()** invokes **Faculty**'s no-arg constructor. Since **Faculty** is a subclass of **Employee**, **Employee**'s no-arg constructor is invoked before any statements in **Faculty**'s constructor are executed. **Employee**'s no-arg constructor invokes **Employee**'s second constructor (line 13). Since **Employee** is a subclass of **Person**, **Person**'s no-arg constructor is invoked before any statements in **Employee**'s second constructor are executed. This process is illustrated in the following figure.



no-arg constructor

Caution

If a class is designed to be extended, it is better to provide a no-arg constructor to avoid programming errors. Consider the following code:

```
1  public class Apple extends Fruit {
2  }
3
4  class Fruit {
5    public Fruit(String name) {
6      System.out.println("Fruit's constructor is invoked");
7    }
8  }
```

Since no constructor is explicitly defined in **Apple**, **Apple**'s default no-arg constructor is defined 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, because **Fruit** has an explicit constructor defined. Therefore, the program cannot be compiled.

no-arg constructor

Design Guide

If possible, you should provide a no-arg constructor for every class to make the class easy to extend and to avoid errors.

11.3.3 Calling Superclass Methods

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

```
super.method(parameters);
```

You could rewrite the `printCircle()` method in the `Circle` class as follows:

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

It is not necessary to put `super` before `getDateCreated()` in this case, however, because `getDateCreated` is a method in the `GeometricObject` class and is inherited by the `Circle` class. Nevertheless, in some cases, as shown in the next section, the keyword `super` is needed.

11.4 What is the output 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)

11.5 How does a subclass invoke its superclass's constructor?

11.6 True or false? When invoking a constructor from a subclass, its superclass's no-arg constructor is always invoked.

11.4 Overriding Methods

To override a method, the method must be defined in the subclass using the same signature and the same return type as in its superclass.

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*.

method overriding

The `toString` method in the `GeometricObject` class (lines 46–49 in Listing 11.1) returns the string representation of a geometric object. This method can be overridden to return the string representation of a circle. To override it, add the following new method in the `Circle` class in Listing 11.2.

```
1 public class CircleFromSimpleGeometricObject
2     extends SimpleGeometricObject {
3   // Other methods are omitted
4
5   // Override the toString method defined in the superclass
6   public String toString() {
7     return super.toString() + "\nradius is " + radius;
8   }
9 }
```

`toString` in superclass

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()` (line 7).

no super.super.methodName()

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:

override accessible instance method

- 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.

cannot override static method

- 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`.

Check Point

11.7 True or false? You can override a private method defined in a superclass.

11.8 True or false? You can override a static method defined in a superclass.

11.9 How do you explicitly invoke a superclass's constructor from a subclass?

11.10 How do you invoke an overridden superclass method from a subclass?

11.5 Overriding vs. Overloading

Key Point

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.

You learned about overloading methods in Section 6.8. To override a method, the method must be defined in the subclass using the same signature and the same return type.

Let us use an example to show the differences between overriding and overloading. In (a) below, the method `p(double i)` in class `A` overrides the same method defined in class `B`. In (b), however, the class `A` 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)

When you run the `Test` class in (a), both `a.p(10)` and `a.p(10.0)` invoke the `p(double i)` method defined in class `A` to display `10.0`. When you run the `Test` class in (b), `a.p(10)` invokes the `p(int i)` method defined in class `A` to display `10`, and `a.p(10.0)` invokes the `p(double i)` method defined in class `B` to display `20.0`.

Note the following:

- 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.

override annotation

To avoid mistakes, you can use a special Java syntax, called *override annotation*, to place `@Override` before the method in the subclass. For example:

toString in superclass

```
1  public class CircleFromSimpleGeometricObject
2      extends SimpleGeometricObject {
3    // Other methods are omitted
4
5    @Override
6    public String toString() {
7      return super.toString() + "\nradius is " + radius;
8    }
9  }
```

This annotation denotes that the annotated method is required to override a method in the superclass. If a method with this annotation does not override its superclass's method, the compiler will report an error. For example, if `toString` is mistyped as `tostring`, a compile error is reported. If the override annotation isn't used, the compile won't report an error. Using annotation avoids mistakes.

11.11 Identify the problems in the following code:

```
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
```

```
26   public double getArea() {
27     return getArea() * length;
28   }
29 }
```

11.12 Explain the difference between method overloading and method overriding.

11.13 If a method in a subclass has the same signature as a method in its superclass with the same return type, is the method overridden or overloaded?

11.14 If a method in a subclass has the same signature as a method in its superclass with a different return type, will this be a problem?

11.15 If a method in a subclass has the same name as a method in its superclass with different parameter types, is the method overridden or overloaded?

11.16 What is the benefit of using the `@Override` annotation?

11.6 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 {
  ...
}
```

Classes such as `String`, `StringBuilder`, `Loan`, and `GeometricObject` are implicitly subclasses of `Object` (as are all the main classes you have seen in this book so far). It is important to be familiar with the methods provided by the `Object` class so that you can use them in your classes. This section introduces the `toString` method in the `Object` class.

toString()

The signature of the `toString()` method is:

```
public String toString()
```

string representation

Invoking `toString()` on an object returns a string that describes the object. By default, it returns a string consisting of a class name of which the object is an instance, an at sign (@), and the object's memory address in hexadecimal. For example, consider the following code for the `Loan` class defined in Listing 10.2:

```
Loan loan = new Loan();
System.out.println(loan.toString());
```

The output for this code displays something like `Loan@15037e5`. This message is not very helpful or informative. 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 in lines 46–49 in Listing 11.1 as follows:

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

g. Is `orange instanceof Fruit`?

h. Is `orange instanceof Apple`?

i. Suppose the method `makeAppleCider` is defined in the `Apple` class. Can `fruit` invoke this method? Can `orange` invoke this method?

j. Suppose the method `makeOrangeJuice` is defined in the `Orange` class. Can `orange` invoke this method? Can `fruit` invoke this method?

k. Is the statement `Orange p = new Apple()` legal?

l. Is the statement `McIntosh p = new Apple()` legal?

m. Is the statement `Apple p = new McIntosh()` legal?

11.27 What is wrong in the following code?

```
1  public class Test {
2    public static void main(String[] args) {
3      Object fruit = new Fruit();
4      Object apple = (Apple)fruit;
5    }
6  }
7
8  class Apple extends Fruit {
9  }
10
11 class Fruit {
12 }
```

11.10 The Object's equals Method

Like the `toString()` *method, the* `equals(Object)` *method is another useful method defined in the* `Object` *class.*

Another method defined in the **Object** class that is often used is the **equals** method. Its signature is

```
public boolean equals(Object o)
```

This method tests whether two objects are equal. The syntax for invoking it is:

```
object1.equals(object2);
```

The default implementation of the **equals** method in the **Object** class is:

```
public boolean equals(Object obj) {
  return (this == obj);
}
```

This implementation checks whether two reference variables point to the same object using the == operator. You should override this method in your custom class to test whether two distinct objects have the same content.

The **equals** method is overridden in many classes in the Java API, such as **java.lang.String** and **java.util.Date**, to compare whether the contents of two objects are equal. You have already used the **equals** method to compare two strings in Section 4.4.7, The **String** Class. The **equals** method in the **String** class is inherited from the **Object** class and is overridden in the **String** class to test whether two strings are identical in content.

You can override the **equals** method in the **Circle** class to compare whether two circles are equal based on their radius as follows:

```
public boolean equals(Object o) {
  if (o instanceof Circle)
    return radius == ((Circle)o).radius;
  else
    return this == o;
}
```

== vs. equals

Note

The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The **equals** method is intended to test whether two objects have the same contents, provided that the method is overridden in the defining class of the objects. The == operator is stronger than the **equals** method, in that the == operator checks whether the two reference variables refer to the same object.

Caution

Using the signature **equals(SomeClassName obj)** (e.g., **equals(Circle c)**) to override the **equals** method in a subclass is a common mistake. You should use **equals(Object obj)**. See CheckPoint Question 11.29.

equals(Object)

11.28 Does every object have a **toString** method and an **equals** method? Where do they come from? How are they used? Is it appropriate to override these methods?

11.29 When overriding the **equals** method, a common mistake is mistyping its signature in the subclass. For example, the **equals** method is incorrectly written as **equals(Circle circle)**, as shown in (a) in following the code; instead, it should be **equals(Object circle)**, as shown in (b). Show the output of running class **Test** with the **Circle** class in (a) and in (b), respectively.

```
public class Test {
  public static void main(String[] args) {
    Object circle1 = new Circle();
    Object circle2 = new Circle();
    System.out.println(circle1.equals(circle2));
  }
}
```

```
class Circle {
  double radius;

  public boolean equals(Circle circle) {
    return this.radius == circle.radius;
  }
}
```

(a)

```
class Circle {
  double radius;

  public boolean equals(Object circle) {
    return this.radius ==
      ((Circle)circle).radius;
  }
}
```

(b)

If **Object** is replaced by **Circle** in the **Test** class, what would be the output to run **Test** using the **Circle** class in (a) and (b), respectively?

11.11 The ArrayList Class

VideoNote

The ArrayList class

An **ArrayList** *object can be used to store a list of objects.*

Now we are ready to introduce a very useful class for storing objects. You can create an array to store objects. But, once the array is created, its size is fixed. Java provides the **ArrayList**

remove

push

```
17
18   public Object pop() {
19     Object o = list.get(getSize() - 1);
20     list.remove(getSize() - 1);
21     return o;
22   }
23
24   public void push(Object o) {
25     list.add(o);
26   }
27
28   @Override
29   public String toString() {
30     return "stack: " + list.toString();
31   }
32 }
```

An array list is created to store the elements in the stack (line 4). The **isEmpty()** method (lines 6–8) returns **list.isEmpty()**. The **getSize()** method (lines 10–12) returns **list.size()**. The **peek()** method (lines 14–16) retrieves the element at the top of the stack without removing it. The end of the list is the top of the stack. The **pop()** method (lines 18–22) removes the top element from the stack and returns it. The **push(Object element)** method (lines 24–26) adds the specified element to the stack. The **toString()** method (lines 28–31) defined in the **Object** class is overridden to display the contents of the stack by invoking **list.toString()**. The **toString()** method implemented in **ArrayList** returns a string representation of all the elements in an array list.

Design Guide

composition
is-a
has-a

In Listing 11.10, **MyStack** contains **ArrayList**. The relationship between **MyStack** and **ArrayList** is *composition*. While inheritance models an *is-a* relationship, composition models a *has-a* relationship. You could also implement **MyStack** as a subclass of **ArrayList** (see Programming Exercise 11.10). Using composition is better, however, because it enables you to define a completely new stack class without inheriting the unnecessary and inappropriate methods from **ArrayList**.

11.14 The **protected** Data and Methods

A protected member of a class can be accessed from a subclass.

So far you have used the **private** and **public** keywords to specify whether data fields and methods can be accessed from outside of the class. Private members can be accessed only from inside of the class, and public members can be accessed from any other classes.

why protected?

Often it is desirable to allow subclasses to access data fields or methods defined in the superclass, but not to allow nonsubclasses to access these data fields and methods. To accomplish this, you can use the **protected** keyword. This way you can access protected data fields or methods in a superclass from its subclasses.

The modifiers **private**, **protected**, and **public** are known as *visibility* or *accessibility modifiers* because they specify how classes and class members are accessed. The visibility of these modifiers increases in this order:

Visibility increases
→
private, default (no modifier), protected, public

Table 11.2 summarizes the accessibility of the members in a class. Figure 11.5 illustrates how a public, protected, default, and private datum or method in class **C1** can be accessed from a class **C2** in the same package, from a subclass **C3** in the same package, from a subclass **C4** in a different package, and from a class **C5** in a different package.

Use the **private** modifier to hide the members of the class completely so that they cannot be accessed directly from outside the class. Use no modifiers (the default) 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.

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 in a different package*	*Accessed from a different package*
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default (no modifier)	✓	✓	–	–
private	✓	–	–	–

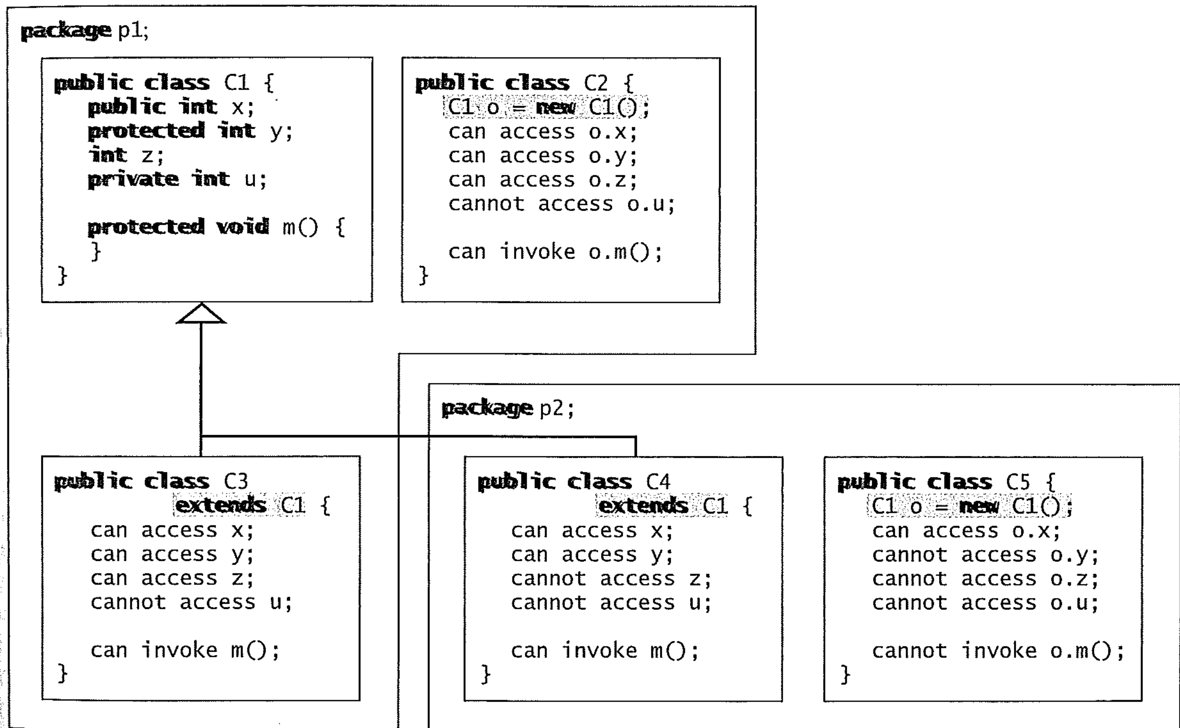


FIGURE 11.5 Visibility modifiers are used to control how data and methods are accessed.

Your class can be used in two ways: (1) for creating instances of the class and (2) for defining 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 for 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. A class with no modifier (i.e., not a public class) is not accessible by classes from other packages.

change visibility

Note

A subclass may override a protected method defined 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.

11.37 What modifier should you use on a class so that a class in the same package can access it, but a class in a different package cannot access it?

11.38 What modifier should you use so that a class in a different package cannot access the class, but its subclasses in any package can access it?

11.39 In the following code, the classes A and B are in the same package. If the question marks in (a) are replaced by blanks, can class B be compiled? If the question marks are replaced by **private**, can class B be compiled? If the question marks are replaced by **protected**, can class B be compiled?

```
package p1;

public class A {
  __?__ int i;

  __?__ void m() {
    ...
  }
}
```

(a)

```
package p1;

public class B extends A {
  public void m1(String[] args) {
    System.out.println(i);
    m();
  }
}
```

(b)

11.40 In the following code, the classes A and B are in different packages. If the question marks in (a) are replaced by blanks, can class B be compiled? If the question marks are replaced by **private**, can class B be compiled? If the question marks are replaced by **protected**, can class B be compiled?

```
package p1;

public class A {
  __?__ int i;

  __?__ void m() {
    ...
  }
}
```

(a)

```
package p2;

public class B extends A {
  public void m1(String[] args) {
    System.out.println(i);
    m();
  }
}
```

(b)

11.15 Preventing Extending and Overriding

Neither a final class nor a final method can be extended. A final data field is a constant.

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 A is final and cannot be extended:

```
public final class A {
    // 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 `m` is final and cannot be overridden:

```
public class Test {
  // Data fields, constructors, and methods omitted

  public final void m() {
    // Do something
  }
}
```

> **Note**
> The modifiers **`public`**, **`protected`**, **`private`**, **`static`**, **`abstract`**, and **`final`** are used on classes and class members (data and methods), except that the **`final`** modifier can also be used on local variables in a method. A **`final`** local variable is a constant inside a method.

11.41 How do you prevent a class from being extended? How do you prevent a method from being overridden?

11.42 Indicate true or false for the following statements:

a. A protected datum or method can be accessed by any class in the same package.

b. A protected datum or method can be accessed by any class in different packages.

c. A protected datum or method can be accessed by its subclasses in any package.

d. A final class can have instances.

e. A final class can be extended.

f. A final method can be overridden.

Key Terms

actual type 424
casting objects 427
constructor chaining 417
declared type 424
dynamic binding 424
inheritance 410
`instanceof` 428
is-a relationship 440
method overriding 419
multiple inheritance 416
override 419
polymorphism 423
`protected` 440
single inheritance 416
subclass 410
subtype 423
superclass 410
supertype 423
type inference 433

Chapter Summary

1. You can define a new class from an existing class. This is known as class *inheritance*. The new class is called a *subclass*, *child class*, or *extended class*. The existing class is called a *superclass*, *parent class*, or *base class*.

2. A constructor is used to construct an instance of a class. Unlike properties and methods, the constructors of a superclass are not inherited in the subclass. They can be invoked only from the constructors of the subclasses, using the keyword **`super`**.