

Part II: Object-Oriented Programming

In Part I, "Fundamentals of Programming," you learned how to write simple Java applications using primitive data types, control statements, methods, and arrays, all of which are features commonly available in procedural programming languages. Java, however, is a class-centric object-oriented programming language that uses abstraction, encapsulation, inheritance, and polymorphism to provide great flexibility, modularity, and reusability for developing software. In this part of the book you will learn how to define, extend, and work with classes and their objects.

Chapter 6 Programming with Objects and Classes

Chapter 7 Strings

Chapter 8 Class Inheritance and Interfaces

Chapter 9 Object-Oriented Software Development

Objects and Classes

Objectives

- Understand objects and classes and the relationship between them.
- Learn how to define a class and how to create an object of the class.
- Comprehend the roles of constructors.
- Know object references and how to pass objects to methods.
- Understand instance variables and methods.
- Understand static variables, constants, and methods.
- Use objects as array elements.
- Use UML graphical notations to describe classes and objects.
- Understand the scope of variables in the context of a class.
- Become familiar with the organization of the Java API.

Introduction

Programming in procedural languages like C, Pascal, BASIC, Ada, and COBOL involves choosing data structures, designing algorithms, and translating algorithms into code. An object-oriented language like Java combines the power of procedural languages with an added dimension that provides such benefits as abstraction, encapsulation, reusability, and inheritance.

In procedural programming, data and operations on the data are separate, and this methodology requires sending data to procedures and functions. Object-oriented programming places data and the operations that pertain to them within a single entity called *object*; this approach solves many of the problems inherent in procedural programming. The object-oriented programming approach organizes programs in a way that mirrors the real world, in which all objects are associated with both attributes and activities. Programming in Java involves thinking in terms of objects; a Java program can be viewed as a collection of cooperating objects.

This chapter introduces the fundamentals of object-oriented programming: declaring classes, creating objects, manipulating objects, and making objects work together.

Objects and Classes

Object-oriented programming (OOP) involves programming using objects. An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, and even a mortgage loan can all be viewed as objects. An object has a unique identity, a state and behaviors. The state of an object consists of a set of fields, or fields with their current values. The behavior of an object is defined by a set of methods. Figure 6.1 shows a diagram of an object with its data fields and methods.

*****Insert Figure 6.1 (Same as Figure 5.1 in the 3rd Edition, p139)**

Figure 6.1

An object contains both state and behavior. The state defines the object, and the behavior defines what the object does.

A Circle object, for example, has a data field radius, which is the property that characterizes a circle. One behavior of a circle is that its area can be computed. A Circle object is shown in Figure 6.2.

***Insert Figure 6.2 (Same as Figure 5.2 in the 3rd Edition, p139)

Figure 6.2

A Circle object contains the radius data field and the findArea method.

Classes are constructs that define objects of same type. In a Java class, data are used to describe properties, and methods are used to define behaviors. A class for an object contains a collection of method and data definitions. Here is an example of the class for a circle:

```
class Circle
{
    /**The radius of this circle*/
    double radius = 1.0;

    /**Return the area of this circle*/
    double findArea()
    {
        return radius*radius*3.14159;
    }
}
```

This class is different from all of the other classes you have seen thus far. The Circle class does not have a main method. Therefore, you cannot run this class; it is merely a definition used to declare and create Circle objects. For convenience, the class that contains the main method will be referred to as the *main class* in this book.

A class is a blueprint that defines what an object's data and methods will be. An object is an instance of a class. You can create many instances of a class (see Figure 6.3). Creating an instance is referred to as *instantiation*. The terms *object* and *instance* are often interchangeable. The relationship between classes and objects is analogous to the relationship between apple pie recipes and apple pies. You can make as many apple pies as you want from a single recipe.

***Insert Figure 6.3

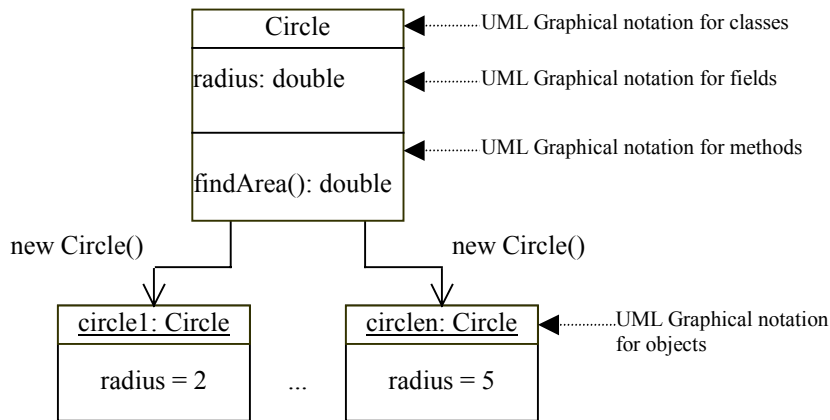


Figure 6.3

A class can have many different objects.

NOTE: Figure 6.3 uses the graphical notations adopted in the Unified Modeling Language (UML) to illustrate classes and objects. UML has become the standard for object-oriented modeling. For more information on UML, see www.rational.com/uml. For a summary of the graphical notations used in this book, see Appendix G, "UML Graphical Notations."

Creating Objects and Object Reference Variables

Objects are created using the new operator as follows:

```
new ClassName();
```

For example, new Circle() creates an object of the Circle class. Newly created objects are allocated in the memory. Objects are accessed via *object reference variables*, which contain references to the objects. Such variables are declared using the following syntax:

```
ClassName objectReference;
```

The types of reference variables are known as *reference types*. The following statement declares the variable myCircle to be of the Circle type:

```
Circle myCircle;
```

The variable myCircle can reference a Circle object. The following statement creates an object and assigns its reference to myCircle.

```
myCircle = new Circle();
```

You can combine the declaration of an object reference variable, creation of an object, and assigning object reference to the variable together in one statement using the following syntax:

```
ClassName objectReference = new ClassName();
```

Below is such an example:

```
Circle myCircle = new Circle();
```

The variable myCircle holds a reference to a Circle object.

An object reference variable that appears to hold an object actually contains a reference to that object. Strictly speaking, an object reference variable and an object are different, but the distinction between them can be ignored for most of the time. So, it is fine to say that myCircle is a Circle object, for simplicity; instead using a long phrase that myCircle is a variable that contains a reference to a Circle object. When the distinction makes a subtle difference, the long phrase should be used.

NOTE: Arrays are treated as objects in Java. Arrays are created using the new operator. An array variable is actually a variable that contains a reference to an array.

Differences between Variables of Primitive Types and Reference Types

Every variable represents a memory location that holds a value. For a variable of a primitive type, the value is of the primitive type. For a variable of a reference type, the value is a reference to where an object is located. For example, as shown in Figure 6.4, the value of int variable i is int value 1, and the value of Circle object c holds the reference to where the contents of the Circle object are stored in the memory.

*****Insert Figure 6.4**

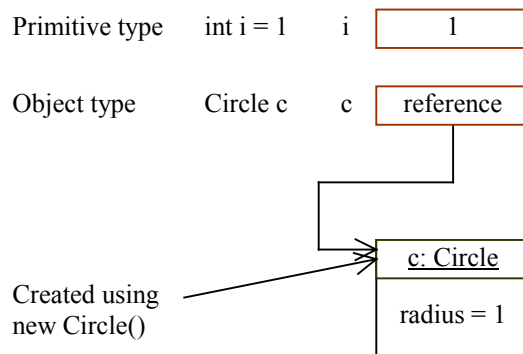


Figure 6.4

A variable of a primitive type holds the value of the primitive type, and a variable of a reference type holds the reference to where an object is stored in the memory.

When a variable of a reference type is declared, the variable holds a special Java value, null, which means that the variable does not reference any object. Once an object is created, its reference can be assigned to a variable. For example, the statement

```
c = new Circle();
```

creates a Circle object by allocating the memory space for the object and assigns its memory reference to the variable c.

When you assign one variable into another, the other variable is set to the same value. For a variable of a primitive type, the real value of one variable is assigned to the other variable. For a variable of a reference type, the reference of one variable is assigned to the other variable. As shown in Figure 6.5, the assignment statement i = j copies the contents of j into i for primitive variables, and the assignment statement c1 = c2 copies the reference of c2 into c1 for object variables. After the assignment, variable c1 and c2 refer to the same object.

*****Insert Figure 6.5 (Same as Figure 5.5 in the 3rd Edition, p142)**

Figure 6.5

Copying an object variable to another does not make a copy of the object; it merely assigns the reference of one object to the other variable.

Garbage Collection

As shown in Figure 6.5, after the assignment statement `c1 = c2`, `c1` points to the same object referenced by `c2`. The object previously referenced by `c1` is no longer useful. This object is known as *garbage*. Garbage occupies memory space. The Java runtime system detects garbage and automatically reclaims the space it occupies. This process is known as *garbage collection*.

TIP: If you know that an object is no longer needed, you can explicitly assign `null` to a reference variable for the object. The Java VM will automatically collect the space if the object is not referenced by any variable.

Accessing an Object's Data and Methods

After an object is created, its data can be accessed and its methods can be invoked using the following dot notation:

`objectReference.data`—References an object's data
`objectReference.method(arguments)`—Invokes an object's method

For example, `myCircle.radius` references the radius of `myCircle`, and `myCircle.findArea()` invokes the `findArea` method of `myCircle`. Methods are invoked as operations on objects.

The data field `radius` is referred to as an *instance variable* because it is dependent on a specific instance. For the same reason, the method `findArea` is referred to as an *instance method*, because you can only invoke it on a specific instance.

NOTE: Most of the time, you create an object and assign it to a variable. You can later reference the object using the variable. Occasionally, an object does not need to be referenced later. In this case, you can create an object without explicitly assigning it to a variable, as shown below:

```
new Circle();
```

Or

```
System.out.println("Area is " + new Circle().findArea());
```

The former statement creates a `Circle` object. The latter statement creates a `Circle` object and invokes its `findArea` method to return its area.

An object created in this way is known as an *anonymous object*.

*****End of NOTE**

Example 6.1 Using Objects

The program in this example creates a Circle object from the Circle class and uses the data and method in the object. The output of the program is shown in Figure 6.6.

```
// TestCircle.java: Demonstrate creating and using an object
package chapter6;

public class TestCircle
{
    /**Main method*/
    public static void main(String[] args)
    {
        Circle myCircle = new Circle(); // Create a Circle object
        System.out.println("The area of the circle of radius "
            + myCircle.radius + " is " + myCircle.findArea());
    }
}

// Define the Circle class
class Circle
{
    double radius = 1.0;

    /**Return the area of this circle*/
    double findArea()
    {
        return radius*radius*3.14159;
    }
}
```

*****Insert Figure 6.6**

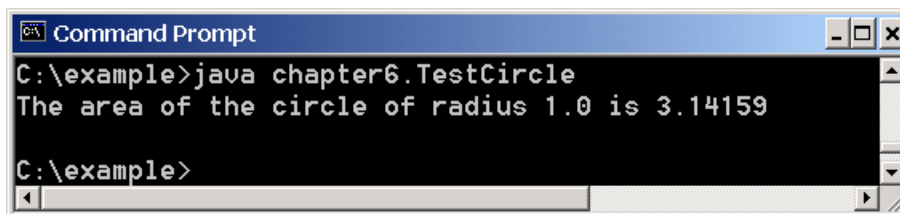


Figure 6.6

This program creates a Circle object and displays its radius and area.

Example Review

The program contains two classes. The first class, TestCircle, is the main class. Its sole purpose is to test the second class, Circle. Every time you run the program, the Java runtime system invokes its main method in the main class.

You can put the two classes into one file, but only one class in the same file is a public class. Furthermore, the public class must have the same name as the file name. Therefore, the file name is TestCircle.java if both TestCircle and Circle classes are in the same file.

The main class contains the main method that creates an object of the Circle class and prints its radius and area. The Circle class contains the findArea method and the radius data field.

To write the findArea method in a procedural programming language like Pascal, you would pass radius as an argument to the method. But in the object-oriented programming, radius and findArea are defined in the same class. The radius is a data member in the Circle class, which is accessible by the findArea method. In procedural programming languages, data and methods are separated, but in an object-oriented programming language, data and methods are defined together in a class.

The findArea method is an instance method that is always invoked by an instance in which the radius is specified.

There are many ways to write Java programs. For instance, you can combine the two classes in the example into one, as follows:

```
public class Circle
{
    double radius = 1.0;

    /**Find the area of this circle*/
    double findArea()
    {
        return radius*radius*3.14159;
    }

    /**Main method*/
    public static void main(String[] args)
    {
        // Create a Circle object
        Circle myCircle = new Circle();
        System.out.println("The area of the circle of radius "
            + myCircle.radius + " is " + myCircle.findArea());
    }
}
```

Since the combined class has a main method, it can be executed by the Java interpreter. The main method

creates myCircle to be a Circle object and then displays radius and finds area in myCircle. This demonstrates that you can test a class by simply adding a main method in the same class.

CAUTION: You must always create an object before referencing it through a reference variable. Referencing an object that has not been created would cause a runtime NullPointerException exception. Exception handling will be introduced in Chapter 13, "Exception Handling."

NOTE: The default value of a data field is null for a reference type, 0 for a numeric type, false for the boolean type, and '\u0000' for the char type. For example, if radius is not initialized in the Circle class, Java assigns a default value of 0 to radius. However, Java assigns no default value to a local variable inside a method. The following code is erroneous because x is not initialized:

```
class Test
{
    public static void main(String[] args)
    {
        int x; // x has no default value
        System.out.println("x is " + x);
    }
}
```

*****End Note**

Constructors

One problem with the Circle class that was just discussed is that all of the objects created from it have the same radius (1.0). Wouldn't it be more useful to create circles with radii of various lengths? Java enables you to define a special method in the class, known as the *constructor*, that can be used to initialize an object's data. You can use a constructor to assign an initial radius when you create an object.

The constructor has exactly the same name as the defining class. Like methods, constructors can be overloaded, making it easier to construct objects with different initial data values. Let's see what happens when the following constructors are added to the Circle class:

```
/**Construct a circle with the specified radius*/
Circle(double r)
{
    radius = r;
}
```

```

/**Construct a circle with the default radius*/
Circle()
{
    radius = 1.0;
}

```

When creating a new Circle object that has a radius of 5.0, you can use the following, which assigns 5.0 to myCircle.radius:

```
myCircle = new Circle(5.0);
```

If you create a circle using the following statement, the second constructor is used, which assigns the default radius 1.0 to myCircle.radius:

```
myCircle = new Circle();
```

A constructor with no parameters is referred to as a *default constructor*. The second constructor is a default constructor. Now you know why the syntax ClassName() is used to create the object. This syntax calls a constructor. A constructor can perform any action like a method, but constructors are designed to perform initializing actions, such as initializing the data fields of objects.

NOTE: Constructors are a special kind of method, with three differences:

- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors are invoked using the new operator when an object is created. Constructors play the role of initializing objects.

*****End of NOTE**

Example 6.2 Using Constructors

In this example, a program is written that will use constructors in the CircleWithConstructors class to create two objects of different radii. The output of the program is shown in Figure 6.7.

```

// TestCircleWithConstructors.java: Demonstrate constructors
package chapter6;

public class TestCircleWithConstructors
{
    /**Main method*/
    public static void main(String[] args)
    {
        // Create a Circle with radius 5.0
        CircleWithConstructors myCircle =
            new CircleWithConstructors(5.0);
        System.out.println("The area of the circle of radius "
            + myCircle.radius + " is " + myCircle.findArea());

        // Create a Circle with default radius
        CircleWithConstructors yourCircle = new CircleWithConstructors();
        System.out.println("The area of the circle of radius "

```

```

        + yourCircle.radius + " is " + yourCircle.findArea());

    // Modify circle radius
    yourCircle.radius = 100;
    System.out.println("The area of the circle of radius "
        + yourCircle.radius + " is " + yourCircle.findArea());
}

// Define the Circle class with two constructors
class CircleWithConstructors
{
    double radius;

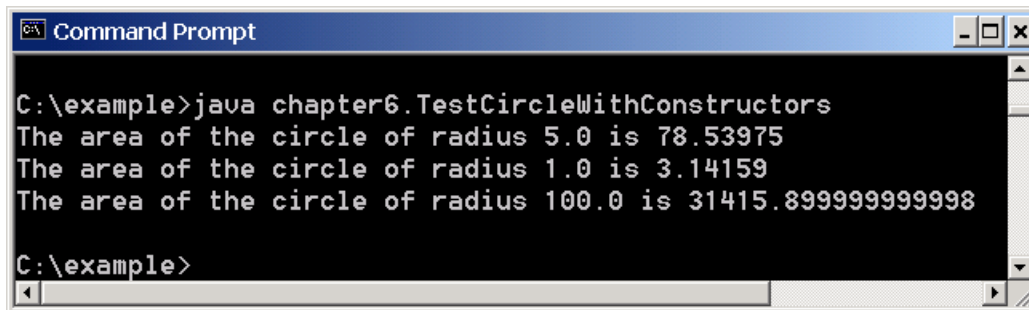
    /**Default constructor*/
    CircleWithConstructors()
    {
        radius = 1.0;
    }

    /**Construct a circle with a specified radius*/
    CircleWithConstructors(double r)
    {
        radius = r;
    }

    /**Return the area of this circle*/
    double findArea()
    {
        return radius*radius*3.14159;
    }
}

```

*****Insert Figure 6.7**



```

C:\example>java chapter6.TestCircleWithConstructors
The area of the circle of radius 5.0 is 78.53975
The area of the circle of radius 1.0 is 3.14159
The area of the circle of radius 100.0 is 31415.899999999998
C:\example>

```

Figure 6.7

The program constructs two circles of radii 5 and 1, and displays their radii and areas.

Example Review

The new circle class is named CircleWithConstructors. This new class has two constructors. You can specify a radius or use the default radius to create a circle object. In this example, two objects were created. The constructor CircleWithConstructors(5.0) was used to create myCircle with a radius of 5.0, and the constructor CircleWithConstructors() was used to create yourCircle with a default radius of 1.0.

These two objects (referenced by myCircle and yourCircle) have different data but share the same methods. Therefore, you can compute their respective areas by using the findArea() method.

NOTE: If a class does not define any constructors explicitly, a default constructor is defined implicitly. If a class defines constructors explicitly, a default constructor does not exist unless it is defined explicitly. Therefore, you cannot use the default constructor to create an object using new ClassName(). See Review Questions 6.11 and 6.12.

CAUTION: If the circle class were not renamed, you would get a compilation error in JBuilder indicating the duplication of the Circle class in the project. One way to fix it is to name them differently as in this example, and the other way is to put them in different packages, as will be demonstrated in Chapter 8, "Class Inheritance and Interfaces."

Visibility Modifiers and Accessor Methods

Example 6.2 works fine, but it is not a good practice to let the client modify the fields directly through the object reference. Doing so often causes programming errors that are difficult to debug. To prevent direct modifications of the properties through the object reference, you can declare the field private, using the private modifier. Java provides several modifiers that control access to data, methods, and classes. This section introduces the public, private, and default modifiers.

- **public**: Defines classes, methods, and data in such a way that all programs can access them.
- **private**: Defines methods and data in such a way that they can be accessed by the declaring class, but not by any other classes.

NOTE: The modifier private applies solely to data or methods, not to classes (except inner classes). Inner classes will be introduced in Chapter 8, "Class Inheritance and Interfaces." If public or private is not used, then by default the classes, methods, and data are accessible by any class in the same package. Java has another visibility modifier called protected, which will be introduced in Chapter 8. Appendix D, "Java Modifiers," contains a table that summarizes all the Java modifiers.

The visibility modifiers are used for the members of the class, not local variables inside the methods. Using visibility modifiers inside a method body would cause a compilation error.

*****End of NOTE**

NOTE: In most cases, the constructor should be public. However, if you want to prohibit the user from creating an instance of a class, you can use a private constructor. For example, there is no reason to create an instance from the Math class because all of the data and methods are static. One solution is to define a dummy private constructor in the class. The Math class has a private constructor, as follows:

```
private Math()  
{  
}
```

Therefore, the Math class cannot be instantiated. The Math class comes with the Java system, which was introduced in the section "The Math Class" in Chapter 4, "Methods."

*****End of NOTE**

A private data field cannot be accessed by the object through a direct reference outside the class that defines the private field. But often a client needs to retrieve and modify the data field. To make a private data field accessible, you can provide a *get* method and a *set* (or *mutator*) method for a private data field. These methods that regulates access to internal data fields are referred to as the *accessor methods* colloquially. These data fields are called *properties* of the object.

NOTE: Colloquially, a *get* method is referred to as a *getter*, and a *set* method is referred to as a *setter*.

A *get* method has the following signature:

```
public returnType getPropertyNames()
```

If the returnType is boolean, the *get* method should be defined as follows by convention:

```
public boolean isPropertyName()
```

A set has the following signature:

```
public void setPropertyName(dataType propertyValue)
```

The example given below demonstrates the use of the private modifier and accessor methods.

Example 6.3 Using the private Modifier and Accessor Methods

This example declares a new circle class, in which private data are used for the radius, and the accessor methods getRadius and setRadius are provided for the clients to retrieve and modify the radius. The program creates an instance of the CircleWithAccessor class and modifies the radius using the setRadius method. The output of the sample run is shown in Figure 6.8.

```
// TestCircleWithAccessors.java: Demonstrate private modifier  
package chapter6;  
  
public class TestCircleWithAccessors  
{  
    /**Main method*/  
    public static void main(String[] args)  
    {  
        // Create a Circle with radius 5.0  
        CircleWithAccessors myCircle = new CircleWithAccessors(5.0);  
        System.out.println("The area of the circle of radius "  
            + myCircle.getRadius() + " is " + myCircle.findArea());  
  
        // Increase myCircle's radius by 10%  
        myCircle.setRadius(myCircle.getRadius()*1.1);  
        System.out.println("The area of the circle of radius "  
            + myCircle.getRadius() + " is " + myCircle.findArea());  
    }  
}
```

******Layout: Insert a separator line here. AU***

```
// CircleWithAccessors.java: The circle class with accessor methods  
package chapter6;  
  
public class CircleWithAccessors  
{  
    /**The radius of the circle*/  
    private double radius;  
  
    /**Default constructor*/  
    public CircleWithAccessors()  
    {  
        radius = 1.0;  
    }  
  
    /**Construct a circle with a specified radius*/  
    public CircleWithAccessors(double r)  
    {  
        radius = r;  
    }  
  
    /**Return radius*/  
    public double getRadius()  
    {  
        return radius;  
    }  
  
    /**Set a new radius*/
```



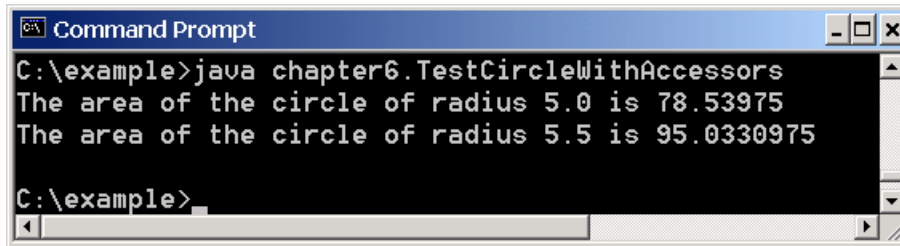
```

    public void setRadius(double newRadius)
    {
        radius = newRadius;
    }

    /**Return the area of this circle*/
    public double findArea()
    {
        return radius*radius*3.14159;
    }
}

```

*****Insert Figure 6.8**



```

C:\example>java chapter6.TestCircleWithAccessors
The area of the circle of radius 5.0 is 78.53975
The area of the circle of radius 5.5 is 95.0330975
C:\example>

```

Figure 6.8

This program creates a circle object and uses the get method to read radius, and the set method to modify its radius.

Example Review

The data field radius is declared private. Private data can only be accessed within their defining class. You cannot use myCircle.radius in the client program. A compilation error would occur if you attempted to access private data from a client.

To access radius, you have to use the getRadius method to retrieve the radius and the setRadius method to modify the radius.

Suppose you combine TestCircleWithAccessors and CircleWithAccessors into one class by moving the main method in TestCircleWithAccessors into CircleWithAccessors, can you use myCircle.radius in the main method? See Review Question 6.16 for the answer.

NOTE: The CircleWithAccessors class will be used for the projects in Chapter 6. Therefore, this class is declared public. Since one file can have only one public class, TestCircleWithAccessors and CircleWithAccessors are stored in two separate files. If classes in

different files are listed one after the other,
the book separates them using a separator line.

Passing Objects to Methods

So far, you learned to pass the parameters of primitive types and arrays to methods, you can also pass the objects to methods. Like passing arrays, passing objects is actually passing the reference of the object. The following code passes the myCircle object as an argument to the method printCircle():

```
class TestPassingObject
{
    public static void main(String[] args)
    {
        CircleWithAccessors myCircle = new CircleWithAccessors(5.0);
        printCircle(myCircle);
    }

    public static void printCircle(CircleWithAccessors c)
    {
        System.out.println("The area of the circle of radius "
            + c.getRadius() + " is " + c.findArea());
    }
}
```

Java uses exactly one mode of passing parameters, i.e. pass by value. In the above code, the value of myCircle is passed to the printCircle method. This value contains a reference to a circle object.

Example 6.4 Passing Objects as Arguments

In this example, a program is written to pass a Circle object and an integer value to the method printAreas, which prints a table of areas for radii 1, 2, 3, 4, and 5. The output of the program is shown in Figure 6.9.

```
// TestPassingObject.java: Demonstrate passing objects to methods
package chapter6;

public class TestPassingObject
{
    /**Main method*/
    public static void main(String[] args)
    {
        // Create a Circle object with default radius 1
        CircleWithAccessors myCircle = new CircleWithAccessors();

        // Print areas for radius 1, 2, 3, 4, and 5.
        int n = 5;
        printAreas(myCircle, n);

        // See myCircle.radius and times
        System.out.println("\n" + "Radius is " + myCircle.getRadius());
        System.out.println("n is " + n);
    }

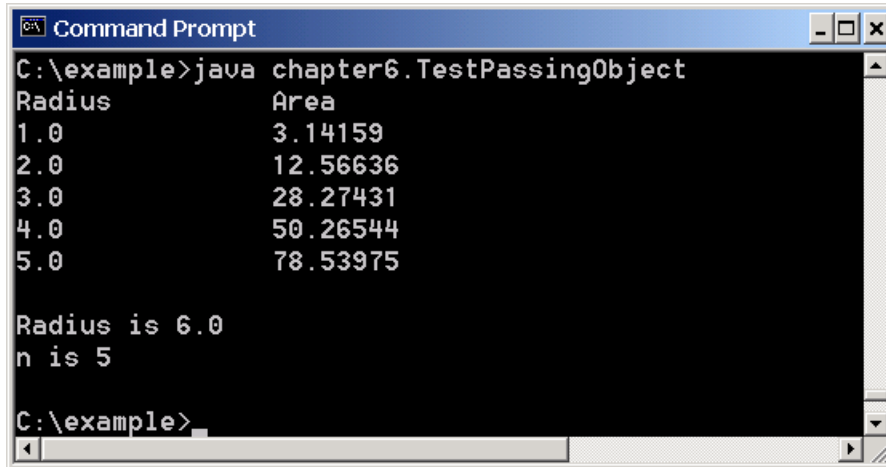
    /**Print a table of areas for radius*/
    public static void printAreas(CircleWithAccessors c, int times)
    {
        System.out.println("Radius \t\tArea");
        while (times >= 1)
        {
            System.out.println(c.getRadius() + "\t\t" + c.findArea());
        }
    }
}
```

```

        c.setRadius(c.getRadius()+1);
        times--;
    }
}

```

***Insert Figure 6.9



```

C:\example>java chapter6.TestPassingObject
Radius      Area
1.0         3.14159
2.0         12.56636
3.0         28.27431
4.0         50.26544
5.0         78.53975

Radius is 6.0
n is 5

C:\example>

```

Figure 6.9

The program passes a circle object myCircle and an integer value n as parameters to the printAreas method, which displays a table of the areas for radii 1, 2, 3, 4, and 5.

Example Review

The main method invokes the printAreas method by passing an object myCircle and an integer n to the printAreas method, as shown in Figure 6.10.

***Insert Figure 6.10 (Same as Figure 5.9 in the 3rd Edition, p149)

Figure 6.10

The value of n is passed to times, and the reference of myCircle object is passed to c in the printAreas method.

When passing a parameter of a primitive data type, the value of the actual parameter is passed. In this case, the value of n (5) is passed to times. Inside the printAreas method, the content of times is changed; this does not affect the content of n. When passing a parameter of a reference type, the reference of the object is passed. In this case, c contains a reference

for the object that is also referenced via myCircle. Therefore, changing properties of the object through c inside the printAreas method has the same effect as doing so outside the method through variable myCircle.

Static Variables, Constants, and Methods

The variable radius in the circle classes in the previous examples is known as an *instance variable*. An instance variable is tied to a specific instance of the class; it is not shared among objects of the same class. For example, suppose that you create the following objects:

```
Circle circle1 = new Circle();  
Circle circle2 = new Circle(5);
```

The radius in circle1 is independent of the radius in circle2, and is stored in different memory location. Changes made to circle1's radius do not affect circle2's radius, and vice versa.

If you want all the instances of a class to share data, use *static variables*, also known as *class variables*. Static variables store values for the variables in a common memory location. Because of this common location, all objects of the same class are affected if one object changes the value of a static variable.

To declare a static variable, put the modifier static in the variable declaration. Suppose that you want to track the number of objects of the CircleWithStaticVariable class created, you can define the static variable as follows:

```
static int numOfObjects;
```

Figure 6.11 pictures the instance variables and static variables.

*****Insert Figure 6.11**

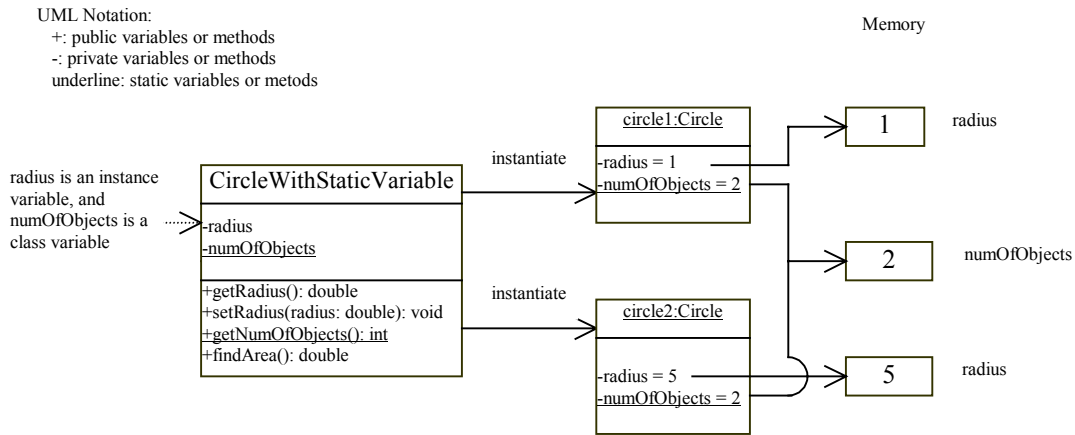


Figure 6.11

The instance variables, which belong to the instances, have memory storage independent of one other. The static variables are shared by all the instances of the same class.

To declare a class constant, add the final keyword in the above declaration. For example, the constant PI in the Math class is defined as follows:

```
public final static double PI = 3.14159265358979323846;
```

Instance methods belong to instances and can only be applied after the instances are created. They are called by the following:

```
objectName.methodName();
```

The methods defined in the previous circle classes are instance methods. Java supports static methods as well as static variables. *Static methods*, also known as *class methods*, can be called without creating an instance of the class. To define a static method, put the modifier static in the method declaration, as follows:

```
static returnType staticMethod();
```

Examples of static methods are the readDouble() and the readInt() in the class MyInput, and all the methods in the Math class. In fact, so are all the methods used in Chapters 2, 3, and 4 and the static methods, including the main method.

Static methods are called by one of the following syntaxes:

```
ClassName.methodName();
```

```
objectName.methodName();
```

For example, `MyInput.readInt()` is a call that reads an integer from the keyboard. `MyInput` is a class, not an object.

*****NOTE Begin**

NOTE: Instance variables can be accessed from instance methods in a class, but not from a static method in a class. Thus the following code would be wrong.

```
class Foo
{
    int i = 5;

    static void p()
    {
        int j = i; // Wrong because p() is static and i is non-static
    }
}
```

*****NOTE End**

*****TIP Begin**

TIP: A method that does not use instance variables can be defined as a static method. This method can be invoked without creating an object of the class.

You should define a constant as static data that can be shared by all class objects. Do not change the value of a constant.

Variables that describe common properties of objects should be declared as static variables.

*****TIP END**

The following example demonstrates how to use instance and static variables and methods, and illustrates the effects of using them.

Example 6.5 Using Instance and Static Variables and Methods

This example adds a static variable `numOfObjects` to track the number of circle objects created. The new circle class can be shown using graphical notations in Figure 6.12.

***Insert Figure 6.12

CircleWithStaticVariable
-radius: double -numOfObjects: int
+getRadius(): double +setRadius(newRadius: double): void +getNumOfObjects(): void +findArea(): double

Figure 6.12

The CircleWithStaticVariable class defines instance variable radius, static variable numOfObjects, instance methods getRadius, setRadius, findArea, and a static method getNumOfObjects.

The main method creates two circles, circle1 and circle2, and modifies the instance and static variables. You will see the effect of using instance and static variables after changing the data in the circles. The output of the program is shown in Figure 6.13.

```
// TestCircleWithStaticVariable.java: Demonstrate using instance and
// static variables
package chapter6;

public class TestCircleWithStaticVariable
{
    /**Main method*/
    public static void main(String[] args)
    {
        // Create circle1
        CircleWithStaticVariable circle1 = new CircleWithStaticVariable();

        // Display circle1 BEFORE circle2 is created
        System.out.println("Before creating circle2");
        System.out.print("circle1 is : ");
        printCircle(circle1);

        // Create circle2
        CircleWithStaticVariable circle2 = new CircleWithStaticVariable(5);

        // Change the radius in circle1
        circle1.setRadius(9);

        // Display circle1 and circle2 AFTER circle2 was created
        System.out.println("\nAfter creating circle2 and modifying " +
            "circle1's radius to 9");
        System.out.print("circle1 is : ");
        printCircle(circle1);
        System.out.print("circle2 is : ");
        printCircle(circle2);
    }

    /**Print circle information*/
    public static void printCircle(CircleWithStaticVariable c)
    {
        System.out.println("radius (" + c.getRadius() +
            ") and number of Circle objects (" +
            c.getNumOfObjects() + ")");
    }
}
```

```

}

class CircleWithStaticVariable
{
    /**The radius of the circle*/
    private double radius;

    /**The number of the objects created*/
    private static int numObjects = 0;

    /**Default constructor*/
    public CircleWithStaticVariable()
    {
        radius = 1.0;
        numObjects++;
    }

    /**Construct a circle with a specified radius*/
    public CircleWithStaticVariable(double r)
    {
        radius = r;
        numObjects++;
    }

    /**Return radius*/
    public double getRadius()
    {
        return radius;
    }

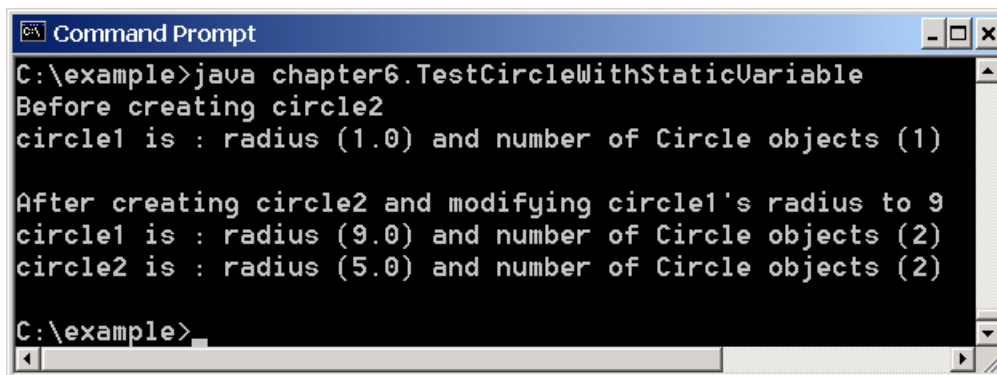
    /**Set a new radius*/
    public void setRadius(double newRadius)
    {
        radius = newRadius;
    }

    /**Return numObjects*/
    public static int getNumObjects()
    {
        return numObjects;
    }

    /**Return the area of this circle*/
    public double findArea()
    {
        return radius*radius*Math.PI;
    }
}

```

*****Insert Figure 6.13**



```

C:\example>java chapter6.TestCircleWithStaticVariable
Before creating circle2
circle1 is : radius (1.0) and number of Circle objects (1)

After creating circle2 and modifying circle1's radius to 9
circle1 is : radius (9.0) and number of Circle objects (2)
circle2 is : radius (5.0) and number of Circle objects (2)

C:\example>

```

Figure 6.13

The program uses the instance variable radius as well as the static variable numOfObjects. All of the objects share the same numOfObjects.

Example Review

The instance variable radius in circle1 is modified to become 9. This change does not affect the instance variable radius in circle2, since these two instance variables are independent. The static variable numOfObjects becomes 1 after circle1 is created, and it becomes 2 after circle2 is created. This change affects all the instances of the Circle class, since the static variable numOfObjects is shared by all the instances of the Circle class.

Note that Math.PI was used to access PI, and that c.numOfObjects in the printCircle method is used to access numOfObjects. Math is the class name, and c is an object of the Circle class. To access a constant like PI, you can use either the ClassName.CONSTANTNAME or the objectName.CONSTANTNAME. To access an instance variable like radius, you need to use objectName.variableName.

TIP: I recommend that you invoke static variables and methods using ClassName.variable and ClassName.method. This improves readability because the reader can easily recognize the static variables and methods. Thus you should replace c.getNumOfObjects() by Circle.getNumOfObjects() in this example.

TIP: How do you decide whether a variable (method) should be an instance or a static variable (method)? A variable (method) that is dependent on a specific instance of the class should be an instance variable (method). A variable (method) that is not dependent on a specific instance of the class should be a static variable (method). For instance, every circle has its own radius. Radius is dependent on a specific circle. Therefore, radius is an instance variable of the Circle class. Since the findArea method is dependent on a specific circle, this method is an instance method. All the methods in the Math class such as random, pow, sin, and cos are not dependent on a specific instance. Therefore, these methods are static methods. The main method is static, which can be invoked directly from a class.

The Scope of Variables

Chapter 4, "Methods," discussed local variables and their scope rules. Local variables are declared and used inside a method locally. This section discusses the scope rules of all variables in the context of a class.

The scope of a class's variables (instance and static variables) is the entire class, regardless of where the variables are declared. The class's variables can be declared anywhere in a class. For example, you can declare a variable at the end of a class, and use the variable in the method that is defined earlier in a class, as shown below.

```
class Circle
{
  double findArea()
  {
    return radius*radius*Math.PI;
  }

  double radius = 1;
}
```

NOTE: The data fields and methods are the members of the class. There is no order among them. Therefore, they can be declared in any order in a class. One exception is that if a data field is initialized based on a reference of another data field. The other data field must be declared before this data. For example, in the following class, i must be declared before j, because i's value is used to initialize j.

```
public class Foo
{
  int i;
  int j = i + 1;
}
```

*****End of NOTE**

You can declare a variable only once as a class member (instance variable or static variable), but you can declare the same variable in a method multiple times in different non-nesting blocks.

If a local variable has the same name as a class's variable, the local variable takes precedence and the class's variable with the same name is hidden. For example, in the following program, x is defined as an instance variable and as a local variable in the method.

```
class Foo
{
  int x = 0; // instance variable
  int y = 0;
```

```

    Foo()
    {
    }

    void p()
    {
        int x = 1; // local variable
        System.out.println("x = " + x);
        System.out.println("y = " + y);
    }
}

```

What is the printout for `f.p()`, where `f` is an instance of `Foo`? The printout for `f.p()` is 1 for `x` and 0 for `y`, based on the following reasons:

- `x` is declared as a data field with an initial value 0 in the class, but is also defined in the method `p()` with an initial value of 1. The latter `x` is referenced in the `System.out.println` statement.
- `y` is declared outside the method `p()`, but is accessible inside it.

TIP: As demonstrated in the example, it is easy to make mistakes. To avoid confusion, do not declare the same variable names in a class, except for method parameters.

The Keyword `this`

Sometime you need to reference a class's hidden variable in a method. For example, it is common to use a property name as the parameter name in a set method for the property. In this case, you need to reference the hidden property name in the method in order to set a new value to it. A hidden static variable can be accessed simply by using the `ClassName.StaticVariable` reference. A hidden instance variable can be accessed by using the keyword `this`, as shown in the following code:

```

class Foo
{
    int i = 5;

    void setI(int i)
    {
        this.i = i;
    }
}

```

The line `this.i = i` means "assign argument `i` to the object's data field `i`."

You can also use the keyword `this` in the constructor. For example, you can redefine the `Circle` class as follows:

```

public class Circle
{
    private double radius;
}

```

```

public Circle(double radius)
{
    this.radius = radius;
}

public Circle()
{
    this(1.0);
}

public double findArea()
{
    return radius*radius*Math.PI;
}
}

```

The line this(1.0) invokes the constructor with a double value argument in the class.

TIP: If a class has multiple constructors, I recommend you to implement the constructors using this(arg-list) as much as possible. In general, a constructor with no or less arguments can invoke the constructor with more arguments using this(arg-list). This often simplifies coding and makes the class easier to read.

NOTE: Java requires the this(arg-list) statement to appear first in the constructor before any other statements.

Array of Objects

In Chapter 5, "Arrays," arrays of primitive type elements were created. You can also create arrays of objects. For example, the following statement declares and creates an array of 10 Circle objects:

```
Circle[] circleArray = new Circle[10];
```

To initialize the circleArray, you can use a for loop like this one:

```

for (int i=0; i<circleArray.length; i++)
{
    circleArray[i] = new Circle();
}

```

An array of objects is actually an *array of reference variables*. So invoking circleArray[1].findArea() involves two levels of referencing as shown in Figure 6.14. circleArray references to the entire array. circleArray[1] references to a Circle object.

*****Insert Figure 6.14**

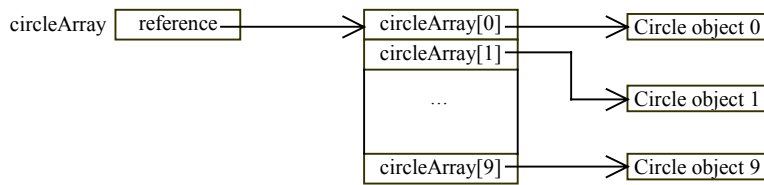


Figure 6.14

In an array of objects, an element of the array contains a reference to an object.

NOTE: When an array of objects is created using the new operator, each element is a reference variable with a default value null.

The next example demonstrates how to use an array of objects.

Example 6.6 Summarizing the Areas of the Circles

In this example, a program is written that will summarize the areas of an array of circles. The program creates circleArray—an array composed of 10 circle objects—and initializes circle radii with random values, then display the total area of the circles in the array. The output of a sample run of the program is shown in Figure 6.15.

```
// TotalArea.java: Test passing an array of objects to the method
package chapter6;

public class TotalArea
{
    /**Main method*/
    public static void main(String[] args)
    {
        // Declare circleArray
        CircleWithAccessors[] circleArray;

        // Create circleArray
        circleArray = createCircleArray();

        // Print circleArray and total areas of the circles
        printCircleWithAccessorsArray(circleArray);
    }

    /**Create an array of CircleWithAccessors objects*/
    public static CircleWithAccessors[] createCircleArray()
    {
        CircleWithAccessors[] circleArray = new CircleWithAccessors[10];

        for (int i=0; i<circleArray.length; i++)
        {
            circleArray[i] = new CircleWithAccessors(Math.random()*100);
        }

        // Return CircleWithAccessors array
        return circleArray;
    }
}
```

```

    /**Print an array of circles and their total area*/
    public static void printCircleWithAccessorsArray
    (CircleWithAccessors[] circleArray)
    {
        System.out.println("The radii of the circles are");
        for (int i=0; i<circleArray.length; i++)
        {
            System.out.print("\t\t\t\t" +
                circleArray[i].getRadius() + '\n');
        }

        System.out.println("\t\t\t\t\t-----");

        // Compute and display the result
        System.out.println("The total areas of circles is \t" +
            sum(circleArray));
    }

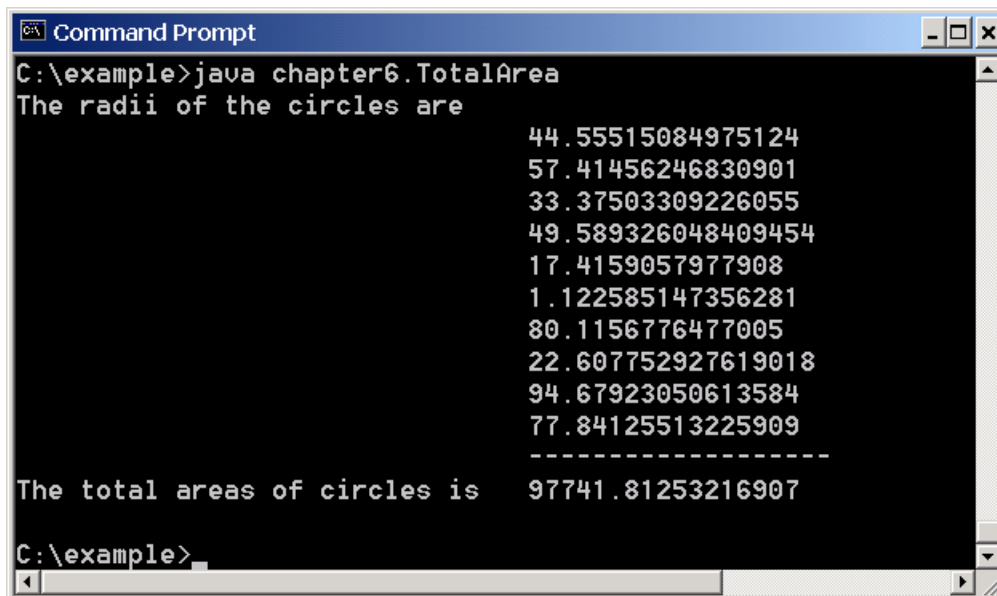
    /**Add circle areas*/
    public static double sum(CircleWithAccessors[] circleArray)
    {
        // Initialize sum
        double sum = 0;

        // Add areas to sum
        for (int i = 0; i < circleArray.length; i++)
            sum += circleArray[i].findArea();

        return sum;
    }
}

```

***Insert Figure 6.15



```

C:\example>java chapter6.TotalArea
The radii of the circles are
44.55515084975124
57.41456246830901
33.37503309226055
49.589326048409454
17.4159057977908
1.122585147356281
80.1156776477005
22.607752927619018
94.67923050613584
77.84125513225909
-----
The total areas of circles is 97741.81253216907
C:\example>

```

Figure 6.15

The program creates an array of circle objects, displays their radii and total area.

Example Review

The program uses the `createCircleArray` method to create an array of 10 circle objects. Several circle

classes were introduced in this chapter. This example uses the CircleWithAccessors class introduced in Example 6.3, "Using the private Modifier and Accessor Methods."

The circle radii are randomly generated using the Math.random() method. The createCircleObject method returns an array of CircleWithAccessors objects. The array is passed to the printCircleArray method, which displays the radii of the total area of the circles.

The sum of the areas of the circle is computed using the sum method, which takes the array of circle objects as the argument and returns a double value for the total area.

Class Abstraction

In Chapter 4, "Methods," you learned about method abstraction and used it in program development. Java provides many levels of abstraction. *Class abstraction* is the separation of the class implementation from the use of the class. The creator of the class provides a description of the class and lets the user know how the class can be used. The collection of methods and fields that are accessible from outside a class, together with the description of how these members are expected to behave, serves as the *class's contract*. The user of the class does not need to know how the class is implemented. The details of implementation are encapsulated and hidden from the user. For example, you can create a Circle object and find the area of the circle without knowing how the area is computed. There are many real-life examples that illustrate the concept of class abstraction.

Consider building a computer system, for instance. Your personal computer is made up of many components, such as a CPU, CD-ROM, floppy disk, motherboard, fan, and so on. Each component can be viewed as an object that has properties and methods. To get the components to work together, all you need to know is how each component is used and how it interacts with the others. You don't need to know how it works internally. The internal implementation is encapsulated and hidden from you. You can build a computer without knowing how a component is implemented.

The computer system analogy just presented precisely mirrors the object-oriented approach. Each component can be viewed as an object of the class for the component. For example, you might have a class that models all kinds of fans for use on a computer with properties like fan size, speed, and so on, and methods like start, stop, and so on. A specific fan is an instance of this class with specific property values.

Consider paying a mortgage, for another example. A specific mortgage can be viewed as an object of a Mortgage class. Interest rate, loan amount, and loan period are its data properties, and computing monthly payment and total payment are its methods. When you buy a house, a mortgage object is created by instantiating the class with your mortgage interest rate, loan amount, and loan period. You can then use the mortgage methods to easily find the monthly payment and total payment of your loan. As a user of the Mortgage class, you don't need to know how these methods are implemented.

Case Studies

This section uses two examples to demonstrate creating and using classes.

Example 6.7 Using the Mortgage Class

In this example, a mortgage class named Mortgage is created with the following data fields: annualInterestRate, numOfYears, and loanAmount, and the methods getAnnualInterestRate, getNumOfYears, getLoanAmount, setAnnualInterestRate, setNumOfYears, setLoanAmount, monthlyPayment, and totalPayment, as shown in Figure 6.16. The monthlyPayment method returns the monthly payment, and the totalPayment method returns the total payment.

***Insert Figure 6.16

Mortgage
-annualInterestRate: double -numOfYears: int -loanAmount: double
+Mortgage() +Mortgage(annualInterestRate: double, numOfYears: int, loanAmount: double) +getAnnualInterestRate(): double +getNumOfYears(): int +getLoanAmount(): double +setAnnualInterestRate(annualInterestRate: double): void +setNumOfYears(numOfYears: int): void +setLoanAmount(loanAmount: double): void +monthlyPayment(): double +totalPayment(): double

Figure 6.16

The Mortgage class models the properties and behaviors of mortgages.

The Mortgage class is given below followed by a test program. Figure 6.17 shows the output of a sample run of the program.

```
// Mortgage.java: Encapsulate mortgage information
package chapter6;

public class Mortgage
{
    private double annualInterestRate;
    private int numOfYears;
    private double loanAmount;

    /**Default constructor*/
    public Mortgage()
    {
        this(7.5, 30, 100000);
    }

    /**Construct a mortgage with specified annual interest rate,
        number of years and loan amount
    */
    public Mortgage(double annualInterestRate, int numOfYears,
        double loanAmount)
    {
        this.annualInterestRate = annualInterestRate;
        this.numOfYears = numOfYears;
        this.loanAmount = loanAmount;
    }

    /**Return annualInterestRate*/
    public double getAnnualInterestRate()
    {
        return annualInterestRate;
    }

    /**Set a new annualInterestRate*/
    public void setAnnualInterestRate(double annualInterestRate)
    {
        this.annualInterestRate = annualInterestRate;
    }

    /**Return numOfYears*/
    public int getNumOfYears()
    {
        return numOfYears;
    }

    /**Set a new numOfYears*/
    public void setNumOfYears(int numOfYears)
    {
        this.numOfYears = numOfYears;
    }

    /**Return loanAmount*/
    public double getLoanAmount()
    {
        return loanAmount;
    }

    /**Set a newloanAmount*/
    public void setLoanAmount(double loanAmount)
    {
        this.loanAmount = loanAmount;
    }

    /**Find monthly payment*/
    public double monthlyPayment()
    {
        double monthlyInterestRate = annualInterestRate/1200;
        return loanAmount*monthlyInterestRate/
            (1 - (Math.pow(1/(1 + monthlyInterestRate), numOfYears*12)));
    }

    /**Find total payment*/
    public double totalPayment()
    {

```

```

    return monthlyPayment()*numOfYears*12;
}
}

```

*****Layout: Insert a separator line here. AU**

```

// TestMortgageClass.java: Demonstrate using the Mortgage class
package chapter6;

import chapter2.MyInput;
public class TestMortgageClass
{
    /**Main method*/
    public static void main(String[] args)
    {
        // Enter interest rate
        System.out.print(
            "Enter yearly interest rate, for example 8.25: ");
        double interestRate = MyInput.readDouble();

        // Enter years
        System.out.print(
            "Enter number of years as an integer, for example 5: ");
        int year = MyInput.readInt();

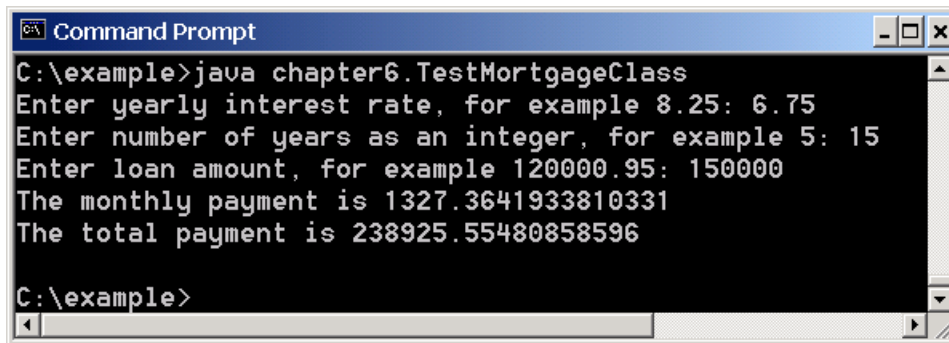
        // Enter loan amount
        System.out.print(
            "Enter loan amount, for example 120000.95: ");
        double loan = MyInput.readDouble();

        // Create Mortgage object
        Mortgage m = new Mortgage(interestRate, year, loan);

        // Display results
        System.out.println("The monthly payment is " +
            m.monthlyPayment());
        System.out.println("The total payment is " +
            m.totalPayment());
    }
}

```

*****Insert Figure 6.17**



```

C:\example>java chapter6.TestMortgageClass
Enter yearly interest rate, for example 8.25: 6.75
Enter number of years as an integer, for example 5: 15
Enter loan amount, for example 120000.95: 150000
The monthly payment is 1327.3641933810331
The total payment is 238925.55480858596
C:\example>

```

Figure 6.17

The program creates a Mortgage instance with the annual interest rate, number of years, and loan amount, and displays monthly payment and total payment by invoking the methods of the instance.

Example Review

The Mortgage class contains a constructor, three get methods, and the methods for finding monthly payment and total payment. You can construct a Mortgage object by using three parameters: annual interest rate, number of years, and loan amount. The three get methods getAnnualInterest, getNumOfYears, and getLoanAmount return annual interest rate, payment years, and loan amount, respectively. All the data properties and methods in this class are tied to a specific instance of the Mortgage class. Therefore, they are instance variables or methods.

The main class reads interest rate, payment period (in years), and loan amount; creates a Mortgage object; and then obtains the monthly payment and total payment using the instance methods in the Mortgage class.

Since the Mortgage class will be used later in the boob, this class must be declared public and stored in a separate file.

Example 6.8 The Count Class

In this example, a class named Count is created with the following data fields count, numOfCounts, and methods getCount, setCount, getNumOfCounts, clear, increment, and decrement, as shown in Figure 6.18. The getCount, setCount, and getNumOfCounts are the accessor methods for Count. The clear method sets the count to 0. The increment and decrement methods increase and decrease the count.

*****Insert Figure 6.18**

Count
-count: int -numOfCounts: int
+Count() +getCount(): int +setCount(count: int): void +getNumOfCounts(): int +clear(): void +increment(): void +decrement(): void

Figure 6.18

The Count class models the counts.

The Count class along with a test program is presented below. The test program counts votes for two candidates for student body president. The votes are entered from the keyboard. Number 1 is a vote for Candidate 1 and number 2 is a vote for Candidate 2. -1 is to deduct a vote from Candidate 1, and -2 is to deduct a vote from candidate 2. Number 0 signifies the end of the count. Figure 6.19 shows the sample run of the program.

```
// TestCountClass.java: Count votes
import chapter2.MyInput;

public class TestCountClass
{
    public static void main(String[] args)
    {
        // Create Count object for each candidate
        Count count1 = new Count();
        Count count2 = new Count();

        // Count votes
        while (true)
        {
            System.out.print("Enter a vote: ");
            int vote = MyInput.readInt();
            if (vote == 0) break; // End of the votes
            if (vote == 1) count1.increment();
            if (vote == 2) count2.increment();
            if (vote == -1) count1.decrement();
            if (vote == -2) count2.decrement();
        }

        System.out.println("The total number of candidates is " +
            Count.getNumOfCounts());
        System.out.println("The votes for Candidate 1 is " +
            count1.getCount());
        System.out.println("The votes for Candidate 2 is " +
            count2.getCount());
    }
}

// Define the Count class
class Count
{
    /**The count value in a Count object*/
    private int count = 0;

    /**The number of Count objects created*/
    private static int numOfCounts = 0;

    /**Construct a count*/
    public Count()
    {
        numOfCounts++;
    }

    /**Return the count*/
    public int getCount()
    {
        return count;
    }

    /**Set a new count*/
    public void setCount(int count)
    {
        this.count = count;
    }

    /**Return number of count objects*/
    public static int getNumOfCounts()
    {
        return numOfCounts;
    }
}
```

```

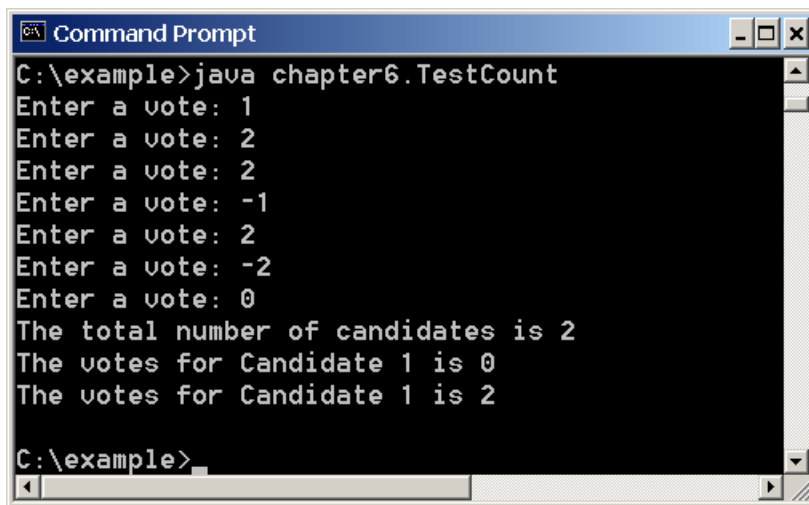
    /**Clear this count*/
    public void clear()
    {
        count = 0;
    }

    /**Increment this count*/
    public void increment()
    {
        count++;
    }

    /**Decrement this count*/
    public void decrement()
    {
        count--;
    }
}

```

*****Insert Figure 6.19**



```

C:\example>java chapter6.TestCount
Enter a vote: 1
Enter a vote: 2
Enter a vote: 2
Enter a vote: -1
Enter a vote: 2
Enter a vote: -2
Enter a vote: 0
The total number of candidates is 2
The votes for Candidate 1 is 0
The votes for Candidate 1 is 2
C:\example>

```

Figure 6.19

The program creates two instances of the Count class and counts the votes.

Example Review

The Count class has a private property count, and the public methods clear, increment, and decrement for handling a count. All of these are tied to a specific instance of the Count class. Therefore, they are instance variables or methods. The Count class also has a private static property numOfCounts, which keeps track the number of the Count objects created.

The default value for count is 0. When a Count instance is constructed, its initial count value is 0. Whenever a Count object is created, numOfCounts is incremented by 1.

Java Application Programmer Interface

Java comes with a rich set of classes that you can use to build applications. These classes are grouped into packages that provide a convenient way to organize classes. You can put the classes you have developed in packages and distribute them to other people. You can think of packages as libraries to be shared by many users.

You learned the Math class in the java.lang package in Chapter 4, "Methods." You will learn more about Java system predefined packages in the coming chapters.

The Java *Application Programmer Interface*, known as *API*, consists of numerous classes and interfaces grouped into 15 core packages, such as java.lang, java.awt, java.event, javax.swing, java.applet, java.util, java.io, and java.net. Interfaces are new construct to be introduced in Chapter 8, "Class Inheritance and Interfaces."

- **java.lang**: Contains core Java classes (such as System, Math, Object, String, StringBuffer, Number, Character, Boolean, Byte, Short, Integer, Long, Float, and Double). This package is implicitly imported to every Java program.
- **java.awt**: Contains classes for drawing geometrical figures, managing component layout, and creating peer-based (so-called heavyweight) components, such as windows, frames, panels, menus, buttons, fonts, lists, and many others.
- **java.awt.event**: Contains classes for handling events in event-driven programming.
- **javax.swing**: Contains the lightweight graphical user interface components.
- **java.applet**: Contains classes for supporting applets.
- **java.io**: Contains classes for input and output streams and files.
- **java.util**: Contains many utilities, such as date, calendar, locale, system properties, vectors, hashing, and stacks.
- **java.text**: Contains classes for formatting information, such as date and time, in a number of formatting styles based on language, country, and culture.

- **java.net**: Contains classes for supporting network communications.

The java.lang is the most fundamental package supporting basic operations. Many of the popular classes in java.lang are introduced later in the book. See the following chapters for information on these classes:

- Chapter 7, "Strings," introduces the String class and StringBuffer class in the java.lang package, and the StringTokenizer class in the java.util package.
- Chapter 8, "Class Inheritance and Interfaces," covers the Object class and the interfaces Comparable and Cloneable in the java.lang package.
- Chapter 9, "Object-Oriented Software Development," introduces wrapper classes, such as Integer, and Double, in the java.lang package.
- Chapter 10, "Getting Started with Graphics Programming," and Chapter 11, "Creating User Interfaces," introduce the classes in the packages java.awt, java.awt.event, and javax.swing, which are used for drawing geometrical figures, event-driven programming, and creating graphical user interfaces.
- Chapter 12, "Applets and Advanced Graphics," introduces the Applet in the java.applet package, which is used to program Java applets.
- Chapter 13, "Exception Handling," discusses using the java.lang.Throwable class and its subclasses for exception handling.
- Chapter 14, "Input and Output," discusses the use of the classes in the java.io package by input and output streams.
- Chapter 15, "Internationalization," introduces classes java.util.Date, java.util.TimeZone, java.util.Calendar, and java.text.DateFormat for processing and formatting date and time based on locale.
- Chapter 16, "Multithreading," focuses on the java.lang.Thread class and the java.lang.Runnable interface, which are used for multithreading.

- Chapter 17, "Multimedia," addresses the use of multimedia by several classes in the packages java.awt and java.applet.
- Chapter 18, "Networking," discusses using the classes in the java.net package for network programming.
- Chapter 19, "Java Data Structures," introduces the classes and interfaces in the Java Collections Framework. These classes are in the java.util package.

TIP: You can use Search, Browse Classes to view the class source code and documentation. You need to enter the full name of the class including its package name. For example, to view the class String, you must type java.lang.String in the Browse Classes dialog box.

Once you understand the concept of Java programming, the most important lesson in Java is learning how to use the API to develop useful programs. For example, to create a window as shown in Figure 6.20, you can use the Frame class in the java.awt package, as follows:

```
import java.awt.*;

class Test
{
    public static void main(String[] args)
    {
        Frame frame = new Frame();

        frame.setTitle("A Magic Window");
        frame.setSize(300, 300);
        frame.setLocation(200, 100);
        frame.setVisible(true);
    }
}
```

*****Insert Figure 6.20**



Figure 6.20

The program creates a window using the Frame class.

This program creates an object of the Frame class and then uses the methods setTitle, setSize, setLocation, and setVisible to set the properties of the object. The setTitle method sets a title for the window. The setSize method sets the window's width and height. The setLocation method specifies the location of the window's upper left corner. The setVisible method displays the window. You can add graphical user interface components such as buttons, labels, text fields, combo boxes, lists, and menus to the window. The components are defined using classes. Graphics programming will be introduced in Part III, "Graphics Programming."

Chapter Summary

In this chapter, you learned how to program using objects and classes. You learned how to define classes, create objects, and use objects. You also learned about visibility modifiers, instance variables, static variables, instance methods, and static methods.

A class is a template for objects. It defines the generic properties of objects and provides methods for manipulating them.

An object is an instance of a class. It is declared in the same way as a primitive type variable. You use the new operator to create an object, and you use the dot (.) operator to access members of that object.

A constructor is a special method that is called when an object is created. Constructors can be overloaded.

Modifiers specify how the class, method, and data are accessed. You learned about public, private, and static modifiers. A public class, method, or data is accessible to all clients. A private method or data is only visible inside the class. You should make instance data private. You can provide a get method or a set method to enable clients to see or modify the data. A static variable or a static method is defined using the keyword static.

All the parameters are passed to methods using pass by value. For a parameter of a primitive type, the actual value is passed; for a parameter of a reference type, the reference for the object is passed.

An instance variable is a variable that belongs to an instance of a class. Its use is associated with individual instances. A static variable is a variable shared by all instances of the same class.

An instance method is a method that belongs to an instance of a class. Its use is associated with individual instances. A static method is a method that can be invoked without using instances.

The scope of instance and static variables is the entire class, regardless of where the variables are declared. The instance and static variables can be declared anywhere in the class.

You can use the this keyword to reference the members of the class, including the constructors.

A Java array is an object that can contain primitive type values or object type values. When an array is created, its elements are assigned the default value of 0 for the numeric primitive data types, '\u0000' for char types, false for boolean types, and null for object types.

Review Questions

- 6.1 Describe the relationship between an object and its defining class. How do you declare a class? How do you declare an object? How do you create an object? How do you declare and create an object in one statement?
- 6.2 What are the differences between constructors and methods?
- 6.3 Describe the difference between passing a parameter of a primitive type and passing a parameter of a reference type. Show the output of the following program:

```
public class Test
{
    public static void main(String[] args)
    {
        Count myCount = new Count();
        int times = 0;

        for (int i=0; i<100; i++)
            increment(myCount, times);

        System.out.println("count is " + myCount.count);
        System.out.println("times is " + times);
    }

    public static void increment(Count c, int times)
    {
        c.count++;
        times++;
    }
}

class Count
{
    public int count;

    Count(int c)
    {
        count = c;
    }
}
```

```

    {
        count = c;
    }

    Count()
    {
        count = 1;
    }
}

```

6.4 Show the output of the following program:

```

public class Test
{
    public static void main(String[] args)
    {
        Circle circle1 = new Circle(1);
        Circle circle2 = new Circle(2);

        // Attempt to swap circle1 with circle2
        System.out.println("Before swap: circle1 = " +
            circle1.radius + " circle2 = " + circle2.radius);
        swap(circle1, circle2);
        System.out.println("After swap: circle1 = " +
            circle1.radius + " circle2 = " + circle2.radius);
    }

    public static void swap(Circle x, Circle y)
    {
        System.out.println("Before swap: x = " +
            x.radius + " y = " + y.radius);

        Circle temp = x;
        x = y;
        y = temp;

        System.out.println("After swap: x = " +
            x.radius + " y = " + y.radius);
    }
}

```

6.5 Suppose that the class Foo is defined as follows:

```

public class Foo
{
    int i;
    static String s;

    void imethod()
    {
    }

    static void smethod()
    {
    }
}

```

Let f be an instance of Foo. Are the following statements correct?

```

System.out.println(f.i);

```

```

System.out.println(f.s);

f.imethod();

f.smethod();

System.out.println(Foo.i);

System.out.println(Foo.s);

Foo.imethod();

Foo.smethod();

```

6.6 What is the output of the following program?

```

public class Foo
{
    static int i = 0;
    static int j = 0;

    public static void main(String[] args)
    {
        int i = 2;
        int k = 3;

        {
            int j = 3;
            System.out.println("i + j is " + i+j);
        }

        k = i + j;
        System.out.println("k is "+k);
        System.out.println("j is "+j);
    }
}

```

6.7 What is wrong with the following program?

```

public class ShowErrors
{
    public static void main(String[] args)
    {
        int i;
        int j;

        j = MyInput.readInt();
        if (j > 3)
            System.out.println(i+4);
    }
}

```

6.8 What is wrong with the following program?

```

public class ShowErrors
{
    public static void main(String[] args)
    {
        for (int i=0; i<10; i++);
            System.out.println(i+4);
    }
}

```

6.9 List the modifiers that you learned in this chapter and describe their purposes.

6.10 Describe the role of the this keyword.

6.11 Analyze the following code:

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

class A
{
    String s;

    A(String s)
    {
        this.s = s;
    }

    public void print()
    {
        System.out.print(s);
    }
}
```

- a. The program does not compile because Test does not have a constructor Test().
- b. The program does not compile because new A() is used in class Test, but class A does not have a default constructor.
- c. The program does not compile because class A does not have a default constructor, and a default constructor must be defined explicitly for every class.
- d. The program compiles, but it has a runtime error due to the conflict on the method name print.
- e. None of the above.

6.12 What is wrong in the following code?

```
class Test
{
    public static void main(String[] args)
    {
        C c = new C(5.0);
        System.out.println(c.value);
    }
}

class C
{
    int value = 2;
}
```

- a. The program has a compilation error because class C does not have a default constructor.
- b. The program has a compilation error because class C does not have a constructor with a double argument.
- c. The program compiles fine, but it does not run because class C is not public.
- d. a and b.

6.13 What is wrong in the following code?

```
public class Foo
{
    public void method1()
    {
        Circle c;
        System.out.println("What is radius " + c.getRadius());
        c = new Circle();
    }
}
```

- a. The program has a compilation error because class Foo does not have a main method.
- b. The program has a compilation error because class Foo does not have a default constructor.
- c. The program has a compilation error in the println statement where c has not been defined.
- d. The program compiles fine, but it has a runtime error because variable c is null when the println statement is executed.

6.14 What is wrong in the following code?

```
public class Foo
{
    public static void main(String[] args)
    {
        method1();
    }

    public void method1()
    {
        method2();
    }

    public void method2()
    {
        System.out.println("What is radius " + c.getRadius());
    }

    Circle c = new Circle();
}
```

- a. method2 should be declared before method1, since method2 is invoked from method1.
- b. c should be declared before method2, since c is used in method2.
- c. The program has a compilation error in the println statement where c has not been defined.
- d. The program compiles fine, but it has a runtime error because variable c is null when the println statement is executed.
- e. The program compiles and runs fine.

6.15 Analyze the following code and choose the best answer:

```
public class Foo
{
    private int x;

    public static void main(String[] args)
    {
        Foo foo = new Foo();
        System.out.println(foo.x);
    }
}
```

- a. Since x is private, it cannot be accessed from an object foo.
- b. Since x is defined in the class Foo, it can be accessed by any methods inside the class without using an object. So you can write the code to access x without creating the object such as foo in this code.
- c. Since x is an instance variable, it cannot be directly used inside a main method. However, it can be accessed through an object such as foo in this code.
- d. You cannot create a self-referenced object, i.e. foo is created inside the class Foo.

6.16 In the following class, radius is private in the Circle class and myCircle is an object of the Circle class. Can the following code compile and run? Explain why.

```
public class Circle
{
    private double radius = 1.0;

    /**Find the area of this circle*/
    double findArea()
    {
        return radius*radius*3.14159;
    }

    public static void main(String[] args)
    {
```

```

        Circle myCircle = new Circle();
        System.out.println("Radius is " + myCircle.radius);
    }
}

```

6.17 Is an array an object or a primitive type value? Can an array contain elements of a primitive type as well as an object type? Describe the default value for the elements of an array.

6.18 Show the printout of the following code:

a.

```

public class Test
{
    public static void main(String[] args)
    {
        int[] a = {1, 2};
        swap(a[0], a[1]);
        System.out.println("a[0] = " + a[0] + " a[1] = " + a[1]);
    }

    public static void swap(int n1, int n2)
    {
        int temp = n1;
        n1 = n2;
        n2 = temp;
    }
}

```

b.

```

public class Test
{
    public static void main(String[] args)
    {
        int[] a = {1, 2};
        swap(a);
        System.out.println("a[0] = " + a[0] + " a[1] = " + a[1]);
    }

    public static void swap(int[] a)
    {
        int temp = a[0];
        a[0] = a[1];
        a[1] = temp;
    }
}

```

c.

```

public class Test
{
    public static void main(String[] args)
    {
        T t = new T();
        swap(t);
    }
}

```



```

        System.out.println("e1 = " + t.e1 + " e2 = " + t.e2);
    }

    public static void swap(T t)
    {
        int temp = t.e1;
        t.e1 = t.e2;
        t.e2 = temp;
    }
}

class T
{
    int e1 = 1;
    int e2 = 2;
}

```

d.

```

public class Test
{
    public static void main(String[] args)
    {
        T t1 = new T();
        T t2 = new T();
        System.out.println("t1's i="+t1.i+" and j="+t1.j);
        System.out.println("t2's i="+t2.i+" and j="+t2.j);
    }
}

class T
{
    static int i = 0;
    int j = 0;

    T()
    {
        i++;
        j = 1;
    }
}

```

Programming Exercises

- 6.1 Write a class named Rectangle to represent rectangles. The data fields are width, height, and color. Use double for width and height, and String for color. Suppose that all rectangles have the same color. Use a static variable for color. You need to provide the accessor methods for the properties and a method findArea() for computing the area of the rectangle.

The outline of the class is given as follows:

```

public class Rectangle

```

```

{
    private double width = 1;
    private double height = 1;
    private static String color = "white";

    public Rectangle()
    {
    }

    public Rectangle(double width, double height, String color)
    {
    }

    public double getWidth()
    {
    }

    public void setWidth(double width)
    {
    }

    public double getHeight()
    {
    }

    public void setHeight(double height)
    {
    }

    public static String getColor()
    {
    }

    public static void setColor(String color)
    {
    }

    public double findArea()
    {
    }
}

```

Write a client program to test the class Rectangle. In the client program, create two Rectangle objects. Assign any widths and heights to the two objects. Assign the first object the color red, and the second, yellow. Display both objects' properties and find their areas.

- 6.2 Write a class named Fan to model the fans. The properties are speed, on, radius, and color. You need to provide the accessor methods for the properties, and the toString method for returning a string consisting of all string values of all the properties in this class. Suppose the fan has three fixed speeds. Use constants 1, 2, and 3 to denote slow, medium, and fast speed.

The outline of the class is given as follows:

```

public class Fan
{
    public static int SLOW = 1;
    public static int MEDIUM = 2;
    public static int FAST = 3;

    private int speed = SLOW;

```

```

private boolean on = false;
private double radius = 5;
private String color = "white";

public Fan()
{
}

public int getSpeed()
{
}

public void setSpeed(int speed)
{
}

public boolean isOn()
{
}

public void setOn(boolean trueOrFalse)
{
}

public double getRadius()
{
}

public void setRadius(double radius)
{
}

public String getColor()
{
}

public void setColor(String color)
{
}

public String toString()
{
}
}

```

Write a client program to test the Fan class. In the client program, create a Fan object. Assign maximum speed, radius 10, color yellow, and turn it on. Display the object by invoking its toString method.

- 6.3 Write a class named Account to model accounts. The properties and methods of the class are shown in Figure 6.21. Interest is compounded monthly.

***Insert Figure 6.21

Account
-id: int -balance: double -annualInterestRate: double
+Account() +Account(id: int, balance: double, annualInterestRate: double) +getId(): int +getBalance(): double +getAnnualInterestRate():double +setId(id: int): void +setBalance(balance: double): void +setAnnualInterestRate(annualInterestRate: double): void +getMonthlyInterest(): double +withdraw(amount: double): void +deposit(amount: double): void

Figure 6.21

The Account class contains properties id, balance, annual interest rate, accessor methods, and the methods for computing interest, withdrawing money, and depositing money.

Write a client program to test the Account class. In the client program, create an Account object with account id of 1122, balance of 20000, and an annual interest rate of 4.5%. Use the withdraw method to withdraw \$2500, use the deposit method to deposit 3000, and print the balance and the monthly interest.

- 6.4 Write a class named Stock to model a stock. The properties and methods of the class are shown in Figure 6.22. The method changePercent computes the percent of the change of the current price vs. the previous closing price.

***Insert Figure 6.22

Stock
-symbol: String -name: String -previousClosingPrice: double -currentPrice: double
+Stock() +Stock(symbol: String, name: String) +getSymbol():String +getName():String +getPreviousClosingPrice(): double +getCurrentPrice(): double +setSymbol(symbol: String): void +setName(name: String): void +setPreviousClosingPrice(price: double): void +setCurrentPrice(price: double): void +changePercent(newPercent: double): double

Figure 6.22

The Stock class contains properties symbol, name, previous closing price, and current price, accessor methods, and the methods for computing price changes.

Write a client program to test the Stock class. In the client program, create a Stock object with stock symbol SUNW, name Sun Microsystems Inc, previous closing price of 100. Set a new current price randomly and display the price change percentage.