[CHAPTER] 3

OBJECT-ORIENTED Programming Using Java

3.1 Introduction

After reading Chapters 1 and 2, it should be clear that software development includes much more than writing code. In Chapter 1, you learned that the first step in the software life cycle is to describe the proposed software's behavior and to document that behavior formally in the requirements and specifications documents. In Chapter 2, you learned how to use object-oriented techniques to develop a design that, when implemented, satisfies the conditions spelled out in the requirements and specifications documents. This chapter focuses on aspects of Java that enable you to write a correct and efficient object-oriented program. Although this chapter focuses on implementation, keep in mind that coding is only one step in the software life cycle. A considerable amount of work must be done before and after the coding phase.

While reading this chapter, note two important points. First, the chapter only discusses aspects of Java that are necessary to write object-oriented programs. Java provides many additional features that are useful in developing software, but they are discussed later in the text. Second, we assume that you have already learned to program in a high-level language and therefore understand basic programming concepts such as types, variables, parameter passing, and control structures. However, if you are not fully comfortable with these concepts, consider reviewing an introductory Java programming text before reading this chapter.

As we mentioned in Chapter 2, an object-oriented program consists of a collection of objects that interact to accomplish a specified task. Because objects are instantiated from classes, the primary activity of an object-oriented programmer is to write class definitions that can be used to instantiate the objects in a program. Therefore, given the importance of class definitions, this chapter begins by describing how to write classes and use inheritance, followed by a discussion of the nuts and bolts of running a Java program.

JAMES GOSLING

James Gosling is the "father of Java" and the designer of the Java Virtual Machine (JVM). He was born in Calgary, Alberta, and received his PhD in computer science in 1983 from Carnegie Mellon. He was a prolific software developer; in addition to his work with Java, he created the original Emacs text editor, a satellite data acquisition system, a multiprocessor version of UNIX, and a number of compilers, windowing systems, and mail programs. He joined Sun Microsystems in 1984.

Gosling began working on Java (then known as Oak) with his colleagues at Sun in the early 1990s. Their goal was to create a programming language that could run on any platform—a style known as WORA (Write Once, Run Anywhere). Such a language would be

continued

ideal for use on the World Wide Web, where pages are downloaded and executed on a client's computer. Gosling based the design of this new language on C/C++, which was then quite popular, but he brought greater simplicity, elegance, and consistency to its overall structure. Gosling also designed the bytecode into which Java programs are translated and the Java virtual machine (JVM) that would execute his bytecode.

Java 1.0 was released in 1995, and it was soon incorporated into all major Web browsers. The language achieved rapid popularity, and in just a few years became one of the most popular and widely used languages in the academic and commercial sectors. According to the Tiobe Programming Community Index (*www.tiobe.com/tpci.htm*), a Web site that ranks the popularity of programming languages, Java has been the most widely used language around the world since June 2001.

For his work in the design of Java and JVM, Gosling was elected to membership in the National Academy of Engineering. He is currently a vice president and fellow at Sun Microsystems in Santa Clara, California, as well as chief technical officer of Sun's Developer Products Division.

3.2 Class Definitions in Java

In Java, a class declaration defines a new class from which objects can be instantiated. The class declaration defines the state and behavior that instances of the class will exhibit. Instance variables define the state of an object, and methods define the object's behavior. A Java program is a collection of one or more class declarations.

Classes in Java are organized into packages. A **package** is a collection of related classes. Every class definition in Java, whether you write it yourself or get it from a class library, is a member of a package. If a class does not explicitly name the package to which it belongs, then it automatically becomes a member of the "unnamed" package. All Java systems must support at least one unnamed package. Understanding the concept of package membership will become important later when we discuss accessibility.

A Java class can be declared as either a top-level class or as one of four different types of nested classes. As the name implies, a **nested class** is defined inside another class. Nested classes are often referred to as **inner classes** in the Java literature. The general form of a top-level class declaration is shown in Figure 3-1.[1]

[1] See the Java tutorial (*http://java.sun.com/docs/books/tutorial/*) for an in-depth discussion of inner classes and how to use them in a Java program.

```
ClassModifier class identifier {
      Class State...
      Class Behavior...
}
```

[FIGURE 3-1] General form of a top-level class declaration

The *identifier* specifies the name of the class. In Java, a name consists of an arbitrary number of letters (including _ and $) and digits. The first character of an identifier must be a letter. Identifiers cannot have the same spelling as any of the keywords or literals used in the Java language. By convention, the first letter of an identifier that specifies a class name is capitalized, and the first letter of an identifier that specifies a variable is not. If a name contains multiple words, the first letter of each word is capitalized, with the exception of the first word if the name refers to a variable. For example, the names `counter`, `xCoordinate`, and `numberOfSides` are all valid variable names. The names `Rectangle`, `Thermostat`, and `GasFurnace` are examples of class names. This naming style is a coding convention, and not a requirement of the Java language. Programmers use coding conventions to make code easier to read.

Class declarations are placed in source files; each Java source file contains a single public class. The name of the file and the public class it contains must be the same. By convention, files that contain Java source code use the suffix `.java`. For example, Figure 3-2 contains the skeletal outline of a class that defines a point. Because the name of this class is `Point`, the file that contains this class definition must be named `Point.java`.

```
/**
 * A class that defines a point
 */
public class Point {
    /* This class defines points in 2-dimensional space */
    // State of a point
    // Behavior of a point
} // Point
```

[FIGURE 3-2] Skeletal outline of a `Point` class

In a top-level class, the *ClassModifier* is used to determine what classes are allowed to access the class declaration. The *ClassModifier* is optional; if it is omitted, the class is defined as having **package scope**, in which only classes that are members of the same package can access the declaration and create instances of the class. If a top-level class is defined as public, any class, regardless of the package in which it is defined, can access the declaration and use it to create instances of the class. Most of the classes that you will write have public access.

As illustrated in Figure 3-2, Java permits programmers to use two types of comments to document their code. Multiline comments begin with the characters /* and end with */. Anything between the starting /* and the terminating */ is ignored by the compiler. Single-line comments start with the characters // and terminate at the end of the line. Everything on the line following the // is ignored by the compiler. Comments do not nest, which means that the character sequences /* and // have no special meaning inside a comment.

Some multiline comments in Figure 3-2 start with /** instead of /*. The extra asterisk is a flag that indicates the comment contains information that should be processed by the Javadoc program. These are often referred to as **Javadoc comments**. In Chapter 1, you learned that Javadoc is a program included with the standard Java distribution; you can use it to generate HTML-based documentation. The Javadoc program processes the Javadoc comments, then incorporates the information into the documentation generated by the program. Table 3-1 lists the Javadoc tags that appear in this chapter. Many other tags can be used to create an extensive set of online documentation for a Java program.

TAG	DESCRIPTION
`@author`	Identifies the author(s) of the code: `@author Paul Tymann...`
`@version`	Specifies the version number of the source code: `@version  Version 1.3, developed 5/1/2007`
`@param`	Provides information about method and constructor parameters. The tag is followed by a parameter name and a comment: `@param count  Number of elements in the list`
`@return`	Description of the return value for nonvoid methods. The body of the class definition contains declarations that define the state and behavior associated with the class. The next section examines how state information is declared in a Java class definition. `@return  the sum of all nonzero values`

[TABLE 3-1] Javadoc tags

3.2.1 State

The variables that define the state associated with instances of a class are referred to as **instance variables**. The format for the declaration of an instance variable is shown in Figure 3-3.

```
Modifiers  Type   VariableName;
```

[FIGURE 3-3] General form of an instance variable declaration

In Figure 3-4, two instance variables, `xCoordinate` and `yCoordinate`, have been added to the `Point` class. The variables are used to store the point's x- and y-coordinates.

```
/**
 * A class that defines a point
 */
public class Point  {
    // State of a point
    private int xCoordinate; // The x-coordinate
    private int yCoordinate; // The y-coordinate

    // Behavior of a point
} // Point
```

[FIGURE 3-4] Adding state to the `Point` class

Every time a new instance of a class is created, a new set of instance variables is created for that object. The instance variables are permanently associated with the object in which they are declared and represent the state of that object. As long as the object exists, its instance variables exist, which means it has state. If you assign a value to an instance variable, that value remains stored in the instance variable until you either change it or the object is destroyed. This is considerably different from the way in which **local variables** are created and destroyed. Local variables are automatically created when the method in which they are defined is invoked, and they are destroyed when the method returns. As a consequence, consider what happens if you assign a value to both an instance variable and a local variable during a method invocation. The next time the method is invoked, the instance variable still contains the value that was assigned during the previous invocation, whereas the local variable does not. Local variables are created "fresh" every time the method is invoked, but instance variables are created only once when the object is instantiated.

Like a class definition, every instance variable has an associated accessibility that determines its visibility. Instance variables with public accessibility can be accessed by instances of any class. You should avoid using public instance variables because they break the encapsulation provided by the class structure.

Private instance variables can be accessed by any instance of the same class. The scope of a private variable is not restricted to a single instance of the class. Any instance of the same class can access the private instance variables of other instances of the same class. In the `Point` class of Figure 3-4, this means that any `Point` object can access the `xCoordinate` and `yCoordinate` of any other `Point` object.

In Java, the modifiers `public` and `private` specify the access of an instance variable. The modifiers appear in the declaration of the instance variable before the keyword that specifies the type of the variable. Only one access modifier can appear in the declaration of an instance variable. In Figure 3-4, the instance variables `xCoordinate` and `yCoordinate` have private access. Package access is the default for an instance variable, so if you do not provide an access modifier for an instance variable, it has package scope. The class definition in Figure 3-5 shows how to declare instance variables with private, public, and package access.

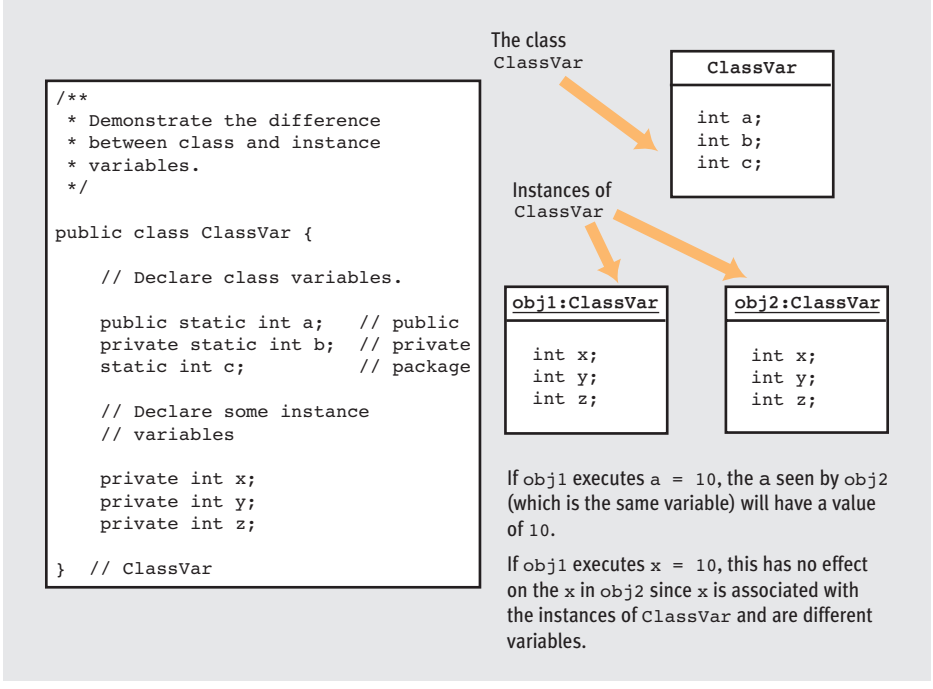
```
/**
 * Illustrate how to specify the scope of instance variables
 */
public class InstanceScope {
      // x is private, which means x can only be accessed by
      // instances of the InstanceScope class
      private int x;

      // y is public, which means y can be accessed by any
      // instance of any class
      public int y;

      // z has package scope, which means z can be accessed
      // by any instance of a class that is in the same
      // package as InstanceScope
      int z;

} // InstanceScope
```

[FIGURE 3-5] Specifying the access of instance variables

The state of an object is normally private and accessed using methods that are provided by the class. Finally, any class in the same package can access instance variables with package access. Like public instance variables, you should avoid instance variables with package access.

Instance variables are associated with an instance of a class. You can associate a state variable with a Java class by declaring the variable as `static`. **Static variables** represent state that is associated with the entire class, not with individual instances of the class. This means that exactly one copy of the static variable exists, regardless of whether zero, six, or 1000 instances of the class have been created. Because these variables are associated with the class and not with instances of the class, they are referred to as **class variables**. You can almost think of a class variable as a global variable that is associated with a class. Whether instances of other classes can access a class variable depends on its associated access modifier.

You specify the accessibility of a class variable in the same way as that of an instance variable. The keyword `static` in the declaration of a state variable specifies that it is a class variable. If a modifier specifies the access of the class variable, it should precede the static modifier. The class definition in Figure 3-6 illustrates how you can declare class variables.



[FIGURE 3-6] Class and instance variables

You can initialize both instance and class variables when they are declared by adding an initialization expression to the variable declaration. The initialization expression takes the form of an assignment statement, in which the variable declaration appears on the left side of the assignment operator and an expression appears on the right side:

```
type variable-name = initialization-expression;
```

After memory for the variables has been allocated, the expression on the right side of the assignment operator is evaluated, and the result is stored in the variable. An initialization expression for a class variable is executed only once, when the class is initialized. If the initialization expression is for an instance variable, it is executed every time an instance of the class is created. Initialization expressions can also appear in the declaration of local variables. Figure 3-7 illustrates these expressions in a class definition.

```
/**
 * Illustrate how to use initializers in a class definition
 */
public class Initializers {
    private static int a = 13;      // a is initialized to 13

    private static int b = a * 2;   // b is initialized to 26

    private int x = a;              // a copy of the value
                                    // in a is placed in x

    private int y = 0;              // y is initialized to 0

} // Initializers
```

[FIGURE 3-7] Initialization statements in a class declaration

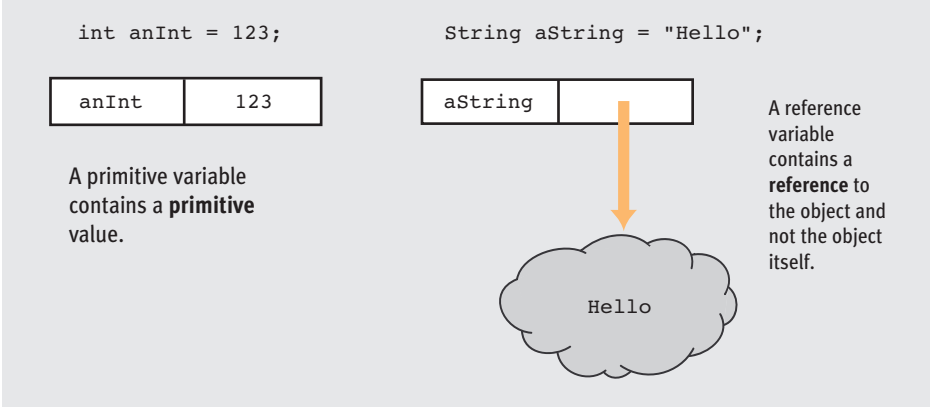
The use of initialization expressions is optional. By default, all instance and class variables are initialized—numerical variables are initialized to 0, character variables are initialized to the null character, Boolean variables are initialized to false, and reference variables are initialized to null. Even though the variables that represent the state of an object are initialized by default, it is good coding practice to explicitly initialize these variables, either when they are declared or (preferably) in the class constructor.

The default initialization rules apply only to instance and class variables. However, all variables in a Java class must be initialized before they can be used, which means local variables must be explicitly initialized before you can use them. A good habit is to explicitly initialize all local variables as soon as they are declared.

The standard programming rules apply to the expressions in initialization statements. What might not be as obvious is that instance variables cannot initialize the value of a class variable. Recall that a class variable is initialized when the class is initialized, which happens before any instances of the class are created. If there are no instances of the class, then the instance variables do not exist yet. If the instance variables do not yet exist, they cannot be used in an initialization expression. Even if instances of the class existed, how would the compiler know which instance variable to use to initialize the class variables if multiple instances exist?

The last modifier to discuss is `final`. A final variable can only be assigned a value when it is declared. Any attempt to change the value of the variable, except during initialization, is flagged as an error. In essence, when you declare a state variable as final, it behaves as a constant. The `final` modifier applies to the contents of the state variable, not to any object to which the variable may refer. For reference types, this means that the reference stored in the variable cannot change, but the object to which the reference refers may be changed.

To make this last point a little clearer, recall from our earlier discussion that variables in Java can be divided into two broad groups: primitives and references. Primitive variables are declared to be one of the following eight types: `char`, `byte`, `short`, `int`, `long`, `float`, `double`, or `boolean`. If a variable is defined to be of any other type, it is a reference variable. A primitive variable stores a value directly in memory, whereas a reference variable stores a reference to an object. This difference is illustrated in Figure 3-8.



[FIGURE 3-8] Primitive and reference variables

If the variables in Figure 3-8 were declared as `final`, it would not be possible to change the contents of either one. The compiler would reject any statement that attempted to change the 123 stored in `anInt` or a statement that tried to change the reference (such as the arrow in Figure 3-8) stored in the variable `aString`. In other words, the statements `anInt = 456;` or `aString = ADifferentStringVariable;` would be rejected by the compiler.

The most common use of `static` and `final` modifiers in variable declarations is in the creation of symbolic constants in a Java program. By convention, the name associated with a symbolic constant is written using all uppercase characters, and the underscore character (_) is used to separate words within the name. For example, `MAX_VALUE` is a valid name for a symbolic constant. Like the conventions for naming classes and variables, the compiler does not enforce naming conventions for symbolic constants.

Note that a symbolic constant does not have to be declared as `static`. A copy of a nonstatic constant is present in each instance of the class. If the variable is a constant, it is slightly more efficient to declare the constant as `static`. The `Point` class in Figure 3-9 has been modified to include a symbolic constant named `MAX_VALUE`, which you can use to determine the maximum value that can be assigned to either coordinate value.

```
/**
 * A class that defines a point
 */
public class Point {
    // Constant that gives the maximum value that may
    // be assigned to either the x- or y-coordinates
    public final static int MAX_VALUE = 1024;

    // State of a point
    private int xCoordinate;    // The x-coordinate
    private int yCoordinate;    // The y-coordinate

    // Behavior of a point
} // Point
```

[FIGURE 3-9] Adding a symbolic constant to the `Point` class

This section examined how the state associated with an object is modeled in Java using variables. To summarize, state information in Java is stored in instance and class variables. Instance variables are associated with an instance of a class, which means there is a different copy of an instance variable in each instance of a class. Class variables, on the other hand, belong to the class, which means that only one copy of the variable is shared by all instances of the class. Class variables are declared as `static`, but instance variables are not.

Every state variable has an associated accessibility. Members of any class can access public variables, but only members of the same class can access private variables. Variables without an explicit access modifier have package accessibility, meaning that only classes in the same package can access the variable. The modifier `final` can create read-only variables that are often used to create symbolic constants.

State is only one part of the definition of a class. In Chapter 2, you learned that an object has state, behavior, and identity. The next section discusses how to add methods to a class definition to define the behavior associated with an object.

3.2.2 Behavior

The behavior of an object is modeled in Java using **methods**. You can think of a method as a procedure, or function, associated with a particular class or object. To call, or invoke, the method, you must have a reference to the object with which the method is associated. The general form of a method declaration appears in Figure 3-10.

```
MethodModifiers ReturnType MethodName( FormalParameterList ){
    method body
}
```

[FIGURE 3-10] General form of a method declaration

In Chapter 2, we defined four categories of methods that can be associated with an object: accessors, mutators, constructors, and destructors. The code in Figure 3-11 contains two accessors: `getXCoordinate()` and `getYCoordinate()`. The `getXCoordinate()` method returns the value stored in the `xCoordinate` instance variable, and `getYCoordinate()` returns the value stored in the `yCoordinate` instance variable. Using these methods, you can access the state of a `Point` object. By convention, the name of an accessor method starts with the word *get*. This convention is not enforced by the compiler, but is stated in a coding standard. The code in Figure 3-11 would be placed in the `Point` class in Figure 3-9 below the comment that reads "Behavior of a point."

```
/**
 * @return the x-coordinate of this point
 */
public int getXCoordinate() {
    return xCoordinate;
}

/**
 * @return the y-coordinate of this point
 */
public int getYCoordinate() {
    return yCoordinate;
}
```

[FIGURE 3-11] Adding accessor methods to the `Point` class

Note the general form of a method declaration in Figure 3-10 and the method declarations in Figure 3-11. The method modifiers for both the `getXCoordinate()` and `getYCoordinate()` methods are `public`, and their return type is `int`.

A method modifier specifies a method's accessibility in the same way that the modifier associated with class or instance variables specifies the variable's accessibility. The method modifier determines which objects can see and subsequently invoke the method. The accessibility rules for methods are essentially the same as those of class and instance variable declarations. Any object, regardless of the class from which it was instantiated, can invoke methods that have been declared public. Only objects that are members of the same class are allowed to invoke private methods. Private methods behave like private instance variables in that the methods are private to the class, meaning that any object can access the private methods of any instance of

the same class. Finally, methods that do not specify a modifier have package scope, which means they can only be invoked by objects that are members of the same package.

The return type in a method declaration determines the type of value, if any, the method returns when invoked. In nonobject-oriented programming languages, methods that return values are referred to as **functions**. Every method, with the exception of a constructor, must specify a return type. If a method does not return a value (what is often termed a **procedure** in nonobject-oriented languages), the return type is specified as `void`. The methods in Figure 3-11 are all defined as returning integer values when they are invoked. This is specified in the method declaration by the appearance of the keyword `int` before the method name. A method can return a single value of any known type. If a method is defined to return a value, then a return statement that specifies a value of the correct type must appear somewhere in the method. Typically, but not always, the return statement is the last statement in a method. When a return statement is executed, the method is finished.

Mutators, which are defined in Table 2-1, change the state of an object. Mutators typically do not return a value and are declared using the `void` return type. A mutator often requires additional information that specifies how the state of the object is to be changed. The methods `setXCoordinate()`, `setYCoordinate()`, and `setXY()` in Figure 3-12 are all mutator methods in the `Point` class. The `setXCoordinate()` method changes the value of the `xCoordinate` instance variable, and the `setYCoordinate()` method changes the value of the `yCoordinate` instance variable. The `setXY()` method changes the x- and y-coordinate values at the same time. By convention, the names of mutator methods start with `set`. The code in Figure 3-12 would be placed inside the class definition shown in Figure 3-9.

```
/**
 * Set the x-coordinate of this point
 * @param newX the new x-coordinate
 */
public void setXCoordinate( int newX ) {
    xCoordinate = newX;
}

/**
 * Set the y-coordinate of this point
 * @param newY the new y-coordinate
 */
public void setYCoordinate( int newY ) {
     yCoordinate = newY;
}
```

continued

```
/**
 * Set both the x- and y-coordinates of this point
 * @param newX the new x-coordinate
 * @param newY the new y-coordinate
 */
public void setXY( int newX, int newY ) {
     xCoordinate = newX;
     yCoordinate = newY;
}
```

[FIGURE 3-12] Mutators for the `Point` class

Mutators, unlike accessors, usually do not return a value when invoked. The `void` return type in the declaration of the `setXCoordinate()` method specifies that it does not return a value. Additionally, a mutator must be provided with additional information to change the state of the object. For example, the `setXCoordinate()` method must be provided with the new value of the x-coordinate. These values, which are specified when the method is invoked, are referred to as **arguments**. A method's **formal parameters** define the number and types of arguments that must be passed to a method when it is invoked. The formal parameters are specified as a list of comma-separated variable declarations that appear inside the parentheses following the name of the method. For example, the declaration `int newX` in the definition of the `setXCoordinate()` method of Figure 3-12 specifies that when the method is invoked, the programmer must provide a single integer value as an argument. The comma-separated list `int newX, int newY` in the definition of the `setXY()` method in Figure 3-12 specifies that the programmer must provide two integer arguments when invoking the method. The empty formal parameter list in the `getXCoordinate()` method in Figure 3-11 specifies that no arguments are required when invoking the method.

Every method, whether it is an accessor or a mutator, has an associated signature. The method **signature** consists of the method name and the types of the formal parameters to the method. For example, the `getXCoordinate()` method in Figure 3-11 has the signature `getXCoordinate(void)`, where the word `void` indicates that the method has no formal parameters. The `setXY()` method in Figure 3-12 has the signature `setXY(int,int)`. The Java language specification requires that all of the methods in a class have different signatures. Because the method signature includes the number and types of formal parameters, you may have two or more methods that have the same name in a class, provided that their signatures are different.

Consider the second version of the `setXY()` method that has been added to the `Point` class in Figure 3-13. The first form of `setXY()` requires two integer parameters that specify the point's new x- and y-coordinates. The second form of `setXY()` requires only a single integer parameter that specifies the new value of both coordinates. Although the names of these methods are the same, their signatures `setXY(int,int)` and `setXY(int)` are distinct; thus, both forms of the method may be defined in the same class. The compiler can determine which method to use based on the number and types of the arguments.

```
/**
 * Set both the x- and y-coordinates of this point
 * @param newX the new x-coordinate
 * @param newY the new y-coordinate
 */
public void setXY( int newX, int newY ) {
     xCoordinate = newX;
     yCoordinate = newY;
}

/**
 * Set the x- and y-coordinate to the same value
 * @param value value for both the x- and y-coordinate
 */
public void setXY( int value ) {
     xCoordinate = value;
     yCoordinate = value;
}
```

[FIGURE 3-13] Overloaded methods in the `Point` class

Providing multiple forms of a method in the same class is a common practice; such methods are said to be **overloaded**. At first glance, overloading a method seems strange, but you have almost certainly used them before. Consider, for example, the addition operator (+). There is only one name (symbol) for the addition operation, yet the same symbol can add two integer values or two real values. The compiler determines which version of (+) to use based on the parameters passed to it when it is called. Note that because a method's signature in Java does not include the return type, it is not possible to have two or more methods with the same signature returning different types.

In the `Point` class of Figure 3-9, the comment before the constant `MAX_VALUE` states that the x- and y-coordinate values are never greater than the value stored in the constant. As the `Point` class is currently written, invoking the `setXCoordinate()` method with the argument 4096 is valid, although the value 4096 is outside the range of valid values. Invoking the method in this way results in an object whose internal state is inconsistent with its specification. Fortunately, correcting this problem is a simple matter; the mutator methods can be modified so that the value of the argument is verified as valid before assigning it to the appropriate instance variable. Only if the argument is within the range of valid values is it assigned to the instance variable. The `setXCoordinate()` and `setYCoordinate()` mutators in Figure 3-14 have been written so that you cannot set the x- or y-coordinates to invalid values.

```
/**
 * Set the x-coordinate of this point
 * @param newX the new x-coordinate
 */
public void setXCoordinate( int newX ) {
    if ( newX > MAX_VALUE ) {
        xCoordinate = MAX_VALUE;
    }
    else {
        xCoordinate = newX;
    }
}

/**
 * Set the y-coordinate of this point
 * @param newY the new y-coordinate
 */
public void setYCoordinate( int newY ) {
   if ( newY > MAX_VALUE ) {
       yCoordinate = MAX_VALUE;
   }
   else {
       yCoordinate = newY;
   }
}
```

[FIGURE 3-14] Safe mutators for the `Point` class

The mutators in Figure 3-14 illustrate why using public instance variables to store the state of an object is bad programming practice. If an instance variable is public, then any object can change its value to whatever it wants. On the other hand, if instance variables are declared private, then the only way to change their values is to use a mutator. The mutator method shown in Figure 3-14 can verify that the new value is valid and make a change only if it is appropriate. By declaring instance variables private and allowing access to these variables only through accessors or mutators, you control access to the instance variables and thus ensure that the state of the object is valid.

Neither version of the `setXY()` mutators in Figure 3-13 checks its parameters to ensure that they do not exceed the maximum value for a point. Changing these methods to make such checks is trivial; essentially, all you need to do is copy the code added to the `setXCoordinate()` and `setYCoordinate()` methods that perform the checks. However, it is rarely a good idea to copy code from one method to another. A better practice, when code is used by several methods, is to place the code in a single method and then invoke this

method to get the work done. If you isolate this shared code in a single method and then find an error or need to modify the code, you only need to make changes in one place rather than searching through the program for all occurrences of the duplicated code. In the `Point` class, the `setXCoordinate()` and `setYCoordinate()` methods have already been modified to check their parameter before assigning it the appropriate instance variable. Ideally, we would like the `setXY()` methods to use the `setXCoordinate()` and `setYCoordinate()` methods to change the values of the `xCoordinate` and `yCoordinate` instance variables.

The methods we have discussed up to this point are called instance methods. An **instance method** is associated with an instance of a class, and is always invoked with respect to an object. To invoke an instance method, you must specify both the name of the method you want to invoke and the object upon which the method is executed. In other words, it is not enough to specify that you want to invoke the `setXCoordinate()` method. You must also specify the object whose state will be changed as a result of executing the method. The general form of instance method invocation appears in Figure 3-15.

```
ObjectReference.MethodName( ArgumentList );
```

[FIGURE 3-15] Instance method invocation in Java

The *`ObjectReference`* in Figure 3-15 identifies the object upon which the method *`MethodName`* is executed. The *`ArgumentList`* specifies the actual arguments that are passed to the method when it is invoked. The type and number of the actual arguments must match the type and number of the formal parameters in the method definition.

Within a class definition, you can invoke methods within the same class without explicitly specifying the object. In other words, if the *`ObjectReference`* is omitted, the compiler assumes the method being invoked is in the class whose definition contains the method invocation and that the method is invoked on the current object. The `setXY()` mutators in Figure 3-16 have been modified to invoke the `setXCoordinate()` and `setYCoordinate()` methods to change the appropriate instance variables within the object.

```
/**
 * Set both the x- and y-coordinates of this point
 * @param newX the new x-coordinate
 * @param newY the new y-coordinate
 */
public void setXY( int newX, int newY ) {
    setXCoordinate( newX );
    setYCoordinate( newY );
}
```

continued

```
/**
 * Set the x- and y-coordinates to the same value
 * @param value value for both the x- and y-coordinate
 */
public void setXY( int value ) {
     setXCoordinate( value );
     setYCoordinate( value );
}
```

[FIGURE 3-16] Invoking a method

The third type of method defined in Table 2-1 is a constructor. A **constructor** initializes the state of an object. In the `Point` class, the constructor is used to initialize the values of the `xCoordinate` and `yCoordinate` instance variables because they make up the state of the object. A constructor always has the same name as the class, and cannot return a value. Java supports overloaded constructors so that a class can have any number of constructors. Overloaded constructors are often provided to give the user flexibility in the way that instances of the class can be created.

The `Point` class in Figure 3-17 contains two constructors: The first constructor requires two integer arguments that are used to initialize the x- and y-coordinates of the new `Point` object, and the second constructor initializes the x- and y-coordinates to the same value (similar in function to the overloaded version of the `setXY()` method in Figure 3-13). The compiler has no problem determining which constructor to use in a program because the signatures of the two constructors, `Point(int,int)` and `Point(int)`, are different.

```
/**
 * Create a point at the specified location
 * @param x the x-coordinate
 * @param y the y-coordinate
 */
public Point( int x, int y ) {
    setXY( x, y );
}

/**
 * Create a point at(value,value)
 * @param value value for both the x- and y-coordinates
 */
public Point( int value ) {
    setXY( value );
}
```

[FIGURE 3-17] Adding constructors to the `Point` class

The modifier associated with a constructor determines what objects can use the constructor to create instances of the class. A class with a `public` constructor can be invoked by any object, allowing any object to create an instance of the class. A class with a `private` constructor only allows objects that are members of the same class to create instances of the class. Because the constructors in Figure 3-17 are declared `public`, any class can create `Point` objects.

A constructor that takes no parameters is called the **default constructor** of the class. If you do not explicitly write a constructor for a class, the Java compiler automatically creates a default constructor that initializes all instance variables to the default values specified in Section 3.2.1. In this way, Java guarantees that every class contains at least one constructor.

In Java and other object-oriented programming languages, memory is allocated whenever a new object is created. When an object is no longer needed, the run-time system should be informed so that the memory allocated for that object can be used for other purposes. Failing to release the memory allocated by the program as it is running leads to a problem called a **memory leak**, which can monopolize all of the memory resources in a computer over time. It is not difficult to keep track of allocated memory so it can be reused when it is no longer needed, but it is tedious. The programmer must be careful to write classes so that memory resources allocated to an object are released when it is no longer needed. Most object-oriented programming languages assist in this effort by guaranteeing that any memory allocated by the compiler is released, but it is often not possible for memory that has been dynamically allocated by the object during its lifetime.

A **destructor**, the fourth type of method introduced in Chapter 2, provides a mechanism for a programmer to release any system resources, including memory, that have been allocated by an object before it is removed from the system. Given the important role that a destructor plays in an object-oriented system, you might be surprised to learn that Java has no destructors. The closest thing in Java to a destructor is the `finalize()` method, which is discussed in Section 3.3.5.

The Java run-time system automatically tracks the use of memory and releases memory resources when it determines they are no longer needed. The Java run-time system uses a technique called **garbage collection** to manage memory. In addition to keeping track of all the objects in existence while a program is running, the Java run-time system also keeps track of the total number of references to every object. Because the only way to access an object is through a reference, when the reference count of an object goes to zero, no other object in the program can access it. An object that can no longer be accessed is called **garbage**; from time to time, the Java run-time system makes a sweep through memory and deletes any objects marked as garbage.

In Section 3.2.1, we made a distinction between instance and class variables. Instance variables are associated with an instance of a class, whereas class variables are associated with a class. In a similar fashion, a distinction can be made between instance methods, which are associated with an instance of a class, and class methods. **Class methods**, like class variables, are associated with a class and not an instance of a class.

The class method `getNumPoints()` has been added to the `Point` class in Figure 3-18. This accessor method returns the current value of the class variable `numPoints`. The constructors in Figure 3-18 have been modified to add 1 to the current value of `numPoints` every time a constructor is invoked. Because `numPoints` is a class variable and not an instance variable, there is exactly one copy of `numPoints`, regardless of how many `Point` objects have been instantiated. Because the constructors increment the same copy of `numPoints`, this variable indicates how many times the constructors have been invoked. This means that `numPoints` keeps track of the number of `Point` objects that have been created. Note that `numPoints` does not indicate how many `Point` objects are still in existence (in other words, not garbage collected).

```
/**
 * A class that defines a point
 */
public class Point {
    // Constant that gives the maximum value that may
    // be assigned to either the x- or y-coordinates
    public final static int MAX_VALUE = 1024;

    // State of a point
    private int xCoordinate;    // The x-coordinate
    private int yCoordinate;    // The y-coordinate

    // The number of Point objects created so far
    private static int numPoints = 0;

    /**
     * Return the number of Point objects created so far
     * @param the number of Point objects created
     */
    public static int getNumPoints() {
        return numPoints;
    }

    /**
     * Create a new point at the specified location
     * @param x the x-coordinate
     * @param y the y-coordinate
     */
    public Point( int x, int y ) {
        setXY( x, y );
        numPoints = numPoints + 1;
    }
```

continued

```
    /**
     * Create a new point at (value,value)
     * @param value the value to set the x- and y-coordinates to
     */
    public Point( int value ) {
        setXY( value );
        numPoints = numPoints + 1;
    }

    // Rest of class definition omitted
}
```

[FIGURE 3-18] Adding a class method to the `Point` class

Class methods are invoked just like instance methods, but instead of specifying the object upon which the method is executed, you specify the class with which the method is associated. If you do not specify a class when invoking a class method, the compiler assumes that the method is defined in the same class in which it is being invoked.

Whenever an instance method is invoked, the method is executed on an object, which means that the method can access the state of the object on which it was invoked. Because a class method is associated with the class and not one of its instances, the class method cannot access the state of an object. Likewise, a class method cannot invoke an instance method without specifying the object upon which the method should be invoked. In the context of the `Point` class, the class method `getNumPoints()` cannot invoke the `getXCoordinate()` method, nor can it access the `xCoordinate` instance variable without specifying an object reference. This makes perfect sense if you think about the nature of a class method. Because a class method is associated with a class and not with an instance, you can access a class method before any instances of the class have been instantiated. A class method, however, cannot access an instance of its own class without specifying a reference to an object, because it is possible that no such instances exist.

You may be wondering why an object-oriented programming language, such as Java, would include the ability to define class methods. As it turns out, class methods can be useful when writing class definitions. Consider, for example, writing a class that consists exclusively of `static` variables and `static` methods. It would be possible to access any member of this class without ever creating an instance of the class. A class defined in this way could be viewed as a repository of methods. Probably the best example of this type of class is the `Math` class provided by the standard Java application programming interface (API) in the `java.lang` package. Table 3-2 lists a few of the methods defined in `java.lang.Math`. The Javadoc page for the class lists all the methods in the class.

METHOD	DESCRIPTION
`static double abs( double a )`	Returns the absolute value
`static double cos( double a )`	Returns the cosine of an angle
`static double max( double a, double b )`	Returns the larger argument
`static double pow( double a, double b )`	Returns the result of raising the first argument to the power of the second
`static double sqrt( double a )`	Returns the square root

[TABLE 3-2] Some useful methods defined in `java.lang.Math`

The class `java.lang.Math` contains more than 30 methods that provide implementations of the most common math functions that might be needed in a Java program. The instance method `distanceFrom()` in Figure 3-19 shows how two of the methods in the `Math` class can be used to determine the distance between a `Point` object, specified by the instance variables `xCoordinate` and `yCoordinate` and the origin. Methods from the `Math` class are invoked using a class reference, as opposed to an object reference.

```
/**
 * Compute the distance between this point and the origin
 */
public double distanceFrom() {
    return ( Math.sqrt( Math.pow( xCoordinate, 2 ) +
                        Math.pow( yCoordinate, 2 ) ) );
}
```

[FIGURE 3-19] Computing the distance between a point and the origin

We have spent considerable time talking about classes and objects in this chapter, but we have not yet discussed how to write a Java program. In Section 3.1, we stated that an object-oriented program consisted of a number of objects that interact to accomplish a specified task. You now know how to create the classes that can instantiate the objects that interact in the program, but you still do not know how to start this process. It would seem that to run a Java program, you would have to identify a class that the Java run-time system would use to instantiate the first object that would start the program. However, there are problems with this approach: How do you pass the parameters to the constructors of the class, and how do you specify which method should be invoked once the object has been created? What we need is a special method that the Java run-time system can invoke without having

to instantiate an object. If you give this a little thought, you will realize that a class method fills this role quite effectively.

In Java, the class method `main()` provides a way to begin a Java program. When the Java run-time system is started, it is provided with the name of a class that contains a `main()` method. After the run-time system has initialized its environment, it invokes the `main()` method in the class that was specified on the command line. Figure 3-20 contains an example of a class that contains a `main()` method. If this class is passed to the Java run-time system, the program prints the string "Hello World" and terminates.

```
/**
 * A Hello World program in Java
 */
public class HelloWorld {
    /**
     * The entry point for this class; when executed,
     * the program prints "Hello World" and terminates
     * @param args command-line arguments
     */
    public static void main( String args[] ) {
        System.out.println( "Hello World" );
    }
} // HelloWorld
```

[FIGURE 3-20] Using `main()` in Java

Even though you can execute the class in Figure 3-20 from the command line, it is defined like any other class in Java. What makes `main()` different from all other class methods is that the Java run-time system invokes it to start a program. There is nothing special about the `main()` method; it is merely a `public` class method that does not return a value when invoked. For the run-time system to be able to invoke `main()`, the method must be defined exactly as shown in Figure 3-20 (meaning it must have the signature `main( String[] )`. Finally, any class can have a `main()` method; adding one to a class does not force you to "run" that class from the command line.

Based on the signature of the `main()` method, the method must be provided with an array that contains references to `String` objects. When the Java run-time system invokes `main()`, it passes an array that contains references to the command arguments that were specified by the user. The arguments passed to `main()` include everything that is typed on the command line after the class name. The `main()` method in the class of Figure 3-21 prints the contents of the array passed as an argument to the method.

```
/**
 * A program that echoes the command line
 */
public class EchoCommandLine {
    /**
     * Print the contents of the args array that is
     * passed to the method when it is invoked
     * @param args the command-line arguments
     */
    public static void main( String args[] ) {
        // Print args
        for ( int i = 0; i < args.length; i++ ) {
            System.out.println( args[ i ] );
        }
    }
} // EchoCommandLine
```

[FIGURE 3-21] Program that prints the contents of the `args` array

The `main()` method in Figure 3-21 uses a simple loop to iterate over the elements in the array passed in as a parameter. There is no need to pass the number of arguments to the `main()` method because this information can be obtained from the array using the `length` instance variable. The output generated by the program in Figure 3-21 is shown in Figure 3-22—twice with different command-line arguments each time.

```
% java EchoCommandLine arg1 arg2 arg3
arg1
arg2
arg3
%java EchoCommandLine
%
```

[FIGURE 3-22] Execution of the `EchoCommandLine` program

This section has illustrated how the behavior of an object is modeled in Java using methods. A method is nothing more than a function, or a procedure, associated with a class or an object. Like instance variables, methods have an accessibility that determines how they can be accessed. As in other programming languages, the definition of a method specifies the type of the value returned by the method, if any, and the number and type of formal parameters that must be provided when the method is invoked.

Every method in Java has an associated signature, which consists of the method name followed by the number and types of its formal parameters. All of the methods in a Java class must have unique signatures. Overloaded methods are methods in a single class that have the same name but different signatures.

Finally, methods, like instance variables, can be declared as `static`. Class methods, or static methods, are associated with a class and not with an object. One of the most common uses of a `static` method is to write a `main()` method that can be invoked by the Java run-time system; such methods serve as the starting point for execution of most Java programs.

Up to this point in the chapter, we have discussed how the state and behavior of an object are defined, but you have not yet learned how to access the state of an object or invoke its methods. The next section discusses how objects are created from classes and how to access the state and behavior of these objects.

PAIR PROGRAMMING

The stereotypical view of a programmer is of the "lone wolf," a socially inept person working through the night, eating pizza, and rarely sleeping. He appears, days later, with a working version of the software that no one else in the company can begin to comprehend.

Today that view is completely out of date. Given the cost and complexity of software, a company cannot afford quirky, idiosyncratic code that has not been subject to rigid quality control throughout its life cycle. Furthermore, if a developer left a company for a better job, no one would be left who was familiar with a particular project. For these reasons, all software today is designed and implemented by teams of developers working as a coordinated unit.

One of the most interesting approaches is **pair programming**, a fundamental characteristic of the development philosophy called extreme programming, which you read about in Chapter 2. In pair programming, programmers work in pairs on a single piece of code at a single workstation. One person, the *driver*, enters the code or test data at the keyboard. The other person, the *navigator*, watches the driver and makes suggestions, checks for correctness, and evaluates the quality of the code being entered.

The users of pair programming claim that this coordinated approach to software development has a number of important benefits:

- *Better code*—When two minds work on a single problem, it is much more likely that one of them will come up with a good solution.
- *Fewer errors*—It is often difficult for people to locate their own mistakes. We often see what we want to see, not what is actually there. An independent observer can locate more errors and correct them.

continued

- *Uniform coding style*—Because your partner (and others) must understand what you are writing, you cannot use nonstandard coding styles. Your code must be readable and legible.
- *Increased discipline*—When someone is looking over your shoulder, you are much less likely to take shortcuts, skip key development steps, or take long breaks.
- *Continuity*—At least two people are always familiar with the code of every unit in the software package.

Preliminary studies of pair programming have shown significant increases in productivity as well as programmer morale. In the case of software development, it does seem that "two heads are better than one."

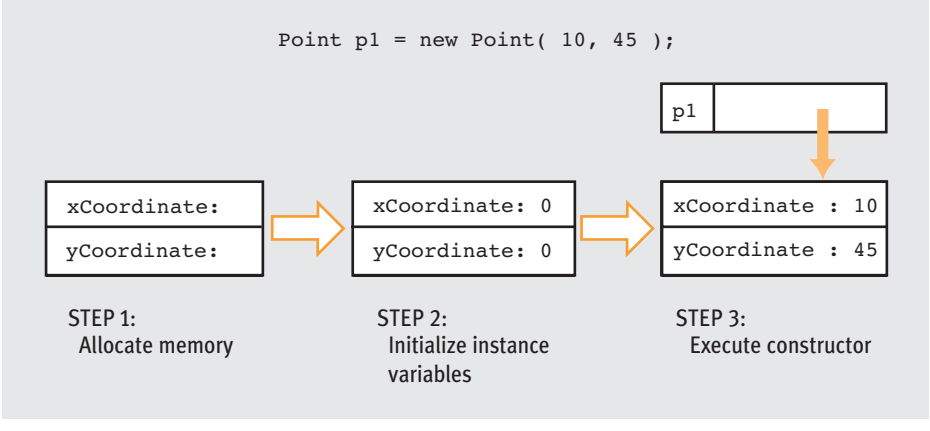
3.2.3 Identity

Once a class definition is available, you can use the Java operator `new` to create instances of the class. This operator takes as parameters any arguments that are passed to the appropriate constructor and returns a reference to the newly instantiated object. At compile time, the compiler checks to make sure that a constructor matches the arguments to `new`. If no such constructor is found, the compiler generates an error message. The program in Figure 3-23 illustrates how the `new` operator can create several points at different locations.

```
/**
 * This program creates several points at different locations
 */
public class CreatePoints {
    /**
     * The main method
     * @param args command-line arguments (ignored)
     */
    public static void main( String args[] ) {
        Point p1, p2, p3;

        p1 = new Point( 10, 45 ); // Create point at (10,45)
        p2 = new Point( 10, 10 ); // Create point at (10,10)
        p3 = new Point( 10, 10 ); // Create point at (10,10)
    }
} // CreatePoints
```

[FIGURE 3-23] Using the `new` operator

Instantiating an object is a three-step process: (1) the memory required to store the object is allocated, (2) the default initialization rules and/or initialization expressions, if any, are applied to the instance variables, and (3) the appropriate constructor is invoked to initialize the state of the object. The `new` operator performs all three steps, as shown in Figure 3-24.



[FIGURE 3-24] Instantiating an object

As you can see, the Java run-time system uses the default initialization rules described in Section 3.2.1 to initialize the instance variables of a class before the constructor is invoked. Although it is not strictly necessary to initialize the state of an instance variable using an initialization expression or within a constructor, it is good programming practice to explicitly initialize the instance variables of a class. This makes it absolutely clear what the initial value is.

Now that we know how to create objects, we can invoke methods on these objects using the syntax in Figure 3-15. Remember, to invoke an instance method, you have to specify the object upon which the method is invoked and the name of the method to invoke. The program in Figure 3-25 creates three `Point` objects, invokes mutators on some of the points to modify their state, and then prints the resulting coordinates by invoking the appropriate accessor methods. The output is shown in Figure 3-26.

```
/**
 * This program creates several points at different locations
 */
public class CreatePoints {
    /**
     * The main method
     * @param args command-line arguments (ignored)
     */
```

continued

```
    public static void main( String args[] ) {
        Point p1, p2, p3;

        p1 = new Point( 10, 45 ); // Create point at (10,45)
        p2 = new Point( 10, 10 ); // Create point at (10,10)
        p3 = new Point( 10, 10 ); // Create point at (10,10)

        // Place p1 at the origin
        p1.setXY( 0 );

        // Locate p2 at the same location as p1
        p2.setXCoordinate( p1.getXCoordinate() );
        p2.setYCoordinate( p1.getYCoordinate() );

        // Place p3 10 units away from p2 in each dimension
        p3.setXCoordinate( p2.getXCoordinate() + 10 );
        p3.setYCoordinate( p2.getYCoordinate() + 10 );

        // Print the results
        System.out.println("p1 -> " + pointToString( p1 ));
        System.out.println("p2 -> " + pointToString( p2 ));
        System.out.println("p3 -> " + pointToString( p3 ));
    }

    /**
     * Convert the given point to string form suitable for
     * printing
     * @param p the point to convert to a string
     */
    public static String pointToString( Point p ) {
        return "(" + p.getXCoordinate() + "," +
                     p.getYCoordinate() + ")";
    }
} // CreatePoints
```

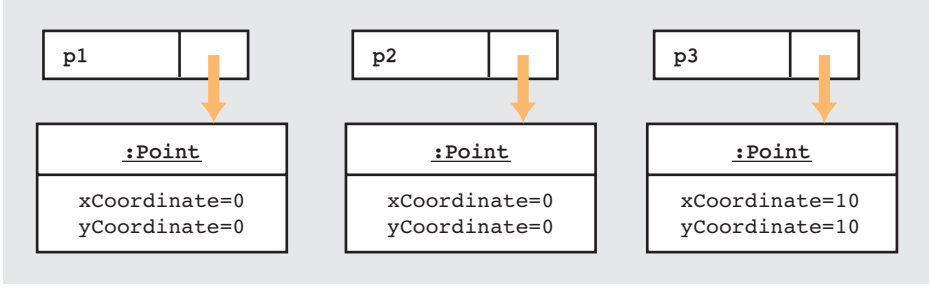
[FIGURE 3-25] Invoking methods

```
p1 -> (0,0)
p2 -> (0,0)
p3 -> (10,10)
```

[FIGURE 3-26] Output generated by `CreatePoints` program in Figure 3-25

The variables `p1`, `p2`, and `p3` in Figure 3-25 hold references to the objects created by `new`. The references serve as a means of identity for an object; using a reference, you can specify the object on which you want to invoke a method. It is important to remember that

p1, p2, and p3 are not points; they are reference variables that hold references to different Point objects, as illustrated in Figure 3-27.



[FIGURE 3-27] Reference variables

Consider the program in Figure 3-28. Like the program in Figure 3-25, it creates three points. In Figure 3-28, however, the reference variable p3 is initialized using the value of the variable p2.

```
/**
 * This program creates several points at different locations
 */
public class AssignPoints {
    /**
     * The main method
     * @param args command-line arguments (ignored)
     */
    public static void main( String args[] ) {
        Point p1, p2, p3;

        p1 = new Point( 10, 45 ); // Create point at (10,45)
        p2 = new Point( 10, 10 ); // Create point at (10,10)
        p3 = p2  // Use assignment to initialize p3 so that
                 // it refers to a point at (10,10)

        // Place p1 at the origin
        p1.setXY( 0 );

        // Locate p2 at the same location as p1
        p2.setXCoordinate( p1.getXCoordinate() );
        p2.setYCoordinate( p1.getYCoordinate() );
```

continued

```
        // Place p3 10 units away from p2 in each dimension
        p3.setXCoordinate( p2.getXCoordinate() + 10 );
        p3.setYCoordinate( p2.getYCoordinate() + 10 );

        // Print the results
        System.out.println("p1 -> " + pointToString( p1 ));
        System.out.println("p2 -> " + pointToString( p2 ));
        System.out.println("p3 -> " + pointToString( p3 ));
    }

    /**
     * Convert the given point to string form suitable for
     * printing
     * @param p the point to convert to a string
     */
    public static String pointToString( Point p ) {
        return "(" + p.getXCoordinate() + "," +
                     p.getYCoordinate() + ")";
    }
} // AssignPoints
```

[FIGURE 3-28] Using assignment to initialize a `Point` variable

You would think that the output from the program in Figure 3-28 would be the same as that in Figure 3-25 because the only change was the way the variable `p3` was initialized. The output from this program is shown in Figure 3-29.

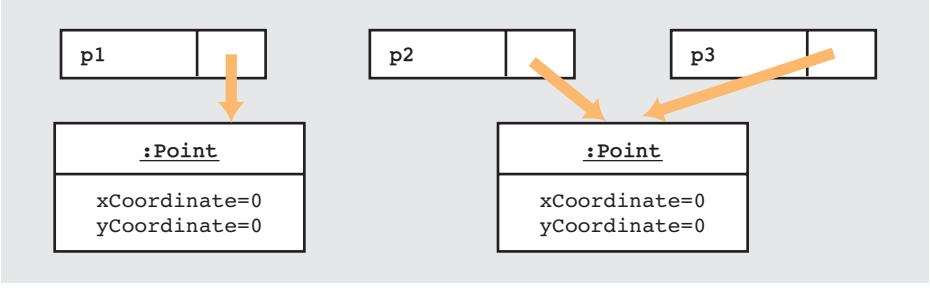
```
p1 -> (0,0)
p2 -> (10,10)
p3 -> (10,10)
```

[FIGURE 3-29] Output generated by the `AssignPoints` program in Figure 3-28

The key to understanding the output by the program in Figure 3-28 is to remember that a reference variable contains a reference to an object, not the object itself. The reference variable holds the identity, or name, of the object. When the assignment operator is applied to a reference variable, the identity of the object is duplicated, not the object that the name identifies. This is an example of the **shallow copy** operation discussed in Chapter 2. The assignment operation, when applied to reference types, performs a shallow copy. It only affects the reference, or name, of the object and not the object itself.

Consider the effect of executing the statement `p3 = p2` in the program in Figure 3-28. The assignment operator copies the contents of the variable on the right side of the assignment operator into the variable on the left side. If these variables were primitive types, the assignment operation would create a new primitive value that is placed into the variable on

the left side. In the case of reference variables, the assignment operator performs the same operation, but here the reference to the object on the right side of the operator is copied, not the object itself. The results are shown in Figure 3-30.



[FIGURE 3-30] Object diagram after executing `p3 = p2`

After the assignment has been executed, `p2` and `p3` both contain a reference to the same object. Assignment does not create a new object; it only copies the value of one reference variable and places it in a second reference variable. The only way to create a completely new object in Java is by using the `new` operator. After the assignment operation is completed, the variables `p2` and `p3` are **name equivalent**. Any attempt to invoke a method on the object referred to by `p2` has the same effect as invoking the same method using `p3`. The invocation of the `setXCoordinate()` and `setYCoordinate()` methods changes the state of the single object referred to by both `p2` and `p3`, even though only the reference variable `p2` is used when the methods are invoked. In fact, although the output in Figure 3-29 appears to list the state of three `Point` objects, it actually lists the state of only two. The last two lines in the output refer to the same `Point` object. When two variables reference the same object, they are said to be **aliases**.

The Java assignment operation performs a shallow copy because it only operates on the contents of the variables, not the objects to which they might refer. There is no "deep assignment" operator in Java; however, you can add methods to a class definition that allow you to create duplicates of an object. Consider the constructor for the `Point` class in Figure 3-31.

```
/**
 * Create a duplicate Point object
 * @param p the point to duplicate
 */
public Point( Point p ) {
    xCoordinate = p.xCoordinate;
    // OR xCoordinate = p.getXCoordinate()
```

continued

```
    yCoordinate = p.yCoordinate;
    // OR yCoordinate = p.getYCoordinate()

    numPoints = numPoints + 1;
}
```

[FIGURE 3-31] Adding a copy constructor to the `Point` class

The constructor added to the `Point` class initializes the state of the new `Point` object using the state of an existing `Point` object. This type of constructor is sometimes referred to as a **copy constructor** because it provides a mechanism to create objects that are copies of existing objects. The program in Figure 3-32 uses the copy constructor to initialize the reference variable `p3`. This program creates three distinct `Point` objects because `new` is invoked three times.

```
/**
 * This program creates several points at different locations
 */
public class AssignPoints {
    /**
     * The main method
     * @param args command-line arguments (ignored)
     */
    public static void main( String args[] ) {
        Point p1, p2, p3;

        p1 = new Point( 10, 45 ); // Create point at (10,45)
        p2 = new Point( 10, 10 ); // Create point at (10,10)
        p3 = new Point( p2 );      // Use a copy constructor so p3
                                   // refers to a point at (10,10)

        // Place p1 at the origin
        p1.setXY( 0 );

        // Locate p2 at the same location as p1
        p2.setXCoordinate( p1.getXCoordinate() );
        p2.setYCoordinate( p1.getYCoordinate() );

        // Place p3 10 units away from p2 in each dimension
        p3.setXCoordinate( p2.getXCoordinate() + 10 );
        p3.setYCoordinate( p2.getYCoordinate() + 10 );

        // Print the results
        System.out.println("p1 -> " + pointToString( p1 ));
        System.out.println("p2 -> " + pointToString( p2 ));
        System.out.println("p3 -> " + pointToString( p3 ));
    }
```

continued

```
    /**
     * Convert the given point to string form suitable for
     * printing
     * @param p the point to convert to a string
     */
    public static String pointToString( Point p ) {
        return "(" + p.getXCoordinate() + "," +
                     p.getYCoordinate() + ")";
    }
} // AssignPoints
```

[FIGURE 3-32] Using the copy constructor to initialize a `Point` variable

The reference variables `p2` and `p3` in this program, like the one in Figure 3-28, refer to `Point` objects whose x- and y-coordinates are 10. However, in this program `p2` and `p3` refer to *different* `Point` objects. So, if a change is made to the state of the object referred to by `p2`, it does not cause a change in the state of the object referred to by the variable `p3`. Therefore, the output from this program is the same as that from the program in Figure 3-25.

Now that you understand how assignment works in Java, we can discuss the details of parameter passing. Parameter passing refers to the process by which Java associates the arguments specified in a method invocation with the parameters in the definition of the method being invoked. In Java, parameters are **passed by value**, which means that whenever an argument is passed to a Java method, a copy is made of the value stored in the argument, and the copy is passed to the method. Because the parameters are only copies of the arguments, changes made to the parameters during the execution of the method are not seen outside the method (meaning that the changes do not affect the values of the corresponding arguments).

One way to understand how parameters are passed is to view the parameters of a method as local variables that have been defined within the method. Parameters, like local variables, are created when the method is invoked, and are destroyed when the method returns. Unlike a local variable, however, the initial value of a parameter depends on the value of the corresponding argument. When a method is invoked, the values of the arguments are assigned to the corresponding parameter.

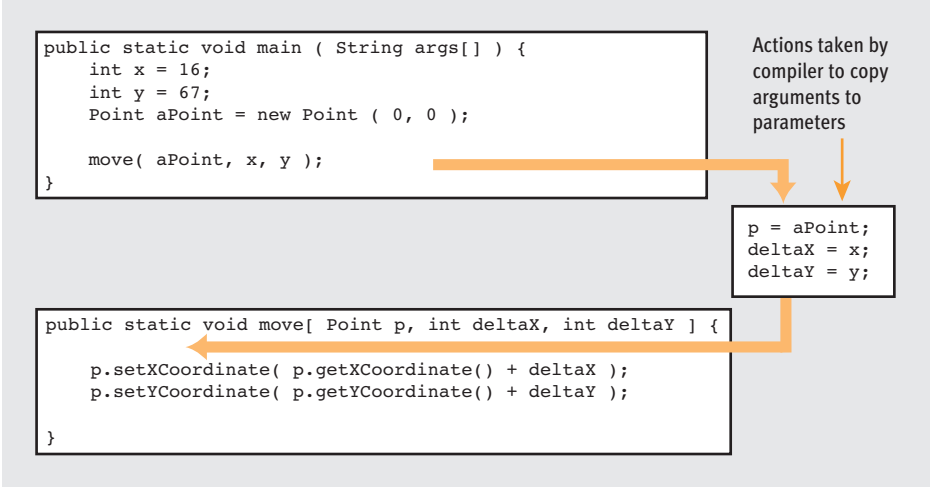
Consider the `main()` method in Figure 3-33, which invokes the `move()` method to move the `Point` object referred to by the variable `aPoint` 16 units along the x-axis and 67 units along the y-axis. When the `main()` method invokes `move()`, the compiler copies the values in the arguments (`aPoint`, `x`, and `y`) to the parameters (`p`, `deltaX`, and `deltaY`) using the basic assignment operation shown in Figure 3-34.

```
/**
 * A simple program to demonstrate parameter passing in Java
 */
public class Param {
    /**
     * Create and initialize a point and an int value.
     * Print the values of these variables before
     * and after invoking the method move().
     *
     * @param args command-line arguments (ignored)
     */
    public static void main( String args[] ) {
        int x = 16;
        int y = 67;
        Point aPoint = new Point( 0, 0 );

        // Before...
        System.out.println( "x=" + x );
        System.out.println( "y=" + y );
        System.out.println( "aPoint=" + pointToString(aPoint) );

        move( aPoint, x, y );

        // After...
        System.out.println( "x=" + x );
        System.out.println( "y=" + y );
        System.out.println( "aPoint=" + pointToString(aPoint) );
    }

    /**
     * Move a point
     *
     * @param p the point to move
     * @param deltaX amount to move the x-coordinate
     * @param deltaY amount to move the y-coordinate
     */
    public static void move( Point p, int deltaX, int deltaY ) {
        p.setXCoordinate( p.getXCoordinate() + deltaX );
        p.setYCoordinate( p.getYCoordinate() + deltaY );
    }

    /**
     * Convert the given point to string form suitable for
     * printing
     *
     * @param p the point to convert to a string
     */
```

continued

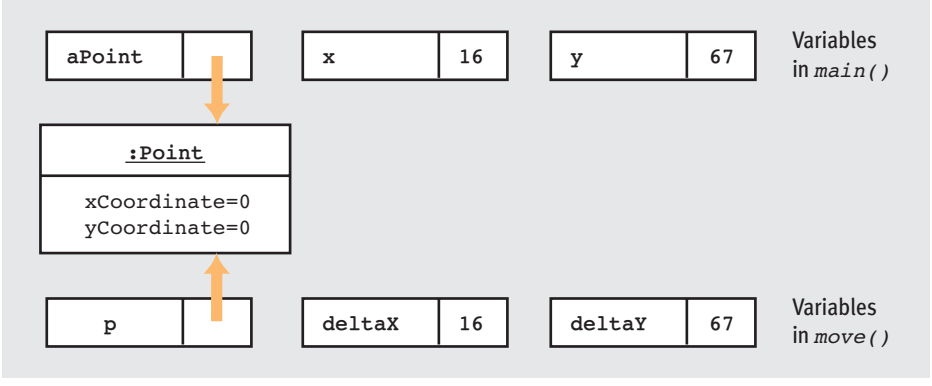
```
    public static String pointToString( Point p ) {
        return "(" + p.getXCoordinate() + "," +
                     p.getYCoordinate() + ")";
    }

} // Param
```

[FIGURE 3-33] Parameter passing in Java



[FIGURE 3-34] Parameter passing

Because assignment is used to initialize the values of the parameters to a method, the copies are shallow, which means if a reference variable is passed to a method, the reference is duplicated and not the object that is referenced. So, although changes made to the reference variable are not seen outside the method, if the method uses the reference variable to change the state of the object to which it refers, the change is seen by any method that has a reference to the affected object. This behavior is diagrammed in Figure 3-35.



[FIGURE 3-35] Object diagram during invocation of `move()`

The equality operator (==), like assignment, is a shallow operator that compares the references stored in the variables and not the objects to which the variables refer. The equality operator, when applied to reference variables, returns true if the references stored in the two variables being compared are the same. In other words, the equality operator returns true when two reference variables refer to the same object. It returns false when the two reference variables refer to different objects, even if the state of the objects is the same. Consider the program in Figure 3-36.

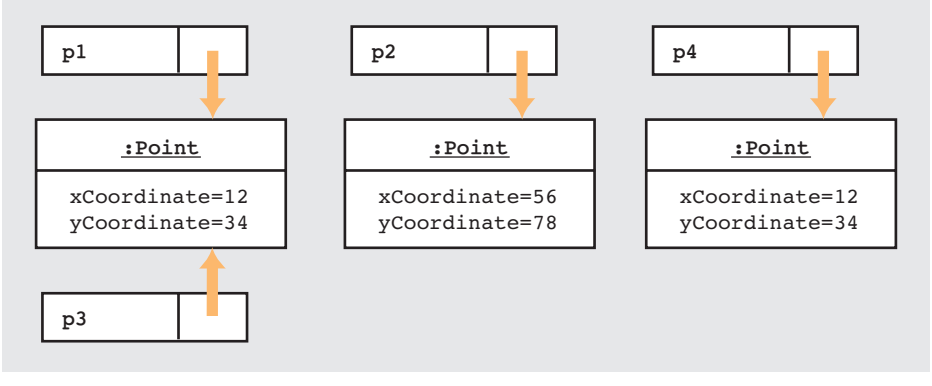
```
/**
 * Test the results of applying the equality operator ( == ) to
 * different points
 */
public class Equality {
    public static void main( String args[] ) {
        // Create some points
        Point p1 = new Point( 12, 34 );
        Point p2 = new Point( 56, 78 );
        Point p3 = p1;
        Point p4 = new Point( 12, 34 );

        // False; two different points
        System.out.println( p1 == p2 );

        // True; both refer to the same object
        System.out.println( p1 == p3 );
```

continued

```
        // False; two different objects, even though their
        // state is identical
        System.out.println( p1 == p4 );
    }
} // Equality
```

[FIGURE 3-36] Using the equality operator

The diagram in Figure 3-37 illustrates the reference variables and objects that are created by the program in Figure 3-36. Clearly, because `p1` and `p3` are the only two reference variables that refer to the same object, the test `p1==p3` returns true. Any other statements that compare `p1` to any other variables in the program (`p2` and `p4`) return false, even though the state of the objects referred to by these variables is the same.

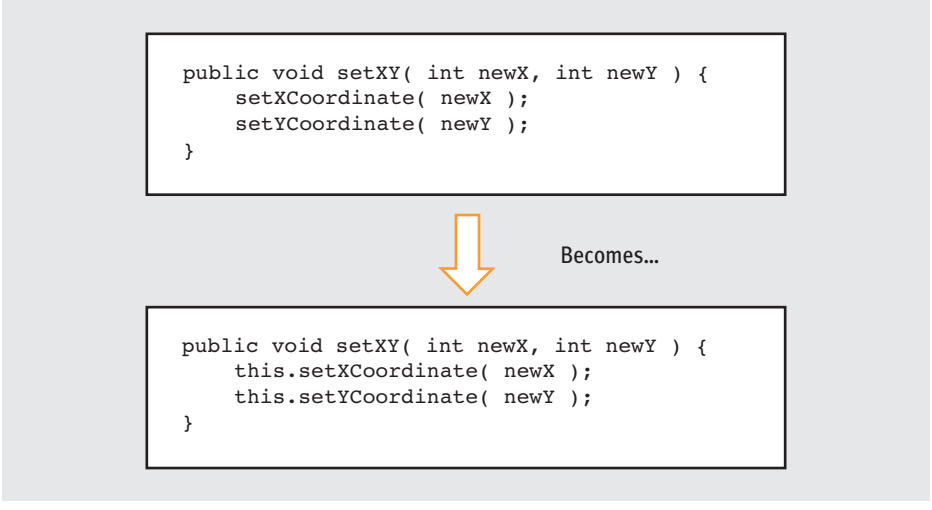


[FIGURE 3-37] Object diagram for the program in Figure 3-36

The last piece of Java syntax we address in our discussion of identity is the keyword `this`. You can only use this keyword in the body of an instance method, in a constructor, or in the initializer of an instance variable of a class. When used in an expression, `this` acts as a reference variable that contains a reference to the object upon which the method was invoked or to the object that is being constructed. The keyword `this` is automatically declared and initialized by the Java compiler.

Before we discuss how you can use the keyword `this` in a program, let's take a minute to explain the method invocation syntax discussed earlier in this chapter. In Figure 3-15, we implied that whenever you invoke a method, you place a reference to the object whose method you want to invoke on the left side of the dot (.) and the method name on the right side of the dot. You may have noticed that in all of the `Point` class examples, we left out the object reference and the dot whenever invoking a method within the same object. In an instance method, whenever Java sees a method invocation that does not specify an

object, it automatically inserts an implicit `this` in front of the method invocation, as shown in Figure 3-38. This conforms to the method invocation syntax in Figure 3-15.



[FIGURE 3-38] Implicit `this` placed in front of method invocations

When writing Java programs, the keyword `this` comes in handy in a couple of common situations. The first occurs whenever an object needs to pass a reference to itself as an argument to a method, or needs to return a reference to itself from a method. Another common situation occurs when the names of parameters to a method, or local variables defined in a method, have the same name as instance variables in the class. Consider the `setXCoordinate()` method in Figure 3-39. The parameter list for this method has been changed so that the parameter name is the same as the name of an instance variable of the class.

```
/**
 * Set the x-coordinate of this point
 * @param xCoordinate the new x-coordinate
 */
public void setXCoordinate( int xCoordinate ) {
    if ( xCoordinate > MAX_VALUE ) {
        this.xCoordinate = MAX_VALUE;
    }
    else {
        this.xCoordinate = xCoordinate;
}
```

[FIGURE 3-39] Using `this` in an expression

Within the `setXCoordinate()` method, the parameter `xCoordinate` shadows or hides the instance variable with the same name from view. As long as the parameter is in scope, any reference made within the method to the name `xCoordinate` refers to the parameter and not the instance variable. Using the keyword `this`, it is possible to access the instance variable of the same name. The expression `this.xCoordinate` refers to the instance variable `xCoordinate` instead of the parameter `xCoordinate`. Therefore, the assignment statement `this.xCoordinate = xCoordinate` copies the value stored in the parameter to the appropriate instance variable. Note that omitting the keyword `this` from the assignment statement, while still syntactically valid, results in an expression that copies the reference stored in the parameter `xCoordinate` to itself. It should be clear that shadowing instance variables is poor programming. It clearly makes a program more difficult to read, and worse yet, if you forget to place the keyword `this` in the appropriate place, your program is likely to fail.

You can also use the keyword `this` within a constructor as a special constructor invocation statement. Using `this`, you can write a constructor that invokes another constructor. If you think about the constructors added to the `Point` class in Figure 3-17, you can see that each constructor does basically the same thing; the only difference is how the initial x and y values are passed as parameters. A common strategy when writing constructors for a class is to write one constructor that actually does the initialization and to write others that simply call the real constructor with the appropriate arguments. Consider the constructors in the `Point` class of Figure 3-40.

```
/**
 * Create a new point at the origin
 */
public Point() {
    this( 0, 0 );
}

/**
 * Create a new point at ( value, value )
 * @param value the value to set the x- and y-coordinate to
 */
public Point( int value ) {
    this( value, value );
}

/**
 * Create a new point at the specified location
 * @param x the x-coordinate
 * @param y the y-coordinate
 */
public Point( int x, int y ) {
    setXY( x, y );
}
```

[FIGURE 3-40] Using `this` in constructors

In Figure 3-40, the constructor with the signature `Point(int, int)` is the one that does the actual work. The other constructors use the keyword `this` to invoke the `Point(int, int)` constructor with the appropriate arguments. When used as a constructor invocation statement, the keyword `this` must be the first statement within the body of the constructor. Using the keyword `this` to invoke constructors has the benefit of reducing the amount of duplicate code that you need to write and places all of the real initialization work in a single method, which makes debugging and maintaining the class easier.

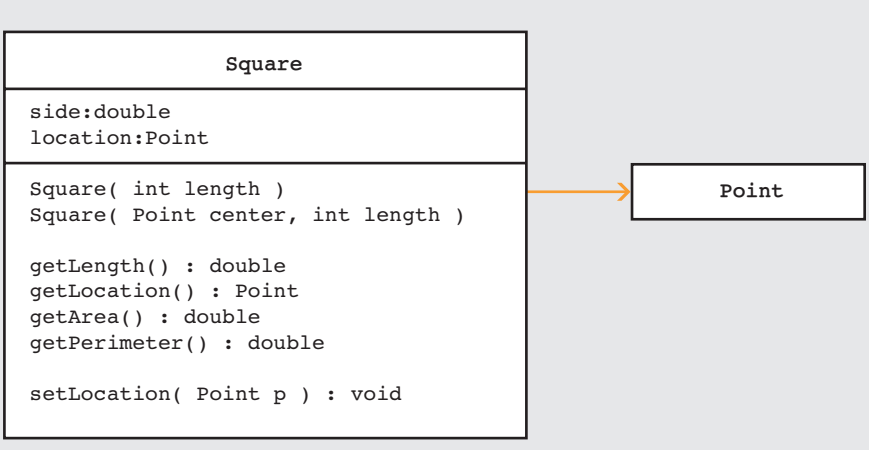
This section discussed how the third component in the definition of an object, identity, is implemented in Java. Objects are created in Java by invoking the `new` operator, which returns a reference to the newly created object and serves as the identity of the object. The next section uses an example to tie together the basic object-oriented syntax of Java.

3.2.4 Example: `Square` Class

Let's write a program that provides users with the tools necessary to draw and manipulate simple two-dimensional shapes (such as circles, squares, rectangles, and triangles). After performing an object-oriented analysis of the specifications and requirements for this program, you would determine that you would need classes to represent each of the shapes that the program can manipulate. The program would include classes that defined objects such as circles, squares, rectangles, and triangles. In this section, we design and implement a class that you can use to represent squares within the program.

Before we can write the code that implements the `Square` class, we need to determine the state and behavior of a square. To make this class as simple as possible, we only store the length of one of the sides of the square and the location of the square on the drawing surface. We use an integer value to represent the length of one of the sides of the square and an instance of the `Point` class to keep track of the location of the square's center.

Because we know very little about how a square is used in this program, defining the square's behavior is more difficult. If we were developing an actual program, we would have to review its specifications and requirements to understand the behavior a square would be required to exhibit. Because this is only an example, we simply define typical behaviors exhibited by all squares. The UML diagram in Figure 3-41 defines the state and behavior of a square object.



[FIGURE 3-41] Design of the `Square` class

The UML diagram in Figure 3-41 shows two classes: `Square` and `Point`. The arrow that connects these two indicates that a relationship exists between the classes. In other words, a `Square` knows about a `Point`. The arrow indicates that the relationship does not go in both directions, which means that `Square` knows about `Point`, but `Point` does not know about the `Square`. The relationship between these two classes is part of the structure of a `Square`; a `Square` consists of a `Point` that specifies its location. This is the reason the relationship between the `Square` and `Point` classes has been drawn as an association—the connection is permanent and makes up part of the state of a `Square`. From a programming perspective, this means that the `Square` class contains an instance variable that refers to the `Point` object giving the current location of the center of the `Square`.

The state and behavior for the `Point` class are not included in the UML diagram. Recall from Chapter 2 that when drawing a UML diagram, you only want to include as much information as the reader needs to understand the system. After reading the previous sections of this chapter, you already know about the state and behavior of a `Point` object, so there is no need to include this information.

Based on the UML diagram of Figure 3-41, you can determine that the state of a square consists of a `double` value that gives the length of one of the sides of the square and a reference to a `Point` object that gives the location of the square. The `Square` class provides two constructors and several methods that can be used to obtain information about the square and manipulate its state. The `Square` class provides two accessors, `getLength()` and `getLocation()`, that return the current values associated with the state of the square. The `Square` class also provides one mutator, `setLocation()`, that can change the location of a square object. Finally, two methods, `getArea()` and `getPerimeter()`, determine the area and perimeter of the square.

Based only on the information provided in the UML diagram, it is possible to write programs that use square objects. The program in Figure 3-42 illustrates how to instantiate a square object and then print its length, location, area, and perimeter. The Java code that implements the `Square` class is shown in Figure 3-43.

```
/**
 * A simple program to illustrate how to use the Square class
 */
public class TestSquare {
    /**
     * Create a square and then print its length, location,
     * perimeter, and area
     */
    public static void main( String args[] ) {
        Square aSquare = new Square( 10 );

        System.out.println( "Square:" );
        System.out.println( " Length == " +
                            aSquare.getLength() );
        System.out.println(
            " Location == (" +
            aSquare.getLocation().getXCoordinate() + "," +
            aSquare.getLocation().getYCoordinate() + ")" );

        System.out.println( " Perimeter == " +
                            aSquare.getPerimeter() );
        System.out.println( " Area == " +
                            aSquare.getArea() );
    }

} // TestSquare
```

[FIGURE 3-42] Using the `Square` class

```
/**
 * A class that represents a square. The state of a square
 * consists of the length of its side and a Point object.
 * The Point object specifies the location of the center of
 * the square.
 */
public class Square {
    private double side;     // Length of a side
    private Point location;  // Location of the square
```

continued

```
/**
 * Create a new square at location (0,0)
 * @param length the length of one side of the square
 */

public Square( double length ) {
    this( new Point( 0, 0 ), length );
}

/**
 * Create a square at the specified location and length
 * @param center the location of the center of the square
 * @param length the length of one side of the square
 */
public Square( Point center, double length ) {
    side = length;
    location = new Point( center );
}

/**
 * Determine the length of one side of the square
 * @return the length of one side of the square
 */
public double getLength() {
    return side;
}

/**
 * Determine the location of the center of the square
 * @return the location of the center of the square
 */
public Point getLocation() {
    return location;
}

/**
 * Determine the area of the square
 * @return the area of the square
 */
public double getArea() {
    return side * side;
}
```

continued

```
    /**
     * Determine the perimeter of the square
     * @return the perimeter of the square
     */
    public double getPerimeter() {
        return 4 * side;
    }

    /**
     * Set the location of the center of this square
     * @param p the location of the center of the square
     */
    public void setLocation( Point p ) {
        location.setXY( p );
    }
} // Square
```

[FIGURE 3-43] `Square` class

Consider for a moment how we described the example classes in the previous section. We told you to write a program that provides a user with the tools necessary to draw and manipulate simple two-dimensional shapes (such as circles, squares, rectangles, and triangles). We described the classes we were planning to write, not individually but collectively, using the word *shape*. Using your understanding of this term, we were able to convey basic information about every class we were planning to write. For example, all shapes know about their location and provide behaviors that allow a programmer to determine and set the location of a specific shape object.

Inheritance in an object-oriented programming language provides the same expressive power we used in the previous paragraph, but instead of collecting a group of related words, a programming language can group together related classes. The next section discusses how to use inheritance in a Java program.

3.3 Inheritance

Now that you have written the `Square` class for the drawing application, you might want to write the class that represents a circle. Circles are certainly different from squares, but you will find that some methods in the `Circle` class also appear in the `Square` class. In particular, anything in the `Square` class that deals with location must be included in the `Circle` class as well.

Instead, you might think about writing a `Shape` class to represent the state and behavior that are common to *all* shapes rather than duplicating the code in each one. This makes perfect sense because a circle and a square are both shapes. The class definition in Figure 3-44 contains the state and behavior common to all shapes.

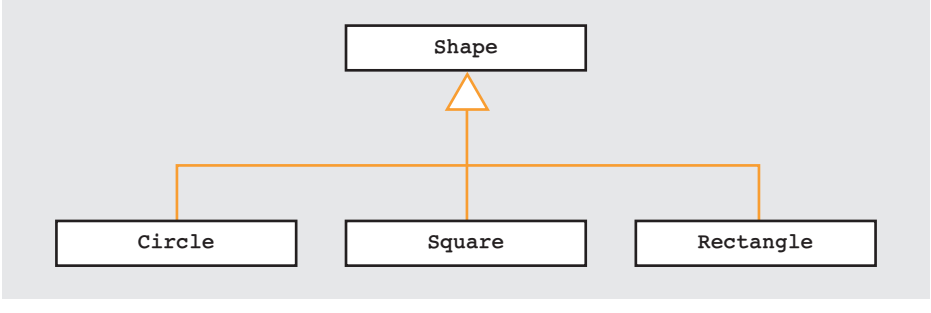
```
/**
 * The base class for shape objects
 */
public class Shape {
    private Point location;  // Location of the shape

    /**
     * Create a new shape at location (0,0)
     */
    public Shape() {
        this( new Point( 0, 0 ) );
    }

    /**
     * Create a shape at the specified location
     *
     * @param center the location of the center of the shape
     */
    public Shape( Point center ) {
        location = new Point( center );
    }

    /**
     * Determine the location of the center of the shape
     *
     * @return the location of the center of the shape
     */
    public Point getLocation() {
        return location;
    }
```

continued

```
    /**
     * Set the location of this shape to the coordinates
     * specified by the given point
     *
     * @param p the location of the shape
     */

    public void setLocation( Point p ) {
        location.setXY( p );
    }
} // Shape
```

[FIGURE 3-44] `Shape` class

Because a circle and a square are both shapes, it makes sense to make `Circle` and `Square` subclasses of the `Shape` class. In Chapter 2, we stated that two classes are related by inheritance when the state and behavior of one class are a subset of the state and behavior of another. The more general class (`Shape`) is referred to as the **superclass** of the second, more specialized class (`Circle` or `Square`). The second class is called a **subclass** of the superclass and is said to inherit the state and behavior of the superclass. The UML diagram of Figure 3-45 illustrates the inheritance relationship between the classes in our drawing program.



[FIGURE 3-45] `Shape` class inheritance hierarchy

A subclass is a more specialized form of the superclass. It satisfies the basic specifications of the superclass and can differentiate itself from its parent by adding to, or extending, the behavior of the superclass in some way. For example, in the drawing application, both the `Circle` and `Square` classes provide the state and behavior needed to manage their location, but they also add state and behavior specific to their class. For example, the `Square` class includes a variable that allows an instance of the class to determine the length of one of its sides, and it provides behaviors that allow a programmer to determine the area or perimeter of a square.

In Table 2-2, we introduced five forms of inheritance. The next section examines how you can use the specialization form of inheritance in a Java program.

3.3.1 Extending a Class

A subclass specifies its superclass class using the keyword `extends`. A subclass can extend any public class that is not declared as `final`. The general form of a subclass is shown in Figure 3-46. At most, one class name can be listed after the keyword `extends`. Java does not support inheritance from multiple superclasses. This capability is called **multiple inheritance** in other object-oriented programming languages.

```
modifier class subClassName extends superClassName {
     Class state...
     Class behavior...
}
```

[FIGURE 3-46] General form of a subclass

A subclass inherits all the `public` and `protected` states and behaviors of the class that it extends. Constructors, which are not considered members of a class, are not inherited by a subclass. Any member of a class with `protected` access is accessible not only to classes in the same package as the superclass but also to any of its subclasses, regardless of the package to which the subclass belongs. The `Square` class in Figure 3-47 illustrates how to define a class using inheritance.

```
/**
 * A class that represents a square. The state of a
 * square consists of the length of its side and a Point
 * object. The Point object specifies the location of
 * the center of the square.
 */
public class Square extends Shape {
    private double side;     // Length of a side

    /**
     * Create a new square at location (0,0)
     * @param length the length of a side of the square
     */
```

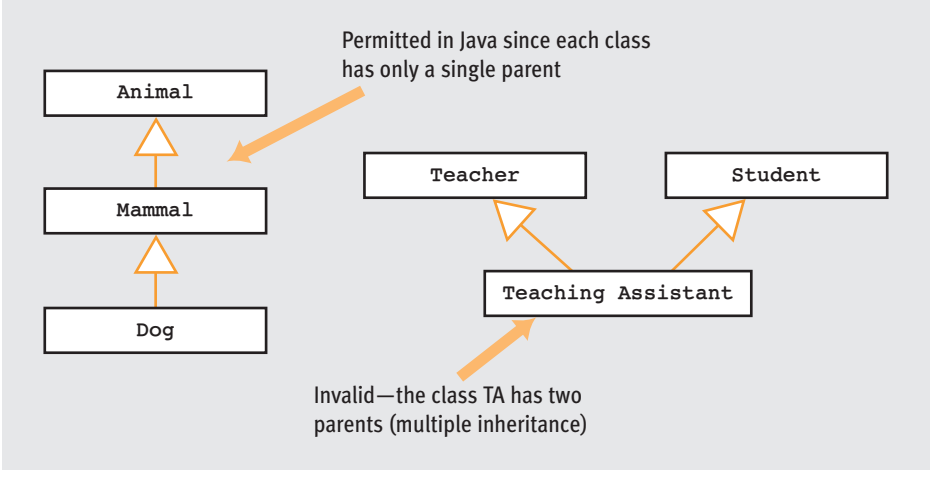
continued

```
    public Square( double length ) {
        this( new Point( 0, 0 ), length );
    }

    /**
     * Create a square at the specified location with the
     * given length
     * @param center the location of the center of the square
     * @param length the length of a side of the square
     */
    public Square( Point center, double length ) {
        super( center );
        side = length;
    }

    /**
     * Determine the length of a side of the square
     * @return the length of a side of the square
     */
    public double getLength() {
        return side;
    }

    /**
     * Determine the area of the square
     * @return the area of the square
     */
    public double getArea() {
        return side * side;
    }

    /**
     * Determine the perimeter of the square
     * @return the perimeter of the square
     */
    public double getPerimeter() {
        return 4 * side;
    }
} // Square
```

[FIGURE 3-47] `Square` subclass

You can extend a class as many times as you want. There is no limit on the number of subclasses that a single class can have. In terms of the drawing application, the `Shape` class can have as many subclasses as needed, but it is not valid for `Circle` to extend both `Shape`

and some other class. A class can inherit from superclasses many levels away. For example, if `Dog` is a subclass of `Mammal`, and `Mammal` is a subclass of `Animal`, then `Dog` inherits from both the `Mammal` and `Animal` classes. This is not an example of multiple inheritance because each class in the inheritance hierarchy has at most a single parent, as shown in the UML diagram in Figure 3-48.

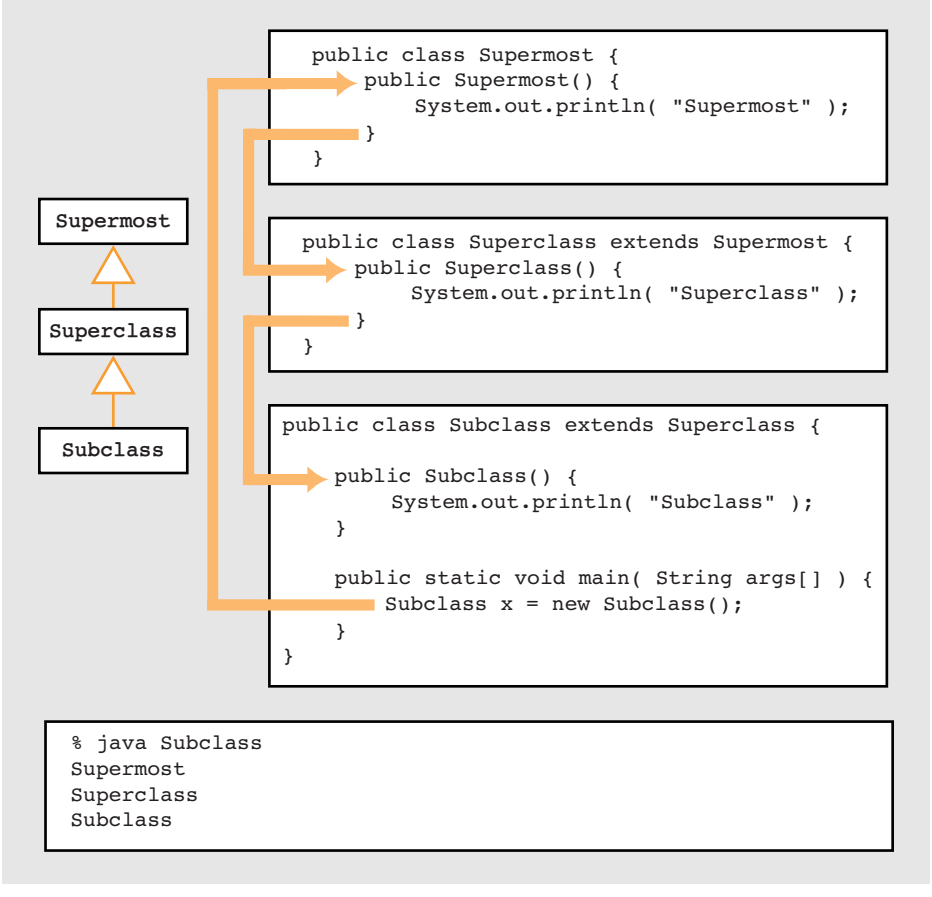


[FIGURE 3-48] Single versus multiple inheritance

When a class is extended, the accessibility rules are still enforced. This means a subclass cannot access the private members of any of its superclasses. Because the class `Square` is a subclass of `Shape`, it inherits all of the public states and behaviors of `Shape`. This means it is valid to invoke the `getLocation()` and `setLocation()` methods on an instance of the `Square` class, even though these methods are not explicitly defined in the class. Also, because `location` is a private member of the `Shape` class, it is not inherited by `Square`. Because the `getLocation()` and `setLocation()` methods are members of the `Shape` class, they are permitted to access the private instance variable `location`.

In the second constructor in the `Square` class of Figure 3-47, notice the use of the keyword `super`. It provides a reference to the superclass of a class in much the same way that the keyword `this` provides access to the object in which it is used. The keyword `super` can be used by a subclass to directly access the state or behavior of its immediate superclass, or it can be used in a constructor to invoke a superclass constructor. The keyword `super` can only be used to invoke the constructor of a superclass inside the body of a constructor of a subclass; like the keyword `this`, it must be the first statement in the body of the constructor.

Given that a superclass may have state that is inaccessible to a subclass, you may wonder how the state of the superclasses is initialized when an instance of the subclass is created. Unless the programmer has specified differently, Java attempts to invoke the default constructors of all the superclasses when a subclass is instantiated, starting with the constructor in the highest superclass in the inheritance hierarchy down to the lowest (Figure 3-49). The output generated by executing the `main()` method in `Subclass` clearly indicates the order in which the superclass constructors are invoked.

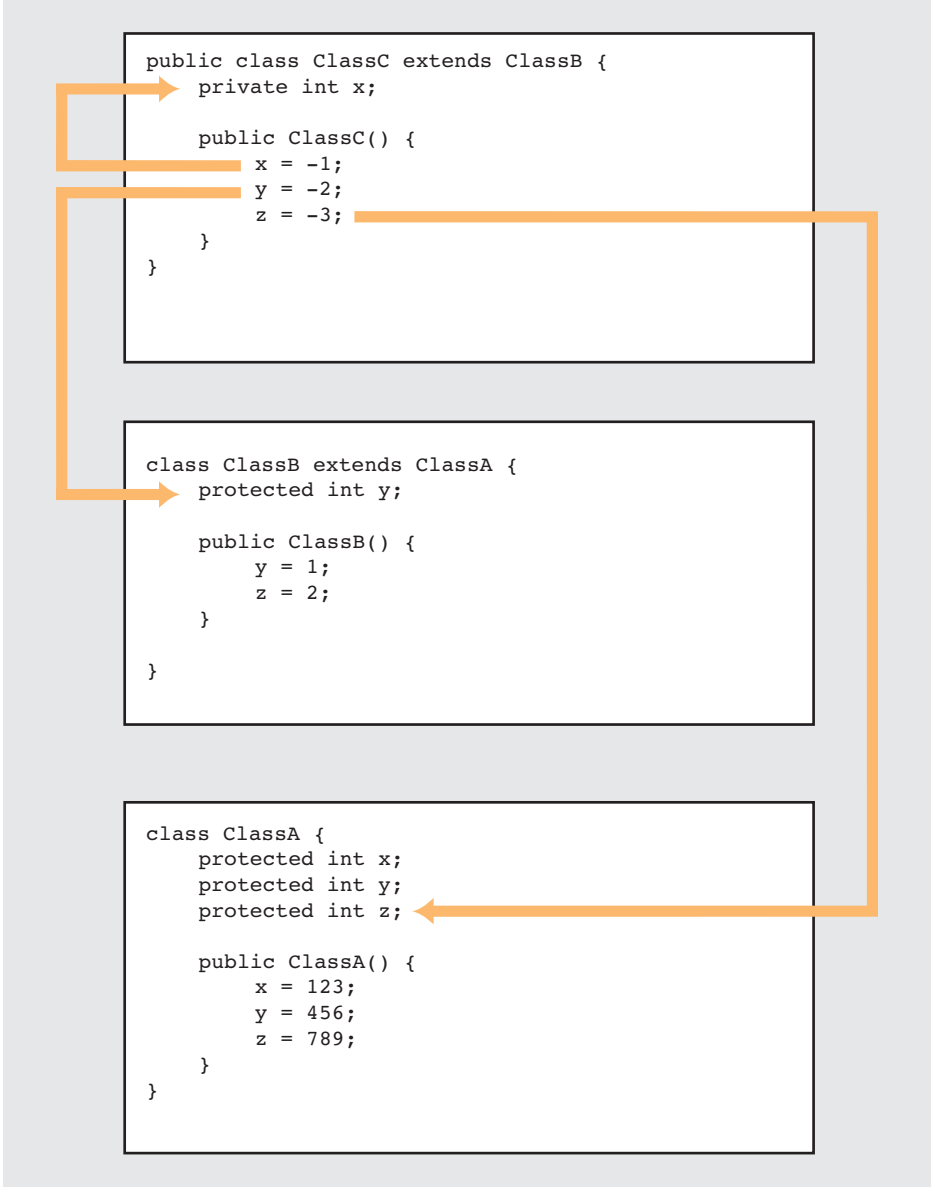


[FIGURE 3-49] Superclass constructor invocation

What happens if a superclass does not have a default constructor or if you want to use a constructor other than the default? The answer is to use `super` to specify the constructor that you want to invoke, as we did in the second constructor of the `Square` class of Figure 3-47. Because this constructor takes as a parameter the location of the square being created, the `super(Point)` constructor of the `Shape` class must be invoked to properly initialize the instance variable `location`. The use of `super` in the `Square` constructor invokes the appropriate superclass constructor because the signature matches a nondefault constructor of the `Shape` class. If you are extending a class without a default constructor, in most situations you must use `super` in the subclass constructor to indicate which superclass constructor to invoke. If you fail to do so, the program does not compile.

Inheritance increases the number of classes that need to be searched when an attempt is made to access either a variable or a method. Java searches the local scope first, then checks the class scope, and then checks the scope of each superclass in turn up to the top of the inheritance hierarchy. If a variable or method with the same name is declared in several scopes, the first one that is found is used.

Consider how Java resolves the reference to the instance variable `x` in the constructor for `ClassC` in Figure 3-50. Java starts by searching the local scope in which `x` is used. In this case, because no local variables or parameters are named `x` in the constructor, the class scope is checked next. An instance variable named `x` is defined within the class. So, the `x` in the constructor for `ClassC` resolves to the instance variable `x` defined in `ClassC`. The variable `y` in the constructor resolves to the instance variable `y` defined in `ClassB`. In this situation, `y` is not declared in either the constructor or in `ClassC`. The search then moves up the inheritance chain to `ClassB`, which contains a definition for an instance variable `y`. Similarly, the variable `z` in the constructor refers to the instance variable `z` defined in `ClassA`.



[FIGURE 3-50] Scope and inheritance

Earlier in this chapter, we discussed overloaded methods—methods with the same name but different parameters. A subclass can also **override** superclass methods. A subclass

overrides a superclass method by providing an implementation of that method with the same signature. The definition in the subclass overrides the method in the superclass. This is not the same as overloading a method—when a method is overloaded, both versions of the method are still accessible. When a method is overridden, an object outside the class can no longer access the version of the method defined in the superclass.

You can change the inheritance hierarchy we have been using in this section. Instead of having a `Shape` class, imagine writing a class named `Polygon`. The state of a polygon would consist of an array that contains the vertices of the `Polygon`. Figure 3-51 contains the code to implement a basic `Polygon` class.

```
/**
 * A very simple implementation of a polygon
 */
public class Polygon {
    private Point vertices[];  // The vertices of the polygon
    private int numPoints;     // Number of points in the array

    /**
     * Create a polygon
     * @param numVertices the number of vertices in the polygon
     */
    public Polygon( int numVertices ) {
     vertices = new Point[ numVertices ];
     numVertices = 0;
    }

    /**
     * Add a new vertex to the polygon
     * @param p a point that represents the vertex to be added
     */
    public void addVertex( Point p ) {
     // Cannot add more vertices than the array will hold
     if ( numPoints < vertices.length ) {
         vertices[ numPoints ] = p;
         numPoints = numPoints + 1;
     }
    }

    /**
     * Compute the perimeter of the polygon
     * @return the perimeter of the polygon
     */
    public double getPerimeter() {
     double retVal = 0;
```

continued

```
      // Compute the perimeter of the polygon by stepping
      // through the vertices and computing the distance
      // of each side

      for ( int i = 0; i < numPoints - 1; i++ ) {
         Point p1 = vertices[ i ];
         Point p2 = vertices[ ( i + 1 ) % numPoints ];
         retVal = retVal + p1.distanceFrom( p2 );
      }

      return retVal;
     }

} // Polygon
```

[FIGURE 3-51] `Polygon` class

Now consider writing a `Square` class that extends `Polygon`. The `Square` class could use all of the methods that it inherits from `Polygon`; however, we know that we can calculate the perimeter of the square much more efficiently than the perimeter of an arbitrary polygon. Instead of stepping through the array of vertices and summing the distances between each pair, we can return four times the length of any one side. In this situation, it would definitely make sense to override the `getPerimeter()` method that `Square` inherited from `Polygon`, as shown in Figure 3-52.

```
public class Square extends Polygon {
    /**
     * Create a square
     */
    public Square() {
     super( 4 );  // A square has four vertices
    }

    /**
     * Return the perimeter of the square
     * @return the perimeter of the square
     */
    public double getPerimeter() {
     return 4 * vertices[ 0 ].distanceFrom( vertices[ 1 ] );
    }
} // Square
```

[FIGURE 3-52] Overriding the `getPerimeter()` method in `Polygon`

The `getPerimeter()` method in the `Square` class of Figure 3-52 has the same signature as the `getPerimeter()` method in the `Polygon` class of Figure 3-51. Thus, when an attempt is made to invoke `getPerimeter()` on a square, the version of the method defined in the `Square` class is executed. Once a method has been overridden, an object can no longer access that method. Within a subclass, however, you can use the keyword `super` to access an overridden version of a method in the superclass.

Sometimes, you may not want a class to be extended or a particular method in a superclass to be overridden. Maybe it makes no sense for a class to have subclasses. Another reason might be for security. Declaring a class as `final` prevents that class from ever being extended. The keyword `final` is placed before the `class` keyword in the class declaration. If you want to allow a class to be extended, but there are methods in the class that you do not want overridden, you can declare those individual methods as `final`. When the `final` keyword is in the declaration of a method, it prevents that method from being overridden. You might want to make a method `final` if it has an implementation that should not be changed and is critical to the consistent state of the object.

This section has shown you how to use the keyword `extends` in Java to use the specialization form of inheritance. Sometimes, you want to specify that a subclass must have a particular behavior, but it is not possible to provide an implementation for that behavior in the superclass. In the next section, we discuss how abstract classes can help you specify the behavior that a subclass should exhibit.

QUANTUM COMPUTING

The standard for all computer design is the **Von Neumann architecture**, originally described in a 1945 paper by John Von Neumann. Even though computers have changed enormously since the appearance of the paper, and it would seem that a Univac I (1951) and a 3-GHz, 2-TB Dual-Core Macintosh Pro (2006) have nothing in common, their internal structures are both based on the same Von Neumann model proposed more than 60 years ago. Furthermore, the design of virtually every programming language, from FORTRAN to Java, is based on the internal characteristics of that very same architecture.

However, computer scientists today are researching totally new approaches to machine design. One of the most revolutionary and exciting is **quantum computing**, in which computers are designed using the principles of quantum mechanics, which describe the behavior of matter at the atomic and subatomic level. A quantum computer encodes information using some aspect of quantum-mechanical state, such as electron spin, superposition, or photon polarization. Unlike traditional data bits, which at any instant of time must be either a 0 or a 1 but not both, quantum theory says that a quantum bit, or **qubit**, can be either a 0 or a 1 or

continued

both a 0 and a 1 *at the same time*. In theory, a quantum computer simultaneously could do multiple computations on different numbers. In fact, with just 500 qubits of memory, each of which could be viewed as being both a 0 and a 1, we could theoretically perform 2^{500} simultaneous computations—a number larger than the total number of atoms in the universe!

Many obstacles must be overcome before quantum computers become a reality, but there is enormous interest in this area (especially from the military) because of the ability to attack tremendously large problems that are essentially unsolvable using traditional computers, even massively parallel ones. The best known of these problems are encryption and decryption, which require finding all prime factors of an *n*-digit integer. When *n* is sufficiently large, a solution may require centuries of traditional machine time. The American computer scientist Peter Shor has described a quantum computing algorithm that can factor large numbers in exponentially less time than any known method, but only if a quantum computer can be built.

Analysts debate whether a workable quantum computer will take 10 years, 25 years, or perhaps another century to design and construct. There is also great uncertainty about what a quantum programming language will look like. However, the underlying theory is sound, and a quantum computer will likely become a reality, even if we are not sure exactly when.

3.3.2 Abstract Classes

If you think about the behavior that should be included in every subclass of the `Shape` class, you would probably include the ability to determine the shape's area and perimeter. Although it is possible to compute the area and perimeter of all shapes, there is no single universal way to implement these computations. The formulas for the area of a triangle, circle, and square are all quite different. What we would really like to do is include the methods `getArea()` and `getPerimeter()` in the superclass for shapes but not provide any implementation. This way, we are guaranteed that all shapes can determine their area and perimeter, but each shape must provide its own implementation.

In an **abstract class**, one or more methods are specified, but no implementation is provided. This allows us to include the definition of methods in a superclass, but forces the implementation of those methods on the appropriate subclasses. You can view an abstract class as a placeholder for declaring shared methods for use by subclasses. Because it is missing some implementation, it is not possible to create an instance of an abstract class. An abstract class exists just so other classes can extend it. An abstract class contains one or more methods that have been defined as abstract. An **abstract method** does not have a method body, and the declaration of the method ends with a semicolon instead of a compound statement. A

class that contains one or more abstract methods must be declared as an abstract class. Also, `private`, `final`, and `static` members cannot be abstract.

The `Shape` class of Figure 3-53 has been modified to include the two abstract methods `getArea()` and `getPerimeter()`.

```
/**
 * The base class for shape objects
 */
public abstract class Shape {
    private Point location;  // Location of the shape

    // public Shape() … code omitted
    // public Shape( Point center ) … code omitted
    // public Point getLocation() … code omitted
    // public void setLocation( Point p ) … code omitted

    /**
     * Return the area of this shape
     * @return the area of this shape
     */
    public abstract double getArea();

    /**
     * Return the perimeter of this shape
     * @return the perimeter of this shape
     */
    public abstract double getPerimeter();

} // Shape
```

[FIGURE 3-53] Abstract base class for the `Shape` classes

Note the use of the keyword `abstract` in the `Shape` class of Figure 3-53. It appears in the class header, and it appears in the header of each of the two abstract methods. For an abstract class to successfully compile, it is not enough to declare individual methods as `abstract`; you must also declare the class containing them as `abstract`.

With the concept of abstract classes, we are coming close to achieving the **specification form** of inheritance first defined in Table 2-2. Recall that in the specification form of inheritance, the superclass defines the behavior that is to be implemented by a subclass, but it does not provide an implementation for that behavior. That is exactly what is done in the `getArea()` and `getPerimeter()` methods in Figure 3-53. The abstract `Shape` class is declaring that these two methods must be included in any subclass, but it does not provide an implementation. Compare that to the `getLocation()` and `setLocation()` methods. In this case, the abstract class specifies that these methods will be included, and it provides an implementation that all subclasses inherit and may use.

The syntax for writing a class definition that extends an abstract class is no different from the syntax used to extend a nonabstract class. You only need to ensure that the subclass includes the implementation of all abstract methods in the abstract class. If you forget to implement one of them, you will not be able to compile the subclass correctly.

3.3.3 Interfaces

The specification form of inheritance defines the behavior that a subclass must provide, but not the implementation of that behavior. So far, the only way we have seen to use Java's specification form of inheritance is to write an abstract class. Java provides another structure, the **interface**, which allows you to exploit the specification form of inheritance.

An interface is similar to a class in that it consists of a set of method headers and constant definitions. Unlike a class, however, an interface contains no executable code. An interface is a pure behavioral specification that does not provide any implementation.

The purpose of an interface is to specify the common behaviors that a group of classes are required to implement. In Chapter 2, we described the specification form of inheritance in terms of using a clock. In school, you were taught how to tell time in general, not how to read a Timex Model 921 travel series alarm clock. Your teacher explained that all analog clocks have hour, minute, and second hands. You were then taught how to read the hands on the dial to determine the current time.

An interface provides the same capability in a programming language. In an interface, you specify the behaviors that a class must implement to implement the interface. You do not care how this behavior is actually carried out; you only care that it must be there. An interface is similar to an abstract class that has no state, and *every* method is abstract.

In an interface, all methods are `public`, even if the `public` modifier is omitted. An interface cannot define instance variables, but it can define constants that are declared both `static` and `final`. Because an interface is a behavioral specification, you cannot instantiate it. Therefore, an interface does not contain a constructor. To illustrate how to write an interface and a class that implements it, let's rewrite the `Shape` class of Figure 3-44 as an interface. The code that defines the `Shape` interface is shown in Figure 3-54.

```
/**
 * An interface for shape objects
 */
public interface Shape {
    /**
     * Determine the location of the center of the shape
     * @return the location of the shape
     */
```

continued

```
    public Point getLocation();

    /**
     * Set the location of this shape to the coordinates
     * specified by the given point
     * @param p the location of the center of the shape
     */
    public void setLocation( Point p );

    /**
     * Return the area of this shape
     * @return the area of this shape
     */
    public double getArea();

    /**
     * Return the perimeter of this shape
     * @return the perimeter of this shape
     */
    public double getPerimeter();
} // Shape
```

[FIGURE 3-54] `Shape` interface

The `Shape` interface in Figure 3-54 should not look all that different to you. It has the same basic syntax as a class definition, but the word `interface` appears in the class declaration in place of `class`. Also, every method in the class is without an implementation. In fact, the methods look identical to the abstract methods discussed in the previous section.

An interface defines a set of behaviors a class is required to exhibit. A class that provides all the behaviors specified by the interface is said to **implement** the interface. The `Square` class of Figure 3-55 implements the `Shape` interface of Figure 3-54.

```
/**
 * A class that represents a square. The state of a
 * square consists of the length of its side and a Point
 * object. The Point object specifies the location of
 * the center of the square.
 */
public class Square implements Shape {
    private double side;     // Length of a side
    private Point location;  // Location of the square
```

continued

```
/**
 * Create a new square at location (0,0)
 * @param length the length of a side of the square
 */
public Square( double length ) {
    this( new Point( 0, 0 ), length );
}

/**
 * Create a square at the specified location with the
 * given length
 * @param center the location of the center of the square
 * @param length the length of a side of the square
 */
public Square( Point center, double length ) {
    side = length;
    location = new Point( center );
}

/**
 * Determine the length of one of the sides of the square
 * @return the length of a side of the square
 */
public double getLength() {
    return side;
}

/**
 * Determine the location of the center of the square
 * @return the location of the center of the square
 */
public Point getLocation() {
    return location;
}

/**
 * Determine the area of the square
 * @return the area of the square
 */
public double getArea() {
    return side * side;
}

/**
 * Determine the perimeter of the square
 * @return the perimeter of the square
 */
```

continued

```
    public double getPerimeter() {
        return 4 * side;
    }

    /**
     * Set the location of the center of this square to the
     * coordinates specified by the given point
     * @param p the location of the center of the square
     */
    public void setLocation( Point p ) {
        location.setXY( p );
    }
} // Square
```

[FIGURE 3-55] `Square` class that implements `Shape`

Again, the `Square` class of Figure 3-55 should not look much different from the other classes presented in this chapter. The only difference is in the header of the class, where the keyword `implements` declares that this class satisfies all the behavioral requirements of the `Shape` interface. Unlike the keyword `extends`, in which a class can only extend a single superclass, a class can implement different interfaces. A single class can also extend one class and implement one or more interfaces at the same time.

You might consider it strange that Java supports multiple interfaces but not multiple inheritance. Recall that an interface uses the specification form of inheritance and specifies only the behaviors that a class must provide. In the specialization form of inheritance, the superclass provides the implementation of the method included in the subclass. Now consider what might happen if you were to extend multiple classes. If you inherited two methods from two different superclasses that had the same signature but different implementations, which one would you use? On the other hand, if you wrote a class that implemented two different interfaces and both interfaces specify the same method, there is no problem. As long as your class provides an implementation for the common method, it satisfies both interfaces.

3.3.4 Polymorphism

Whether you are using the specialization or specification forms of inheritance, the result is the same: The subclass has the same public behavior as its superclass. It does not matter if the implementation of that behavior is the same as in the superclass or specializes the behavior of the superclass. What does matter is that we know the behavior in the superclass is present in the subclass. Because all instances of a subclass exhibit the same behavior as their superclass, they can mimic the behavior of the superclass and be indistinguishable from an instance of the superclass. Therefore, it should be possible to substitute instances of a

subclass for a superclass in any situation with no observable effect. For example, if I gave you a reference to a subclass but told you it was a reference to the superclass, you should be able to use the object without knowing exactly what type it was.

The observation in the previous paragraph should change the way you think about assignment. Previously, you were taught that the type of the expression on the right side of an assignment statement had to match the type of the variable on the left side. Consider the following statement:

```
Shape aShape = new Square( 10 );
```

Because `Square` is a subclass of `Shape` (in other words, a `Square` "is a" `Shape`), an instance of the `Square` class can be substituted for an instance of the `Shape` class with no observable effect, and the above assignment is valid.

If the types in an assignment statement do not match, Java automatically converts the type of the expression on the right side of the assignment statement to match the type of the variable on the left, whenever possible. Otherwise, a syntax error occurs. For example, you are already aware that if you attempt to assign an `int` value to a `double` variable, Java automatically promotes the `int` to a `double` and executes the assignment. Similarly, if the class of the expression on the right side of an assignment statement is a subclass of the variable's class on the left, Java automatically converts the expression's type on the right side to the variable's type on the left. The same holds true for interfaces and classes that implement the interface. The reverse, however, is not true. Assigning an object of a superclass to a subclass variable causes an error. The following statement is not valid:

```
Square aSquare = new Shape();  // This is an error
```

Although a `Square` is a `Shape` and can stand in for a `Shape`, not all `Shapes` are `Squares`.

To see how you can use this in a program, assume that the classes `Circle`, `Square`, and `Triangle` all extend the `Shape` class of Figure 3-53. Now consider the program shown in Figure 3-56.

```
public class PolyShapes {
    /**
     * Compute the total area of the shapes in the array
     * @param shapes array containing the shapes
     * @return the total area of all the shapes
     */
    public static double totalArea( Shape shapes[] ) {
        double retVal = 0;
```

continued

```
        for ( int i = 0; i < shapes.length; i++ ) {
            retVal = retVal + shapes[ i ].getArea();
        }

        return retVal;
    }

    public static void main( String args[] ) {
        Shape theShapes[] = new Shape[ 3 ];

        theShapes[ 0 ] = new Square( 10 );
        theShapes[ 1 ] = new Circle( 40 );
        theShapes[ 2 ] = new Triangle( 10, 30 );

        System.out.println( "Total area:  " +
                            totalArea( theShapes ) );
    }

} //PolyShapes
```

[FIGURE 3-56] Using polymorphism

The `totalArea()` method of Figure 3-56 takes an array of shapes as a parameter. Even though the method has no idea what specific shapes are referenced by this array, it can still compute the total area of all the shapes because each one has a `getArea()` method. For each of the shapes referenced by the area, the program invokes the same method (`getArea()`), but how the area is actually calculated depends on the specific shape. If the shape is a square, the area is calculated by squaring the length; if the shape is a circle, the radius of the circle is squared and multiplied by the constant `Math.PI`. The different effects of invoking the same method on different types of objects is an example of **polymorphism**.

The word *polymorphism* means "many forms"; in this case, different types of calculations may take place, and the correct calculation is determined dynamically at run time. Therefore, another term for polymorphism is **run-time binding**. In an object-oriented program, polymorphism occurs when the invoked method can change depending on the type of object used in the program. Java supports polymorphism through inheritance and interfaces.

3.3.5 The `Object` Class

In Java, every class is either directly or indirectly a subclass of the `Object` class. This class defines the basic state and behavior that all objects must exhibit. The methods that may be overridden by subclasses of the `Object` class (meaning the nonfinal methods of the class) are listed in Table 3-3.

METHOD	DESCRIPTION
`protected Object clone()`	Performs a deep copy operation (i.e., creates a new copy of the object)
`public boolean equals(Object obj)`	Returns true if the object is equal to the object referred to by the parameter and false otherwise
`protected void finalize()`	Invoked when there are no more references to the object and sometime before the object is removed from memory by the garbage collector
`public int hashCode()`	Returns a hash code value for the object
`public String toString()`	Returns a string representation of this object; by default, the method returns a string consisting of the class name and the hexadecimal representation of the object's hash code

[TABLE 3-3] Nonfinal methods in the `Object` class

Each method in Table 3-3 can play an important role in a Java program. The `clone()` method allows a class to create a deep copy of itself. The `finalize()` method is the closest thing Java has to a destructor. This method is guaranteed to be invoked before the object is removed from memory by the garbage collector. If you have written a class that dynamically allocates system resources, you can use the `finalize()` method to return those resources before the object is removed from memory. We will discuss the `hashCode()` method in Chapter 8.

The `toString()` method in the `Object` class is probably one of the most useful methods you will ever write. The `toString()` method returns a `String` that provides a textual representation of the object. When Java needs to convert an object to a string format, it automatically invokes the `toString()` method on the object. For example, the `toString()`method in Figure 3-57 could be added to the `Square` class to return a string that contains the length of the side, the area, and the perimeter of the square.

```
/**
 * Return a textual representation of a square
 * @return a string representation of a square
 */
public String toString() {
    return ( "Square:  length=" + side +
             " area=" + getArea() +
             " perimeter=" + getPerimeter() );
}
```

[FIGURE 3-57] A `toString()` method for the `Square` class

A class that has a `toString()` method can be "printed" using the `System.out.println()` method. The method takes a string as a parameter, but if you pass it an arbitrary object, the `toString()` method is invoked to obtain a string that can print. Given the version of the `Square` class in Figure 3-57, the program in Figure 3-58 creates three squares of various sizes and then prints them.

A well-designed `toString()` method can be extremely useful when you debug your programs. A good coding habit is to write a `toString()` method for every class you write. This allows you to obtain a customized, printable version of every object you create.

```
public class SquareToString {
    /**
     * Illustrate the toString() method in the Square class
     */
    public static void main( String args[] ) {
     Square s1 = new Square( 10 );
     Square s2 = new Square( 1 );
     Square s3 = new Square( 4 );

     System.out.println( s1 );
     System.out.println( s2 );
     System.out.println( s3 );
    }
} // SquareToString
```

[FIGURE 3-58] Using the `toString()` method in the `Square` class

You can use the `equals()` method to perform a deep comparison on a class. The parameter that is passed to the `equals()` method identifies the object of comparison. Because the `equals()` method uses an object reference, you can compare any two objects regardless of their class. The `equals()` method in Figure 3-59 could be added to the `Square` class to override the `equals()` method in the `Object` class.

```
/**
 * Compare this square to another object
 * @param o the comparison object
 * @return true if the objects are equal and false otherwise
 */
public boolean equals( Object o ) {
    boolean retVal = false;

    if ( o instanceof Square ) {
        Square other = (Square)o;
        retVal = getLength() == other.getLength();
    }

    return retVal;
}
```

[FIGURE 3-59] An `equals()` method for the `Square` class

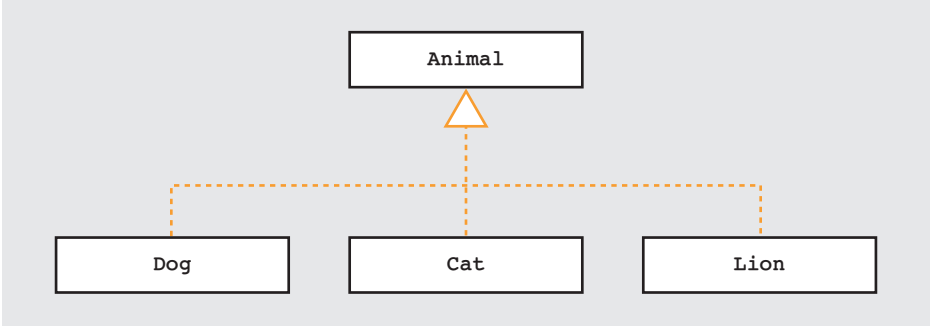
The `equals()` method in Figure 3-59 uses the `instanceof` operator, which returns true if the reference on the left side is either an instance, a subclass, or an implementation of the class specified on the right side. The `equals()` method first checks to determine if the argument refers to an instance of a `Square`. If it does, the method then compares the state of the two objects (the length of their sides) to determine whether they are equal.

3.4 Generic Types

Generic types were introduced in version 1.5 of Java. Generic types allow programmers to parameterize the types used in a class in much the same way as formal parameters are used in methods. The next section introduces the basic concepts of generic types and illustrates how to write classes that use generic types.

3.4.1 Using Generics

Imagine that you are writing a program to keep track of animals in a zoo, in which each enclosure can hold one specific type of animal. After doing an object-oriented analysis of the problem, you decide to create a class called `Enclosure`; it represents a cage object that can hold a single instance of an object that implements the `Animal` interface. The interface could be implemented by a number of classes such as `Dog`, `Cat`, `Lion`, and so on. The UML diagram in Figure 3-60 illustrates these classes and how they interact with each other.



[FIGURE 3-60] `Animal` superclass and subclasses

An implementation of the `Animal` and `Dog` classes is shown in Figure 3-61.

```
public interface Animal {
  public String speak();

  public boolean isCarnivore();
}

public class Dog implements Animal {
  private String name;

  public Dog( String n ) {
    name = n;
  }

  public String speak() {
    return "Woof Woof";
    }

  public boolean isCarnivore() {
    return true;
  }
}
```

[FIGURE 3-61] Implementation of `Animal` and `Dog` classes

The advantage of writing `Enclosure` to hold an instance of `Animal` is that you can put any type of animal you want into any one of the enclosures. One implementation of the `Enclosure` class is shown in Figure 3-62.

```
public class Enclosure {
    Animal occupant;

    public void addOccupant( Animal o ) {
     occupant = o;
    }

    public Animal getOccupant() {
     return occupant;
    }

    public Animal removeOccupant() {
     Animal retVal = occupant;
     occupant = null;
     return retVal;
    }
}
```

[FIGURE 3-62] `Enclosure` class

Putting animals into the `Enclosure` of Figure 3-62 is quite simple. For example, the following code creates two enclosures, then places a dog named Buster in one and a cat named Grumpy in the other.

```
Enclosure cage1 = new Enclosure();
Enclosure cage2 = new Enclosure();

cage1.addOccupant( new Dog( "Buster" ) );
cage2.addOccupant( new Cat( "Grumpy" ) );
```

A problem with this approach is that once you place an animal in an enclosure, you no longer know the actual type of the animal there, because `Enclosure` holds objects of type `Animal`, not of type `Dog` or `Cat`. The compiler does not understand that `cage1` is a dog cage and `cage2` is a cat cage. One consequence of this loss of information is the need to cast the objects returned by the `removeOccupant()` method. For example, to remove the dog from `cage1`, you would write code similar to the following:

```
Dog d = (Dog) cage1.removeOccupant();
```

You must cast the value returned by `removeOccupant()` to a dog. A more serious consequence of this loss of information is that you could write the following code:

```
Dog d = (Dog) cage2.removeOccupant();
```

You know that `cage2` holds a cat, so you know the preceding statement is incorrect. The compiler, on the other hand, only knows that an animal is in `cage2`. When you attempt to execute the resulting program, it generates a run-time error because the program is removing an instance of a cat from the enclosure and attempting to assign it to a variable that refers to a dog. One advantage of a **type-safe language**, such as Java, is that it forces programmers to provide type information for the items manipulated in the program. As a result, the compiler can determine if the items are being used correctly and consistently. For example, if you tried to execute the following code segment:

```
int a = 10;
String b = "1";
a = a + b;
```

The compiler would generate an error message because it knows that `a` is an integer, `b` is a string, and you cannot add an integer to a string. One way to solve this problem is to write several different versions of the `Enclosure` class—one version for each type of animal you want to put in a cage. Using the classes in Figure 3-61, this would mean writing an `Enclosure` class for dogs, another for cats, and another for lions (see Figure 3-63).

```
public class DogEnclosure {
    Dog occupant;

    public void addOccupant( Dog o ) {
      occupant = o;
    }

    public Dog getOccupant() {
      return occupant;
    }

    public Dog removeOccupant() {
      Dog retVal = occupant;
      occupant = null;
      return retVal;
    }
}
```

```
public class CatEnclosure {
    Cat occupant;

    public void addOccupant( Cat o ) {
      occupant = o;
    }

    public Cat getOccupant() {
      return occupant;
    }

    public Cat removeOccupant() {
      Cat retVal = occupant;
      occupant = null;
      return retVal;
    }
}
```

The only significant difference is the type of animal held in the enclosure

```
public class LionEnclosure {
    Lion occupant;

    public void addOccupant( Lion o ) {
      occupant = o;
    }

    public Lion getOccupant() {
      return occupant;
    }

    public Lion removeOccupant() {
      Lion retVal = occupant;
      occupant = null;
      return retVal;
    }
}
```

[FIGURE 3-63] Three different enclosure classes

The classes in Figure 3-63 are identical except for the declaration of the animal data type they hold. It would be nice if we could "parameterize" the type of animal the enclosure will hold and then specify the actual type only when we create the enclosure. This approach would be similar to method parameters that are specified only when a user invokes the method.

Java **generics** allow you to do this. Using a generic data type, you can write an `Enclosure` class so that a user must specify the type of animal that will be held in the enclosure when a new one is created. The advantage of generic types is that you can provide the compiler with additional information that lets it determine the data type of the object you are dealing with. The compiler can then monitor your class usage, in much the same way it monitors your usage of variables, to determine if you are using them consistently.

Figure 3-64 shows an implementation of the `Enclosure` class that uses generics. The main syntactic change is the use of angle brackets, (< >), in the class header to specify the generic type parameter. This example uses the identifier `T` to represent the type of animal in the enclosure. The `T` appears in the class definition everywhere we would expect to see an `Animal` data type (indicated by circles in Figure 3-63). Using `T` as a formal type parameter is really no different from using `o` as a formal parameter to the `addOccupant()` method of the `Enclosure` class. `T` can represent any class, any interface, or another type variable; the only thing `T` cannot represent is a primitive type. Interfaces can be parameterized in the same way.

```
public class Enclosure<T> {
    private T occupant;

    public void addOccupant( T o ) {
     occupant = o;
    }

    public T getOccupant() {
     return occupant;
    }

    public T removeOccupant() {
     T retVal = occupant;
     occupant = null;
     return retVal;
    }
}
```

[FIGURE 3-64] `Enclosure` class

Using a parameterized class like the one in Figure 3-64 is straightforward. When you create an instance of a parameterized class, you specify the actual type that you want substituted for the parameterized data type—the name `T` in Figure 3-64. The following code fragment illustrates this point:

```
Enclosure<Dog> cage1 = new Enclosure<Dog>();
Enclosure<Cat> cage2 = new Enclosure<Cat>();

cage1.addOccupant( new Dog( "Buster" ) );
cage2.addOccupant( new Cat( "Grumpy" ) );
```

The preceding statements indicate that `cage1` is an instance of the `Enclosure` class, in which every instance of the parameter `T` is replaced by the data type `Dog`. Similarly, `cage2` is an instance of the `Enclosure` class in which every instance of `T` is replaced by the type name `Cat`. Because the compiler now knows that `cage1` refers to an `Enclosure` that holds a dog, it is no longer necessary to cast the reference returned by `removeOccupant()`:

```
Dog d = cage1.removeOccupant(); // d must be a dog
```

More importantly, the compiler has the information required to determine that the following statement is not valid.

```
Dog d = cage2.removeOccupant();   // This is incorrect
```

The compiler knows that `cage2` holds an instance of `Cat` and can detect that the code is attempting to assign a reference to an instance of `Cat` to a variable that refers to an instance of `Dog`. This error can now be detected at compile time, alerting the programmer before running the program.

One problem with the `Enclosure` class of Figure 3-64 is that you can create an enclosure that will hold *any* type of object, even one that makes no sense. For example, it would be possible to create an enclosure to hold integers:

```
Enclosure<Integer> cage = new Enclosure<Integer>();
```

If `Enclosure` was intended to hold animals, then we might want to restrict the types that may be used as parameters to those that either implement or extend the `Animal` class. This is possible in Java using a **bounded type parameter**. To declare one, you specify the data type parameter as before, but it is followed by the keyword `extends`, as shown in Figure 3-65.

```
public class Enclosure<T extends Animal> {
    T occupant;

    public void addOccupant( T o ) {
     occupant = o;
    }

    public T getOccupant() {
     return occupant;
    }

    public T removeOccupant() {
     T retVal = occupant;
     occupant = null;
     return retVal;
    }
}
```

[FIGURE 3-65] Using a bounded type in the `Enclosure` class

Now you cannot create an `Enclosure` using a type that does not implement the `Animal` interface. You can use either an interface or a class as a bound for a type parameter, but you still use the keyword `extends` in the class definition. In this context, `extends` means to either extend a class or implement an interface.

You can also declare type parameters within method headers to create generic methods. This is the same as defining a generic type for a class, except that the scope of the parameter is limited to the method in which it is used. Consider the `fill()` method defined in Figure 3-66.

```
public static <U extends Animal> void fill( U[] pets,
                                            Enclosure<U>[] cages ) {
        for ( int i = 0; i < pets.length; i++ ) {
            cages[ i ] = new Enclosure<U>( pets[ i ] );
        }
}
```

[FIGURE 3-66] A generic method

The `fill()` method takes as parameters an array of animals and an array of enclosures capable of holding those animals. When invoked, the method creates an enclosure and then places the corresponding animal in it. To use the `fill()` method, you need to define an array of cages. You might be tempted to do this using the following code:

```
Enclosure<Dog> c[] = new Enclosure<Dog>[ 5 ];
```

Here you are trying to create a "generic" array of enclosures, each of which is capable of holding a dog. Unfortunately, this code does not compile because arrays require special consideration when working with generic types. Consider the following code segment:

```
Dog myDogs[] = new Dog[ 10 ];
Object objArray[] = myDogs;
objArray[ 0 ] = "Fido";  // Type error; exception thrown
```

This code compiles, but when you attempt to execute it, an exception is thrown because you are trying to store the string `"Fido"` in an array that holds dogs. For the Java virtual machine to detect this error, it must perform a check at run time to see if the type of the object you are attempting to place in an array conforms to the type of that array. The Java virtual machine must know both the type of the array and the type of the object being placed into the array at run time.

The way generics are implemented in Java, the value of a parameterized type is only known at compile time; the information is not provided to the Java virtual machine at run time. Although this approach provides the compiler with the information it needs to ensure that your code is type safe, it means that the Java virtual machine does not know the actual type of a parameterized type at run time. Consider the following code:

```
Enclosure<Dog> dogCages[] = new Enclosure<Dog>[ 5 ];

Enclosure<Cat> aCatCage = new Enclosure<Cat>();
aCatCage.addOccupant( new Cat( "Grumpy" ) );

Object objArray[] = dogCages;
objArray[ 0 ] = aCatCage;  // No exception generated

// ERROR!!! dogCages[ 0 ] is now an Enclosure<Cat>
Enclosure<Dog> d = dogCages[ 0 ];
```

When this code is executed, the Java virtual machine does not have the information required to know that assigning `aCatCage` to `objArray[ 0 ]` is not type safe. At run time, `objArray` and `aCatCage` are known to be arrays that hold references to instances of `Enclosures`, but the type of animals held in the enclosures is not known. Based on what is known at run time, it appears that the left and right sides of the assignment statement are the same, and the assignment statement is allowed to execute. We now have a problem because the array referred to by `objArray` is the same array referred to by `dogCages`, and the element at position 0 in that array is now of type `Enclosure<Cat>`. The next statement generates an exception because we are attempting to assign an enclosure that holds a cat to a variable that refers to an enclosure that holds a dog.

One of the primary goals of adding generics to Java was to achieve type safety. The discussion in the previous paragraph illustrates that, given how generics are implemented in Java, allowing a programmer to create generic arrays can result in code that is not type safe. Therefore, Java does not support the creation of generic arrays. However, you can write code that uses generic arrays, such as the `fill()` method in Figure 3-66.

So, how do we use the `fill()` method if we cannot create generic arrays? There are a couple of ways to get around this restriction; our text uses the following technique:

```
Enclosure<Dog> dogCages[] = (Enclosure<Dog>[])new Object[ 5 ];
```

Now that we can create arrays capable of holding references to parameterized types, we can write the code to invoke the `fill()` method. The following code creates an array of enclosures capable of holding dogs and an array that holds dogs. It then invokes the `fill()` method to place the dogs in the cages:

```
Enclosure<Dog> dogCages[] = (Enclosure<Dog>[])new Object[ 3 ];
Dog myDogs = new Dog[ 3 ];

myDogs[ 0 ] = new Dog( "Buster" );
myDogs[ 1 ] = new Dog( "Spencer" );
myDogs[ 2 ] = new Dog( "Roxy" );

Enclosure.<Dog>fill( myDogs, dogCages );
```

Because `fill()` is a generic method, as is the case when you create an instance of a generic class, you must specify the type to use when the method is invoked. Here, all occurrences of `U` in the definition of `fill()` are replaced by `Dog`.

You can infer the value of the type parameter when the method is invoked. Here, because you are passing in an array of dogs and an array of enclosures that hold dogs as parameters, the compiler can infer that `U` should be replaced with `Dog`. It is possible to invoke the method without specifying the type parameter, as shown:

```
Enclosure.fill( myDogs, dogCages );
```

3.4.2 Generic Types and Inheritance

Earlier in this chapter, we saw the power of inheritance. Continuing with the zoo example in this section, you might be tempted to write an `isDangerous()` method that accepts an enclosure as a parameter that holds an animal and returns true if the animal is dangerous.

For the purpose of this example, we assume that carnivorous animals are dangerous. You might be tempted to write the header for this method as follows:

```
public boolean isDangerous( Enclosure<Animal> cage ) {}
```

You might think that because specific animals (dogs, cats, and lions) all implement the animal interface, an enclosure capable of holding any type of animal would be a reasonable candidate for a superclass.

Unfortunately, this approach does not work, for the same reason you cannot create generic arrays in Java. Because the value of the parameterized type is not known when the program is executed, the run-time environment only sees an enclosure, regardless of whether it holds a dog, cat, or a lion. To help deal with this problem in Java, you can use a wildcard to specify a type. For example, the following method header:

```
public boolean isDangerous( Enclosure<?> cage ) {}
```

indicates that `isDangerous()` requires as a parameter a reference to an enclosure that holds something. This tells the compiler that you must pass a parameterized class to the method, but the method cannot determine the value of the type parameter for the class. You can place a bound on the wildcard to require a parameter that refers to an enclosure containing some sort of animal, as shown in the following code:

```
public boolean isDangerous( Enclosure<? extends Animal> cage ) {}
```

This approach has an impact on the use of casts and the `instanceof` operator when working with generic classes. For example, it typically makes no sense to ask if an instance is of a particular generic type:

```
if ( x instanceof Enclosure<Dog>) …
```

At run time, all enclosures, regardless of the type of animal they hold, belong to the same class.

3.5 Compiling and Running a Java Program

Any program, regardless of language or the sophistication of its algorithms, must ultimately be executed. Processors are devices that understand a finite set of relatively simple instructions. Any program written in a high-level programming language must first be converted into an equivalent sequence of machine-language instructions to be run on a computer.

This conversion can occur either before the program is presented to the computer or as the program is running. A **compiled language** is converted to machine language before the program is run, using a translator called a **compiler**. A compiler is a piece of software that checks the syntactical and semantic structure of a program. If the compiler determines that the program is valid, it generates the machine-language equivalent of the high-level language program for a specific processor. If the program will be run on a different processor, the program must be recompiled and converted into machine-language instructions that are understood by the new processor. Although the program itself may be run several times, the compilation process only has to occur once. C and C++ are examples of compiled languages.

An **interpreted language**, on the other hand, does not convert the program to machine language until the program is executed. An interpreted program is never actually converted into machine-language form. Instead, a piece of software called an **interpreter** is responsible for analyzing the program and executing the appropriate actions. If part of the program is executed repeatedly, it is reinterpreted each time by the interpreter. Although the program is never really converted into machine-language form, the interpreter must be expressed in machine-language form to run on a specific processor.

To appreciate the differences between compiled and interpreted languages, imagine that you want to eat at a restaurant where the employees speak only German. If you do not speak German, you could use two basic strategies to order food. The first approach would be to determine exactly what you wanted to order before you went to the restaurant, and then ask someone who speaks German to translate your order. You could then go to the restaurant with your translation in hand. Of course, you could only order the food in your translation, and you could not answer any questions about your order. Furthermore, if you instead decided to eat at a Chinese restaurant, you would need to start the process all over.

A second approach would be to find someone who speaks both English and German and invite them to join you for dinner. Your friend could then serve as an interpreter. If you were asked a question about your order, your friend could help you answer. You could also eat at a Chinese restaurant if you could find a Chinese speaker to go with you. The biggest disadvantage to this approach is that all the translation would increase the amount of time it took to place your order.

The approach of translating your order before going to the restaurant is equivalent to using a compiler. Instead of translating from one natural language to another, the compiler converts the program from the high-level language to machine language. After conversion,

the program can be presented to the processor to be run. The clear disadvantage to this approach is that if you need to run the program on a different processor, you must recompile your program. The advantage to the compiled approach is speed—in a compiled language, the translation to machine language is done once, before the program is ever run.

Bringing a friend to act as a translator is similar to how an interpreted program is run. The translation is done as the order is placed, or as the program is run. The advantage to this approach is flexibility. You can order food in English at restaurants where English is not spoken, provided that each restaurant has translators. However, this approach is slower than the compiled approach. The translation occurs as you place the order; whenever you order the same thing, the translation process must take place all over again.

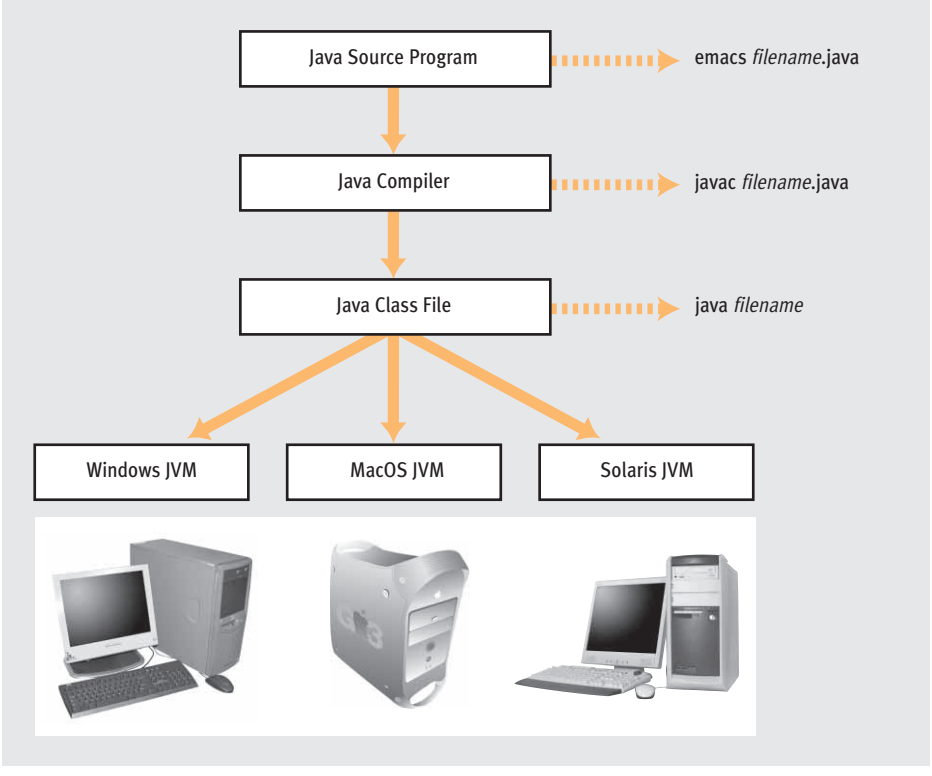
The next section examines how Java uses both compiled and interpreted technologies to prepare and run programs on a variety of platforms.

3.5.1 Compilation and Execution

After reading the previous section, you might be under the impression that languages are either compiled or interpreted. However, Java uses both interpreter and compiler technology to execute a program. Java is different from most programming languages because its Java compiler does not produce code for an actual processor; instead, it produces code, called **bytecode**, for a hypothetical computer called the **Java Virtual Machine** (JVM). The JVM is the cornerstone of the Java system, and the reason Java programs can run on any platform or operating system. The JVM implements an abstract computer; like a real computer, it has an instruction set and manipulates various memory areas at run time.

The power of using JVM to execute programs is that they only need to be compiled into bytecode. Any valid JVM can interpret bytecode and run the corresponding Java program. Thus, as long as you have a JVM that runs on your platform, you can run any compiled Java program on that platform as well.

Even though a Java program is interpreted by the JVM, you first need to compile your programs to JVM bytecode before they can run. If your program is syntactically correct, the compiler places the bytecode to be interpreted by the JVM into a **class file**. The JVM then interprets the bytecode in the class file to execute your program (see Figure 3-67).



[FIGURE 3-67] Compiling and running a Java program

The bytecode associated with each class in a Java program is placed in a separate class file. For the JVM to run your program, it must be able to locate the class files containing the bytecode for the classes it needs to use. This is one reason you must place top-level classes into files that have the same name. For example, if your program requires a `Square` class, the JVM knows that the file containing the bytecode version of this class must be in a file named `Square.class`.

This naming convention does not necessarily provide enough information for JVM to locate the desired file. For example, what happens if the file that contains the bytecode is stored in a different directory? Every JVM uses an environment variable named `CLASSPATH` to help it search directories and locate the class files required to run a program. Details for setting the `CLASSPATH` variable and the directories to search depend on the operating system on which JVM runs. Ask your instructor if you need more details.

Now that you understand why you need both a Java compiler and a JVM to run your programs, you should know that these required tools are available for free from Sun Microsystems. The Java tools you need are bundled into a package called the Java

Development Kit (JDK), which includes the compiler, JVM, debugger, class libraries, and a number of demonstration programs. A variety of JDKs are available for different types of computing platforms. If you download and install the standard JDK, you can do all the exercises in this text. You can obtain a copy of the standard JDK at the Java home page (*www.java.sun.com*).

3.6 Summary

This has been a long and detailed chapter, and it should keep you occupied for a while. It is also one of this text's most important chapters, so it is crucial that you understand and master its concepts and techniques. Object-oriented programming is an enormously powerful software development technique, but only if you implement its ideas in a high-level language such as Java.

Chapter 1 overviewed the entire software development life cycle, which included the specification, design, and implementation phases. Chapter 2 discussed the fundamental concepts of object-oriented design in a high-level, language-independent manner. Finally, this chapter presented an extensive catalog of how one particular language, Java, supports the object-oriented principles presented in Chapter 2. These concepts include classes, objects, inheritance, and polymorphism. However, the best way to appreciate the benefits of object-oriented design is to follow the solution of a single, interesting problem from start to finish. This solution includes:

- The development of the requirements and specifications documents in some formal notation such as UML
- The design of classes to solve the problem, including their state, behavior, and identity
- Their translation into correct, elegant, and easy-to-read Java code.

This is exactly what we will do in Chapter 4. It presents a detailed case study that ties together much of the material from the first three chapters. The problem we will address is one we frequently referred to at the beginning of the text: the design and development of a home heating system.

EXERCISES

1. Explain why virtually all Java classes are declared to be `public`, while instance variables are declared as either `protected` or `private`. What would be the effect of making an instance variable `public`?

2. Briefly explain why it is important to make mutable data members `private`. What consequences does this have, and how do we commonly get around them?

3. Explain the difference in how the compiler handles the following variable declarations if they appear in the `Point` class:

```
private int zCoordinate;
private static int zCoordinate;
private final int zCoordinate;
private static final int zCoordinate;
```

4. Create a class constant for the value π (3.14159) and give it `public` scope in the `Point` class. What would be the effect of giving π private scope?

5. Is there any reason to make an instance method `private`? If so, give an example of a possible `private` method that might be useful in the `Point` class.

6. Change the code in Figure 3-12 so that it handles a three-dimensional point. What are the signatures of the methods in the class?

7. Write a class method for the `Point` class that, given a three-dimensional (x, y, z) coordinate, computes the distance of that point from the origin (0, 0, 0). The formula to compute the distance is `Math.sqrt`($x^2 + y^2 + z^2$). Rewrite the method to be an instance method instead of a class method. How does this change the signature of the method?

8. Write an instance method `flip()` for the `Point` class that, given a three-dimensional (x, y, z) coordinate, returns the point that has the same (x, y) position but is on the opposite side of the z-axis—that is, the point (x, y, –z).

9. Write a main program that uses your new three-dimensional `Point` class and does the following:
   - Creates three different three-dimensional points
   - Computes and prints the distance of each of the points from the origin
   - Prints the "flip" of each point
   - Writes a copy constructor for the three-dimensional `Point` class

10 Study the following code:

```
public static int q10( int a, float b ) {
    int d;
    float e;

    a = (int)b;
    d = (int)b;
    b = 2 * b;
    e = b;

    return d;
}
```

If this method is invoked as follows:

```
int x = 5;
float y = 6.0;
int z = q10( x, y );
```

what are the values of `x`, `y`, and `z` when these three lines of code are executed?

11 Given the method declaration for `q10` shown in Exercise 10, is the following code valid? Explain why or why not.

```
this.q10( x, y );
```

12 Write a subclass of the `Polygon` class in Figure 3-51 that can handle triangle objects. Your class should create a triangle, determine the perimeter, determine the area, and print the vertices.

13 Write an interface for polygon objects. Decide what behaviors you want to include.

14 Write a class called `Nagon` that models a regular polygon with n sides. `Nagon` should extend `Shape` and provide a two-parameter constructor `Nagon(int numSides, int sideLength)`. This figure has a perimeter of `numSides * sideLength` and an area of (`numSides * sideLength`^{2}`) / (4*tan(`π`/numSides)`). Java provides π and the tangent function in the `Math` class. Be sure to provide a `toString()` method, which returns the string "3-agon" for a `Nagon` with three sides, the string "4-agon" for one with four sides, and so on.

15 Write a class that models the basic functionality of a duck. Ducks quack, eat, and fly. They have names, weights, and a best friend (also a duck). Be sure to provide accessors, mutators, and a `toString()` method for your `Duck` class. Draw a UML class diagram that describes the class.

16 Give an example in which two `Duck` objects satisfy (a) deep equality but not shallow equality, (b) shallow equality but not deep equality, (c) both shallow and deep equality, (d) neither shallow nor deep equality.

17 Write a subclass of `Duck` called `ProgrammerDuck`, which has a `String` variable containing the duck's favorite programming language, suitable accessors and mutators, and an `evangelize()`method that prints a message proclaiming the duck's favorite programming language to be the best language ever.

18 Write a method called `argue(ProgrammerDuck otherDuck)` that asks the other `ProgrammerDuck` to name its favorite language. If the other duck likes a different language, the `ProgrammerDuck` should quack at the other duck repeatedly.

19 Write a simple program that prints its command-line arguments in "middle-out" order. For example, given the arguments 1 2 3 4 5, your program should print 3 2 4 1 5. If an even number of arguments are provided, start with the element to the right of the middle. Thus, if given arguments 1 2 3 4 5 6, your program should print 4 3 5 2 6 1.

20 Given the following abstract class:

```
abstract class Room {
   protected int size;
   protected int numberOfDoors;

   public Room ( int theSize, int theDoors ) {
        size = theSize;
        doors = theDoors;
   }

   public abstract String report();
   public int getSize() { return size; }
}
```

write the complete class `BedRoom` that inherits from the `Room` class. A `BedRoom` constructor accepts three arguments: the size, the number of doors, and the number of beds. Implement the `report()` method, which returns a string containing the number of beds in the room: "The bedroom has *x* beds." The *x* represents the number of beds in the room.

21 Given the following interface:

```
public interface Publication{
   public int getNumberOfPages();
   public boolean isPeriodical();
   public boolean isBook();
}
```

write a class `TextBook` that implements the `Publication` interface. Your class should include a constructor that takes a single parameter: the number of pages in the book.

22 An interface defines a collection of methods that any implementing class must define. This sounds like an abstract class in which all the methods are abstract and no instance variables are inherited. What is the advantage of an interface?

23 Java allows a variable of type `J` to be assigned an instance of class `C`—that is, `J = C`—with certain conditions. What are those conditions?

24 What is the difference between overriding a method and overloading a method in a class?

25 Consider the following code:

```
 1|      public class AnyClass {
 2|
 3|           public String toString() {
 4|                 return "AnyClass object";
 5|           }
 6|
 7|           public String whatAmI() {
 8|                 return "AnyClass object";
 9|           }
10|
11|           public static void main( String[] args ) {
12|
13|                 AnyClass anyObj = new AnyClass();
14|                 List v = new ArrayList();
15|
16|                 v.add( anyObj );
17|                 System.out.println( v.get(0).toString() );
18|                 System.out.println( v.get(0).whatAmI() );
19|           }
20|       }
```

Use the numbers on the left as line number references. If line 17 is used, the program compiles and prints the text `AnyClass object`. If line 18 is used, it generates a compiler error. Why does the compiler handle these two statements differently?

CHALLENGE WORK EXERCISES

1. Java provides support for nested classes. Basically, they allow you to define a class inside of another class, inside a method, or as you create an instance of the class. Go to the home page for the Java tutorials (*http://java.sun.com/docs/books/tutorial/index.html*). Find the discussion that discusses nested classes. After reading about them, develop a set of examples that illustrates each of the nested classes and how they can be used.

2. C# (pronounced *C-sharp*) is a programming language similar to Java that was developed by Microsoft and is part of their .NET framework. Compare the features of Java and C#. Identify how the languages are similar and how they are different. Are there plans to include any C# features in the next release of Java?

3. Java is not the only programming language that provides support for generic types. For example, C++ has provided generics in the form of templates for several years. Research how generics are implemented in C++. How are generics in C++ and Java similar? What is a fundamental difference between the two implementations?

4. The first challenge work exercise asked you to learn more about nested classes. One form of a nested class, called an anonymous class, is sometimes referred to as a "poor man's closure" in Java. What is a closure, and how is it useful in a programming language? Find at least two languages that support closures and give examples of how they can be used. How does a "real" closure compare to an anonymous class in Java? What are the fundamental differences, and how do they affect what you can and cannot do in Java?

5. Find a copy of the Java language specification on the Web. Use the specification to learn three language features that were not discussed in this chapter. (We are asking for language features, not about classes provided by the API.) What functionality do these features provide? Develop a simple program that illustrates how each function works.