



Object-Oriented Programming

Objectives

- To understand inheritance and software reusability.
- To understand superclasses and subclasses.
- To appreciate how polymorphism makes systems extensible and maintainable.
- To understand the distinction between abstract classes and concrete classes.
- To learn how to create abstract classes and interfaces.

Say not you know another entirely, till you have divided an inheritance with him.

Johann Kasper Lavater

This method is to define as the number of a class the class of all classes similar to the given class.

Bertrand Russell

Good as it is to inherit a library, it is better to collect one.

Augustine Birrell

General propositions do not decide concrete cases.

Oliver Wendell Holmes

A philosopher of imposing stature doesn't think in a vacuum. Even his most abstract ideas are, to some extent, conditioned by what is or is not known in the time when he lives.

Alfred North Whitehead



Outline

- 9.1 Introduction
- 9.2 Superclasses and Subclasses
- 9.3 `protected` Members
- 9.4 Relationship between Superclass Objects and Subclass Objects
- 9.5 Constructors and Finalizers in Subclasses
- 9.6 Implicit Subclass-Object-to-Superclass-Object Conversion
- 9.7 Software Engineering with Inheritance
- 9.8 Composition vs. Inheritance
- 9.9 Case Study: Point, Circle, Cylinder
- 9.10 Introduction to Polymorphism
- 9.11 Type Fields and `switch` Statements
- 9.12 Dynamic Method Binding
- 9.13 `final` Methods and Classes
- 9.14 Abstract Superclasses and Concrete Classes
- 9.15 Polymorphism Examples
- 9.16 Case Study: A Payroll System Using Polymorphism
- 9.17 New Classes and Dynamic Binding
- 9.18 Case Study: Inheriting Interface and Implementation
- 9.19 Case Study: Creating and Using Interfaces
- 9.20 Inner Class Definitions
- 9.21 Notes on Inner Class Definitions
- 9.22 Type-Wrapper Classes for Primitive Types
- 9.23 (Optional Case Study) Thinking About Objects: Incorporating Inheritance into the Elevator Simulation
- 9.24 (Optional) Discovering Design Patterns: Introducing Creational, Structural and Behavioral Design Patterns

Summary • Terminology • Self-Review Exercises • Answers to Self-Review Exercises • Exercises

9.1 Introduction

In this chapter, we discuss object-oriented programming (OOP) and its key component technologies—*inheritance* and *polymorphism*. Inheritance is a form of software reusability in which new classes are created from existing classes by absorbing their attributes and behaviors and adding new capabilities the new classes require. Inheritance takes advantage of *class* relationships where objects of a certain class—such as a class of vehicles—have the same characteristics. Newly created classes of objects are derived by absorbing characteristics of existing classes and adding unique characteristics of their own. An object of class “convertible” certainly has the characteristics of the more general class “automobile,” but a convertible’s roof goes up and down.

© Copyright 1992–2002 by Deitel & Associates, Inc. All Rights Reserved. 7/7/01

Software reusability saves time in program development. It encourages reuse of proven and debugged high-quality software, thus reducing problems after a system becomes operational. These are exciting possibilities. Polymorphism enables us to write programs in a general fashion to handle a wide variety of existing and yet-to-be-specified related classes. Polymorphism makes it easy to add new capabilities to a system. Inheritance and polymorphism are effective techniques for dealing with software complexity.

When creating a new class, instead of writing completely new instance variables and instance methods, the programmer can designate that the new class is to *inherit* the instance variables and instance methods of a previously defined *superclass*. The new class is referred to as a *subclass*. Each subclass itself becomes a candidate to be a superclass for some future subclass.

The *direct superclass* of a class is the superclass from which the class explicitly inherits (via the keyword **extends**). An indirect superclass is inherited from two or more levels up the class hierarchy. For example, class **JApplet** (package **javax.swing**) extends class **Applet** (package **java.applet**). Thus, each applet class we have defined is a direct subclass of **JApplet** and an indirect subclass of **Applet**.

With *single inheritance*, a class is derived from one superclass. Java does not support *multiple inheritance* (as C++ does) but it does support the notion of *interfaces*. Interfaces help Java achieve many of the advantages of multiple inheritance without the associated problems. We will discuss the details of interfaces in this chapter. We consider both general principles and a detailed specific example of creating and using interfaces.

A subclass normally adds instance variables and instance methods of its own, so a subclass is generally larger than its superclass. A subclass is more specific than its superclass and represents a smaller, more specialized group of objects. With single inheritance, the subclass starts out essentially the same as the superclass. The real strength of inheritance comes from the ability to define in the subclass additions to, or replacements for, the features inherited from the superclass.

Every subclass object is also an object of that class's superclass. For example, every applet we have defined is considered to be an object of class **JApplet**. Also, because **JApplet** extends **Applet**, every applet we have defined is considered to be an **Applet**. This information is critical when developing applets, because an applet container can execute a program only if it is an **Applet**. Although a subclass object always can be treated as one of its superclass types, superclass objects are not considered to be objects of their subclass types. We will take advantage of this "subclass-object-is-a-superclass-object" relationship to perform some powerful manipulations. For example, a drawing application can maintain a list of shapes to display. If all the shape types extend the same superclass directly or indirectly, the drawing program can store all the shapes in an array (or other data structure) of superclass objects. As we will see in this chapter, this ability to process a set of objects as a single type is a key thrust of object-oriented programming.

We add a new form of member access control in this chapter, namely **protected** access. Subclass methods and methods of other classes in the same package as the superclass can access **protected** superclass members.

Experience in building software systems indicates that significant portions of the code deal with closely related special cases. It becomes difficult in such systems to see the "big picture" because the designer and the programmer become preoccupied with the special cases. Object-oriented programming provides several ways of "seeing the forest through the trees."

The programmer and designer concentrate on the big picture—the commonality among objects in the system—rather than the special cases. This process is called *abstraction*.

If a procedural program has many closely related special cases, then it is common to see **switch** structures or nested **if/else** structures that distinguish among the special cases and provide the processing logic to deal with each case individually. We will show how to use inheritance and polymorphism to replace such **switch** logic with much simpler logic.

We distinguish between the “*is a*” relationship and the “*has a*” relationship. “Is a” is inheritance. In an “is a” relationship, an object of a subclass type may also be treated as an object of its superclass type. “Has a” is composition (as we discussed in Chapter 8). In a “has a” relationship, a class object has one or more objects of other classes as members. For example, a car *has a* steering wheel.

A subclass’s methods might need to access certain of its superclass’s instance variables and methods. A crucial aspect of software engineering in Java is that a subclass cannot access the **private** members of its superclass. If a subclass could access the superclass’s **private** members, this would violate information hiding in the superclass.



Software Engineering Observation 9.1

A subclass cannot directly access **private** members of its superclass.



Testing and Debugging Tip 9.1

Hiding **private** members is a tremendous help in testing, debugging and correctly modifying systems. If a subclass could access its superclass’s **private** members, it would be possible for classes derived from that subclass to access that data as well, and so on. This would propagate access to what is supposed to be **private** data, and the benefits of information hiding would be lost throughout the class hierarchy.

However, a subclass can access the **public** and **protected** members of its superclass. A subclass also can use the package access members of its superclass if the subclass and superclass are in the same package. Superclass members that should not be accessible to a subclass via inheritance are declared **private** in the superclass. A subclass can effect state changes in superclass **private** members only through **public**, **protected** and package access methods provided in the superclass and inherited into the subclass. [Note: We use **protected** instance variables in this chapter to demonstrate how they work. Several of the exercises in this chapter require that you use only private instance variables, to maintain encapsulation.]



Software Engineering Observation 9.2

To preserve encapsulation, all instance variables should be declared **private** and should be accessible only via set and get methods of the class.

A problem with inheritance is that a subclass can inherit methods that it does not need or should not have. It is the class designer’s responsibility to ensure that the capabilities provided by a class are appropriate for future subclasses. Even when the superclass methods are appropriate for the subclasses, it is common for a subclass to require the method to perform a task in a manner that is specific to the subclass. In such cases, the superclass method can be *overridden* (redefined) in the subclass with an appropriate implementation.

Perhaps most exciting is the notion that new classes can inherit from abundant *class libraries*, such as those provided with the Java API. Organizations develop their own class

libraries and can take advantage of other libraries available worldwide. Someday, most software might be constructed from *standardized reusable components*, just as hardware is often constructed today. This will help meet the challenges of developing the ever more powerful software we will need in the future.

9.2 Superclasses and Subclasses

Often an object of one class “is an” object of another class as well. A rectangle certainly *is* a quadrilateral (as are squares, parallelograms and trapezoids). Thus, class **Rectangle** can be said to *inherit* from class **Quadrilateral**. In this context, class **Quadrilateral** is a superclass, and class **Rectangle** is a subclass. A rectangle *is a* specific type of quadrilateral, but it is incorrect to claim that a quadrilateral *is a* rectangle (the quadrilateral could be a parallelogram). Figure 9.1 shows several simple inheritance examples of superclasses and potential subclasses.

Inheritance normally produces subclasses with *more* features than their superclasses, so the terms superclass and subclass can be confusing. There is another way, however, to view these terms that makes perfectly good sense. Because every subclass object “is an” object of its superclass, and because one superclass can have many subclasses, the set of objects represented by a superclass is normally larger than the set of objects represented by any of that superclass’s subclasses. For example, the superclass **Vehicle** represents in a generic manner all vehicles, such as cars, trucks, boats, bicycles and so on. However, subclass **Car** represents only a small subset of all the **Vehicles** in the world.

Inheritance relationships form tree-like hierarchical structures. A superclass exists in a hierarchical relationship with its subclasses. A class can certainly exist by itself, but it is when a class is used with the mechanism of inheritance that the class becomes either a superclass that supplies attributes and behaviors to other classes or a subclass that inherits those attributes and behaviors. Frequently, one class is both a subclass and a superclass.

Superclass	Subclasses
Student	GraduateStudent UndergraduateStudent
Shape	Circle Triangle Rectangle
Loan	CarLoan HomeImprovementLoan MortgageLoan
Employee	FacultyMember StaffMember
Account	CheckingAccount SavingsAccount

Fig. 9.1 Some simple inheritance examples in which the subclass “is a” superclass.

Let us develop a simple inheritance hierarchy. A typical university community has thousands of people who are community members. These people consist of employees, students and alumni. Employees are either faculty members or staff members. Faculty members are either administrators (such as deans and department chairpersons) or teaching faculty. This yields the inheritance hierarchy shown in Fig. 9.2. Note that the inheritance hierarchy could contain many other classes. For example, students can be graduate students or undergraduate students. Undergraduate students can be freshman, sophomores, juniors, and seniors. And so on. The arrows in the hierarchy represent the “is a” relationship. For example, based on this class hierarchy that we can state, “an **Employee** is a **CommunityMember**,” or “a **Teacher** is a **Faculty** member.” **CommunityMember** is the *direct superclass* of **Employee**, **Student** and **Alumni**. **CommunityMember** is an *indirect superclass* of all the other classes in the hierarchy diagram. Note that class **Employee** is both a subclass of **CommunityMember** and a superclass of **Faculty** and **Staff**.

Also, starting from the bottom of the diagram, you can follow the arrows and apply the *is a* relationship all the way up to the topmost superclass in the hierarchy. For example, an **Administrator** is a **Faculty** member, is an **Employee** and is a **CommunityMember**. And, in Java, an **Administrator** also is an **Object** because all classes in Java have **Object** as one of their direct or indirect superclasses. Thus, all classes in Java are related in a hierarchical relationship in which they share the 11 methods defined by class **Object**, which include the **toString** and **finalize** methods discussed previously. Other methods of class **Object** are discussed as they are needed in the text.

Another substantial inheritance hierarchy is the **Shape** hierarchy of Figure 9.3. There are abundant examples of hierarchies in the real world, but students are not accustomed to categorizing the real world in this manner, so it takes some adjustment in their thinking. Actually, biology students have had some practice with hierarchies. Everything we study in biology is grouped into a hierarchy headed by living things and these can be plants or animals and so on.

To specify that class **TwoDimensionalShape** is derived from (or inherits from) class **Shape**, class **TwoDimensionalShape** could be defined in Java as follows:

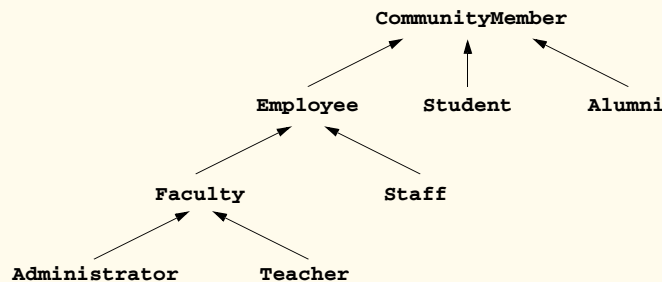


Fig. 9.2 An inheritance hierarchy for university **CommunityMembers**.

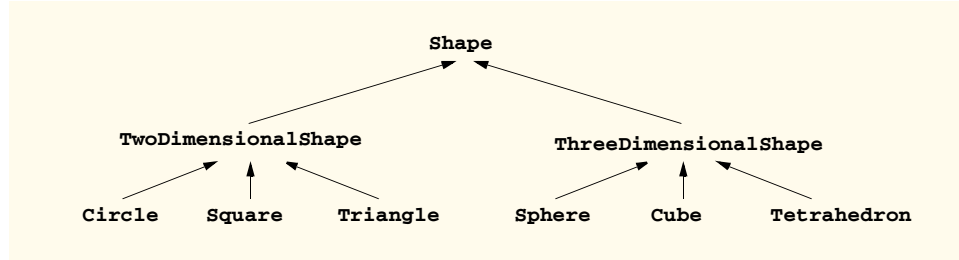


Fig. 9.3 A portion of a **Shape** class hierarchy.

```

public class TwoDimensionalShape extends Shape {
    ...
}
  
```

With inheritance, **private** members of a superclass are not directly accessible from that class's subclasses. Package access members of the superclass are accessible in a subclass only if both the superclass and its subclass are in the same package. All other superclass members become members of the subclass, using their original member access (i.e., **public** members of the superclass become **public** members of the subclass, and **protected** members of the superclass become **protected** members of the subclass).



Software Engineering Observation 9.3

Constructors are never inherited—they are specific to the class in which they are defined.

It is possible to treat superclass objects and subclass objects similarly; that commonality is expressed in the attributes and behaviors of the superclass. Objects of all classes derived from a common superclass can all be treated as objects of that superclass.

We will consider many examples in which we can take advantage of this inheritance relationship with an ease of programming not available in non-object-oriented languages such as C.

9.3 protected Members

A superclass's **public** members are accessible anywhere the program has a reference to that superclass type or one of its subclass types. A superclass's **private** members are accessible only in methods of that superclass.

A superclass's **protected** access members serve as an intermediate level of protection between **public** and **private** access. A superclass's **protected** members may be accessed only by methods of the superclass, by methods of subclasses and by methods of other classes in the same package (**protected** members have package access).

Subclass methods can normally refer to **public** and **protected** members of the superclass simply by using the member names. When a subclass method overrides a superclass method, the superclass method may be accessed from the subclass by preceding the superclass method name with keyword **super** followed by the dot operator (.). This technique is illustrated several times throughout the chapter.

9.4 Relationship between Superclass Objects and Subclass Objects

An object of a subclass can be treated as an object of its superclass. This makes possible some interesting manipulations. For example, despite the fact that objects of a variety of classes derived from a particular superclass might be quite different from one another, we can create an array of references to them—as long as we treat them as superclass objects. But the reverse is not true: A superclass object cannot always be treated a subclass object. For example, a **Shape** is not always a **Circle**.

However, an explicit cast can be used to convert a superclass reference to a subclass reference. This can be done only when the superclass reference is referencing a subclass object; otherwise, Java will indicate a **ClassCastException**—an indication that the cast operation is not allowed. Exceptions are discussed in detail in Chapter 14.



Common Programming Error 9.1

Assigning an object of a superclass to a subclass reference (without a cast) is a syntax error.



Software Engineering Observation 9.4

If an object has been assigned to a reference of one of its superclasses, it is acceptable to cast that object back to its own type. In fact, this must be done in order to send that object any of its messages that do not appear in that superclass.

Our first example consists of two classes. Figure 9.4 shows a **Point** class definition. Figure 9.5 shows a **Circle** class definition. We will see that class **Circle** inherits from class **Point**. Figure 9.6 shows application class **InheritanceTest**, which demonstrates assigning subclass references to superclass references and casting superclass references to subclass references.

Every applet defined previously has used some of the techniques presented here. We now formalize the inheritance concept. In Chapter 3, we stated that every class definition in Java must extend another class. However, notice in Fig. 9.4 that class **Point** (line 4) does not use the **extends** keyword explicitly. If a new class definition does not extend an existing class definition explicitly, Java implicitly uses class **Object** (package **java.lang**) as the superclass for the new class definition. Class **Object** provides a set of methods that can be used with any object of any class.



Software Engineering Observation 9.5

Every class in Java implicitly extends **Object**, unless it is specified otherwise in the first line of the class definition, in which case the class indirectly extends **Object**. Thus, class **Object** is the superclass of the entire Java class hierarchy.

Let us first examine the **Point** class definition (Fig. 9.4). The **public** services of class **Point** include methods **setPoint**, **getX**, **getY**, **toString** and two **Point** constructors. The instance variables **x** and **y** of **Point** are specified as **protected**. This prevents clients of **Point** objects from accessing the data directly (unless they are classes in the same package), but enables classes derived from **Point** to access the inherited instance variables directly. If the data were specified as **private**, the non**private** methods of **Point** would have to be used to access the data, even by subclasses. Note that class **Point**'s **toString** method overrides the original **toString** method inherited from class **Object**.

```

1  // Fig. 9.4: Point.java
2  // Definition of class Point
3
4  public class Point {
5      protected int x, y; // coordinates of Point
6
7      // No-argument constructor
8      public Point()
9      {
10         // implicit call to superclass constructor occurs here
11         setPoint( 0, 0 );
12     }
13
14     // constructor
15     public Point( int xCoordinate, int yCoordinate )
16     {
17         // implicit call to superclass constructor occurs here
18         setPoint( xCoordinate, yCoordinate );
19     }
20
21     // set x and y coordinates of Point
22     public void setPoint( int xCoordinate, int yCoordinate )
23     {
24         x = xCoordinate;
25         y = yCoordinate;
26     }
27
28     // get x coordinate
29     public int getX()
30     {
31         return x;
32     }
33
34     // get y coordinate
35     public int getY()
36     {
37         return y;
38     }
39
40     // convert into a String representation
41     public String toString()
42     {
43         return "[" + x + ", " + y + "]";
44     }
45
46 } // end class Point

```

Fig. 9.4 **Point** class definition.

Class **Point**'s constructors (lines 8–12 and 15–19) must call class **Object**'s constructor. In fact, every subclass constructor is required to call its direct superclass's constructor as its first task, either implicitly or explicitly (the syntax for this call is discussed with class **Circle** momentarily). If there is no explicit call to the superclass constructor,

Java automatically attempts to call the superclass's default constructor. Note that lines 10 and 17 are comments indicating where the call to the superclass **Object**'s default constructor occurs.



Software Engineering Observation 9.6

Every subclass constructor must call one of the direct superclass constructors explicitly or implicitly. Implicit calls can be made only to the no-argument constructor of the superclass. If the superclass does not provide a no-argument constructor, all direct subclasses of that class must call one of superclass's constructors explicitly.

Class **Circle** (Fig. 9.5) inherits from class **Point** as specified with the **extends** keyword on line 4. Keyword **extends** in the class definition indicates inheritance. All the (non**private**) members of class **Point** (except the constructors) are inherited into class **Circle**. Thus, the **public** interface to **Circle** includes the **Point** class's **public** methods as well as the two overloaded **Circle** constructors and **Circle** methods **setRadius**, **getRadius**, **area** and **toString**. Notice that method **area** (lines 38–41) uses predefined constant **Math.PI** from class **Math** (package **java.lang**) to calculate the area of a circle.

```

1  // Fig. 9.5: Circle.java
2  // Definition of class Circle
3
4  public class Circle extends Point { // inherits from Point
5      protected double radius;
6
7      // no-argument constructor
8      public Circle()
9      {
10         // implicit call to superclass constructor occurs here
11         setRadius( 0 );
12     }
13
14     // constructor
15     public Circle( double circleRadius, int xCoordinate,
16                 int yCoordinate )
17     {
18         // call superclass constructor to set coordinates
19         super( xCoordinate, yCoordinate );
20
21         // set radius
22         setRadius( circleRadius );
23     }
24
25     // set radius of Circle
26     public void setRadius( double circleRadius )
27     {
28         radius = ( circleRadius >= 0.0 ? circleRadius : 0.0 );
29     }
30

```

Fig. 9.5 **Circle** class definition (part 1 of 2).

```

31 // get radius of Circle
32 public double getRadius()
33 {
34     return radius;
35 }
36
37 // calculate area of Circle
38 public double area()
39 {
40     return Math.PI * radius * radius;
41 }
42
43 // convert the Circle to a String
44 public String toString()
45 {
46     return "Center = " + "[" + x + ", " + y + "]" +
47         "; Radius = " + radius;
48 }
49
50 } // end class Circle

```

Fig. 9.5 Circle class definition (part 2 of 2).

The **Circle** constructors (lines 8–12 and 15–23) must invoke a **Point** constructor to initialize the superclass portion of a **Circle** object (i.e., variables **x** and **y** inherited from **Point**). The default constructor at lines 8–12 does not call a **Point** constructor explicitly, so Java automatically calls class **Point**'s default constructor (defined at line 8 of Fig. 9.4) that initializes superclass members **x** and **y** to zeros. If class **Point** contained only the constructor with arguments (i.e., did not provide a default constructor), a compiler error would occur.

Line 19 in the body of the second **Circle** constructor explicitly invokes the **Point** constructor (defined at line 15 of Fig. 9.4) by using the *superclass constructor call syntax*—keyword **super** followed by a set of parentheses containing the arguments to the superclass constructor. In this case, the arguments are the values **xCoordinate** and **yCoordinate** that are used by the **Point** constructor to initialize the superclass members **x** and **y**. The call to the superclass constructor must be the first line in the body of the subclass constructor. To call the superclass default constructor explicitly, use the statement

```
super(); // explicit call to superclass default constructor
```



Common Programming Error 9.2

It is a syntax error if a **super** call by a subclass to its superclass constructor is not the first statement in the subclass constructor.



Common Programming Error 9.3

It is a syntax error if the arguments to a **super** call by a subclass to its superclass constructor do not match the parameters specified in one of the superclass constructor definitions.

A subclass can redefine a superclass method by using the same signature; this is called *overriding* a superclass method. When that method is mentioned by name in the subclass,

the subclass version is automatically called. We have actually been overriding methods in every applet in the book. When we extend **JApplet** to create a new applet class, the new class inherits versions of **init** and **paint** (and many other methods). Each time we defined **init** or **paint**, we were overriding the original version that was inherited. Also, when we provided method **toString** for many of the classes in Chapter 8, we were overriding the original version of **toString** provided by class **Object**. As we will see in Fig. 9.8, the **super** reference followed by the dot operator can be used to access the original superclass version of that method from the subclass.

Note that class **Circle**'s **toString** method (lines 44–48) overrides the **Point** class **toString** method (lines 41–44 of Fig. 9.4). Class **Point**'s **toString** method overrides the original **toString** method provided by class **Object**. Actually, every class inherits a **toString** method, because class **Object** provides the original **toString** method. This method converts an object of any class into a **String** representation and is sometimes called implicitly by the program (e.g., when an object is concatenated with a **String**). **Circle** method **toString** directly accesses the **protected** instance variables **x** and **y** that were inherited from class **Point**. Method **toString** uses the values of **x** and **y** as part of the **Circle**'s **String** representation. Actually, if you study class **Point**'s **toString** method and class **Circle**'s **toString** method, you will notice that **Circle**'s **toString** uses the same formatting as **Point**'s **toString** for the **Point** parts of the **Circle**. Also, recall *Software Engineering Observation 8.14*, indicating that, if a method exists that performs part of another method's task, call the method. **Point**'s **toString** performs part of the task of **Circle**'s **toString**. To call **Point**'s **toString** from class **Circle**, use the expression

```
super.toString()
```



Software Engineering Observation 9.7

A redefinition of a superclass method in a subclass need not have the same signature as the superclass method. Such a redefinition is not method overriding; rather, it is an example of method overloading.



Software Engineering Observation 9.8

Any object can be converted to a **String** with an explicit or implicit call to the object's **toString** method.



Software Engineering Observation 9.9

Each class should override method **toString** to return useful information about objects of that class.



Common Programming Error 9.4

It is a syntax error if a method in a superclass and a method in its subclass have the same signature but a different return type.

The **InheritanceTest** application (Fig. 9.6) instantiates **Point** object **point1** and **Circle** object **circle1** at lines 18–19 in **main**. The **String** representations of each object are assigned to **String output** to show that they were initialized correctly (lines 21–22). See the first two lines in the output screen capture to confirm this.

```
1 // Fig. 9.6: InheritanceTest.java
2 // Demonstrating the "is a" relationship
3
4 // Java core packages
5 import java.text.DecimalFormat;
6
7 // Java extension packages
8 import javax.swing.JOptionPane;
9
10 public class InheritanceTest {
11
12     // test classes Point and Circle
13     public static void main( String args[] )
14     {
15         Point point1, point2;
16         Circle circle1, circle2;
17
18         point1 = new Point( 30, 50 );
19         circle1 = new Circle( 2.7, 120, 89 );
20
21         String output = "Point point1: " + point1.toString() +
22             "\nCircle circle1: " + circle1.toString();
23
24         // use "is a" relationship to refer to a Circle
25         // with a Point reference
26         point2 = circle1;    // assigns Circle to a Point reference
27
28         output += "\n\nCircle circle1 (via point2 reference): " +
29             point2.toString();
30
31         // use downcasting (casting a superclass reference to a
32         // subclass data type) to assign point2 to circle2
33         circle2 = ( Circle ) point2;
34
35         output += "\n\nCircle circle1 (via circle2): " +
36             circle2.toString();
37
38         DecimalFormat precision2 = new DecimalFormat( "0.00" );
39         output += "\n\nArea of c (via circle2): " +
40             precision2.format( circle2.area() );
41
42         // attempt to refer to Point object with Circle reference
43         if ( point1 instanceof Circle ) {
44             circle2 = ( Circle ) point1;
45             output += "\n\nCast successful";
46         }
47         else
48             output += "\n\npoint1 does not refer to a Circle";
49
50         JOptionPane.showMessageDialog( null, output,
51             "Demonstrating the \"is a\" relationship",
52             JOptionPane.INFORMATION_MESSAGE );
53     }
```

Fig. 9.6 Assigning subclass references to superclass references (part 1 of 2).

```

54      System.exit( 0 );
55  }
56
57 } // end class InheritanceTest

```

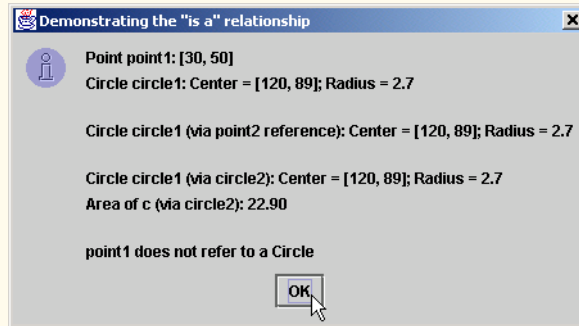


Fig. 9.6 Assigning subclass references to superclass references (part 2 of 2).

Line 26 assigns **circle1** (a reference to a subclass **Circle** object) to **point2** (a superclass **Point** reference). It is always acceptable in Java to assign a subclass reference to a superclass reference, because of the “is a” relationship of inheritance. A **Circle** *is a* **Point** because class **Circle** extends class **Point**. Assigning a superclass reference to a subclass reference is dangerous, as we will see.

Lines 28–29 append the result of **point2.toString()** to **output**. Interestingly, when **point2** is sent the **toString** message, Java knows that the object really is a **Circle**, so it chooses the **Circle** class’s **toString** rather than the **Point** class’s **toString** as you might have expected. This is an example of *polymorphism* and *dynamic binding*—concepts we treat in depth later in this chapter. The compiler looks at the preceding expression and asks the question, “Does the data type of the reference **point2** (i.e., **Point**) have a **toString** method with no arguments?” The answer to this question is yes (per **Point**’s **toString** definition on line 41 of Fig. 9.4). The compiler simply checks the syntax of the expression and ensures that the method exists. At execution time, the interpreter asks the question, “What type is the object to which **point2** refers?” Every object in Java knows its own data type, so the answer to the question is **point2** refers to a **Circle** object. Based on this answer, the interpreter calls the **toString** method of the actual object’s data type—class **Circle**’s **toString** method. See the third line of the screen capture to confirm this. The two key programming techniques we used to achieve this polymorphism effect are

1. extending class **Point** to create class **Circle**, and
2. overriding method **toString** with the exact same signature in class **Point** and class **Circle**.

Line 33 casts **point2**, which admittedly is referencing a **Circle** at this time in the program’s execution, to a **Circle** and assigns the result to **circle2** (this cast would be dangerous if **point2** were really referencing a **Point**, as we will soon discuss). Then we use **circle2** to append to **output** the various facts about **circle2**. Lines 35–36

invoke method **toString** to append the **String** representation of the **Circle**. Lines 39–40 append the **area** of the **Circle** to **output**.

Next, the **if/else** structure at lines 43–48 attempts a dangerous cast in line 44. We cast **point1**, which refers to a **Point** object, to a **Circle**. If the program attempts to execute this statement, Java would determine that **point1** really refers to a **Point**, recognize the cast to **Circle** as being dangerous and indicate an improper cast with **ClassCastException** message. However, we prevent this statement from executing with the **if** condition

```
if ( point1 instanceof Circle ) {
```

that uses operator **instanceof** to determine whether the object to which **point1** refers is a **Circle**. This condition evaluates to **true** only if the object to which **point1** refers is a **Circle**; otherwise, the condition evaluates to **false**. Reference **point1** does not refer to a **Circle**, so the condition fails, and a **String** indicating that **point1** does not refer to a **Circle** is appended to **output**.

If we remove the **if** test from the program and execute the program, the following message is generated at execution time:

```
Exception in thread "main" java.lang.ClassCastException: Point
    at InheritanceTest.main(InheritanceTest.java:43)
```

Such error messages normally include the file name (**InheritanceTest.java**) and line number at which the error occurred (**43**) so you can go to that specific line in the program for debugging.

9.5 Constructors and Finalizers in Subclasses

When an object of a subclass is instantiated, the superclass's constructor should be called to do any necessary initialization of the superclass instance variables of the subclass object. An explicit call to the superclass constructor (via the **super** reference) can be provided as the first statement in the subclass constructor. Otherwise, the subclass constructor will call the superclass default constructor (or no-argument constructor) implicitly.

Superclass constructors are not inherited by subclasses. Subclass constructors, however, can call superclass constructors via the **super** reference.



Software Engineering Observation 9.10

*When an object of a subclass is created, first the subclass constructor calls the superclass constructor (explicitly via **super** or implicitly), the superclass constructor executes, then the remainder of the subclass constructor's body executes.*

If the classes in your class hierarchy define **finalize** methods, the subclass **finalize** method as its last action should invoke the superclass **finalize** method to ensure that all parts of an object are finalized properly if the garbage collector reclaims the memory for the object.

The application of Fig. 9.7–Fig. 9.9 shows the order in which superclass and subclass constructors and finalizers are called. For the purpose of this example, class **Point** and class **Circle** are simplified.

Class **Point** (Fig. 9.7) contains two constructors, a finalizer, a **toString** method and **protected** instance variables **x** and **y**. The constructor and finalizer each print that they are executing, then display the **Point** for which they are invoked. Note the use of

this in the **System.out.println** calls to cause an implicit call to method **toString**. Notice the first line of the **finalize** method (line 23). Method **finalize** should always be defined as **protected** so subclasses have access to the method but classes that simply use **Point** objects do not.

Class **Circle** (Fig. 9.8) derives from **Point** and contains two constructors, a finalizer, a **toString** method and protected instance variable **radius**. The constructor and finalizer each print that they are executing, then display the **Circle** for which they are invoked. Note that the **Circle** method **toString** invokes **Point**'s **toString** via **super** (line 19).



Software Engineering Observation 9.11

When a superclass method is overridden in a subclass, it is common to have the subclass version call the superclass version and do some additional work. In this scenario, the superclass method performs the tasks common to all subclasses of that class, and the subclass method performs additional tasks specific to a given subclass.

```

1  // Fig. 9.7: Point.java
2  // Definition of class Point
3  public class Point extends Object {
4      protected int x, y; // coordinates of the Point
5
6      // no-argument constructor
7      public Point()
8      {
9          x = 0;
10         y = 0;
11         System.out.println( "Point constructor: " + this );
12     }
13
14     // constructor
15     public Point( int xCoordinate, int yCoordinate )
16     {
17         x = xCoordinate;
18         y = yCoordinate;
19         System.out.println( "Point constructor: " + this );
20     }
21
22     // finalizer
23     protected void finalize()
24     {
25         System.out.println( "Point finalizer: " + this );
26     }
27
28     // convert Point into a String representation
29     public String toString()
30     {
31         return "[" + x + ", " + y + "]";
32     }
33
34 } // end class Point

```

Fig. 9.7 **Point** class definition to demonstrate when constructors and finalizers are called.

```

1  // Fig. 9.8: Circle.java
2  // Definition of class Circle
3  public class Circle extends Point { // inherits from Point
4      protected double radius;
5
6      // no-argument constructor
7      public Circle()
8      {
9          // implicit call to superclass constructor here
10         radius = 0;
11         System.out.println( "Circle constructor: " + this );
12     }
13
14     // Constructor
15     public Circle( double circleRadius, int xCoordinate,
16                   int yCoordinate )
17     {
18         // call superclass constructor
19         super( xCoordinate, yCoordinate );
20
21         radius = circleRadius;
22         System.out.println( "Circle constructor: " + this );
23     }
24
25     // finalizer
26     protected void finalize()
27     {
28         System.out.println( "Circle finalizer: " + this );
29         super.finalize(); // call superclass finalize method
30     }
31
32     // convert the Circle to a String
33     public String toString()
34     {
35         return "Center = " + super.toString() +
36               "; Radius = " + radius;
37     }
38
39 } // end class Circle

```

Fig. 9.8 **Circle** class definition to demonstrate when constructors and finalizers are called.



Common Programming Error 9.5

When an overridden method calls the superclass version of the same method, not using keyword **super** to reference the superclass's method causes infinite recursion, because the subclass method actually calls itself.



Common Programming Error 9.6

Cascading **super** reference to refer to a member (method or variable) several levels up the hierarchy (as in **super.super.x**) is a syntax error.

Application class **Test** (Fig. 9.9) uses this **Point/Circle** inheritance hierarchy. The application begins in method **main** by instantiating **Circle** object **circle1** (line 11). This invokes the **Circle** constructor at line 15 of Fig. 9.8, which immediately invokes the **Point** constructor at line 15 of Fig. 9.7. The **Point** constructor outputs the values received from the **Circle** constructor by implicitly calling method **toString** and returns program control to the **Circle** constructor. Then the **Circle** constructor outputs the complete **Circle** by calling method **toString**. Notice that the first two lines of the output from this program both show values for **x**, **y** and **radius**. Polymorphism is once again causing the **Circle**'s **toString** method to execute because it is a **Circle** object that is being created. When **toString** is invoked from the **Point** constructor, **0.0** is displayed for the **radius** because the **radius** has not yet been initialized in the **Circle** constructor.

Circle object **circle2** is instantiated next. Again, the **Point** and **Circle** constructors both execute. Notice, in the command-line output window, that the body of the **Point** constructor is performed before the body of the **Circle** constructor, showing that objects are constructed “inside out.”

```

1  // Fig. 9.9: Test.java
2  // Demonstrate when superclass and subclass
3  // constructors and finalizers are called.
4  public class Test {
5
6      // test when constructors and finalizers are called
7      public static void main( String args[] )
8      {
9          Circle circle1, circle2;
10
11         circle1 = new Circle( 4.5, 72, 29 );
12         circle2 = new Circle( 10, 5, 5 );
13
14         circle1 = null; // mark for garbage collection
15         circle2 = null; // mark for garbage collection
16
17         System.gc();    // call the garbage collector
18     }
19
20 } // end class Test

```

```

Point constructor: Center = [72, 29]; Radius = 0.0
Circle constructor: Center = [72, 29]; Radius = 4.5
Point constructor: Center = [5, 5]; Radius = 0.0
Circle constructor: Center = [5, 5]; Radius = 10.0
Circle finalizer: Center = [72, 29]; Radius = 4.5
Point finalizer: Center = [72, 29]; Radius = 4.5
Circle finalizer: Center = [5, 5]; Radius = 10.0
Point finalizer: Center = [5, 5]; Radius = 10.0

```

Fig. 9.9 Order in which constructors and finalizers are called.

Lines 14–15 set **circle1** to **null**, then set **circle2** to **null**. Each of these objects is no longer needed in the program, so Java marks the memory occupied by **circle1** and **circle2** for *garbage collection*. Java guarantees that, before the garbage collector runs to reclaim the space for each of these objects, the **finalize** methods for each object will be called. The garbage collector is a low-priority thread that runs automatically whenever processor time is available. We choose here to ask the garbage collector to run with a call to class **System**'s **static** method **gc** in line 17. Java does not guarantee the order in which objects will be garbage collected; therefore, it cannot guarantee which object's finalizer will execute first. Notice, in the command-line output window, that **finalize** methods are called for both the **Circle** and **Point** when each **Circle** object is garbage collected.

9.6 Implicit Subclass-Object-to-Superclass-Object Conversion

Despite the fact that a subclass object also “is a” superclass object, the subclass type and the superclass type are different. Subclass objects can be treated as superclass objects. This makes sense because the subclass has members corresponding to each of the superclass members—remember that the subclass normally has more members than the superclass has. Assignment in the other direction is not allowed because assigning a superclass object to a subclass reference would leave the additional subclass members undefined.

A reference to a subclass object could be implicitly converted into a reference to a superclass object because a subclass object *is a* superclass object through inheritance.

There are four possible ways to mix and match superclass references and subclass references with superclass objects and subclass objects:

1. Referring to a superclass object with a superclass reference is straightforward.
2. Referring to a subclass object with a subclass reference is straightforward.
3. Referring to a subclass object with a superclass reference is safe, because the subclass object *is an* object of its superclass as well. Such code can refer only to superclass members. If this code refers to subclass-only members through the superclass reference, the compiler will report a syntax error.
4. Referring to a superclass object with a subclass reference is a syntax error.

As convenient as it might be to treat subclass objects as superclass objects, and to do this by manipulating all these objects with superclass references, there appears to be a problem. In a payroll system, for example, we would like to be able to walk through an array of employees and calculate the weekly pay for each person. But intuition suggests that using superclass references would enable the program to call only the superclass payroll calculation routine (if indeed there is such a routine in the superclass). We need a way to invoke the proper payroll calculation routine for each object, whether it is a superclass object or a subclass object, and to do this simply by using the superclass reference. Actually, this is precisely how Java behaves and is discussed in this chapter when we consider polymorphism and dynamic binding.

9.7 Software Engineering with Inheritance

We can use inheritance to customize existing software. When we use inheritance to create a new class from an existing class, the new class inherits the attributes and behaviors of an existing class; then we can add attributes and behaviors or override superclass behaviors to customize the class to meet our needs.

It can be difficult for students to appreciate the problems faced by designers and implementers on large-scale software projects in industry. People experienced on such projects will invariably state that a key to improving the software development process is encouraging software reuse. Object-oriented programming in general, and Java in particular, certainly does this.

It is the availability of substantial and useful class libraries that delivers the maximum benefits of software reuse through inheritance. As interest in Java grows, interest in Java class libraries will increase. Just as shrink-wrapped software produced by independent software vendors became an explosive growth industry with the arrival of the personal computer, so, too, will the creation and sale of Java class libraries. Application designers will build their applications with these libraries, and library designers will be rewarded by having their libraries wrapped with the applications. What we see coming is a massive worldwide commitment to the development of Java class libraries for a huge variety of applications arenas.



Software Engineering Observation 9.12

Creating a subclass does not affect its superclass's source code or the superclass's Java byte-codes; the integrity of a superclass is preserved by inheritance.

A superclass specifies commonality. All classes derived from a superclass inherit the capabilities of that superclass. In the object-oriented design process, the designer looks for commonality among a set of classes and factors it out to form desirable superclasses. Subclasses are then customized beyond the capabilities inherited from the superclass.



Software Engineering Observation 9.13

Just as the designer of non-object-oriented systems should avoid unnecessary proliferation of functions, the designer of object-oriented systems should avoid unnecessary proliferation of classes. Proliferating classes creates management problems and can hinder software re-usability, simply because it is more difficult for a potential user of a class to locate that class in a huge collection. The trade-off is to create fewer classes, each providing substantial additional functionality, but such classes might be too rich for certain users.



Performance Tip 9.1

When creating a new class, inherit from the class “closest” to what you need—i.e., the one that provides the minimum set of capabilities required for a new class to perform its tasks. Subclasses could inherit data and functionality that they will not use, in which case memory and processing resources may be wasted.

Note that reading a set of subclass declarations can be confusing because inherited members are not shown, but inherited members are nevertheless present in the subclasses. A similar problem can exist in the documentation of subclasses.



Software Engineering Observation 9.14

In an object-oriented system, classes are often closely related. “Factor out” common attributes and behaviors and place these in a superclass. Then use inheritance to form subclasses without having to repeat common attributes and behaviors.



Software Engineering Observation 9.15

Modifications to a superclass do not require subclasses to change as long as the public interface to the superclass remains unchanged.

9.8 Composition vs. Inheritance

We have discussed *is a* relationships that are implemented by inheritance. We have also discussed *has a* relationships (and seen examples in preceding chapters) in which a class has objects of other classes as members—such relationships create new classes by *composition* of existing classes. For example, given the classes **Employee**, **BirthDate** and **TelephoneNumber**, it is improper to say that an **Employee** *is a* **BirthDate** or that an **Employee** *is a* **TelephoneNumber**. But it is certainly appropriate to say that an **Employee** *has a* **BirthDate** and that an **Employee** *has a* **TelephoneNumber**.

9.9 Case Study: Point, Circle, Cylinder

Now let us consider a substantial inheritance example. We consider a point, circle, cylinder hierarchy. First we develop and use class **Point** (Fig. 9.10 and Fig. 9.11). Then we present an example in which we derive class **Circle** from class **Point** (Fig. 9.12 and Fig. 9.13). Finally, we present an example in which we derive class **Cylinder** from class **Circle** (Fig. 9.14 and Fig. 9.15).

Figure 9.10 is the class **Point** definition. Class **Point** is defined as part of package **com.deitel.jhtp4.ch09** (line 3). Note that **Point**'s instance variables are **protected**. Thus, when class **Circle** is derived from class **Point**, the methods of class **Circle** will be able to reference coordinates **x** and **y** directly rather than using access methods. This could result in better performance.

```

1  // Fig. 9.10: Point.java
2  // Definition of class Point
3  package com.deitel.jhtp4.ch09;
4
5  public class Point {
6      protected int x, y; // coordinates of Point
7
8      // No-argument constructor
9      public Point()
10     {
11         // implicit call to superclass constructor occurs here
12         setPoint( 0, 0 );
13     }
14
15     // constructor
16     public Point( int xCoordinate, int yCoordinate )
17     {
18         // implicit call to superclass constructor occurs here
19         setPoint( xCoordinate, yCoordinate );
20     }
21

```

Fig. 9.10 **Point** class definition (part 1 of 2).

```
22 // set x and y coordinates of Point
23 public void setPoint( int xCoordinate, int yCoordinate )
24 {
25     x = xCoordinate;
26     y = yCoordinate;
27 }
28
29 // get x coordinate
30 public int getX()
31 {
32     return x;
33 }
34
35 // get y coordinate
36 public int getY()
37 {
38     return y;
39 }
40
41 // convert into a String representation
42 public String toString()
43 {
44     return "[" + x + ", " + y + "]";
45 }
46
47 } // end class Point
```

Fig. 9.10 **Point** class definition (part 2 of 2).

Figure 9.11 shows a **Test** application for testing class **Point**. The **main** method must use **getX** and **getY** to read the values of **protected** instance variables **x** and **y**. Remember that **protected** instance variables are accessible only to methods of their class, their subclasses and other classes in the same package. Also, note the implicit call to **toString** when **point** is added to a **String** at line 25.

```
1 // Fig. 9.11: Test.java
2 // Applet to test class Point
3
4 // Java extension packages
5 import javax.swing.JOptionPane;
6
7 // Deitel packages
8 import com.deitel.jhtp4.ch09.Point;
9
10 public class Test {
11
12     // test class Point
13     public static void main( String args[] )
14     {
15         Point point = new Point( 72, 115 );
16     }
```

Fig. 9.11 Testing class **Point** (part 1 of 2).

```

17      // get coordinates
18      String output = "X coordinate is " + point.getX() +
19                      "\nY coordinate is " + point.getY();
20
21      // set coordinates
22      point.setPoint( 10, 10 );
23
24      // use implicit call to point.toString()
25      output += "\n\nThe new location of point is " + point;
26
27      JOptionPane.showMessageDialog( null, output,
28                                     "Demonstrating Class Point",
29                                     JOptionPane.INFORMATION_MESSAGE );
30
31      System.exit( 0 );
32  }
33
34  } // end class Test

```

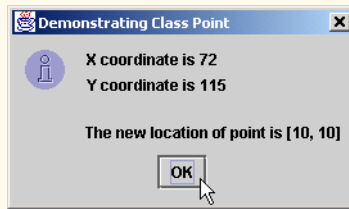


Fig. 9.11 Testing class **Point** (part 2 of 2).

Our next example imports the **Point** class definition from Fig. 9.10, so we do not show the class definition again here. Figure 9.12 shows the **Circle** class definition with the **Circle** method definitions. Note that class **Circle** **extends** class **Point**. This means that the **public** interface to **Circle** includes the **Point** methods as well as the **Circle** methods **setRadius**, **getRadius**, **area**, **toString** and the **Circle** constructors.

```

1  // Fig. 9.12: Circle.java
2  // Definition of class Circle
3  package com.deitel.jhtp4.ch09;
4
5  public class Circle extends Point { // inherits from Point
6      protected double radius;
7
8      // no-argument constructor
9      public Circle()
10     {
11         // implicit call to superclass constructor occurs here
12         setRadius( 0 );
13     }
14

```

Fig. 9.12 **Circle** class definition (part 1 of 2).

```

15  // constructor
16  public Circle( double circleRadius, int xCoordinate,
17               int yCoordinate )
18  {
19      // call superclass constructor to set coordinates
20      super( xCoordinate, yCoordinate );
21
22      // set radius
23      setRadius( circleRadius );
24  }
25
26  // set radius of Circle
27  public void setRadius( double circleRadius )
28  {
29      radius = ( circleRadius >= 0.0 ? circleRadius : 0.0 );
30  }
31
32  // get radius of Circle
33  public double getRadius()
34  {
35      return radius;
36  }
37
38  // calculate area of Circle
39  public double area()
40  {
41      return Math.PI * radius * radius;
42  }
43
44  // convert the Circle to a String
45  public String toString()
46  {
47      return "Center = " + "[" + x + ", " + y + "]" +
48             "; Radius = " + radius;
49  }
50
51  } // end class Circle

```

Fig. 9.12 **Circle** class definition (part 2 of 2).

Application **Test** (Fig. 9.13) instantiates an object of class **Circle** (line 19), then uses *get* methods to obtain the information about the **Circle** object. Method **main** indirectly references the **protected** data of class **Circle** through method calls. Method **main** then uses *set* methods **setRadius** and **setPoint** to reset the radius and coordinates of the center of the circle. Finally, **main** displays the **Circle** object **circle** and calculates and displays its area.

```

1  // Fig. 9.13: Test.java
2  // Applet to test class Circle
3

```

Fig. 9.13 Testing class **Circle** (part 1 of 2).

```

4  // Java core packages
5  import java.text.DecimalFormat;
6
7  // Java extension packages
8  import javax.swing.JOptionPane;
9
10 // Deitel packages
11 import com.deitel.jhtp4.ch09.Circle;
12
13 public class Test {
14
15     // test class Circle
16     public static void main( String args[] )
17     {
18         // create a Circle
19         Circle circle = new Circle( 2.5, 37, 43 );
20         DecimalFormat precision2 = new DecimalFormat( "0.00" );
21
22         // get coordinates and radius
23         String output = "X coordinate is " + circle.getX() +
24             "\nY coordinate is " + circle.getY() +
25             "\nRadius is " + circle.getRadius();
26
27         // set coordinates and radius
28         circle.setRadius( 4.25 );
29         circle.setPoint( 2, 2 );
30
31         // get String representation of Circle and calculate area
32         output +=
33             "\n\nThe new location and radius of c are\n" + circle +
34             "\nArea is " + precision2.format( circle.area() );
35
36         JOptionPane.showMessageDialog( null, output,
37             "Demonstrating Class Circle",
38             JOptionPane.INFORMATION_MESSAGE );
39
40         System.exit( 0 );
41     }
42
43 } // end class Test

```

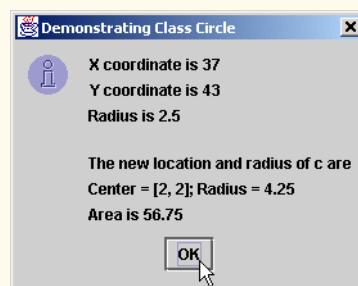


Fig. 9.13 Testing class **Circle** (part 2 of 2).

Our last example is shown in Fig. 9.14 and Fig. 9.15. Figure 9.14 shows the **Cylinder** class definition with the **Cylinder** method definitions. Note that class **Cylinder** extends class **Circle**. This means that the **public** interface to **Cylinder** includes the **Circle** methods and **Point** methods as well as the **Cylinder** constructor and **Cylinder** methods **setHeight**, **getHeight**, **area** (which overrides the **Circle** **area** method), **volume** and **toString**.

```

1  // Fig. 9.14: Cylinder.java
2  // Definition of class Cylinder
3  package com.deitel.jhtp4.ch09;
4
5  public class Cylinder extends Circle {
6      protected double height; // height of Cylinder
7
8      // no-argument constructor
9      public Cylinder()
10     {
11         // implicit call to superclass constructor here
12         setHeight( 0 );
13     }
14
15     // constructor
16     public Cylinder( double cylinderHeight, double cylinderRadius,
17                     int xCoordinate, int yCoordinate )
18     {
19         // call superclass constructor to set coordinates/radius
20         super( cylinderRadius, xCoordinate, yCoordinate );
21
22         // set cylinder height
23         setHeight( cylinderHeight );
24     }
25
26     // set height of Cylinder
27     public void setHeight( double cylinderHeight )
28     {
29         height = ( cylinderHeight >= 0 ? cylinderHeight : 0 );
30     }
31
32     // get height of Cylinder
33     public double getHeight()
34     {
35         return height;
36     }
37
38     // calculate area of Cylinder (i.e., surface area)
39     public double area()
40     {
41         return 2 * super.area() +
42                2 * Math.PI * radius * height;
43     }
44

```

Fig. 9.14 Class **Cylinder** definition (part 1 of 2).

```

45 // calculate volume of Cylinder
46 public double volume()
47 {
48     return super.area() * height;
49 }
50
51 // convert the Cylinder to a String
52 public String toString()
53 {
54     return super.toString() + "; Height = " + height;
55 }
56
57 } // end class Cylinder

```

Fig. 9.14 Class **Cylinder** definition (part 2 of 2).

Method **main** of the **Test** application (Fig. 9.15) instantiates an object of class **Cylinder** (line 19), then uses *get* methods (lines 23–26) to obtain information about the **Cylinder** object. Again, the **Test** application's **main** method cannot reference directly the **protected** data of class **Cylinder**. Method **main** uses *set* methods **setHeight**, **setRadius** and **setPoint** (lines 29–31) to reset the **height**, **radius** and coordinates of the **Cylinder**. Then **main** uses **toString**, **area** and **volume** to print the attributes and some facts about the **Cylinder**. Figure 9.15 is a **Test** application to test class **Cylinder**'s capabilities.

```

1 // Fig. 9.15: Test.java
2 // Application to test class Cylinder
3
4 // Java core packages
5 import java.text.DecimalFormat;
6
7 // Java extension packages
8 import javax.swing.JOptionPane;
9
10 // Deitel packages
11 import com.deitel.jhtp4.ch09.Cylinder;
12
13 public class Test {
14
15     // test class Cylinder
16     public static void main( String args[] )
17     {
18         // create Cylinder
19         Cylinder cylinder = new Cylinder( 5.7, 2.5, 12, 23 );
20         DecimalFormat precision2 = new DecimalFormat( "0.00" );
21
22         // get coordinates, radius and height
23         String output = "X coordinate is " + cylinder.getX() +
24             "\nY coordinate is " + cylinder.getY() +
25             "\nRadius is " + cylinder.getRadius() +
26             "\nHeight is " + cylinder.getHeight();

```

Fig. 9.15 Testing class **Cylinder** (part 1 of 2).

```

27
28     // set coordinates, radius and height
29     cylinder.setHeight( 10 );
30     cylinder.setRadius( 4.25 );
31     cylinder.setPoint( 2, 2 );
32
33     // get String representation of Cylinder and calculate
34     // area and volume
35     output += "\n\nThe new location, radius " +
36             "and height of cylinder are\n" + cylinder +
37             "\nArea is " + precision2.format( cylinder.area() ) +
38             "\nVolume is " + precision2.format( cylinder.volume() );
39
40     JOptionPane.showMessageDialog( null, output,
41     "Demonstrating Class Cylinder",
42     JOptionPane.INFORMATION_MESSAGE );
43
44     System.exit( 0 );
45 }
46
47 } // end class Test

```

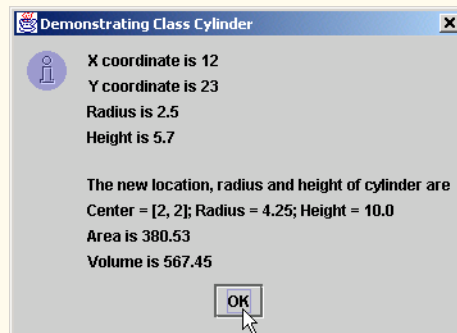


Fig. 9.15 Testing class **Cylinder** (part 2 of 2).

The series of examples in this section nicely demonstrates inheritance and defining and referencing **protected** instance variables. The reader should now be confident with the basics of inheritance. In the next several sections, we show how to program with inheritance hierarchies in a general manner, using polymorphism. Data abstraction, inheritance and polymorphism are the crux of object-oriented programming.

9.10 Introduction to Polymorphism

With *polymorphism*, it is possible to design and implement systems that are more easily *extensible*. Programs can be written to process generically—as superclass objects—objects of all existing classes in a hierarchy. Classes that do not exist during program development can be added with little or no modifications to the generic part of the program—as long as those classes are part of the hierarchy that is being processed generically. The only parts of

a program that need modification are those parts that require direct knowledge of the particular class that is added to the hierarchy. We will study two substantial class hierarchies and will show how objects throughout those hierarchies are manipulated polymorphically.

9.11 Type Fields and `switch` Statements

One means of dealing with objects of many different types is to use a **switch** statement to take an appropriate action on each object, based on that object's type. For example, in a hierarchy of shapes in which each shape has a **shapeType** instance variable, a **switch** structure could determine which **print** method to call based on the object's **shapeType**.

There are many problems with using **switch** logic. The programmer might forget to make such a type test when one is warranted. The programmer might forget to test all possible cases in a **switch**. If a **switch**-based system is modified by adding new types, the programmer might forget to insert the new cases in existing **switch** statements. Every addition or deletion of a class demands that every **switch** statement in the system be modified; tracking these down can be time consuming and error prone.

As we will see, polymorphic programming can eliminate the need for **switch** logic. The programmer can use Java's polymorphism mechanism to perform the equivalent logic automatically, thus avoiding the kinds of errors typically associated with **switch** logic.



Testing and Debugging Tip 9.2

An interesting consequence of using polymorphism is that programs take on a simplified appearance. They contain less branching logic in favor of simpler sequential code. This simplification facilitates testing, debugging, and program maintenance.

9.12 Dynamic Method Binding

Suppose a set of shape classes such as **Circle**, **Triangle**, **Rectangle**, **Square**, are all derived from superclass **Shape**. In object-oriented programming, each of these classes might be endowed with the ability to draw itself. Each class has its own **draw** method, and the **draw** method implementation for each shape is quite different. When drawing a shape, whatever that shape may be, it would be nice to be able to treat all these shapes generically as objects of the superclass **Shape**. Then, to draw any shape, we could simply call method **draw** of superclass **Shape** and let the program determine dynamically (i.e., at execution time) which subclass **draw** method to use from the actual object's type.

To enable this kind of behavior, we declare **draw** in the superclass, and then we override **draw** in each of the subclasses to draw the appropriate shape.



Software Engineering Observation 9.16

When a subclass chooses not to redefine a method, the subclass simply inherits its immediate superclass's method definition.

If we use a superclass reference to refer to a subclass object and invoke the **draw** method, the program will choose the correct subclass's **draw** method dynamically (i.e., at execution time). This is called *dynamic method binding*. Dynamic method binding is an important mechanism for implementing polymorphic processing of objects and will be illustrated in the case studies later in this chapter

9.13 **final** Methods and Classes

We saw in Chapter 7 that variables can be declared **final** to indicate that they cannot be modified after they are declared and that they must be initialized when they are declared. It is also possible to define methods and classes with the **final** modifier.

A method that is declared **final** cannot be overridden in a subclass. Methods that are declared **static** and methods that are declared **private** are implicitly **final**. Because a **final** method's definition can never change, the compiler can optimize the program by removing calls to **final** methods and replacing them with the expanded code of their definitions at each method call location—a technique known as *inlining the code*.

A class that is declared **final** cannot be a superclass (i.e., a class cannot inherit from a **final** class). All methods in a **final** class are implicitly **final**.



Performance Tip 9.2

The compiler can decide to inline a **final** method call and will do so for small, simple **final** methods. Inlining does not violate encapsulation or information hiding (but does improve performance because it eliminates the overhead of making a method call).



Software Engineering Observation 9.17

A class declared **final** cannot be extended, and every method in it is implicitly **final**.



Software Engineering Observation 9.18

In the Java API, the vast majority of the thousands of classes are not declared **final**. This enables inheritance and polymorphic processing—the fundamental capabilities of object-oriented programming. However, in some cases it is important to declare classes **final**—typically for security¹ or performance reasons.

9.14 Abstract Superclasses and Concrete Classes

When we think of a class as a type, we assume that objects of that type will be instantiated. However, there are cases in which it is useful to define classes for which the programmer never intends to instantiate any objects. Such classes are called *abstract classes*. Because these are used as superclasses in inheritance situations, we will normally refer to them as *abstract superclasses*. No objects of abstract superclasses can be instantiated.

The sole purpose of an abstract class is to provide an appropriate superclass from which other classes may inherit interface and/or implementation (we will see examples of each shortly). Classes from which objects can be instantiated are called *concrete classes*.

1. Class **String** is an example of a **final** class. This class cannot be extended, so programs that use **Strings** can rely on the functionality of **String** objects as specified in the Java API. Making the class **final** also prevents programmers from creating subclasses that might bypass security restrictions. For example, when a Java program attempts to open a file on your computer, the program supplies a **String** representing the name of the file. In many cases, opening a file is subject to security restrictions. If it were possible to create a subclass of **String**, that subclass might be implemented in a manner that enables it to specify one **String** to pass a security or permissions test, then specify a different name when the program actually opens the file. For more information on final classes and methods visit: java.sun.com/docs/books/tutorial/java/javaOO/final.html [Note: **Strings** are covered in detail in Chapter 10 and file processing is covered in detail in Chapter 16.]

We could have an abstract superclass **TwoDimensionalObject** and derive concrete classes, such as **Square**, **Circle**, **Triangle**. We could also have an abstract superclass **ThreeDimensionalObject** and derive such concrete classes as **Cube**, **Sphere** and **Cylinder**. Abstract superclasses are too generic to define real objects; we need to be more specific before we can think of instantiating objects. For example, if someone tells you to “draw the shape,” what shape would you draw? Concrete classes provide the specifics that make it reasonable to instantiate objects.

A class is made abstract by declaring it with keyword **abstract**. A hierarchy does not need to contain any **abstract** classes, but, as we will see, many good object-oriented systems have class hierarchies headed by **abstract** superclasses. In some cases, **abstract** classes constitute the top few levels of the hierarchy. A good example of this is the shape hierarchy in Fig. 9.3. The hierarchy begins with **abstract** superclass **Shape**. On the next level down we have two more **abstract** superclasses, namely **TwoDimensionalShape** and **ThreeDimensionalShape**. The next level down would start defining concrete classes for such two-dimensional shapes as **Circle** and **Square** and for such three-dimensional shapes as **Sphere** and **Cube**.

9.15 Polymorphism Examples

Here is an example of polymorphism. If class **Rectangle** is derived from class **Quadrilateral**, then a **Rectangle** object is a more specific version of a **Quadrilateral** object. An operation (such as calculating the perimeter or the area) that can be performed on an object of class **Quadrilateral** can also be performed on an object of class **Rectangle**. Such operations can also be performed on other “kinds of” **Quadrilaterals**, such as **Squares**, **Parallelograms** and **Trapezoids**. When a request is made through a superclass reference to use a method, Java chooses the correct overridden method polymorphically in the appropriate subclass associated with the object.

Here is another example of polymorphism. Suppose we have a video game that manipulates objects of many varieties, including objects of class **Martian**, **Venutian**, **Plutonian**, **SpaceShip**, **LaserBeam** and the like. Each of these classes extends a common superclass, **GamePiece**, that contains a method called **drawYourself**. This method is defined by each subclass. A Java screen manager program would simply maintain some kind of collection (such as a **GamePiece** array) of references to objects of these various classes. To refresh the screen periodically, the screen manager would simply send each object the same message, namely **drawYourself**. Each object would respond in its own unique way. A **Martian** object would draw itself with the appropriate number of antennae. A **SpaceShip** object would draw itself bright and silvery. A **LaserBeam** object would draw itself as a bright red beam across the screen. Thus, the same message sent to a variety of objects would have “many forms” of results—hence the term *polymorphism*.

Such a polymorphic screen manager makes it especially easy to add new types of objects to a system with minimal impact. Suppose we want to add **Mercurians** to our video game. We certainly have to build a new class **Mercurian** that extends **GamePiece** and provides its own definition of the **drawYourself** method. Then, when objects of class **Mercurian** appear in the collection, the screen manager need not be modified. It simply sends the message **drawYourself** to every object in the collection

regardless of the object's type, so the new **Mercurian** objects just “fit right in.” Thus, with polymorphism, new types of objects not even envisioned when a system is created can be added without modifications to the system (other than the new class itself, of course).

Through the use of polymorphism, one method call can cause different actions to occur, depending on the type of the object receiving the call. This gives the programmer tremendous expressive capability. We will see examples of the power of polymorphism in the next several sections.



Software Engineering Observation 9.19

With polymorphism, the programmer can deal in generalities and let the execution-time environment concern itself with the specifics. The programmer can command a wide variety of objects to behave in manners appropriate to those objects without even knowing the types of those objects.



Software Engineering Observation 9.20

Polymorphism promotes extensibility: Software written to invoke polymorphic behavior is written independent of the types of the objects to which messages (i.e., method calls) are sent. Thus, new types of objects that can respond to existing messages can be added into such a system without modifying the base system.



Software Engineering Observation 9.21

*If a method is declared **final**, it cannot be overridden in subclasses, so method calls may not be sent polymorphically to objects of those subclasses. The method call may still be sent to subclasses, but they will all respond identically rather than polymorphically.*



Software Engineering Observation 9.22

*An **abstract** class defines a common interface for the various members of a class hierarchy. The **abstract** class contains methods that will be defined in the subclasses. All classes in the hierarchy can use this same interface through polymorphism.*

Although we cannot instantiate objects of **abstract** superclasses, we *can* declare references to **abstract** superclasses. Such references can be used to enable polymorphic manipulations of subclass objects when such objects are instantiated from concrete classes.

Let us consider more applications of polymorphism. A screen manager needs to display a variety of objects, including new types of objects that will be added to the system even after the screen manager is written. The system may need to display various shapes (the superclass is **Shape**), such as **Circle**, **Triangle** and **Rectangle**. Each shape class is derived from superclass **Shape**. The screen manager uses superclass **Shape** references to manage the objects to be displayed. To draw any object (regardless of the level at which that object appears in the inheritance hierarchy), the screen manager uses a superclass reference to the object and simply sends a **draw** message to the object. Method **draw** has been declared **abstract** in superclass **Shape** and has been overridden in each of the subclasses. Each **Shape** object knows how to draw itself. The screen manager does not have to worry about what type each object is or whether the screen manager has seen objects of that type before—the screen manager simply tells each object to **draw** itself.

Polymorphism is particularly effective for implementing layered software systems. In operating systems, for example, each type of physical device could operate quite differently from the others. Even so, commands to *read* or *write* data from and to devices can have a

certain uniformity. The *write* message sent to a device-driver object needs to be interpreted specifically in the context of that device driver and how that device driver manipulates devices of a specific type. However, the *write* call itself is really no different from the *write* to any other device in the system—simply place some number of bytes from memory onto that device. An object-oriented operating system might use an **abstract** superclass to provide an interface appropriate for all device drivers. Then, through inheritance from that **abstract** superclass, subclasses are formed that all operate similarly. The capabilities (i.e., the **public** interface) offered by the device drivers are provided as **abstract** methods in the **abstract** superclass. The implementations of these **abstract** methods are provided in the subclasses that correspond to the specific types of device drivers.

It is common in object-oriented programming to define an *iterator class* that can walk through all the objects in a collection (such as an array). If you want to print a list of objects in a linked list, for example, an iterator object can be instantiated that will return the next element of the linked list each time the iterator is called. Iterators are commonly used in polymorphic programming to walk through an array or a linked list of objects from various levels of a hierarchy. The references in such a list would all be superclass references (see Chapter 19, Data Structures, for more on linked lists). A list of objects of superclass class **TwoDimensionalShape** could contain objects from the classes **Square**, **Circle**, **Triangle** and so on. Sending a **draw** message to each object in the list would, using polymorphism, draw the correct picture on the screen.

9.16 Case Study: A Payroll System Using Polymorphism

Let us use **abstract** classes, **abstract** methods and polymorphism to perform payroll calculations based on the type of employee (Fig. 9.16). We use an **abstract** superclass **Employee**. The subclasses of **Employee** are **Boss** (Fig. 9.17)—paid a fixed weekly salary regardless of the number of hours worked, **CommissionWorker** (Fig. 9.18)—paid a flat base salary plus a percentage of sales, **PieceWorker** (Fig. 9.19)—paid by the number of items produced and **HourlyWorker** (Fig. 9.20)—paid by the hour and receives overtime pay. Each subclass of **Employee** has been declared **final**, because we do not intend to inherit from them again.

An **earnings** method call certainly applies generically to all employees. But the way each person's earnings are calculated depends on the class of the employee, and these classes are all derived from the superclass **Employee**. So **earnings** is declared **abstract** in superclass **Employee** and appropriate implementations of **earnings** are provided for each of the subclasses. Then, to calculate any employee's earnings, the program simply uses a superclass reference to that employee's object and invokes the **earnings** method. In a real payroll system, the various **Employee** objects might be referenced by individual elements in an array of **Employee** references. The program would simply walk through the array one element at a time, using the **Employee** references to invoke the **earnings** method of each object.



Software Engineering Observation 9.23

*If a subclass is derived from a superclass with an **abstract** method, and if no definition is supplied in the subclass for that **abstract** method (i.e., if that method is not overridden in the subclass), that method remains **abstract** in the subclass. Consequently, the subclass is also an **abstract** class and must be explicitly declared as an **abstract** class.*

**Software Engineering Observation 9.24**

The ability to declare an **abstract** method gives the class designer considerable power over how subclasses will be implemented in a class hierarchy. Any new class that wants to inherit from this class is forced to override the **abstract** method (either directly or by inheriting from a class that has overridden the method). Otherwise, that new class will contain an **abstract** method and thus be an **abstract** class, unable to instantiate objects.

**Software Engineering Observation 9.25**

An **abstract** can still have instance data and non**abstract** methods subject to the normal rules of inheritance by subclasses. An **abstract** class can also have constructors.

**Common Programming Error 9.7**

Attempting to instantiate an object of an **abstract** class (i.e., a class that contains one or more **abstract** methods) is a syntax error.

**Common Programming Error 9.8**

It is a syntax error if a class with one or more **abstract** methods is not explicitly declared **abstract**.

Let us consider the **Employee** class (Fig. 9.16). The **public** methods include a constructor that takes the first name and last name as arguments; a **getFirstName** method that returns the first name; a **getLastName** method that returns the last name; a **toString** method that returns the first name and last name separated by a space; and an **abstract** method—**earnings**. Why is this method **abstract**? The answer is that it does not make sense to provide an implementation of this method in the **Employee** class. We cannot calculate the earnings for a generic employee—we must first know *what kind* of employee it is. By making this method **abstract** we are indicating that we will provide an implementation in each concrete subclass, but not in the superclass itself.

```

1  // Fig. 9.16: Employee.java
2  // Abstract base class Employee.
3
4  public abstract class Employee {
5      private String firstName;
6      private String lastName;
7
8      // constructor
9      public Employee( String first, String last )
10     {
11         firstName = first;
12         lastName = last;
13     }
14
15     // get first name
16     public String getFirstName()
17     {
18         return firstName;
19     }
20

```

Fig. 9.16 **Employee abstract** superclass (part 1 of 2).

```

21     // get last name
22     public String getLastName()
23     {
24         return lastName;
25     }
26
27     public String toString()
28     {
29         return firstName + ' ' + lastName;
30     }
31
32     // Abstract method that must be implemented for each
33     // derived class of Employee from which objects
34     // are instantiated.
35     public abstract double earnings();
36
37 } // end class Employee

```

Fig. 9.16 **Employee abstract** superclass (part 2 of 2).

Class **Boss** (Fig. 9.17) is derived from **Employee**. The **public** methods include a constructor that takes a first name, a last name and a weekly salary as arguments and passes the first name and last name to the **Employee** constructor to initialize the **firstName** and **lastName** members of the superclass part of the subclass object. Other **public** methods include a **setWeeklySalary** method to assign a new value to **private** instance variable **weeklySalary**; an **earnings** method defining how to calculate a **Boss**'s earnings; and a **toString** method that forms a **String** containing the type of the employee (i.e., "**Boss:** ") followed by the boss's name.

```

1  // Fig. 9.17: Boss.java
2  // Boss class derived from Employee.
3
4  public final class Boss extends Employee {
5      private double weeklySalary;
6
7      // constructor for class Boss
8      public Boss( String first, String last, double salary )
9      {
10         super( first, last ); // call superclass constructor
11         setWeeklySalary( salary );
12     }
13
14     // set Boss's salary
15     public void setWeeklySalary( double salary )
16     {
17         weeklySalary = ( salary > 0 ? salary : 0 );
18     }
19

```

Fig. 9.17 **Boss extends abstract** class **Employee**.

```

20     // get Boss's pay
21     public double earnings()
22     {
23         return weeklySalary;
24     }
25
26     // get String representation of Boss's name
27     public String toString()
28     {
29         return "Boss: " + super.toString();
30     }
31
32 } // end class Boss

```

Fig. 9.17 **Boss** extends **abstract** class **Employee**.

Class **CommissionWorker** (Fig. 9.18) is derived from **Employee**. The **public** methods include a constructor that takes a first name, a last name, a salary, a commission and a quantity of items sold as arguments and passes the first name and last name to the **Employee** constructor; *set* methods to assign new values to instance variables **salary**, **commission** and **quantity**; an **earnings** method to calculate a **CommissionWorker**'s earnings; and a **toString** method that forms a **String** containing the employee type (i.e., "**Commission worker:** ") followed by the worker's name.

```

1  // Fig. 9.18: CommissionWorker.java
2  // CommissionWorker class derived from Employee
3
4  public final class CommissionWorker extends Employee {
5      private double salary;        // base salary per week
6      private double commission;    // amount per item sold
7      private int quantity;         // total items sold for week
8
9      // constructor for class CommissionWorker
10     public CommissionWorker( String first, String last,
11                             double salary, double commission, int quantity )
12     {
13         super( first, last ); // call superclass constructor
14         setSalary( salary );
15         setCommission( commission );
16         setQuantity( quantity );
17     }
18
19     // set CommissionWorker's weekly base salary
20     public void setSalary( double weeklySalary )
21     {
22         salary = ( weeklySalary > 0 ? weeklySalary : 0 );
23     }
24

```

Fig. 9.18 **CommissionWorker** extends **abstract** class **Employee** (part 1 of 2).

```

25 // set CommissionWorker's commission
26 public void setCommission( double itemCommission )
27 {
28     commission = ( itemCommission > 0 ? itemCommission : 0 );
29 }
30
31 // set CommissionWorker's quantity sold
32 public void setQuantity( int totalSold )
33 {
34     quantity = ( totalSold > 0 ? totalSold : 0 );
35 }
36
37 // determine CommissionWorker's earnings
38 public double earnings()
39 {
40     return salary + commission * quantity;
41 }
42
43 // get String representation of CommissionWorker's name
44 public String toString()
45 {
46     return "Commission worker: " + super.toString();
47 }
48
49 } // end class CommissionWorker

```

Fig. 9.18 **CommissionWorker** extends **abstract** class **Employee** (part 2 of 2).

Class **PieceWorker** (Fig. 9.19) is derived from **Employee**. The **public** methods include a constructor that takes a first name, a last name, a wage per piece and a quantity of items produced as arguments and passes the first name and last name to the **Employee** constructor; **set** methods to assign new values to instance variables **wagePerPiece** and **quantity**; an **earnings** method defining how to calculate a **PieceWorker**'s earnings; and a **toString** method that forms a **String** containing the type of the employee (i.e., "**Piece worker:** ") followed by the pieceworker's name.

```

1 // Fig. 9.19: PieceWorker.java
2 // PieceWorker class derived from Employee
3
4 public final class PieceWorker extends Employee {
5     private double wagePerPiece; // wage per piece output
6     private int quantity;        // output for week
7
8     // constructor for class PieceWorker
9     public PieceWorker( String first, String last,
10         double wage, int numberOfItems )
11     {
12         super( first, last ); // call superclass constructor
13         setWage( wage );
14         setQuantity( numberOfItems );
15     }

```

Fig. 9.19 **PieceWorker** extends **abstract** class **Employee** (part 1 of 2).

```

16
17 // set PieceWorker's wage
18 public void setWage( double wage )
19 {
20     wagePerPiece = ( wage > 0 ? wage : 0 );
21 }
22
23 // set number of items output
24 public void setQuantity( int numberOfItems )
25 {
26     quantity = ( numberOfItems > 0 ? numberOfItems : 0 );
27 }
28
29 // determine PieceWorker's earnings
30 public double earnings()
31 {
32     return quantity * wagePerPiece;
33 }
34
35 public String toString()
36 {
37     return "Piece worker: " + super.toString();
38 }
39
40 } // end class PieceWorker

```

Fig. 9.19 **PieceWorker** extends **abstract** class **Employee** (part 2 of 2).

Class **HourlyWorker** (Fig. 9.20) is derived from **Employee**. The **public** methods include a constructor that takes a first name, a last name, a wage and the number of hours worked as arguments and passes the first name and last name to the **Employee** constructor; **set** methods to assign new values to instance variables **wage** and **hours**; an **earnings** method defining how to calculate an **HourlyWorker**'s earnings; and a **toString** method that forms a **String** containing the type of the employee (i.e., "Hourly worker: ") followed by the hourly worker's name.

```

1 // Fig. 9.20: HourlyWorker.java
2 // Definition of class HourlyWorker
3
4 public final class HourlyWorker extends Employee {
5     private double wage; // wage per hour
6     private double hours; // hours worked for week
7
8     // constructor for class HourlyWorker
9     public HourlyWorker( String first, String last,
10         double wagePerHour, double hoursWorked )
11     {
12         super( first, last ); // call superclass constructor
13         setWage( wagePerHour );
14         setHours( hoursWorked );
15     }

```

Fig. 9.20 **HourlyWorker** extends **abstract** class **Employee** (part 1 of 2).

```

16
17     // Set the wage
18     public void setWage( double wagePerHour )
19     {
20         wage = ( wagePerHour > 0 ? wagePerHour : 0 );
21     }
22
23     // Set the hours worked
24     public void setHours( double hoursWorked )
25     {
26         hours = ( hoursWorked >= 0 && hoursWorked < 168 ?
27             hoursWorked : 0 );
28     }
29
30     // Get the HourlyWorker's pay
31     public double earnings() { return wage * hours; }
32
33     public String toString()
34     {
35         return "Hourly worker: " + super.toString();
36     }
37
38 } // end class HourlyWorker

```

Fig. 9.20 **HourlyWorker** extends **abstract** class **Employee** (part 2 of 2).

Method **main** of the **Test** application (Fig. 9.21) begins by declaring **Employee** reference, **ref**. Each of the types of **Employees** is handled similarly in **main**, so we will discuss only the case in which **main** deals with a **Boss** object.

```

1  // Fig. 9.21: Test.java
2  // Driver for Employee hierarchy
3
4  // Java core packages
5  import java.text.DecimalFormat;
6
7  // Java extension packages
8  import javax.swing.JOptionPane;
9
10 public class Test {
11
12     // test Employee hierarchy
13     public static void main( String args[] )
14     {
15         Employee employee; // superclass reference
16         String output = "";
17
18         Boss boss = new Boss( "John", "Smith", 800.0 );
19

```

Fig. 9.21 Testing the **Employee** class hierarchy using an **abstract** superclass.

```

1      CommissionWorker commisionWorker =
2          new CommissionWorker(
3              "Sue", "Jones", 400.0, 3.0, 150 );
4
5      PieceWorker pieceWorker =
6          new PieceWorker( "Bob", "Lewis", 2.5, 200 );
7
8      HourlyWorker hourlyWorker =
9          new HourlyWorker( "Karen", "Price", 13.75, 40 );
10
11      DecimalFormat precision2 = new DecimalFormat( "0.00" );
12
13      // Employee reference to a Boss
14      employee = boss;
15
16      output += employee.toString() + " earned $" +
17          precision2.format( employee.earnings() ) + "\n" +
18          boss.toString() + " earned $" +
19          precision2.format( boss.earnings() ) + "\n";
20
21      // Employee reference to a CommissionWorker
22      employee = commissionWorker;
23
24      output += employee.toString() + " earned $" +
25          precision2.format( employee.earnings() ) + "\n" +
26          commissionWorker.toString() + " earned $" +
27          precision2.format(
28              commissionWorker.earnings() ) + "\n";
29
30      // Employee reference to a PieceWorker
31      employee = pieceWorker;
32
33      output += employee.toString() + " earned $" +
34          precision2.format( employee.earnings() ) + "\n" +
35          pieceWorker.toString() + " earned $" +
36          precision2.format( pieceWorker.earnings() ) + "\n";
37
38      // Employee reference to an HourlyWorker
39      employee = hourlyWorker;
40
41      output += employee.toString() + " earned $" +
42          precision2.format( employee.earnings() ) + "\n" +
43          hourlyWorker.toString() + " earned $" +
44          precision2.format( hourlyWorker.earnings() ) + "\n";
45
46      JOptionPane.showMessageDialog( null, output,
47          "Demonstrating Polymorphism",
48          JOptionPane.INFORMATION_MESSAGE );
49
50      System.exit( 0 );
51  }
52
53  } // end class Test

```

Fig. 9.21 Testing the **Employee** class hierarchy using an **abstract** superclass.

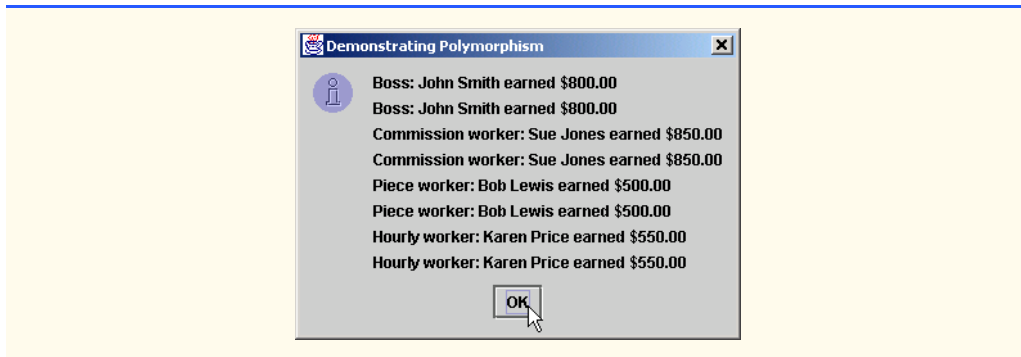


Fig. 9.21 Testing the **Employee** class hierarchy using an **abstract** superclass.

Line 18 instantiates subclass object class **Boss** and provides various constructor arguments, including a first name, a last name and a fixed weekly salary. The new object is assigned to **Boss** reference **boss**.

Line 33 assigns to superclass **Employee** reference **employee** a reference to the subclass **Boss** object to which **boss** refers. This is precisely what we must do to effect polymorphic behavior.

The method call in line 35 invokes the **toString** method of the object referenced by **employee**. The system determines that the referenced object is a **Boss** and invokes subclass **Boss**'s **toString** method—again, polymorphic behavior. This method call is an example of dynamic method binding—the decision as to what method to invoke is deferred until execution time.

The method call in line 36 invokes the **earnings** method of the object to which **employee** refers. The system determines that the object is a **Boss** and invokes the subclass **Boss**'s **earnings** method rather than the superclass's **earnings** method. This is also an example of dynamic method binding.

The method call in line 37 explicitly invokes the **Boss** version of method **toString** by following reference **boss** with the dot operator. We included line 37 for comparison purposes to ensure that the dynamically bound method invoked with **employee.toString()** was indeed the proper method.

The method call in line 38 explicitly invokes the **Boss** version of method **earnings** by using the dot operator with the specific **Boss** reference **boss**. This call is also included for comparison purposes to ensure that the dynamically bound method invoked with **employee.earnings()** was indeed the proper method.

To prove that the superclass reference **employee** can be used to invoke **toString** and **earnings** for the other types of employees, lines 41, 50 and 58 each assign a different type of employee object (**CommissionWorker**, **PieceWorker** and **HourlyWorker**, respectively) to the superclass reference **employee**; then the two methods are called after each assignment to show that Java is always capable of determining the type of the referenced object before invoking a method.

9.17 New Classes and Dynamic Binding

Polymorphism certainly works nicely when all possible classes are known in advance. But it also works when new kinds of classes are added to systems.

New classes are accommodated by dynamic method binding (also called *late binding*). An object's type need not be known at compile time for a polymorphic call to be compiled. At execution time, the call is matched with the method of the called object.

A screen manager program can now handle (without recompilation) new types of display objects as they are added to the system. The **draw** method call remains the same. The new objects themselves each contain a **draw** method implementing the actual drawing capabilities. This makes it easy to add new capabilities to systems with minimal impact. It also promotes software reuse.



Performance Tip 9.3

The kinds of polymorphic manipulations made possible with dynamic binding can also be accomplished by using hand-coded **switch** logic based on type fields in objects. The polymorphic code generated by the Java compiler runs with performance comparable to that of efficiently coded **switch** logic.



Software Engineering Observation 9.26

Java provides mechanisms for loading classes into a program dynamically to enhance the functionality of an executing program. In particular, **static** method **forName** of class **Class** (package **java.lang**) can be used to load a class definition, then create new objects of that class for use in a program. This concept is beyond the scope of this book. For more information, see the online API documentation for **Class**.

9.18 Case Study: Inheriting Interface and Implementation

Our next example (Fig. 9.22–Fig. 9.26) reexamines the **Point**, **Circle**, **Cylinder** hierarchy, except that we now head the hierarchy with **abstract** superclass **Shape** (Fig. 9.22). This hierarchy mechanically demonstrates the power of polymorphism. In the exercises, we explore a more realistic hierarchy of shapes.

Shape contains **abstract** method **getName**, so **Shape** must be declared an **abstract** superclass. **Shape** contains two other methods, **area** and **volume**, each of which has an implementation that returns zero by default. **Point** inherits these implementations from **Shape**. This makes sense because both the area and volume of a point are zero. **Circle** inherits the **volume** method from **Point**, but **Circle** provides its own implementation for the **area** method. **Cylinder** provides its own implementations for both the **area** (interpreted as the surface area of the cylinder) and **volume** methods.

In this example, class **Shape** is used to define a set of methods that all **Shapes** in our hierarchy have in common. Defining these methods in class **Shape** enables us to generically call these methods through a **Shape** reference. Remember, the only methods that can be called through any reference are those **public** methods defined in the reference's declared class type and any **public** methods inherited into that class. Thus, we can call **Object** and **Shape** methods through a **Shape** reference.

Note that although **Shape** is an **abstract** superclass, it still contains implementations of methods **area** and **volume**, and these implementations are inheritable. The **Shape** class provides an inheritable interface (set of services) in the form of three methods that all classes of the hierarchy will contain. The **Shape** class also provides some implementations that subclasses in the first few levels of the hierarchy will use.

This case study emphasizes that a subclass can inherit interface and/or implementation from a superclass.



Software Engineering Observation 9.27

Hierarchies designed for implementation inheritance tend to have their functionality high in the hierarchy—each new subclass inherits one or more methods that were defined in a superclass and uses the superclass definitions.



Software Engineering Observation 9.28

Hierarchies designed for interface inheritance tend to have their functionality lower in the hierarchy—a superclass specifies one or more methods that should be called identically for each object in the hierarchy (i.e., they have the same signature), but the individual subclasses provide their own implementations of the method(s).

Superclass **Shape** (Fig. 9.22) extends **Object**, consists of three **public** methods and does not contain any data (although it could). Method **getName** is **abstract**, so it is overridden in each of the subclasses. Methods **area** and **volume** are defined to return **0.0**. These methods are overridden in subclasses when it is appropriate for those classes to have a different **area** calculation (classes **Circle** and **Cylinder**) and/or a different **volume** calculation (class **Cylinder**).

Class **Point** (Fig. 9.23) is derived from **Shape**. A **Point** has an area of 0.0 and a volume of 0.0, so the superclass methods **area** and **volume** are not overridden here—they are inherited as defined in **Shape**. Other methods include **setPoint** to assign new **x** and **y** coordinates to a **Point** and **getX** and **getY** to return the **x** and **y** coordinates of a **Point**. Method **getName** is an implementation of the **abstract** method in the superclass. If this method were not defined, class **Point** would be an abstract class.

```

1  // Fig. 9.22: Shape.java
2  // Definition of abstract base class Shape
3
4  public abstract class Shape extends Object {
5
6      // return shape's area
7      public double area()
8      {
9          return 0.0;
10     }
11
12     // return shape's volume
13     public double volume()
14     {
15         return 0.0;
16     }
17
18     // abstract method must be defined by concrete subclasses
19     // to return appropriate shape name
20     public abstract String getName();
21
22 } // end class Shape

```

Fig. 9.22 **Shape** abstract superclass for **Point**, **Circle**, **Cylinder** hierarchy.

```
1 // Fig. 9.23: Point.java
2 // Definition of class Point
3
4 public class Point extends Shape {
5     protected int x, y; // coordinates of the Point
6
7     // no-argument constructor
8     public Point()
9     {
10         setPoint( 0, 0 );
11     }
12
13     // constructor
14     public Point( int xCoordinate, int yCoordinate )
15     {
16         setPoint( xCoordinate, yCoordinate );
17     }
18
19     // set x and y coordinates of Point
20     public void setPoint( int xCoordinate, int yCoordinate )
21     {
22         x = xCoordinate;
23         y = yCoordinate;
24     }
25
26     // get x coordinate
27     public int getX()
28     {
29         return x;
30     }
31
32     // get y coordinate
33     public int getY()
34     {
35         return y;
36     }
37
38     // convert point into String representation
39     public String toString()
40     {
41         return "[" + x + ", " + y + "]";
42     }
43
44     // return shape name
45     public String getName()
46     {
47         return "Point";
48     }
49
50 } // end class Point
```

Fig. 9.23 **Point** subclass of abstract class **Shape**.

Class **Circle** (Fig. 9.24) is derived from **Point**. A **Circle** has a volume of 0.0, so superclass method **volume** is not overridden—it is inherited from class **Point**, which inherited it from **Shape**. A **Circle** has an area different from that of a **Point**, so the **area** method is overridden. Method **getName** is an implementation of the **abstract** method in the superclass. If this method is not overridden here, the **Point** version of **getName** would be inherited. Other methods include **setRadius** to assign a new **radius** to a **Circle** and **getRadius** to return the **radius** of a **Circle**.

```

1  // Fig. 9.24: Circle.java
2  // Definition of class Circle
3
4  public class Circle extends Point { // inherits from Point
5      protected double radius;
6
7      // no-argument constructor
8      public Circle()
9      {
10         // implicit call to superclass constructor here
11         setRadius( 0 );
12     }
13
14     // constructor
15     public Circle( double circleRadius, int xCoordinate,
16                 int yCoordinate )
17     {
18         // call superclass constructor
19         super( xCoordinate, yCoordinate );
20
21         setRadius( circleRadius );
22     }
23
24     // set radius of Circle
25     public void setRadius( double circleRadius )
26     {
27         radius = ( circleRadius >= 0 ? circleRadius : 0 );
28     }
29
30     // get radius of Circle
31     public double getRadius()
32     {
33         return radius;
34     }
35
36     // calculate area of Circle
37     public double area()
38     {
39         return Math.PI * radius * radius;
40     }
41

```

Fig. 9.24 **Circle** subclass of **Point**—indirect subclass of **abstract** class **Shape** (part 1 of 2).

```

42 // convert Circle to a String representation
43 public String toString()
44 {
45     return "Center = " + super.toString() +
46         "; Radius = " + radius;
47 }
48
49 // return shape name
50 public String getName()
51 {
52     return "Circle";
53 }
54
55 } // end class Circle

```

Fig. 9.24 **Circle** subclass of **Point**—indirect subclass of **abstract** class **Shape** (part 2 of 2).



Software Engineering Observation 9.29

A subclass always inherits the most recently defined version of each **public** and **protected** method from its direct and indirect superclasses.

Class **Cylinder** (Fig. 9.25) is derived from **Circle**. A **Cylinder** has area and volume different from those of class **Circle**, so the **area** and **volume** methods are both overridden. Method **getName** is an implementation of the **abstract** method in the superclass. If this method had not been overridden here, the **Circle** version of **getName** would be inherited. Other methods include **setHeight** to assign a new **height** to a **Cylinder** and **getHeight** to return the **height** of a **Cylinder**.

```

1 // Fig. 9.25: Cylinder.java
2 // Definition of class Cylinder.
3
4 public class Cylinder extends Circle {
5     protected double height; // height of Cylinder
6
7     // no-argument constructor
8     public Cylinder()
9     {
10         // implicit call to superclass constructor here
11         setHeight( 0 );
12     }
13
14     // constructor
15     public Cylinder( double cylinderHeight,
16         double cylinderRadius, int xCoordinate,
17         int yCoordinate )
18     {
19         // call superclass constructor
20         super( cylinderRadius, xCoordinate, yCoordinate );

```

Fig. 9.25 **Cylinder** subclass of **Circle**—indirect subclass of **abstract** class **Shape** (part 1 of 2).

```

21
22     setHeight( cylinderHeight );
23 }
24
25 // set height of Cylinder
26 public void setHeight( double cylinderHeight )
27 {
28     height = ( cylinderHeight >= 0 ? cylinderHeight : 0 );
29 }
30
31 // get height of Cylinder
32 public double getHeight()
33 {
34     return height;
35 }
36
37 // calculate area of Cylinder (i.e., surface area)
38 public double area()
39 {
40     return 2 * super.area() + 2 * Math.PI * radius * height;
41 }
42
43 // calculate volume of Cylinder
44 public double volume()
45 {
46     return super.area() * height;
47 }
48
49 // convert Cylinder to a String representation
50 public String toString()
51 {
52     return super.toString() + "; Height = " + height;
53 }
54
55 // return shape name
56 public String getName()
57 {
58     return "Cylinder";
59 }
60
61 } // end class Cylinder

```

Fig. 9.25 **Cylinder** subclass of **Circle**—indirect subclass of abstract class **Shape** (part 2 of 2).

Method **main** of class **Test** (Fig. 9.26) instantiates **Point** object **point**, **Circle** object **circle** and **Cylinder** object **cylinder** (lines 16–18). Next, array **arrayOfShapes** is instantiated (line 21). This array of superclass **Shape** references will refer to each subclass object instantiated. At line 24, the reference **point** is assigned to array element **arrayOfShapes[0]**. At line 27, the reference **circle** is assigned to array element **arrayOfShapes[1]**. At line 30, the reference **cylinder** is assigned to array element **arrayOfShapes[2]**. Now, each superclass **Shape** reference in the array refers to a subclass object of type **Point**, **Circle** or **Cylinder**.

```

1  // Fig. 9.26: Test.java
2  // Class to test Shape, Point, Circle, Cylinder hierarchy
3
4  // Java core packages
5  import java.text.DecimalFormat;
6
7  // Java extension packages
8  import javax.swing.JOptionPane;
9
10 public class Test {
11
12     // test Shape hierarchy
13     public static void main( String args[] )
14     {
15         // create shapes
16         Point point = new Point( 7, 11 );
17         Circle circle = new Circle( 3.5, 22, 8 );
18         Cylinder cylinder = new Cylinder( 10, 3.3, 10, 10 );
19
20         // create Shape array
21         Shape arrayOfShapes[] = new Shape[ 3 ];
22
23         // aim arrayOfShapes[ 0 ] at subclass Point object
24         arrayOfShapes[ 0 ] = point;
25
26         // aim arrayOfShapes[ 1 ] at subclass Circle object
27         arrayOfShapes[ 1 ] = circle;
28
29         // aim arrayOfShapes[ 2 ] at subclass Cylinder object
30         arrayOfShapes[ 2 ] = cylinder;
31
32         // get name and String representation of each shape
33         String output =
34             point.getName() + ": " + point.toString() + "\n" +
35             circle.getName() + ": " + circle.toString() + "\n" +
36             cylinder.getName() + ": " + cylinder.toString();
37
38         DecimalFormat precision2 = new DecimalFormat( "0.00" );
39
40         // loop through arrayOfShapes and get name,
41         // area and volume of each shape in arrayOfShapes
42         for ( int i = 0; i < arrayOfShapes.length; i++ ) {
43             output += "\n\n" + arrayOfShapes[ i ].getName() +
44                 ": " + arrayOfShapes[ i ].toString() +
45                 "\nArea = " +
46                 precision2.format( arrayOfShapes[ i ].area() ) +
47                 "\nVolume = " +
48                 precision2.format( arrayOfShapes[ i ].volume() );
49         }
50
51         JOptionPane.showMessageDialog( null, output,
52             "Demonstrating Polymorphism",
53             JOptionPane.INFORMATION_MESSAGE );

```

Fig. 9.26 Shape, Point, Circle, Cylinder hierarchy (part 1 of 2).

```

54
55     System.exit( 0 );
56 }
57
58 } // end class Test

```

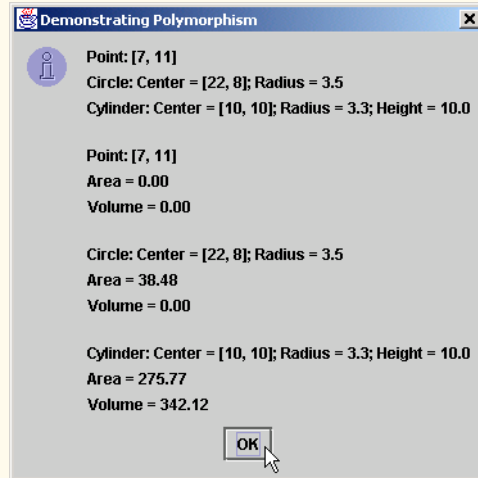


Fig. 9.26 **Shape, Point, Circle, Cylinder** hierarchy (part 2 of 2).

Lines 33 through 36 invoke methods **getName** and **toString** to illustrate that the objects are initialized correctly (as in the first three lines of the screen capture).

Next, the **for** structure at lines 42–49 walks through **arrayOfShapes**, and the following calls are made during each iteration of the loop:

```

arrayOfShapes[ i ].getName()
arrayOfShapes[ i ].toString()
arrayOfShapes[ i ].area()
arrayOfShapes[ i ].volume()

```

Each of these method calls is invoked on the object to which **arrayOfShapes[i]** currently refers. When the compiler looks at each of these calls, it is simply trying to determine whether a **Shape** reference (**arrayOfShapes[i]**) can be used to call these methods. For methods **getName**, **area** and **volume** the answer is yes, because each of these methods is defined in class **Shape**. For method **toString**, the compiler first looks at class **Shape** to determine that **toString** is not defined there, then the compiler proceeds to **Shape**'s superclass (**Object**) to determine whether **Shape** inherited a **toString** method that takes no arguments (as it did, because all **Objects** have a **toString** method).

The screen capture illustrates that all four methods are invoked properly based on the type of the referenced object. First, the string "**Point:** " and the coordinates of the object **point** (**arrayOfShapes[0]**) are output; the area and volume are both **0**. Next, the string "**Circle:** ", the coordinates of object **circle**, and the radius of object **circle** (**arrayOfShapes[1]**) are output; the area of **circle** is calculated and the volume is

0. Finally, the string **"Cylinder: "**, the coordinates of object **cylinder**, the radius of object **cylinder** and the height of object **cylinder** (**arrayOfShapes[2]**) are output; the area of **cylinder** is calculated and the volume of **cylinder** is calculated. All the method calls to **getName**, **toString**, **area** and **volume** are resolved at run-time with dynamic binding.

9.19 Case Study: Creating and Using Interfaces

Our next example (Fig. 9.27–Fig. 9.31) reexamines the **Point**, **Circle**, **Cylinder** hierarchy one last time, replacing **abstract** superclass **Shape** with the interface **Shape** (Fig. 9.27). An interface definition begins with the keyword **interface** and contains a set of **public abstract** methods. Interfaces may also contain **public static final** data. To use an interface, a class must specify that it **implements** the interface and the class must define every method in the interface with the number of arguments and the return type specified in the interface definition. If the class leaves any method of the interface undefined, the class becomes an **abstract** class and must be declared **abstract** in the first line of its class definition. Implementing a interface is like signing a contract with the compiler that states, “I will define all the methods specified by the interface.”



Common Programming Error 9.9

Leaving a method of an **interface** undefined in a class that **implements** the interface results in a compile error indicating that the class must be declared **abstract**.



Software Engineering Observation 9.30

Declaring a **final** reference indicates that the reference always refers to the same object. However, this does not affect the object to which the reference refers—the object’s data still can be modified.

We started using the concept of an interface when we introduced GUI event handling in Chapter 6, “Methods.” Recall that our applet class included **implements ActionListener** (an interface in package **java.awt.event**). The reason we were required to define **actionPerformed** in the applets with event handling is that **ActionListener** is an interface that specifies that **actionPerformed** must be defined. Interfaces are an important part of GUI event handling, as we will discuss in the next section.

An interface is typically used in place of an **abstract** class when there is no default implementation to inherit—i.e., no instance variables and no default method implementations. Like **public abstract** classes, **interfaces** are typically **public** data types, so they are normally defined in files by themselves with the same name as the interface and the **.java** extension.

The definition of interface **Shape** begins in Fig. 9.27 at line 4. Interface **Shape** has **abstract** methods **area**, **volume** and **getName**. By coincidence, all three methods take no arguments. However, this is not a requirement of methods in an interface.

In Fig. 9.28, line 4 indicates that class **Point** extends class **Object** and implements interface **Shape**. Class **Point** provides definitions of all three methods in the interface. Method **area** is defined at lines 45–48. Method **volume** is defined at lines 51–54. Method **getName** is defined at lines 57–60. These three methods satisfy the implementation requirement for the three methods defined in the interface. We have fulfilled our contract with the compiler.

```
1 // Fig. 9.27: Shape.java
2 // Definition of interface Shape
3
4 public interface Shape {
5
6     // calculate area
7     public abstract double area();
8
9     // calculate volume
10    public abstract double volume();
11
12    // return shape name
13    public abstract String getName();
14 }
```

Fig. 9.27 Point, circle, cylinder hierarchy with a **Shape** interface.

```
1 // Fig. 9.28: Point.java
2 // Definition of class Point
3
4 public class Point extends Object implements Shape {
5     protected int x, y; // coordinates of the Point
6
7     // no-argument constructor
8     public Point()
9     {
10         setPoint( 0, 0 );
11     }
12
13     // constructor
14     public Point( int xCoordinate, int yCoordinate )
15     {
16         setPoint( xCoordinate, yCoordinate );
17     }
18
19     // Set x and y coordinates of Point
20     public void setPoint( int xCoordinate, int yCoordinate )
21     {
22         x = xCoordinate;
23         y = yCoordinate;
24     }
25
26     // get x coordinate
27     public int getX()
28     {
29         return x;
30     }
31 }
```

Fig. 9.28 **Point** implementation of interface **Shape** (part 1 of 2).

```

32     // get y coordinate
33     public int getY()
34     {
35         return y;
36     }
37
38     // convert point into String representation
39     public String toString()
40     {
41         return "[" + x + ", " + y + "]";
42     }
43
44     // calculate area
45     public double area()
46     {
47         return 0.0;
48     }
49
50     // calculate volume
51     public double volume()
52     {
53         return 0.0;
54     }
55
56     // return shape name
57     public String getName()
58     {
59         return "Point";
60     }
61
62 } // end class Point

```

Fig. 9.28 **Point** implementation of interface **Shape** (part 2 of 2).

When a class implements an interface, the same *is a* relationship provided by inheritance applies. For example, class **Point** implements **Shape**. Therefore, a **Point** object *is a Shape*. In fact, objects of any class that extends **Point** are also **Shape** objects. Using this relationship, we have maintained the original definitions of class **Circle**, class **Cylinder** and application class **Test** from Section 9.18 (repeated in Fig. 9.29–Fig. 9.31) to illustrate that an interface can be used instead of an **abstract** class to process **Shapes** polymorphically. Notice that the output for the program (Fig. 9.31) is identical to Fig. 9.22. Also, notice that **Object** method **toString** is called through a **Shape** interface reference (line 44).



Software Engineering Observation 9.31

*All methods of class **Object** can be called by using a reference of an interface data type—a reference refers to an object, and all objects have the methods defined by class **Object**.*

One benefit of using interfaces is that a class can implement as many interfaces as it needs in addition to extending a class. To implement more than one interface, simply provide a comma-separated list of interface names after keyword **implements** in the class definition. This is particularly useful in the GUI event-handling mechanism. A class that implements more than one event-listener interface (such as **ActionListener** in earlier

examples) can process different types of GUI events, as we will see in Chapter 12 and Chapter 13.

```

1  // Fig. 9.29: Circle.java
2  // Definition of class Circle
3
4  public class Circle extends Point { // inherits from Point
5      protected double radius;
6
7      // no-argument constructor
8      public Circle()
9      {
10         // implicit call to superclass constructor here
11         setRadius( 0 );
12     }
13
14     // constructor
15     public Circle( double circleRadius, int xCoordinate,
16                 int yCoordinate )
17     {
18         // call superclass constructor
19         super( xCoordinate, yCoordinate );
20
21         setRadius( circleRadius );
22     }
23
24     // set radius of Circle
25     public void setRadius( double circleRadius )
26     {
27         radius = ( circleRadius >= 0 ? circleRadius : 0 );
28     }
29
30     // get radius of Circle
31     public double getRadius()
32     {
33         return radius;
34     }
35
36     // calculate area of Circle
37     public double area()
38     {
39         return Math.PI * radius * radius;
40     }
41
42     // convert Circle to a String representation
43     public String toString()
44     {
45         return "Center = " + super.toString() +
46             "; Radius = " + radius;
47     }

```

Fig. 9.29 **Circle** subclass of **Point**—indirect implementation of interface **Shape** (part 1 of 2).

```

48
49 // return shape name
50 public String getName()
51 {
52     return "Circle";
53 }
54
55 } // end class Circle

```

Fig. 9.29 **Circle** subclass of **Point**—indirect implementation of interface **Shape** (part 2 of 2).

```

1 // Fig. 9.30: Cylinder.java
2 // Definition of class Cylinder.
3
4 public class Cylinder extends Circle {
5     protected double height; // height of Cylinder
6
7     // no-argument constructor
8     public Cylinder()
9     {
10         // implicit call to superclass constructor here
11         setHeight( 0 );
12     }
13
14     // constructor
15     public Cylinder( double cylinderHeight,
16                     double cylinderRadius, int xCoordinate,
17                     int yCoordinate )
18     {
19         // call superclass constructor
20         super( cylinderRadius, xCoordinate, yCoordinate );
21
22         setHeight( cylinderHeight );
23     }
24
25     // set height of Cylinder
26     public void setHeight( double cylinderHeight )
27     {
28         height = ( cylinderHeight >= 0 ? cylinderHeight : 0 );
29     }
30
31     // get height of Cylinder
32     public double getHeight()
33     {
34         return height;
35     }
36
37     // calculate area of Cylinder (i.e., surface area)
38     public double area()
39     {

```

Fig. 9.30 **Cylinder** subclass of **Circle**—indirect implementation of interface **Shape** (part 1 of 2).

```

40     return 2 * super.area() + 2 * Math.PI * radius * height;
41 }
42
43 // calculate volume of Cylinder
44 public double volume()
45 {
46     return super.area() * height;
47 }
48
49 // convert Cylinder to a String representation
50 public String toString()
51 {
52     return super.toString() + "; Height = " + height;
53 }
54
55 // return shape name
56 public String getName()
57 {
58     return "Cylinder";
59 }
60
61 } // end class Cylinder

```

Fig. 9.30 **Cylinder** subclass of **Circle**—indirect implementation of interface **Shape** (part 2 of 2).

```

1 // Fig. 9.31: Test.java
2 // Test Point, Circle, Cylinder hierarchy with interface Shape.
3
4 // Java core packages
5 import java.text.DecimalFormat;
6
7 // Java extension packages
8 import javax.swing.JOptionPane;
9
10 public class Test {
11
12     // test Shape hierarchy
13     public static void main( String args[] )
14     {
15         // create shapes
16         Point point = new Point( 7, 11 );
17         Circle circle = new Circle( 3.5, 22, 8 );
18         Cylinder cylinder = new Cylinder( 10, 3.3, 10, 10 );
19
20         // create Shape array
21         Shape arrayOfShapes[] = new Shape[ 3 ];
22
23         // aim arrayOfShapes[ 0 ] at subclass Point object
24         arrayOfShapes[ 0 ] = point;
25

```

Fig. 9.31 **Shape, Point, Circle, Cylinder** hierarchy (part 1 of 2).

```

26 // aim arrayOfShapes[ 1 ] at subclass Circle object
27 arrayOfShapes[ 1 ] = circle;
28
29 // aim arrayOfShapes[ 2 ] at subclass Cylinder object
30 arrayOfShapes[ 2 ] = cylinder;
31
32 // get name and String representation of each shape
33 String output =
34     point.getName() + ": " + point.toString() + "\n" +
35     circle.getName() + ": " + circle.toString() + "\n" +
36     cylinder.getName() + ": " + cylinder.toString();
37
38 DecimalFormat precision2 = new DecimalFormat( "0.00" );
39
40 // loop through arrayOfShapes and get name,
41 // area and volume of each shape in arrayOfShapes
42 for ( int i = 0; i < arrayOfShapes.length; i++ ) {
43     output += "\n\n" + arrayOfShapes[ i ].getName() +
44         ": " + arrayOfShapes[ i ].toString() +
45         "\nArea = " +
46         precision2.format( arrayOfShapes[ i ].area() ) +
47         "\nVolume = " +
48         precision2.format( arrayOfShapes[ i ].volume() );
49 }
50
51 JOptionPane.showMessageDialog( null, output,
52     "Demonstrating Polymorphism",
53     JOptionPane.INFORMATION_MESSAGE );
54
55 System.exit( 0 );
56 }
57
58 } // end class Test

```

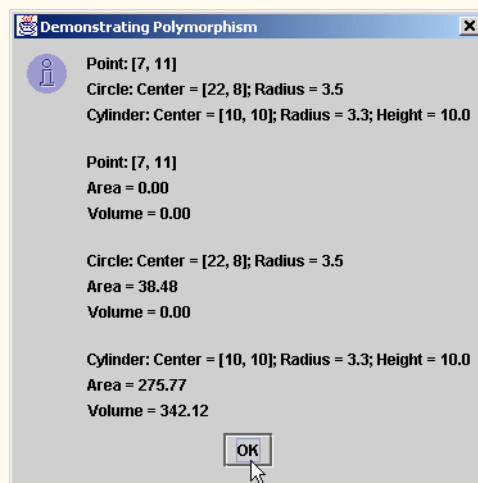


Fig. 9.31 Shape, Point, Circle, Cylinder hierarchy (part 2 of 2).

Another use of interfaces is to define a set of constants that can be used in many class definitions. Consider interface **Constants**

```
public interface Constants {
    public static final int ONE = 1;
    public static final int TWO = 2;
    public static final int THREE = 3;
}
```

Classes that implement interface **Constants** can use **ONE**, **TWO** and **THREE** anywhere in the class definition. A class can even use these constants by importing the interface, then referring to each constant as **Constants.ONE**, **Constants.TWO** and **Constants.THREE**. There are no methods declared in this interface, so a class that implements the interface is not required to provide any method implementations.

9.20 Inner Class Definitions

All the class definitions discussed so far were defined at file scope. For example, if a file contained two classes, one class was not nested in the body of the other class. Java provides a facility called *inner classes*, in which classes can be defined inside other classes. Such classes can be complete class definitions or *anonymous inner class* definitions (classes without a name). Inner classes are used mainly in event handling. However, they have other benefits. For example, the implementation of the queue abstract data type discussed in Section 8.16.1 might use an inner class to represent the objects that store each item currently in the queue. Only the queue data structure requires knowledge of how the objects are stored internally, so the implementation can be hidden by defining an inner class as part of class **Queue**.

Inner classes frequently are used with GUI event handling, we take this opportunity not only to show you inner class definitions, but also to demonstrate an application that executes in its own window. After you complete this example, you will be able to use in your applications the GUI techniques shown only in applets so far.

To demonstrate an inner class definition, Fig. 9.33 uses a simplified version of the **Time3** class (renamed **Time** in Fig. 9.32) from Figure 8.8. Class **Time** provides a default constructor, the same *set/get* methods as Figure 8.8 and a **toString** method. Also, this program defines class **TimeTestWindow** as an application. The application executes in its own window. [Note: We do not discuss class **Time** here, because all its features were discussed in Chapter 8.]

```
1  // Fig. 9.32: Time.java
2  // Time class definition.
3
4  // Java core packages
5  import java.text.DecimalFormat;
6
7  // This class maintains the time in 24-hour format
8  public class Time extends Object {
9      private int hour;        // 0 - 23
10     private int minute;     // 0 - 59
11     private int second;     // 0 - 59
```

Fig. 9.32 **Time** class (part 1 of 3).

```
12
13 // Time constructor initializes each instance variable
14 // to zero. Ensures that Time object starts in a
15 // consistent state.
16 public Time()
17 {
18     setTime( 0, 0, 0 );
19 }
20
21 // Set a new time value using universal time. Perform
22 // validity checks on the data. Set invalid values to zero.
23 public void setTime( int hour, int minute, int second )
24 {
25     setHour( hour );
26     setMinute( minute );
27     setSecond( second );
28 }
29
30 // validate and set hour
31 public void setHour( int h )
32 {
33     hour = ( ( h >= 0 && h < 24 ) ? h : 0 );
34 }
35
36 // validate and set minute
37 public void setMinute( int m )
38 {
39     minute = ( ( m >= 0 && m < 60 ) ? m : 0 );
40 }
41
42 // validate and set second
43 public void setSecond( int s )
44 {
45     second = ( ( s >= 0 && s < 60 ) ? s : 0 );
46 }
47
48 // get hour
49 public int getHour()
50 {
51     return hour;
52 }
53
54 // get minute
55 public int getMinute()
56 {
57     return minute;
58 }
59
60 // get second
61 public int getSecond()
62 {
63     return second;
64 }
```

Fig. 9.32 Time class (part 2 of 3).

```

65
66     // convert to String in standard-time format
67     public String toString()
68     {
69         DecimalFormat twoDigits = new DecimalFormat( "00" );
70
71         return ( ( getHour() == 12 || getHour() == 0 ) ?
72             12 : getHour() % 12 ) + ":" +
73             twoDigits.format( getMinute() ) + ":" +
74             twoDigits.format( getSecond() ) +
75             ( getHour() < 12 ? " AM" : " PM" );
76     }
77
78 } // end class Time

```

Fig. 9.32 Time class (part 3 of 3).

```

1  // Fig. 9.33: TimeTestWindow.java
2  // Demonstrating the Time class set and get methods
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class TimeTestWindow extends JFrame {
12     private Time time;
13     private JLabel hourLabel, minuteLabel, secondLabel;
14     private JTextField hourField, minuteField,
15         secondField, displayField;
16     private JButton exitButton;
17
18     // set up GUI
19     public TimeTestWindow()
20     {
21         super( "Inner Class Demonstration" );
22
23         time = new Time();
24
25         // create an instance of inner class ActionEventHandler
26         ActionEvent handler = new ActionEvent();
27
28         // set up GUI
29         Container container = getContentPane();
30         container.setLayout( new FlowLayout() );
31
32         hourLabel = new JLabel( "Set Hour" );
33         hourField = new JTextField( 10 );
34         hourField.addActionListener( handler );

```

Fig. 9.33 Demonstrating an inner class in a windowed application (part 1 of 4).

```
35     container.add( hourLabel );
36     container.add( hourField );
37
38     minuteLabel = new JLabel( "Set minute" );
39     minuteField = new JTextField( 10 );
40     minuteField.addActionListener( handler );
41     container.add( minuteLabel );
42     container.add( minuteField );
43
44     secondLabel = new JLabel( "Set Second" );
45     secondField = new JTextField( 10 );
46     secondField.addActionListener( handler );
47     container.add( secondLabel );
48     container.add( secondField );
49
50     displayField = new JTextField( 30 );
51     displayField.setEditable( false );
52     container.add( displayField );
53
54     exitButton = new JButton( "Exit" );
55     exitButton.addActionListener( handler );
56     container.add( exitButton );
57
58 } // end constructor
59
60 // display time in displayField
61 public void displayTime()
62 {
63     displayField.setText( "The time is: " + time );
64 }
65
66 // create TimeTestWindow and display it
67 public static void main( String args[] )
68 {
69     TimeTestWindow window = new TimeTestWindow();
70
71     window.setSize( 400, 140 );
72     window.setVisible( true );
73 }
74
75 // inner class definition for handling JTextField and
76 // JButton events
77 private class ActionEventHandler
78     implements ActionListener {
79
80     // method to handle action events
81     public void actionPerformed( ActionEvent event )
82     {
83         // user pressed exitButton
84         if ( event.getSource() == exitButton )
85             System.exit( 0 ); // terminate the application
86     }
```

Fig. 9.33 Demonstrating an inner class in a windowed application (part 2 of 4).

```

87      // user pressed Enter key in hourField
88      else if ( event.getSource() == hourField ) {
89          time.setHour(
90              Integer.parseInt( event.getActionCommand() ) );
91          hourField.setText( "" );
92      }
93
94      // user pressed Enter key in minuteField
95      else if ( event.getSource() == minuteField ) {
96          time.setMinute(
97              Integer.parseInt( event.getActionCommand() ) );
98          minuteField.setText( "" );
99      }
100
101      // user pressed Enter key in secondField
102      else if ( event.getSource() == secondField ) {
103          time.setSecond(
104              Integer.parseInt( event.getActionCommand() ) );
105          secondField.setText( "" );
106      }
107
108      displayTime();
109  }
110
111  } // end inner class ActionEventHandler
112
113  } // end class TimeTestWindow

```

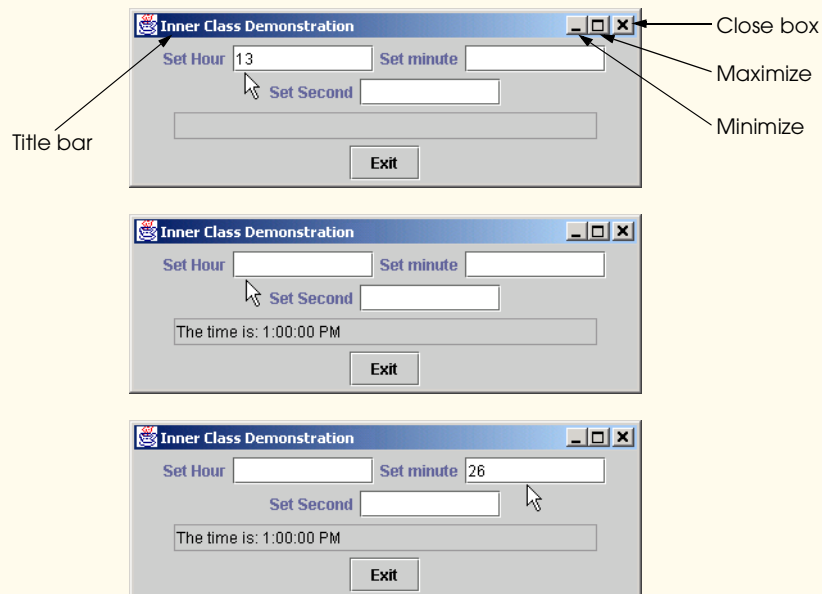


Fig. 9.33 Demonstrating an inner class in a windowed application (part 3 of 4).

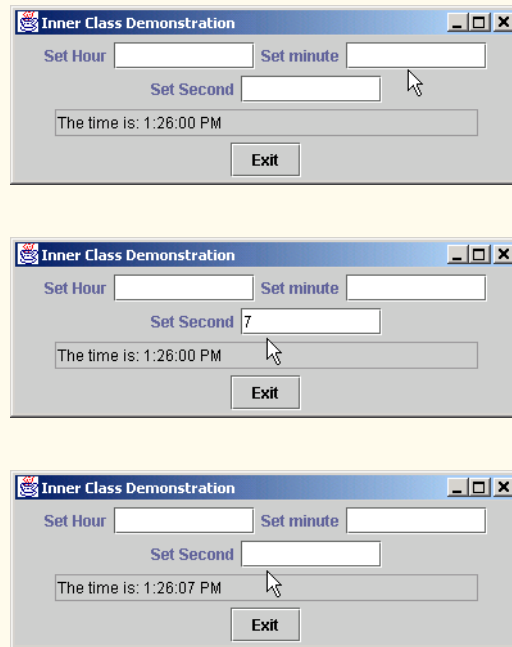


Fig. 9.33 Demonstrating an inner class in a windowed application (part 4 of 4).

In Fig. 9.33, line 11 indicates that class **TimeTestWindow** extends class **JFrame** (from package **javax.swing**) rather than class **JApplet** (as shown in Figure 8.8). Superclass **JFrame** provides the basic attributes and behaviors of a window—a *title bar* and buttons to *minimize*, *maximize* and *close* the window (all labeled in the first screen capture). Class **TimeTestWindow** uses the same GUI components as the applet of Fig. 8.8, except that the button (line 16) is now called **exitButton** and is used to terminate the application.

The **init** method of the applet has been replaced by a constructor (lines 19–54) to guarantee that the window’s GUI components are created as the application begins executing. Method **main** (lines 67–73) defines a **new** object of class **TimeTestWindow** that results in a call to the constructor. Remember, **init** is a special method that is guaranteed to be called when an applet begins execution. However, this program is not an applet, so if we did define the **init** method, it would not be called automatically.

Several new features appear in the constructor. Line 21 calls the superclass **JFrame** constructor with the string **"Inner Class Demonstration"**. This string is displayed in the title bar of the window by class **JFrame**’s constructor. Line 26 defines one instance of our inner class **ActionEventHandler** (defined at lines 77–111) and assigns it to **handler**. This reference is passed to each of the four calls to **addActionListener** (lines 34, 40, 46 and 55) that register the event handlers for each GUI component that generates events in this example (**hourField**, **minuteField**, **secondField** and **exitButton**). Each call to **addActionListener** requires an object of type

ActionListener to be passed as an argument. Actually, **handler** is an **ActionListener**. Line 77 (the first line of the inner class definition) indicates that inner class **ActionEventHandler** implements **ActionListener**. Thus, every object of type **ActionEventHandler** is an **ActionListener**. The requirement that **addActionListener** be passed an object of type **ActionListener** is satisfied! The *is a* relationship is used extensively in the GUI event-handling mechanism, as you will see over the next several chapters. The inner class is defined as **private** because it will be used only in this class definition. Inner classes can be **private**, **protected** or **public**.

An inner class object has a special relationship with the outer class object that creates it. The inner class object is allowed to access directly all the instance variables and methods of the outer class object. The **actionPerformed** method (line 77–105) of class **ActionEventHandler** does just that. Throughout the method, the instance variables **time**, **exitButton**, **hourField**, **minuteField** and **secondField** are used, as is method **displayTime**. Notice that none of these needs a “handle” to the outer class object. This is a free relationship created by the compiler between the outer class and its inner classes.



Software Engineering Observation 9.32

An inner class object is allowed to access directly all the variables and methods of the outer class object that defined it.

This application must be terminated by pressing the **Exit** button. Remember, an application that displays a window must be terminated with a call to **System.exit(0)**. Also note that a window in Java is 0 pixels wide, 0 pixels tall and not displayed by default. Lines 71–72 use methods **resize** and **setVisible** to size the window and display it on the screen. These methods are defined in class **java.awt.Component** originally and inherited into class **JFrame**.

An inner class can also be defined inside a method of a class. Such an inner class has access to its outer class’s members. However, it has limited access to the local variables for the method in which it is defined.



Software Engineering Observation 9.33

*An inner class defined in a method is allowed to access directly all the instance variables and methods of the outer class object that defined it and any **final** local variables in the method.*

The application of Fig. 9.34 modifies class **TimeTestWindow** of Fig. 9.33 to use *anonymous inner classes* defined in methods. An anonymous inner class has no name, so one object of the anonymous inner class must be created at the point where the class is defined in the program. We demonstrate anonymous inner classes two ways in this example. First, we use separate anonymous inner classes that implement an interface (**ActionListener**) to create event handlers for each of the three **JTextFields** **hourField**, **minuteField** and **secondField**. We also demonstrate how to terminate an application when the user clicks the Close box on the window. The event handler is defined as an anonymous inner class that extends a class (**WindowAdapter**). The **Time** class used is identical to Fig. 9.32, so it is not included here. Also, the **Exit** button has been removed from this example.

```
1  // Fig. 9.34: TimeTestWindow.java
2  // Demonstrating the Time class set and get methods
3
4  // Java core packages
5  import java.awt.*;
6  import java.awt.event.*;
7
8  // Java extension packages
9  import javax.swing.*;
10
11 public class TimeTestWindow extends JFrame {
12     private Time time;
13     private JLabel hourLabel, minuteLabel, secondLabel;
14     private JTextField hourField, minuteField,
15         secondField, displayField;
16
17     // set up GUI
18     public TimeTestWindow()
19     {
20         super( "Inner Class Demonstration" );
21
22         // create Time object
23         time = new Time();
24
25         // create GUI
26         Container container = getContentPane();
27         container.setLayout( new FlowLayout() );
28
29         hourLabel = new JLabel( "Set Hour" );
30         hourField = new JTextField( 10 );
31
32         // register hourField event handler
33         hourField.addActionListener(
34
35             // anonymous inner class
36             new ActionListener() {
37
38                 public void actionPerformed((ActionEvent event) )
39                 {
40                     time.setHour(
41                         Integer.parseInt( event.getActionCommand() ) );
42                     hourField.setText( "" );
43                     displayTime();
44                 }
45
46             } // end anonymous inner class
47
48         ); // end call to addActionListener
49
50         container.add( hourLabel );
51         container.add( hourField );
52     }
```

Fig. 9.34 Demonstrating anonymous inner classes (part 1 of 4).

```
53     minuteLabel = new JLabel( "Set minute" );
54     minuteField = new JTextField( 10 );
55
56     // register minuteField event handler
57     minuteField.addActionListener(
58
59         // anonymous inner class
60         new ActionListener() {
61
62             public void actionPerformed((ActionEvent event) )
63             {
64                 time.setMinute(
65                     Integer.parseInt( event.getActionCommand() ) );
66                 minuteField.setText( "" );
67                 displayTime();
68             }
69
70         } // end anonymous inner class
71
72     ); // end call to addActionListener
73
74     container.add( minuteLabel );
75     container.add( minuteField );
76
77     secondLabel = new JLabel( "Set Second" );
78     secondField = new JTextField( 10 );
79
80     secondField.addActionListener(
81
82         // anonymous inner class
83         new ActionListener() {
84
85             public void actionPerformed((ActionEvent event) )
86             {
87                 time.setSecond(
88                     Integer.parseInt( event.getActionCommand() ) );
89                 secondField.setText( "" );
90                 displayTime();
91             }
92
93         } // end anonymous inner class
94
95     ); // end call to addActionListener
96
97     container.add( secondLabel );
98     container.add( secondField );
99
100     displayField = new JTextField( 30 );
101     displayField.setEditable( false );
102     container.add( displayField );
103 }
104
```

Fig. 9.34 Demonstrating anonymous inner classes (part 2 of 4).

```

105 // display time in displayField
106 public void displayTime()
107 {
108     displayField.setText( "The time is: " + time );
109 }
110
111 // create TimeTestWindow, register for its window events
112 // and display it to begin application's execution
113 public static void main( String args[] )
114 {
115     TimeTestWindow window = new TimeTestWindow();
116
117     // register listener for windowClosing event
118     window.addWindowListener(
119
120         // anonymous inner class for windowClosing event
121         new WindowAdapter() {
122
123             // terminate application when user closes window
124             public void windowClosing( WindowEvent event )
125             {
126                 System.exit( 0 );
127             }
128
129         } // end anonymous inner class
130     ); // end call to addWindowListener
131
132     window.setSize( 400, 120 );
133     window.setVisible( true );
134 }
135
136
137 } // end class TimeTestWindow

```

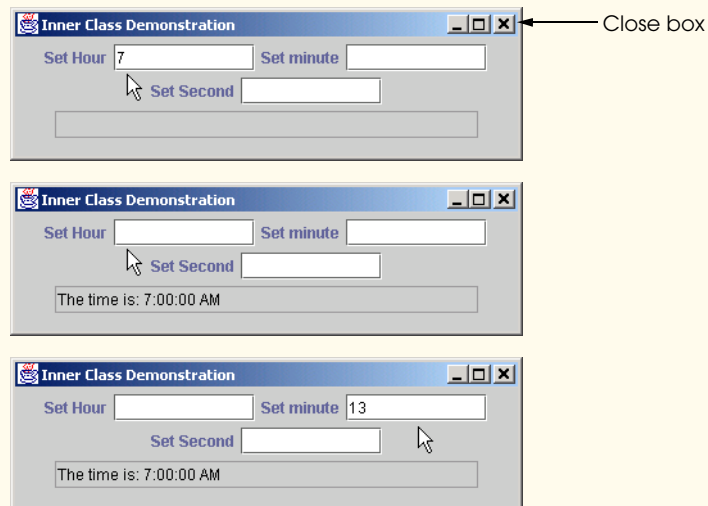


Fig. 9.34 Demonstrating anonymous inner classes (part 3 of 4).

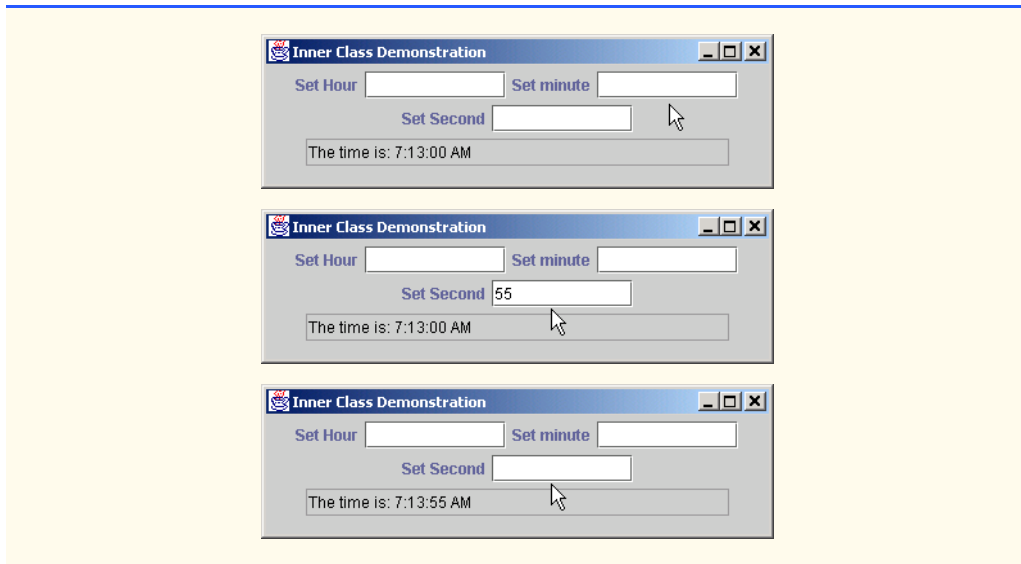


Fig. 9.34 Demonstrating anonymous inner classes (part 4 of 4).

Each of the three **JTextField**s that generate events in this program has a similar anonymous inner class to handle its events, so we discuss only the anonymous inner class for **hourField** here. Lines 33–48 are a call to **hourField**'s **addActionListener** method. The argument to this method must be an object that *is an **ActionListener*** (i.e., any object of a class that implements **ActionListener**). Lines 36–46 use special Java syntax to define an anonymous inner class and create one object of that class that is passed as the argument to **addActionListener**. Line 36 uses operator **new** to create an object. The syntax **ActionListener()** begins the definition of an anonymous inner class that implements interface **ActionListener**. This is similar to beginning a class definition with

```
public class MyHandler implements ActionListener {
```

The parentheses after **ActionListener** indicate a call to the default constructor of the anonymous inner class.

The opening left brace (**{**) at the end of line 36 and the closing right brace (**}**) at line 46 define the body of the class. Lines 38–44 define the **actionPerformed** method that is required in any class that implements **ActionListener**. Method **actionPerformed** is called when the user presses **Enter** while typing in **hourField**.



Software Engineering Observation 9.34

When an anonymous inner class implements an interface, the class must define every method in the interface.

Method **main** creates one instance of class **TimeTestWindow** (line 115), sizes the window (line 133) and displays the window (line 134).

Windows generate a variety of events that are discussed in Chapter 13. For this example we discuss the one event generated when the user clicks the window's close box—a *window closing event*. Lines 118–131 enable the user to terminate the application by

clicking the window's close box (labeled in the first screen capture). Method **addWindowListener** registers a window event listener. The argument to **addWindowListener** must be a reference to an object that *is a WindowListener* (package **java.awt.event**) (i.e., any object of a class that implements **WindowListener**). However, there are seven different methods that must be defined in every class that implements **WindowListener** and we only need one in this example—**windowClosing**. For event handling interfaces with more than one method, Java provides a corresponding class (called an *adapter class*) that already implements all the methods in the interface for you. All you need to do is extend the adapter class and override the methods you require in your program.



Common Programming Error 9.10

Extending an adapter class and misspelling the name of the method you are overriding is a logic error.

Lines 121–129 use special Java syntax to define an anonymous inner class and create one object of that class that is passed as the argument to **addWindowListener**. Line 118 uses operator **new** to create an object. The syntax **WindowAdapter()** begins the definition of an anonymous inner class that extends class **WindowAdapter**. This is similar to beginning a class definition with

```
public class MyHandler extends WindowAdapter {
```

The parentheses after **WindowAdapter** indicate a call to the default constructor of the anonymous inner class. Class **WindowAdapter** implements interface **WindowListener**, so every **WindowAdapter** object *is a WindowListener*—the exact type required for the argument to **addWindowListener**.

The opening left brace (**{**) at the end of line 121 and the closing right brace (**}**) at line 129 define the body of the class. Lines 124–127 override the **windowClosing** method of **WindowAdapter** that is called when the user clicks the window's close box. In this example, **windowClosing** terminates the application.

In the last two examples, we have seen that inner classes can be used to create event handlers and that separate anonymous inner classes can be defined to handle events individually for each GUI component. In Chapter 12 and Chapter 13, we revisit this concept as we discuss the event handling mechanism in detail.

9.21 Notes on Inner Class Definitions

This section presents several notes of interest to programmers regarding the definition and use of inner classes.

1. Compiling a class that contains inner classes results in a separate **.class** file for every class. Inner classes with names have the file name *OuterClassName\$InnerClassName.class*. Anonymous inner classes have the file name *OuterClassName\$#.class*, where **#** starts at 1 and is incremented for each anonymous inner class encountered during compilation.
2. Inner classes with class names can be defined as **public**, **protected**, package access or **private** and are subject to the same usage restrictions as other members of a class.

3. To access the outer class's **this** reference, use *OuterClassName.this*.
4. The outer class is responsible for creating objects of its inner classes. To create an object of another class's inner class, first create an object of the outer class and assign it to a reference (we will call it **ref**). Then use a statement of the following form to create an inner class object:

```
OuterClassName.InnerClassName innerRef = ref.new InnerClassName();
```

5. An inner class can be declared **static**. A **static** inner class does not require an object of its outer class to be defined (whereas a **nonstatic** inner class does). A **static** inner class does not have access to the outer class's **nonstatic** members.

9.22 Type-Wrapper Classes for Primitive Types

Each of the primitive types has a *type-wrapper class*. These classes are called **Character**, **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double** and **Boolean**. Each type-wrapper class enables you to manipulate primitive types as objects of class **Object**. Therefore, values of the primitive data types can be processed polymorphically if they are maintained as objects of the type-wrapper classes. Many of the classes we will develop or reuse manipulate and share **Objects**. These classes cannot polymorphically manipulate variables of primitive types, but they can polymorphically manipulate objects of the type-wrapper classes, because every class ultimately is derived from class **Object**.

Each of the numeric classes—**Byte**, **Short**, **Integer**, **Long**, **Float** and **Double**—inherits from class **Number**. Each of the type wrappers is declared **final**, so their methods are implicitly **final** and may not be overridden. Note that many of the methods that process the primitive data types are defined as **static** methods of the type-wrapper classes. If you need to manipulate a primitive value in your program, first refer to the documentation for the type-wrapper classes—the method you need might already be defined. We will use the type-wrapper classes polymorphically in our study of data structures in Chapters 22 and 23.

9.23 (Optional Case Study) Thinking About Objects: Incorporating Inheritance into the Elevator Simulation

We now examine our elevator simulator design to see whether it might benefit from inheritance. In previous chapters, we have been treating **ElevatorButton** and **FloorButton** as separate classes. These classes, however, have much in common—both have attribute **pressed** and behaviors **pressButton** and **resetButton**. To apply inheritance, we first look for commonality between these classes. We extract this commonality, place it into superclass **Button**, then derive subclasses **ElevatorButton** and **FloorButton** from **Button**.

Let us now examine the similarities between classes **ElevatorButton** and **FloorButton**. Figure 9.35 shows the attributes and operations of each class. Both classes have their attribute (**pressed**) and operations (**pressButton** and **resetButton**) in common.

FloorButton	ElevatorButton
- pressed : Boolean = false	- pressed : Boolean = false
+ resetButton() : void	+ resetButton() : void
+ pressButton() : void	+ pressButton() : void

Fig. 9.35 Attributes and operations of classes **FloorButton** and **ElevatorButton**.

We must now examine whether these objects exhibit distinct behavior. If the **ElevatorButton** and **FloorButton** objects have the same behavior, then we cannot justify using two separate classes to instantiate these objects. However, if these objects have distinct behaviors, we can place the common attributes and operations in a superclass **Button**; then **ElevatorButton** and **FloorButton** inherit both the attributes and operations of **Button**.

We have been treating the **FloorButton** as if it behaves differently from the **ElevatorButton**—the **FloorButton** requests the **Elevator** to move to the **Floor** of the request, and the **ElevatorButton** signals the **Elevator** to move to the opposite **Floor**. Under closer scrutiny, we notice that both the **FloorButton** and the **ElevatorButton** signal the **Elevator** to move to a **Floor**, and the **Elevator** decides whether to move. The **Elevator** will sometimes move in response to a signal from the **FloorButton**, and the **Elevator** will always move in response to a signal from the **ElevatorButton**—however, neither the **FloorButton** nor the **ElevatorButton** decides for the **Elevator** that the **Elevator** must move to the other **Floor**. We conclude that both **FloorButton** and **ElevatorButton** have the same behavior—both signal the **Elevator** to move—so we *combine* (not inherit) classes into one **Button** class and discard class **FloorButton** and **ElevatorButton** from our case study.

We turn our attention to classes **ElevatorDoor** and **FloorDoor**. Figure 9.36 shows the attributes and operations of classes **ElevatorDoor** and **FloorDoor**—these two classes are structurally similar to each other as well, because both classes possess the attribute **open** and the two operations **openDoor** and **closeDoor**. In addition, both the **ElevatorDoor** and **FloorDoor** have the same behavior—they inform a **Person** that a door in the simulation has opened. The **Person** then decides either to enter or exit the **Elevator**, depending on which door opened. In other words, neither the **ElevatorDoor** nor the **FloorDoor** decides for the **Person** that the **Person** must enter or exit the **Elevator**. We combine both **ElevatorDoor** and **FloorDoor** into one **Door** class and eliminate classes **ElevatorDoor** and **FloorDoor** from our case study.²

2. As we continue to discuss inheritance throughout this section, we refer to the class diagram of Fig. 3.23 to determine similarities among objects. However, this diagram contains classes **FloorButton** and **ElevatorButton**, which we have eliminated recently from the case study. Therefore, during our discussion of inheritance, when we mention the **FloorButton**, we refer to that **Floor**'s **Button** object, and when we mention the **ElevatorButton**, we refer to the **Elevator**'s **Button** object.

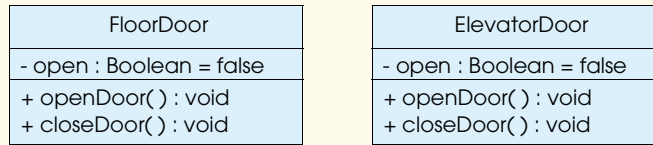


Fig. 9.36 Attributes and operations of classes **FloorDoor** and **ElevatorDoor**.

In Section 4.14, we encountered the problem of representing the location of the **Person**—on what **Floor** is the **Person** located when riding in the **Elevator**? Using inheritance, we may now model a solution. Both the **Elevator** and the two **Floors** are locations at which the **Person** exists in the simulator. In other words, the **Elevator** and the **Floors** are *types of* locations. Therefore, classes **Elevator** and **Floor** should inherit from an abstract superclass called **Location**. We declare class **Location** as abstract, because, for the purposes of our simulation, a location is too generic a term to define a real object. The UML requires that we place abstract class names (and abstract methods) in italics. Superclass **Location** contains the **protected** attribute **locationName**, which contains a **String** value of "**firstFloor**", "**secondFloor**" or "**elevator**". Therefore, only classes **Elevator** and **Floor** have access to **locationName**. In addition, we include the **public** method **getLocationName** to return the name of the location.

We search for more similarities between classes **Floor** and **Elevator**. First of all, according to the class diagram of Fig. 3.23, **Elevator** contains a reference to both a **Button** and a **Door**—the **ElevatorButton** (the **Elevator**'s **Button**) and the **ElevatorDoor** (the **Elevator**'s **Door**). Class **Floor**, through its association with class **ElevatorShaft** (class **ElevatorShaft** "connects" class **Floor**), also contains a reference to a **Button** and a **Door**—the **FloorButton** (that **Floor**'s **Button**) and the **FloorDoor** (that **Floor**'s **Door**). Therefore, in our simulation, the **Location** class, which is the superclass of classes **Elevator** and **Floor**, will contain **public** methods **getButton** and **getDoor**, which return a **Button** or **Door** reference, respectively. Class **Floor** overrides these methods to return the **Button** and **Door** references of that **Floor**, and class **Elevator** overrides these methods to return the **Button** and **Door** references of the **Elevator**. In other words, class **Floor** and **Elevator** exhibit distinct behavior from each other but share attribute **locationName** and methods **getButton** and **getDoor**—therefore, we may use inheritance for these classes.

The UML offers a relationship called a *generalization* to model inheritance. Figure 9.35 is the generalization diagram of superclass **Location** and subclasses **Elevator** and **Floor**. The empty-head arrows specify that classes **Elevator** and **Floor** inherit from class **Location**. Note that attribute **locationName** has an *access modifier* that we have not yet seen—the pound sign (#), indicating that **locationName** is a **protected** member, so **Location** subclasses may access this attribute. Note that classes **Floor** and **Elevator** contain additional attributes and methods that further distinguish these classes.

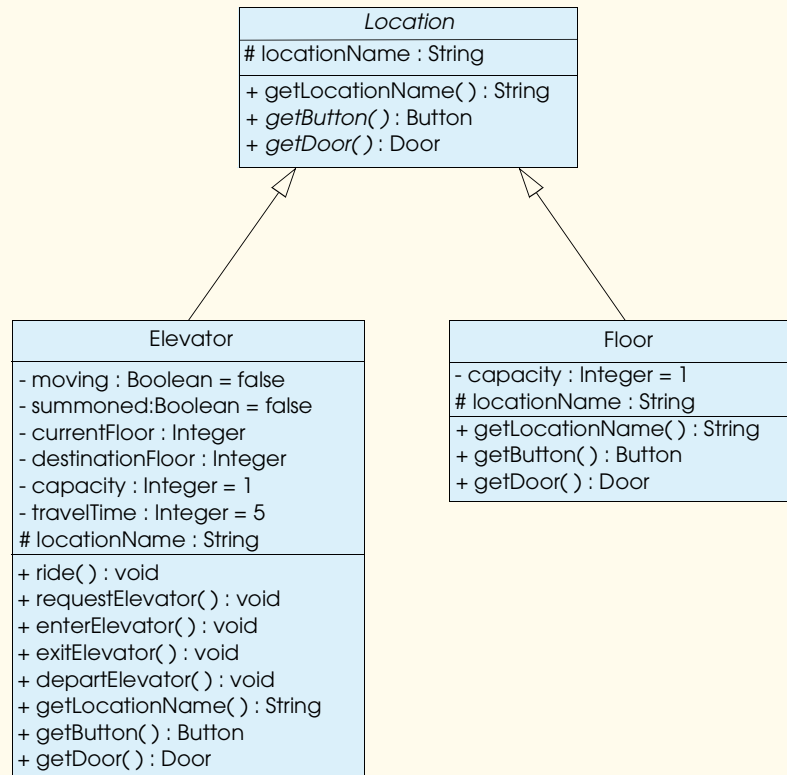


Fig. 9.37 Generalization diagram of superclass **Location** and subclasses **Elevator** and **Floor**.

Classes **Floor** and **Elevator** override methods **getButton** and **getDoor** from their superclass **Location**. In Fig. 9.37, these methods are italicized in class **Location**, indicating that they are *abstract methods*. However, the methods in the subclasses are not italicized—these methods became *concrete methods* by overriding the abstract methods. Class **Person** now contains a **Location** object representing whether the **Person** is on the first or second **Floor** or inside the **Elevator**. We remove the association between **Person** and **Elevator** and the association between **Person** and **Floor** from the class diagram, because the **Location** object acts as both the **Elevator** and **Floor** reference for the **Person**. A **Person** sets its **Location** object to reference the **Elevator** when that **Person** enters the **Elevator**. A **Person** sets its **Location** object to reference a **Floor** when the **Person** is on that **Floor**. Lastly, we assign class **Elevator** two **Location** objects to represent the **Elevator**'s reference to the current **Floor** and destination **Floor** (we originally used integers to describe these references). Figure 9.38 provides an updated class diagram of our model by incorporating inheritance and eliminating classes **FloorButton**, **ElevatorButton**, **FloorDoor** and **ElevatorDoor**.

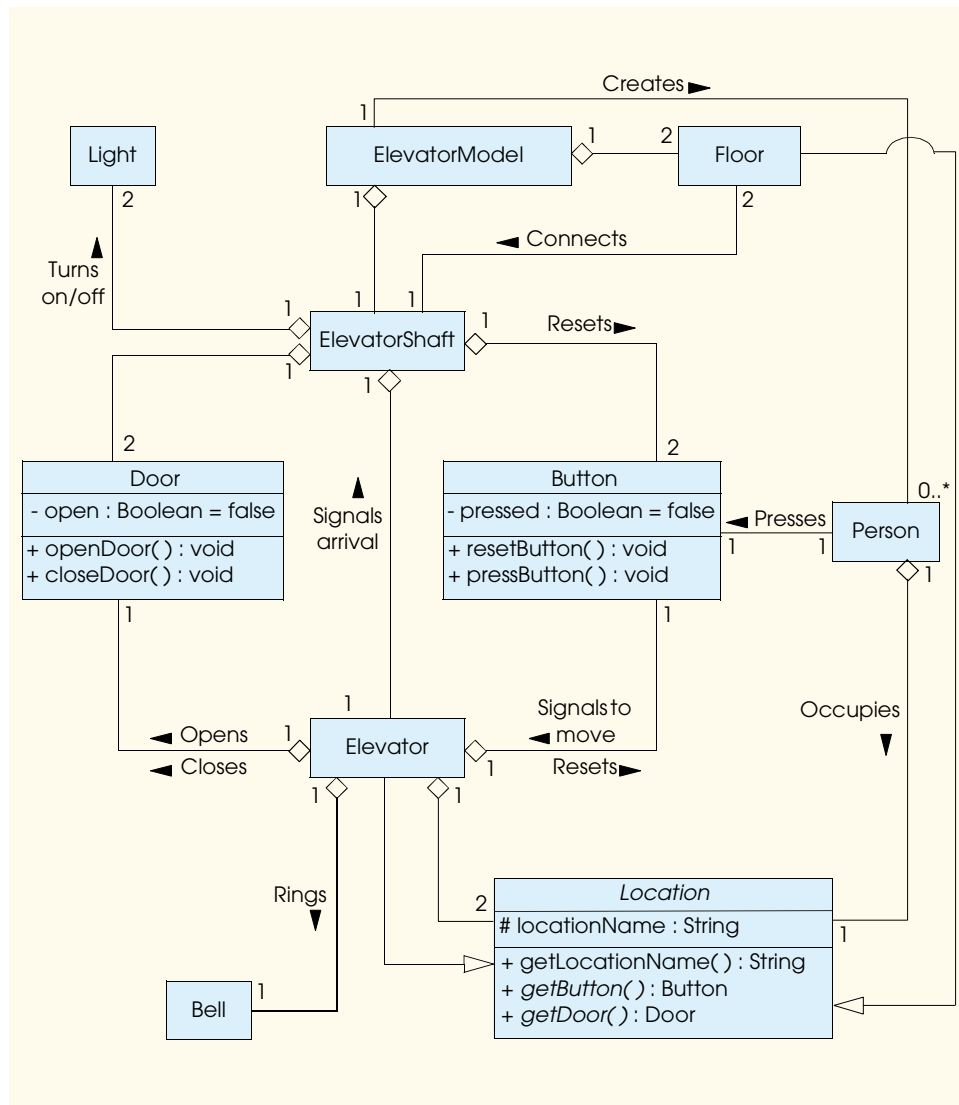


Fig. 9.38 Class diagram of our simulator (incorporating inheritance).

We have allowed a **Person**, occupying a **Location**, to interact with several of the objects in the simulator. For example, a **Person** can press a **Button** or be informed when a **Door** opens from that **Person**'s specific **Location**. In addition, because a **Person** may occupy only one **Location** at a time, that **Person** may interact with only the objects known to that **Location**—a **Person** will never perform an illegal action, such as pressing the first **Floor**'s **Button** while riding the **Elevator**.

We presented the class attributes and operations with access modifiers in Fig. 8.13. Now, we present a modified diagram incorporating inheritance in Fig. 9.39.

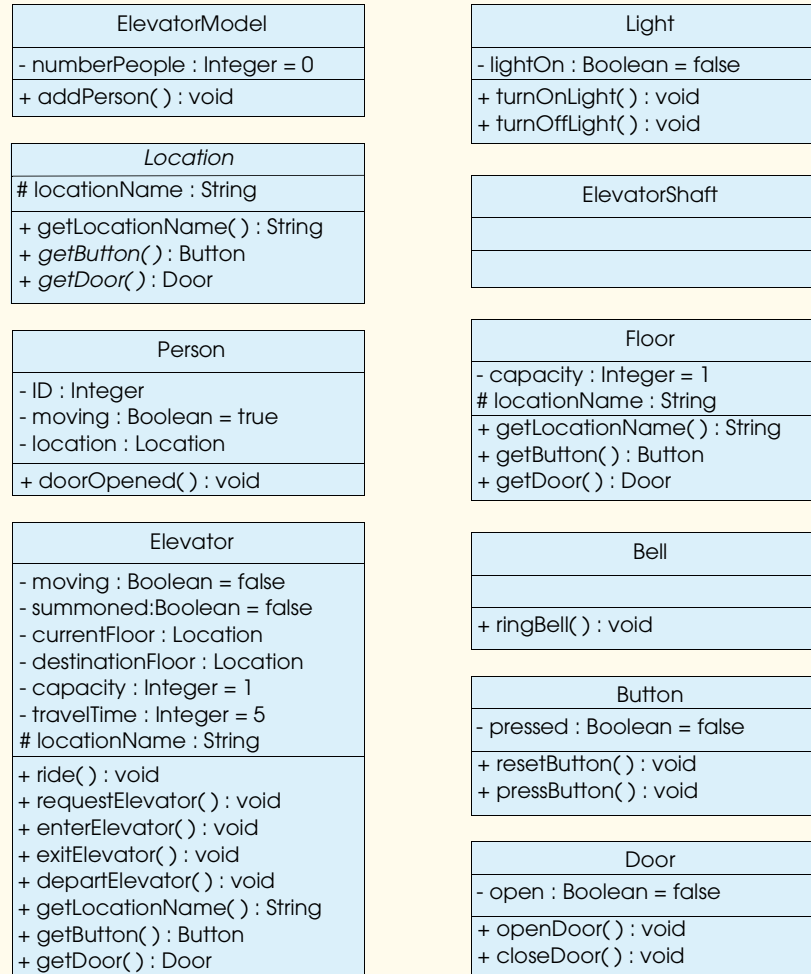


Fig. 9.39 Class diagram with attributes and operations (incorporating inheritance).

Class **Elevator** now contains two **Location** objects, called **currentFloor** and **destinationFloor**, replacing the integer values we used previously to describe the **Floors**. Lastly, **Person** contains a **Location** object named **location**, which indicates whether **Person** is on a **Floor** or in the **Elevator**.

Implementation: Forward Engineering (Incorporating Inheritance)

“Thinking About Objects” Section 8.17 used the UML to express the Java class structure for our simulation. We continue our implementation while incorporating inheritance, using class **Elevator** as an example. For each class in the class diagram of Fig. 9.38,

1. If a class **A** is a subclass of class **B**, then **A** extends **B** in the class declaration and calls the constructor of **B**. For example, class **Elevator** is a subclass of abstract superclass **Location**, so the class declaration should read

```
public class Elevator extends Location {

    public Elevator
    {
        super();
    }
    ...
}
```

2. If class **B** is an abstract class and class **A** is a subclass of class **B**, then class **A** must override the abstract methods of class **B** (if class **A** is to be a concrete class). For example, class **Location** contains abstract methods **getLocationName**, **getButton** and **getDoor**, so class **Elevator** must override these methods (note that **getButton** returns the **Elevator**'s **Button** object, and **getDoor** returns the **Elevator**'s **Door** object—**Elevator** contains associations with both objects, according to the class diagram of Fig. 9.38).

```
public class Elevator extends Location {

    // class attributes
    private boolean moving;
    private boolean summoned;
    private Location currentFloor;
    private Location destinationFloor;
    private int capacity = 1;
    private int travelTime = 5;

    // class objects
    private Button elevatorButton;
    private Door elevatorDoor;
    private Bell bell;

    // class constructor
    public Elevator()
    {
        super();
    }

    // class methods
    public void ride() {}
    public void requestElevator() {}
    public void enterElevator() {}
    public void exitElevator() {}
    public void departElevator() {}

    // method overriding getLocationName
    public String getLocationName()
    {
        return "elevator";
    }
}
```

```

// method overriding getButton
public Button getButton() {}
{
    return elevatorButton;
}

// method overriding getDoor
public Door getDoor() {}
{
    return elevatorDoor;
}
}

```

Using forward engineering provides a sound beginning for code implementation in any language. “Thinking About Objects” Section 11.10 returns to interactions and focuses on how objects generate and handle the messages passed in collaborations. In addition, we forward engineer more class diagrams to implement these interactions. Eventually, we present the Java code for our simulator in Appendix G, Appendix H and Appendix I.

9.24 (Optional) Discovering Design Patterns: Introducing Creational, Structural and Behavioral Design Patterns

Now that we have introduced object-oriented programming, we begin our deeper presentation of design patterns. In Section 1.17, we mentioned that the “gang of four” described 23 design patterns using three categories—creational, structural and behavioral. In this and the remaining “Discovering Design Patterns” sections, we discuss the design patterns of each type and their importance, and we mention how each pattern relates to the Java material in the book. For example, several Java Swing components that we introduce in Chapters 12 and 13 use the Composite design pattern, so we introduce the Composite design pattern in Section 13.18. Figure 9.40 identifies the 18 gang-of-four design patterns discussed in this book.

Figure 9.40 lists 18 of the most widely used patterns in the software-engineering industry. There are many popular patterns that have been documented since the gang-of-four book—these include the *concurrent design patterns*, which are especially helpful in the design of multithreaded systems. Section 15.13 discusses some of these patterns used in industry. Architectural patterns, as we discuss in Section 17.10, specify how subsystems interact with each other. Figure 9.41 specifies the concurrency patterns and architectural patterns that we discuss in this book.

Section	Creational design patterns	Structural design patterns	Behavioral design patterns
9.24	Singleton	Proxy	Memento State

Fig. 9.40 The 18 Gang-of-four design patterns discussed in *Java How to Program 4/e* (part 1 of 2).

Section	Creational design patterns	Structural design patterns	Behavioral design patterns
13.18	Factory Method	Adapter Bridge Composite	Chain-of-Responsibility Command Observer Strategy Template Method
17.10	Abstract Factory	Decorator Facade	
21.12	Prototype		Iterator

Fig. 9.40 The 18 Gang-of-four design patterns discussed in *Java How to Program 4/e* (part 2 of 2).

Section	Concurrent design patterns	Architectural patterns
15.13	Single-Threaded Execution Guarded Suspension Balking Read/Write Lock Two-Phase Termination	
17.10		Model-View-Controller Layers

Fig. 9.41 Concurrent design patterns and architectural patterns discussed in *Java How to Program, 4/e*.

9.24.1 Creational Design Patterns

Creational design patterns address issues related to the creation of objects, such as preventing a system from creating more than one object of a class (the Singleton creational design pattern) or deferring until execution time the decision as to what types of objects are going to be created (the purpose of the other creational design patterns discussed here). For example, suppose we are designing a 3-D drawing program, in which the user can create several 3-D geometric objects, such as cylinders, spheres, cubes, tetrahedrons, etc. At compile time, the program does not know what shapes the user will choose to draw. Based on user input, this program should be able to determine the class from which to instantiate an object. If the user creates a cylinder in the GUI, our program should “know” to instantiate an object of class **Cylinder**. When the user decides what geometric object to draw, the program should determine the specific subclass from which to instantiate that object.

The gang-of-four book describes five creational patterns (four of which we discuss in this book):

- Abstract Factory (Section 17.10)
- Builder (not discussed)

- Factory Method (Section 13.18)
- Prototype (Section 21.12)
- Singleton (Section 9.24)

Singleton

Occasionally, a system should contain exactly one object of a class—that is, once the program instantiates that object, the program should not be allowed to create additional objects of that class. For example, some systems connect to a database using only one object that manages database connections, which ensures that other objects cannot initialize unnecessary connections that would slow the system. The *Singleton design pattern* guarantees that a system instantiates a maximum of one object of a class.

Figure 9.42 demonstrates Java code using the Singleton design pattern. Line 5 declares class **Singleton** as **final**, so methods in this class cannot be overridden to handle multiple instantiations. Lines 11–14 declare a **private** constructor—only class **Singleton** can instantiate a **Singleton** object using this constructor. The **static** method **getSingletonInstance** (lines 17–24) allows the one-time instantiation of a **static Singleton** object (declared on line 8) by calling the **private** constructor. If the **Singleton** object has been created, line 23 merely returns a reference to the previously instantiated **Singleton** object.

Lines 10–11 of class **SingletonExample** (Fig. 9.43) declare two references to **Singleton** objects—**firstSingleton** and **secondSingleton**. Lines 14–15 call method **getSingletonInstance** and assign **Singleton** references to **firstSingleton** and **secondSingleton**, respectively. Line 18 tests if these objects reference the same **Singleton** object. Figure 9.44 shows that **firstSingleton** and **secondSingleton** share the same reference to the **Singleton** object, because each time method **getSingletonInstance** is called, it returns a reference to the same **Singleton** object.

```

1  // Singleton.java
2  // Demonstrates Singleton design pattern
3  package com.deitel.jhtp4.designpatterns;
4
5  public final class Singleton {
6
7      // Singleton object returned by method getSingletonInstance
8      private static Singleton singleton;
9
10     // constructor prevents instantiation from other objects
11     private Singleton()
12     {
13         System.err.println( "Singleton object created." );
14     }
15
16     // create Singleton and ensure only one Singleton instance
17     public static Singleton getSingletonInstance()
18     {

```

Fig. 9.42 Class **Singleton** ensures that only one object of its class is created (part 1 of 2).

```

19     // instantiate Singleton if null
20     if ( singleton == null )
21         singleton = new Singleton();
22
23     return singleton;
24 }
25 }

```

Fig. 9.42 Class **Singleton** ensures that only one object of its class is created (part 2 of 2).

```

1  // SingletonExample.java
2  // Attempt to create two Singleton objects
3  package com.deitel.jhtp4.designpatterns;
4
5  public class SingletonExample {
6
7      // run SingletonExample
8      public static void main( String args[] )
9      {
10         Singleton firstSingleton;
11         Singleton secondSingleton;
12
13         // create Singleton objects
14         firstSingleton = Singleton.getSingletonInstance();
15         secondSingleton = Singleton.getSingletonInstance();
16
17         // the "two" Singletons should refer to same Singleton
18         if ( firstSingleton == secondSingleton )
19             System.out.println( "firstSingleton and " +
20                                 "secondSingleton refer to the same Singleton " +
21                                 "object" );
22     }
23 }

```

Fig. 9.43 Class **SingletonExample** attempts to create **Singleton** object more than once.

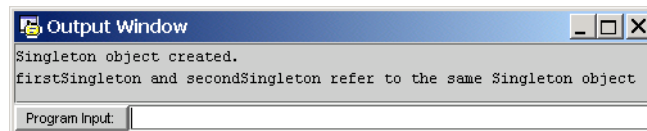


Fig. 9.44 Class **SingletonExample** output shows that the **Singleton** object may be created only once.

9.24.2 Structural Design Patterns

Structural design patterns describe common ways to organize classes and objects in a system. The gang-of-four book describes seven structural design patterns (six of which we discuss in this book):

- Adapter (Section 13.18)
- Bridge (Section 13.18)
- Composite (Section 13.18)
- Decorator (Section 17.10)
- Facade (Section 17.10)
- Flyweight (not discussed)
- Proxy (Section 9.24)

Proxy

An applet should always display something while images load. Whether that “something” is a smaller image or a string of text informing the user that the images are loading, the *Proxy design pattern* can be applied to achieve this effect. This pattern allows one object to act as a replacement for another. Consider loading several large images (several megabytes) in a Java applet. Ideally, we would like to see these images instantaneously—however, loading large images into memory can take time to complete (especially on a slow processor). The Proxy design pattern allows the system to use one object—called a *proxy object*—in place of another. In our example, the proxy object could be a gauge that informs the user of what percentage of a large image has been loaded. When this image finishes loading, the proxy object is no longer needed—the applet can then display an image instead of the proxy.

9.24.3 Behavioral Design Patterns

There are many different examples of *behavioral design patterns*, which provide proven strategies to model how objects collaborate with one another in a system and offer special behaviors appropriate for a wide variety of applications. Let us consider the Observer behavioral design pattern—a classic example of a design pattern illustrating collaborations between objects. For example, GUI components collaborate with their listeners to respond to user interactions. GUI components use this pattern to process user interface events. A listener observes state changes in a particular GUI component by registering to handle that GUI component’s events. When the user interacts with that GUI component, the component notifies its listeners (also known as its observers) that the GUI component’s state has changed (e.g., a button has been pressed).

Another pattern we consider is the Memento behavioral design pattern—an example of offering special behavior for a wide variety of applications. The Memento pattern enables a system to save an object’s state, so that state can be restored at a later time. For example, many applications provide an “undo” capability that allows users to revert to previous versions of their work.

The gang-of-four book describes 11 behavioral design patterns (eight of which we discuss in this book):

- Chain-of-Responsibility (Section 13.18)
- Command (Section 13.18)
- Interpreter (not discussed)
- Iterator (Section 21.12)

- Mediator (not discussed)
- Memento (Section 9.24)
- Observer (Section 13.18)
- State (Section 9.24)
- Strategy (Section 13.18)
- Template Method (Section 13.18)
- Visitor (not discussed)

Memento

Consider a painting program. This type of program allows a user to create graphics. Occasionally the user may position a graphic improperly in the drawing area. Painting programs offer an “undo” feature that allows the user to unwind such an error. Specifically, the program restores the drawing area’s state to that before the user placed the graphic. More sophisticated painting programs offer a *history*, which stores several states in a list, so the user can restore the program to any state in the history. The *Memento design pattern* allows an object to save its state, so that—if necessary—the object can be restored to its former state.

The Memento design pattern requires three types of objects. The *originator object* occupies some *state*—the set of attribute values at a specific time in program execution. In our painting-program example, the drawing area acts as the originator, because it contains attribute information describing its state—when the program first executes, the area contains no elements. The *memento object* stores a copy of all attributes associated with the originator’s state—i.e., the memento saves the drawing area’s state. The memento is stored as the first item in the history list, which acts as the *caretaker object*—the object that contains references to all memento objects associated with the originator.

Now, suppose the user draws a circle in the drawing area. The area contains different information describing its state—a circle object centered at specified *x-y* coordinates. The drawing area then uses another memento to store this information. This memento is stored as the second item in the history list. The history list displays all mementos on screen, so the user can select which state to restore. Suppose the user wishes to remove the circle—if the user selects the first memento from the list, the drawing area uses the first memento to restore itself to a blank area.

State

In certain designs, we must convey an object’s state information or represent the various states that an object can occupy. Our optional elevator simulation case study in the “Thinking About Objects” sections uses the *State design pattern*. Our simulation includes an elevator that moves between floors in a two-story building. A person walks across a floor and rides the elevator to the other floor. Originally, we used an integer value to represent on which floor the person is walking. However, we encountered a problem when we tried to answer the question “on what floor is the person when riding the elevator?” Actually, the person is located on neither floor—rather the person is located inside the elevator. We also realized that the elevator and the floors are locations that the person can occupy in our simulation. We created an abstract superclass called **Location** to represent a “location.” Subclasses **Elevator** and **Floor** inherit from superclass **Location**. Class **Person** contains a reference to a **Location** object, which represents the current location—elevator.

tor, first floor or second floor—of that person. Because a superclass reference can hold a subclass reference, the person's **Location** attribute references the appropriate **Floor** object when that person is on a floor and references the **Elevator** object when that person is inside the elevator.

The elevator and floors contain buttons. (The elevator's button signals the elevator to move to the other floor, and the floors' buttons summon the elevator to the floor of the request.) Because all locations in our simulation contain buttons, class **Location** provides abstract method **getButton**. Class **Elevator** implements method **getButton** to return a reference to the **Button** object inside the elevator, and class **Floor** implements method **getButton** to return a reference to the **Button** object on the floor. Using its **Location** reference, the person is able to press the correct button—i.e., the person will not press a floor's button when inside the elevator and will not press the elevator's button when on a floor.

The State design pattern uses an abstract superclass—called the *State class*—which contains methods that describe behaviors for states that an object (called the *context object*) can occupy. In our elevator simulation, the State class is superclass **Location**, and the context object is the object of class **Person**. Note that class **Location** does not describe all states of class **Person** (e.g., whether that person is walking or waiting for the elevator)—class **Location** describes only the location the **Person** and contains method **getButton** so the **Person** can access the **Button** object at various locations.

A *State subclass*, which extends the State class, represents an individual state that the context can occupy. The State subclasses in our simulation are classes **Elevator** and **Floor**. Each State subclass contains methods that implement the State class' abstract methods. For example, both classes **Elevator** and **Floor** implement method **getButton**.

The context contains exactly one reference to an object of the State class—this reference is called the *state object*. In the simulation, the state object is the object of class **Location**. When the context changes state, the state object references the State subclass object associated with that new state. For example, when the person walks from the floor into the elevator, the **Person** object's **Location** is changed from referencing one of the **Floor** objects to referencing the **Elevator** object. When the person walks onto the floor from the elevator, the **Person** object's **Location** references the appropriate **Floor** object.

9.24.4 Conclusion

In “Discovering Design Patterns” Section 9.24, we listed the three types of design patterns introduced in the gang-of-four book, we identified 18 of these design patterns that we discuss in this book and we discussed specific design patterns, including Singleton, Proxy, Memento and State. In “Discovering Design Patterns” Section 13.18, we introduce some design patterns associated with AWT and Swing GUI components. After reading this section, you should understand better how Java GUI components take advantage of design patterns.

9.24.5 Internet and World-Wide-Web Resources

The following URLs provide further information on the nature, importance and applications of design patterns.

*Design Patterns***www.hillside.net/patterns**

This page displays links to information on design patterns and languages.

www.hillside.net/patterns/books/

This site lists books on design patterns.

www.netobjectives.com/design.htm

This site introduces the importance of design patterns.

umbc7.umbc.edu/~tarr/dp/dp.html

This site links to design patterns Web sites, tutorials and papers.

www.links2go.com/topic/Design_Patterns

This site links to sites and information on design patterns.

www.c2.com/ppr/

This site discusses recent advances in design patterns and ideas for future projects.

*Design Patterns in Java***www.research.umbc.edu/~tarr/cs491/fall100/cs491.html**

This site is for a Java design patterns course at the University of Maryland and contains numerous examples of how to apply design patterns in Java.

www.interact.com/~bradapp/javapats.html

This site discusses Java design patterns and presents design patterns in distributed computing.

www.meurrens.org/ip-Links/java/designPatterns/

This site displays numerous links to resources and information on Java design patterns.

*Design Patterns in C++ & Visual Basic***journal.iftech.com/articles/9904_shankel_patterns/**

This site provides insight to design patterns (the Iterator design pattern, in particular) in C++.

mspress.microsoft.com/prod/books/sampchap/2322.htmThis site overviews the book, *Microsoft Visual Basic Design Patterns* (Microsoft Press: 2000).*Architectural Patterns***compsci.about.com/science/compsci/library/weekly/aa030600a.htm**

This site provides an overview the Model-View-Controller architecture.

www.javaworld.com/javaworld/jw-04-1998/jw-04-howto.html

This site contains an article discussing how Swing components use Model-View-Controller architecture.

www.ootips.org/mvc-pattern.html

This site provides information and tips on using MVC.

www.ftech.co.uk/~honeyg/articles/pda.htm

This site contains an article on the importance of architectural patterns in software.

www.tml.hut.fi/Opinnot/Tik-109.450/1998/niska/sld001.htm

This site provides information on architectural patterns, design pattern, and idioms (patterns targeting a specific language).

SUMMARY

- One of the keys to the power of object-oriented programming is achieving software reusability through inheritance.
- Through inheritance, a new class inherits the instance variables and methods of a previously defined superclass. In this case, the new class is referred to as a subclass.
- With single inheritance, a class is derived from one superclass. With multiple inheritance, a subclass inherits from multiple superclasses. Java does not support multiple inheritance, but Java does provide the notion of interfaces, which offer many of the benefits of multiple inheritance.
- A subclass normally adds instance variables and methods of its own, so a subclass generally is larger than its superclass. A subclass represents a smaller set of more specific objects than its superclass.
- A subclass cannot access the **private** members of its superclass. A subclass can, however, access the **public**, **protected** and package access members of its superclass. The subclass must be in the superclass's package to use superclass members with package access.
- A subclass constructor always calls the constructor for its superclass first (either explicitly or implicitly) to create and initialize the subclass members inherited from the superclass.
- Inheritance enables software reusability, which saves time in development and encourages the use of previously proven and debugged high-quality software.
- An object of a subclass can be treated as an object of its corresponding superclass, but the reverse is not true.
- A superclass exists in a hierarchical relationship with its subclasses.
- When a class is used with the mechanism of inheritance, it becomes either a superclass that supplies attributes and behaviors to other classes or a subclass that inherits those attributes and behaviors.
- An inheritance hierarchy can be arbitrarily deep within the physical limitations of a particular system, but most inheritance hierarchies have only a few levels.
- Hierarchies are useful for understanding and managing complexity. With software becoming increasingly complex, Java provides mechanisms for supporting hierarchical structures through inheritance and polymorphism.
- Modifier **protected** serves as an intermediate level of protection between **public** access and **private** access. Superclass **protected** members may be accessed by methods of the superclass, by methods of subclasses and by methods of classes in the same package.
- A superclass may be either a direct superclass or an indirect superclass. A direct superclass is the class that a subclass explicitly **extends**. An indirect superclass is inherited from several levels up the class hierarchy tree.
- When a superclass member is inappropriate for a subclass, the programmer must override that member in the subclass.
- In a "has a" relationship, a class object has a reference to an object of another class as a member. In an "is a" relationship, an object of a subclass type may also be treated as an object of the superclass type. "Is a" is inheritance. "Has a" is composition.
- A reference to a subclass object may be converted implicitly to a reference for a superclass object.
- It is possible to convert a superclass reference to a subclass reference by using an explicit cast. If the target object is not a subclass object, a **ClassCastException** is thrown.
- A superclass specifies commonality. All classes derived from a superclass inherit the capabilities of that superclass. In the object-oriented design process, the designer looks for commonality among classes and factors it out to form superclasses. Then, subclasses are customized beyond the capabilities inherited from the superclass.

- Reading a set of subclass declarations can be confusing, because inherited superclass members are not listed in the subclass declarations, but these members are indeed present in the subclasses.
- With polymorphism, it becomes possible to design and implement systems that are more extensible. Programs can be written to process objects of types that may not exist when the program is under development.
- Polymorphic programming can eliminate the need for **switch** logic, thus avoiding the kinds of errors associated with **switch** logic.
- An abstract method is declared by preceding the method's definition with the keyword **abstract** in the superclass.
- There are many situations in which it is useful to define classes for which the programmer never intends to instantiate any objects. Such classes are called **abstract** classes. Because these are used only as superclasses, typically they are called **abstract** superclasses. A program cannot instantiate objects of an **abstract** class.
- Classes from which a program can instantiate objects are called concrete classes.
- A class is made abstract by declaring it with the keyword **abstract**.
- If a subclass is derived from a superclass with an **abstract** method without supplying a definition for that **abstract** method in the subclass, that method remains **abstract** in the subclass. Consequently, the subclass is also an **abstract** class.
- When a request is made through a superclass reference to use a method, Java chooses the correct overridden method in the subclass associated with the object.
- Through the use of polymorphism, one method call can cause different actions to occur, depending on the type of the object receiving the call.
- Although we cannot instantiate objects of **abstract** superclasses, we can declare references to **abstract** superclasses. Such references can then be used to enable polymorphic manipulations of subclass objects when such objects are instantiated from concrete classes.
- New classes are regularly added to systems. New classes are accommodated by dynamic method binding (also called late binding). The type of an object need not be known at compile time for a method call to be compiled. At execution time, the appropriate method of the receiving object is selected.
- With dynamic method binding, at execution time the call to a method is routed to the method version appropriate for the class of the object receiving the call.
- When a superclass provides a method, subclasses can override the method, but they do not have to override it. Thus a subclass can use a superclass's version of a method.
- An interface definition begins with the keyword **interface** and contains a set of **public abstract** methods. Interfaces may also contain **public static final** data.
- To use an interface, a class must specify that it **implements** the interface and that class must define every method in the interface with the number of arguments and the return type specified in the interface definition.
- An interface is typically used in place of an abstract class when there is no default implementation to inherit.
- When a class implements an interface, the same "is a" relationship provided by inheritance applies.
- To implement more than one interface, simply provide a comma-separated list of interface names after keyword **implements** in the class definition.
- Inner classes are defined inside the scope of other classes.

- An inner class can also be defined inside a method of a class. Such an inner class has access to its outer class's members and to the **final** local variables for the method in which it is defined.
- Inner class definitions are used mainly in event handling.
- Class **JFrame** provides the basic attributes and behaviors of a window—a title bar and buttons to minimize, maximize and close the window.
- An inner class object has access to all the variables and methods of the outer class object.
- Because an anonymous inner class has no name, one object of the anonymous inner class must be created at the point where the class is defined in the program.
- An anonymous inner class can implement an interface or extend a class.
- The event generated when the user clicks the window's close box is a window closing event.
- Method **addWindowListener** registers a window event listener. The argument to **addWindowListener** must be a reference to an object that is a **WindowListener** (package **java.awt.event**).
- For event handling interfaces with more than one method, Java provides a corresponding class (called an adapter class) that already implements all the methods in the interface for you. Class **WindowAdapter** implements interface **WindowListener**, so every **WindowAdapter** object is a **WindowListener**.
- Compiling a class that contains inner classes results in a separate **.class** file for every class.
- Inner classes with class names can be defined as **public**, **protected**, package access or **private** and are subject to the same usage restrictions as other members of a class.
- To access the outer class's **this** reference, use *OuterClassName.this*.
- The outer class is responsible for creating objects of its non**static** inner classes.
- An inner class can be declared **static**.

TERMINOLOGY

abstract class	hierarchical relationship
abstract method	implementation inheritance
abstract superclass	implicit reference conversion
abstraction	indirect superclass
anonymous inner class	infinite recursion error
base class	inheritance
Boolean class	inheritance hierarchy
Character class	inner class
class hierarchy	Integer class
client of a class	interface
composition	interface inheritance
direct superclass	"is a" relationship
Double class	JFrame class
dynamic method binding	late binding
extends	Long class
extensibility	member access control
final class	member object
final instance variable	method overriding
final method	multiple inheritance
garbage collection	Number class
"has a" relationship	Object class

object-oriented programming (OOP)
 override a method
 override an **abstract** method
 overriding vs. overloading
 polymorphism
protected member of a class
 reference to an **abstract** class
setSize method
setVisible method
 single inheritance
 software reusability
 standardized software components
 subclass
 subclass constructor

subclass reference
super
 superclass
 superclass constructor
 superclass reference
switch logic
this
 type-wrapper class
 “uses a” relationship
WindowAdapter class
windowClosing method
WindowEvent class
WindowListener interface

SELF-REVIEW EXERCISES

9.1 Fill in the blanks in each of the following statements:

- If the class **Alpha** inherits from the class **Beta**, class **Alpha** is called the _____ class and class **Beta** is called the _____ class.
- Inheritance enables _____, which saves time in development and encourages using previously proven and high-quality software components.
- An object of a _____ class can be treated as an object of its corresponding _____ class.
- The four member access specifiers are _____, _____, _____ and _____.
- A “has a” relationship between classes represents _____ and an “is a” relationship between classes represents _____.
- Using polymorphism helps eliminate _____ logic.
- If a class contains one or more **abstract** methods, it is an _____ class.
- A method call resolved at run-time is referred to as _____ binding.
- A subclass may call any non**private** superclass method by prepending _____ to the method call.

9.2 State whether each of the following statements is *true* or *false*. If *false*, explain why.

- A superclass typically represents a larger number of objects than its subclass represents (true/false).
- A subclass typically encapsulates less functionality than does its superclass. (true/false).

ANSWERS TO SELF-REVIEW EXERCISES

9.1 a) sub, super. b) software reusability. c) sub, super. d) **public**, **protected**, **private** and package access. e) composition, inheritance. f) **switch**. g) **abstract**. h) dynamic. i) **super**.

- 9.2 a) True.
 b) False. A subclass includes all the functionality of its superclass.

EXERCISES

9.3 Consider the class **Bicycle**. Given your knowledge of some common components of bicycles, show a class hierarchy in which the class **Bicycle** inherits from other classes, which, in turn, inherit from yet other classes. Discuss the instantiation of various objects of class **Bicycle**. Discuss inheritance from class **Bicycle** for other closely related subclasses.

- 9.4** Define each of the following terms: single inheritance, multiple inheritance, interface, superclass and subclass.
- 9.5** Discuss why casting a superclass reference to a subclass reference is potentially dangerous.
- 9.6** Distinguish between single inheritance and multiple inheritance. What feature of Java helps realize the benefits of multiple inheritance?
- 9.7** (*True/False*) A subclass is generally smaller than its superclass.
- 9.8** (*True/False*) A subclass object is also an object of that subclass's superclass.
- 9.9** Some programmers prefer not to use **protected** access because it breaks information hiding in the superclass. Discuss the relative merits of using **protected** access vs. **private** access in superclasses.
- 9.10** Many programs written with inheritance could be solved with composition instead, and vice versa. Discuss the relative merits of these approaches in the context of the **Point**, **Circle**, **Cylinder** class hierarchy in this chapter. Rewrite the program of Fig. 9.22–Fig. 9.26 (and the supporting classes) to use composition rather than inheritance. After you do this, reassess the relative merits of the two approaches both for the **Point**, **Circle**, **Cylinder** problem and for object-oriented programs in general.
- 9.11** Rewrite the **Point**, **Circle**, **Cylinder** program of Fig. 9.22–Fig. 9.26 as a **Point**, **Square**, **Cube** program. Do this two ways—once with inheritance and once with composition.
- 9.12** In the chapter, we stated, “When a superclass method is inappropriate for a subclass, that method can be overridden in the subclass with an appropriate implementation.” If this is done, does the subclass-is-a-superclass-object relationship still hold? Explain your answer.
- 9.13** Study the inheritance hierarchy of Fig. 9.2. For each class, indicate some common attributes and behaviors consistent with the hierarchy. Add some other classes (e.g., **UndergraduateStudent**, **GraduateStudent**, **Freshman**, **Sophomore**, **Junior**, **Senior**), to enrich the hierarchy.
- 9.14** Write an inheritance hierarchy for classes **Quadrilateral**, **Trapezoid**, **Parallelogram**, **Rectangle** and **Square**. Use **Quadrilateral** as the superclass of the hierarchy. Make the hierarchy as deep (i.e., as many levels) as possible. The **private** data of **Quadrilateral** should include the (x, y) coordinate pairs for the four endpoints of the **Quadrilateral**. Write a driver program that instantiates and displays objects of each of these classes. [In Chapter 11, “Graphics and Java2D,” you will learn how to use Java’s drawing capabilities.]
- 9.15** Write down all the shapes you can think of—both two-dimensional and three-dimensional—and form those shapes into a shape hierarchy. Your hierarchy should have superclass **Shape**, from which class **TwoDimensionalShape** and class **ThreeDimensionalShape** are derived. Once you have developed the hierarchy, define each of the classes in the hierarchy. We will use this hierarchy in the exercises to process all shapes as objects of superclass **Shape**.
- 9.16** How is it that polymorphism enables you to program “in the general” rather than “in the specific”? Discuss the key advantages of programming “in the general.”
- 9.17** Discuss the problems of programming with **switch** logic. Explain why polymorphism is an effective alternative to using **switch** logic.
- 9.18** Distinguish between inheriting interface and inheriting implementation. How do inheritance hierarchies designed for inheriting interface differ from those designed for inheriting implementation?
- 9.19** Distinguish between non**abstract** methods and **abstract** methods.
- 9.20** (*True/False*) All methods in an **abstract** superclass must be declared **abstract**.
- 9.21** Suggest one or more levels of **abstract** superclasses for the **Shape** hierarchy discussed in the beginning of this chapter (the first level is **Shape** and the second level consists of the classes **TwoDimensionalShape** and **ThreeDimensionalShape**).

9.22 How does polymorphism promote extensibility?

9.23 You have been asked to develop a flight simulator that will have elaborate graphical outputs. Explain why polymorphic programming would be especially effective for a problem of this nature.

9.24 Develop a basic graphics package. Use the **Shape** class inheritance hierarchy from Figure 9.3. Limit yourself to two-dimensional shapes such as squares, rectangles, triangles and circles. Interact with the user. Let the user specify the position, size, shape and fill colors to be used in drawing each shape. The user can specify many items of the same shape. As you create each shape, place a **Shape** reference to each new **Shape** object into an array. Each class has its own **draw** method. Write a polymorphic screen manager that walks through the array sending **draw** messages to each object in the array to form a screen image. Redraw the screen image each time the user specifies an additional shape. Investigate the methods of class **Graphics** to help draw each shape.

9.25 Modify the payroll system of Fig. 9.16–Fig. 9.21 to add **private** instance variables **birthDate** (use class **Date** from Figure 8.13) and **departmentCode** (an **int**) to class **Employee**. Assume this payroll is processed once per month. Then, as your program calculates the payroll for each **Employee** (polymorphically), add a \$100.00 bonus to the person's payroll amount if this is the month in which the **Employee**'s birthday occurs.

9.26 In Exercise 9.15, you developed a **Shape** class hierarchy and defined the classes in the hierarchy. Modify the hierarchy so that class **Shape** is an **abstract** superclass containing the interface to the hierarchy. Derive **TwoDimensionalShape** and **ThreeDimensionalShape** from class **Shape**—these classes should also be **abstract**. Use an **abstract print** method to output the type and dimensions of each class. Also include **area** and **volume** methods so these calculations can be performed for objects of each concrete class in the hierarchy. Write a driver program that tests the **Shape** class hierarchy.

9.27 Rewrite your solution to Exercise 9.26 to use a **Shape** interface instead of an abstract **Shape** class.

9.28 (*Drawing Application*) Modify the drawing program of Exercise 8.19 to create a drawing application that draws random lines, rectangles and ovals. [Note: Like an applet, a **JFrame** has a **paint** method that you can override to draw on the background of the **JFrame**.]

For this exercise, modify the **MyLine**, **MyOval** and **MyRect** classes of Exercise 8.19 to create the class hierarchy in Fig. 9.45. The classes of the **MyShape** hierarchy should be “smart” shape classes where objects of these classes know how to draw themselves (if provided with a **Graphics** object that tells them where to draw). The only **switch** or **if/else** logic in this program should be to determine the type of shape object to create (use random numbers to pick the shape type and the coordinates of each shape). Once an object from this hierarchy is created, it will be manipulated for the rest of its lifetime as a superclass **MyShape** reference.

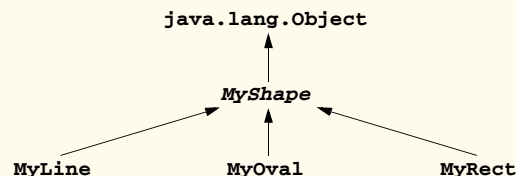


Fig. 9.45 The **MyShape** hierarchy.

Class **MyShape** in Fig. 9.45 *must* be **abstract**. The only data representing the coordinates of the shapes in the hierarchy should be defined in class **MyShape**. Lines, rectangles and ovals can all be drawn if you know two points in space. Lines require *x1*, *y1*, *x2* and *y2* coordinates. The **drawLine** method of the **Graphics** class will connect the two points supplied with a line. If you have the same four coordinate values (*x1*, *y1*, *x2* and *y2*) for ovals and rectangles, you can calculate the four arguments needed to draw them. Each requires an upper-left *x*-coordinate value (minimum of the two *x*-coordinate values), an upper-left *y*-coordinate value (minimum of the two *y* coordinate values), a *width* (difference between the two *x*-coordinate values; must be nonnegative) and a *height* (difference between the two *y*-coordinate values; must be nonnegative). [Note: In Chapter 12, each *x,y* pair will be captured by using mouse events from mouse interactions between the user and the program's background. These coordinates will be stored in an appropriate shape object as selected by the user. As you begin the exercise, you will use random coordinate values as arguments to the constructor.]

In addition to the data for the hierarchy, class **MyShape** should define at least the following methods:

- a) A constructor with no arguments that sets the coordinates to 0.
- b) A constructor with arguments that sets the coordinates to the supplied values.
- c) Set methods for each individual piece of data that allow the programmer to independently set any piece of data for a shape in the hierarchy (e.g., if you have an instance variable **x1**, you should have a method **setX1**).
- d) Get methods for each individual piece of data that allow the programmer to independently retrieve any piece of data for a shape in the hierarchy (e.g., if you have an instance variable **x1**, you should have a method **getX1**).
- e) The **abstract** method

```
public abstract void draw( Graphics g );
```

This method will be called from the program's **paint** method to draw a shape onto the screen.

The preceding methods are required. If you would like to provide more methods for flexibility, please do so. However, be sure that any method you define in this class is a method that would be used by *all* shapes in the hierarchy.

All data *must* be **private** to class **MyShape** in this exercise (this forces you to use proper encapsulation of the data and provide proper *set/get* methods to manipulate the data). You are not allowed to define new data that can be derived from existing information. As explained previously, the upper-left *x*, upper-left *y*, *width* and *height* needed to draw an oval or rectangle can be calculated if you already know two points in space. All subclasses of **MyShape** should provide two constructors that mimic those provided by class **MyShape**.

Objects of the **MyOval** and **MyRect** classes should not calculate their upper-left *x*-coordinate, upper-left *y*-coordinate, *width* and *height* until they are about to draw. Never modify the *x1*, *y1*, *x2* and *y2* coordinates of a **MyOval** or **MyRect** object to prepare to draw them. Instead, use the temporary results of the calculations described above. This will help us enhance the program in Chapter 12 by allowing the user to select each shape's coordinates with the mouse.

There should be no **MyLine**, **MyOval** or **MyRect** references in the program—only **MyShape** references that refer to **MyLine**, **MyOval** and **MyRect** objects are allowed. The program should keep an array of **MyShape** references containing all shapes. The program's **paint** method should walk through the array of **MyShape** references and draw every shape (i.e., call every shape's **draw** method).

Begin by defining class **MyShape**, class **MyLine** and an application to test your classes. The application should have a **MyShape** instance variable that can refer to one **MyLine** object (created

in the application's constructor). The **paint** method (for your subclass of **JFrame**) should draw the shape with a statement like

```
currentShape.draw( g );
```

where **currentShape** is the **MyShape** reference and **g** is the **Graphics** object that the shape will use to draw itself on the background of the window.

9.29 Next, change the single **MyShape** reference into an array of **MyShape** references and hard code several **MyLine** objects into the program for drawing. The application's **paint** method should walk through the array of shapes and draw every shape.

After the preceding part is working, you should define the **MyOval** and **MyRect** classes and add objects of these classes into the existing array. For now, all the shape objects should be created in the constructor for your subclass of **JFrame**. In Chapter 12, we will create the objects when the user chooses a shape and begins drawing it with the mouse.

In Exercise 9.28, you defined a **MyShape** hierarchy in which classes **MyLine**, **MyOval** and **MyRect** subclass **MyShape** directly. If the hierarchy was properly designed, you should be able to see the tremendous similarities between the **MyOval** and **MyRect** classes. Redesign and reimplement the code for the **MyOval** and **MyRect** classes to “factor out” the common features into the **abstract** class **MyBoundedShape** to produce the hierarchy in Fig. 9.46.

Class **MyBoundedShape** should define two constructors that mimic the constructors of class **MyShape** and methods that calculate the upper-left *x*-coordinate, upper-left *y*-coordinate, *width* and *height*. No new data pertaining to the dimensions of the shapes should be defined in this class. Remember, the values needed to draw an oval or rectangle can be calculated from two (*x*,*y*) coordinates. If designed properly, the new **MyOval** and **MyRect** classes should each have two constructors and a **draw** method.

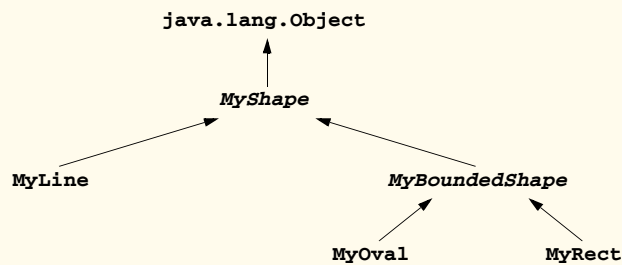


Fig. 9.46 The **MyShape** hierarchy.